Programmation impérative / procédurale

- Style de programmation offert par la plupart des langages de programmation (machine, assembleur, FORTRAN, Algol, SIMULA, C/C++, Pascal, Ada, Java, Lisp, Scheme, ML, ...)

- Basé sur le modèle d'une **machine**; le programme indique la **séquence d'opérations** que la machine doit effectuer pour faire le calcul

- Déf: un **point de contrôle** c'est un emplacement du programme entre deux opérations

```
begin  ! programme SIMULA ;
  integer x, y;
1
  x := inint;
2
  y := inint;
3
  x := x+y*y;
4
  outint (x, 0);
5
end;
```

- Note: dépendant du point de vue certains énoncés du programme peuvent donner lieu à plus d'un point de contrôle si ils font plus qu'une opération (p.e. `x:=inint;`)

Exécution d'un programme

- Déf: l'**état** de l'exécution d'un programme (si on suppose une seule entrée et sortie et pas d'appel de procédure) =

  1. **point de contrôle** où se trouve l'exécution
  2. **contenu** de toutes les variables et structures de données
  3. partie de l'**entrée** qui reste à lire et **sortie** produite jusqu'à date

```
begin  ! programme SIMULA ;
  integer x, y;
1
  x := inint;
2
  y := inint;
3
  x := x+y*y;
4
  outint (x, 0);
5
end;
```

EXÉCUTION

pt.	variables	entrée	sortie
1	x=? y=?	1 2	
2	x=1 y=?	2	
3	x=1 y=2		
4	x=5 y=2		
5	x=5 y=2		5

- L'état change après l'exécution de chaque opération (revenir à un état précédent => boucle infinie)

- L'**ordre des opérations** est important

```
y := inint; ! ce programme n'est pas équivalent ;
x := inint;
x := x+y*y;
outint (x, 0);
```


Énoncés de contrôle

- Normalement, après l'exécution d'un énoncé, le point d'exécution se trouve juste avant le prochain énoncé (i.e. exécution **séquentielle**)

- Pour obtenir un ordre d'exécution autre que séquentiel, il faut utiliser des **énoncés de contrôle**

- Exemple:

```
1
if x < 0 then
  2
  x := 0
else
  3
  x := x*2;
4
```

L'ordre d'exécution sera 1-2-4 ou 1-3-4 dépendant de la valeur de la variable `x`

Énoncé de contrôle "`goto`"

- Énoncé de contrôle le plus primitif; correspond directement avec les instructions de branchement en langage machine; offert par FORTRAN, SIMULA, C, Pascal, ...

- Change le point de contrôle de façon arbitraire

- Destination du `goto` indiqué à l'aide d'une **étiquette**

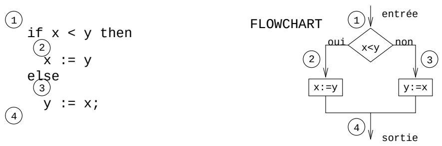
```
       x := 1;  ! programme SIMULA ;
       goto fin;
debut: x := x*2;
fin:   if x<10 then goto debut;
       outint (x, 0);
```

- Problème: difficile de comprendre un programme à base de `goto` car la séquence d'opérations obtenue n'est pas évidente

- L'utilisation abusive de `goto` mène à des programmes incompréhensibles (i.e. programmes **spaghetti**)

Programmation structurée

- Afin d'éviter le spaghetti les langages modernes prônent la **programmation structurée**
  - idée: la structure syntaxique d'un programme doit aider à comprendre ce que fait le programme (en particulier au niveau du séquencement des opérations)
- Déf: le **flux de contrôle** c'est la progression du point d'exécution pendant l'exécution (la séquence de points de contrôle atteints)
- Déf: un programme est **structuré** si le flux de contrôle est directement lié à sa structure syntaxique
  - chaque énoncé (simple ou composé) a **un seul** point de contrôle d'entrée et **un seul** point de contrôle de sortie



Énoncé "bloc"

- Exécution séquentielle d'un groupe d'énoncés
- Exemple en Pascal: `begin x:=x+1; y:=y+1 end`
- Exemple en C: `{ int t; t=x; x=y; y=t; }`
- Certains langages (p.e. C et SIMULA) permettent d'introduire des variables **locales** au bloc (en général au début seulement, C++ et Java sont des exceptions notables: `{ x++; int y=x/2; z=y*y; }`)
- Le point-virgule est un **séparateur** d'énoncés dans certains langages (p.e. Pascal et SIMULA) et un **terminateur** d'énoncés dans d'autres (p.e. C, C++ et Java)
  - En Pascal ce bloc a 3 énoncés:

```
begin x:=x+1; y:=y+1; end   (* 3ème = énoncé vide *)
```

  - En Pascal ceci est illégal:

```
if x < y then
  x := y;
else          (* erreur: else inattendu *)
  y := x;
```

  - ";" comme terminateur est plus intuitif

Énoncé conditionnel (1)

- La plupart des langages offrent deux formes:
  - `if` *<expr>* `then` *<énoncé>*
  - `if` *<expr>* `then` *<énoncé>* `else` *<énoncé>*
- En Modula-2, le `IF` est terminé par un `END`, ce qui n'est pas très esthétique pour les cascades de `IF`s:

```
(* sans ELSIF *)              (* avec ELSIF *)

IF c1 THEN                    IF c1 THEN
  e1                            e1
ELSE                          ELSIF c2 THEN
  IF c2 THEN                    e2
    e2                        ELSIF c3 THEN
  ELSE                          e3
    IF c3 THEN                ELSE
      e3                        e4
    ELSE                      END
      e4
    END
  END
END
```

- Syntaxe Modula-2 (EBNF):
  `IF` *<expr>* `THEN` *<séq_énoncés>*
  { `ELSIF` *<expr>* `THEN` *<séq_énoncés>* }
  [`ELSE` *<séq_énoncés>*]
  `END`

Énoncé conditionnel (2)

- Certains langages (p.e. SIMULA, C, C++ et Java) offrent également des expressions conditionnelles:
  - SIMULA: `if` *<expr1>* `then` *<expr2>* `else` *<expr3>*
  - C/C++/Java: *<expr1>* `?` *<expr2>* `:` *<expr3>*
- Exemple, calcul de la valeur absolue de `x`:

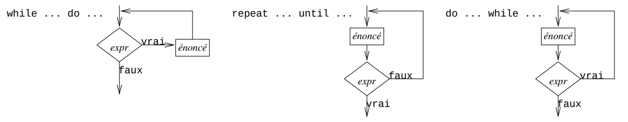
```
x := if x<0 then -x else x;  ! SIMULA ;

x = (x<0) ? -x : x;          /* C */
```

- En C, toute expression est aussi un énoncé (en particulier C traite *<var>*=*<expr>* comme une expression)
- Cela permet d'utiliser les opérateurs "`&&`", "`||`" et "`,`" pour faire des traitements conditionnels:

```
rappel:      X && Y  -->  0 si X==0 sinon Y!=0
             X || Y  -->  1 si X!=0 sinon Y!=0
             X , Y   -->  Y  (après avoir évalué X)

x<0 && (x=-x);          /* x = abs (x); */

x==5 || (x=y=0, z=1);  /* if (x!=5) { x=0; y=0; z=1; } */
```

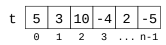
Énoncés `while` et `repeat`

- Nombre d'itérations **pas connu à l'avance**
- Pascal: `while` *<expr>* `do` *<énoncé>*
  - *<expr>* évaluée **avant** chaque itération et après dernière itération
  - ≥ 0 itérations
- Pascal: `repeat` *<séq_énoncés>* `until` *<expr>*
  - *<expr>* évaluée **après** chaque itération
  - ≥ 1 itérations
  - équivalent:
    <séq_énoncés>`;`
    `while not` *<expr>* `do begin` *<séq_énoncés>* `end;`
  - variante C: `do` *<énoncé>* `while (` *<expr>* `);`



Énoncé `break` de C

- `break;` => quitter immédiatement la boucle courante
- Exemple: imprimer la première valeur négative dans un tableau `t` d'entiers de longueur `n`



1.
```
i = 0;
while (i < n && t[i] >= 0) i++;
if (i < n) printf ("%d", t[i]);
/* pas facile de se convaincre que ça marche */
```
2.
```
for (i=0; i<n; i++)
  if (t[i] < 0)
    { printf ("%d", t[i]); break; }
/* plus lisible car isole le cas exceptionnel */
```
3.
```
t[n] = -1;  /* placer une sentinelle */
i = 0;
while (t[i] >= 0) i++;
if (i < n) printf ("%d", t[i]);
/* lisible et rapide */
```


Énoncé `exit` de Ada

- Le `break` de C ne s'appliquent qu'à **la boucle englobante la plus proche**

```
while (...)              /* boucle 1 */
  {
    while (...)          /* boucle 2 */
      {
        ... break; ...   /* sort de la boucle 2 */
      }
  }
```

- Ada permet de nommer les boucles et offre une variante de `break`:
  `exit` [*<nom_de_boucle>*] [`when` *<expr>*] `;`

```
boucle1:
  while x<10 loop                -- boucle 1
    ...
    boucle2:
      while x<5 loop             -- boucle 2
        ...
        if x<0 then exit end if; -- sort de la boucle 2
        exit when x<0;           -- sort de la boucle 2
        exit boucle1;            -- sort de la boucle 1
        exit boucle1 when x<0;   -- sort de la boucle 1
        ...
      end loop;
    ...
  end loop;
```


Énoncé `continue` de C

- `continue;` => passer immédiatement à la prochaine itération
- Exemple: copier l'entrée à la sortie en enlevant les lignes vides
  - sous UNIX, fin de ligne = caractère `'\n'`
  - idée: ne pas copier `'\n'` précédé d'un `'\n'` ou au début

```
int main (void)
{
  int c, fdl;
  fdl = TRUE;
  while ((c = getchar ()) != EOF)
    {
      if (c == '\n')
        {
          if (fdl) continue;
          fdl = TRUE;
        }
      else
        fdl = FALSE;
      putchar (c);
    }
  return 0;
}
```


Énoncé `return` de C et Ada

- `return;` => quitter immédiatement la procédure courante
- `return` *<expr>*`;` => quitter immédiatement la fonction courante
  - En Pascal, SIMULA et Modula-2 il faut terminer l'exécution du corps (après avoir affecté le résultat au nom de la fonction dans le cas d'une fonction)
- Exemple: obtenir la position dans un tableau ou se trouve un élément particulier

```
int position (int element)
{
  int i = 0;
  while (i < n)
    {
      if (t[i] == element) return i;
      i++;
    }
  return -1; /* indiquer échec */
}
```


Énoncé `loop` de Ada et Modula-3

- `loop` *<séq_énoncés>* `end loop;`
- Donne une boucle infinie, sauf si sortie prématurée (par exemple avec `exit`)
- Exemples:

```
-- Ada                        (* en Pascal sans goto *)

loop                          while x <= 10 do
  exit when x > 10;             x := x*2
  x := x*2;
end loop;


loop                          repeat
  x := x*2;                     x := x*2
  exit when x > 10;           until x > 10
end loop;


loop                          x := read (f);
  x := read (f);              x := x*x;
  x := x*x;                   while x <= 10 do
  exit when x > 10;             begin
  write (x);                      write (x);
end loop;                         x := read (f);
                                  x := x*x
                                end
                              (* duplication de code
                                 pas élégant *)
```


Énoncés de boucle défini (1)

- Nombre d'itérations **connu à l'avance**
- Pascal:
  `for` *<var>* `:=` *<expr1>* (`to` | `downto`) *<expr2>* `do` *<énoncé>*
  => les bornes doivent être de type entier ou énuméré
- Exemple: `for i := x+1 to x*x do writeln (i)`
- Une implantation possible de `for`:

```
(* cas ‘‘to’’ *)              (* cas ‘‘downto’’ *)
LIMITE := <expr2>;            LIMITE := <expr2>;
<var> := <expr1>;             <var> := <expr1>;
while <var> <= LIMITE do      while <var> >= LIMITE do
  begin                         begin
    <enoncé>;                     <enoncé>;
    <var> := <var>+1              <var> := <var>-1
  end                           end
```


Énoncés de boucle défini (2)

- Pour permettre plus de liberté à l'implanteur du compilateur, en Pascal:
  - il est **interdit de modifier** *<var>* tant que la boucle n'est pas finie
  - la valeur de *<var>* est **indéfinie** à la sortie de la boucle
- Cela admet cette implantation qui est plus performante sur plusieurs machines:

```
(* cas ‘‘to’’ *)
LIMITE := <expr2>;       (* LIMITE et TEMP stockées dans *)
TEMP := <expr1>;         (* des registres (accès rapide) *)
while TEMP <= LIMITE do
  begin
    <var> := TEMP;
    <enoncé>;            (* modification de <var> ici ne *)
    TEMP := TEMP+1       (* changerais pas le nombre *)
  end                    (* d'itérations *)
```

- Note: à la fin du `for`:
  - Implantation 1: *<var>* = *<expr2>*+1 (si >=1 itérations) et *<expr1>* (si 0 itération)
  - Implantation 2: *<var>* = *<expr2>* (si >=1 itérations) et valeur avant le `for` (si 0 itération)

Énoncés de boucle défini (3)

- SIMULA:

  `for` *<var>* `:=` *<item>* { , *<item>* } `do` *<énoncé>*

  <item> ::= *<exp>* | *<exp1>* `step` *<exp2>* `until` *<exp3>*

- Exemple: `for i := 2, 3, 5 step 5 until 30 do f (i)`

- C: `for ([`*<exp1>*`]; [`*<exp2>*`]; [`*<exp3>*`]) do` *<énoncé>*

- Le `for` de C est approximativement équivalent à:

```
{ <exp1>; while (<exp2>) { <enoncé>; <exp3>; } }

/* <exp2> est remplacé par TRUE si absent */
/* en C++ <exp1> peut déclarer une variable locale */
```

  sauf qu'un `continue` saute à *<exp3>*

- Exemples:

```
for (i=1; i<5; i++) printf ("%d", i);     /* 1 2 3 4 */
for (i=1; i<9; i=i*2) printf ("%d", i);   /* 1 2 4 8 */
for (p=list; p!=NULL; p=p->next) ...;
for (int i=1; i<5; i++) printf ("%d", i); /* 1 2 3 4 */

for (;;)
  { if ((c = getchar ()) == EOF) break;
    putchar (c);
  }
```


Énoncés de boucle défini (4)

- Variations sémantiques:
  - **exécution du corps au moins une fois?** (FORTRAN=oui, en général=non)
  - **évaluation unique des bornes?** (Pascal=oui, C=non, SIMULA=oui sauf pour *<exp3>* après le `until`)
  - **progression de la variable de contrôle?** (Pascal=pas de +1 ou -1, C=pas variable, SIMULA=pas quelconque mais constant pour la durée de la boucle)
  - **type de la variable de contrôle?** (Pascal=entier, C=quelconque, SIMULA=entier ou flottant)
  - **déclaration locale de la variable de contrôle?** (Pascal=non, C=non; SIMULA=non, Ada=oui, Modula-3=oui, C++=possible)
  - **variable de contrôle modifiable pendant la boucle?** (Pascal=non, C=oui, SIMULA=oui)
  - **variable de contrôle définie après la boucle?** (Pascal=non, C=oui, SIMULA=oui)

Énoncé de sélection par cas (1)

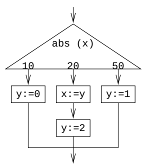
- Sélection d'un énoncé parmis plusieurs

- Pascal:

  `case` *<expr>* `of`
  <constante> { , *<constante>* } : *<énoncé1>* ;
  <constante> { , *<constante>* } : *<énoncé2>* ;
  ...
  `end`

- Exemple

```
case abs (x) of
  10: y := 0;
  50: y := 1;
  20: begin x := y; y :=2; end;
end
```



- En général
  - constantes entières ou énumérées seulement
  - constantes dans n'importe quel ordre
  - ≥ 1 constantes par énoncé
  - constantes distinctes

Énoncé de sélection par cas (2)

- Variations sémantiques:
  - **action lorsque cas pas précisé?** (Pascal=erreur, C=rien, Modula-2=erreur)
  - **action de défaut permise?** (Pascal=pas standard, C=oui, Modula-2=oui)
  - **intervalle de cas permis (p.e. 10..20)?** (Pascal=non, C=non, Modula-2=oui)

Énoncé de sélection par cas (3)

- C offre: `switch` (*<expr>*) *<énoncé>*

  => *<énoncé>* peut contenir comme sous-énoncés:

  `case` *<constante>* `:` *<énoncé>*

  `default:` *<énoncé>*

  `break;`

-
```
switch (x)
  {
    case 10: y = 1; break; /* le break sort du switch */
    case 20:
    case 30: y = 2; break;
    default: y = 3;
  }
```

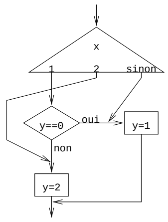
- La syntaxe de C est une source de bugs:

```
switch (x)
  {
    case 1: y = 1; break;
    defaut: y = 2;        /* ne sera jamais exécuté */
  }
```


Énoncé de sélection par cas (4)

- L'énoncé `switch` n'est pas structuré

```
switch (x)
  case 1:
    if (y==0)
      default: y = 1;
    else
      case 2: y = 2;
```



- Cela est parfois utile:

```
void copier (char *src, char *dst, int n)
{
  while (n > 0) { *dst++ = *src++; n--; }
}

void copier (char *src, char *dst, int n)
{
  switch (n & 3)
    {
      case 0: while (n > 0) { *dst++ = *src++;
      case 3:                 *dst++ = *src++;
      case 2:                 *dst++ = *src++;
      case 1:                 *dst++ = *src++;
                              n -= 4;
                            }
    }
}
```


Types de données et leur représentation

- Il faut distinguer **objet** et **représentation**
  - **Objet** = donnée du point de vue de l'application
  - **Représentation** = organisation des valeurs composant l'objet (i.e. encodage de l'objet)
- Exemple: objet = date du 3 février
  - Représentation 1: entier 34
  - Représentation 2: entiers 3 et 2
  - Représentation 3: chaîne "03/02"
- Déf: **Type** = ensemble d'objets possiblement infini
- Exemple:
  1. Type booléen = { VRAI, FAUX }
  2. Type entier = { 0, 1, -1, 2, -2, ... }

Types de données (1)

- Classifiés en 2 groupes
- **Types simples** (normalement prédéfinis), p.e. entier, caractère, booléen, point flottant
- **Types composés** (normalement définis explicitement), formés à l'aide d'autres types (éléments de même type=**homogène**, sinon **hétérogène**)
- **Tableau** = groupe ordonné d'objets accessibles par leur position
  - Index connu à l'exécution
  - En général homogène, mais hétérogène en Lisp et les langages OO
  - 2 bornes (Pascal, Ada) / longueur + borne inférieure fixée à 0 (C, Lisp) ou 1 (FORTRAN)
  - **Bornes fixées?** à la compilation (**tableau statique**: C) ou à l'exécution (**tableau dynamique**: Pascal, Lisp)

Types de données (2)

- **Enregistrement** = groupe ordonné d'objets accessibles par leur nom
  - Nom connu à la compilation
  - En général hétérogène
- **Ensemble** = groupe non-ordonné d'objets sans duplication, accessibles par élément
  - En général homogène avec un type discret de petite taille, mais hétérogène en Lisp
  - Exemple: ensemble de booléens = { {}, {VRAI}, {FAUX}, {VRAI,FAUX} }
  - opérations: union, intersection, différence, "élément de", inclus, égal

Représentation des entiers (1)

- Unités de mesure = bit, octet, mot (taille naturelle pour l'UCT, 32/64 bits = 4/8 octets)
- **Type entier**
  - Entier = sous-ensemble de **Z** (sauf Lisp où: entier = **Z** = { 0, 1, -1, 2, -2, ... })
  - Entier = entiers signés représentables par un mot: complément à 2, i.e. `-2^31..2^31-1`, ou complément à 1, i.e. `-2^31+1..-0,0..2^31-1` (Cray et CDC)
  - C permet de choisir la taille des entiers et signé ou pas:

    [`signed` | `unsigned`] (`int` | `short` | `long` | `char`)

    * Note 1: `char` est un type entier qui occupe une "unité" d'espace (normalement 1 octet) Exemple: `unsigned char x;` 0<=`x`<=255
    * Note 2: `char` ⊆ `short` ⊆ `int` ⊆ `long`
    * Note 3: `#include <limits.h>` pour avoir les limites de chaque type: `INT_MIN`/`INT_MAX`

Représentation des entiers (2)

- Une autre variation dans la représentation des entiers est l'ordre des octets en mémoire
- Les deux ordres principaux sont "**big-endian**" (MIPS, SPARC, M68000) et "**little-endian**" (Intel, DEC Alpha)
- Exemple, l'entier 16 bits 258 = 1*256 + 2

- Cela cause principalement des problèmes lors de la communication de données binaires entre ordinateurs de type différent (par exemple sur l'internet)
  - Il faut convertir dans une représentation "standard", envoyer la donnée, et convertir à l'autre bout (si nécessaire)

Représentation des entiers (3)

- **Type sous-intervalle**
  - Intervalle du type entier
  - Pascal: <const1>`..`<const2> = { <const1>,<const1>+1,...,<const2> }
  - Nombre bits = plafond(log2(C2-C1+1))
  - Exemple: `1..2` −> 1 bit, `5..7` −> 2 bits
  - Souvent les compilateurs arrondissent vers l'octet le plus proche pour un accès rapide
  - Utilisé en Pascal pour définir les bornes des tableaux:

```
var T1 : array [10..20] of integer;
    T2 : array [char] of integer;
```

Représentation des types énumérés

- **Type énuméré**
  - Chaque objet élément du type a un nom symbolique
  - Exemple en Pascal:

```
var j : (lundi,mardi,mercredi,jeudi,vendredi,
         samedi,dimanche);

begin
  j := mercredi;
  if j = lundi then ...;
  j := succ(j); (* j=jeudi *)
  j := pred(j)  (* j=mercredi *)
end
```

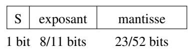
  - Représenté comme type sous-intervalle
  - Exemple en C:

```
enum {lundi,mardi,mercredi,jeudi,vendredi,
      samedi,dimanche} j;
```

  - Le type booléen de Pascal est un type énuméré: `boolean = (false,true)`

Représentation des types réels

- **Type réel**
  - Réel = sous-ensemble de **R** (en fait sous-ensemble de **Q**, les rationnels)
  - Réel = nombres rationnels représentables avec un encodage point-flottant (le plus répandu = IEEE-754)



→ `+/- (1.mantisse`$_2$`) x 2`$^{\texttt{exposant}}$

  - Précision "simple" (32 bits) <= `3.4x10^38` (6 chiffres significatifs)
  - Précision "double" (64 bits) <= `1.8x10^308` (15 chiffres significatifs)
  - En C: `float` $\subseteq$ `double`

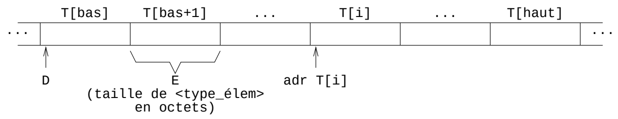
Représentation des tableaux (1)

- **Type tableau**
  - Pascal:
    `var T : array [`*<type_index>*`] of` *<type_élem>*`;`
    => *<type_index>* doit être sous-intervalle ou énuméré
  - C: *<type_élem>* `T[`*<constante>*`];`
  - Exemple

```
Pascal:
  var T : array [boolean] of real;                 T[false] := 1.5;
  var T : array [(rouge,vert,bleu)] of boolean;   T[vert] := true;
  var T : array [10..12] of integer;              T[11] := 123;
  var T : array [0..4] of array [0..9] of real;   T[2][3] / T[2,3]
  var T : array [0..4,0..9] of real;              T[2][3] / T[2,3]

C:
  int T[2];           T[0] = 123;
  float T[5][10];     T[2][3] = 1.5;  T[2,3] = 1.5;
  float T[5,10];      T[2][3] = 1.5;  T[2,3] = 1.5;
```


Représentation des tableaux (2)

- Représentation de
  `var T : array [ bas .. haut ] of integer;`



- Taille de `T` = (`haut` - `bas` + 1)*E

- Adr de `T[i]` = D + (`i`-`bas`)*E = `i`*E + K
  où K = D − `bas`*E

- Quand indexation rapide?
  1. Utilisation d'un tableau statique, car K est calculable à la compilation
  2. Utilisation d'une taille d'élément E = puissance de 2, car décalage plus rapide que multiplication

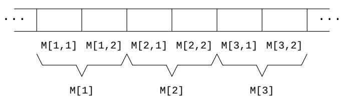
Représentation des tableaux (3)

- Tableau multidimentionnel:

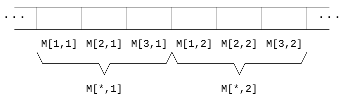
```
var M : array [1..3] of array [1..2] of integer;
```

définition d'une rangée de la matrice

$$\begin{bmatrix} M[1,1] & M[1,2] \\ M[2,1] & M[2,2] \\ M[3,1] & M[3,2] \end{bmatrix}$$



''row-major''



''column-major''
(FORTRAN)

- Taille de `M` = (`h1-b1`+1)*(`h2-b2`+1)*E

- Adr de `M[i,j]` (en représentation "row-major")
  = D + (`i-b1`)*(`h2-b2`+1)*E + (`j-b2`)*E
  = `i`*(`h2-b2`+1)*E + `j`*E + (D-`b1`*(`h2-b2`+1)*E-`b2`*E)
  = `i`*K1 + `j`*E + K2

- Donc `array[1..3,1..2]` est plus rapide à indexer que `array[1..2,1..3]`

Représentation des tableaux (4)

- SIMULA offre les tableaux dynamiques

```
procedure p(n); integer n;
  begin
    integer array t(1:n);
    ...
  end;

p(10);  p(20);
```

- Le calcul d'adresse doit être **dynamique** (donc il faut stocker les bornes en plus des éléments)

- Représentation 1: permet de vérifier les indices à l'exécution

bas1	haut1	bas2	haut2	...	elem1	elem2	...

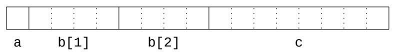
- Représentation 2: plus rapide mais pas de vérification et suppose borne du bas = 0 (Di = `haut_i`+1)

D1*E	D1*D2*E	...	elem1	elem2	...

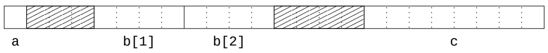
Représentation des enregistrements (1)

- Disposition contiguë des champs en mémoire

```
record
  a : char;                    (* adr relative = 0 *)
  b : array [1..2] of integer; (* adr relative = 1 *)
  c : real;                    (* adr relative = 9 *)
end
```

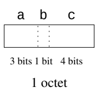


- Les contraintes de la machine peuvent affecter la position (p.e. `integer` (`real`) à des adresses qui sont des multiples de 4 (8))



- C permet un contrôle fin (au niveau des bits) sur la disposition des champs

```
struct
  { unsigned int a : 3; /* 0..7 */
    unsigned int b : 1; /* 0..1 */
    unsigned int c : 4; /* 0..15 */
  }
```

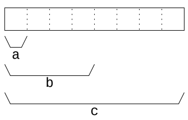


Représentation des enregistrements (2)

- Déf: **type union** = union de plusieurs types

- En C la syntaxe est similaire aux structures

```
union
  { char a;    /* adr relative = 0 */
    int b;     /* adr relative = 0 */
    double c;  /* adr relative = 0 */
  }
```



- Taille = taille du plus grand champs

- Seulement un des champs est actif à un instant donné et c'est impossible de savoir lequel uniquement en examinant l'objet

```
struct                   for (i=0; i<10; i++)
 { int sorte; /*0..2*/    switch (T[i].sorte)
   union                   {
    { char a;                case 0: printf("%c",T[i].val.a); break;
      int b;                 case 1: printf("%d",T[i].val.b); break;
      double c;              case 2: printf("%f",T[i].val.c); break;
    } val;                 }
 } T[10];
```

Représentation des ensembles

- Pascal: **set of** <*type*>
  => <*type*> doit être sous-intervalle ou énuméré

- Représenté par un mot machine avec 1 bit par élément possible (1=élement de l'ensemble, 0=pas élement de l'ensemble)

- Exemple

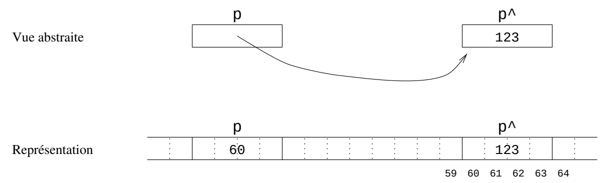
```
var s1, s2, s3 : set of 1..5;

s1 := [1,4,5];   (* 0000000000011001 = 25 *)
s2 := [2,4];     (* 0000000000001010 = 10 *)
s3 := s1 + s2;   (* 0000000000011011 = 27 *)
s3 := s1 * s2;   (* 0000000000001000 = 8  *)
s3 := s1 - s2;   (* 0000000000010001 = 17 *)
if 2 in s1 then ...;
```


Représentation des pointeurs (1)

- Déf: **pointeur** = objet qui permet un accès **indirect** à un autre objet

- Pascal: ^<*type*>

```
var p : ^integer;
```



- La taille du pointeur est indépendante de la taille de l'objet pointé (typiquement 1 mot machine)

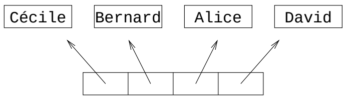
- En général, un pointeur est représenté par l'adresse en mémoire de l'objet pointé

- Cas spécial: pointeur nul (`nil` en Pascal) => 0 ou autre adresse invalide

Représentation des pointeurs (2)

- Applications
  - Structures de données dont la taille varie à l'exécution (chaînes, listes, arbres, graphes)
  - Manipulation rapide d'objets (la copie d'un pointeur est plus rapide que la copie de l'objet pointé)

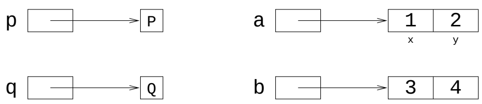
    Exemple: trier des enregistrements



- Opérations principales sur les pointeurs
  - Allocation d'un objet dans le tas (trouve espace inutilisé + marque comme utilisé)
  - Récupération de l'espace alloué à un objet (marque inutilisé)
  - Indirection (accès à l'objet pointé)

Représentation des pointeurs (3)

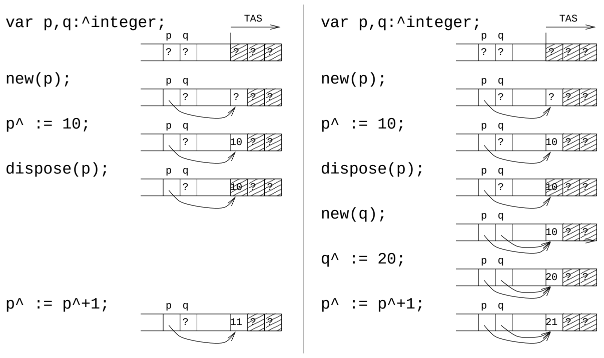
```
  Pascal                C                            SIMULA

DECLARATION DE TYPE ET VARIABLE

  type R =              struct R                     class R;
    record               {                            begin
      x, y : integer;     int x, y;                    integer x, y;
    end;                 };                           end;

  var p, q : ^char;     char *p, *q;
      a, b : ^R;        struct R *a, *b;             ref(R) a, b;

INDIRECTION

  p^ := 'X';            *p = 'X';
  a^.x := 20;           (*a).x = 20; a->x = 20;      a.x := 20;

AFFECTATION DE POINTEUR

  q := p;               q = p;
  b := a;               b = a;                       b :- a;
  b := nil;             b = NULL;                    b :- none;

AFFECTATION DE L'OBJET POINTE

  b^ := a^;             *b = *a;                     b := a;

EGALITE

  if a=b then ...       if (a==b) then ...           if a==b then ...

ALLOCATION

  new(a);               a = (struct R*) malloc(      a :- new R;
                              sizeof(struct R));

RECUPERATION

  dispose(a);           free(a);                     **AUTOMATIQUE**
```



Représentation des pointeurs (4)

- Il y a des dangers importants lorsque la récupération de la mémoire est sous la responsabilité du programmeur (**récupération manuelle**)

- Le programmeur risque de récupérer l'espace **trop tôt** ce qui engendre des **pointeurs fous** (pointeur vers un espace réservé pour un autre usage)



Représentation des pointeurs (5)

- Le programmeur risque de récupérer l'espace **trop tard** (ou même jamais) ce qui engendre des **fuites de mémoire**

```
new(p);  (* allocation de l'objet 1 *)
p^ := 10;
new(q);  (* allocation de l'objet 2 *)
p := q;  (* fuite de mémoire: l'objet 1 n'est plus
            accessible et ne peut plus être récupéré *)
```

- Déf: **objet mort** = objet qui ne sera jamais plus utilisé par le programme

- Une fois qu'un objet n'est plus atteignable il est mort pour toujours

- Déf: **fuite de mémoire** = création d'objets morts

- Bug insidieu car le programme peut marcher longtemps sans problèmes mais tranquillement la performance se dégrade et cause éventuellement un arrêt total (p.e. Microsoft Word)

Représentation des pointeurs (6)

- Lisp, SIMULA et Java possèdent des "**ramasses-mièttes**" (**garbage collectors**) qui détecte la création d'objets morts et récupère automatiquement l'espace associé

- Deux techniques répandues:

  1. Associer un **compteur de référence** à chaque objet = nombre de pointeurs vers l'objet (initialisé à 1, incrémenté à chaque copie de pointeur, et décrémenté à chaque fois qu'un pointeur disparait, p.e. écrasé par un autre pointeur). Récupérer espace lorsque compteur = 0. Ne fonctionne pas dans le cas des structures cycliques.

  2. À partir des variables globales et la pile, **traverser tous les objets atteignables** en passant par des pointeurs et marquer ces objets. À la fin, les objets non-marqués sont récupérés.

- La récupération manuelle est une source de bugs majeurs (30%-40% du temps de débogage) dans les programmes complèxes car il est difficile de savoir quel module est responsable de récupérer l'espace

Représentation des pointeurs (7)

- C possède plusieurs routines de gestion mémoire (disponibles dans `"stdlib.h"`)

```
void *malloc (int taille_bloc);
void *calloc (int nb_elem, int taille_elem);
void *realloc (void *ptr, int nouvelle_taille);
void free (void *ptr);
```

- `malloc` alloue un bloc, `calloc` alloue un tableau et `realloc` change la taille d'un objet alloué dans le tas (ce qui demande parfois d'allouer un nouveau bloc et de copier l'objet)

- Le pointeur retourné est de type `void*` (pointeur universel) et il faut le convertir vers le bon type de pointeur avec un "cast"

- `NULL` est retourné si la mémoire est épuisée

- C++ possède 2 opérateurs de gestion mémoire de plus haut niveau que C

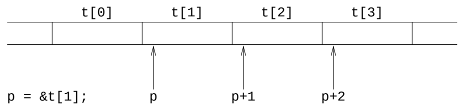
```
int *p;
char *s;
p = new int;      // p = (int*)malloc(sizeof(int));
s = new char[5]; // s = (char*)calloc(5,sizeof(char));
delete p;         // free(p);
delete [] s;      // free(s);
```

Représentation des pointeurs (8)

- Le langage C permet une forme d'**arithmétique sur les pointeurs** (opérateurs + et -)

- &<*var*> => pointeur vers <*var*>

- `p+n` (où `n` est de type entier et `p` est de type `T*`) => pointeur vers `n`ième objet de type `T` après objet pointé par `p`



- `p2-p1` (où `p1` et `p2` sont de type `T*`) => nombre d'objets de type `T` de `*p1` à `*p2`

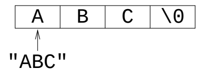
- `t` (où `t` est un tableau) => pointeur vers premier élément du tableau

- C définit: `x[y]` = `*(x+y)` = `*(y+x)`
  Donc (!): `1[t]` = `t[1]` et `1[t][t]` = `t[t[1]]`

- Les opérateurs `++` et `--` sont utilisables sur les pointeurs pour avancer ou reculer d'un élément

Représentation des pointeurs (9)

- Le traitement de chaînes de caractères en C est basé sur les pointeurs




```
char t[4];
char *p, *q;

p = "ABC";

putchar (p[2]);      /* imprime C */
putchar ("XYZ"[2]);  /* imprime Z */

q = t;               /* copier dans t */
while (*p != '\0')
  *q++ = *p++;
*q = '\0';

if (t == "ABC") ...; /* toujours faux! */

/* utiliser strcmp(t,"ABC") de <string.h> */
```


Typage et sûreté (1)

- La notion de type est utile pour définir la notion d'**erreur**

- Exemples

  1. `int *p; ... abs(p) ...` est incorrect car (en C) l'argument de `abs` doit être de type entier

  2. `int t[10]; ... t[1.5] ...` est incorrect car l'index doit être de type entier

  3. `int t[10]; ... t[i] ...` est incorrect si `i` est négatif (ou plus grand que 9)

  4. `int n; ... 100/n ...` est incorrect si `n` est égal à 0

- Déf: Une **erreur de type** survient lorsqu'une fonction (ou opérateur) est appelée avec un argument qui n'est pas du type attendu

Typage et sûreté (2)

- Les **vérifications de type** permettent de garantir qu'aucune opération incorrecte ne sera exécutée

- Il est avantageux d'effectuer les vérifications de type **à la compilation** (**typage statique**) car cela évite de ralentir l'exécution du programme

- Cela rend le programme plus **sûr**, c'est-à-dire qu'il y a peu ou aucune possibilité d'erreur de type à l'exécution

- Les vérifications de type peuvent également se faire à l'exécution (**typage dynamique**)
  - C'est requis dans certains cas (exemples 3 et 4)
  - C'est dangereux si la partie du programme contenant l'erreur est exécutée rarement car il sera difficile d'éliminer l'erreur pendant la phase de développement (il est bon d'avoir des **tests de couverture** qui exécutent au moins une fois chaque partie du programme)