Programmation Impérative II
Sémantique d'un programme C++

Florent Hivert

Mél : Florent.Hivert@lri.fr
Adresse universelle : http://www.lri.fr/~hivert

Syntaxe et Sémantique

Définition

*La **syntaxe** d'un langage de programmation (ou d'un langage réel) est la description des expressions correctes du langage,* ***indépendamment de leur sens****.*

*La **sémantique** d'un langage est la description du sens des constructions du langage.*

Comme dans une langue naturelle, les phrases syntaxiquement correctes d'un langage de programmation n'ont pas toutes un sens.
Ex : le chat est analytique à l'origine.

Statique / Dynamique

Le **compilateur** traduit le programme **source** (par exemple en C++) en un programme exécutable **binaire** (codé en langage machine).

Définition

*Ce qui est fait (décision, réservation de ressource, détection d'erreur) pendant la compilation est dit **statique**.*

*La même chose faite pendant l'exécution est dite **dynamique**.*

Exemple :

- incohérence de sens identifiée sans exécuter le programme : erreur de sémantique **statique**.
- incohérence qui apparaît lors de l'exécution du programme : erreur de sémantique **dynamique**.

Exemple d'erreurs de sémantique statique

sem-stat.cpp

```
 1  #include <iostream>
 2  #include <vector>
 3  using namespace std;
 4
 5  void pp (float z) {
 6      int i;
 7      i = 2;
 8      z = i;
 9  }
10
11  int main () {
12      int x;
13      vector<int> t(10);
14      cin >> i;
15      x = t;
16      pp(x, t);
17      return 0;
18  }
```

- Ligne 14 : La variable `i` est déclarée seulement à l'intérieur de la fonction `pp`
- Ligne 15 : Affectation d'un vecteur dans un entier
- Ligne 16 : Appel de la fonction `pp` avec deux paramètres au lieu d'un.

Erreurs de sémantique statique

Incohérence entre la **déclaration** d'un objet et son **utilisation** : il faut déclarer les objets (types, variables, fonctions, etc.) avant de les utiliser, pour vérifier que l'utilisation est conforme à la déclaration.

- erreurs sur la **portée des variables** : une variable déclarée n'est utilisable que dans une certaine partie du programme
- erreurs sur le **contrôle des types** : les utilisations d'une variable doivent être cohérentes avec son type déclaré
- erreurs lors de **l'appel des fonctions**

Nous allons voir cela en détail...

Sémantique statique : Bilan

L'ensemble de ces contrôles correspond à la vérification de la sémantique statique du programme, c'est-à-dire de la **partie de la sémantique qui peut être vérifiée sans exécuter le programme.**

Retenir

*La **sémantique statique** est contrôlée par le compilateur :*

La traduction (en langage machine) et l'exécution d'un programme n'est pas possible tant que la sémantique statique n'est pas correcte.

On ne peut pas traduire et a fortiori exécuter un programme qui n'a pas de sens.

Sémantique dynamique

Problème

On ne peut pas décrire toute la sémantique d'un langage uniquement par sa sémantique statique !

Des erreurs peuvent apparaître à l'exécution, par exemple :

- division par zéro
- accès en dehors des bornes d'un tableau
- plus généralement : mauvais accès à la mémoire
- ressource (fichier, réseau, ...) indisponible.

Sémantique dynamique

sem-dyn.cpp

```
 1  int f (int n) {
 2      int res ;
 3      if (n > 2)  res = f(n-1) + f(n-2);
 4      else        res = n;
 5      return res;
 6  }
 7
 8  int main () {
 9      int i;
10      vector<int> t(10);
11      i = 1 ;
12      // ... ici l'initialisation du tableau t ...
13      while (f(i) < 1000) {
14          cout << t[i] << endl; // Accès en dehors du tableau
15          i++;
16      };
17      // ...
18  }
```

Sémantique dynamique...

Définition

*La **sémantique dynamique** est la partie de la sémantique qui ne peut être contrôlée que lors de l'exécution du programme.*

Avantages des langages compilés :

- diminuer la part de la sémantique dynamique par rapport à la sémantique statique.
- un plus grand nombre d'erreurs sémantiques détectées à la compilation
- les erreurs d'exécution, plus difficiles à découvrir et à corriger, sont rares.

Encore plus de sémantique

Compléments

Dans les langages modernes, on essaye de mettre encore plus de sémantique :

- Qui est responsable de quelle portion de mémoire ?
  Langage objets (`C++`, `Java`), notion de propriétaire (`rust`)
- Spécification, preuve
  `Why`, `Spark`, assistant de preuve `Coq`, `Isabelle`

Comment décrire la sémantique ?

Mécanismes de description

Objectif de la première partie du cours : **décrire la sémantique** d'un langage de programmation impératif :

- environnement et mémoire ;
- portée des déclarations ;
- appel des fonctions ;
- passage de paramètres.

Il nous faut un modèle de machine simplifié qui permette de décrire assez fidèlement le comportement des programmes.

La mémoire

La mémoire principale d'un ordinateur est composée de

- un très grand nombre (10^{10}–10^{11}) de **condensateurs** pouvant être soit chargés soit déchargés (0 ou 1)
- un gigantesque **réseau de fils et d'aiguillages** (multiplexeur)

Retenir

*On groupe les condensateurs par groupes (typiquement 8, 16, 32 ou 64) appelés **mots**. On repère un mot en mémoire grâce à un nombre appelé **adresse**.*

L'opérateur `&`, retourne l'adresse d'une variable :

addr.cpp

```
1    int a = 5;
2    cout << a << " " << &a << endl;
```

Affiche quelque chose comme `5 0x7ffe94f3a8bc`

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.

La mémoire en vrai

Problème

Plus une mémoire est **grosse**, plus il faut commuter d'aiguillages pour accéder à une case mémoire, plus la mémoire est **lente**.

Compléments

Hiérarchie de mémoires :

1. registres : mémoire très rapide, très proche des unités de calcul
2. caches (souvent plusieurs niveaux L1, L2, L3)
3. mémoire centrale

Du point de vue de la sémantique, c'est juste une **optimisation** qui permet au programme d'être exécuté (beaucoup) plus rapidement.

Initialisation

Retenir

*La **première écriture** dans un emplacement mémoire s'appelle **l'initialisation**.*

*Si l'on n'a rien écrit, l'emplacement mémoire contient quand même une valeur (les condensateurs correspondants contiennent ou non une charge), mais cette valeur est **imprévisible**.*

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

Certains langages ont décidé de tout initialiser à 0. Ce n'est en général **pas le cas du** `C/C++`

Exemple de problème en cas de non initialisation

noninit.cpp

```
1  int add(int x, int y) {
2      int res;
3      cout << "Debut :  " << res << endl;   // pas encore initialisée
4      res = x + y;
5      cout << "Fin :  " << res << endl;
6      return res;
7  }
```

Après deux appels consécutifs, on peut voir afficher :

```
Debut : 0
Fin : 37
[...]
Debut : 32619
Fin : 37
```

Mais le compilateur affiche un message d'avertissement :

`warning: ‘res’ is used uninitialized in this function.`

Rappel : La notion de variable

- but : stocker des informations en mémoire centrale durant l'exécution d'un programme ;
- on veut éviter d'avoir à manipuler directement les adresses ; on manipule des **variables** ;
- le programmeur donne aux variables des noms de son choix (**identificateur**) ;

Définition (Notion de Variable)

*Une **variable** est un espace de stockage où le programme peut mémoriser une donnée.*

Les variables désignent une ou plusieurs cases mémoires contenant une suite de 0 et de 1 qui code la valeur de la variable.

Rappel : La notion de variable (2)

Retenir

Une variable en cours d'utilisation possède quatre propriétés :

- *un nom ;*
- *une adresse ;*
- *un type ;*
- *une valeur.*

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.

Variables : portée et durée de vie

Une variable n'est ni valable dans tout le code, ni présente en mémoire tout le temps...

Définition

*La **portée d'une déclaration** d'une variable indique dans quelle portion du programme source la variable est **visible**.*
C'est une question de sémantique statique.

*La **durée de vie** d'une variable indique la portion de temps d'exécution pendant laquelle la **place mémoire** associée à la variable est **réservée**.*
C'est une question de sémantique dynamique.

Exemple : portée et durée de vie

addr-glob-loc.cpp

```
1  #include<iostream>
2
3  int g = 3; // Variable globale, allocation statique
4
5  int main() {
6      std::cout << "Globale " << g << " " << &g << std::endl;
7
8      int h = 5;  // Variable locale, allocation automatique
9      std::cout << "Locale "<< h << " " << &h << std::endl;
10 }
```

Portée des déclarations...

Occurrence : apparition d'une variable.

Définition

- ***occurrence de liaison*** *: déclaration (**relie** un nom de variable à un emplacement mémoire).*
- ***occurrence d'utilisation*** *: toutes les autres apparitions. On utilise une variable*
  - *pour lire son contenu (évaluation)*
  - *pour le changer (affectation)*
  - *pour connaître son adresse (référence – voir section 6)*

Note : Il peut y avoir aussi plusieurs occurrences de liaison correspondant à un même nom de variable.

Portée des déclarations...

Retenir

*Les **règles de portée** permettent de déterminer si une occurrence d'utilisation correspond bien à une occurrence de liaison et, le cas échéant, à laquelle. Dans le cas contraire, une **erreur de sémantique** a été commise !*

Règles de Portée de C++ (simplifiées)

Retenir

1. *Une variable est **locale** lorsqu'elle est déclarée **à l'intérieur** d'une fonction*
   $\Longrightarrow$ *La portée d'une variable locale s'étend depuis le point de déclaration jusqu'à la fin du bloc (paire d'accolades) où se trouve cette déclaration*
2. *Une variable est **globale** lorsqu'elle est défini **en dehors** d'une fonction, d'une classe, ou d'un espace de noms...*
   $\Longrightarrow$ *La portée d'un nom global s'étend du point de déclaration à la fin du fichier.*

Ainsi, des variables déclarées dans des fonctions différentes sont différentes, même si elles portent le même nom : elles correspondent à des **emplacements mémoire différents**.

Exemple

polynom.cpp

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  double power(double a, int n) {
6      double res = 1.;
7      int i;
8      for (i = 0; i<n; i++) {
9          res = res *a;
10     }
11     // Un affichage de i ici serait valide.
12     return res;
13 }
14
15 double evalpoly(vector<double> v, double x) {
16     double res = 0;
17     int i;
18     for (i = 0; i< v.size(); i++) {
19         res = res + power(x, i) * v[i];
20     }
21     return res;
22 }
23
24
25 int main() {
26     vector<double> pol = {1.5, 2.5, 2.5, 4.0};
27     cout << "adress " << &pol << endl;
28     cout << evalpoly(pol, 3.1) << endl;
29     return 0;
30 }
```

Règles de Portée de C++ : cas particulier du `for`

Retenir

Dans le cas des boucles (en particulier `for`), ***ce qui est entre parenthèses fait partie du bloc***

```
1  for (int i=0; i<10; i++) {
2      cout << i << endl; // valide
3  }
4  cout << i << endl; // erreur
```

Note : quelques (vieux) compilateurs acceptent ce code, mais il est invalide en C++ 2011.

Exemple avec for

polynom-for.cpp

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   double power(double a, int n) {
6       double res = 1.;
7       for (int i = 0; i<n; i++) {
8           res = res *a;
9       }
10      // Un affichage de i ici ne serait pas valide.
11      return res;
12  }
13
14  double evalpoly(vector<double> v, double x) {
15      double res = 0;
16      for (int i = 0; i< v.size(); i++)
17          res = res + power(x, i) * v[i];
18      return res;
19  }
20
21  int main() {
22      vector<double> pol = {1.5, 2.5, 2.5, 4.0};
23      cout << evalpoly(pol, 3.1) << endl;
24      return 0;
25  }
```

Portée des déclarations : Masquage

Retenir

*Une déclaration d'un nom dans un bloc peut en **masquer** une autre située dans le bloc conteneur.*

Exemple de programmation rigoureusement déconseillée :

```
1  int x; // x global
2
3  void f() {
4      int x;  // x local masque le x global
5      x = 1;  // affectation au x local
6      {
7          int x; // masque le premier x local
8          x = 2; // affectation au second x local
9      }
10     x = 3;  // affectation au premier x local
11 }
```

Retenir (masquages)

*On ajoute un **indice pour relier les occurrences** d'utilisation avec la bonne occurrence de liaison. On est ainsi ramené au cas simple du programme qui ne présente pas de masquage.*

```
1  int x;
2
3  void f() {
4      int x;
5      x=1;
6      {
7          int x;
8          x=2;
9      }
10     x=3;
11 }
```

```
1  int x1;
2
3  void f() {
4      int x2;
5      x2 = 1;
6      {
7          int x3;
8          x3 = 2;
9      }
10     x2 = 3;
11 }
```

Portée des déclarations : Bilan

Retenir

Pour chercher la déclaration d'une variable (occurrence de liaison), on part toujours du bloc le plus interne ; si l'on ne l'a pas trouvée, on cherche dans le bloc conteneur sans rentré dans les sous-blocs précédent, . . ., jusqu'aux déclarations globales.

Maintenant que l'on sait quelle utilisation correspond à quelle déclaration, il faut décrire les mécanismes de réservation de mémoire.

Variables globales

Retenir

*Les **variables globales** (valables dans tout le programme) sont **statiques**, c'est-à-dire :*

- *elles ont pour durée de vie tout le temps d'exécution du programme*
- *elles sont allouées statiquement dans le segment de données statiques*

Le compilateur décide de l'emplacement mémoire de la variable. Cet emplacement est réservé au chargement du programme en mémoire et est libéré à la fin du programme.

Remarque

L'usage des variables globales est, sauf bonne raison, **déconseillé**.

Variables locales

Rappel : bloc = portion entre deux accolades qui se correspondent (exemple : corps de fonction, corps de boucle ...).

Retenir

*Les **variables locales** (déclarées dans un bloc) sont (par défaut) **automatiques**, c'est-à-dire :*

- *ont pour durée de vie le temps d'exécution du bloc*
- *sont allouées dynamiquement dans le segment de pile*

L'emplacement mémoire est décidé et réservé quand on exécute le début du bloc. Il est libéré à la fin du bloc.

Retenir

*La valeur de la variable est **perdue** entre deux exécutions du même bloc.*

Adresses des variables

addr-glob-loc.cpp

```
1  #include<iostream>
2
3  int g = 3; // Variable globale, allocation statique
4
5  int main() {
6      std::cout << "Globale " << g << " " << &g << std::endl;
7
8      int h = 5;  // Variable locale, allocation automatique
9      std::cout << "Locale "<< h << " " << &h << std::endl;
10 }
```

- L'adresse de g est fixée à la compilation
- L'adresse de h change d'une exécution à l'autre

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 - Exemple de compilation

Considérons le programme suivant :

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

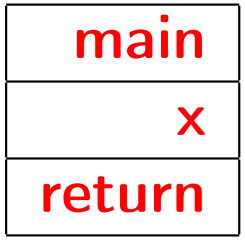
Lors de la **compilation**, on lit que ce programme ne contient pas de variables globales et qu'il est composé d'une unique fonction, le `main`, qui contient une déclaration (`int x`), une instruction d'affectation (`x = 5`) et une instruction de retour (`return 0`).

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 !

Pile = LIFO = Last In First Out

Retenir

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

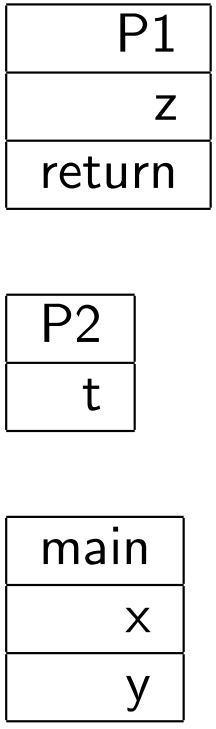
- *On **alloue** un nouveau tableau d'activation en le posant **sur le sommet de la pile**.*
- *On **désalloue** toujours le tableau qui est au sommet de la pile.*

Exemple de fonction appelant une autre fonction

expl-call.cpp

```
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 :

expl-call.cpp

```
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`

expl-call.cpp

```
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

expl-call.cpp

```
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

expl-call.cpp

```
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

expl-call.cpp

```
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	?

Exécution de P1 : ligne 4

expl-call.cpp

```
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

expl-call.cpp

```
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

expl-call.cpp

```
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 :

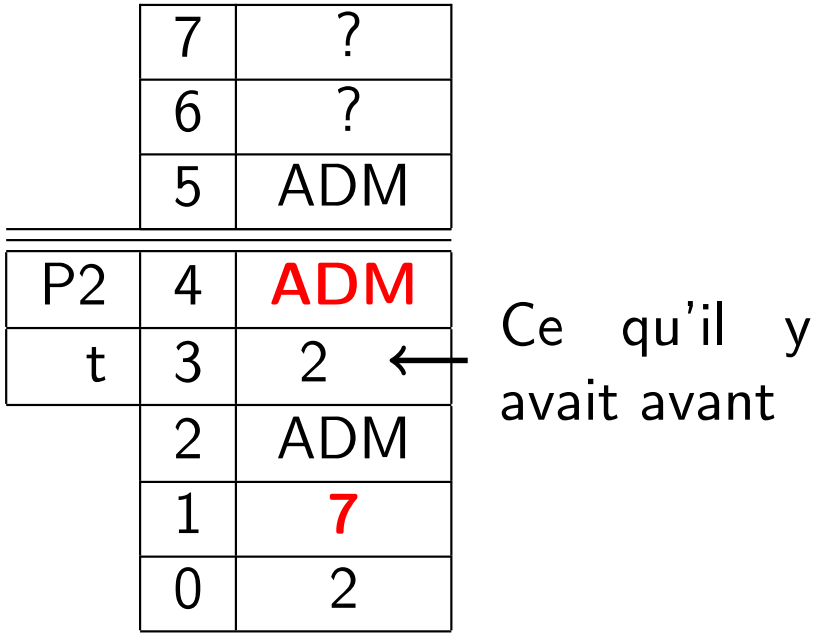
	7	?
	6	?
	5	ADM
	4	1
	3	2
main	2	ADM
x	1	7
y	0	2

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

expl-call.cpp

```
 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

expl-call.cpp

```
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

`expl-call.cpp`

```
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

expl-call.cpp

```
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`.

locstatic.cpp

```
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 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.*

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**

somme.cpp

```
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 }
```

Exemple de passage d'un résultat

On a souvent besoin d'une fonction qui retourne **deux** résultats.
Pour le moment, avec l'instruction `return`, on ne peut en retourner qu'un seul.

Fonction qui retourne si un entier est une puissance de deux et qui renvoie l'exposant de la plus petite puissance supérieure ou égale à l'entier :

- pour 4, on doit répondre : vrai, 2
- pour 9, on doit répondre : faux, 4

Exemple de passage d'un **résultat**

Fonction qui retourne si un entier est une puissance de deux et qui renvoie l'exposant de la plus petite puissance supérieure ou égale à l'entier :

```
bool estPuissanceDe2(int n, int &exp);
```

- Le paramètre `n` est une **donnée**
- Le paramètre `exp` est un **résultat**

Remarque

- Noter le «&» devant le paramètre `exp`.
- Lors de l'appel, la variable `exp` n'est **pas initialisée**, c'est le rôle de la fonction `estPuissanceDe2` de le faire.

Exemple de passage d'un **résultat**

estpuiss2.cpp

```
1   /** Teste si un nombre est une puissance de 2 et calcule l'exposant.
2    * @param[in]  n     un nombre entier positif
3    * @param[out] exp   l'exposant de la plus petite
4    *                   puissance de 2 supérieure ou égale à n
5    * @return un boolean
6    **/
7   bool estPuissance2(int n, int &exp) {
8       exp = 0;
9       int puiss = 1;
10      while (puiss < n) {
11          exp++;
12          puiss *= 2;
13      }
14      return puiss == n;
15  }
16
17  int main() {
18      int a, e;
19      bool b;
20      cin >> a;
21      b = estPuissance2(a, e);
22      // Le booléen b est affiché avec 0 pour false et 1 pour true
23      cout << b << " " << e << endl;
24  }
```

Exemple de passage de **donnée/résultat**

C'est le cas où l'on veut que la fonction modifie une variable qui contient déjà une valeur :

- fonction `incrémente` qui fait la même chose que ++ :

  ```
  void incrémente(int &n)
  ```

- fonction `échange` qui échange le contenu de deux variables :

  ```
  void échange(int &n, int &m)
  ```

Remarque

- Le C++ ne fait **pas la différence** entre les modes de passage résultat et donnée/résultat.
- Dans le mode donnée/résultat, la variable doit être **initialisée avant l'appel**.

Exemple de passage de **donnée/résultat**

ajoute2.cpp

```
1  /** Ajoute 2 à un entier
2  * @param[in/out] a une variable entière
3  **/
4  void ajoute2(int &a) {
5      a = a + 2;
6  }
7
8
9  int main() {
10     int n;
11     cin >> n;
12     cout << "Avant :  " << n << endl;
13     ajoute2(n);
14     cout << "Apres :  " << n << endl;
15 }
```

Exemple de passage de **donnée/résultat**

echange.cpp

```
1  /** Échange les contenus de deux variables
2   * @param[in/out] a b deux variables entières.
3   **/
4  void echange(int &a, int &b) {
5      int t;
6      t = a;
7      a = b;
8      b = t;
9  }
10
11 int main() {
12     int x, y;
13     cin >> x >> y;
14     cout << "Avant :  " << x << " " << y << endl;
15     echange(x, y);
16     cout << "Apres :  " << x << " " << y << endl;
17 }
```

Passage de paramètres...

Deux modes principaux de passage de paramètres sont utilisés :

Retenir

- *le **passage par valeur**, utilisé pour passer une donnée. En C++, c'est le mode de passage par défaut.*
- *le **passage par référence** (ou par adresse) : correspond aux paramètres résultats ou données-résultats.*
  En C++, ce mode de passage est obtenu par l'utilisation de références (symbole «&»).

Sémantique des passages de paramètres

Question

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

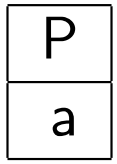
Nous allons voir maintenant, de manière simplifiée, comment les fonctions communiques les unes avec les autres.

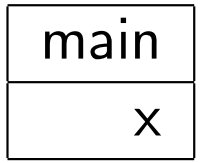
Passage par valeur

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 :*

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

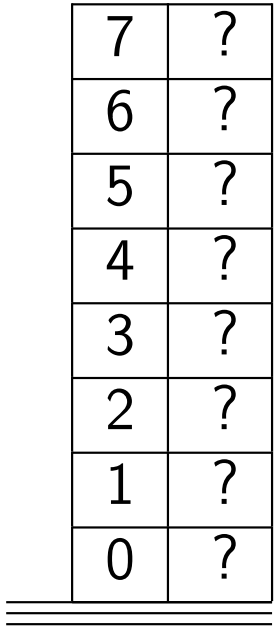
On recopie les valeurs des paramètres réels. Certains appellent donc le passage par valeur **passage par copie**

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 :



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 :

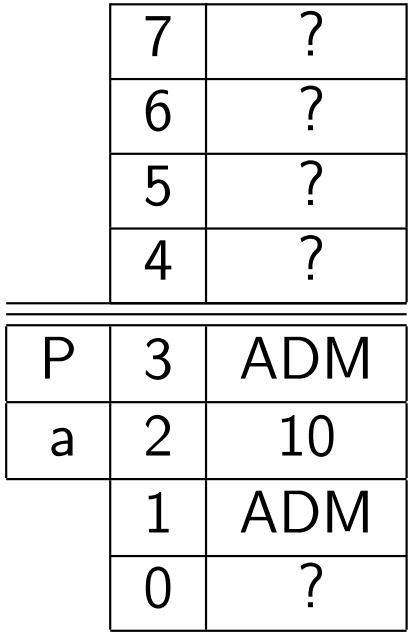
	7	?
	6	?
	5	?
	4	?
P	3	ADM
a	2	10
	1	ADM
	0	?

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 :



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

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)`.

Pour reporter les modifications, on fait un passage par référence

On en a besoin pour un paramètre en mode résultat ou donnée/résultat.

Notion de référence

Le C++ permet de manipuler un type particulier, dit «référence». **Une référence est un identificateur synonyme d'un autre identificateur**. Elle permet de manipuler une variable sous un autre nom que celui sous lequel cette variable a été déclarée.

Syntaxe

On déclare une référence par

$$type\ \&nom = variable\,;$$

Par exemple :

```
1       int i=1;
2       int  &ri = i; // référence sur la variable i
3       ri = ri * 2; // double la valeur de ri et de i !!
```

Référence

Dans l'exemple :

```
1       int i=1;
2       int  &ri = i; // référence sur la variable i
3       ri = ri * 2; // double la valeur de ri et de i !!
```

Les deux identificateurs `i` et `ri` **représentent alors la même variable** (le même emplacement mémoire) qu'ils peuvent tous les deux **accéder et modifier.**

Référence

Retenir

*En pratique, une **variable de type référence contient l'adresse de la variable référencée**.*
Nous utiliserons les conventions de dessin suivantes :

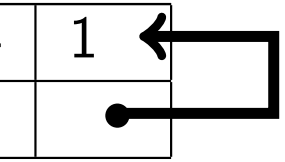
- *Dans l'environnement, **les variables de type référence seront précédées du symbole** `&`.*
- *On dessinera une flèche entre la référence et la variable référencée.*

```
1       int i=1;
2       int  &ri = i; // référence sur la variable i
3       ri = ri * 2; // double la valeur de ri et de i !!
```

i	4	1
&ri	3	**4**

sera représenté par

i	4	1
&ri	3	



Passage par référence

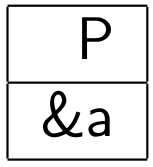
Comme dans le passage par valeur, des emplacements sont prévus dans les tableaux d'activation pour être alloués aux paramètres passés par référence.

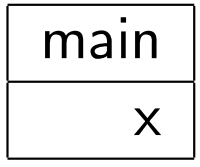
Retenir

Passage par référence *Mais, au contraire du passage par valeur où la valeur du paramètre effectif est recopiée dans le nouvel emplacement,* ***on transfère ici la référence*** *(l'adresse) du paramètre effectif. Le paramètre formel et le paramètre effectif correspondent alors* ***au même emplacement.***

Exemple de Passage par référence

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





Remarquez qu'ici, l'appel à `P(10)` a été supprimé car 10 est une constante (et pas un identificateur de variable). Donc on ne peut pas lui associer une référence....

Exemple de passage par référence

Entrée dans 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) serait une erreur */
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 référence

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) serait une erreur */
 8    x = 5;
 9    P(x);
10    cout << "x = "  << x << endl;
11  }
```

Pile :

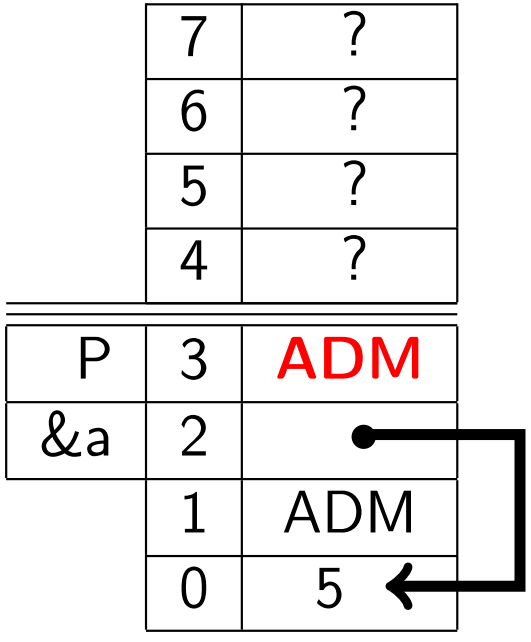
	7	?
	6	?
	5	?
	4	?
	3	?
	2	?
main	1	ADM
x	0	5

Exemple de passage par référence

`main` ligne 9 : appel de `P(x)`, c'est-à-dire `P(&0)`

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

Pile :

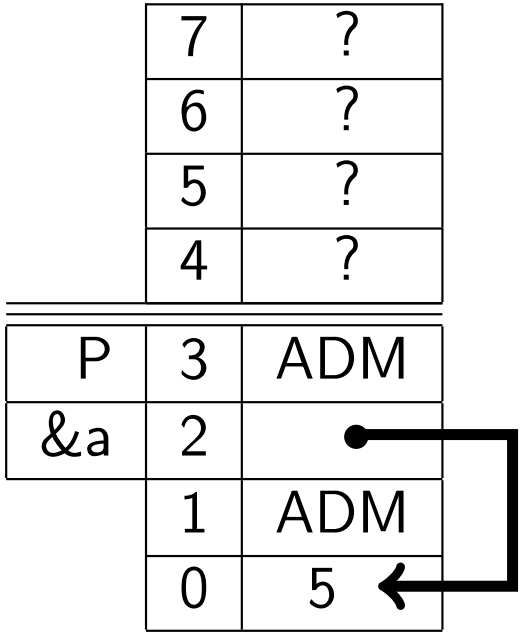


Exemple de passage par référence

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) serait une erreur */
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

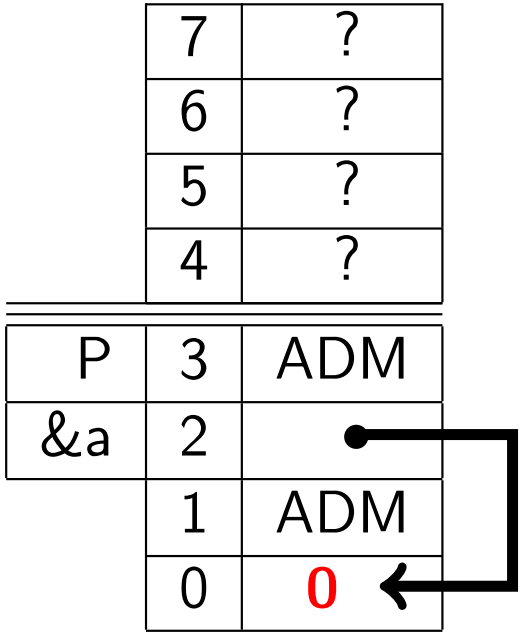


Exemple de passage par référence

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) serait une erreur */
8    x = 5;
9    P(x);
10   cout << "x = "  << x << endl;
11 }
```

Pile :

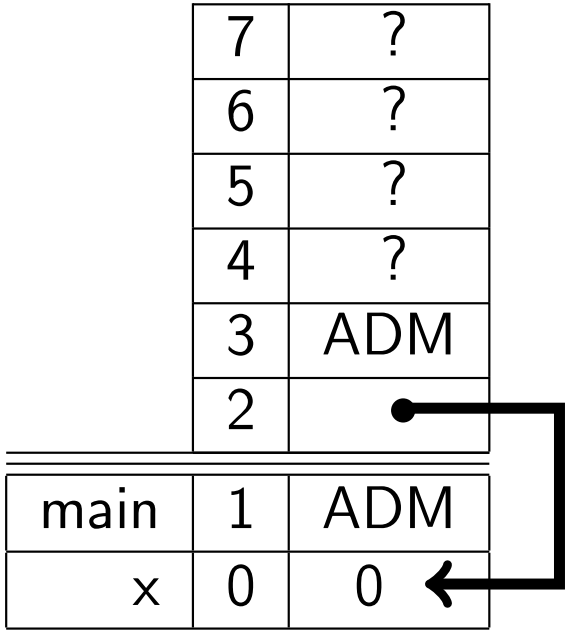


Exemple de passage par référence

Retour au `main` ligne 10 : affichage de «`x = 0`»

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

Pile :



Bilan : passage par valeur / référence

Retenir

	Valeur	*Référence*
Param. réel	*valeur ou variable* *copie*	*variable* *adresse*
modifications	*non reportées*	*reportées*
utilisation	*donnée*	*don/rés ou rés* $*$

Référence vers un objet constant

Dans le cas d'un passage par valeur, la copie peut être coûteuse si l'objet passé est complexe (structure, vecteur...)

Compléments

Dans certains cas, on fait un **passage par référence pour éviter la copie**. Le C++ permet alors d'**interdire la modification** en déclarant l'objet passé par référence constant grâce au mot clé `const`.

```
float moyenne(vector<float> const &notes)
```

Adresse d'une référence

Retenir

En C++, l'adresse d'une référence est l'adresse de la variable référencée.

Le programme :

```
1  void main () {
2    int x = 5;
3    int &y = x;
4    cout << &x << " " << &y << endl;
5  }
```

affiche deux fois la même adresse.

Passage d'une référence

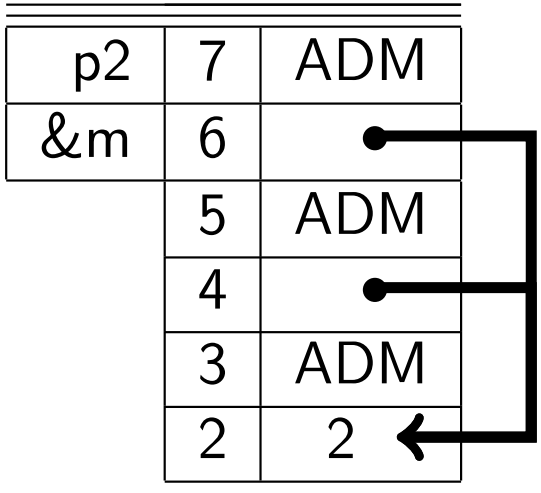
Retenir

En conséquence, on peut passer une ***référence comme paramètre réel d'un passage par référence****.*

doubleref.cpp

```
1  void p2(int &n) {
2      cout << "p2 " << &n << endl;
3  }
4  void p1(int &m) {
5      cout << "p1 " << &m << endl;
6      p2(m);
7  }
8  int main() {
9      int a = 2;
10     cout << "main " << &a << endl;
11     p1(a);
12 }
```

Dans P2 :

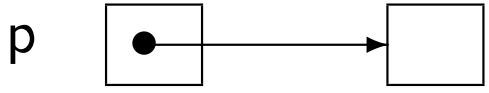


Pointeurs

Définition

Un pointeur est une variable qui contient ***l'adresse d'une autre variable*** *(on dit aussi une référence à une autre variable).*

Représentation graphique :



Le type pointeur est un type de base comme `int`, `float` ou `bool`, mais **sa déclaration dépend du type de la variable qu'il référence** :

Syntaxe

```
type_de_base *nom_de_la_variable;
```

Utilisation d'un pointeur :

Retenir

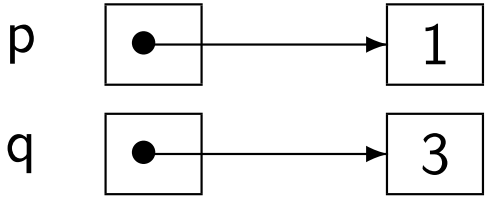
Si p est une variable pointeur :

- *p désigne l'adresse pointée*
- **p désigne la valeur contenue dans l'adresse pointée*

```
int *p, *q;

// Ici : Init de p et q...

*p = 1;
*q = 3;
```



Pointeur null

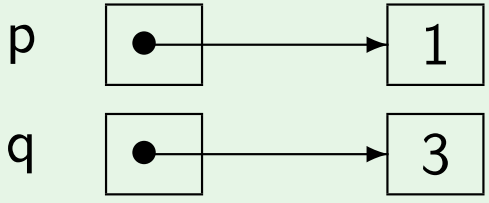
Retenir

Il existe une valeur spéciale d'un pointeur, la valeur `NULL`*, qui indique que le pointeur ne contient l'adresse d'aucune variable.*

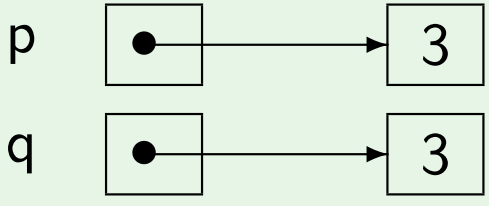
Si un pointeur p vaut NULL, `*p` *n'a pas de sens, et peut avoir un effet imprévisible à l'exécution.*

Pointeurs et affectations ...

Exemple



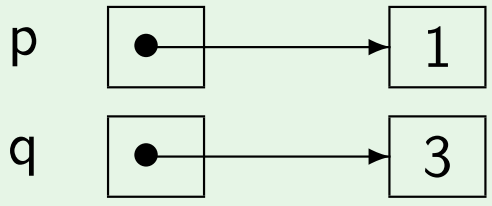
Copie de la valeur de la variable entière pointée par q, à l'adresse de la variable entière pointée par p : `*p = *q;`



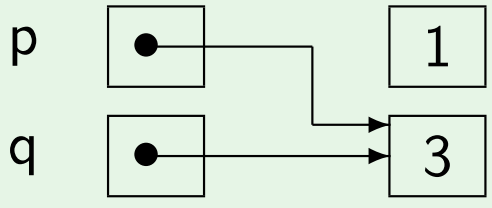
Note : `*p` et `*q` ont maintenant la même valeur.

Pointeurs et Affectations ...

Exemple



Affectation de la valeur du pointeur q au pointeur p `p = q;`



`*p` et `*q` ont aussi la même valeur, mais parce qu'**ils désignent ici la même variable.** On dira que q est un **alias** de p

Lien pointeur et référence

Retenir

*Le **contenu** de l'adresse pointée par le pointeur p est accessible en écriture et en lecture par* `*p`.

*Inversement, **l'adresse** d'une variable x est obtenue par* `&x`.

En C++, les deux suites d'instructions ci-dessous ont le même effet :

```
1  int i;
2
3  int &ri = i; // référence sur la variable i
4  ri = 5;        // affecte à i la valeur 5
5
6  int *p;
7  p=&i;
8  *p=5; //  affecte à i la valeur 5
```

Les paramètres en Donnée-Résultat et Résultat en C

La notion de référence n'existe qu'en C++, pas en C où l'on doit effectuer un passage par adresse. Les paramètres formels de type résultat doivent être des pointeurs.

```
1  void echange (int *x, int *y) {
2      int t;
3      t = *x;
4      *x = *y;
5      *y = t;
6  }
7
8  int a, b;
9  /* appel */
10 echange (&a, &b)
```

Si les paramètres n'étaient pas passés par adresse, la fonction ne produirait aucun effet perceptible !

Passage par référence

Retenir

En C++, il faut comprendre la notion de référence comme un ***pointeur dont on a initialisé l'adresse*** *au moment de sa création, en l'associant à un identificateur existant.*

On ne peut que ***modifier la valeur pointée****, mais* ***pas son adresse*** *: c'est un* ***pointeur constant.***

De ce fait, le langage C++ permet de l'utiliser ensuite comme un raccourci syntaxique ce qui permet d'éviter de devoir expliciter les * devant les variables paramètres.

Récursivité et récurrence

Deux notions très proches :

- mathématiques : récurrence
- informatique : récursivité

De nombreuses définitions mathématiques sont récursives :

Définition (Peano)

- *0 est un entier naturel.*
- *Tout entier n a un successeur unique Sn* $(= n + 1)$ *;*
- *Tout entier sauf* 0 *est le successeur d'un unique entier ;*
- *Pour tout énoncé* $P(n)$ *si* $P(0)$ *est vrai et si pour tout n,* $P(n)$ *implique* $P(Sn)$ *alors l'énoncé* $\forall n : P(n)$ *est vrai.*

Récursivité : définition

Moyen simple et élégant de résoudre certains problèmes.

Définition

*On appelle **récursive** toute fonction ou procédure qui **s'appelle elle-même***

```
unsigned int fact(unsigned int N)
{
   if (N == 0) return 1;
   else        return N*fact(N-1);
}
```

Ça marche ! ! !

Comment ça marche ?

```
Appel à fact(4)
.   4*fact(3) = ?
.   Appel à fact(3)
.   .   3*fact(2) = ?
.   .   Appel à fact(2)
.   .   .   2*fact(1) = ?
.   .   .   Appel à fact(1)
.   .   .   .   1*fact(0) = ?
.   .   .   .   Appel à fact(0)
.   .   .   .   Retour de la valeur 1
.   .   .   .   1*1
.   .   .   Retour de la valeur 1
.   .   .   2*1
.   .   Retour de la valeur 2
.   .   3*2
.   Retour de la valeur 6
.   4*6
Retour de la valeur 24
```

Point terminal

Retenir

Comme dans le cas d'une boucle, il faut un cas d'arrêt où l'on ne fait pas d'appel récursif.

```
récursive(paramètres) {
    if (TEST_D'ARRET) {
        instructions du point d'arrêt
    } else {
        instructions
        récursive(paramètres changés);  // appel récursif
        instructions
    }
}
```

Exemple d'exécution d'une fonction récursive

`main` ligne 8, appel de `fact(5)`

`exrec.cpp`

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

Pile :

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

Exemple d'exécution d'une fonction récursive

Entrée de `fact`

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

	7	?
	6	?
	5	?
fact	4	**ADM**
N	3	**5**
return	2	?
	1	ADM
	0	?

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 appel de `fact(4)` :

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

fact	7	ADM
N	6	4
return	5	?
	4	ADM
	3	5
	2	?
	1	ADM
	0	?

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 appel de `fact(3)` :

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

fact	10	ADM
N	9	3
return	8	?
	7	ADM
	6	4
	5	?
	4	ADM
	3	5
	2	?
	1	ADM
	0	?

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 appel de `fact(2)` :

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

fact	13	ADM
N	12	2
return	11	?
	10	ADM
	9	3
	8	?
	7	ADM
	6	4
	5	?
	4	ADM
	3	5
	2	?
	1	ADM
	0	?

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 appel de `fact(1)` :

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

fact	16	ADM
N	15	1
return	14	?
	13	ADM
	12	2
	11	?
	10	ADM
	9	3
	8	?
	7	ADM
	6	4
	5	?
	4	ADM
	3	5

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 appel de `fact(0)` :

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

fact	19	**ADM**
N	18	**0**
return	17	?
	16	ADM
	15	1
	14	?
	13	ADM
	12	2
	11	?
	10	ADM
	9	3
	8	?
	7	ADM
	6	4

Exemple d'exécution d'une fonction récursive

`fact` ligne 4 retour de 1 :

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

fact	19	ADM
N	18	0
return	17	**1**
	16	ADM
	15	1
	14	?
	13	ADM
	12	2
	11	?
	10	ADM
	9	3
	8	?
	7	ADM
	6	4

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 retour de $1 * \texttt{fact}(0) = 1 * 1 = 1$:

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

	19	ADM
	18	0
	17	1
fact	16	ADM
N	15	1
return	14	**1**
	13	ADM
	12	2
	11	?
	10	ADM
	9	3
	8	?
	7	ADM
	6	4

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 retour de $2 * \texttt{fact}(1) = 2 * 1 = 2$:

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

	16	ADM
	15	1
	14	1
fact	13	ADM
N	12	2
return	11	**2**
	10	ADM
	9	3
	8	?
	7	ADM
	6	4
	5	?
	4	ADM
	3	5

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 retour de $3 * \texttt{fact}(2) = 3 * 2 = 6$:

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

	13	ADM
	12	2
	11	2
fact	10	ADM
N	9	3
return	8	**6**
	7	ADM
	6	4
	5	?
	4	ADM
	3	5
	2	?
	1	ADM
	0	?

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 retour de $4 * \texttt{fact}(3) = 4 * 6 = 24$:

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

	10	ADM
	9	3
	8	6
fact	7	ADM
N	6	4
return	5	**24**
	4	ADM
	3	5
	2	?
	1	ADM
	0	?

Exemple d'exécution d'une fonction récursive

`fact` ligne 5 retour de $5 * \texttt{fact}(4) = 5 * 24 = 120$:

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

	10	ADM
	9	3
	8	6
	7	ADM
	6	4
	5	24
fact	4	ADM
N	3	5
return	2	**120**
	1	ADM
	0	?

Exemple d'exécution d'une fonction récursive

`main` ligne 8 : affichage de 120

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

	10	ADM
	9	3
	8	6
	7	ADM
	6	4
	5	24
	4	ADM
	3	5
	2	**120**
main	1	ADM
return	0	?

Exemple d'exécution d'une fonction récursive

retour du `main` ligne 9

exrec.cpp

```
1  #include <iostream>
2  using namespace std;
3  int fact(int N) {
4      if (N == 0) return 1;
5      else return N*fact(N-1);
6  }
7  int main() {
8      cout << fact(5) << endl;
9      return 0;
10 }
```

	10	ADM
	9	3
	8	6
	7	ADM
	6	4
	5	24
	4	ADM
	3	5
	2	120
main	1	ADM
return	0	**0**

Récursion et Pile

Retenir

Chaque appel récursif consomme de la mémoire dans la pile !

Attention ! *La pile a une taille fixée au départ du programme, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile.*

C'est le fameux **Stack Overflow** !