Méthodologie pour la Programmation

Master EEA - UMSIE 131

Pascal Giorgi

LIRMM - Université Montpellier 2 - CNRS

Objectif de ce cours

Principalement

Acquérir une méthodologie pour programmer de manière simple, efficace et claire des algorithmes dans un langage de programmation donné.

Introduction

Qualités minimales d'un programme

1. **lisible** : comportement et mise en œuvre doivent être compréhensibles ;
2. **maintenable** : il doit être facile de localiser des erreurs et de les corriger ; il doit être facile de modifier une partie de l'implémentation sans toucher au reste ;

Introduction

Objectifs plus ambitieux

Il peut être :

1. **portable** : on peut l'adapter à un nouvel environnement (par ex un passage UNIX vers WINDOWS) ;
2. **extensible** : on peut ajouter des fonctionnalités sans modifier le comportement de l'ensemble ;
3. **réutilisable** : on peut s'en resservir dans un autre contexte (si c'est un morceau de programme : on peut l'assembler avec d'autres morceaux que ceux d'origine).

S'il doit l'être, mieux vaut y penser dès le début.

Introduction

Structurer et organiser

Pour avoir au moins les qualités minimales, il faut structurer le code qu'on produit, et ne pas le faire n'importe comment :

- a) la structuration doit avoir un sens pour le programme (ou l'application),
- b) elle doit être visible,
- c) elle doit être compréhensible pour quelqu'un d'autre,
- d) elle doit être modulaire.

Niveaux de structuration

Il y en a plusieurs :

1. obligatoire : structuration du code du programme ;
2. dès que le programme est un peu gros : structuration du stockage (un unique fichier ne suffit plus) ;
3. si on veut conserver plusieurs versions d'un gros programme ou d'un ensemble de programmes : utilisation simple d'un gestionnaire de versions ;
4. si on travaille à plusieurs : une utilisation d'un gestionnaire de travail collaboratif. Une gestionnaire de version peut suffir pour organiser la mise en commun du travail des collaborateurs.

Algorithme

Définition (*informelle*)

Schéma de calcul, forme déjà modélisée de la solution, non encore écrite dans un langage de programmation.

Exemple : somme des éléments d'un tableau T de taille n

```
som ← 0 ;
pour i allant de 1 a n faire
 som← som+T[i] ;
fin pour
```

Organiser les calculs : les fonctions

Fonctions ou « Sous-algorithme »

Dès qu'un algorithme dépasse quelques lignes, il est nécessaire de l'organiser, de le structurer autrement qu'en alignant les actions les unes après les autres.

Attention :
dans le cas général, une bonne *modélisation* d'un problème à résoudre permet d'identifier les différents morceaux logiques de la solution complète, et les articulations entre eux. Une phase préalable sur papier est nécessaire.

Ce qu'on décrit ici se situe à un niveau simple, où la modélisation et l'écriture des algorithmes sont faciles. Le problème est de les traduire techniquement.

Notion de fonctions ou de sous-algorithmes

Besoin de structure

La façon la plus simple en apparence de structurer un algorithme consiste à le découper en morceaux “naturels”.

Exemple

si on veut ranger des valeurs dans un tableau, les afficher, puis faire des calculs dessus, on a un morceau “saisie des valeurs”, un morceau “affichage des valeurs”, et au moins un morceau “calculs”, peut-être plusieurs s'il y a plusieurs calculs distincts à faire.

Pas simple, car pour avoir une bonne organisation, il faut bien délimiter les morceaux : logiquement homogènes, pas trop gros ou trop petits.

Notion de fonctions ou de sous-algorithmes

Découpage en morceaux

Ces morceaux sont appelés *sous-algorithmes*, et on donne un nom particulier à chacun, correspondant à une fonction en programmation.

On décrit chaque sous-algorithme séparément, et on décrit l'algorithme complet (*algorithme principal*) en réutilisant les fonction correspondantes.

Algorithme *vs* sous-algorithme

un sous-algorithme n'est ni plus ni moins qu'un algorithme qui est réutilisé à plus haut niveau dans un autre algorithme.

Remarque

On peut, à l'intérieur d'un sous-algorithme, refaire un découpage en morceaux. Un calcul forme un tout logique qui se décompose en plusieurs étapes : chaque étape correspond à un sous-algorithme et porte un nom permettant de le réutiliser.

Sous-algorithme avec paramètres

Découverte expérimentale

Quand on découpe certains algorithmes en sous-algorithmes, on s'aperçoit très vite qu'on obtient des morceaux qui se ressemblent beaucoup :

par exemple, "afficher les valeurs d'un tableau x de n entiers" et "afficher les valeurs d'un tableau z de p entiers" ont des chances d'être identiques, sauf pour le nom du tableau et la taille.

Nécessité d'introduire des algorithmes paramétrés

On choisit des noms abstraits pour représenter les futurs variables.
ex : le nom du tableau et de la taille. On parle de *paramètres formels*.

On écrit un unique sous-algorithme en employant ces noms-là.
Dans chaque appel du sous-algorithme, on indique les valeurs que vont prendre les paramètres formels. Ces paramètres s'appellent des *paramètres effectifs*.

Sous-algorithme avec paramètres

Exemple

afficher les valeurs d'un tableau x de n entiers
afficher les valeurs d'un tableau z de p entiers

Algorithme : `afficheTabInt(T, k)`

```
pour i allant de 1 à k faire
 afficher(T[i]) ;
fin pour
fin algorithme

afficheTabInt(x,n)
afficheTabInt(z,p)
```

Ce système de paramètres est très pratique, mais limité : pour chaque paramètre formel, on indique un type que les paramètres effectifs correspondants doivent respecter.

Qu'est-ce qu'un paramètre en réalité ?

Définition (informelle)

C'est un moyen de communication entre un sous-algorithme et son appelant (algorithme principal ou autre sous-algorithme).

Exemple : Que faut-il communiquer ?

- lecture d'un tableau : l'appelant fournit un tableau "vide" et sa taille ; le sous-algorithme rend le tableau rempli.
- calcul de la somme des éléments d'un tableau : l'appelant fournit un tableau "plein" et sa taille ; le sous-algorithme renvoie la somme calculée.
- affichage d'un tableau : l'appelant fournit un tableau "plein" et sa taille ; le sous-algorithme n'a rien à communiquer à son appelant.

Le nombre des informations à faire passer dans un sens ou dans l'autre n'est théoriquement pas limité (ce n'est pas vrai en pratique).

Méthodes pour une bonne utilisation des paramètres

Réflexion *a priori*

Quand on pense avoir un bon « candidat » comme sous-algorithme, il faut identifier les paramètres qui permettront d'interagir avec son appelant.

Examiner les variables du candidat

1. la variable est-elle supposée remplie au début du sous-algorithme ?
2. la valeur qu'elle contient à la fin du sous-algorithme doit-elle être réutilisée ?

Si la réponse est :

- *non aux deux* : ce n'est pas un paramètre, c'est une *variable locale* ;
- *oui aux deux* : c'est un paramètre, qui communique dans les deux sens ;
- *oui à 1)* : c'est un paramètre, que l'appelant fournit au sous-algorithme ;
- *oui à 2)* : c'est un paramètre, ou le *résultat* renvoyé par le sous-algorithme.

Méthodes pour une bonne utilisation des paramètres

Après examen de toutes les variables : compter les paramètres

- aucun paramètre : en général sous-algorithme pas intéressant sauf pour faire une macro (ex. règles d'un jeu, ...)
- beaucoup (plus de 5 ou 6) : sous-algorithme trop général ou assemblage de cas différents (paramètres trop contextuel)
  $\hookrightarrow$ séparer les cas possibles dans plusieurs sous-algorithmes

Éliminer si possibles les paramètres partitionneurs de code

Exemple : un sous-algorithme *extremum* qui reçoit un tableau, sa taille, et un nombre entier , et qui calcule le maximum du tableau si le nombre entier vaut 1, son minimum si le nombre entier vaut 2.

Mieux vaut écrire à la place deux sous-algorithmes *maximum* et *minimum*.

Méthodes pour une bonne utilisation des paramètres

Communication du résultat

Au lieu d'utiliser un paramètre pour communiquer cette valeur, le sous-algorithme peut la *retourner* comme résultat à l'appelant qui devra la *récupérer*.
Il s'agit toujours pour le sous-algorithme de transmettre une information à son appelant, c'est juste la méthode employée qui diffère[a]. Le choix de l'une ou l'autre méthode est affaire de commodité.

[a] Imaginez que quelqu'un vous a demandé de faire un achat pour lui et que vous lui apportez l'objet : vous pouvez le laisser dans sa boite aux lettres ou le lui donner directement.

Méthodes pour une bonne utilisation des paramètres

Établir les préconditions des sous-algorithmes

Les *préconditions* décrivent les contraintes sur les paramètres pour que le sous-algorithme est un comportement cohérent et correct.

Ex : " affichage d'un tableau".
L'entier n donné comme taille doit être strictement positif, Le tableau doit être de taille au moins égale à n et avoir été rempli jusqu'à la case d'indice n-1.

Le sous-algorithme ne vérifie pas tout cela, ce sont des conditions implicites.

Établir les postconditions des sous-algorithmes

Les *postconditions* décrivent la situation après l'exécution du sous-algorithme, ie ce que l'appelant va trouver et qui n'est pas entièrement exprimé par les paramètres et l'éventuel retour.

Ex : " saisie d'un tableau" : Les cases d'indice 0 à n-1 sont remplies.

Algorithme contenant des sous-algorithmes

Notion de paramètre modifié ou non modifié

Un paramètre est dit *modifié* dans un sous-algorithme si on lui affecte une nouvelle valeur qui doit être transmise à son appelant.

Un paramètre est *non modifié* s'il n'est pas modifié par le sous-algorithme.

Attention : toutes modifications sur un paramètre ne signifie pas que celui-ci doit être forcement définit comme modifiable. C'est la communication à l'appelant de cette modification qui le rend nécessaire.

Algorithme contenant des sous-algorithmes

Syntaxe

- **décrire (déclarer) un sous-algorithme**
  sous-algorithme `identificateur`
  paramètres non modifiés : liste de descriptions de paramètres formels
  paramètres modifiés : liste de descriptions de paramètres formels
  valeur retournée : description de la valeur ou *rien*
  { actions, et déclarations de variables utilisées uniquement dans ces actions }
- **utiliser (appeler) un sous-algorithme**
  - si l'algorithme retourne une valeur :
    `variable ← identificateur(liste de paramètres effectifs) ;`
  - s'il ne retourne rien :
    `identificateur(liste de paramètres effectifs) ;`

Algorithme contenant des sous-algorithmes

Exemple : somme des éléments d'un tableau

Déclaration

sous-algorithme somme
  paramètres non modifiés : n entier, t tableau de n réels
  valeur retournée : réel
  {entier i ; réel s ;
  s←0 ;
  pour i←0 à n-1
    s←s+ t[i] ;
  retourner s ;
}

Utilisation

algorithme principal
  {som : un réels ;
   t : un tableau de 8 réels ;
  ...
   som←somme(8,t) ;
}

Programme avec fonctions en C

Les fonctions

Chaque fonction est la traduction en C d'un sous-algorithme.

Déclaration

La déclaration d'une fonction se compose :

1. d'une *signature* (ou *en-tête* ou *prototype*) qui comprend le nom de la fonction, la description de ce qu'elle retourne, et la description de ses paramètres ;
2. d'un *corps* qui contient la traduction des déclarations et actions du-sous algorithme.

Programme avec fonctions en C

syntaxe

```
 type-retour ident(liste des paramètres formels)
{
    déclarations des variables
    instructions
}
```

signature de la fonction

1. *ident* est le nom de la fonction ;
2. Le *type-retour* est
   - un des types de base ou un des types construits par l'utilisateur
   - `void` si la fonction ne retourne rien.
3. *déclaration des paramètres formels* :
   - *type identificateur* si le paramètre est une variable simple nom modifiée ;
   - *type * identificateur* si le paramètre est une variable simple modifiée ;
   - *type identificateur* `[n]` si le paramètre est un tableau à `n` valeurs ;

   où *type* est un type quelconque connu du compilateur

Programme avec fonctions en C

syntaxe

```
 type-retour ident(liste des paramètres formels)
{
    déclarations des variables
    instructions
}
```

corps de la fonction

1. *déclarations de variables* annoncent les variables **locales** à la fonction.
2. *instructions* :
   - on peut trouver toutes les instructions habituelles (`for, while,...`),
   - si la fonction retourne quelque chose, on **doit** trouver au moins une fois l'instruction : `return` *expression* ;
   - si la fonction ne retourne rien, on **ne doit jamais** trouver `return` *expression* ;

Programme avec fonctions en C

ATTENTION

La déclaration d'une fonction est une description formelle d'un calcul, elle n'exécute rien toute seule : il faut *appeler* la fonction pour que les instructions du corps soient effectuées.

Appel de fonctions en C

L'appel

C'est l'instruction qui lance une exécution du corps de la fonction après avoir donné les "vrais" paramètres à utiliser.

syntaxe

Un appel de fonction est identique à un appel de sous-algorithme :

- si la fonction retourne une valeur :

```
variable = ident(liste de paramètres effectifs) ;
```

- si elle ne retourne rien :

```
ident(liste de paramètres effectifs) ;
```

- Un *paramètre effectif* est la "vraie" variable ou valeur à substituer au paramètre formel correspondant.

Appel de fonctions en C

Caractéristiques des paramètres effectifs

Un paramètre effectif « non modifié » peut être :

- une variable simple,
- un élément de tableau,
- une expression,
- un tableau

ayant le type exact du paramètre formel ou un type compatible.

Un paramètre effectif « modifié » peut être :

- une variable simple → syntaxe : `&` *identificateur*
- un élément de tableau → syntaxe : `&` *identificateur* `[entier]`

ayant le type exact du paramètre formel

Un paramètre effectif correspondant à un paramètre formel de type tableau *doit* être un identificateur de tableau dont les éléments ont exactement le type indiqué dans le paramètre formel.

Appel de fonctions en C

Ordre de déclaration des fonctions

Lorsque le corps d'une fonction contient un appel à une autre fonction, la déclaration de la fonction appelée doit être placée avant celle de la fonction appelante.

La signature de la fonction est suffisante : *le compilateur vérifie uniquement la cohérence syntaxique des appels*

Méthode

1. placer toutes les signatures, chacune suivie d'un point-virgule, en début de fichier
2. accompagner chaque signature d'un commentaire décrivant la fonction
3. écrire toutes les fonctions complètes (signature + corps)

Exemples

tri par sélection-échange

```
#include<stdio.h>

/*******************signatures des fonctions***************/
void saisie(int n,int t[]);

void affiche(int n,int t[]);

void ech(int *a, int *b );

int indiceDuMin(int t[], int debut, int fin);
/*  t, debut, fin param non modifie
    suppose que  debut < fin;
    retourne l'indice du premier element de plus petite valeur qui
    se trouve dans le tableau t entre debut et fin inclus*/
/*-----------------------------------------------------*/

void triSelectionEchange(int n,int t[]);
```

Exemples

tri par sélection-échange

```
/**********************code complet*******************************/
void saisie(int n,int t[])
 {printf( "valeurs a trier\n");
  for (int i=0; i<n; i++)  {scanf("%d",&t[i]);}}

void ech(int *a, int *b )
 {int c; c=*a;*a=*b;*b=c;}

int indiceDuMin(int t[], int debut, int fin)
 {int imin=debut;
  for (int i=debut+1; i<=fin; i++)
    if (t[i]<t[imin]) imin=i;
  return imin;}

void triSelectionEchange(int n,int t[])
 {int k;
  for (int i=0; i<=n-2; i++)
    {k=indiceDuMin(t,i,n-1);
     if (k != i) ech(&t[k],&t[i]);
    }
}
```

Exemples

tri par sélection-échange

```
/**********************programme principal*****************************/

int main()
{
  int t[8]
  saisie(8,t);
  affiche(8,t);
  triSelectionEchange(8,t);
  printf("apres le tri\n");
  affiche(8,t);
return 0;
}
```

Attention au faux amis

Algorithme du calcul de pgcd

sous-algorithme pgcd
  paramètres non modifiés : a,b entiers
  valeur retournée : entier
{entier r ;
 si (b=0) alors
      retourner a ;
 sinon
      r←a modulo b ;
      tant que (r ≠ 0)
            { a←b ;b←r ; r←a modulo b ;}
      retourner b ;
}

faux-amis

Les paramètres a et b sont déclarés non-modifiés mais l'algorithme les modifie ! ! !

Attention au faux amis

paramètres non modifiés signifie :

quelles que soient les instructions contenues dans le sous-algorithme, les paramètres effectifs fournis lors de l'appel sont a la sortie de l'algorithme dans le même état qu'à l'entrée.

souvent implicite au passage de paramètres dans les langages de programmation

pgcd en C

```
#include <stdio.h>
int pgcd(int a, int b)
{int r;
 if (b==0)
   return a;
 else
 {r=a%b;
  while (r!=0)
   { a=b; b=r; r=a%b;}
  return b;
 }
}
```

Les fonctions en Matlab

Les fonctions

Chaque fonction est la traduction en langage Matlab d'un sous-algorithme.

Définition

La définition d'une fonction Matlab se compose d'une déclaration comprenant :

1. le nom de la fonction
2. la description de ses paramètres
3. la description de ce qu'elle retourne
4. un *script* contenant la traduction du sous algorithme.

Les fonctions en Matlab

syntaxe

```
function argout = ident(argin)
   % déclarations des commentaires d'aide
   script de la fonction
end
```

Paramètre de la fonction

1. *ident* est le nom de la fonction
2. *argout* est la ou les variables renvoyées par la fonction
3. *argin* est la liste des variables d'entrée de la fonction

Script de la fonction

correspond au code de la fonction qui travaille sur les paramètres formels (argout, argin).

Appel de fonctions en Matlab

L'appel

C'est l'instruction qui lance une exécution du script de la fonction après avoir donné les "vrais" paramètres à utiliser.

syntaxe

Un appel de fonction est identique à un appel de sous-algorithme :

- si la fonction retourne une valeur :

```
variable = ident(liste de paramètres effectifs)
```

- si elle ne retourne rien :

```
ident(liste de paramètres effectifs)
```

- Un *paramètre effectif* est la "vraie" variable ou valeur à substituer au paramètre formel correspondant.

Appel de fonctions en Matlab

Caractéristiques des paramètres effectifs

Un paramètre effectif peut être :

- une constante,
- une variable scalaire,
- un tableau ou une matrice,
- une fonction ou une commande.

Attention :

Les variables d'entrée d'une fonction Matlab ne sont pas modifiables ! ! !

Appel de fonctions en Matlab

Ordre de déclaration des fonctions

Lorsque une fonction est appelée en Matlab il faut que :

- elle soit définie dans l'environnement courant

  ou

- elle soit accessible dans une fichier de commande "M-File"

Exemples

tri par sélection-échange (M-FILE)

```
function A = triSelectionEchange(T)
% tri du tableau par selection echange
 n=length(T);
 for i=1:n
    k=indiceDuMin(T,i,n);
    if k ~= i
        A(i)=T(k);
        T(k)=T(i);
    else
        A(i)=T(i);
    end
 end

function imin = indiceDuMin(T,deb,fin)
% renvoi l'indice du plus petit elt entre T(deb) et T(fin)
n=length(T);
imin=deb;
 for i=deb+1:fin
    if T(i)<T(imin)
        imin=i;
    end
 end
```

Exemples

tri par sélection-échange (script Matlab)

```
T=[1 5 2 3 6 8 4 9 7]
A=triSelectionEchange(T)
```

Attention au faux amis

paramètres non modifiés signifie :

quelles que soient les instructions contenues dans le sous-algorithme, les paramètres effectifs fournis lors de l'appel sont a la sortie de l'algorithme dans le même état qu'à l'entrée.

souvent implicite au passage de paramètres dans les langages de programmation

la fonction triSelectionEchange en Matlab

```
function A = triSelectionEchange(T)
% tri du tableau par selection echange
 n=length(T);
 for i=1:n
    k=indiceDuMin(T,i,n);
    if k ~= i
        A(i)=T(k);
        T(k)=T(i);
    else
        A(i)=T(i);
    end
 end
```

Structurer les données : les définitions de types

Le problème

Un langage de programmation propose un certain nombre de types prédéfinis pour les variables. On les appelles types de base : entiers, réels, caractères, et les tableaux à une ou plusieurs dimensions d'éléments d'un de ces types-là.

Bon nombre de variables en algorithmique sont plus complexes : matrices, vecteurs, polynômes, listes, file d'attente, images, ... et ne sont pas naturellement exprimées dans le langage.

La solution

Certain langage offre la possibilité de construire ses propres structures de données.

Les structures de données en C : struct

Syntaxe

`struct` *typeident*

```
  {
    déclaration de champ ;
    déclaration de champ ;
    ...
    déclaration de champ ;
  } ;
```

où *typeident* est un identificateur
et *déclaration de champ* a une syntaxe identique à une déclaration de variable.

Attention : cette définition ne réserve aucune place mémoire : ce n'est pas une déclaration de variable

Les structures de données en C : struct

Exemple

Pour pouvoir représenter une entité *place de spectacle*, on définit :

```
struct placeSpect
  {
   char rangee;
   int numero;
  };
```

Pour pouvoir représenter une entité *objet en stock*, on définit :

```
struct objSto
  {
   int ref;
   float prixU;
  };
```

Les structures de données en C : struct

Signification

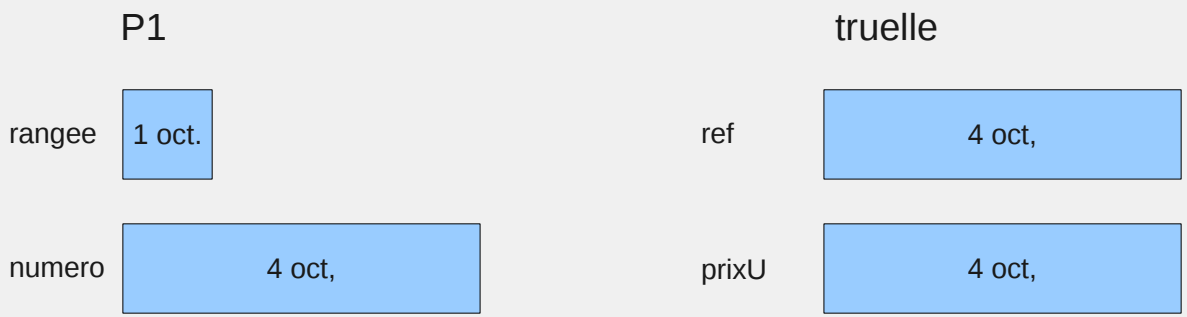
On décrit comment est composée l'entité qu'on aimerait représenter, en donnant la liste de tous ses morceaux (il peut y en avoir plus de deux), et on donne un nom global à l'ensemble.

Ce nom `struct typeident` est un nouveau type de donnée qui s'ajoute aux autres types connus dans le programme.

Utilisation d'une struct

Déclaration de variable

Exemples : `struct placeSpect p1 ;` `struct objSto truelle ;`
Une variable de type *struct machin* est composée de plusieurs parties, correspondant dans l'ordre aux déclarations de champs :



Remarque : la place réellement occupée par la struct en mémoire peut être plus grande que la somme des tailles des champs. L'expression `sizeof(struct` *ident*`)` renvoie la taille effective en nombre d'octets.

Utilisation d'une struct

Accès aux champs d'une variable de type *struct*

On identifie chaque champ par son nom propre et celui de la variable, en utilisant la notation pointée :

- `p1` contient les deux champs `p1.rangee` et `p1.numero`
- `truelle` contient les deux champs `truelle.ref` et `truelle.prixU`
- si on déclare `struct objSto pot ;` `pot` contient les deux champs `pot.ref` et `pot.prixU`

Cette notation *nomvariable.nomchamp* permet de désigner un champ d'une variable sans ambigüité.

Utilisation d'une struct

Affectation, lecture, écriture, test

Si on a déclaré `struct objSto truelle, pot ;` on peut écrire :

```
scanf("%d%f",&truelle.ref, &truelle.prixU);
printf("%d %f \n",truelle.ref, truelle.prixU);
pot.ref=45033;  pot.prixU=13.5;
if (truelle.ref== ...)
if (truelle.prixU >=10) ...
```

On travaille séparément sur chaque champ, exactement comme on travaillerait sur une variable simple de même type que le champ[a].

[a] Si on veut affecter à une variable de type struct les valeurs d'une autre, l'affectation directe entre variables est possible. Par exemple : `struct objSto truellebis ; truellebis=truelle ;`

Utilisation d'une struct

Exemples avec champs de même type

La première raison d'être d'une *struct* est de regrouper des champs de types différents. Utilise également pour regrouper des champs de types identiques.
Exemples :

- une place dans un train est identifiée par trois entiers :
  un numéro de train, un numéro de voiture, un numéro de place ;
- une date est identifiée par trois entiers : jour, mois, année.

Avec les définitions :

```
struct placeTrain {
  int train ;
  int voiture ;
  int place ;
} ;
```

```
struct date {
  int jour ;
  int mois ;
  int an ;
} ;
```

Mieux que d'utiliser des tableaux à 3 élts : variables nommées et non-ordonnées

Fonction et struct

Paramètre formel de type struct

Identique à celle utilisée sur les types de base

```
void f(struct objSto obj)
  {...corps de la fonction.....}
```

Type struct en retour de fonction

Identique à celle utilisée sur les types de base

```
struct objSto f(...
  {...corps de la fonction.....}
```

Exemple

Exemple pour un objet en stock

```
void saisie (struct objSto *x)
  {scanf("%d%f",&(*x).ref,&(*x).prixU);
  }

void affiche(struct objSto x)
  {printf("ref: %5d   prix unitaire: %10.3f ", x.ref,x.prixU);
  }

void saisieTobj(int n,struct objSto t[])
 {printf( "Entrer les elements separes par espace ou retour\n");
  for (int i=0; i<n; i++)  scanf("%d%f",&t[i].ref,&t[i].prixU);
 }

void afficheTobj(int n,struct objSto t[])
 {for (int i=0; i<n; i++)
      printf("ref: %5d   prix unitaire: %10.3f \n", t[i].ref,t[i].prixU);
  printf("\n");
 }
```

Les données en Matlab

par défaut il n'y a aucun typage des données en Matlab, on peut donc écrire :

```
x=32
y=1.5
y=2*x+i*y
```

Il n'y a même pas de différence entre données scalaire et multidimensionnelles

```
x=32
a=[1,2,3]
b=[1 ;2 ;3]
```

Les données sont dynamiques en Matlab ! ! !

- Le type d'une donnée est dynamique
- La dimension d'une donnée est dynamique

Les matrices et Matlab

Matlab est un langage matriciel ! ! !

Toutes les données sont des matrices :

- une donnée scalaire correspond à une matrice 1×1
- un tableau de dimension n correspond à une matrice $n \times 1$
- un vecteur de dimension n correspond à une matrice $1 \times n$

Les matrices et Matlab

Déclaration de données scalaires (x non déclaré) :

```
x=12
x(1)=12
x(1,1)=12
```

Déclaration de données vectorielles (v non déclaré) :

```
v=[1,2]
v(1)=1, v(2)=2
v(1,1)=1, v(1,2)=2
```

Les structures en Matlab

Regroupement de données

Comme en C, les structures Matlab permettent de regrouper des données afférentes à une même entité.

Exemple : simulation de la chute d'une balle

On veut regrouper :

- le poids de la balle : un scalaire
- sa position dans l'espace : un vecteur
- sa vitesse : un vecteur

Les structures en Matlab

Aucune définition de modèle

La structure se construit de manière *ad-hoc* lors de la déclaration de la variable

Syntaxe pour déclarer une variable `X` structurée en n variables

```
X.var1= val1
X.var1= val2
...
X.varn= valn
```

- var1,...,var1 représentent les noms des champs de la structure
- val1,...,valn correspondent aux valeurs attribué au champ de la variable X

Exemple de données structurées en Matlab

Exemple : simulation de la chute d'une balle

On veut regrouper :

- le poids de la balle : un scalaire
- sa position dans l'espace : un vecteur
- sa vitesse : un vecteur

code Matlab pour déclarer la variable `balle`

```
balle.poids = 10 ;
balle.position = [0, 0, 100] ;
balle.vitesse = [0, 0, 0] ;
```

Les structures en Matlab

Accéder aux champs d'une variable structurée

Comme en C, on fait appel à l'opérateur point : var.champ
L'accès est en lecture et en écriture, comme toutes variables Matlab.

Exemple : simulation de la chute d'une balle

```
balle.poids= balle.poids+15
balle.position= balle.position+[ 0, 10, 0]
```

Les structures en Matlab

Extension des variables structurées

Du fait de la structuration dynamique des variables en Matlab, on peut étendre une variable structurée en ajoutant de nouveaux champs.

Exemple : simulation de la chute d'une balle

On peut rajouter un champs pour le rayon de la balle :

```
balle.rayon=2
```

Tableau de structures en Matlab

Collection de données structurées homogènes

Matlab offre la possibilité de regrouper « facilement » des données ayant la même structure au sein d'un tableau. On peut voir cela comme une collection de données.

Syntaxe : utilise la dynamique des tableaux Matlab

Il suffit d'étendre une variable scalaire structurée en un tableau de données structurées. Le meilleur moyen pour créer un tel tableau « proprement » est de faire des copies de la première donnée structurée.

Exemple : simulation de la chute d'une balle

Extension d'une donnée scalaire à un tableau

```
balle(2:3,1)=balle
```

Tableau de structures en Matlab

Accès aux données structurées du tableau

Comme les tableaux classique , on utiliser l'accesseur () : var(i)

Accès aux champs d'une donnée structurée dans un tableau

On combine l'accès à l'élément du tableau avec l'accès aux champs d'une variable structurée : var(i).champ

Accès à un champ de toutes les données structurées d'un tableau

On utilise l'opérateur point directement sur le tableau : var.champ
en retour on obtient la séquence ordonnée des valeurs du champ pour les données du tableau

Exemple de tableau de données structurées en Matlab

Exemple : simulation de la chute de plusieurs balles

```
balle.poids = 10 ;
balle.position = [0, 0, 100] ;
balle.vitesse = [0, 0, 0] ;

% création du tableau de balles
balle(2:3,1)=balle

% modification de la 2ème balle
balle(2).poids=20
balle(2).position = [100, 0, 100] ;
balle(2).vitesse = [0, 2, 0] ;

% calcul du poids total de toutes les balles
sum([balle.poids])
```

Organisation du programme

Méthode

Dans l'ordre :

1. définitions de types sans fonctions associées : type avec commentaire
2. définitions de types avec fonctions associées : type + signatures groupés, avec commentaire pour chacun
3. fonctions indépendantes : signatures + commentaires
4. fonctions complètes dans l'ordre des signatures

On a ainsi un fichier organisé et documenté.

Qualité du code

écriture des intructions et des déclarations

- `si` *condition* `alors` *instructions* `sinon` *recommencer à la condition* est une boucle... ;
- utiliser des variables globales est un risque d'erreur majeur ;
- les syntaxes permises par le langage mais illisibles sont à éviter (si `i` et `j` sont deux entiers, écrire `while(i && j){...}` au lieu de `while(i!=0 && j!=0){...}` ne sert qu'à obscurcir le texte) ;
- les imprévisibles aussi (`t[i++]=i ;` fonctionne différemment suivant les compilateurs ; remède : faire `i++ ;` avant ou après l'affectation, pas en même temps) ;
- les astuces inutiles aussi (`a=a+b ;b=a-b ;a=a-b ;` échange les valeurs de `a` et `b`...est-ce bien évident à la lecture ?) ;
- les erreurs d'arrondi peuvent couler un calcul (ex $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + ...$).

Qualité du code

Conventions de nommage

En choisir et s'y tenir.
Les usuelles (conventions de C, C++, Java, etc) :

- pas de caractères exotiques : lettres, chiffres, et éventuellement souligné (laisser les autres aux identificateurs internes du système) ;
- minuscule au début d'un nom de variable ou de fonction (*pas de* souligné, c'est une convention pour les identificateurs système) ;
- nom tout en majuscules pour les constantes ;
- séparation des mots dans un identificateur (vieux, sauf pour constantes : souligné, actuel : majuscule) ;
- noms "parlants" et pas trop proches les uns des autres (risque de fautes de frappe indécelables...halte aux noms de variables idiots : `d`,`dd`,`ddd`,`dddd`,...).

Conventions beaucoup plus précises pour la prog objet (C++, Java).