École Polytechnique

CSC_52064 – Compilation

Jean-Christophe Filliâtre

stratégies d'évaluation
passage des paramètres

1. stratégie d'évaluation et passage des paramètres
   - Java
   - OCaml
   - Python
   - C
   - C++

2. compiler l'appel par valeur et l'appel par référence
   - illustration avec C++

stratégie d'évaluation et passage des paramètres

dans la **déclaration** d'une fonction

```
function f(x1, ..., xn) =
  ...
```

les variables `x1,...,xn` sont appelées **paramètres formels** de `f`

et dans l'**appel** de cette fonction

```
f(e1, ..., en)
```

les expressions `e1,...,en` sont appelées **paramètres effectifs** de `f`

un peu de vocabulaire

dans un langage comprenant des modifications en place, une affectation

```
e1 := e2
```

modifie un emplacement mémoire désigné par l'expression `e1`

l'expression `e1` est limitée à certaines constructions,
car des affectations comme

```
42 := 17
true := false
```

n'ont en général pas de sens

on parle de **valeur gauche** pour désigner les expressions légales à gauche d'une affectation

stratégie d'évaluation

la stratégie d'évaluation d'un langage spécifie l'ordre dans lequel les calculs sont effectués

on peut la définir à l'aide d'une sémantique formelle (cf cours 2)

le compilateur se doit de respecter la stratégie d'évaluation

en particulier, la stratégie d'évaluation **peut** spécifier

- à quel moment les paramètres effectifs d'un appel sont évalués
- l'ordre d'évaluation des opérandes et des paramètres effectifs

certains aspects de l'évaluation peuvent cependant rester **non spécifiés**

cela laisse alors de la latitude au compilateur, notamment pour effectuer des optimisations (par exemple en ordonnant les calculs comme il le souhaite)

on distingue notamment

- l'**évaluation stricte** : les opérandes / paramètres effectifs sont évalués avant l'opération / l'appel

  exemples : C, C++, Java, OCaml, Python

- l'**évaluation paresseuse** : les opérandes / paramètres effectifs ne sont évalués que si nécessaire

  exemples : Haskell, Clojure
  mais aussi les connectives logiques && et || de la plupart des langages

un langage impératif adopte systématiquement une évaluation stricte, pour garantir une séquentialité des effets de bord qui coïncide avec le texte source

par exemple, le code Java

```
int r = 0;
int id(int x) { r += x; return x; }
int f(int x, int y) { return r; }

{ System.out.println(f(id(40), id(2))); }
```

affiche 42 car les deux arguments de `f` ont été évalués

une exception est faite pour les connectives logiques && et || de la plupart des langages, ce qui est bien pratique

```
void insertionSort(int[] a) {
  for (int i = 1; i < a.length; i++) {
    int v = a[i], j = i;
    for (; 0 < j && v < a[j-1]; j--)
      a[j] = a[j-1];
    a[j] = v;
  }
}
```

la non-terminaison est également un effet

ainsi, le code Java

```
int loop() { while (true); return 0; }
int f(int x, int y) { return x+1; }

{ System.out.println(f(41, loop())); }
```

ne termine pas, bien que l'argument `y` n'est pas utilisé

un langage purement applicatif (= sans traits impératifs) peut en revanche adopter la stratégie d'évaluation de son choix, car une expression aura toujours la même valeur (on parle de **transparence référentielle**)

en particulier, il peut faire le choix d'une évaluation paresseuse

le programme Haskell

```
loop () = loop ()
f x y = x
main = putChar (f 'a' (loop ()))
```

termine (après avoir affiché a)

la sémantique précise également le **mode de passage** des paramètres lors d'un appel

on distingue notamment

- l'**appel par valeur** (*call by value)*
- l'**appel par référence** (*call by reference*)
- l'**appel par nom** (*call by name*)
- l'**appel par nécessité** (*call by need*)

(on parle aussi parfois de **passage** par valeur, par référence, etc.)

de **nouvelles** variables représentant les paramètres formels reçoivent les **valeurs** des paramètres effectifs

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // affiche 41
```

les paramètres formels désignent les **mêmes valeurs gauches** que les paramètres effectifs

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // affiche 42
```

les paramètres effectifs sont **substitués** aux paramètres formels, textuellement, et donc évalués seulement si nécessaire

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // affiche 25
  // 1+2 est évalué deux fois
  // 2+2 est évalué deux fois
  // 1/0 n'est jamais évalué
```

les paramètres effectifs ne sont évalués que si nécessaire,
mais **au plus une fois**

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // affiche 25
  // 1+2 est évalué une fois
  // 2+2 est évalué une fois
  // 1/0 n'est jamais évalué
```

quelques mots sur le langage Java

Java est muni d'une stratégie d'évaluation stricte, avec appel **par valeur**

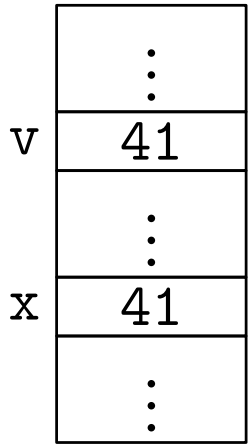
l'ordre d'évaluation est spécifié gauche-droite

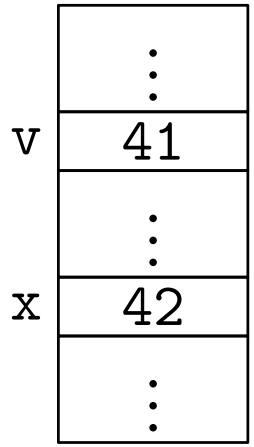
une valeur est

- soit d'un type primitif (booléen, caractère, entier machine, etc.)
- soit un pointeur vers un objet alloué sur le tas

appel par valeur

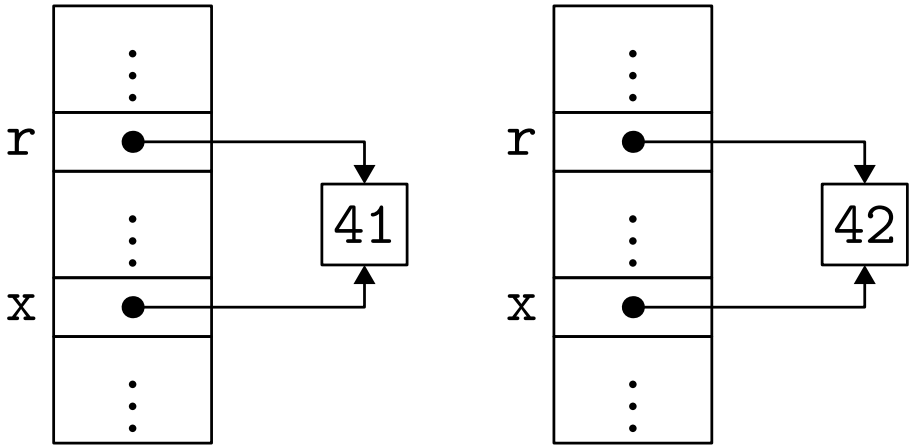
```
void f(int x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut toujours 41
}
```





un objet est alloué sur le tas

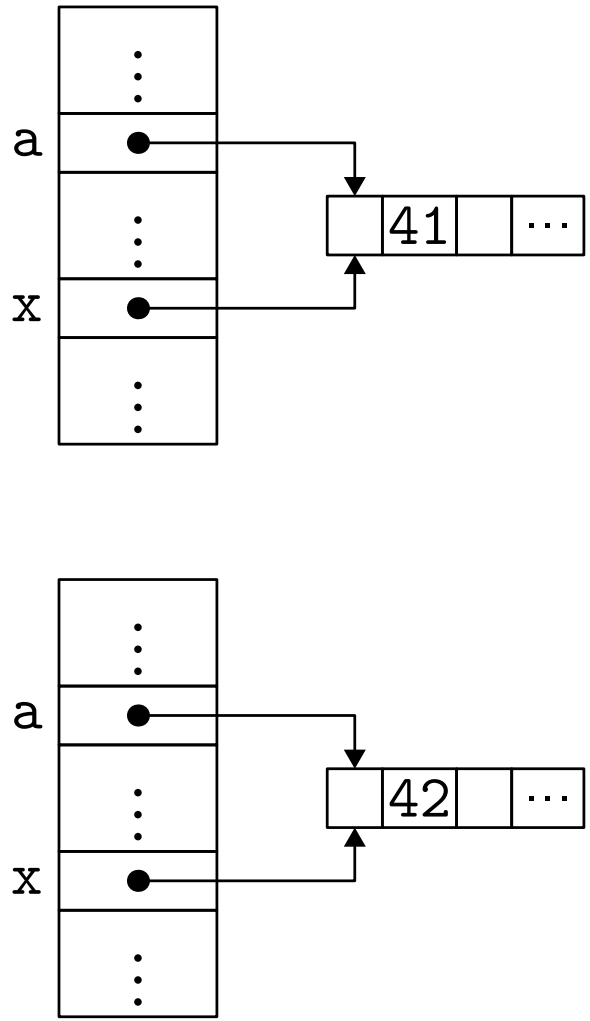
```
class C { int f; }

void incr(C x) {
  x.f += 1;
}
void main () {
  C r = new C();
  r.f = 41;
  incr(r);
  // r.f vaut maintenant 42
}
```



c'est toujours un passage **par valeur**,
d'une valeur qui est un pointeur (implicite) vers un objet

un tableau est un objet comme un autre

```
void incr(int[] x) {
  x[1] += 1;
}
void main () {
  int[] a = new int[17];
  a[1] = 41;
  incr(a);
  // a[1] vaut maintenant 42
}
```



appel par nom en Java

on peut **simuler l'appel par nom** en Java, en remplaçant les arguments par des fonctions ; ainsi, la fonction

```
int f(int x, int y) {
  if (x == 0) return 42; else return y + y;
}
```

peut être réécrite en

```
int f(Supplier<Integer> x, Supplier<Integer> y) {
  if (x.get() == 0)
    return 42;
  else
    return y.get() + y.get();
}
```

et appelée comme ceci

```
int v = f(() -> 0, () -> { throw new Error(); });
```

plus subtilement, on peut aussi **simuler l'appel par nécessité** en Java

```
class Lazy<T> implements Supplier<T> {
  private T cache = null;
  private Supplier<T> f;

  Lazy(Supplier<T> f) { this.f = f; }

  public T get() {
    if (this.cache == null) {
      this.cache = this.f.get();
      this.f = null; // permet au GC de récupérer f
    }
    return this.cache;
  }
}
```

(c'est de la mémoïsation)

et on l'utilise ainsi

```
int w = f(new Lazy<Integer>(() -> 1),
          new Lazy<Integer>(() -> { ...gros calcul... }));
```

quelques mots sur le langage OCaml

OCaml est muni d'une stratégie d'évaluation stricte, avec appel **par valeur**

l'ordre d'évaluation n'est pas spécifié

une valeur est

- soit d'un type primitif (booléen, caractère, entier machine, etc.)
- soit un pointeur vers un bloc mémoire (tableau, enregistrement, constructeur non constant, etc.) alloué sur le tas en général

les valeurs gauches sont les éléments de tableaux

```
a.(2) <- true
```

et les champs mutables d'enregistrements

```
x.age <- 42
```

rappel : une référence est un enregistrement

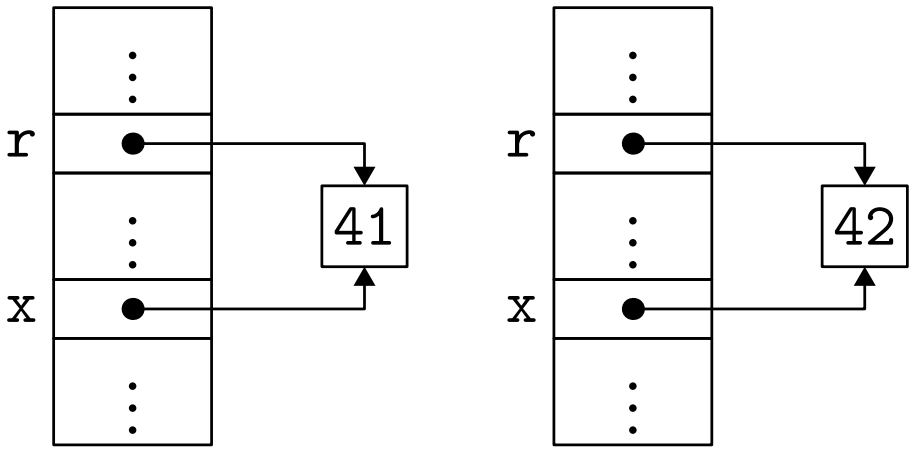
```
type 'a ref = { mutable contents: 'a }
```

et les opérations := et ! sont définies par

```
let (!)  r   = r.contents
let (:=) r v = r.contents <- v
```

une référence est allouée sur le tas

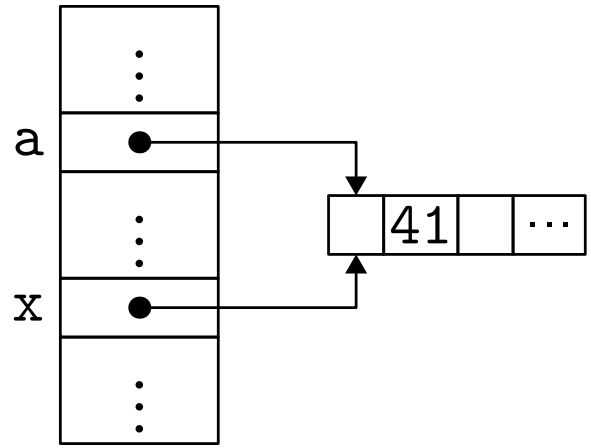
```
let incr x =
  x := !x + 1

let main () =
  let r = ref 41 in
  incr r
  (* !r vaut maintenant 42 *)
```

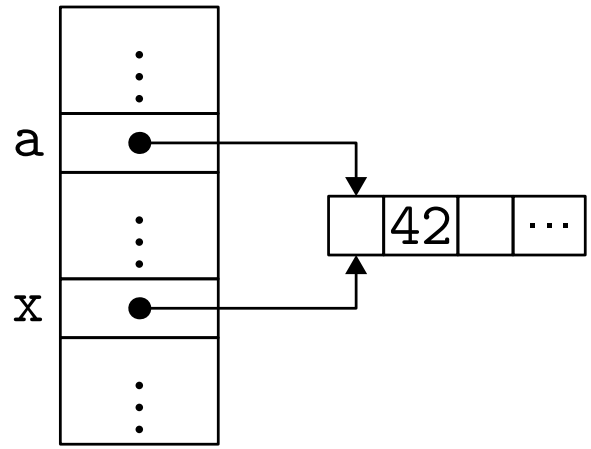


c'est toujours un passage **par valeur**,
d'une valeur qui est un pointeur (implicite) vers une valeur mutable

un tableau est également alloué sur le tas

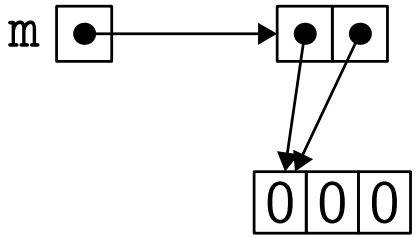
```
let incr x =
  x.(1) <- x.(1) + 1

let main () =
  let a = Array.make 17 0 in
  a.(1) <- 41;
  incr a
  (* a.(1) vaut maintenant 42 *)
```





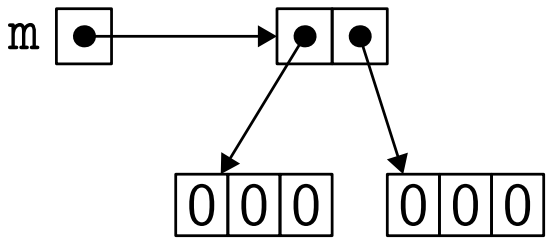
pour construire une matrice, on n'écrit pas

```
let m = Array.make 2 (Array.make 3 0)
```



mais

```
let m = Array.make_matrix 2 3 0
```



on peut **simuler l'appel par nom** en OCaml, en remplaçant les arguments par des fonctions

ainsi, la fonction

```
let f x y =
  if x = 0 then 42 else y + y
```

peut être réécrite en

```
let f x y =
  if x () = 0 then 42 else y () + y ()
```

et appelée comme ceci

```
let v = f (fun () -> 0) (fun () -> failwith "oups")
```

plus subtilement, on peut aussi **simuler l'appel par nécessité** en OCaml

on commence par introduire un type pour représenter les calculs paresseux

```
type 'a value = Value  of 'a
              | Frozen of (unit -> 'a)

type 'a by_need = 'a value ref
```

et une fonction qui évalue un tel calcul si ce n'est pas déjà fait

```
let force l = match !l with
  | Value  v -> v
  | Frozen f -> let v = f () in l := Value v; v
```

(c'est de la mémoïsation)

appel par nécessité en OCaml

on définit alors la fonction `f` comme ceci

```
let f x y =
  if force x = 0 then 42 else force y + force y
```

et on l'utilise ainsi

```
let v = f (ref (Frozen (fun () -> 1)))
          (ref (Frozen (fun () -> ...gros calcul...)))
```

note : la construction `lazy` d'OCaml fait quelque chose de semblable (un peu plus subtilement)

quelques mots sur le langage Python

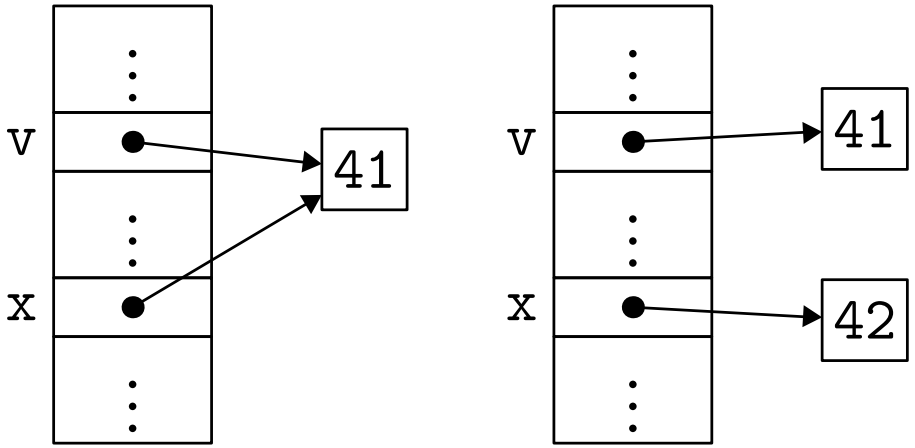
Python est muni d'une stratégie d'évaluation stricte, avec appel **par valeur**

l'ordre d'évaluation est spécifié gauche-droite
(mais droite-gauche pour une affectation)

une valeur est un pointeur vers un objet alloué sur le tas

un entier est un objet immuable

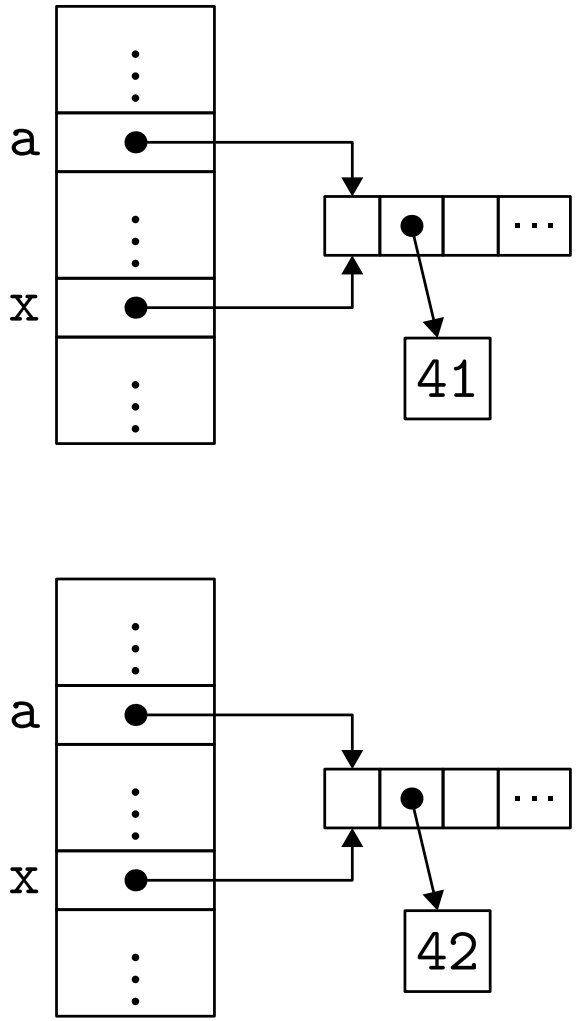
```
def f(x):
    x += 1

v = 41
f(v)
print(v) # affiche 41
```



c'est toujours un passage **par valeur**,
d'une valeur qui est un pointeur (implicite) vers un objet

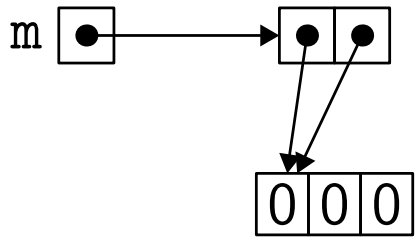
un tableau est un objet mutable

```
def incr(x):
    x[1] += 1

a = [0] * 17
a[1] = 41
incr(a)
# a[1] vaut maintenant 42
```



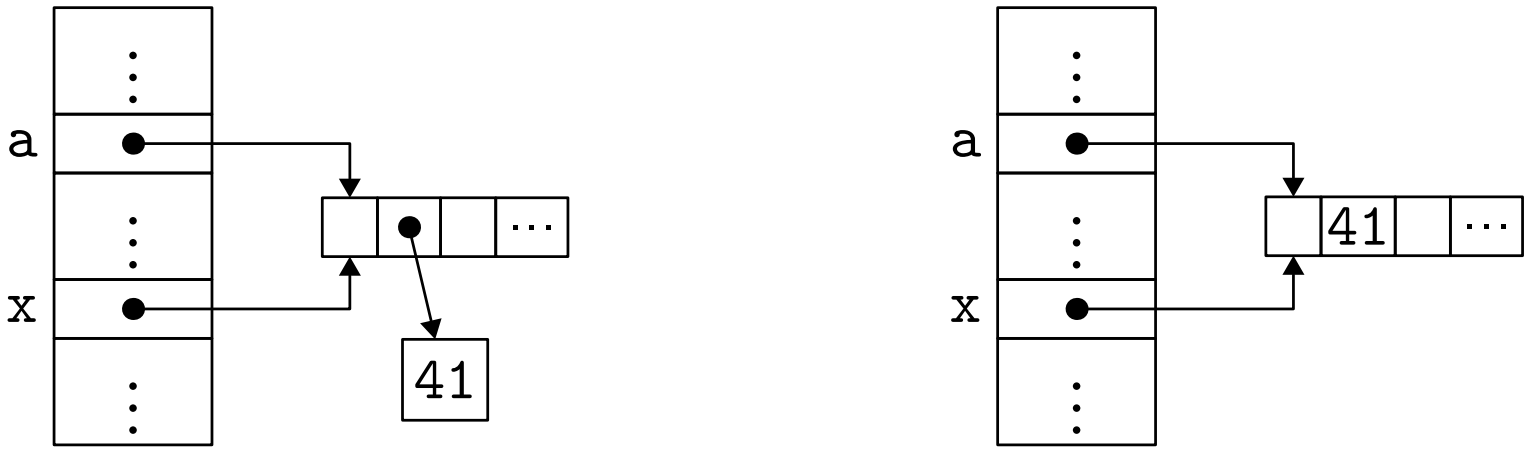
en Python, on ne construit pas non plus une matrice en faisant

```
m = [[0] * 3] * 2
```



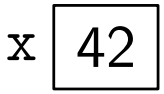
les entiers étant des objets **immuables**, on peut faire abstraction de leur représentation comme des objets

dit autrement, on peut identifier deux représentations comme



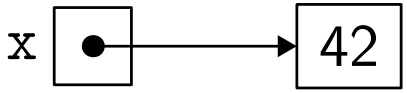
les modèles d'exécution de Java, d'OCaml et de Python sont **très semblables**

même si leurs langages de surface sont très différents



OCaml : variable immuable

Java : variable mutable



OCaml : variable immuable + contenu mutable

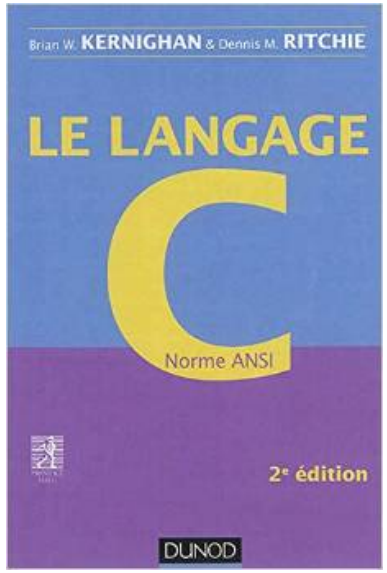
Python : variable mutable + contenu immuable

quelques mots sur le langage C

le langage C est un langage impératif relativement bas niveau, notamment parce que la notion de pointeur, et d'arithmétique de pointeur, y est explicite

on peut le considérer inversement comme un assembleur de haut niveau

un ouvrage toujours d'actualité :
Le langage C
de Brian Kernighan et Dennis Ritchie



le langage C est muni d'une stratégie d'évaluation stricte,
avec appel **par valeur**

l'ordre d'évaluation n'est pas spécifié

les types du C

- on trouve des types de base tels que `char`, `int`, `float`, etc.

- un type $\tau*$ des pointeurs vers des valeurs de type τ

  si p est un pointeur de type $\tau*$, alors $*p$ désigne la valeur pointée par p, de type τ

  si e est une valeur gauche de type τ, alors $\&e$ est un pointeur sur l'emplacement mémoire correspondant, de type $\tau*$

- des enregistrements, appelés *structures*, tels que

```
struct L { int head; struct L *next; };
```

  si `e` a le type `struct L`, on note `e.head` l'accès au champ

valeurs gauches du C

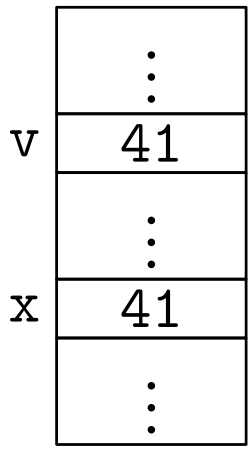
en C, une valeur gauche est de la forme

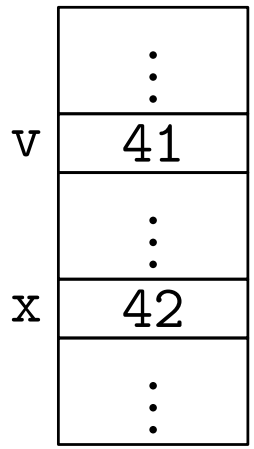
- `x`, une variable
- `*e`, le déréférencement d'un pointeur
- `e.x`, l'accès à un champ de structure,
  si `e` est elle-même une valeur gauche

- `t[e]`, qui n'est autre que `*(t+e)`
- `e->x`, qui n'est autre que `(*e).x`

appel par valeur

```
void f(int x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut toujours 41
}
```

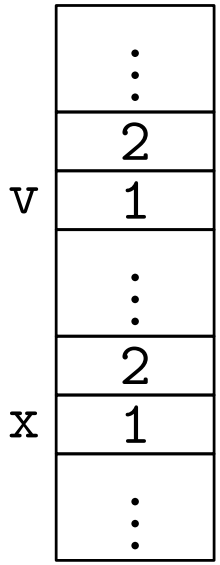


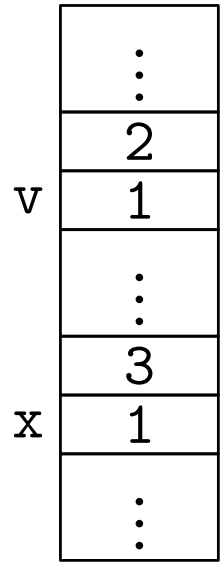


l'appel par valeur implique que les **structures sont copiées** lorsqu'elles sont passées en paramètres ou renvoyées

les structures sont également copiées lors des affectations de structures, *i.e.* des affectations de la forme `x = y`, où `x` et `y` ont le type `struct S`

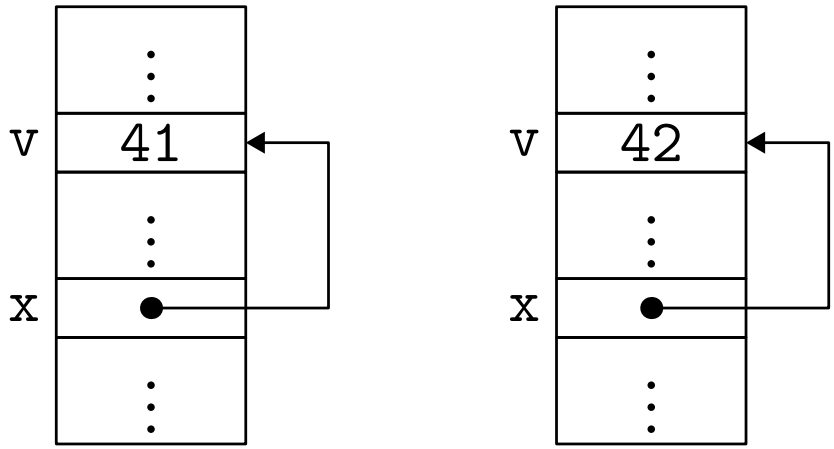
```
struct S { int a; int b; };

void f(struct S x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b vaut toujours 2
}
```





on peut **simuler** un passage par référence en passant un pointeur explicite

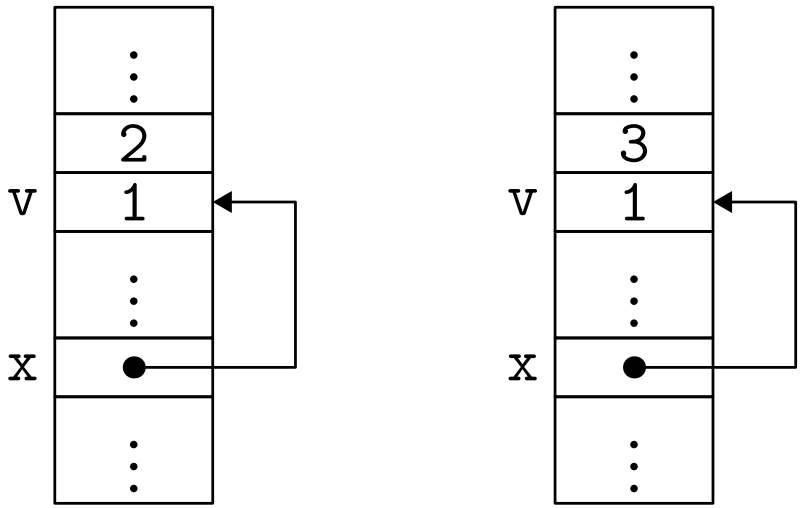
```
void incr(int *x) {
  *x = *x + 1;
}

int main() {
  int v = 41;
  incr(&v);
  // v vaut maintenant 42
}
```



mais ce n'est qu'un passage de pointeur **par valeur**

pour éviter le coût des copies, on passe des pointeurs sur les structures le plus souvent

```
struct S { int a; int b; };

void f(struct S *x) {
  x->b = x->b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(&v);
  // v.b vaut maintenant 3
}
```



la manipulation explicite de pointeurs peut être dangereuse

```
int* p() {
  int x;
  ...
  return &x;
}
```

cette fonction renvoie un pointeur qui correspond à un emplacement sur la pile qui vient justement de disparaître (à savoir le tableau d'activation de p), et qui sera très probablement réutilisé rapidement par un autre tableau d'activation

on parle de référence fantôme (*dangling reference*)

la notation `t[i]` n'est que du sucre syntaxique pour `*(t+i)` où

- `t` désigne un pointeur sur le début d'une zone contenant 10 entiers
- `+` désigne une opération d'*arithmétique de pointeur* (qui consiste à ajouter à `t` la quantité `4i` pour un tableau d'entiers 32 bits)

le premier élément du tableau est donc `t[0]` c'est-à-dire `*t`

un tableau peut être alloué sur la pile, comme ceci

```
int t[10];
```

et il sera désalloué à la sortie de la fonction

ou sur le tas, comme ceci

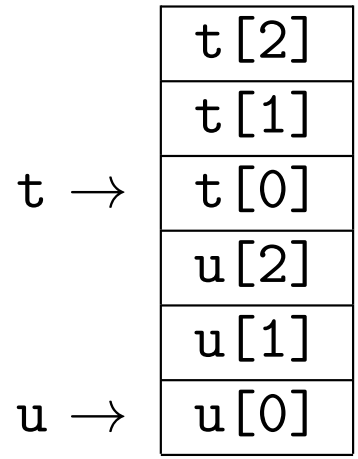
```
int *t = malloc(10 * sizeof(int));
```

et il faudra le désallouer avec `free` (cf cours 9)

on ne peut affecter des tableaux, seulement des pointeurs

ainsi, on ne peut pas écrire

```
void p() {
  int t[3];
  int u[3];
  t = u;     // <- erreur
}
```



car `t` et `u` sont des tableaux (alloués sur la pile) et l'affectation de tableaux n'est pas autorisée

quand on passe un tableau en paramètre, on ne fait que passer le pointeur (par valeur, toujours)

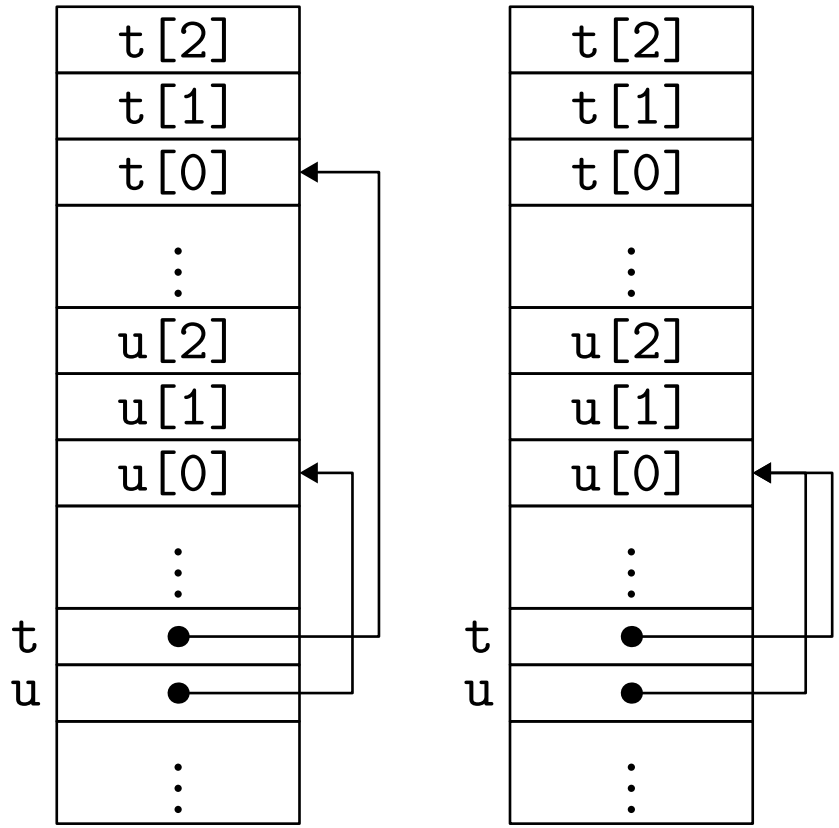
on peut donc écrire

```
void q(int t[3], int u[3]) {
  t = u;
}
```

car c'est exactement la même chose que

```
void q(int *t, int *u) {
  t = u;
}
```

et l'affectation de pointeurs est autorisée



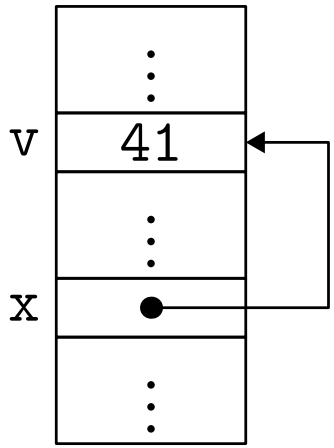
quelques mots sur le langage C++

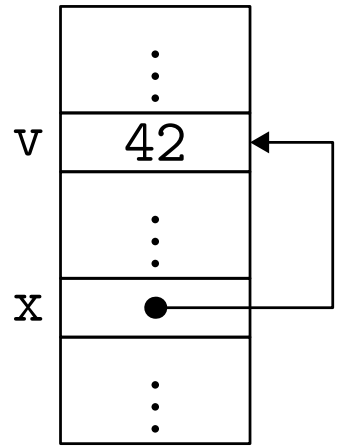
en C++, on trouve (entre autres) les types et constructions du C,
avec une stratégie d'évaluation stricte

le mode de passage est **par valeur** par défaut

mais on trouve aussi un passage **par référence**
indiqué par le symbole & au niveau de l'argument formel

```
void f(int &x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut maintenant 42
}
```



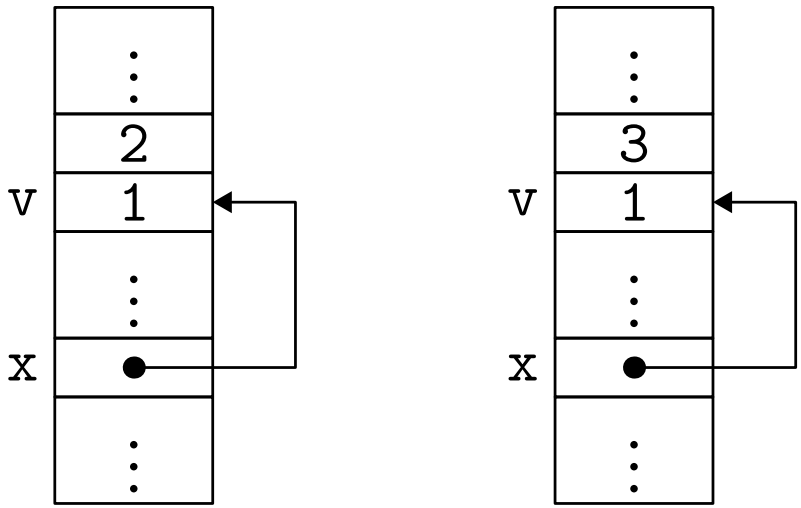


en particulier, c'est le compilateur qui

- a pris l'adresse de `v` au moment de l'appel
- a déréférencé l'adresse `x` dans la fonction `f`

on peut passer une structure par référence

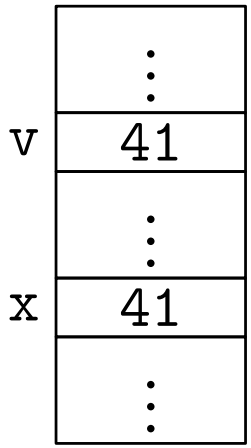
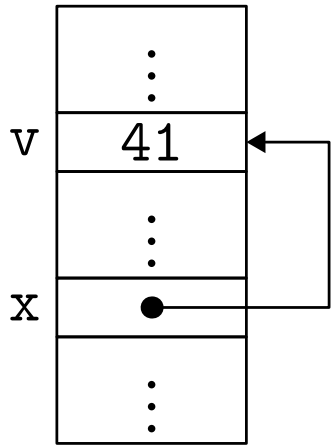
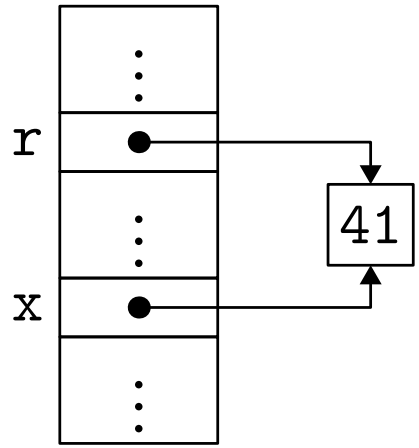
```
struct S { int a; int b; };

void f(struct S &x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b vaut maintenant 3
}
```



on peut passer un pointeur par référence

par exemple pour ajouter un élément dans un arbre

```
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
  if       (t == NULL ) t = create(NULL, x, NULL);
  else if (x < t->elt) add(t->left,  x);
  else if (x > t->elt) add(t->right, x);
}
```

			
Java	entier par valeur	—	pointeur par valeur (objet)
OCaml	entier par valeur	—	pointeur par valeur (`ref`, tableau, etc.)
Python	—	—	pointeur par valeur (objet)
C	entier par valeur	pointeur par valeur	pointeur par valeur
C++	entier par valeur	pointeur par valeur entier par référence	pointeur par valeur ou par référence

compilation du passage par valeur et par référence

considérons la compilation d'un micro fragment de C++ avec

- des entiers
- des fonctions (mais qui ne renvoient rien)
- du passage par valeur et par référence

on considère le fragment suivant

$$
\begin{array}{rcl}
E & \to & n \\
& | & x \\
& | & E + E \mid E - E \\
& | & E * E \mid E \ / \ E \\
& | & - E
\end{array}
\qquad
\begin{array}{rcl}
C & \to & E == E \mid E \ != E \\
& | & E < E \mid E <= E \mid E > E \mid E >= E \\
& | & C \ \&\& \ C \\
& | & C \ || \ C \\
& | & ! \ C
\end{array}
$$

$$
\begin{array}{rcl}
S & \to & x = E; \\
& | & \texttt{if} \ (\ C \ ) \ S \\
& | & \texttt{if} \ (\ C \ ) \ S \ \texttt{else} \ S \\
& | & \texttt{while} \ (\ C \ ) \ S \\
& | & f(E, \ldots, E); \\
& | & \texttt{printf("\%d\textbackslash n", } \ E); \\
& | & \texttt{int } x, \ldots, x; \\
& | & B
\end{array}
\qquad
\begin{array}{rcl}
B & \to & \{ \ S \ldots \ S \ \} \\
F & \to & \texttt{void } f(X, \ldots, X) \ B \\
X & \to & \texttt{int } x \\
& | & \texttt{int \&}x \\
P & \to & F \ldots F \\
& & \texttt{int main() } B
\end{array}
$$

```
void fib(int n, int &r) {
  if (n <= 1)
    r = n;
  else {
    int tmp;
    fib(n - 2, tmp);
    fib(n - 1, r);
    r = r + tmp;
  }
}

int main() {
  int f;
  fib(10, f);
  printf("%d\n", f);
}
```

la **portée** définit les portions du programme où une variable est visible

ici, si le corps d'une fonction f mentionne une variable x alors

- soit x est un paramètre de f
- soit x est déclarée plus haut dans un bloc englobant (y compris le bloc courant)

par ailleurs, une variable peut en cacher une autre

exemple

```
void f(int n) {
  printf("%d\n", n);   // affiche 34
  if (n > 0) {
    int n; n = 89;
    printf("%d\n", n); // affiche 89
  }
  if (n > 21) {
    printf("%d\n", n); // affiche 34
    int n; n = 55;
    printf("%d\n", n); // affiche 55
  }
  printf("%d\n", n);   // affiche 34
}

int main() {
  f(34);
}
```

ici la portée ne dépend que du texte source (on parle de **portée lexicale**)
et on peut la réaliser avant ou pendant le typage

la syntaxe abstraite conserve une trace de cette analyse,
en identifiant chaque variable de façon unique

avant

arbres de syntaxe abstraite issus de l'analyse syntaxique

```
type pint_expr =
  | PEconst of int
  | PEvar   of string
  | ...
type pstmt =
  | PSvars  of string list
  | PSblock of pstmt list
  | ...
```

pour l'instant, les variables sont des chaînes de caractères

après

arbres de syntaxe abstraite après le typage

```
type int_expr =
  | Econst of int
  | Evar   of ident
  | ...

type func = {
  locals: ident list;
  ...
```

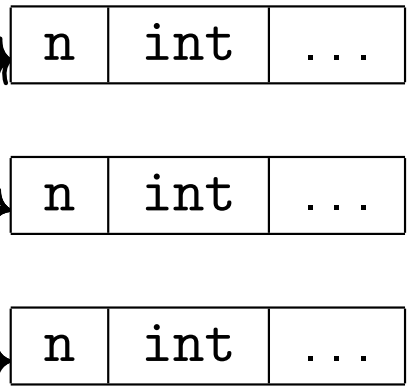
maintenant `ident` est un **identifiant** : entier, nom unique, enregistrement, etc.

on a maintenant un arbre de syntaxe abstraite qui correspond à

```
void f(int n0) {
  printf("%d\n", n0);
  if (n0 > 0) {
    int n1; n1 = 89;
    printf("%d\n", n1);
  }
  if (n0 > 21) {
    printf("%d\n", n0);
    int n2; n2 = 55;
    printf("%d\n", n2);
  }
  printf("%d\n", n0);
}
```

ou plus précisément comme ceci :

```
void f(int ●) {
  printf("%d\n", ●);
  if (● > 0) {
    int ●; ● = 89;
    printf("%d\n", ●);
  }
  if (● > 21) {
    printf("%d\n", n0);
    int ●; ● = 55;
    printf("%d\n", ●);
  }
  printf("%d\n", ●);
}
```



il existe des langages où la portée est **dynamique** i.e. dépend de l'exécution du programme

exemple : `bash`

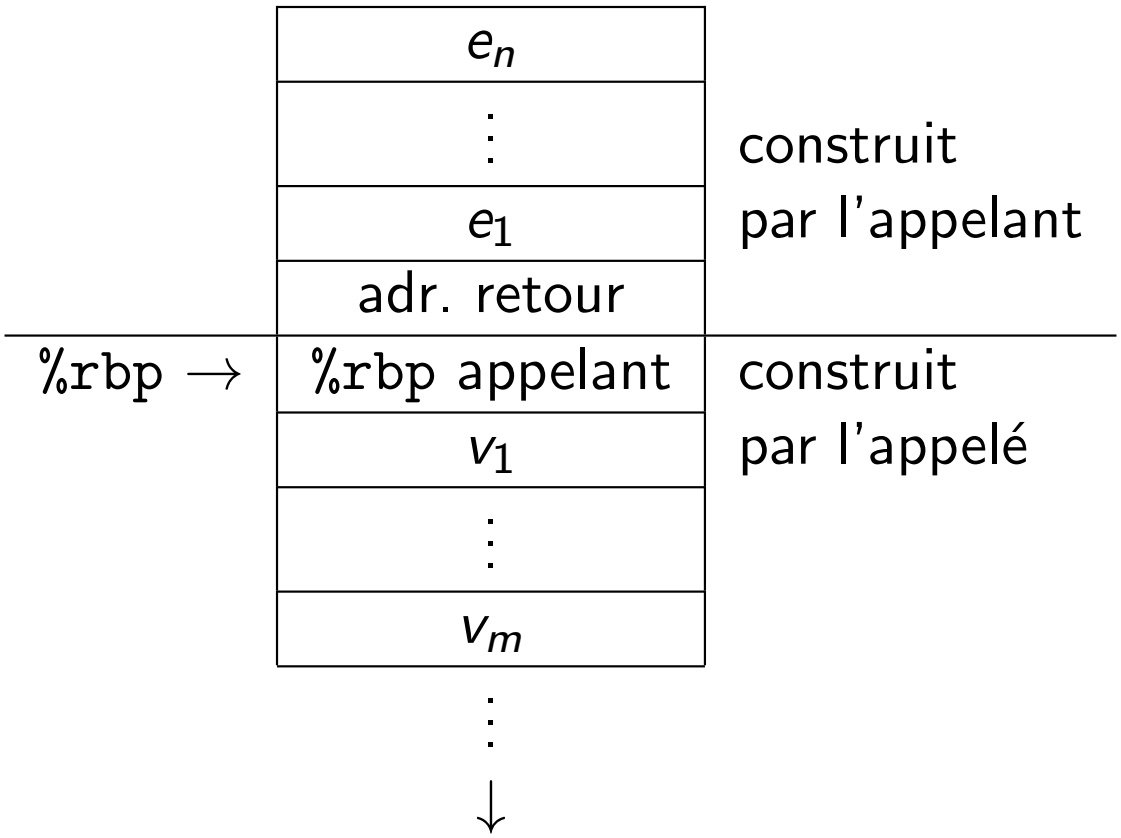
il faut choisir un emplacement mémoire pour chaque variable et être capable de **calculer** cet emplacement à l'exécution

ici les variables vont toutes être stockées sur la pile

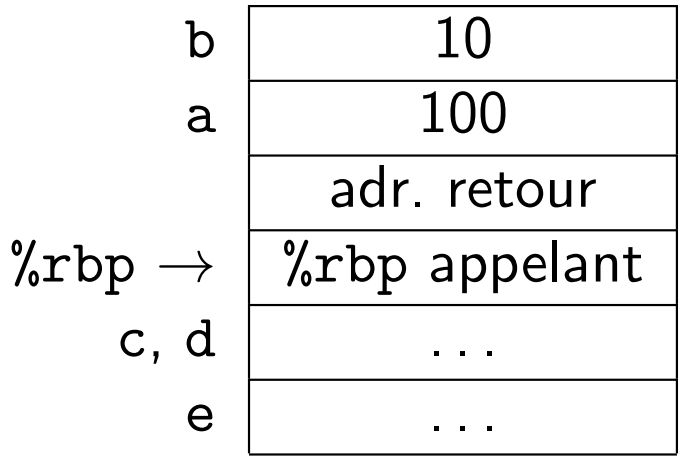
à chaque fonction en cours d'exécution correspond une portion de la pile, appelée **tableau d'activation** (cf cours 1), qui contient notamment

- ses paramètres effectifs
- l'adresse de retour
- ses variables locales

tableau d'activation correspondant à un appel $f(e_1, \ldots, e_n)$ d'une fonction f avec n paramètres



```
void g(int a, int b) {
  if (...) {
    int c;
    ...
  }
  if (...) {
    int d;
    ...
    int e;
    ...
  }
}

int main() {
  g(100, 10);
}
```



rôle de %rbp

positionner ainsi le registre %rbp permet de retrouver facilement l'emplacement d'une variable (par ex. %rbp + 16 ou %rbp − 8)

en effet, le sommet de pile peut bouger si

- on y stocke des calculs intermédiaires
- on est en train de préparer un appel de fonction

pour chaque variable, le compilateur détermine une position dans la pile

par exemple dans le type `ident`

```
type ident = { ofs: int; ... }
```

- pour les paramètres, ce sont $+16$, $+24$, etc.
- pour les variables locales, ce sont -8, -16, etc.,
  avec souvent plusieurs solutions possibles, certaines plus économes

détaillons maintenant la production d'assembleur x86-64 pour micro C++

on se limite pour commencer au **passage par valeur**

on suit un schéma de compilation simpliste, utilisant la pile pour stocker les résultats intermédiaires (on verra comment utiliser efficacement les registres aux cours 10–11)

on note $C(e)$ le code assembleur pour la compilation d'une expression arithmétique e

principe : à l'issue de l'exécution de $C(e)$,

- la valeur de l'expression e se trouve dans le registre `%rdi` (choix arbitraire)
- le pointeur de pile est inchangé
- les registres *caller-saved* peuvent être modifiés

constantes

$$C(n) \stackrel{\text{def}}{=} \texttt{movq } n \texttt{ \%rdi}$$

opérations

$$C(e_1 + e_2) \stackrel{\text{def}}{=} \begin{array}{l} C(e_1) \\ \texttt{pushq \%rdi} \\ C(e_2) \\ \texttt{popq \%rsi} \\ \texttt{addq \%rsi, \%rdi} \end{array}$$

bien entendu, c'est extrêmement naïf ; le code pour 1+2 est

```
        movq  $1, %rdi
        pushq %rdi
        movq  $2, %rdi
        popq  %rsi
        addq  %rsi, %rdi
```

alors même que l'on dispose de 16 registres

pour une **variable**, on utilise l'adressage indirect, car la position par rapport à `%rbp` est une constante connue du compilateur

$$C(x) \stackrel{\text{def}}{=} \texttt{movq}\ \textit{ofs}\texttt{(\%rbp), \%rdi}$$

(rappel : on se limite pour l'instant au passage par valeur)

les expressions booléennes sont compilées de manière très analogue

$$C(e_1 = e_2) \stackrel{\text{def}}{=} \begin{array}{l} C(e_1) \\ \texttt{pushq \%rdi} \\ C(e_2) \\ \texttt{popq \%rsi} \\ \texttt{cmpq \%rdi, \%rsi} \\ \texttt{sete \%dil} \\ \texttt{movzbq \%dil, \%rdi} \end{array}$$

attention : les opérateurs `&&` et `||` doivent être évalués paresseusement *i.e.* e_2 n'est pas évaluée dans e_1 `&&` e_2 (resp. e_1 `||` e_2) si e_1 vaut `false` (resp. `true`)

une instruction s est compilée par un morceau d'assembleur $C(s)$

principe : après l'exécution de $C(s)$,

- le pointeur de pile est inchangé
- les registres *caller-saved* peuvent être modifiés

$C(\texttt{print}(e)) \stackrel{\text{def}}{=}$ $C(e)$
`call print_int`

```
print_int:
        pushq %rbp
        movq  %rsp, %rbp
        andq  $-16, %rsp  # alignement de la pile
        movq  %rdi, %rsi
        movq  $.Sprint_int, %rdi
        movq  $0, %rax
        call  printf
        movq  %rbp, %rsp
        popq  %rbp
        ret
.data
.Sprint_int:
        .string "%d\n"
```

pour un appel à une fonction `f`, il faut

1. empiler les arguments
2. appeler le code situé à l'étiquette `f`
3. dépiler les arguments

$$
\begin{aligned}
C(f(e_1, \ldots, e_n) \stackrel{\text{def}}{=} \quad & C(e_n) \\
& \texttt{pushq \%rdi} \\
& \vdots \\
& C(e_1) \\
& \texttt{pushq \%rdi} \\
& \texttt{call f} \\
& \texttt{addq \$8}n\texttt{, \%rsp}
\end{aligned}
$$

reste l'**affectation** `x = e;`

le membre gauche est ici réduit à une variable `x`
et on sait où cette variable est stockée sur la pile

$$C(x = e) \stackrel{\text{def}}{=} \begin{array}{l} C(e) \\ \texttt{movq \%rdi, } \mathit{ofs}\texttt{(\%rbp)} \end{array}$$

pour l'instant, on a passé les paramètres **par valeur**

i.e. le paramètre formel est une **nouvelle variable** qui prend comme valeur initiale celle du paramètre effectif

en C++, le qualificatif `&` permet de spécifier un passage **par référence**

dans ce cas, le paramètre formel désigne la **même variable** que le paramètre effectif, qui doit donc être une variable (une valeur gauche, de manière plus générale)

exemple

```
void fib(int n, int &r) {
  if (n <= 1)
    r = n;
  else {
    int tmp;
    fib(n - 2, tmp);
    fib(n - 1, r);
    r = r + tmp;
  }
}

int main() {
  int f;
  fib(10, f);          // modifie la valeur de f
  printf("%d\n", f); // affiche 55
}
```

pour prendre en compte le passage par référence, on étend encore le type `ident` pour indiquer s'il s'agit d'une variable passée par référence

```
type ident = { ofs: int; byref: bool; ... }
```

(vaut `false` pour une variable locale)

dans un appel tel que `f(e)` le paramètre effectif `e` n'est plus typé ni compilé de la même manière selon qu'il s'agit d'un paramètre passé par valeur ou par référence

lorsque le paramètre est passé par référence, le typage va donc

1. vérifier qu'il s'agit bien d'une valeur gauche (ici une variable)
2. indiquer qu'elle doit être passée par référence

une façon de procéder consiste à ajouter une construction de « calcul de valeur gauche » dans la syntaxe des expressions

```
type int_expr =
  ...
  | Eaddr of ident
```

et à remplacer, le cas échéant, le paramètre effectif `e` par `Eaddr e`

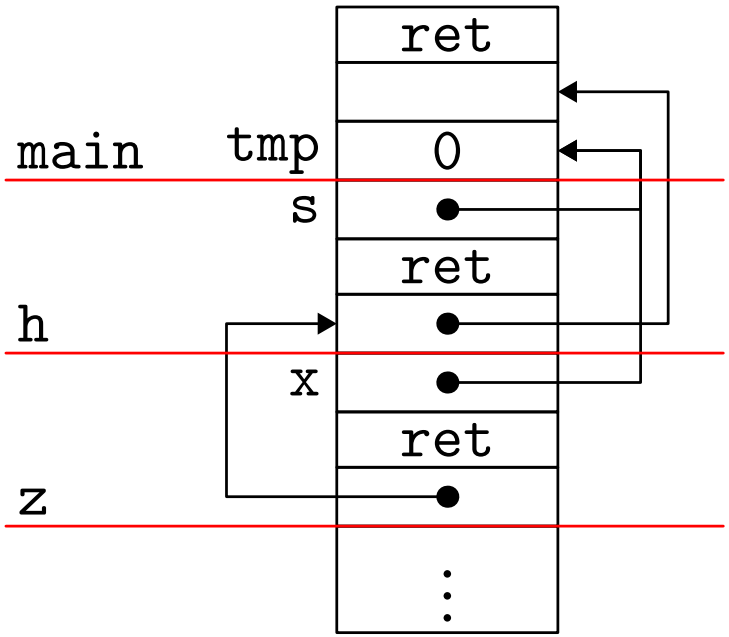
note : c'est l'opérateur `&` de C++, qui n'est pas dans notre fragment

il faut ajouter le code correspondant dans `int_expr` :

$$C(\&x) \stackrel{\text{def}}{=} \texttt{leaq } \textit{ofs}\texttt{(\%rbp), \%rdi}$$
$$\texttt{movq (\%rdi), \%rdi} \qquad \texttt{si } x\texttt{.byref}$$

note : le cas `br = true` correspond au cas d'une variable elle-même passée par référence

```
void z(int &x) { x = 0; }
void h(int &s) { z(s); while (s < 100) s = 2*s+1; }
int main() { int tmp; h(tmp); printf("%d\n", tmp); }
```



il faut aussi modifier le calcul des valeurs droites :

$$C(x) \stackrel{\text{def}}{=} \begin{array}{ll} \texttt{movq } \textit{ofs}\texttt{(\%rbp), \%rdi} & \\ \texttt{movq (\%rdi), \%rdi} & \texttt{si } x\texttt{.byref} \end{array}$$

ainsi que celui de l'affectation :

$$C(x = e) \stackrel{\text{def}}{=}$$

```
C(e)
movq ofs(%rbp), %rsi      si x.byref
leaq ofs(%rbp), %rsi      sinon
movq %rdi, (%rsi)
```

en revanche, il n'y a rien à modifier dans l'appel (grâce à la nouvelle construction `Eaddr`)

il reste à compiler les déclarations des fonctions

```
void f(x1, ..., xn) {
  // variables locales y1,...,ym
  corps
}
```

on calcule

$$fs = \max_{y_i} |y_i.\texttt{ofs}|$$

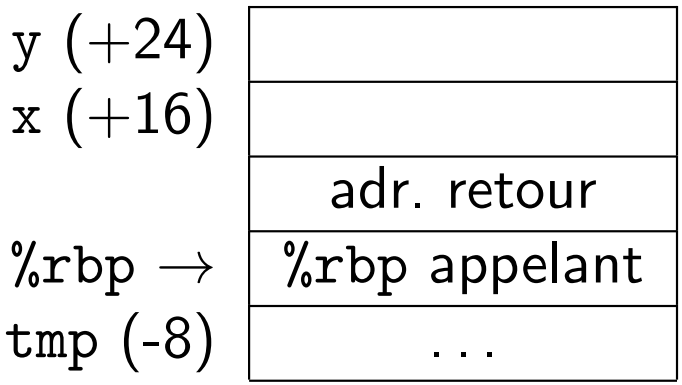
puis

```
f:      pushq %rbp              # sauver %rbp
        movq  %rsp, %rbp        # et le définir
        subq  $fs, %rsp         # allouer fs octets
```

$C(\texttt{corps})$

```
        movq  %rbp, %rsp        # désallouer
        popq  %rbp              # restaurer %rbp
        ret                     # retour
```

```
void swap(int &x, int &y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
```




```
swap:   pushq %rbp
        movq %rsp, %rbp
        subq $8, %rsp
        movq 16(%rbp), %rdi
        movq 0(%rdi), %rdi
        leaq -8(%rbp), %rsi
        movq %rdi, 0(%rsi)
        movq 24(%rbp), %rdi
        movq 0(%rdi), %rdi
        movq 16(%rbp), %rsi
        movq %rdi, 0(%rsi)
        movq -8(%rbp), %rdi
        movq 24(%rbp), %rsi
        movq %rdi, 0(%rsi)
        movq %rbp, %rsp
        popq %rbp
        ret
```

la suite

- TD 5
  - typage Mini Go (suite)
- prochain cours
  - langages OO et fonctionnels