L1-S2 MPI : Programmation Impérative
Correspondances Langage de Réalisation $\Longleftrightarrow$ C++

Ce résumé constitue un aide-mémoire rédigé sous forme d'exemples qui illustrent partiellement l'utilisation des différentes composantes du Langage de Réalisation et la manière de les écrire dans le langage de programmation C++.

1 Instructions

Affectation

a ← 2	`a = 2;`
val ← 2x + 1	`val = 2*x + 1;`

Lecture

☞ En C++, pour les lectures et écritures il faut mettre :

```
#include <iostream>
using namespace std;
```

On peut alors utiliser les commandes suivantes :

lire (x)	`cin >> x;`
lire (a,b,c)	`cin >> a >> b >> c;`

Ecriture

écrire ("Hello, je m'appelle HAL.")	`cout << "Hello, je m'appelle HAL." << endl;`
écrire (x)	`cout << x;`
écrire ("x = ", x)	`cout << "x = " << x;`

☞ On peut imposer la taille de l'objet à insérer en écrivant `setw`(l) avant l'objet (pour ceci il faut inclure le fichier `iomanip`. On peut également régler la précision avec `setprecision`(p). Par exemple, les instructions :

```
cout << setw(10) << 12 << "+" << setprecision(4) << 3.141592653;
```

affiche «␣␣␣␣␣␣␣␣12+3.142» où les espaces sont matérialisés par «␣».

Bloc d'instructions

Lexique	`{`
x, y : entier	`int x,y;`
début	
lire (x)	`cin >> x;`
lire (y)	`cin >> y;`
écrire (x+y)	`cout << x+y;`
fin	`}`

Alternative

☞ Attention à l'opérateur de comparaison = en LR, == en C++ :

```
si a = 0 alors écrire("valeur nulle") finsi
```

```
if (a==0)  cout << "valeur nulle";
```

```
si pair (x)
      alors x ← x - 1
      sinon x ← 3x + 2
finsi
```

```
if (pair(x))
  x--; /* équivalent à x=x-1; */
else
  x=3*x+2;
```

```
si a > b alors
     a ← a+1
     b ← b+1
finsi
```

```
if (a>b) {
    a++; /* équivalent à a=a+1; */
    b++;
  }
```

Choix multiples

Les instructions **selon le cas** ou **switch** permettent de faire plusieurs tests de valeurs sur le contenu d'une variable.

```
selon le cas x
     cas 1 : écrire ("unité")
     cas 10 : écrire ("dizaine")
     cas 100 : écrire ("centaine")
     sinon : écrire ("inconnu")
finselon
```

```
switch(x) {
   case 1 : cout << "unité"; break;
   case 10 : cout << "dizaine"; break;
   case 100 : cout << "centaine"; break;
   default : cout << "inconnu";
  }
```

Itération

```
Tantque pair(x) faire
     x ← x / 2
     écrire(x)
fintantque
```

```
while (pair(x)) {
    x=x/2;
    cout << x;
  }
```

```
Répéter
   lire(x)
tantque x = 0
```

```
do
  cin >> x;
while (x==0);
```

```
s ← 0
Pour i allant de 1 à 10 faire
     s ← s + i
finpour
```

```
s=0;
for (i=1; i<=10; i++)
  s=s+i;
```

```
Pour  i allant de 5 à 0 par pas de -1 faire
   écrire (i)
finpour
```

```
for (i=5; i>=0; i--)
  cout << i;
```

☞ En Langage de Réalisation, il est interdit de modifier les bornes de l'intervalle ou la valeur de l'indice à l'intérieur d'une boucle **Pour**.

2 Types concrets

Types prédéfinis :

— entier	— `short,int,long`
— réel	— `float,double`
— caractère	— `char`
— booléen	— `bool` dont les valeurs sont `true` pour VRAI et `false` pour FAUX
— chaîne	— `string` (inclure le fichier `string`).

☞ En `C++`, les constantes de type `char` peuvent s'écrire comme des caractères (`'a'`,`'\n'`,...) ou comme des entiers. Attention ! `'0'` $\neq$ `0`.

Type énuméré

type couleur = (pique, coeur, carreau, trèfle)

```
enum couleur {pique, coeur,
              carreau, trefle};
```

☞ En `C++`, les types énumérés sont en fait des entiers, on a ainsi déclaré quatre constantes : `pique` = `0`, `coeur` = `1`, `carreau` = `2`, `trefle` = `3`.

Type intervalle

type lettre = caractère dans ['a'..'z']
type chiffre = entier dans [0 .. 9]

n'existe pas en `C++`, utiliser des entiers

Type produit

type complexe : produit
    re : réel
    im : réel
fin

```
struct complexe {
  float re,im;
};
```

lexique c, c_1 : complexe
début
  c.re $\leftarrow$ $-$ 0. 12
  c.im $\leftarrow$ 23.3
  $c_1 \leftarrow$ c
fin

```
{
  complexe c,c1;
  c.re = -.12;
  c.im = 23.3;
  c1 = c;
}
```

☞ Le langage de réalisation et le `C++` autorisent les affectations entre structures.

Type tableau

☞ En C/C++, les tableaux de taille n sont toujours indexés par des entiers compris entre 0 inclus et n exclus donc dans l'intervalle $[0..n-1]$.

t : tableau [0..4] de réel	`float t[5];`
m : tableau [0..9,0..9] de entier	`int m[10][10];`
t[0] ← 2.1	`t[0] = 2.1;`
m[5][2] ← 4	`m[5][2] = 4;`

Définitions de types de tableaux

type vecteur = tableau [0..9] de réel	`typedef float vecteur[10];`
type matrice = tableau [0..9,0..9] de entier	`typedef int matrice[10][10];`
type lettre = caractère dans ['a'..'z']	`typedef char lettre;`
type TabLettres = tableau [lettres] de entier	`typedef int TabLettres[26];`
type carte = (roi, dame, valet, as)	`enum carte {roi,dame,valet,as};`
type valeur_carte = tableau [carte] de entier	`typedef int valeur_carte[4];`

Écriture dans les tableaux : exemples

```
Lexique                                  {
  v : vecteur                              vecteur v;
  m : matrice                              matrice m;
  t : TabLettres                           TabLettres t;
  belote : valeur_carte                    valeur_carte belote;
début
  v[2] ← π                                 v[1]=3.1415;
  m[4,6] ← 3                               m[3][5]=3;
  t['p'] ← 0                               t['p'-'a']=0; /* on décale de -'a' */
  belote[dame] ← 10                        belote[dame]=10;
  ...                                      ...
fin                                      }
```

Type vecteur

☞ Le C++ fournit une structure de donnée plus souple que les tableaux appelée `vecteur`. À l'inverse d'un tableau, la taille d'un vecteur n'est pas fixée et peut changer.

☞ Pour pouvoir utiliser les vecteurs il faut inclure le fichier d'entêtes `vector`.

☞ Comme la taille d'un vecteur n'est pas fixé, le compilateur ne peux pas réserver (allouer) la mémoire. Il faut le faire indépendamment de la déclaration, à l'exécution.

☞ L'accès aux éléments d'un vecteur se fait comme pour un tableau.

☞ Contrairement au tableau, l'affectation des vecteurs est possible. La sémantique est de *recopier* les éléments du vecteur.

— Création d'un vecteur vide (de taille 0) : `vector<int> v;`
— Allocation d'un vecteur : `v = vector<int>(10);`
— Création et initialisation d'un vecteur : `vector<int> v = {3,1,2,5};`
— Calcul de la taille (nombre d'élément) : `v.size()`
— Ajout d'un élément à la fin (change la taille) : `v.push_back(4);`

3 Fonctions et procédures

Procédures

Procédure divE (Données : a, b : entier ;
Résultats : q, r : entier)

```
// où les variables résultats  q et r sont
// respectivement le quotient et le reste de
// la division entière de a par b, deux entiers
début
    q ← 0
    tantque a ≥ b faire
        a ← a - b
        q ← q + 1
    fintantque
    r ← a
fin
...
divE(125,37,x,y)
```

```
/** Division entière (euclidienne)
 *  @param a, b deux entiers
 *  @return q, r reçoivent respectivement
 *     le quotient et le reste
 **/
void divE(int a, int b, int &q, int &r) {
  q=0;
  while (a >= b) {
    a=a-b;
    q++;
  }
  r=a;
}
...
divE(125,37,x,y)
```

Procédure échanger
(Donnée-Résultat x, y : entier)

```
lexique
    t : entier
début
    t ← x
    x ← y
    y ← t
fin
...
échanger(a,b)
```

```
void echanger(int &x, int &y)
{
  int t;

  t = x;
  x = y;
  y =  t;
}

...
echanger(a, b);
```

Fonctions

Fonction puissance
(Données : a, n : entier) → entier

```
// Calcule la puissance n d'un entier a donné
lexique
    res, i : entier
début
    res ← 1
    pour i allant de 1 à n faire
        res ← res * a
    finpour
    retourner res
fin
...
p ← puissance(x+1,4)
```

```
/** Calcule la puissance
 *  @param a un entier
 *  @param n un entier
 *  @return la valeur de a puissance n
 **/
int puissance (int a,int n) {
  int res,i;
  res=1;
  for (i=1;i<=n;i++)
    res=res*a;
  return(res);
}
...
p = puissance(x+1,4);
```

☞ En Langage de Réalisation, il est interdit de mettre l'instruction "retourner" dans une boucle.

Problèmes pratiques en C++

☞ En C++, les paramètres formels correspondant aux Données-Résultats et aux Résultats doivent être précédés de & dans la déclaration de la procédure (voir *échanger*).

☞ **Le point précédent ne s'appliquent pas aux tableaux** qui sont représentés par des pointeurs. Il n'est pas possible de passer un tableau par valeur en C++. Il faut utiliser une structure comme un vecteur.

☞ Une procédure ou fonction C++ doit être définie ou déclarée avant d'être utilisée. Par exemple, pour déclarer la procédure *échanger*, on écrira la ligne suivante : `void echanger(int &a, int &b);`

☞ Déclarer une procédure ou fonction ne dispense pas de la définir ensuite !

Exemples avancés en C/C++

type tab = tableau [0..9] de entier

```
typedef int tab[10];
```

Procédure annule (Résultat : t : tab)
// Remplit un tableau de zéros
lexique
  i : entier
début
  pour i allant de 0 à 9 faire
    t[i] ← 0
  finpour
fin
...
Lexique v : tab
  m : tableau [0..3] de tab
...
annule(v) ; annule(m[2])
// annule v et la troisième ligne de m

```
/** Remplit un tableau de zéros
 *  @param t un tableau de 10 entiers
 **/
void annule (tab t) {
  for (int i=0; i<10; i++)
    t[i]=0;
}

...

tab v, m[4];
...

annule(v); annule(m[2]);
// annule v et la troisième ligne de m
```

Procédure saisie(Résultat c : complexe)
// Lit un complexe au clavier
début
  écrire("entrez la partie réelle :")
  lire(c.re)
  écrire("entrez la partie imaginaire :")
  lire(c.im)
fin
...
lexique z : complexe
début
  initialiser(z)
...

```
/** Lit un complexe au clavier
 *  @param c une variable de type complexe
 **/
void saisie(complexe &c) {
  cout << "entrez la partie réelle:";
  cin >> c.re;
  cout << "entrez la partie imaginaire:";
  cin >> c.im;
}
...
complexe z;
initialiser(z);
...
```