Université Claude Bernard Lyon 1

LIFAPSD – Algorithmique, Programmation et Structures de données

Nicolas Pronost

LIRIS

Chapitre 2
Allocation dynamique de mémoire

Notion de pointeur

- Un pointeur est une variable destinée à contenir une adresse mémoire
- Sur une machine 32 bits, les adresses mémoire sont codées sur 4 octets donc $2^{32} \approx 4$ milliards (10^9) d'adresses possibles
- Sur une machine 64 bits, les adresses mémoire sont codées sur 8 octets donc $2^{64} \approx 18 \times 10^{18}$ d'adresses possibles
- Déclaration d'une variable pointeur en algorithmique

```
p : pointeur sur type
```

- Déclaration d'une variable pointeur en C++

```
type * p;
```

  - Deux lectures possibles équivalentes
    - *p (valeur pointée) est de type *type*
    - p est de type *type** (pointeur sur *type*)

Pointeur et référence

- Variable normale en C++

```
int a;
```

- Référence à une variable

```
int a = 2;
int & b = a; // référence
b = 3;       // a vaut maintenant aussi 3
a = 4;       // b vaut maintenant aussi 4
```

- Pointeur sur une variable

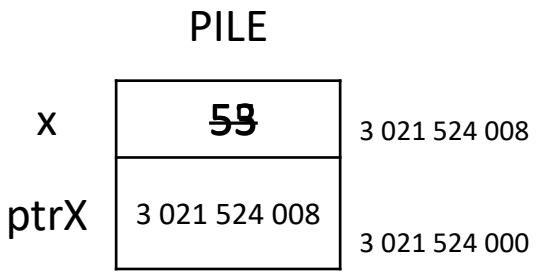
```
int a = 2;
int * b = & a; // pointeur
// b contient l'adresse de a (ex. 3 984 562 410)
```

Opérateurs sur pointeur

- L'opérateur & sur une variable permet de récupérer son adresse mémoire
- L'opérateur * permet d'accéder à la valeur qui se trouve à l'adresse pointée
  - précondition: adresse valide
  - le type de pointeur permet de savoir combien d'octets lire et comment les interpréter
- Exemple

```
Variables
  x : entier
  ptrX : pointeur sur entier
Début
  x ← 53
  ptrX ← &x
  *ptrX ← *ptrX + 2
  afficher(x)
Fin
```

```
int x = 53;
int * ptrX = &x;
*ptrX = *ptrX + 2;
cout << x;
```



Fonction modificatrice

- Comme une référence connecte à une variable et un pointeur décrit l'adresse d'une variable, on peut utiliser des pointeurs pour passer un paramètre en mode donnée-résultat (dit « à la C »)

```
void PrecEtSuiv (int x, int & prec, int & suiv) {
   prec = x-1; suiv = x+1;
}
int main () {
   int x = 100; int y = 15; int z = 8;
   PrecEtSuiv(x,y,z);
   cout << "Précédent=" << y << ", Suivant= " << z;
   return 0;
}
```

```
void PrecEtSuiv (int x, int * prec, int * suiv) {
   *prec = x-1; *suiv = x+1;
}
int main () {
   int x = 100; int y = 15; int z = 8;
   PrecEtSuiv(x,&y,&z);
   cout << "Précédent=" << y << ", Suivant= " << z;
   return 0;
}
```

Pointeur et référence

- Autres différences entre pointeur et référence
  - Une référence est affectée au moment de sa déclaration
  - Les pointeurs peuvent pointer sur rien (pointeur nul)

```
float * ptrReel = nullptr;
```

-  Attention à la confusion entre les opérateurs *
  - déclaration d'un pointeur, déréférencement et multiplication

```
type * nom
```

```
*nom
```

```
nom1 * nom2
```

Exemple de pointeurs

```
int valeur1 = 5;
int valeur2 = 15;
int * p1, * p2;

p1 = &valeur1;    // p1 = adresse de valeur1
p2 = &valeur2;    // p2 = adresse de valeur2
*p1 = 10;         // valeur pointée par p1 = 10
*p2 = *p1;        // valeur pointée par p2 = valeur pointée par p1 = 10
p1 = p2;          // adresse de p2 copiée dans p1
*p1 = 20;         // valeur pointée par p1 = 20

cout << "Valeur 1: " << valeur1 << endl;
cout << "Valeur 2: " << valeur2 << endl;
```

Pointeurs spéciaux

- Plusieurs pointeurs peuvent pointer sur la même variable

```
int x = 1;
int * ptrX1 = &x;
int * ptrX2 = &x;
*ptrX1 = 2;
cout << x << " " << *ptrX1 << " " << *ptrX2;
```

- Pointeur de pointeur

```
int x = 1;
int * ptrX = &x;
int ** ptrptrX = &ptrX;
cout << **ptrptrX;
```

Pointeurs spéciaux

- Cas particulier des pointeurs non typés (**void** *)

```
void * p;
```

  - p est une variable de type « adresse générique »
  - il peut pointer sur n'importe quel type de donnée
  - le déréférencement est interdit (on ne sait pas comment lire la variable pointée puisqu'on a pas son type)
  - opérations arithmétiques interdites (pour les mêmes raisons on ne connait pas la taille des variables pointées)
  - possibilité de conversion en un pointeur typé

```
float * pf = (float*) p;
```

  - utile par exemple pour traiter une suite d'octets à partir d'une adresse ou pour stocker des adresses sur des variables de différents types

Pointeurs spéciaux

- Pointeur sur type complexe (structure et classe)
- Pour accéder à un champ d'une structure à partir d'un pointeur sur la structure
  - l'opérateur . (point) sur le pointeur déréférencé (avec *)
  - l'opérateur équivalent -> (flèche)

```
Date maDate;
Date * ptrMaDate = &maDate;

(*ptrMaDate).jour = 7;     // ou: ptrMaDate->jour = 7;
(*ptrMaDate).mois = 5;     // ou: ptrMaDate->mois = 5;
(*ptrMaDate).annee = 2000; // ou: ptrMaDate->annee = 2000;
```

Pointeur suspendu

- Le problème du pointeur suspendu (*dangling*) apparaît lorsque les données pointées sont supprimées mais le pointeur n'est pas mis à jour

```
char * ptr;
if (true){
   char x = 'x';
   ptr = &x;
}
// ptr est un pointeur suspendu
```

- Le pointeur nul doit être utilisé pour indiquer qu'un pointeur ne pointe (plus) sur rien

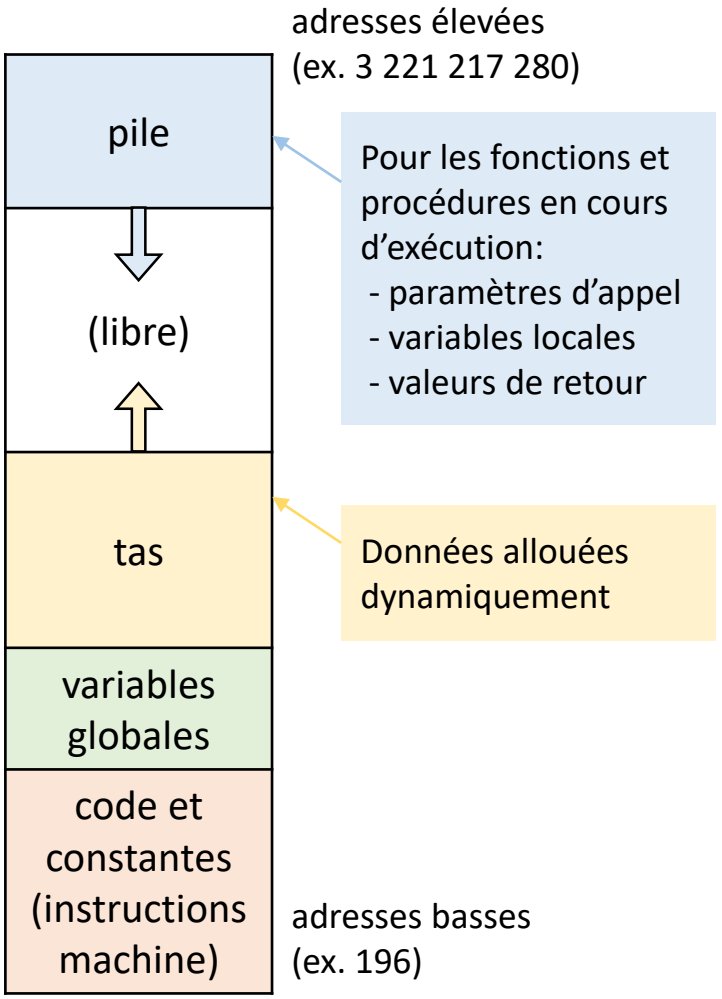
```
int * p = nullptr;
```

  - Vérification avant utilisation de la validité du pointeur

```
if (p != nullptr){
  // vous pouvez utiliser p
}
```

Allocation dynamique de mémoire

- Réservation d'emplacements mémoire sur le tas
- Les données mises sur le tas ne sont pas détruites en sortie de fonction ou de procédure



Allocation dynamique de mémoire

- Norme algorithmique
  - Réservation

```
p ← réserve type
p ← réserve tableau [1...n] de type
```

réserve fait deux choses:

- alloue un emplacement mémoire de la taille nécessaire sur le tas
- renvoie l'adresse mémoire de l'élément sur le tas (si tableau, renvoie l'adresse du premier élément du tableau)

  - Libération

```
libère p
```

libère rend l'emplacement mémoire disponible pour d'autres utilisations éventuelles

Allouer et oublier de libérer = fuite mémoire

Fuite mémoire

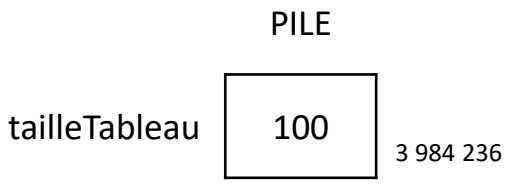
- Soit l'algorithme suivant

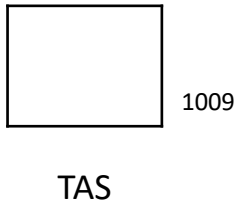
```
Procédure remplirTableau(taille : entier)
Variables locales:
  i : entier
  val : pointeur sur entier
  tab : pointeur sur entier
Début
  val ← réserve entier
  tab ← réserve tableau [1...taille] d'entiers
  Pour i allant de 1 à taille par pas de 1 faire
     saisir(val)
     tab[i] ← val↑
  Fin pour
  libère val
Fin

Variables
  tailleTableau : entier
Début
  saisir(tailleTableau)
  remplirTableau(tailleTableau)
Fin
```

Combien d'octets sont réservés et libérés sur le tas en supposant que tailleTableau vaut 100?

Trace mémoire

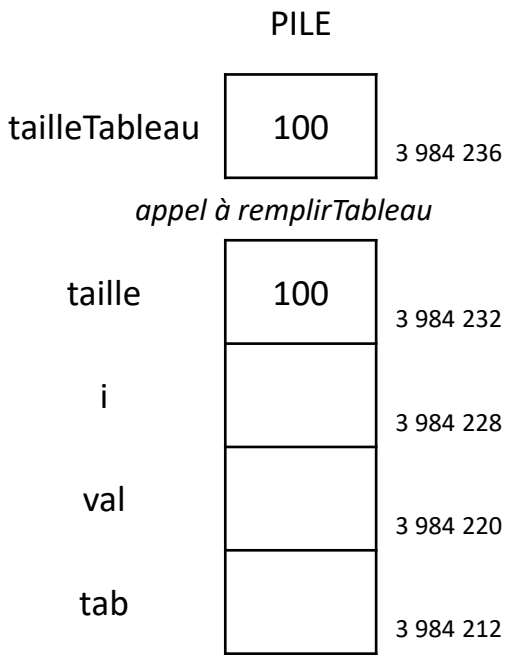
```
Procédure remplirTableau(taille :
entier)
Variables locales:
  i : entier
  val : pointeur sur entier
  tab : pointeur sur entier
Début
  val ← réserve entier
  tab ← réserve tableau
[1...taille] d'entiers
  Pour i allant de 1 à taille par
pas de 1 faire
     saisir(val)
     tab[i] ← val↑
  Fin pour
  libère val
Fin

Variables
  tailleTableau : entier
Début
  saisir(tailleTableau)
  remplirTableau(tailleTableau)
Fin
```

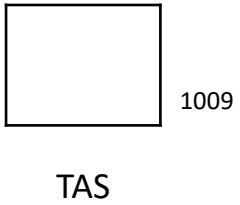




Trace mémoire

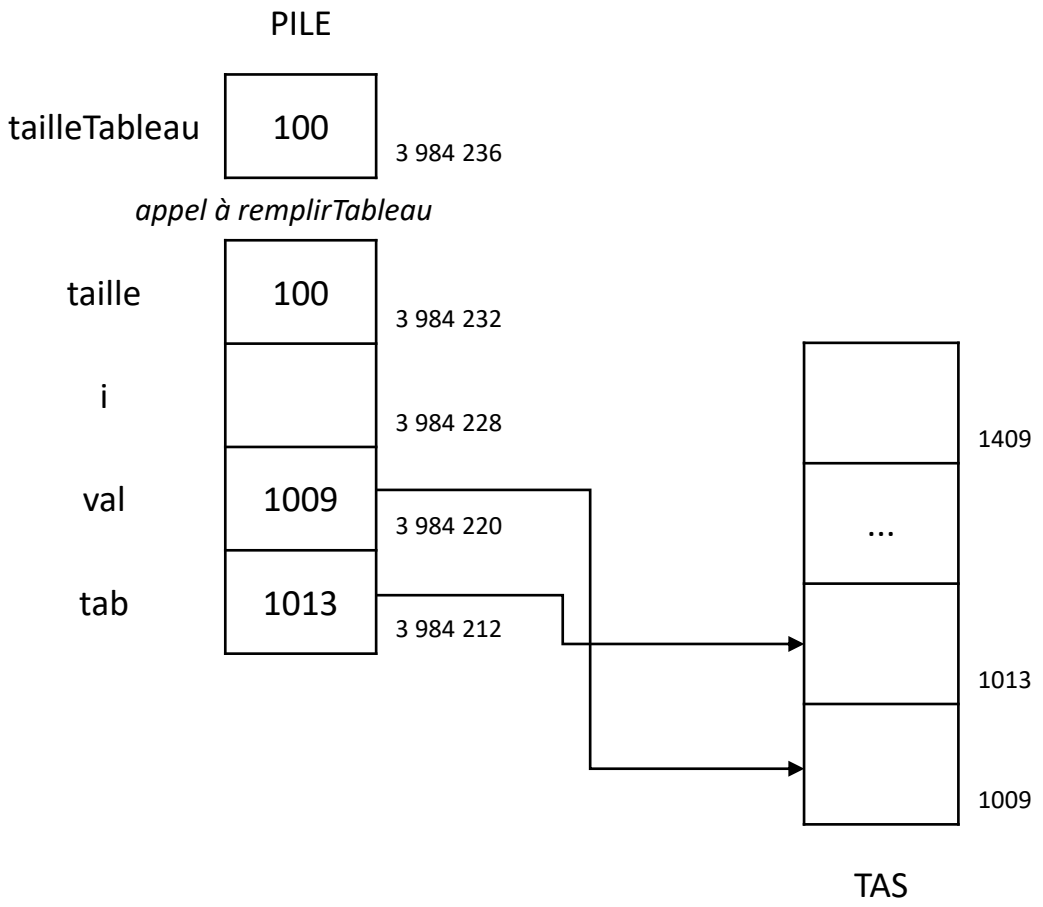
```
Procédure remplirTableau(taille :
entier)
Variables locales:
  i : entier
  val : pointeur sur entier
  tab : pointeur sur entier
Début
  val ← réserve entier
  tab ← réserve tableau
[1...taille] d'entiers
  Pour i allant de 1 à taille par
pas de 1 faire
     saisir(val)
     tab[i] ← val↑
  Fin pour
  libère val
Fin

Variables
  tailleTableau : entier
Début
  saisir(tailleTableau)
  remplirTableau(tailleTableau)
Fin
```





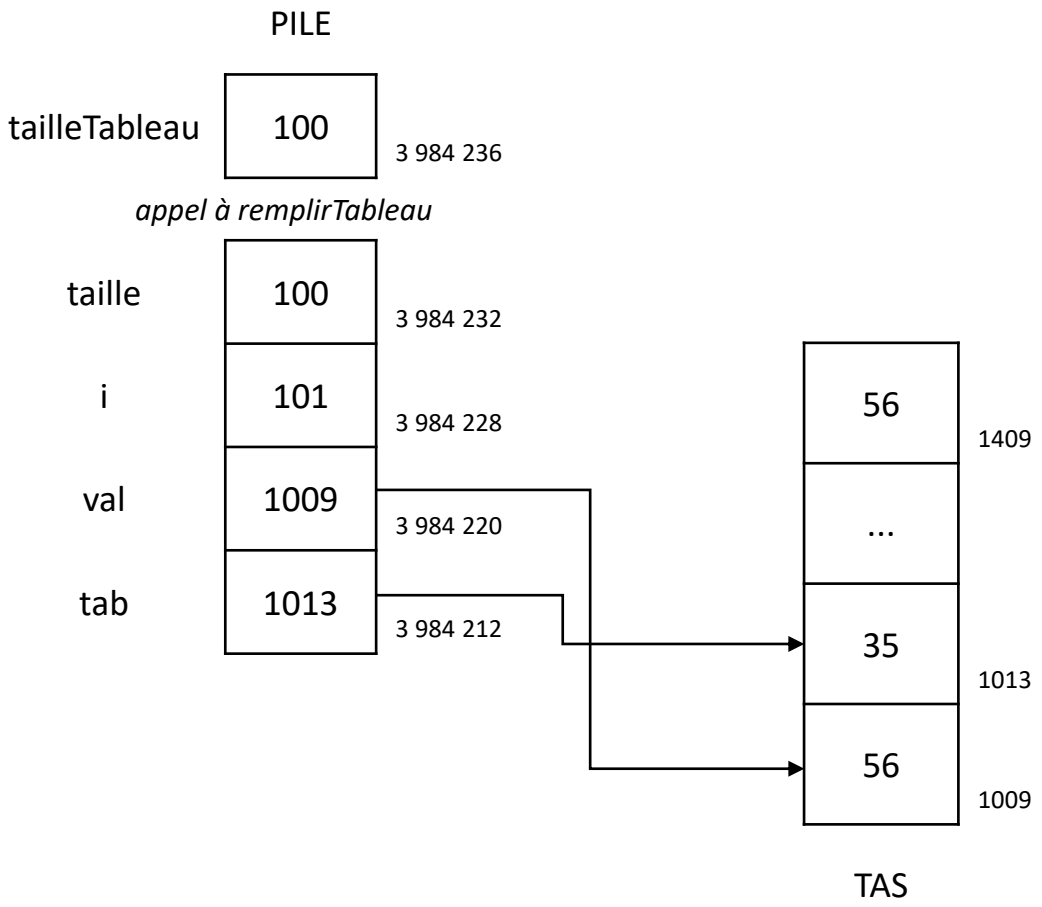
Trace mémoire

```
Procédure remplirTableau(taille :
entier)
Variables locales:
  i : entier
  val : pointeur sur entier
  tab : pointeur sur entier
Début
  val ← réserve entier
  tab ← réserve tableau
[1...taille] d'entiers
  Pour i allant de 1 à taille par
pas de 1 faire
     saisir(val)
     tab[i] ← val↑
  Fin pour
  libère val
Fin

Variables
  tailleTableau : entier
Début
  saisir(tailleTableau)
  remplirTableau(tailleTableau)
Fin
```



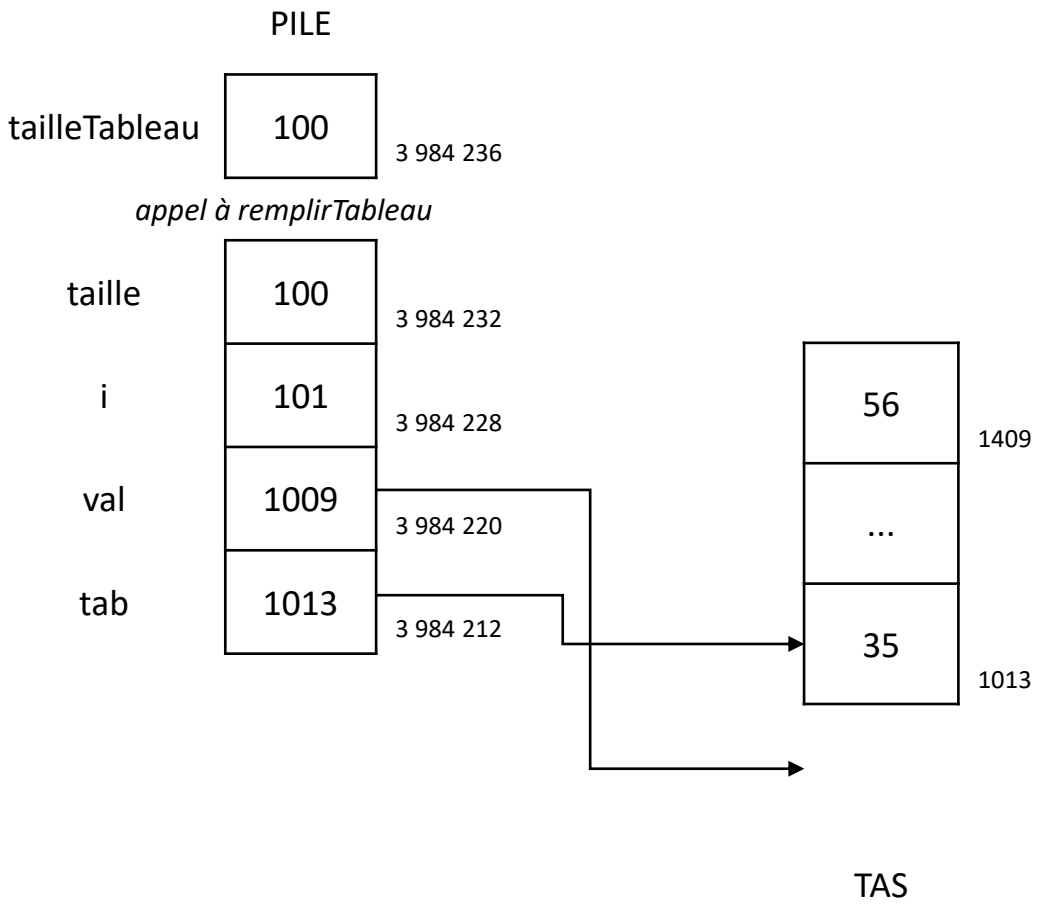
Trace mémoire

```
Procédure remplirTableau(taille :
entier)
Variables locales:
  i : entier
  val : pointeur sur entier
  tab : pointeur sur entier
Début
  val ← réserve entier
  tab ← réserve tableau
[1...taille] d'entiers
  Pour i allant de 1 à taille par
pas de 1 faire
     saisir(val)
     tab[i] ← val↑
  Fin pour
  libère val
Fin

Variables
  tailleTableau : entier
Début
  saisir(tailleTableau)
  remplirTableau(tailleTableau)
Fin
```



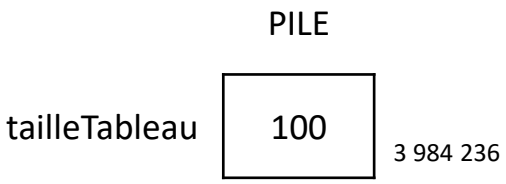
Trace mémoire

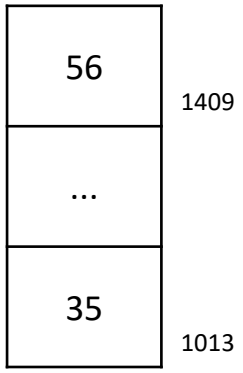
```
Procédure remplirTableau(taille :
entier)
Variables locales:
  i : entier
  val : pointeur sur entier
  tab : pointeur sur entier
Début
  val ← réserve entier
  tab ← réserve tableau
[1...taille] d'entiers
  Pour i allant de 1 à taille par
pas de 1 faire
     saisir(val)
     tab[i] ← val↑
  Fin pour
  libère val
Fin

Variables
  tailleTableau : entier
Début
  saisir(tailleTableau)
  remplirTableau(tailleTableau)
Fin
```



Trace mémoire

```
Procédure remplirTableau(taille :
entier)
Variables locales:
  i : entier
  val : pointeur sur entier
  tab : pointeur sur entier
Début
  val ← réserve entier
  tab ← réserve tableau
[1...taille] d'entiers
  Pour i allant de 1 à taille par
pas de 1 faire
     saisir(val)
     tab[i] ← val↑
  Fin pour
  libère val
Fin

Variables
  tailleTableau : entier
Début
  saisir(tailleTableau)
  remplirTableau(tailleTableau)
Fin
```



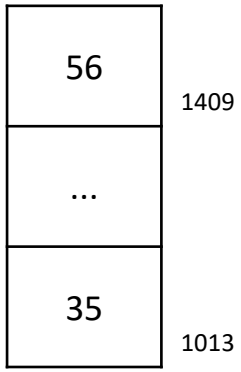


TAS

Trace mémoire

```
Procédure remplirTableau(taille :
entier)
Variables locales:
  i : entier
  val : pointeur sur entier
  tab : pointeur sur entier
Début
  val ← réserve entier
  tab ← réserve tableau
[1...taille] d'entiers
  Pour i allant de 1 à taille par
pas de 1 faire
     saisir(val)
     tab[i] ← val↑
  Fin pour
  libère val
Fin

Variables
  tailleTableau : entier
Début
  saisir(tailleTableau)
  remplirTableau(tailleTableau)
Fin
```

PILE



TAS

Fuite mémoire

- Soit l'algorithme suivant

```
Procédure remplirTableau(taille : entier)
Variables locales:
  i : entier
  val : pointeur sur entier
  tab : pointeur sur entier
Début
  val ← réserve entier
  tab ← réserve tableau [1...taille] d'entiers
  Pour i allant de 1 à taille par pas de 1 faire
     saisir(val)
     tab[i] ← val↑
  Fin pour
  libère val
Fin

Variables
  tailleTableau : entier
Début
  saisir(tailleTableau)
  remplirTableau(tailleTableau)
Fin
```

Combien d'octets sont réservés et libérés sur le tas en supposant que tailleTableau vaut 100?

- $100 \times 4 + 4 = 404$ octets sont réservés
- 4 octets sont libérés
- ➢ 400 octets sont perdus (et impossible à libérer car l'adresse mémoire du premier élément du tableau était locale à la procédure)

Ecrivez vos programmes pour assurer que toute réservation peut être et est libérée

Il existe des outils de diagnostic qui trace les fuites mémoires dans votre code (cf. LIFAPCD)

Allocation dynamique de mémoire

- Mise en œuvre en C++
  - Déclaration

```
type * pointeur;
```

  - Les opérateurs **new** et **delete**

```
pointeur = new type;
delete pointeur;
```

```
pointeur = new type [taille];
delete [] pointeur;
```

- Pour les raisons précédemment expliquées, il faut prendre la bonne habitude de mettre un pointeur à **nullptr** après libération de sa mémoire

```
delete pointeur;
pointeur = nullptr;
```

Allocation dynamique de mémoire

```
unsigned int nb_car,n;
char * nom;
cout << "Combien de caractères à votre nom?";
cin >> nb_car;
if (nb_car == 0) cout << "Erreur: pas de caractère";
else {
   nom = new char [nb_car];
   for (n=0; n<nb_car; n++) {
      cout << "Entrer caractère: ";
      cin >> nom[n];
   }
   cout << "Votre nom est: ";
   for (n=0; n<nb_car; n++) cout << nom[n];
   delete [] nom;
}
```

Allocation dynamique de mémoire

- Résumé pour les tableaux
  - Syntaxe pour déclarer un tableau sur la pile

```
type nom [taille];
```

  - Syntaxe pour déclarer un tableau sur le tas

```
type * nom = new type [taille];
delete [] nom;
```

  - Exemple: un tableau d'entiers de taille 10

```
int tab1 [10]; // pile
int * tab2 = new int [10]; // tas
delete [] tab2;
```

    - dans les deux cas, le type est : **int** [10]

Allocation dynamique de mémoire

- Allocation et libération de la mémoire pour des tableaux multidimensionnels
  - Exemple d'un tableau de 10 par 5

```
int ** tab2D = new int * [10];
for (int i=0; i<10; i++) tab2D[i] = new int [5];
```

```
for (int i=0; i<10; i++) delete [] tab2D[i];
delete [] tab2D;
```

Tableau en paramètre

- Rappel: la taille du tableau n'est pas transmise dans les entêtes suivantes

```
void procedureTab(int tab[4]); //tableau statique
void procedureTab(int tab[]);  //tableau statique ou dynamique
void procedureTab(int * tab);  //tableau dynamique
```

  - il faut donc passer la taille en paramètre avec le tableau

```
void procedureTab(int * tab, int taille);
```

- Rappel: pour un passage en mode donnée, utiliser le mot-clé **const**

```
void procedureTab(const int * tab, int taille);
```

Tableau en paramètre

```
void afficherTab(const double * tab, int taille)
{
//...
}

void doublerElemTab (double * tab, int taille) {
  for (int i = 0; i < taille ; i++)
    tab[i] = tab[i] * 2.0;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  doublerElemTab(nombres,3);
  afficherTab(nombres,3);
  return 0;
}
```

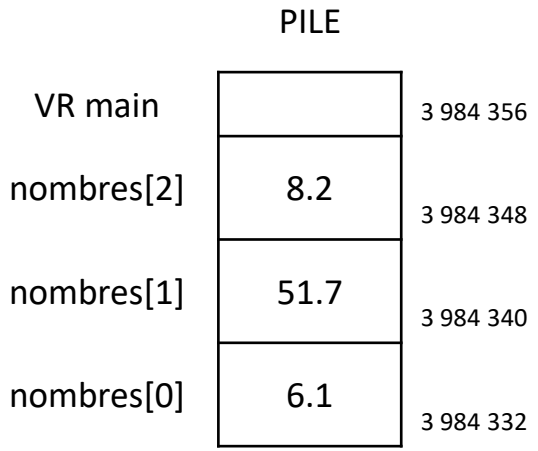


Tableau en paramètre

```
void afficherTab(const int * tab, int taille) {
//...
}

void doublerElemTab (double * tab, int taille) {
  for (int i = 0; i < taille ; i++)
    tab[i] = tab[i] * 2.0;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  doublerElemTab(nombres,3);
  afficherTab(nombres,3);
  return 0;
}
```

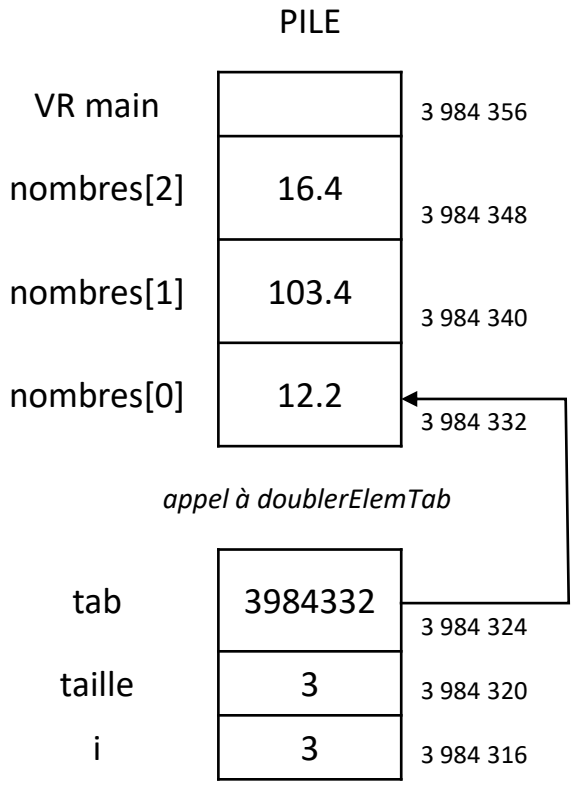


Tableau en paramètre

```
void afficherTab(const int * tab, int taille) {
//...
}

void doublerElemTab (double * tab, int taille) {
  for (int i = 0; i < taille ; i++)
    tab[i] = tab[i] * 2.0;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  doublerElemTab(nombres,3);
  afficherTab(nombres,3);
  return 0;
}
```

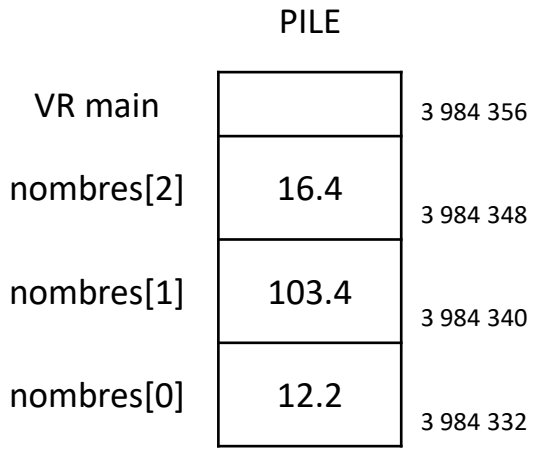


Tableau en paramètre

```
void afficherTab(const int * tab, int taille) {
//...
}

void doublerElemTab (double * tab, int taille) {
  for (int i = 0; i < taille ; i++)
    tab[i] = tab[i] * 2.0;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  doublerElemTab(nombres,3);
  afficherTab(nombres,3);
  return 0;
}
```

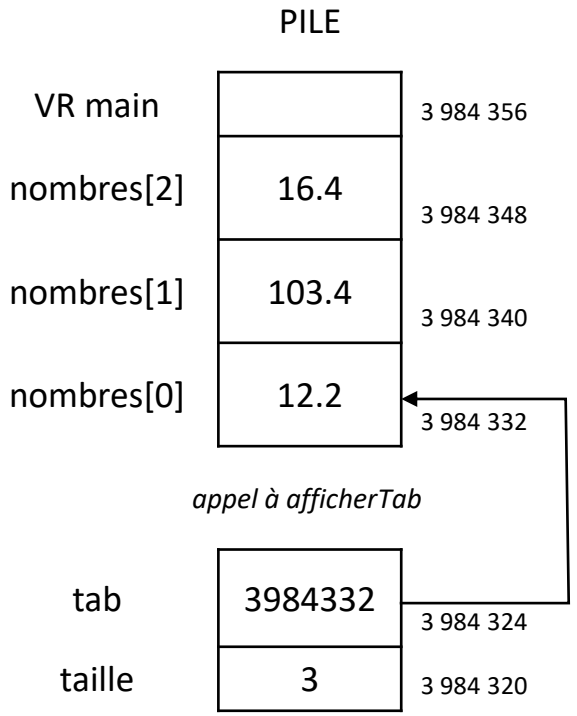


Tableau en paramètre

```
void afficherTab(const int * tab, int taille) {
//...
}

void doublerElemTab (double * tab, int taille) {
  for (int i = 0; i < taille ; i++)
    tab[i] = tab[i] * 2.0;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  doublerElemTab(nombres,3);
  afficherTab(nombres,3);
  return 0;
}
```

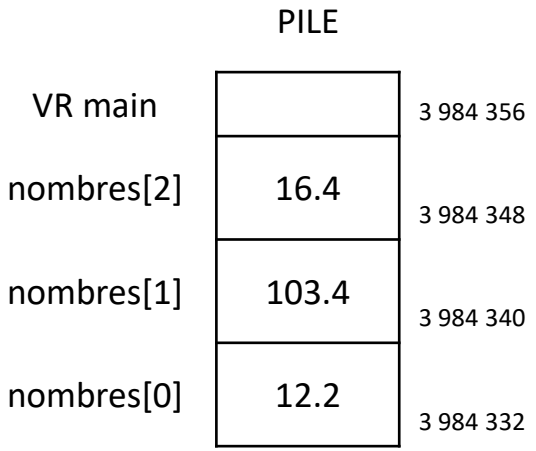


Tableau en paramètre

```
void afficherTab(const int * tab, int taille) {
//...
}

void doublerElemTab (double * tab, int taille) {
  for (int i = 0; i < taille ; i++)
    tab[i] = tab[i] * 2.0;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  doublerElemTab(nombres,3);
  afficherTab(nombres,3);
  return 0;
}
```

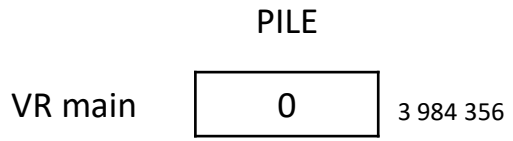


Tableau en paramètre

```
void afficherTab(const int * tab, int taille) {
//...
}

void doublerElemTab (double * tab, int taille) {
  for (int i = 0; i < taille ; i++)
    tab[i] = tab[i] * 2.0;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  doublerElemTab(nombres,3);
  afficherTab(nombres,3);
  return 0;
}
```

PILE

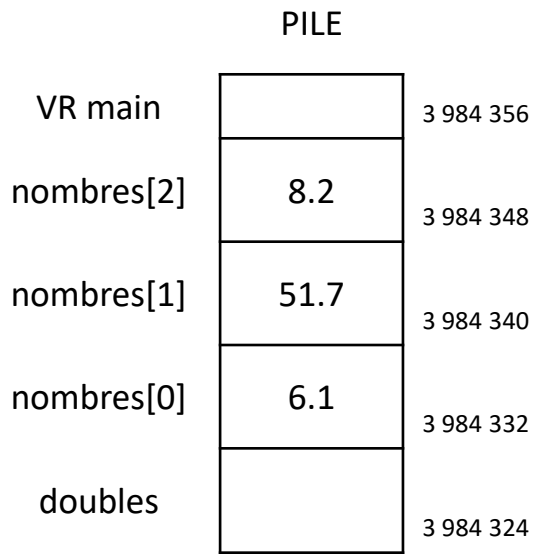
Retourner un tableau

- Que pensez-vous de ce programme?
  - Indication: le compilateur donne
    warning: address of local variable ‘resultat’ returned

```
void afficherTab(const double * tab, int taille) {
//...
}

double * doublerElemTab (const double * tab) {
  // pre-condition: tab est de taille 3
  double resultat[3];
  for (int i = 0; i < 3 ; i++)
    resultat[i] = tab[i] * 2.0;
  return resultat;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  double * doubles;
  doubles = doublerElemTab(nombres);
  afficherTab(doubles,3);
  return 0;
}
```

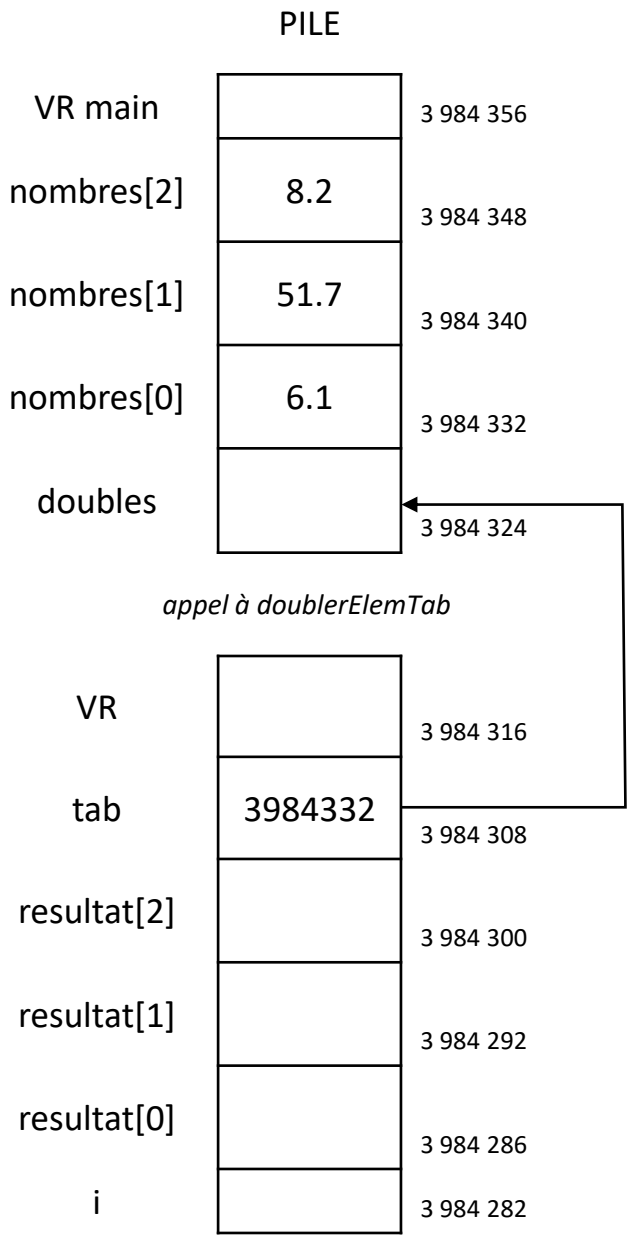
Retourner un tableau

```
void afficherTab(const double * tab, int taille) {
//...
}

double * doublerElemTab (const double * tab) {
  // pre-condition: tab est de taille 3
  double resultat[3];
  for (int i = 0; i < 3 ; i++)
    resultat[i] = tab[i] * 2.0;
  return resultat;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  double * doubles;
  doubles = doublerElemTab(nombres);
  afficherTab(doubles,3);
  return 0;
}
```



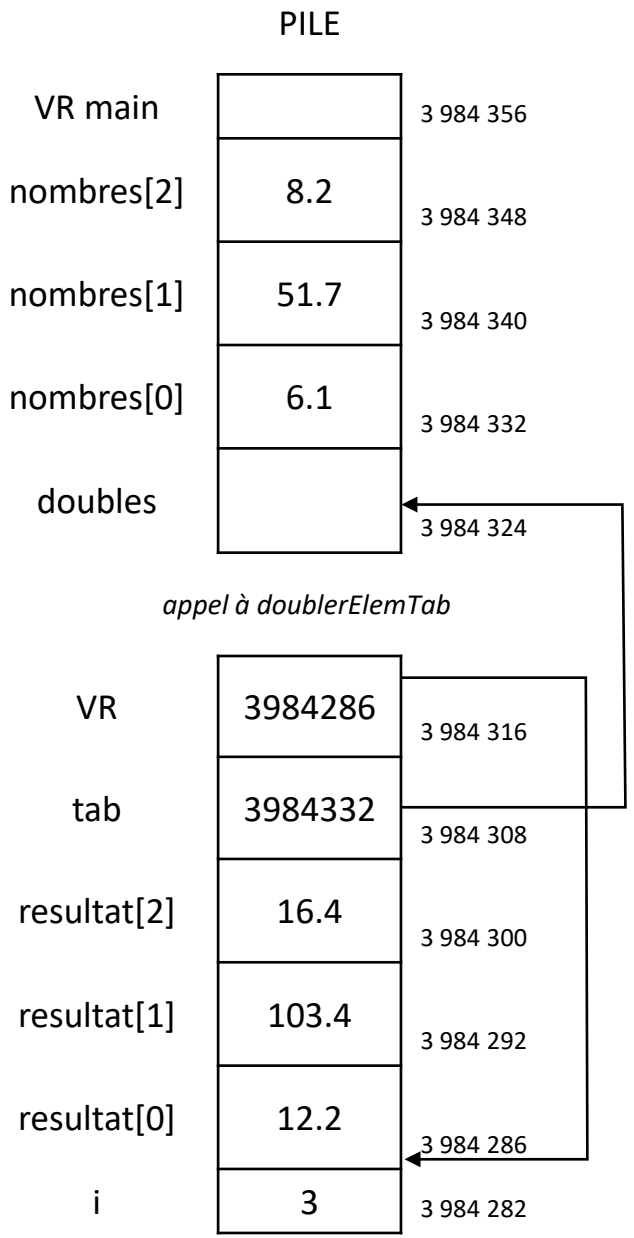
Retourner un tableau

```
void afficherTab(const double * tab, int taille) {
//...
}

double * doublerElemTab (const double * tab) {
  // pre-condition: tab est de taille 3
  double resultat[3];
  for (int i = 0; i < 3 ; i++)
    resultat[i] = tab[i] * 2.0;
  return resultat;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  double * doubles;
  doubles = doublerElemTab(nombres);
  afficherTab(doubles,3);
  return 0;
}
```



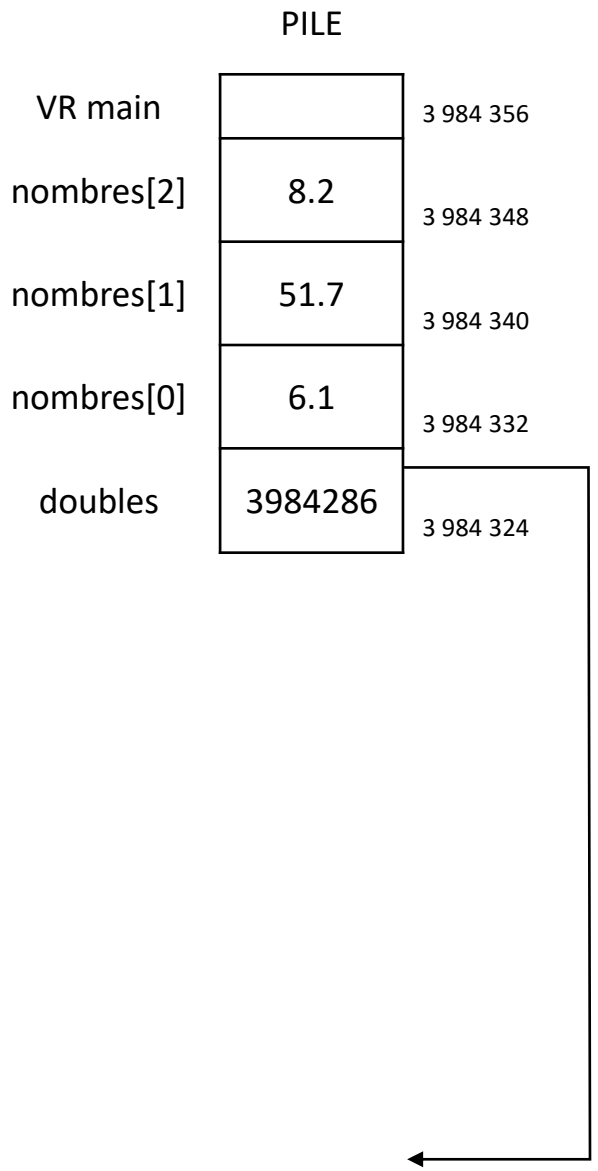
Retourner un tableau

```
void afficherTab(const double * tab, int taille) {
//...
}

double * doublerElemTab (const double * tab) {
  // pre-condition: tab est de taille 3
  double resultat[3];
  for (int i = 0; i < 3 ; i++)
    resultat[i] = tab[i] * 2.0;
  return resultat;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  double * doubles;
  doubles = doublerElemTab(nombres);
  afficherTab(doubles,3);
  return 0;
}
```



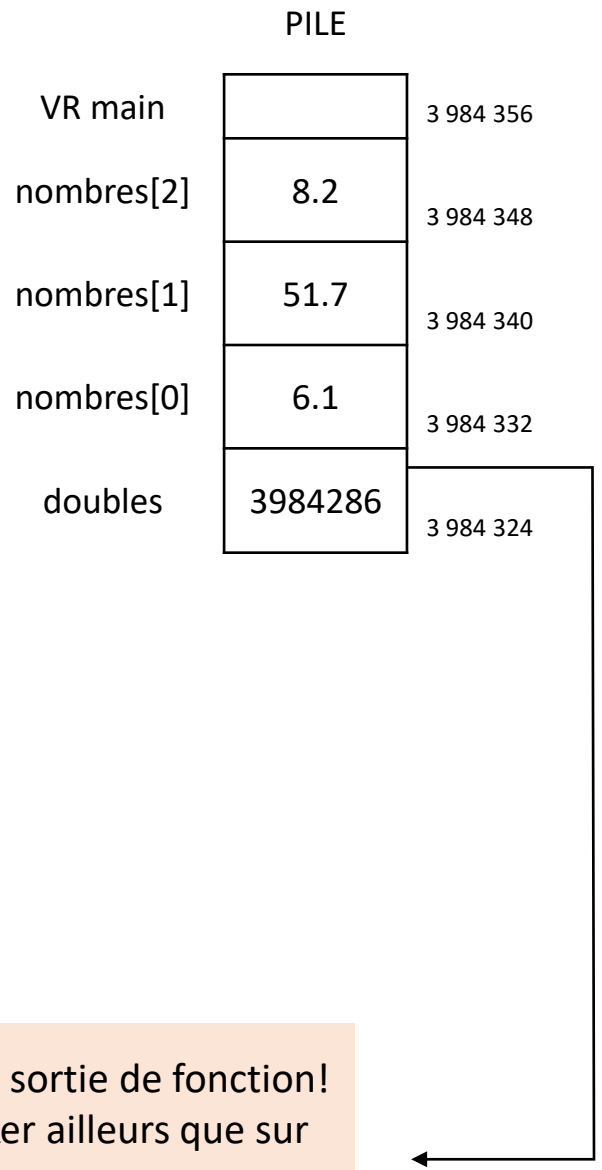
Retourner un tableau

```
void afficherTab(const double * tab, int taille) {
//...
}

double * doublerElemTab (const double * tab) {
  // pre-condition: tab est de taille 3
  double resultat[3];
  for (int i = 0; i < 3 ; i++)
    resultat[i] = tab[i] * 2.0;
  return resultat;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  double * doubles;
  doubles = doublerElemTab(nombres);
  afficherTab(doubles,3);
  return 0;
}
```



Retourner un tableau

```
void afficherTab(const double * tab, int taille) {
//...
}

double * doublerElemTab (const double * tab) {
  // pre-condition: tab est de taille 3
  double resultat[3];
  for (int i = 0; i < 3 ; i++)
    resultat[i] = tab[i] * 2.0;
  return resultat;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  double * doubles;
  doubles = doublerElemTab(nombres);
  afficherTab(doubles,3);
  return 0;
}
```



Aaargh! Retour de l'adresse d'un tableau détruit en sortie de fonction!
A retenir: pour retourner un tableau, il faut le stocker ailleurs que sur la pile (sur le tas)

Retourner un tableau

```
void afficherTab(const double * tab, int taille) {
//...
}

double * doublerElemTab (const double * tab, int taille) {
/*
Précondition: tab est l'adresse d'un tableau contenant taille éléments de type double
Résultat: l'adresse d'un tableau de taille éléments double, situé sur le tas. Le programme
appelant a la responsabilité de stocker et de libérer la mémoire sur ce tableau quand il n'en
aura plus besoin.
*/
  double * resultat = new double [taille];
  for (int i = 0; i < taille ; i++)
    resultat[i] = tab[i] * 2.0;
  return resultat;
}

int main () {
  double nombres[] = {6.1, 51.7, 8.2};
  double * doubles;
  doubles = doublerElemTab(nombres,3);
  afficherTab(doubles,3);
  delete [] doubles;
  doubles = nullptr;
  return 0;
}
```

Retourner un tableau

- Résumé
  - On ne peut pas stocker le tableau sur la pile si on veut le renvoyer
  - Il faut le stocker sur le tas avec une allocation dynamique de la mémoire (**new**)
  - Attention au risque de fuite mémoire si l'on oublie de libérer le tableau
    - à chaque **new** il faut un **delete**