Algorithmique et Structures de données

Thomas Lavergne & Florent Hivert

Mél : thomas.lavergne@lisn.upsaclay.fr

Mémoire simplifiée

Pour simplifier, on va supposer que toutes les informations codées en mémoire tiennent exactement dans un mot. On va représenter la mémoire par un tableau :

⋮	
4	?
3	'a'
2	?
1	?
0	4

? : valeur inconnue = imprévisible = non définie = aléatoire.

Réservation de la mémoire

Définition

réserver *une portion de mémoire s'appelle l'****allocation***

libérer *une portion de mémoire s'appelle la* ***désallocation***

L'allocation peut être

- **statique** ou **dynamique** ;
- **automatique** ou **manuelle**.

Réservation de la mémoire

Définition

réserver *une portion de mémoire s'appelle l'****allocation***

libérer *une portion de mémoire s'appelle la* ***désallocation***

L'allocation peut être

- **statique** ou **dynamique** ;
- **automatique** ou **manuelle**.

Différentes zones de mémoire

Définition

*La mémoire est découpée en zones souvent appelées **segments**.*
Dans chaque segment la mémoire est allouée différemment.

Voici quelques segments :

- **code** qui contient le code binaire du programme
- la zone des **données statiques** qui contient les données globales.
- la **pile** qui contient les variables automatiques et les informations nécessaires à l'exécution des appels de fonctions
- le **tas** qui contient les variables dynamiques manuelles.

Nous allons voir le mécanisme d'allocation automatique en détail...

Environnement

Dans un programme écrit dans un langage impératif, à chaque variable est associée une adresse (i.e. un numéro d'emplacement en mémoire).
Une fonction doit savoir où se trouvent dans la mémoire les variables qu'elle utilise (comment y accéder).

Définition

*On appelle **environnement** d'une fonction la liste des **variables accessibles** à cette fonction. Il contient :*

- *les variables globales (i.e. déclarées en dehors de toute fonction) ;*
- *les paramètres éventuels de la fonction et ses variables locales (i.e. déclarées dans la fonction).*

État d'un programme

Le traitement d'un programme s'effectue en deux phases : la **compilation** et **l'exécution.**

- La compilation : **traduction** du langage de programmation (C++) vers le langage de la machine. Durant cette phase on **planifie** l'usage des ressources, en particulier de la mémoire. Ainsi, l'environnement est déterminé pendant la compilation en fonction des déclarations.
- L'exécution : **mise en œuvre** du programme traduit. L'état d'un programme à un moment donné de l'exécution est constitué par son environnement et sa mémoire.
  Au début de l'exécution, la mémoire est «vide». La **mémoire** est ensuite modifiée pendant l'exécution par les affectations.

État d'un programme

Le traitement d'un programme s'effectue en deux phases : la **compilation** et **l'exécution.**

- La compilation : **traduction** du langage de programmation (C++) vers le langage de la machine. Durant cette phase on **planifie** l'usage des ressources, en particulier de la mémoire. Ainsi, l'environnement est déterminé pendant la compilation en fonction des déclarations.

- L'exécution : **mise en œuvre** du programme traduit. L'état d'un programme à un moment donné de l'exécution est constitué par son environnement et sa mémoire.
  Au début de l'exécution, la mémoire est «vide». La **mémoire** est ensuite modifiée pendant l'exécution par les affectations.

État d'un programme - Exemple de compilation

Considérons le programme suivant :

```
1  int main() {
2      int x;
3      x = 5;
4      return 0;
5  }
```

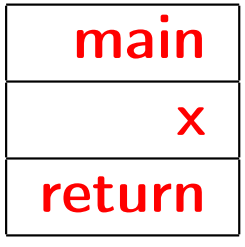
Lors de la **compilation**, on lit que ce programme ne contient pas de variables globales et qu'il est composé d'une unique fonction, le `main`, qui contient une déclaration (`int x`), une instruction d'affectation (`x = 5`) et une instruction de retour (`return 0`).

Tableau d'activation - Exemple

Retenir

*L'environnement local construit par le compilateur ne contiendra que la composante associée directement à cette fonction, sous la forme d'un **tableau d'activation** qui regroupe les **emplacements nécessaires** à cette fonction, pour son exécution future.*

Le tableau d'activation construit pour cette fonction prévoit 3 emplacements :



- un pour des informations administratives (adresse de retour) relatives à la fonction,
- un pour la variable `x` déclarée localement,
- un pour la valeur de retour de la fonction.

Etat d'un programme - Exemple d'exécution

Rappel : Allouer = réserver de la mémoire.

Retenir

*Lors de **l'exécution** du programme, **l'espace nécessaire prévu** pour le tableau d'activation de la fonction est **alloué sur la pile**, aux premières places disponibles dans celle-ci.*

Les informations administratives relatives à la fonction sont initialisées à ce stade.

main	2	**ADM**
x	1	?
return	0	?

L'ensemble des emplacements mémoire ainsi alloués à la fonction est appelé son **tableau d'activation.**

Etat d'un programme - Exemple d'exécution

L'effet de la première instruction du programme est d'affecter la valeur 5 à l'emplacement de la mémoire associé à la variable x. Puis la valeur de retour spécifiée pour la fonction est positionnée :

main	2	ADM
x	1	**5**
return	0	**0**

A la fin de l'exécution de la fonction, son tableau d'activation est **désalloué** (les cases ne sont plus réservées, mais les valeurs restent jusqu'à la prochaine utilisation de la case mémoire).

2	ADM
1	5
0	0

Etat d'un programme - Règles...

Retenir (Règle 0)

Avant l'exécution du programme :

- *la pile est vide*
- *les variables statiques sont réservées et initialisées.*

Etat d'un programme - Règles...

Retenir (Règle 1)

*lorsqu'on **entre** dans une fonction, on **empile son tableau d'activation** dans la pile.*

Pile :

main	2	ADM
x	1	?
return	0	?

On supposera pour simplifier que les emplacements mémoire de la pile sont alloués dans l'ordre croissant des numéros, en partant de 0.

Etat d'un programme - Règles...

Retenir (Règle 2)

*Lorsqu'on a **terminé l'exécution** d'une fonction (instruction `return`) :*

- *le programme **dépile** son tableau d'activation (on libère les emplacements mémoire qui avaient été alloués par cette fonction)*
- *le programme **revient** à l'environnement qui était en cours juste avant l'appel de la fonction considérée.*

Rappel : en fin de procédure (type de retour `void`), le compilateur ajoute automatiquement un `return;` implicite s'il n'y en a pas.

Fonction appelant une autre fonction :

Si une fonction en appelle une autre, un nouveau tableau d'activation est empilé sur le premier.

Retenir

*Les **seules variables visibles dans la pile** sont celles de l'environnement de la fonction **active** (en cours d'exécution). Elles appartiennent au **tableau d'activation visible** qui est **celui du dessus de la pile**.*

Mais les cases mémoires appartenant aux tableaux d'activation des fonctions en attente sont toujours réservées et conservent leur contenu !

Fonction appelant une autre fonction :

Si une fonction en appelle une autre, un nouveau tableau d'activation est empilé sur le premier.

Retenir

*Les **seules variables visibles dans la pile** sont celles de l'environnement de la fonction **active** (en cours d'exécution). Elles appartiennent au **tableau d'activation visible** qui est **celui du dessus de la pile**.*

Mais les cases mémoires appartenant aux tableaux d'activation des fonctions en attente sont toujours réservées et conservent leur contenu !

Pile = LIFO = Last In First Out

Retenir

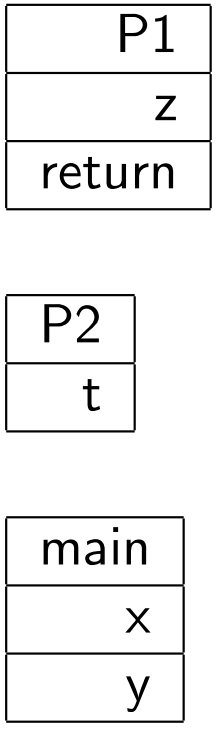
*La **Pile** fonctionne comme une pile d'assiettes :*

- *On **alloue** un nouveau tableau d'activation en le posant **sur le sommet de la pile**.*
- *On **désalloue** toujours le tableau qui est au sommet de la pile.*

Exemple de fonction appelant une autre fonction

```
1  int P1() {
2      int z;
3      z = 1;
4      return z + 1;
5  }
6  void P2() {
7      int t;
8      t = 10;
9  }
10 void main() {
11     int x, y;
12     x = 5;
13     y = P1();
14     x = x + y;
15     P2();
16 }
```

Tableaux d'activations :



Au début :

```
 1  int P1() {
 2      int z;
 3      z = 1;
 4      return z + 1;
 5  }
 6  void P2() {
 7      int t;
 8      t = 10;
 9  }
10  void main() {
11      int x, y;
12      x = 5;
13      y = P1();
14      x = x + y;
15      P2();
16  }
```

Pile :

7	?
6	?
5	?
4	?
3	?
2	?
1	?
0	?

On entre dans `main`

```
1  int P1() {
2      int z;
3      z = 1;
4      return z + 1;
5  }
6  void P2() {
7      int t;
8      t = 10;
9  }
10 void main() {
11     int x, y;
12     x = 5;
13     y = P1();
14     x = x + y;
15     P2();
16 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
	3	?
main	2	ADM
x	1	?
y	0	?

Ligne 12 : Affectation de x

```
1  int P1() {
2      int z;
3      z = 1;
4      return z + 1;
5  }
6  void P2() {
7      int t;
8      t = 10;
9  }
10 void main() {
11     int x, y;
12     x = 5;
13     y = P1();
14     x = x + y;
15     P2();
16 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
	3	?
main	2	ADM
x	1	5
y	0	?

Ligne 13 : appel de P1

```
1  int P1() {
2      int z;
3      z = 1;
4      return z + 1;
5  }
6  void P2() {
7      int t;
8      t = 10;
9  }
10 void main() {
11     int x, y;
12     x = 5;
13     y = P1();
14     x = x + y;
15     P2();
16 }
```

Pile :

	7	?
	6	?
P1	5	ADM
z	4	?
return	3	?
	2	ADM
	1	5
	0	?

Exécution de P1 : ligne 3

```
1  int P1() {
2      int z;
3      z = 1;
4      return z + 1;
5  }
6  void P2() {
7      int t;
8      t = 10;
9  }
10 void main() {
11     int x, y;
12     x = 5;
13     y = P1();
14     x = x + y;
15     P2();
16 }
```

Pile :

	7	?
	6	?
P1	5	ADM
z	4	1
return	3	?
	2	ADM
	1	5
	0	?

Exécution de P1 : ligne 4

```
1  int P1() {
2      int z;
3      z = 1;
4      return z + 1;
5  }
6  void P2() {
7      int t;
8      t = 10;
9  }
10 void main() {
11     int x, y;
12     x = 5;
13     y = P1();
14     x = x + y;
15     P2();
16 }
```

Pile :

	7	?
	6	?
P1	5	ADM
z	4	1
return	3	2
	2	ADM
	1	5
	0	?

Retour à main en ligne 13

```
 1  int P1() {
 2      int z;
 3      z = 1;
 4      return z + 1;
 5  }
 6  void P2() {
 7      int t;
 8      t = 10;
 9  }
10  void main() {
11      int x, y;
12      x = 5;
13      y = P1();
14      x = x + y;
15      P2();
16  }
```

Pile :

	7	?
	6	?
	5	ADM
	4	1
	3	2
main	2	ADM
x	1	5
y	0	2

`main` en ligne 14

```
 1  int P1() {
 2      int z;
 3      z = 1;
 4      return z + 1;
 5  }
 6  void P2() {
 7      int t;
 8      t = 10;
 9  }
10  void main() {
11      int x, y;
12      x = 5;
13      y = P1();
14      x = x + y;
15      P2();
16  }
```

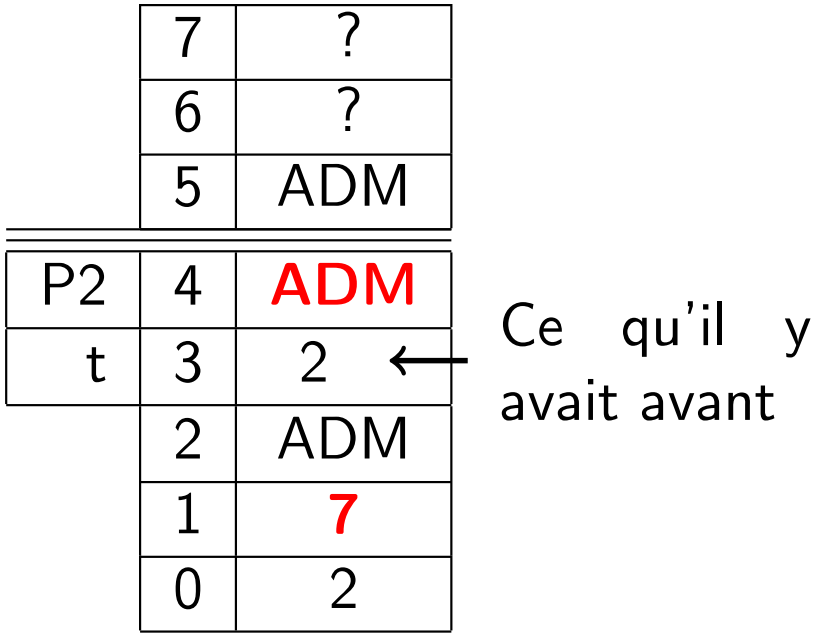
Pile :

	7	?
	6	?
	5	ADM
	4	1
	3	2
main	2	ADM
x	1	7
y	0	2

`main` en ligne 15, appel de `P2`

```
 1  int P1() {
 2      int z;
 3      z = 1;
 4      return z + 1;
 5  }
 6  void P2() {
 7      int t;
 8      t = 10;
 9  }
10  void main() {
11      int x, y;
12      x = 5;
13      y = P1();
14      x = x + y;
15      P2();
16  }
```

Pile :



P2 ligne 8

```
 1  int P1() {
 2      int z;
 3      z = 1;
 4      return z + 1;
 5  }
 6  void P2() {
 7      int t;
 8      t = 10;
 9  }
10  void main() {
11      int x, y;
12      x = 5;
13      y = P1();
14      x = x + y;
15      P2();
16  }
```

Pile :

	7	?
	6	?
	5	ADM
P2	4	ADM
t	3	10
	2	ADM
	1	7
	0	2

retour à `main` ligne 16

```
 1  int P1() {
 2      int z;
 3      z = 1;
 4      return z + 1;
 5  }
 6  void P2() {
 7      int t;
 8      t = 10;
 9  }
10  void main() {
11      int x, y;
12      x = 5;
13      y = P1();
14      x = x + y;
15      P2();
16  }
```

Pile :

	7	?
	6	?
	5	ADM
	4	ADM
	3	10
main	2	ADM
x	1	7
y	0	2

Fin du programme

```
 1  int P1() {
 2      int z;
 3      z = 1;
 4      return z + 1;
 5  }
 6  void P2() {
 7      int t;
 8      t = 10;
 9  }
10  void main() {
11      int x, y;
12      x = 5;
13      y = P1();
14      x = x + y;
15      P2();
16  }
```

Pile :

7	?
6	?
5	ADM
4	ADM
3	10
2	ADM
1	7
0	2

Fonctionnement de la pile

Retenir (pointeur de pile)

En pratique :

- *le processeur ne retient que **l'adresse du sommet de la pile**.*
- *tout ce qui est en dessous est réservé.*
- *tout ce qui est au dessus est libre.*
- *on **alloue** une portion de mémoire en ajoutant au pointeur de pile la taille de la portion*
- *on **désalloue** la même portion en retranchant au pointeur la même taille.*

Pointeur de pile

Compléments

Comme pour le **pointeur d'instruction** qui retient où on en est de l'exécution du programme, il existe un registre nommé **pointeur de pile** qui retient l'adresse du sommet de la pile.

Les adresses des variables de l'environnement actif sont repérées par rapport au sommet de la pile.

Variables locales statiques

Compléments

Par défaut, les variables locales sont automatiques. Il est possible de changer ce comportement avec le mot clé `static`.

```
1  #include<iostream>
2  using namespace std;
3  int compte() {
4      static int res = 0;
5      res++;
6      return res;
7  }
8  int main() {
9      cout << compte() << endl;
10     cout << compte() << endl;
11     cout << compte() << endl;
12 }
```

Affiche

1

2

3

Pour le moment nous n'avons vu que des fonctions sans paramètres...

Rappel : fonction

Syntaxe

Déclaration (mode d'emploi) :

```
type_retour nom(type1 <param1>, type2 <param2>, ...);
```

Définition :

```
type_retour nom(type1 param1, type2 param2, ...) {
    déclaration des variables locales;
    intructions;
    ...
    return valeur_de_retour;
}
```

Appel (utilisation dans une expression) :

```
... nom(valeur_param1, valeur_param2, ...) ...
```

Liste des paramètres formels

- Pour contrôler les incohérences, le compilateur doit connaître le type des valeurs données en entrée d'une fonction.
- On associe ces valeurs à des identificateurs locaux à la fonction.

Définition

- ***paramètre formel*** *: identificateur local qui reçoit une valeur donnée à la fonction ;*
- ***paramètre réel ou effectif*** *: valeur donnée à la fonction.*
- *Dans la déclaration, les paramètres formels, munis de leur type, sont séparés par des «,»*

Rappel : Appel d'une fonction

Retenir

- *Calcul du résultat d'une fonction pour certaines valeurs,*
- *Le **programme appelant** doit appeler la fonction en lui transmettant ces valeurs.*
- *La syntaxe est*

  `nomDeLaFonction(liste des paramètres réels)`

  qui doit apparaître dans une expression.
- *On peut appeler une fonction autant de fois que l'on veut.*

Passage des paramètres

Retenir

- *La liste des paramètres réels est constituée d'une liste d'expressions séparées par des virgules.*
- *Les paramètres réels doivent* ***correspondre en type et en nombre*** *aux paramètres formels, sinon une erreur est détectée lors de la compilation.*
- *Il est important de* ***respecter l'ordre*** *des paramètres formels.*
- *La valeur de chaque expression est calculée et devient ainsi la valeur du paramètre formel correspondant.*

Rappel : La partie déclarations

- Si le calcul du résultat de la fonction est complexe, il peut être utile de définir des objets locaux comme des variables intermédiaires, des constantes.
- La définition de la fonction comprend donc une partie déclarative, exactement comme le programme principal.

Variables locale à la fonction

Retenir

Ne pas confondre *:*

- *paramètre : emplacement mémoire qui permet à la fonction de* ***communiquer en échangeant des valeurs avec le reste du programme****.*
- *variable locale :* ***emplacement de stockage temporaire*** *dont a besoin la fonction pour effectuer son calcul.*

Passage des paramètres par copie

Les paramètres formels d'une fonction sont des variables comme les autres, on peut les modifier. Mais...

Retenir

*Lors d'un appel de fonction ou de procédure, la valeur du paramètre réel est **copiée** dans le paramètre formel.*

En conséquence

- Une modification du paramètre formel n'affecte pas le paramètre réel ;
- Si la variable est compliquée (tableaux, chaîne de caractères etc), cette recopie peut être coûteuse.

Passage de paramètres...

Lors de l'appel d'une fonction ou procédure, les étapes suivantes sont exécutées :

Retenir

1. *Empilement dans la mémoire de son tableau d'activation (**dont les emplacements nécessaires pour ses paramètres**) ;*
2. ***Initialisation des valeurs des paramètres** («passage des paramètres»), et donc modification de la mémoire ;*
3. *Exécution du corps de la fonction, dans l'environnement : la mémoire est ainsi modifiée ;*
4. *Pour les fonctions, retour de la valeur de la fonction ;*
5. *Dépilement du bloc d'activation et retour à l'environnement qui était en place avant l'appel. Seul l'environnement est restauré, les effets de la procédure sur la mémoire persistent.*

Pourquoi passer un paramètre ?

Les paramètres servent à communiquer avec la fonction.

Retenir

Il peut y avoir trois raison de communiquer :

1. *on veut faire passer* ***une donnée*** *à la fonction*
2. *on veut récupérer* ***un résultat*** *de la fonction*
3. *on veut que la fonction* ***modifie une variable*** *qui a déjà une valeur.*

2 et 3 seront traités au prochain cours.

Exemple de passage d'une **donnée**

Fonction calculant la somme de deux entiers :

```
int somme(int a, int b);
```

Les paramêtres `a` et `b` sont des **données** de la fonction.

Exemple de passage d'une **donnée**

```
1  /** La somme de deux entiers
2  * @param[in] a b deux entiers
3  * @return    a+b
4  **/
5  int somme(int a, int b) {
6      int res;
7      res = a + b;
8      return res;
9  }
10
11 int main() {
12     cout << somme(4, 6) << endl;
13     int x;
14     cin >> x;
15     cout << somme(x, 3) << endl;
16 }
```

Sémantique des passages de paramètres

Question

Quels sont les mécanismes utilisé pour les différents passages de paramètres ?

Nous allons voir maintenant, de manière simplifiée, comment les fonctions communiquent les unes avec les autres.

Sémantique des passages de paramètres

Question

Quels sont les mécanismes utilisé pour les différents passages de paramètres ?

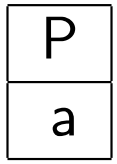
Nous allons voir maintenant, de manière simplifiée, comment les fonctions communiquent les unes avec les autres.

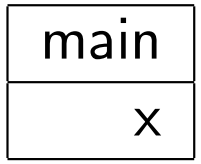
Passage par valeur

Retenir

À la compilation, comme pour les variables locales, ***des emplacements pour les paramêtres sont prévus dans les tableaux d'activation*** *pour être alloués au moment de l'appel de la fonction.*

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```





Passage par valeur

Retenir (Passage par valeur)

Au moment de l'appel :

- *les **paramètres réels sont évalués** dans l'environnement de la fonction appelante.*
- *les valeurs ainsi obtenues sont **affectées** aux paramètres formels correspondants. Il y a donc une affectation :*

$$\textit{paramètre formel} = \textit{paramètre réel}$$

On recopie les valeurs des paramètres réels. Certains appellent donc le passage par valeur **passage par copie**

Exemple de passage par valeur

Début du programme

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

7	?
6	?
5	?
4	?
3	?
2	?
1	?
0	?

Exemple de passage par valeur

Entrée de la fonction `main`

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
	3	?
	2	?
main	1	ADM
x	0	?

Exemple de passage par valeur

Ligne 7 ; Appel de `P(10)`

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
P	3	ADM
a	2	10
	1	ADM
	0	?

Exemple de passage par valeur

Ligne 2 ; affichage de «a = 10»

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
P	3	ADM
a	2	10
	1	ADM
	0	?

Exemple de passage par valeur

Ligne 3

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
P	3	ADM
a	2	**0**
	1	ADM
	0	?

Exemple de passage par valeur

Retour au `main` :

```
 1  void P(int a) {
 2    cout << "a =  " << a << endl;
 3    a = 0;
 4  }
 5  void main () {
 6    int x;
 7    P(10);
 8    x = 5;
 9    P(x);
10    cout << "x = "  << x << endl;
11  }
```

Pile :

	7	?
	6	?
	5	?
	4	?
	3	ADM
	2	0
main	1	ADM
x	0	?

Exemple de passage par valeur

`main` ligne 8 :

```
 1  void P(int a) {
 2    cout << "a =  " << a << endl;
 3    a = 0;
 4  }
 5  void main () {
 6    int x;
 7    P(10);
 8    x = 5;
 9    P(x);
10    cout << "x = "  << x << endl;
11  }
```

Pile :

	7	?
	6	?
	5	?
	4	?
	3	ADM
	2	0
main	1	ADM
x	0	**5**

Exemple de passage par valeur

`main` ligne 9 : appel de `P(x)`, c'est-à-dire `P(5)`

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
P	3	ADM
a	2	5
	1	ADM
	0	5

Exemple de passage par valeur

Ligne 2 ; affichage de «a = 5»

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
P	3	ADM
a	2	5
	1	ADM
	0	5

Exemple de passage par valeur

Ligne 3

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
P	3	ADM
a	2	**0**
	1	ADM
	0	5

Exemple de passage par valeur

Retour au `main` ligne 10 : affichage de «`x = 5`»

```
1  void P(int a) {
2    cout << "a =  " << a << endl;
3    a = 0;
4  }
5  void main () {
6    int x;
7    P(10);
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

	7	?
	6	?
	5	?
	4	?
	3	ADM
	2	0
main	1	ADM
x	0	5

Passage par valeur = copie

Remarque

Lors de l'appel à `P(x)` où `x` vaut 10 il y a une affectation a=10. C'est une copie de la valeur 10.

Par la suite, l'affectation `a = 0`, ne change pas la valeur du paramètre formel `x`, dans la fonction `P`, le paramètre `a` contient une copie de la valeur du paramètre réel `x`.

Passage par valeur = copie

Remarque

Lors de l'appel à `P(x)` où `x` vaut 10 il y a une affectation `a=10`. C'est une copie de la valeur 10.

Par la suite, l'affectation `a = 0`, ne change pas la valeur du paramètre formel `x`, dans la fonction `P`, le paramètre a contient une **copie** de la valeur du paramètre réel `x`.

Passage par valeur = copie

Remarque

Lors de l'appel à `P(x)` où `x` vaut 10 il y a une affectation `a=10`. C'est une copie de la valeur 10.

Par la suite, l'affectation `a = 0`, ne change pas la valeur du paramètre formel `x`, dans la fonction `P`, le paramètre a contient une **copie** de la valeur du paramètre réel `x`.

Passage par valeur = copie

Retenir

*Les **paramètres formels passés par valeur** sont des **variables** comme les autres, qui sont **initialisées lors du passage de paramètres**.*

*Lors du passage par valeur, **toute modification d'un paramètre formel est sans effet sur les paramètres effectifs** et, de façon générale, sur l'environnement qui sera restauré au retour de l'appel.*

Si `x` vaut 10, l'appel par valeur de `P(x)` a un effet strictement identique à l'appel `P(10)`.

Passage par valeur = copie

Retenir

*Les **paramètres formels passés par valeur** sont des **variables** comme les autres, qui sont **initialisées lors du passage de paramètres**.*

*Lors du passage par valeur, **toute modification d'un paramètre formel est sans effet sur les paramètres effectifs** et, de façon générale, sur l'environnement qui sera restauré au retour de l'appel.*

Si `x` vaut 10, l'appel par valeur de `P(x)` a un effet strictement identique à l'appel `P(10)`.

Passage par valeur = copie

Retenir

*Les **paramètres formels passés par valeur** sont des **variables** comme les autres, qui sont **initialisées lors du passage de paramètres**.*

*Lors du passage par valeur, **toute modification d'un paramètre formel est sans effet sur les paramètres effectifs** et, de façon générale, sur l'environnement qui sera restauré au retour de l'appel.*

Si `x` vaut 10, l'appel par valeur de `P(x)` a un effet strictement identique à l'appel `P(10)`.