Introduction à la Programmation

Benoit Donnet

Année Académique 2023 - 2024

Agenda

- Introduction
- Chapitre 1: Bloc, Variable, Instruction Simple
- Chapitre 2: Structures de Contrôle
- Chapitre 3: Méthodologie de Développement
- Chapitre 4: Structures de Données
- Chapitre 5: Modularité du Code
- Chapitre 6: Pointeurs
- Chapitre 7: Allocation Dynamique

Agenda

- Chapitre 5: Structures de Données
  - Tableaux Uni-Dimensionnels
  - Tableaux Multi-Dimensionnels
  - Chaînes de Caractères
  - Enregistrements
  - Enumérations
  - Fichiers

Agenda

- Chapitre 5: Structures de Données
  - Tableaux Uni-Dimensionnels
    - ✓ Principe
    - ✓ Déclaration
    - ✓ Manipulation
    - ✓ Invariant Graphique
    - ✓ Exemple
    - ✓ Algorithmique
  - Tableaux Multi-Dimensionnels
  - Chaînes de Caractères
  - Enregistrements
  - Enumérations
  - Fichiers

Principe

- Soit le problème suivant
  - l'apparitorat FSA souhaite un programme permettant
    - ✓ d'encoder les notes d'un examen pour 20 étudiants
    - ✓ de calculer la moyenne des notes
    - ✓ de déterminer combien d'entre elles sont supérieures à la moyenne
- Comment faire?

Principe (2)

- 1ère tentative de solution
  - autant de variables que de notes à encoder

```
#include <stdio.h>

int main(){
  //déclaration
  unsigned short note1;
  //...
  unsigned short note20;

  //lecture au clavier
  printf("Entrez la note pour le 1er étudiant: ");
  scanf("%hu", &note1);
  //...
  printf("Entrez la note pour le 20è étudiant: ");
  scanf("%hu", &note20);
  //...
}//fin programme
```

Principe (3)

- Inconvénient(s) de cette solution?
  - impossible de faire une boucle
    - ✓ tout doit être traité séquentiellement
    - ✓ risque de multiplication des erreurs
  - difficilement gérable quand le nombre d'étudiants augmente
    - ✓ quid du Bloc 1 Droit/Psycho?

Principe (4)

- Il faudrait pouvoir stocker, en mémoire, toutes les notes sous un même "chapeau" et en même temps pouvoir accéder à chacune des notes séparément
- Solution
  - **tableau**
    - ✓ structure de données *homogène*
      - ensemble d'éléments de même type désignés par un identificateur unique
      - chaque élément est repéré par un **indice** donnant sa position au sein de l'ensemble

Principe (5)

- Plus formellement, un **vecteur** (ou tableau uni-dimensionnel) est une collection $[x_0, x_1, ..., x_{n-1}]$
  - de n variables x_i, avec $n>0$ dont chacune
    - ✓ possède le même type
    - ✓ est accessible sur base de son indice $i \in [0, n-1]$
- Les différents éléments composant le vecteur sont stockés de manière contigüe en mémoire

Déclaration

- Déclaration en C

Type des éléments du tableau
Taille du tableau (>0)
Initialisation de tout le tableau

```
[const] type id[taille] [= {val1, val2, ...}];
```

constante identifiant du tableau

Déclaration (2)

- Exemple 1

```
#include <stdio.h>

int main(){
  int tab[20];

  //suite des déclarations et du programme
}//fin programme
```

- Cette déclaration
  - réserve 20 emplacements mémoires (contigus)
  - pour des valeurs de type `int`

Déclaration (3)

- Exemple 2

```
#include <stdio.h>

int main(){
  const unsigned short N = 20;
  int tab[N];

  //suite des déclarations et du programme
}//fin programme
```

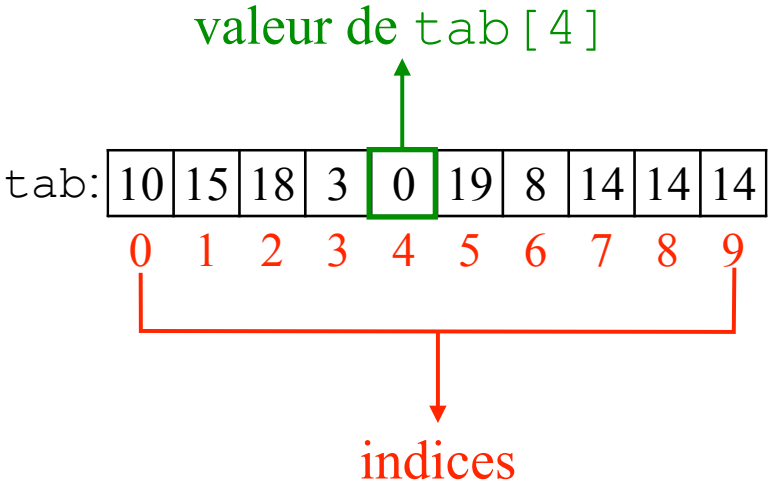
- Cette déclaration
  - réserve 20 emplacements mémoires (contigus)
  - pour des valeurs de type `int`

Manipulation

- Chaque élément est repéré par sa position dans le tableau
  - **indice**
    - ✓ placé entre crochets, après l'identifiant du tableau
    - ✓ le premier indice vaut toujours 0
  - `tab[0]` désigne le 1er élément du tableau, `tab[1]` le 2ème, ..., `tab[19]` le 20ème

Manipulation (2)

- Représentation *graphique* d'un tableau contenant 10 valeurs entières



Manipulation (3)

- Un élément de tableau est une valeur à gauche
  - il peut apparaître à gauche d'une affectation
    - ✓ `tab[3] = 5;`
  - il peut être l'opérande d'un opérateur d'incrémentation/ décrémentation
    - ✓ `tab[3]++;`
    - ✓ `--tab[2];`
- Une affectation globale est impossible
  - si `t1` et `t2` sont des tableaux d'entier
    - ✓ ? `t1 = t2;` ?

Manipulation (4)

- Un indice peut prendre la forme de n'importe quelle expression arithmétique de type entier
  - `tab[i-3]`
  - `tab[3*p-2*k+j%1]`
- Il en va de même si les indices sont de type `char`
- Quid si un indice va trop loin?
  - `int tab[10]; printf("%d\n",tab[25]);`
  - **débordement du tableau**
    - ✓ segmentation fault
  - il n'existe pas de contrôle automatique des limites du tableau
    - ✓ c'est au programmeur de faire attention

Invariant Graphique

- Rappel des règles pour un bon Invariant Graphique
  1. réaliser un dessin pertinent et le nommer
  2. placer sur le dessin les bornes de début et de fin
     - on peut aussi identifier la taille de la structure
  3. placer une (ou plusieurs) ligne(s) de démarcation qui sépare(nt) ce qui a déjà été calculé dans les itérations précédentes et ce qu'il reste à faire
  4. étiqueter chaque ligne de démarcation avec une variable d'itération
     - à gauche ou à droite
  5. décrire ce que les itérations précédentes ont déjà calculé en utilisant des variables
     - ces variables devront se retrouver dans le programme
     - questions à se poser
       - où est stocké ce résultat?
       - comment peut-on décrire ce résultat (forcément partiel)?
  6. identifier ce qu'il reste à faire dans les itérations suivantes
  7. toutes les structures et variables identifiées sont présentes dans le code.

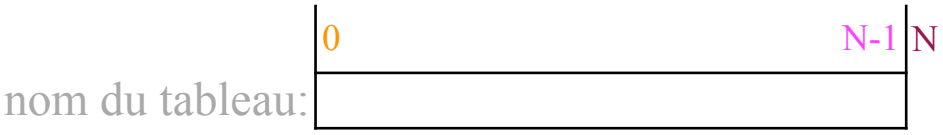
Invariant Graphique (2)

- Rappel du code couleur

Éléments du dessin	Code Couleur	Règle(s) Associée(s)
Nom de la structure		Règle 1
Borne minimale		Règle 2
Borne maximale		
Taille de la structure		
Lignes de démarcation		Règle 3
Étiquette des lignes de démarcation		Règle 4
Ce qui a été réalisée jusqu'à maintenant		Règle 5
zones "à faire"		Règle 6
Propriétés qui sont conservées		Règle 5 + Règle 6

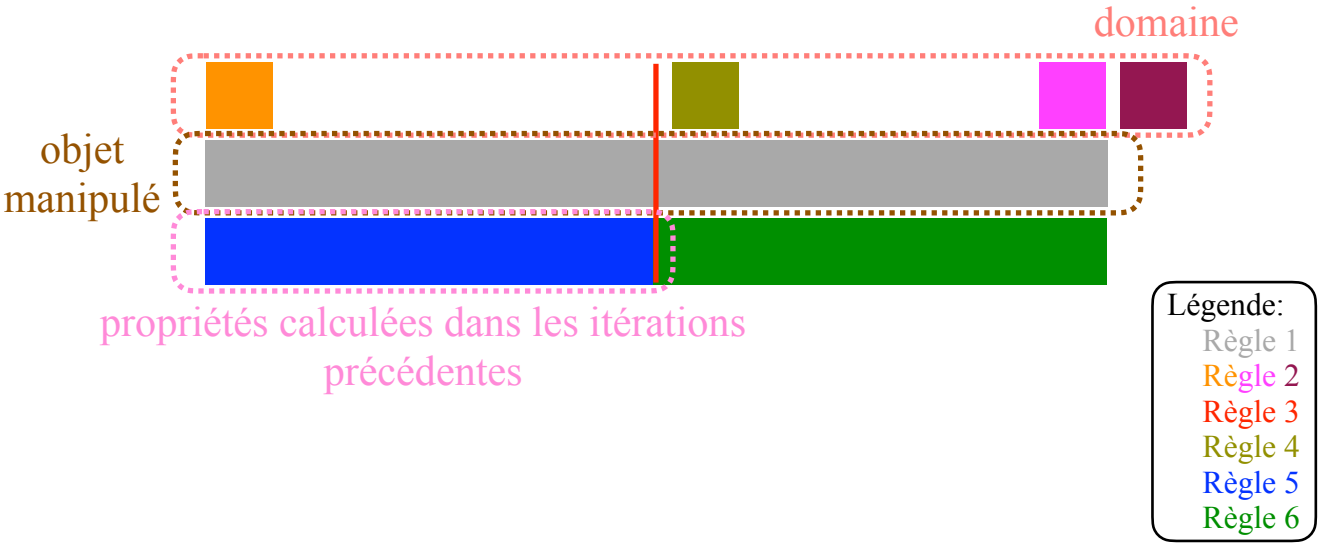
Invariant Graphique (3)

- Rectangle
  - permet la représentation d'un tableau de taille quelconque N
- Construction
  - chaque case représente la valeur à un indice donné
  - se lit de gauche à droite, peu importe le sens de manipulation du tableau
    - ✓ l'indice 0 se trouve à gauche
    - ✓ l'indice maximal se trouve à droite



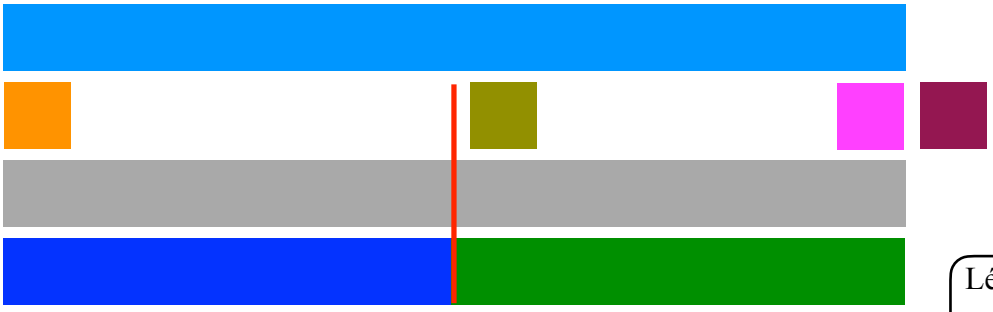
Invariant Graphique (4)

- Formats génériques d'un Invariant Graphique
  1. Invariant Graphique simple



Invariant Graphique (5)

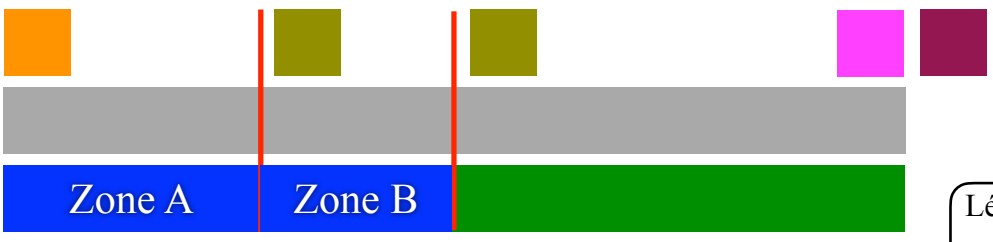
- Formats génériques d'un Invariant Graphique (cont(')
  2. Invariant Graphique simple avec propriété(s) conservée(s)



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Invariant Graphique (6)

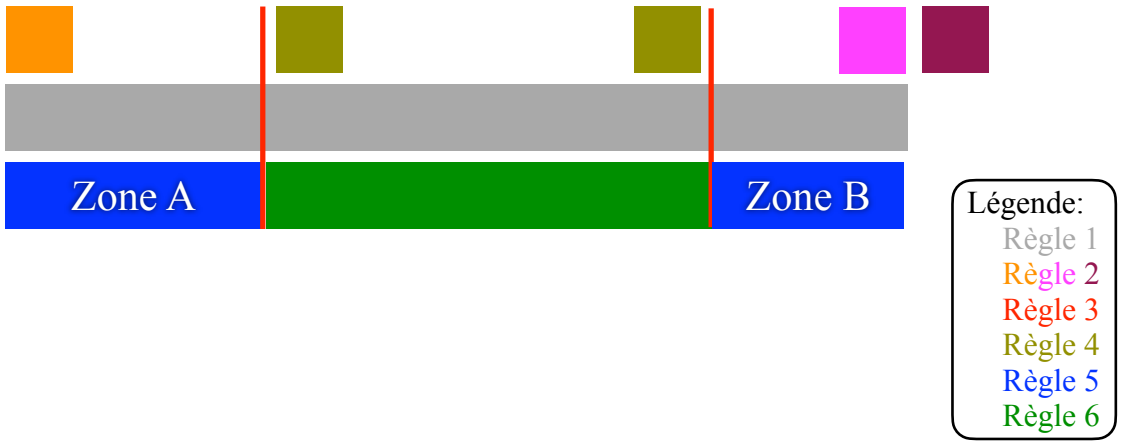
- Formats génériques d'un Invariant Graphique (cont(')
  3. Invariant Graphique simple avec plusieurs zones traitées



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Invariant Graphique (7)

- Formats génériques d'un Invariant Graphique (cont(')
  3. Invariant Graphique simple avec plusieurs zones traitées (cont')

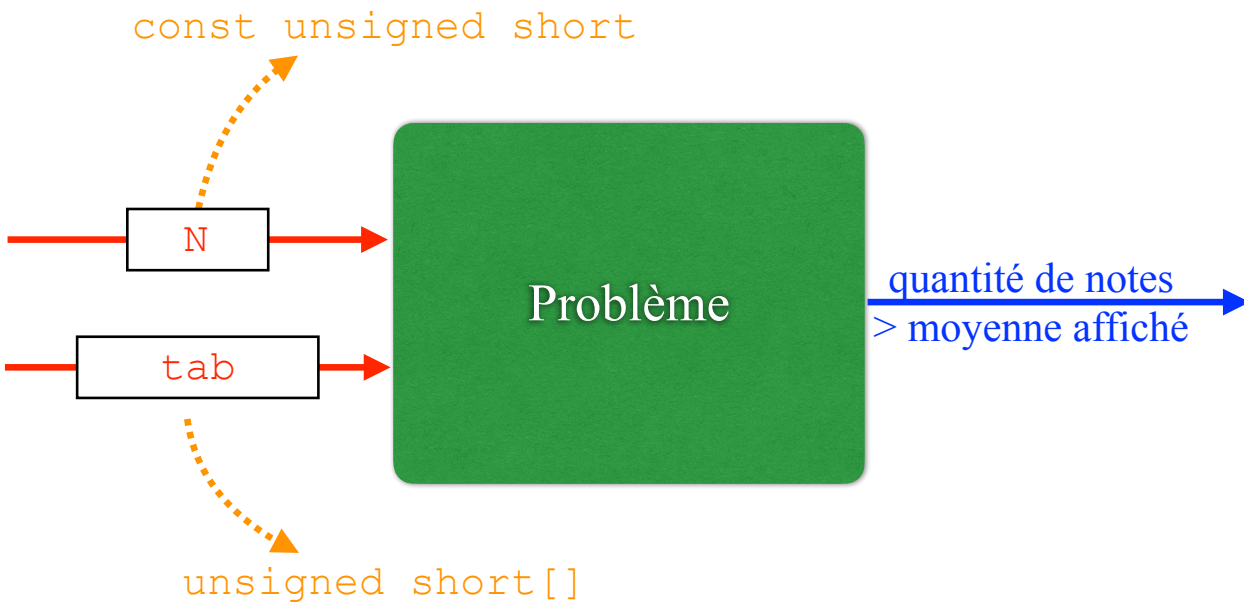


Exemple

- Revenons à notre problème concernant les 20 notes des étudiants
- Étape 1: définition du problème
  - Input
    - les notes des étudiants, lues au clavier
  - Output
    - quantité de notes supérieures à la moyenne
  - Caractérisation des Inputs
    - nombre de notes considérées
      - `const unsigned short N = 20;`
    - tableau de notes
      - `unsigned short tab[N];`

Exemple (2)

- Représentation graphique

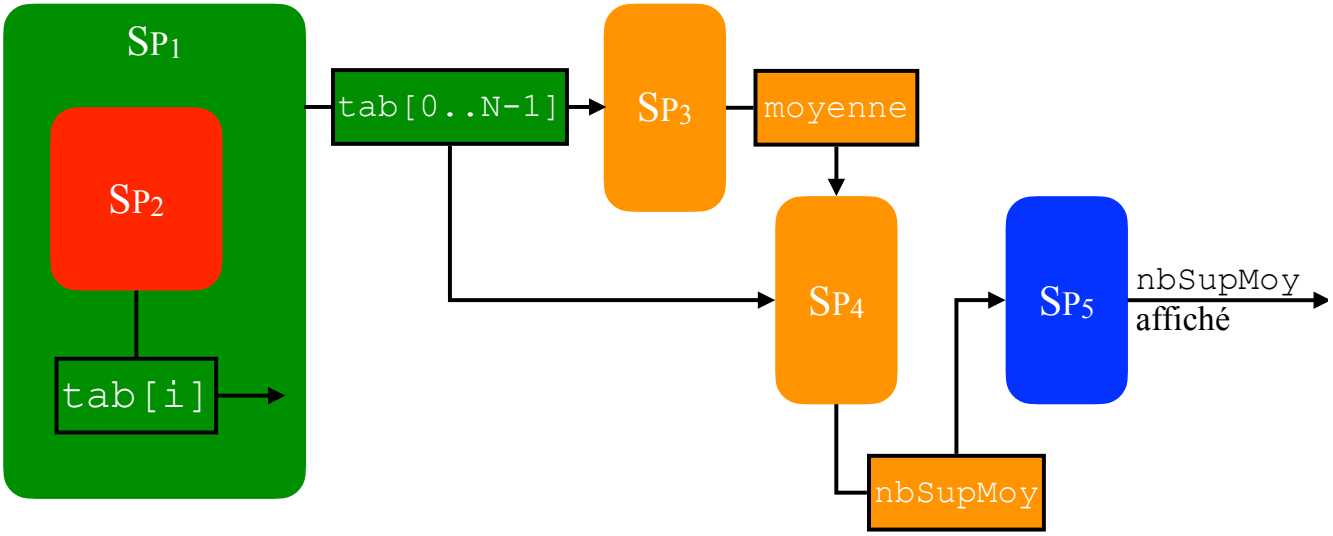


Exemple (3)

- Étape 2: analyse du problème
  - SP_1: énumération et encodage
    - ✓ énumération de tous les étudiants et encodage des notes dans `tab`
  - SP_2: lecture au clavier
    - ✓ lire au clavier la **`i`**ème note dans `tab[i]`
  - SP_3: calcul de moyenne
    - ✓ calcul de la moyenne des valeurs de **`tab`** dans `moyenne`
  - SP_4: calcul du nombre d'occurrences
    - ✓ calcul du nombre de valeurs de **`tab`** > **`moyenne`** dans `nbSupMoy`
  - SP_5: affichage
    - ✓ affichage de **`NbSupMoy`** sur la sortie standard

Exemple (4)

- Structuration des SPs
  - $(SP_2 \subset SP_1) \rightarrow SP_3 \rightarrow SP_4 \rightarrow SP_5$
- Représentation graphique



Exemple (5)

- Canevas général du code

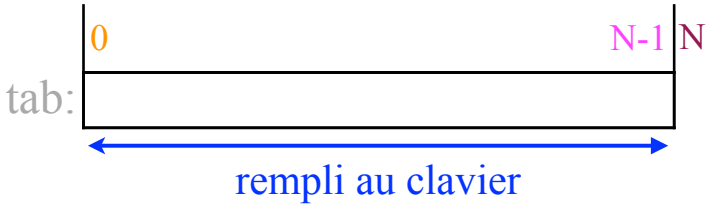
```
#include <stdio.h>

int main(){
  //déclaration des variables
  const unsigned short N = 20;
  unsigned short tab[N];

  //déclaration des variables additionnelles

  //SP1: encodage

  //SP2: calcul de la moyenne

  //SP3: calcul du nombre de notes supérieures à la moyenne
}//fin programme
```

Exemple (6)

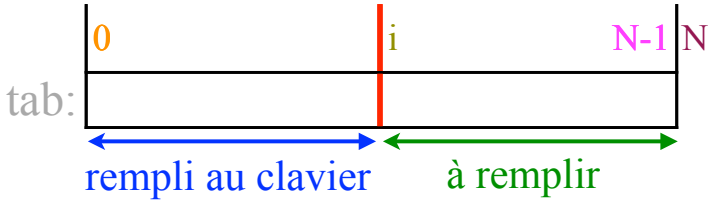
- Étape 3: écriture du code pour SP_1
- Définition
  - Input
    - ✓ /
  - Output
    - ✓ le tableau `tab` est rempli avec les `N` notes lues au clavier
  - Caractérisation de l'Input
    - ✓ /
- Idée de solution
  - demander à l'utilisateur d'entrer la note courante
    - ✓ SP_2!
  - répéter pour les `N` notes
- Présence d'une boucle
  - Invariant!

Exemple (7)

- Représentation graphique de l'Output du SP_1



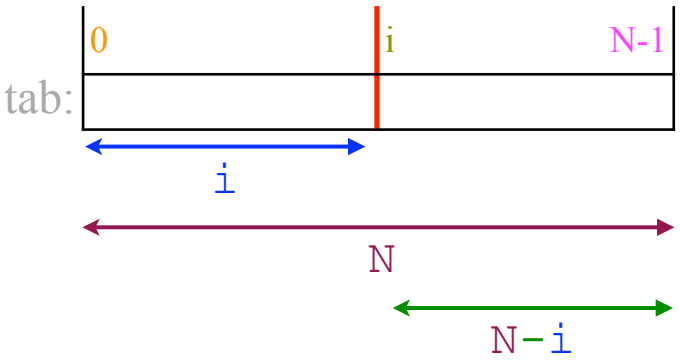
- Construction de l'Invariant Graphique pour le SP_1



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Exemple (8)

- On peut aisément dériver la Fonction de Terminaison de l'Invariant Graphique

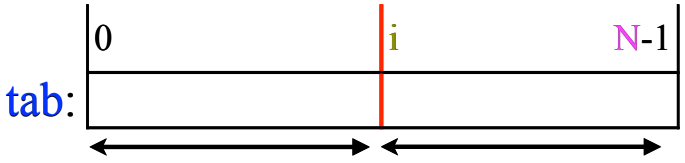


⇒ Fonction de Terminaison: `N-i`

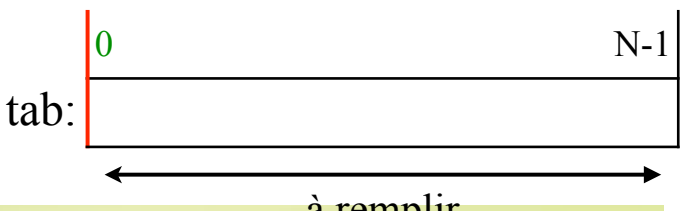
Exemple (9)

- Construction du code sur base de l'Invariant
  - construction de la *ZONE 1*
    - ✓ déclaration et initialisation des variables avant la boucle
      (1) quelles sont les variables dont j'ai besoin?
      (2) quelles sont les valeurs initiales de ces variables?
      (a) `tab, N`
      (b) `i`

```
unsigned short i;
i = 0;

//à suivre
```



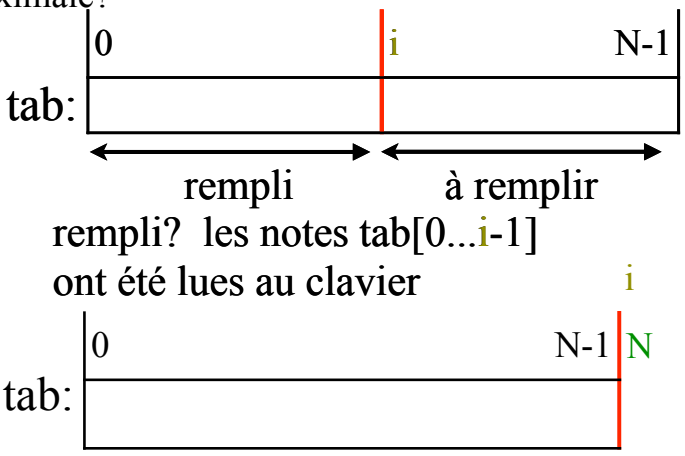
rempli? les notes tab[0...i-1] ont été lues au clavier



Exemple (10)

- Construction du code sur base de l'Invariant (cont.)
  - construction du Gardien de Boucle
    - ✓ variable(s) d'itération et valeur(s) maximale(s) donnent le Critère d'Arrêt
    - ✓ le Gardien est donné par la négation du Critère d'Arrêt
      (1) quelle est la variable d'itération?
      (2) quelle est sa valeur maximale?

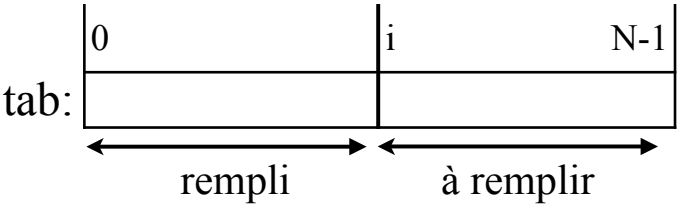
```
//ZONE 1

while(i<N){
  //à suivre
}//fin while - i

//à suivre
```



Exemple (11)

- Construction du code sur base de l'Invariant (cont.)
  - construction de la *ZONE 2*
  - construire le Corps de Boucle
    (1) l'Invariant est vrai
    (2) le Gardien de Boucle est vrai
    (3) dériver les instructions du Corps de Boucle

```
//ZONE 1

while(i<N){
  //à suivre
}//fin while - i

//à suivre
```



rempli? les notes tab[0...i-1]
ont été lues au clavier

Exemple (12)

- Construction du code sur base de l'Invariant (cont.)
  - construction de la *ZONE 2*
  - construire le Corps de Boucle

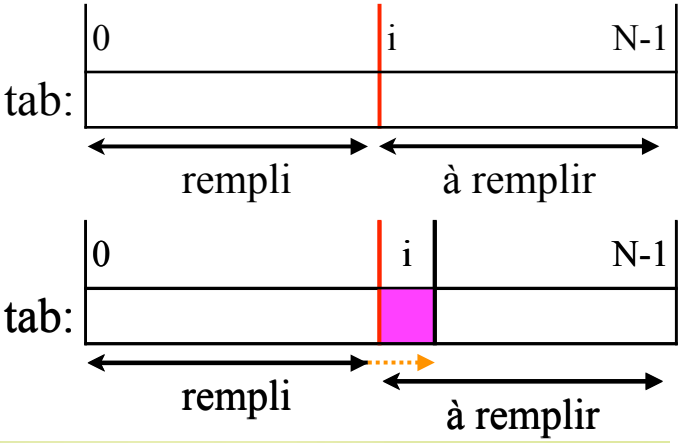
    (1) l'Invariant est vrai

    (2) le Gardien de Boucle est vrai

    (3) dériver les instructions du Corps de Boucle

      (a) lire au clavier

```
//ZONE 1

while(i<N){
  scanf("%hu", &tab[i]);
  //à suivre
}//fin while - i

//à suivre
```



Exemple (13)

- Construction du code sur base de l'Invariant (cont.)
  - construction de la *ZONE 2*
  - construire le Corps de Boucle

    (1) l'Invariant est vrai

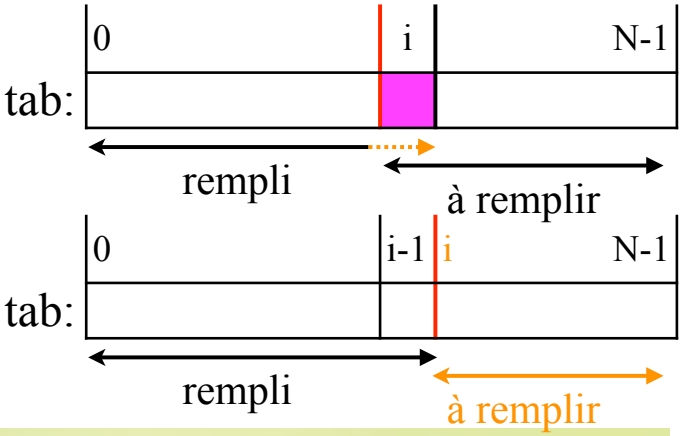
    (2) le Gardien de Boucle est vrai

    (3) dériver les instructions du Corps de Boucle

      (a) lire au clavier

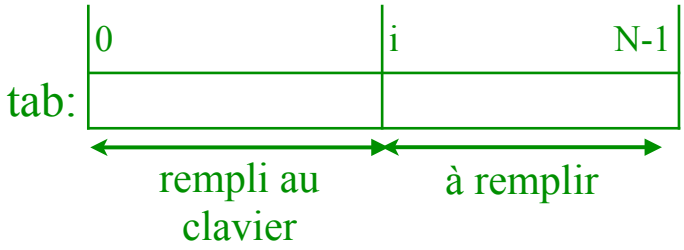
      (b) mettre à jour `i`

```
//ZONE 1

while(i<N){
  scanf("%hu", &tab[i]);
  i++;
}//fin while - i

//à suivre
```



Exemple (14)

- Pour la ZONE 3, il n'y a rien à faire
- A la sortie de la boucle, l'entièreté du tableau a été rempli
- Code complet du SP_1



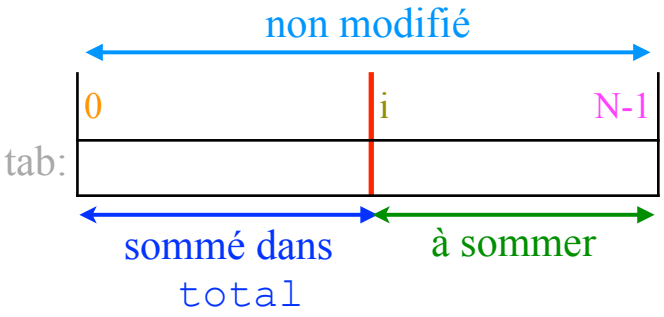

```
unsigned short i;

for(i=0; i<N; i++){
  printf("Donnez la %d eme note: ", i+1);
  scanf("%hu", &tab[i]);
}//fin for - i
```


Exemple (15)

- Étape 3: écriture du code pour le SP_3
- Définition
  - Input
    - ✓ tableau `tab` de `N` notes
      - rempli par le SP_1
  - Output
    - ✓ moyenne des notes dans `moyenne`
  - Caractérisation de l'Input
    - ✓ tableau de notes
      - `tab`
    - ✓ taille du tableau
      - `N`
- Idée de solution
  - calculer la somme des valeurs de `tab`
    - ✓ boucle
    - ✓ Invariant!
  - diviser la somme par `N`

Exemple (16)

- Invariant graphique SP_3



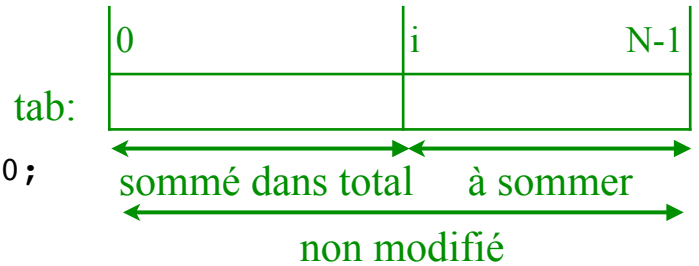
- Fonction de Terminaison
  - N-i

Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Exemple (17)

- Code SP_3

```
float moyenne;
unsigned short total = 0;

for(i=0; i<N; i++)
  total += tab[i];

moyenne = total/N;
printf("Moyenne de la classe: %f\n", moyenne);
```

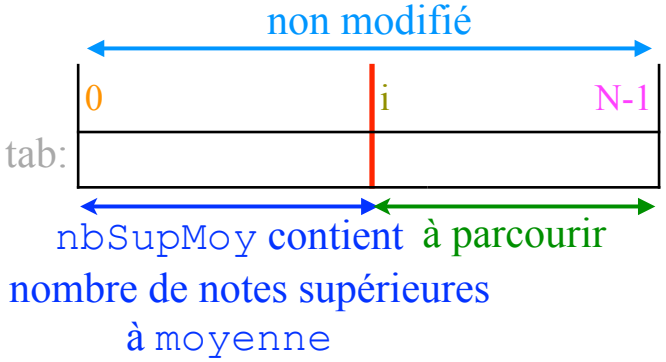


Exemple (18)

- Étape 3: écriture du code pour le SP4
- Définition
  - Input
    - ✓ tableau `tab` de `N` notes
      - rempli par le SP1
    - ✓ `moyenne`, la moyenne des notes
      - fourni par le SP3
  - Output
    - ✓ la quantité de notes supérieures à `moyenne` dans `nbSupMoy`
  - Caractérisation des Inputs
    - ✓ tableau de notes, `tab`
    - ✓ taille du tableau, `N`
    - ✓ la moyenne, `moyenne`
- Idée de solution
  - parcourir `tab` et pour chaque valeur
    - ✓ comparer à `moyenne`
    - ✓ incrémenter un compteur si nécessaire
  - Invariant!

Exemple (19)

- Invariant graphique SP4

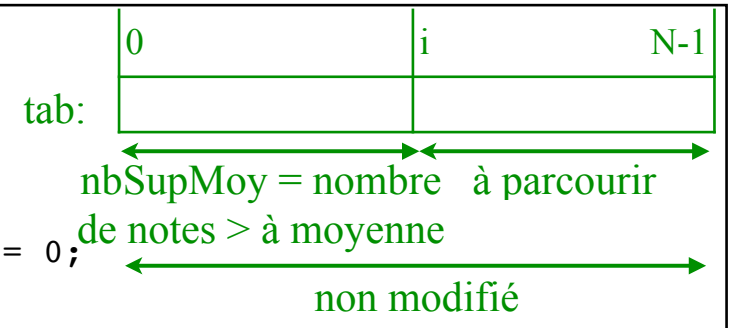


- Fonction de Terminaison
  - N-i

Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Exemple (20)

- Code SP_4 et SP_5




```
unsigned short nbSupMoy = 0;

for(i=0; i<N; i++)
  if(tab[i]>moyenne)
    nbSupMoy++;

printf("%hu eleves supérieurs à la moyenne\n", nbSupMoy);
```


Exemple (21)

- Code complet

```
#include <stdio.h>

int main(){
  //déclaration des variables
  const unsigned short N = 20;
  unsigned short i, total=0, nbSupMoy=0;
  float moyenne;
  unsigned short tab[N];

  //SP1: énumération et encodage
  for(i=0; i<N; i++){
    printf("Donnez la %hu eme note: ", i+1);
    //SP2: lecture au clavier
    scanf("%hu", &tab[i]);
  }//fin for - i

  //à suivre
}//fin programme
```

Exemple (22)

- Code complet (cont.)

```
int main(){
  //cfr. slide précédent
  //SP3: calcul de la moyenne
  for(i=0; i<N; i++)
    total += tab[i];

  moyenne = total/N;
  printf("Moyenne de la classe: %f\n", moyenne);

  //SP4: calcul du nombre d'occurrences
  for(i=0; i<N; i++)
    if(tab[i]>moyenne)
      nbSupMoy++;
  //SP5: affichage
  printf("%hu élèves supérieurs à la moyenne\n", nbSupMoy);
}//fin programme
```


Algorithmique

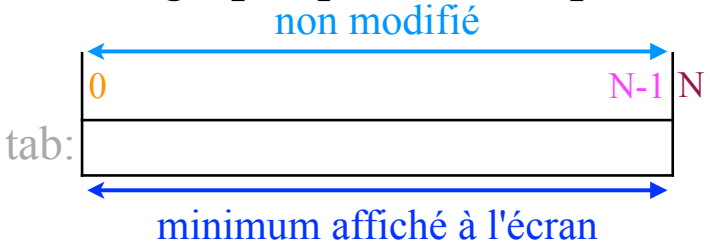
- Problème 1
  - on dispose d'un tableau, `tab`, de `N` entiers précédemment rempli
  - on désire afficher à l'écran la valeur minimum du tableau
- **Problème de recherche du minimum**

Algorithmique (2)

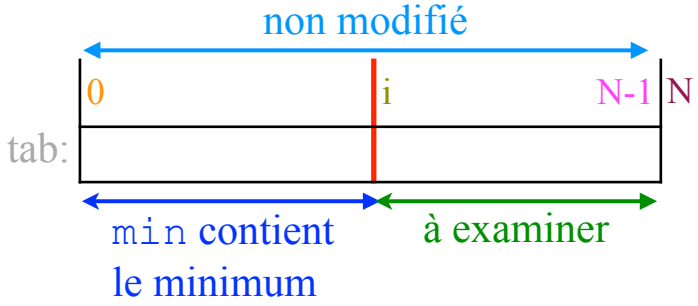
- Définition du problème
  - Input
    - ✓ tableau `tab` à `N` valeurs entières
  - Output
    - ✓ la valeur minimum du tableau est affiché à l'écran
  - Caractérisation des Inputs
    - ✓ `N`, la dimension du tableau
      - `const unsigned int N = ... ;`
    - ✓ `tab`, tableau d'entiers
      - `int tab[N];`
- Analyse du problème
  - ∅
- Idée de solution
  - parcourir le tableau du début à la fin
  - maintenir le minimum "jusque maintenant"
  - afficher à l'écran la valeur du minimum

Algorithmique (3)

- Représentation graphique de l'Output



- Invariant Graphique pour le Problème 1



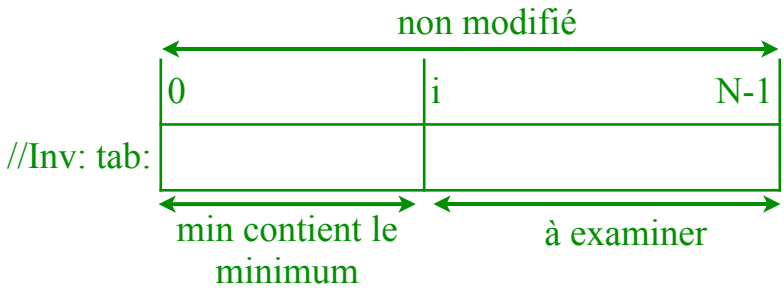
Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

- Fonction de Terminaison
  - N-i

Algorithmique (4)

- Code pour le Problème 1

```
#include <stdio.h>

int main(){
  //déclaration et remplissage
  unsigned int i;
  int min = tab[0];

  for(i=1; i<N; i++){
    if(min > tab[i])
      min = tab[i];
  }//fin for - i

  printf("Le minimum est: %d\n", min);
}//fin programme
```



Algorithmique (5)

- Problème 2
  - on dispose d'un tableau, `tab`, de `N` entiers précédemment rempli
  - le tableau `tab` est trié par ordre croissant
  - on désire déterminer si une valeur `x` donnée par l'utilisateur se trouve dans le tableau
- **<u>Problème de recherche séquentielle dans un tableau trié</u>**

Algorithmique (6)

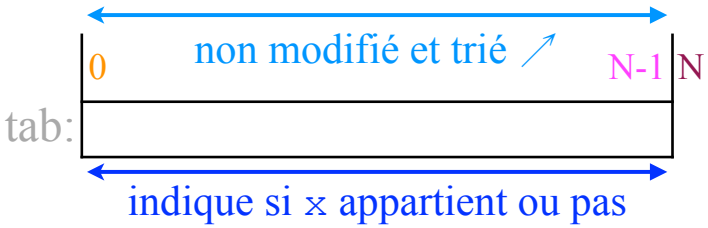
- Définition du problème
  - Input
    - ✓ tableau `tab` trié à `N` valeurs entières
    - ✓ `x`, la valeur à rechercher
  - Output
    - ✓ une indication, à l'écran, indiquant si `x` appartient au tableau
  - Caractérisation des Inputs
    - ✓ `N`, la dimension du tableau
      - `const unsigned int N = ... ;`
    - ✓ `tab`, tableau d'entiers
      - `int tab[N];`
- Analyse du problème
  - SP1: lire au clavier `x`
  - SP2: déterminer l'appartenance de **`x`** dans `tab`
- Enchaînement des SPs
  - SP1 → SP2

Algorithmique (7)

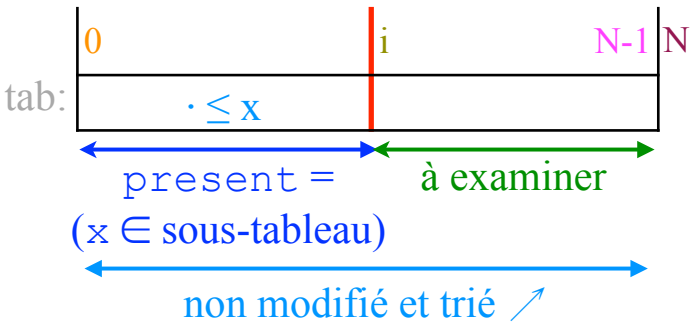
- Idée de solution
  - parcourir le tableau du début à la fin tant que `x` est supérieur ou égal à la valeur courante
  - pour chaque valeur courante
    - ✓ déterminer si elle vaut `x`
    - ✓ si oui, modifier une variable indiquant la présence de `x`
    - ✓ sinon, ne rien faire
  - à la sortie de la boucle, indiquer éventuellement que `x` n'appartient pas au tableau

Algorithmique (8)

- Représentation graphique de l'Output



- Invariant Graphique pour le Problème 2



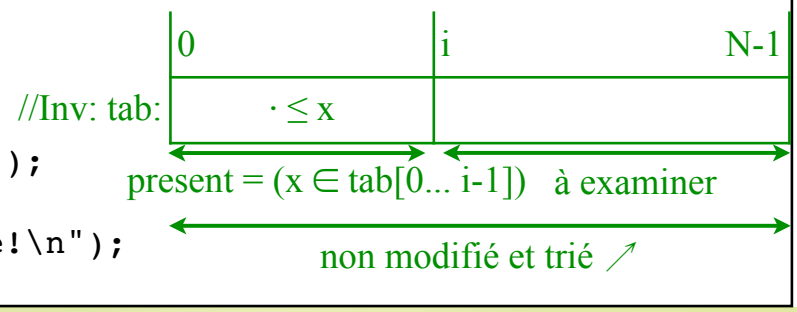
- Fonction de Terminaison
  - N-i

Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Algorithmique (9)

- Code pour le Problème 2

```
#include <stdio.h>

int main(){
  //déclaration et remplissage
  unsigned int i;
  int x, unsigned short present = 0;
  printf("Entrez la valeur de x: ");
  scanf("%d", &x);

  for(i=0; i<N && tab[i]<=x && !present; i++){
    if(tab[i] == x)
      present = 1;
  }//fin for - i
  if(present)
    printf("trouvé!\n");
  else
    printf("pas trouvé!\n");
}//fin programme
```



Algorithmique (10)

- Problème 3
  - on dispose d'un tableau, `tab`, de `N` entiers précédemment rempli
  - le tableau `tab` est trié par ordre croissant
  - on désire déterminer si une valeur `x` donnée par l'utilisateur se trouve dans le tableau
- **Problème de recherche dichotomique**

Algorithmique (11)

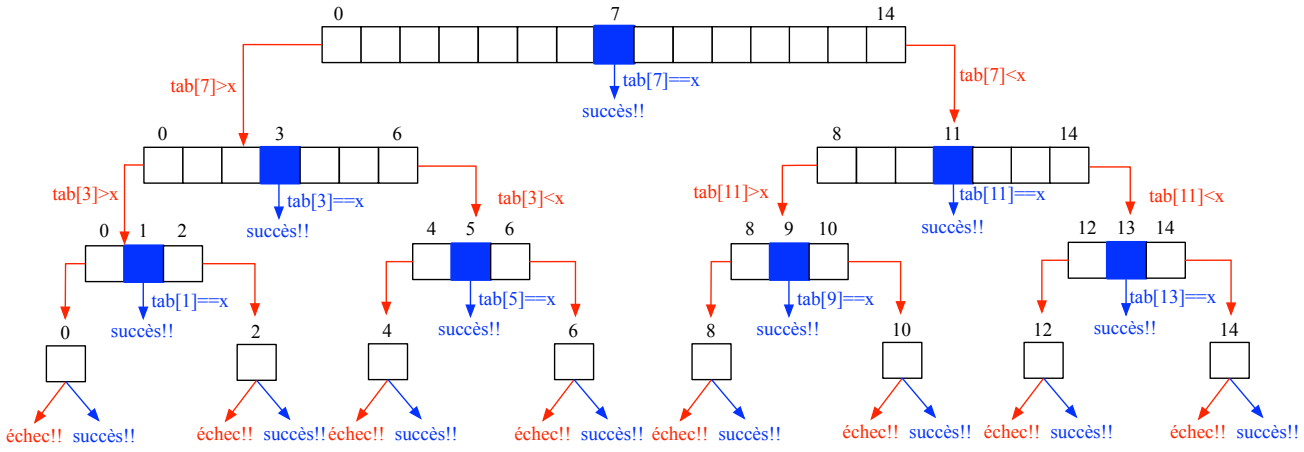
- Définition du problème
  - Input
    - ✓ tableau `tab` trié à `N` valeurs entières
    - ✓ `x`, la valeur à rechercher
  - Output
    - ✓ une indication, à l'écran, indiquant si `x` appartient au tableau
  - Caractérisation des Inputs
    - ✓ `N`, la dimension du tableau
      - `const unsigned int N = ... ;`
    - ✓ `tab`, tableau d'entiers
      - `int tab[N];`
- Analyse du problème
  - SP_1: lire au clavier `x`
  - SP_2: déterminer l'appartenance de **x** dans `tab`
- Enchaînement des SPs
  - $SP_1 \rightarrow SP_2$

Algorithmique (12)

- **recherche dichotomique**
  - on recherche un élément égal à `x` entre 2 indices, `d` et `f`
  - au départ, l'élément peut se trouver n'importe où
    - ✓ `d = 0,`
    - ✓ `f = N-1`
  - réduire l'espace de recherche par 2 (`m = d+f / 2`)
    - ✓ `tab[m] == x`
    - ✓ `tab[m] > x`
    - ✓ `tab[m] < x`

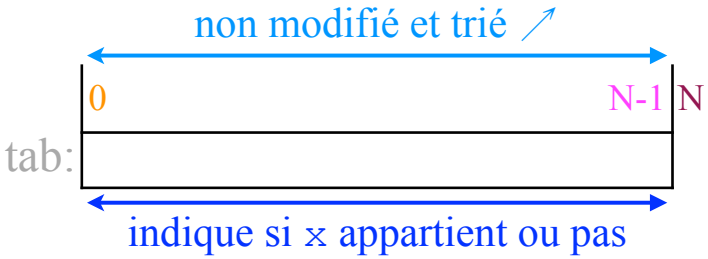
d	m	f
· < tab[m]	??	· > tab[m]

Algorithmique (13)

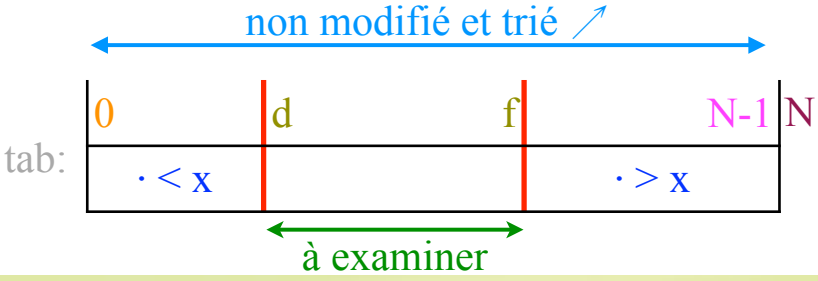


Algorithmique (14)

- Représentation graphique de l'Output



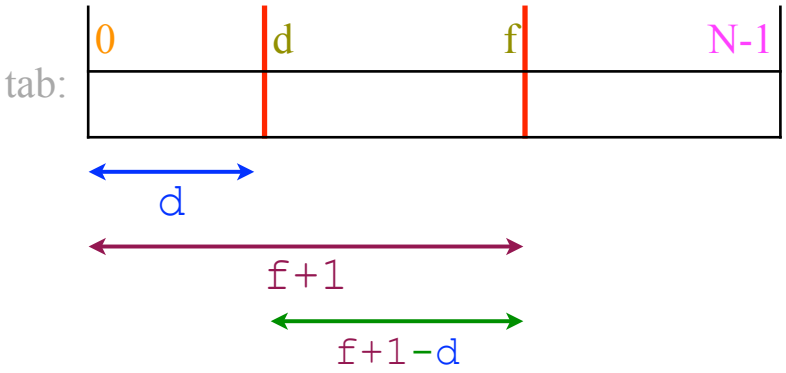
- Invariant Graphique pour le Problème 3



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Algorithmique (15)

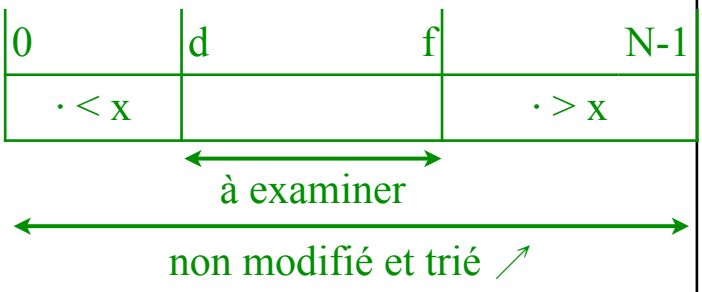
- Fonction de Terminaison



⇒ Fonction de terminaison: f+1-d

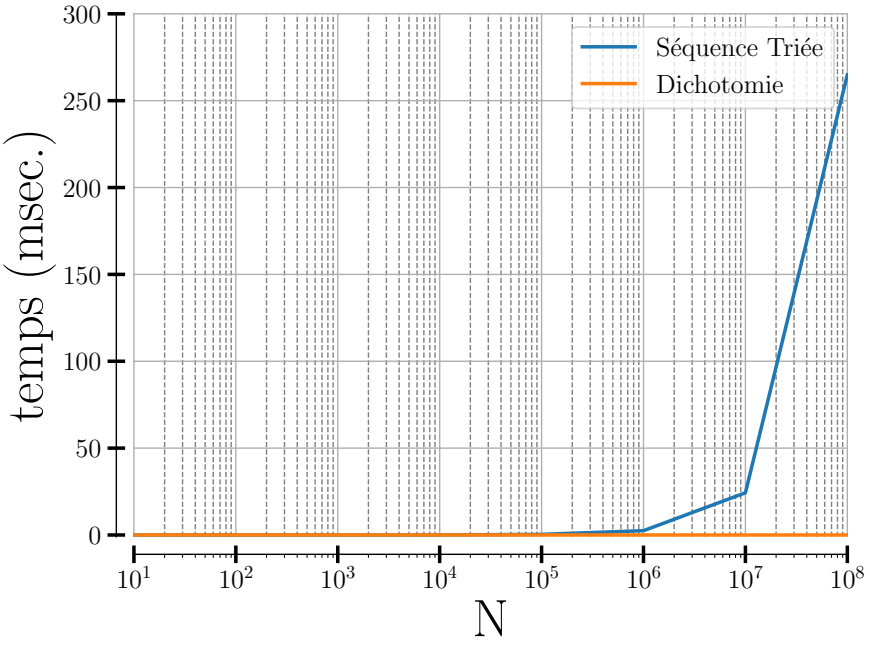
Algorithmique (16)

```
#include <stdio.h>

int main(){
  //déclaration et remplissage
  unsigned int d = 0, f=N-1, m;
  unsigned short present = 0;

  while(d <= f && !present){
    m = (d+f)/2;

    if(x==tab[m])
      present = 1; //Inv: tab:
    else{
      if(x<tab[m])
        f = m - 1;
      else
        d = m + 1;
    }
  }//fin while
}//fin programme
```



Algorithmique (17)

- Comparaison de l'efficacité des algorithmes de recherche



Exercices

- Ecrire un programme qui remplit un tableau de `n` éléments au clavier et affiche, à l'écran, les éléments d'indice impair
- Ecrire un programme qui remplit un tableau de `n` éléments au clavier et affiche, à l'écran, le maximum et minimum du tableau
- Pour chacun des exercices, commencer par proposer un invariant

Agenda

- Chapitre 5: Structures de Données
  - Tableaux Uni-Dimensionnels
  - Tableaux Multi-Dimensionnels
    - ✓ Déclaration
    - ✓ Manipulation
    - ✓ Invariant Graphique
    - ✓ Exemple
  - Chaînes de Caractères
  - Enregistrements
  - Enumérations
  - Fichiers

Déclaration

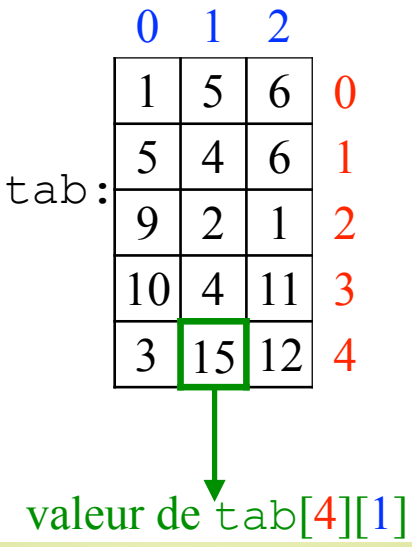
- Les mécanismes liés aux tableaux ne sont pas limités aux vecteurs
- Il y a possibilité de définir (et manipuler) des tableaux multi-dimensionnels

n dimensions & de taille $taille_1 \times ... \times taille_n$

```
[const] type id[taille1][taille2]...[tailleN]
          [= {{...{val1, val2, ...},
              {val1, val2, ...},
              ...
              {val1, val2, ...}...}}];
```

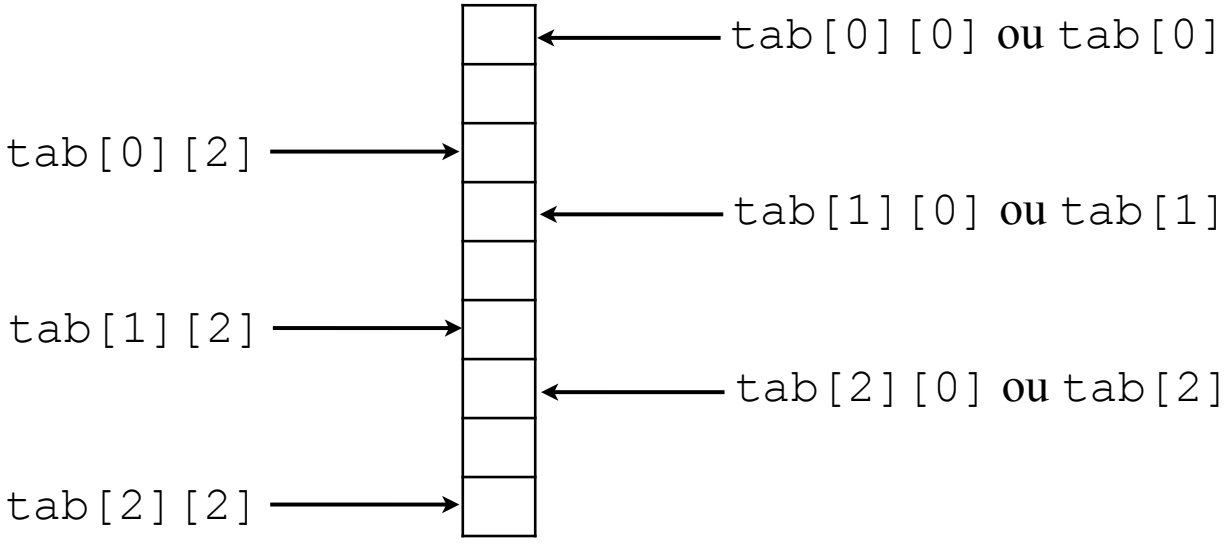

Manipulation

- Manipulation
  - `int tab[5][3];`
    - ✓ tableau de 15 (5 × 3) éléments
  - un élément quelconque de ce tableau se trouve repéré par ses indices
    - ✓ `tab[3][2];`
    - ✓ valeur à gauche



Manipulation (2)

- Comment est représenté en mémoire un tableau à plusieurs dimensions?
  - exemple pour une matrice 3 × 3



Invariant Graphique

- Rappel des règles pour un bon Invariant Graphique
  1. réaliser un dessin pertinent et le nommer
  2. placer sur le dessin les bornes de début et de fin
     - ✓ on peut aussi identifier la taille de la structure
  3. placer une (ou plusieurs) ligne(s) de démarcation qui sépare(nt) ce qui a déjà été calculé dans les itérations précédentes et ce qu'il reste à faire
  4. étiqueter chaque ligne de démarcation avec une variable d'itération
     - ✓ à gauche ou à droite
  5. décrire ce que les itérations précédentes ont déjà calculé en utilisant des variables
     - ✓ ces variables devront se retrouver dans le programme
     - ✓ questions à se poser
       - où est stocké ce résultat?
       - comment peut-on décrire ce résultat (forcément partiel)?
  6. identifier ce qu'il reste à faire dans les itérations suivantes
  7. toutes les structures et variables identifiées sont présentes dans le code.

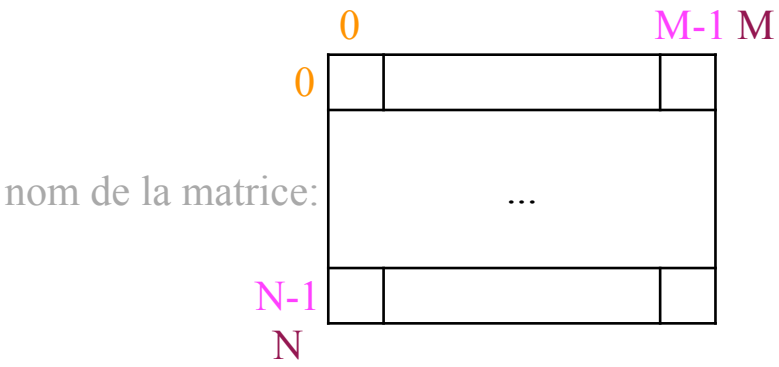
Invariant Graphique (2)

- Rappel du code couleur

Éléments du dessin	Code Couleur	Règle(s) Associée(s)
Nom de la structure		Règle 1
Borne minimale		Règle 2
Borne maximale		
Taille de la structure		
Lignes de démarcation		Règle 3
Étiquette des lignes de démarcation		Règle 4
Ce qui a été réalisée jusqu'à maintenant		Règle 5
zones "à faire"		Règle 6
Propriétés qui sont conservées		Règle 5 + Règle 6

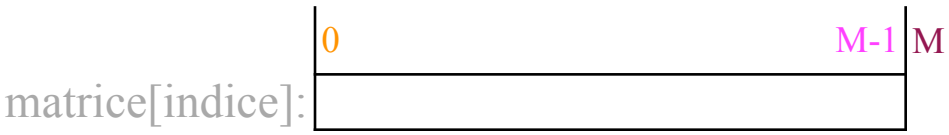
Invariant Graphique (3)

- Plusieurs tableaux
  - permet la représentation d'un tableau à deux dimensions
- Construction
  - chaque case représente une cellule de la matrice
  - se lit du haut vers le bas (lignes), de gauche à droite (colonnes)
    - ✓ l'indice (0,0) se trouve en haut à gauche



Invariant Graphique (4)

- Un seul tableau
  - permet la représentation d'une ligne de la matrice
- Construction
  - même principe qu'un tableau
  - cfr. Slide 19



Exemple

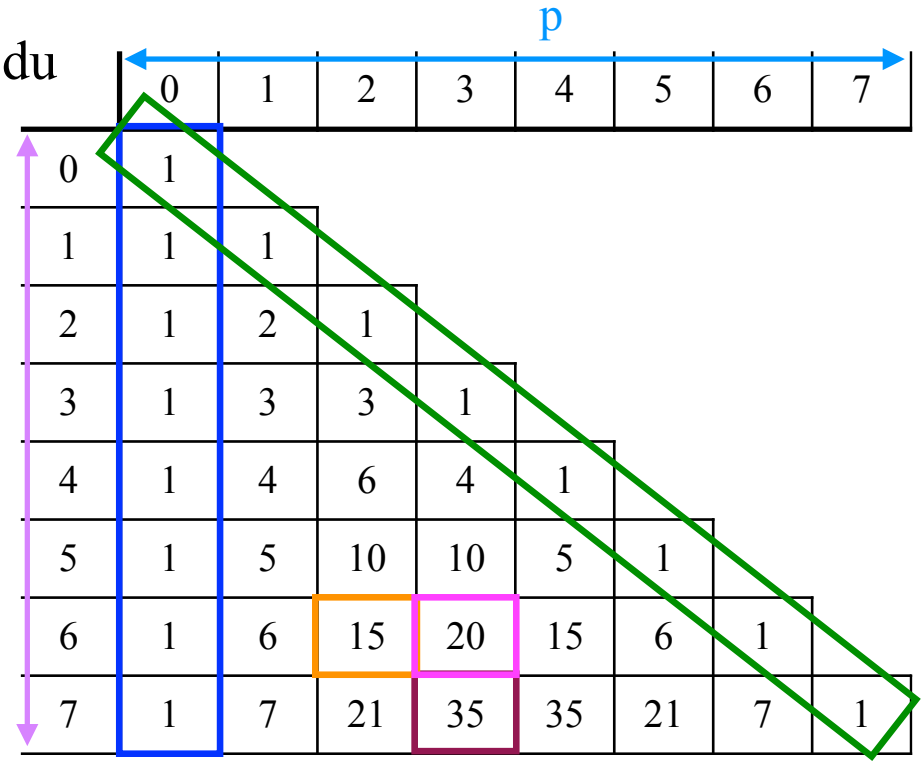
- Exemple: le Triangle de Pascal
  - permet de calculer C^p_n
    - ✓ le nombre de façons de choisir p éléments parmi n éléments distincts
    - ✓ $0 \leq p \leq n$
  - calcul aisé grâce à la formule suivante:
    - ✓ $\forall n \in \mathbb{N}, C_n^0 = C_n^n = 1$
    - ✓ $\forall p, n > 0 \text{ et } p < n, C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$

Exemple (2)

- Construction du triangle

$$C_n^0 = C_n^n = 1$$

$$C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$$



Exemple (3)

- Problème
  - générer les 13 premières valeurs du triangle de Pascal et les stocker dans un tableau
    - ✓ i.e., remplir une matrice triangulaire inférieure avec les valeurs du triangle de Pascal
  - afficher à l'écran le triangle de Pascal généré

Exemple (4)

- Étape 1: définition du problème
  - Input
    - ✓ taille du triangle de Pascal
  - Output
    - ✓ les N premières lignes du triangle de Pascal sont affichées
  - Caractérisation des Inputs
    - ✓ N = 13
      - `const unsigned short N = 13;`
- Représentation graphique

Exemple (5)

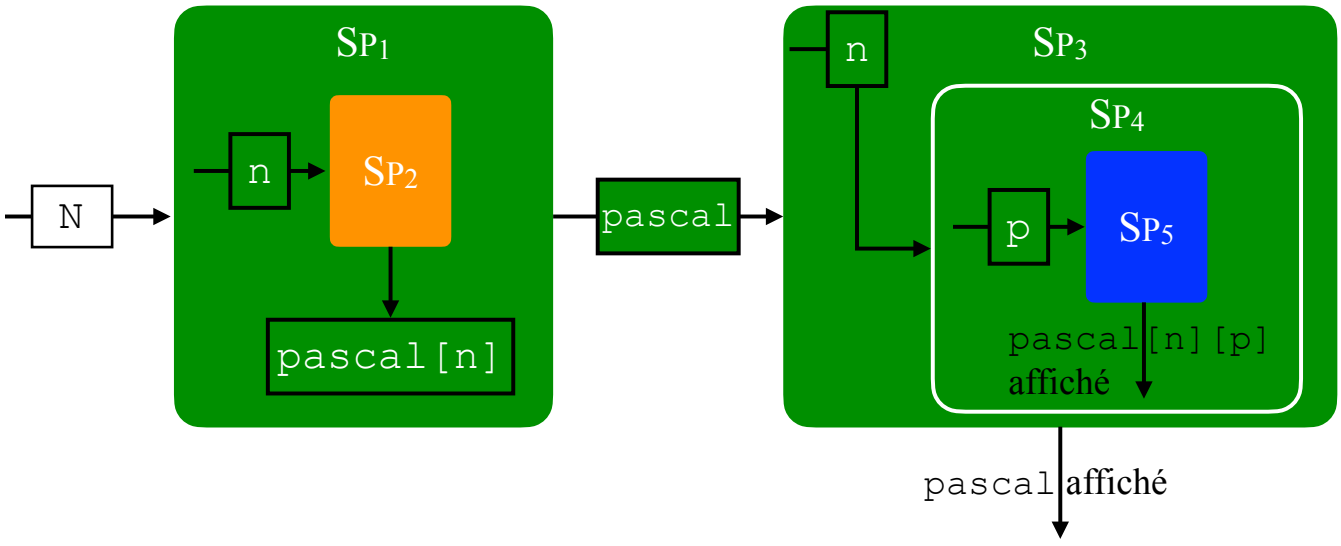
- Étape 2: analyse
- Il y a 2 grandes fonctionnalités
  1. remplir la matrice triangulaire inférieure avec le Triangle de Pascal
  2. afficher la matrice triangulaire inférieure
- On pourrait les considérer comme des SPs
  - structuration linéaire
- Trop simpliste
  - chacun de ces SPs peut se décomposer
    - ✓ cfr. Chapitre 3, Slides 18 & 19

Exemple (6)

- Raffinement des SPs
  - SP_1: énumération et remplissage de la matrice
    - ✓ énumérer les `N` lignes de `pascal` et les remplir
  - SP_2: remplissage d'une ligne
    - ✓ remplir la ligne **`n`** à l'aide de la formule
  - SP_3: énumération et affichage de la matrice
    - ✓ énumérer les `N` lignes de **`pascal`** (rempli) et les afficher
  - SP_4: énumération et affichage d'une ligne
    - ✓ énumérer les colonnes de la ligne **`n`** et les afficher
  - SP_5: affichage d'une cellule
    - ✓ afficher la colonne **`p`** de la ligne **`n`**

Exemple (7)

- Structuration des SPs
  - $(SP_2 \subset SP_1) \rightarrow [\,(SP_5 \subset SP_4) \subset SP_3]$
- Représentation graphique



Exemple (8)

- Étape 3: écriture du Code
- Canevas général du code

```
#include <stdio.h>

int main(){
  const unsigned short N = 13;
  unsigned int pascal[N][N];

  //déclaration de variables supplémentaires

  //résolution des 2 SPs généraux

}//fin programme
```

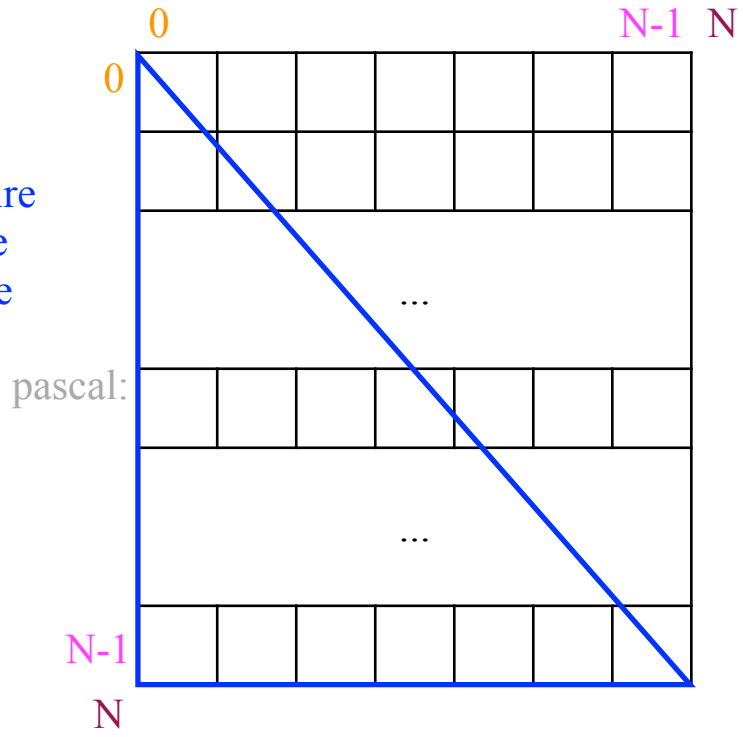

Exemple (9)

- Résolution SP_1 (énumération/remplissage de la matrice)
- Définition
  - Input
    - ✓ `N`, le nombre de lignes de la matrice
  - Output
    - ✓ les lignes de `0` à `N-1` de la matrice `pascal` ont été énumérées et remplies
  - Caractérisation des Inputs
    - ✓ les dimensions de la matrice
      - `N`

Exemple (10)

- Représentation graphique de l'Output du SP_1

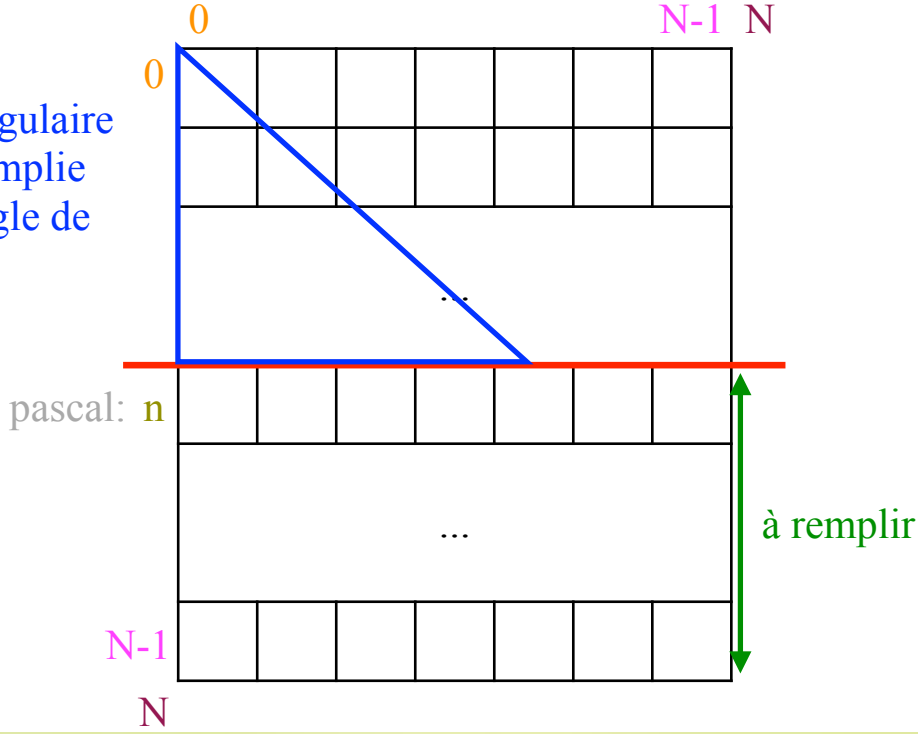
matrice triangulaire inférieure remplie avec le triangle de Pascal



Exemple (11)

- Invariant Graphique SP_1

matrice triangulaire inférieure remplie avec le triangle de Pascal



Légende:
- Règle 1
- Règle 2
- Règle 3
- Règle 4
- Règle 5
- Règle 6

Exemple (12)

- Code SP_1

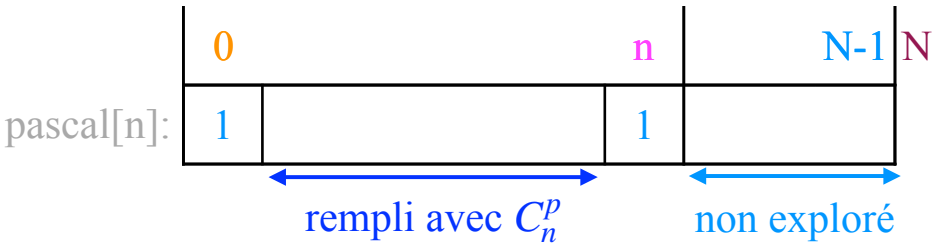
```
unsigned short n;

for(n=0; n<N; n++){
  //appel à SP2
}//fin for - n
```


Exemple (13)

- Résolution SP_2 (remplissage d'une ligne)
- Définition
  - Input
    - ✓ `n`, la ligne courante
      - fourni par le SP_1
  - Output
    - ✓ les colonnes de `0` à `n`, de la ligne `pascal[n]`, ont été remplies
  - Caractérisation des Inputs
    - ✓ la ligne courante, `n`

Exemple (14)

- Représentation graphique de l'Output du SP_2

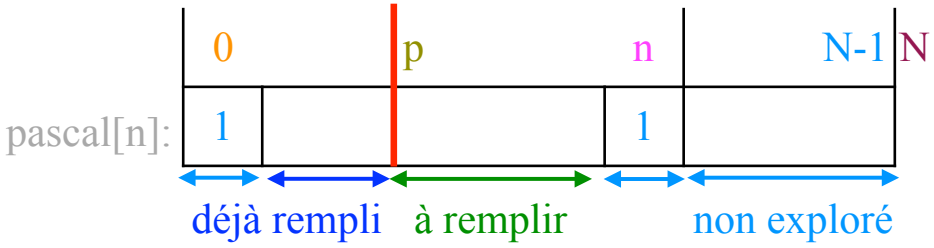


pascal[n][0] = 1 (car $C^0_n = 1$)
pascal[n][n] = 1 (car $C^n_n = 1$)

Exemple (15)

- Invariant Graphique
  - rappel
    - ✓ $C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$
    - ✓ `pascal[n][p] = pascal[n-1][p-1] + pascal[n-1][p]`

Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6



pascal[n][0] = 1 (car $C^0_n = 1$)
pascal[n][n] = 1 (car $C^n_n = 1$)

rempli?
$\forall$ j, 0 < j $\leq$ p-1, pascal[n][j] = C^j_n

Exemple (16)

- Code SP_2

```
unsigned short p;
pascal[n][0] = 1;
pascal[n][n] = 1;

for(p=1; p<n; p++)
  pascal[n][p] = pascal[n-1][p] + pascal[n-1][p-1];
```


Exemple (17)

- Code SP_1 et SP_2

```
unsigned short n, p;

//SP1
for(n=0; n<N; n++){
  //SP2
  pascal[n][0] = 1;
  pascal[n][n] = 1;

  for(p=1; p<n; p++){
    pascal[n][p] = pascal[n-1][p] + pascal[n-1][p-1];
}//fin for - n
```

Exemple (18)

- Exercice
  - faire tout le raisonnement pour les SP_3 à SP_5

Exemple (19)

- Code complet

```
#include <stdio.h>

int main(){
  const unsigned short N = 13;
  unsigned int pascal[N][N];
  unsigned short p, n;

  for(n=0; n<N; n++){
    pascal[n][0] = 1;
    pascal[n][n] = 1;

    for(p=1; p<n; p++)
      pascal[n][p] = pascal[n-1][p] + pascal[n-1][p-1];
  }//fin for - n

  //à suivre
}//fin programme
```

Exemple (20)

- Code complet (cont.)

```
int main(){
  //cfr. slide précédent

  printf("Triangle de Pascal de degré %hu: \n", N);
  for(n=0; n<N; n++){
    printf("N: %2hu", n);

    for(p=0; p<=n; p++)
      printf("%5u", pascal[n][p]);

    printf("\n");
  }//fin for - n
}//fin programme
```


Exercice

- Ecrire un programme qui additionne deux matrices (remplies au clavier) A et B et affiche la matrice résultante C à l'écran.
  - identifier les sous-problèmes
  - proposer les invariants des différents sous-problèmes

Agenda

- Chapitre 5: Structures de Données
  - Tableaux Uni-Dimensionnels
  - Tableaux Multi-Dimensionnels
  - Chaînes de Caractères
    - ✓ Déclaration
    - ✓ Manipulation
  - Enregistrements
  - Enumérations
  - Fichiers

Déclaration

- On est souvent amené à devoir manipuler des données textuelles
  - exemple: chaînes d'ADN
    - ✓ suite de a/t/g/c
- Représentation via des *chaînes de caractères* (ou *string*)
  - en C, ce sont des vecteurs de caractères
    - ✓ `char adn[20];`
- La fin d'une chaîne de caractères est <u>toujours</u> repérée par un *caractère spécial*
  - '\0' ⇒ valeur nulle
  - occupe une position dans le vecteur
    - ✓ `char adn[20];` ⇒ 19 caractères et '\0'

Déclaration (2)

- Une chaîne de caractères peut être initialisée lors de sa déclaration via une forme particulière
  - `char id[] = "texte";`
  - la dimension n'est pas obligatoire
    - ✓ valable uniquement dans ce cas là
- Cette forme ne peut intervenir que lors de la déclaration
  - si par la suite on veut modifier la chaîne de caractères, on doit le faire caractère par caractère

Manipulation

- Une chaîne de caractères se manipule comme un tableau uni-dimensionnel
- On peut donc accéder à chaque case (caractère) à l'aide d'un indice
- Il existe des mécanismes (i.e., formatage) permettant de facilement encoder au clavier et afficher à l'écran une chaîne de caractères

Manipulation (2)

- Comment lire (efficacement) une chaîne de caractères au clavier?
- Soit la déclaration
  - `char str[21];`
    - ✓ 20 caractères utiles + le caractère de terminaison
- Les instructions suivantes
  - `scanf("%20c", str);`
    - ✓ place très exactement 20 caractères dans `str`
  - `scanf(" %20c", str);`
    - ✓ idem supra mais ne considère pas les espaces au début
  - `scanf(" %20[a-zA-Z]", str);`
    - ✓ idem supra mais ne considère que les caractères alphabétiques
  - `scanf("%s", str);`
    - ✓ place une chaîne de max 20 caractères dans `str`
    - ✓ s'arrête au premier espace rencontré

Manipulation (3)

- Comment écrire (efficacement) une chaîne de caractères à l'écran?
- Soit la déclaration
  - `char str[21];`
  - 20 caractères utiles + le caractère de terminaison
  - `str` rempli au préalable
- L'instruction
  - `printf("%s", str);`
  - affiche à l'écran le contenu de `str` jusque '\0'

Exercices

- Déterminer si une chaîne de caractères est un palindrome ou non
- Ecrire un programme qui supprime tous les ‘e’ d’une chaîne de caractères donnée
- Ecrire un programme qui détermine si une chaîne de caractères A est une sous-chaîne d’une chaîne de caractères donnée B

Agenda

- Chapitre 5: Structures de Données
  - Tableaux Uni-Dimensionnels
  - Tableaux Multi-Dimensionnels
  - Chaînes de Caractères
  - Enregistrements
    - ✓ Principe
    - ✓ Déclaration
    - ✓ Type Synonyme
    - ✓ Manipulation
    - ✓ Stockage
    - ✓ Tableaux de Structures
    - ✓ Intérêts
    - ✓ Quelques Erreurs Typiques
  - Enumérations
  - Fichiers

Principe

- Soit le problème suivant
  - l'Apparitorat FSA désire un programme permettant de gérer les étudiants
- Chaque étudiant est représenté par différentes informations
  - nom de famille
    - ✓ `char[50]`
  - matricule ULiège
    - ✓ `unsigned int`
  - année en cours
    - ✓ `unsigned short`
  - moyenne aux examens
    - ✓ `float`
  - nombre de cours suivis
    - ✓ `unsigned short`

Principe (2)

- 1ère tentative de solution
  - un tableau par information à prendre en compte

```
#include <stdio.h>

int main(){
  //déclaration
  const unsigned short NB_ETUDIANTS=250;
  const unsigned short NB_CARACTERES=50;

  char nom[NB_ETUDIANTS][NB_CARACTERES+1];
  unsigned int matricule[NB_ETUDIANTS];
  unsigned short annee[NB_ETUDIANTS];
  float moyenne[NB_ETUDIANTS];
  unsigned short nb_cours[NB_ETUDIANTS];

  //manipulation des variables
}//fin programme
```

Principe (3)

- 1ère tentative de solution
  - un tableau par information à prendre en compte

nom	matricule	annee	moyenne	nb_cours
Martin	20161232	1	12.5	12
Dubois	20153200	2	10.1	13
Legrand	20160023	1	8.7	12
...	...	...	...	...
Dubois	20164560	1	18.3	12
Lefebvre	20161010	1	15.0	12

Principe (4)

- Inconvénient(s) de cette solution?
  - problème de maintien de cohérence (tri, ...)
  - la relation entre Martin, Dubois, Legrand, ... est superficielle
  - impossible de gérer un étudiant particulier sans devoir prendre tout le groupe
- Il faut trouver un autre regroupement

Principe (5)

- Un regroupement naturel met ensemble les données d'un même étudiant

nom	matricule	annee	moyenne	nb_cours
Martin	20161232	1	12.5	12
Dubois	20153200	2	10.1	13
Legrand	20160023	1	8.7	12
...	...	...	...	...
Dubois	20164560	1	18.3	12
Lefebvre	20161010	1	15.0	12

Principe (6)

- Ce qui donne

Martin	20161232	1	12.5	12
Dubois	20153200	2	10.1	13
Legrand	20160023	1	8.7	12

...

Dubois	20164560	1	18.3	12
Lefebvre	20161010	1	15.0	12

Principe (7)

- Ce type de regroupement est possible
  - *enregistrement*
  - *structure*
- **Enregistrement**
  - structure de données hétérogène
  - assemble plusieurs variables, éventuellement de types différents
    - ✓ chaque variable d'un enregistrement est appelé un **champ**
  - sous un nom unique
- Objectif(s)?
  - définir un nouveau type pour des variables
  - être proche de la sémantique de mon problème
    - ✓ modélisation du problème
  - cohérence des données

Déclaration

- Comment déclarer un enregistrement?

identifiant de l'enregistrement

mot-clé

```
struct id_structure{
  type1 id_champ1;
  [type2 id_champ2;
  ...
  typen id_champn;]
};
```

le(s) champ(s) et son/leur type(s)

obligatoire

- Où déclarer un enregistrement?
  - entre les dérives de compilation et le `main()`

Déclaration (2)

- Exemple

```
#include <stdio.h>

struct etudiant{
  char nom[51];
  unsigned int matricule;
  unsigned short annee;
  float moyenne;
  unsigned short nb_cours;
};

int main(){
  //le programme
}//fin programme
```

nom	matricule	annee	moyenne	nb_cours
Martin	20161232	1	12.5	12

Déclaration (3)

- Un enregistrement peut être composé de champ(s) de type(s) plus complexe(s) que `int`, `float`, ou encore `double`
  - tableau
  - enregistrement
  - pointeur
- Exemple
  - gestion d'une équipe de foot
    - ✓ matricule
    - ✓ nom
    - ✓ nombre de matchs joués
    - ✓ nombre de points
    - ✓ nombre de matchs gagnés/perdus/nuls

Déclaration (4)

- Exemple (suite)

```
#include <stdio.h>

struct equipe{
  char nom[51];
  unsigned int matricule;
  unsigned short nb_match;
  unsigned short nb_gagne;
  unsigned short nb_perdu;
  unsigned short nb_nul;
  unsigned short points;
};

int main(){
  //le programme
}//fin programme
```


Déclaration (5)

- Exemple (suite)
  - gestion des personnes
    - ✓ nom
    - ✓ prénom
    - ✓ nombre d'heures effectuées durant le mois

```
#include <stdio.h>

struct personne{
  char nom[31];
  char prenom[31];
  float heures[31];
};

int main(){
  //le programme
}//fin programme
```

Déclaration (6)

- Exemple (suite)
  - gestion d'un étudiant
    - ✓ date de naissance

```
#include <stdio.h>

struct date{
  unsigned short jour;
  unsigned short mois;
  unsigned short annee;
};
struct etudiant{
  unsigned int matricule;
  unsigned short annee;
  float moyenne;
  unsigned short nb_cours;
  struct date date_naissance;
};
```


Déclaration (7)

- Comment déclarer une variable de type enregistrement?

```
#include <stdio.h>

struct etudiant{
  char nom[51];
  unsigned int matricule;
  unsigned short annee;
  float moyenne;
  unsigned short nb_cours;
};

int main(){
  struct etudiant e;
}//fin programme
```

Déclaration (8)

- Il est possible d'initialiser une variable de type enregistrement dès sa déclaration

```
struct equipe{
  char nom[51];
  unsigned int matricule;
  unsigned short nb_match;
  unsigned short nb_gagne;
  unsigned short nb_perdu;
  unsigned short nb_nul;
  unsigned short points;
};

int main(){
  struct equipe standard ={"Standard", 16, 15, 15, 0, 0,
  45};
}//fin programme
```


Type Synonyme

- Il peut devenir fastidieux de devoir répéter le mot-clé **`struct`** lors de chaque définition de variable
  - Peut-on envisager un raccourci?
- Il est possible, en C, de définir des **types synonymes**
  - s'applique à tous les types, pas seulement les structures

mot-clé

```
typedef type id;   le synonyme
```

type pour lequel on veut créer un synonyme

Type Synonyme (2)

- Exemples de définition de synonymes

```
typedef int Entier;

typedef int Vecteur[3];

typedef struct{
  char nom[50];
  unsigned int matricule;
  unsigned short nb_match;
  unsigned short nb_gagne;
  unsigned short nb_perdu;
  unsigned short nb_nul;
  unsigned short points;
}Equipe;
```


Type Synonyme (3)

- Exemple (suite)

```
#include <stdio.h>

typedef struct{
  char nom[51];
  unsigned int matricule;
  unsigned short nb_match;
  unsigned short nb_gagne;
  unsigned short nb_perdu;
  unsigned short nb_nul;
  unsigned short points;
} Equipe;

int main(){
  Equipe standard, anderlecht;
}//fin programme
```

Manipulation

- Comment accéder à un champ?
- Une fois la variable déclarée, on accède à un champ à l'aide de l'opérateur . ("point")
- Format

```
id_structure.id_champ
```

- un champ est traité comme n'importe quelle autre variable du même type
  - peut se trouver à gauche (ou à droite) d'une affectation
  - peut faire l'objet d'une évaluation booléenne

Manipulation (2)

- Exemple

```
#include <stdio.h>

typedef struct{
  char nom[51];
  unsigned int matricule;
  unsigned short nb_match;
  unsigned short nb_gagne;
  unsigned short nb_perdu;
  unsigned short nb_nul;
  unsigned short points;
}Equipe;

int main(){
  Equipe standard;
  standard.matricule = 16;
  standard.nb_match++;
  printf("%u\n", standard.nb_match);
}//fin programme
```

Manipulation (3)

- Exemple (suite)

```
#include <stdio.h>

typedef struct{
  char nom[31];
  char prenom[31];
  float heures[31];
}Personne;

int main(){
  Personne emp = {"Dupont", "Jules", {8,7,8,6,8,0,0,8,..., 8}};

  printf("%f\n", emp.heures[4]);
}//fin programme
```


Manipulation (4)

- Attention, il n'est pas permis de comparer la valeur de 2 structures à l'aide de l'opérateur ==
  - C ne fournit pas d'opérateur permettant de tester l'égalité de 2 structures
- Comment faire?

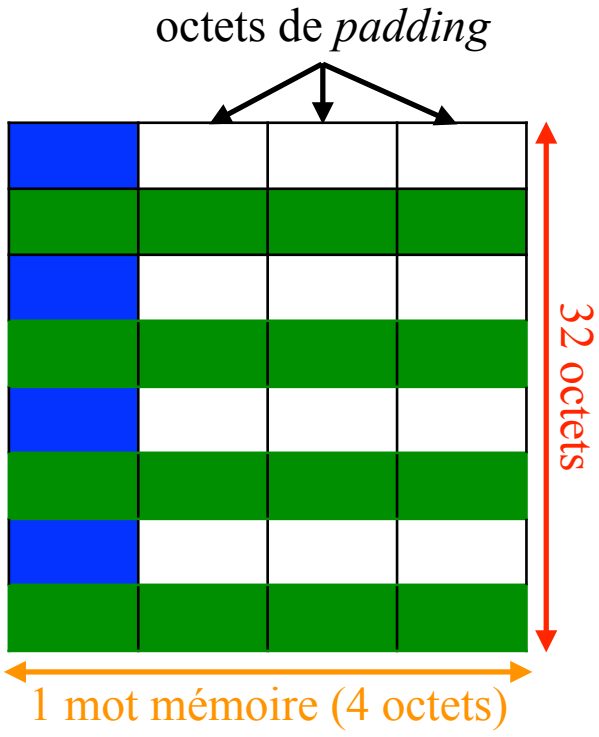
Stockage

- Comment est géré en mémoire un enregistrement?
  - alignement des champs sur un mot mémoire
- L'ordre des champs a de l'importance
  - occupation mémoire plus importante que la somme de chacun des champs si pas d'alignement sur un mot mémoire
  - en général, le compilateur se charge d'optimiser votre code pour l'alignement des champs

Stockage (2)

- Exemple 1
  - mauvais alignement

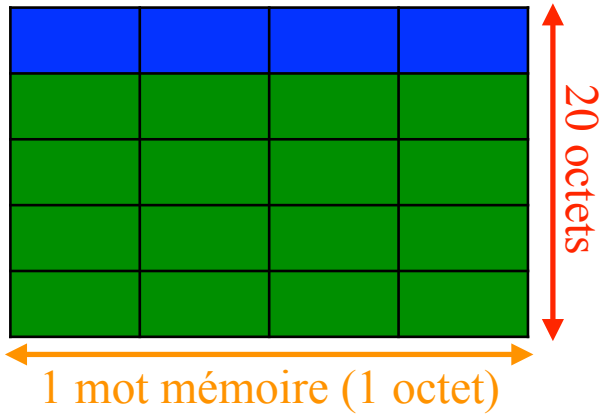
```
struct Fiche1{
  char sexe;
  unsigned int age;
  char permis;
  int assurance;
  char option;
  float salaire;
  char categorie;
  int nb_heures;
};
```



Stockage (3)

- Exemple 2
  - bon alignement

```
struct Fiche2{
  char sexe;
  char permis;
  char option;
  char categorie;
  unsigned int age;
  int assurance;
  float salaire;
  int nb_heures;
};
```



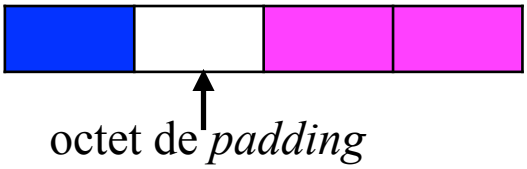
Stockage (4)

- Règles d'alignement en mémoire (sur machine standard 32 bits)
  - `char`
    - ✓ peut occuper n'importe quel octet
  - `short`
    - ✓ sur les demi-mots
    - ✓ octets d'adresse paire
  - `int`, `float`, pointeurs
    - ✓ sur les mots
- Les octets de padding sont ajoutés en fin d'enregistrement pour
  - s'aligner sur le champ de plus grande taille
  - faciliter la manipulation des tableaux

Stockage (5)

- Exemple 1

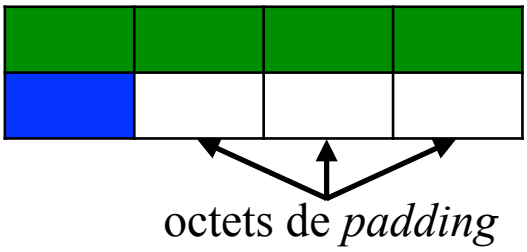
```
struct Exemple1{
  char a;
  short b;
};
```



Stockage (6)

- Exemple 2

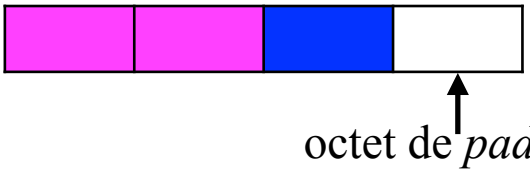
```
struct Exemple2{
  int n;
  char c;
};
```



Stockage (7)

- Exemple 3

```
struct Exemple1{
  short a;
  char b;
};
```



Tableaux de Struct.

- Les enregistrements prennent tout leur sens quand on les utilise avec un tableau
  - **tableau d'enregistrements** (ou tableau de structures)
    - ✓ chaque indice du tableau est un élément de type enregistrement
- Un tableau d'enregistrements se manipule comme n'importe quel tableau

Tableaux de Struct. (2)

- Retour sur le problème de modélisation des étudiants

```
#include <stdio.h>

typedef struct{
  char nom[51];
  unsigned int matricule;
  unsigned short annee;
  float moyenne;
  unsigned short nb_cours;
}Etudiant;

int main(){
   const unsigned short NB_ETUDIANTS=200;
   Etudiant tab_etudiants[NB_ETUDIANTS];
}//fin programme
```


Tableaux de Struct. (3)

indices contenu de `tab_etudiants[0].matricule`

	nom	matricule	annee	moyenne	nb_cours
0	Martin	20161232	1	12.5	12
1	Dubois	20153200	2	10.1	13
2	Legrand	20160023	1	8.7	12
...	...	...	...	...	...
198	Dubois	20164560	1	18.3	12
199	Lefevebre	20161010	1	15.0	12

contenu de `tab_etudiants[2]` tableau `tab_etudiants`

Tableaux de Struct. (4)

```c
#include <stdio.h>

typedef struct{
  char nom[51];
  unsigned int matricule;
  unsigned short nb_match;
  unsigned short nb_gagne;
  unsigned short nb_perdu;
  unsigned short nb_nul;
  unsigned short points;
}Equipe;

int main(){
  Equipe championnat[16];
  championnat[0].matricule=3;
  championnat[1].matricule=7;
  championnat[3].matricule=16;
  Equipe standard = championnat[3];
}//fin programme
```


Intérêts

- Intérêts d'un enregistrement?
  - *lisibilité*
    - ✓ regrouper un ensemble de données sous un même chapeau, nommé de façon explicite, facilite la relecture du code
  - *sémantique*
    - ✓ construction de structure de données proches du problème posé
  - *modularité*
    - ✓ rajout de champs très facile, avec peu de changement
  - *incontournable*
    - ✓ les langages OO généralisent la notion d'enregistrement

Erreurs Typiques

- Message d'erreur étrange à la compilation

```
struct etudiant{
  char nom[51];
  unsigned int matricule;
  unsigned short annee;
  float moyenne;
  unsigned short nb_cours;
}

int main(){
  struct etudiant e;
}//fin programme
```

```
error: two or more data types in declaration specifiers
```

- Oubli du point-virgule!

Erreurs Typiques (2)

- Tentative d'accès à un champ inexistant

```
e.toto = 3;
```

```
prog.c: in function 'main':
prog.c:22: error struct 'etudiant' has no member named
'toto'
```

Exercice

- Une menuiserie gère un stock de panneaux de bois. Chaque panneau possède une largeur, une longueur et une épaisseur en mm, ainsi que le type de bois qui peut être du pin (code ‘p’), le chêne (code ‘c’) ou le hêtre (code ‘h’).
  - Définir la structure panneau permettant d’encoder ces informations
  - Encoder, au clavier, un stock de panneau
  - Indiquer à l’écran la quantité de panneaux de chaque type de bois

Agenda

- Chapitre 5: Structures de Données
  - Tableaux Uni-Dimensionnels
  - Tableaux Multi-Dimensionnels
  - Chaînes de Caractères
  - Enregistrements
  - Enumérations
    - ✓ Principe
    - ✓ Déclaration
    - ✓ Manipulation
  - Fichiers

Principe

- Il peut être parfois utile de pouvoir définir des noms symboliques correspondant à des valeurs entières
- Exemple

```
const unsigned int BLEU = 0;
const unsigned int VERT = 1;
const unsigned int ROUGE = 2;
const unsigned int JAUNE = 3;
const unsigned int NOIR = 4;
```

- Une telle définition n'est pas pratique
- Solution
  - **énumération**

Déclaration

- Format

```
enum nom {id0, id1, id2, ..., idn-1};
```

mot clé

identifiant de l'énumération

liste des noms symboliques

- Exemple

```
enum couleurs {bleu, vert, rouge, jaune, noir};
```

Déclaration (2)

- Les éléments composant une énumération sont de type `int`
- Par défaut, la valeur associée commence à 0 et est incrémentée de 1 à chaque nom symbolique
- Exemple

```
enum couleurs{
  bleu,  //vaut 0
  vert,  //vaut 1
  rouge, //vaut 2
  jaune, //vaut 3
  noir   //vaut 4
};
```


Déclaration (3)

- On peut modifier les valeurs associées aux noms symboliques
  - on peut trouver plusieurs fois la même valeur
- Exemple

```
enum couleurs{bleu=45, vert=45, rouge=45, jaune, noir};
```

Déclaration (4)

- A l'instar des enregistrements, on peut définir un type synonyme
- Exemple

```
typedef enum{
  bleu, vert, rouge, jaune, noir
}couleur;
```


Manipulation

- Généralement, une énumération se manipule en faisant référence à un nom symbolique qui la compose
- Exemple

```
#include <stdio.h>

enum couleurs{bleu, vert, rouge, jaune, noir};

int main(){
  printf("%d %d %d %d %d\n", bleu, vert, rouge, jaune, noir);
}//fin programme
```

Manipulation (2)

- On peut déclarer une variable de type énumération
- Exemple

```
#include <stdio.h>

enum jour_semaine{lundi, mardi, mercredi, jeudi, vendredi};

int main(){
  enum jour_semaine j;
}//fin programme
```


Manipulation (3)

- Si on déclare un type synonyme

```
typedef enum{
  bleu, vert, rouge, jaune, noir
}couleur;

int main(){
  couleur coul;
}//fin programme
```

Manipulation (4)

- Il est possible d'affecter à une variable de type énuméré n'importe quelle valeur entière
  - pour autant qu'elle soit représentable dans un `int`
- Exemple

```
typedef enum{
  jaune, rouge, bleu, vert
}couleur;

int main(){
  couleur c1, c2;
  c1 = 2;  //équivalent à c1 = bleu;
  c2 = 25; //accepté bien que 25 n'appartienne pas au type
           //enum couleur
}//fin programme
```


Manipulation (5)

- Le nom symbolique se manipule comme un entier

```
#include <stdio.h>
typedef enum{
  Mercure, Venus, Terre, Mars, Jupiter, Saturne, Uranus,
  Neptune, Pluton
}Planetes;

int main(){
  Planetes p1, p2;

  p1 = Mars;
  p2 = Terre;

  if(p1>p2)
    printf("Mars est plus éloigné du Soleil que la Terre");
  else
    printf("La Terre est plus éloignée du Soleil que Mars");
}//fin programme
```

Manipulation (6)

```
int main(){
    int provenance;
    printf("De quelle planète venez-vous?");
    scanf("%d", &provenance);
    switch(provenance){
      case Mercure:
        printf("Vous venez de Mercure\n");
        break;
      case Venus:
        printf("Vous venez de Venus\n");
        break;
      ...
      case Pluton:
        printf("Venez de Pluton\n");
        break;
      default:
        printf("Vous ne venez pas de ce système solaire!\n");
    }//fin switch
}//fin programme
```


Agenda

- Chapitre 5: Structures de Données
  - Tableaux Uni-Dimensionnels
  - Tableaux Multi-Dimensionnels
  - Chaînes de Caractères
  - Enregistrements
  - Enumérations
  - Fichiers
    - ✓ Principe
    - ✓ Ouverture
    - ✓ Fermeture
    - ✓ Lecture
    - ✓ Ecriture

Principe

- Un ordinateur est composé, principalement, de deux types de mémoire
  - mémoire centrale
    - volatile
  - mémoire disque
    - permanente
- Un **fichier** est une série de données stockées sur un disque (ou tout autre périphérique de stockage permanent)
  - *fichier texte*: contient du texte ASCII
  - *fichier binaire*: contient des suites de 0 et 1

Principe (2)

- Trois opérations sont possibles sur les fichiers
  - *lecture*
    - transfert des données depuis le fichier vers la mémoire centrale
  - *écriture*
    - transfert des données depuis la mémoire centrale vers le fichier
  - *exécution*
    - le fichier (binaire) est chargé en mémoire centrale et exécuté comme une application
- Dans le cadre du cours, on se limite
  - aux fichiers texte
  - à accès séquentiel

Principe (3)

- La manipulation d'un fichier se fait en 5 étapes
  1. inclure `stdio.h`
  2. déclarer un descripteur de fichier
     - ✓ variable de type `FILE *`
  3. ouvrir le fichier
     - ✓ utilisation de `fopen()`
  4. écrire/lire dans le fichier
     - ✓ utilisation de `fprintf()` pour l'écriture
     - ✓ utilisation de `fscanf()` pour la lecture
  5. fermer le fichier
     - ✓ utilisation de `fclose()`

Ouverture

- Pour lire/écrire dans un fichier, il est nécessaire de l'ouvrir au préalable
- Utilisation de la fonction `fopen()`
  - définie dans `stdio.h`
- Format

```
FILE *fopen(char *, char *);
```

l'ouverture retourne un pointeur sur `FILE`

chemin vers le fichier à ouvrir

mode d'ouverture

Ouverture (2)

- Le chemin d'accès au fichier peut être
  - le nom de fichier seul
    - ✓ le fichier doit, alors, être dans le même répertoire que le programme
    - ✓ "toto.txt"
  - l'explicitation complète du répertoire et du nom de fichier
    - ✓ le fichier peut se trouver n'importe où
    - ✓ “Users/benoit/toto.txt”
- Il y a deux modes d'ouverture du fichier

Mode	Signification
`r`	ouverture en lecture
`w`	ouverture en écriture

Ouverture (3)

- En cas de problème(s) à l'ouverture, `fopen()` retourne `NULL`
- Programmation défensive
  - on vérifie toujours le résultat de `fopen()`

Ouverture (4)

- Exemple 1

```
#include <stdio.h>

int main(){
  FILE *fp;
  fp = fopen("toto.txt", "r");

  if(fp==NULL){
    printf("impossible d'ouvrir le fichier\n");
  }else{
    //manipulation du fichier
  }
}//fin programme
```

1. inclure stdio.h
2. descripteur de fichier
3. ouverture en mode "lecture" du fichier toto.txt

toujours vérifier l'ouverture

Ouverture (5)

- Exemple 2
  - nom du fichier lu au clavier

```
#include <stdio.h>

int main(){
  FILE *fp;
  char nom_fichier[21];

  printf("Entrez le nom du fichier (max. 20 caractères):");
  scanf("%s", nom_fichier);

  fp = fopen(nom_fichier, "r");

  if(fp==NULL){
    printf("impossible d'ouvrir le fichier\n");
  }else{
    //manipulation du fichier
  }
}//fin programme
```

Fermeture

- Après lecture/écriture, un fichier doit toujours être fermé
  - `fclose(FILE*)`

```
#include <stdio.h>

int main(){
  FILE *fp;
  fp = fopen("toto.txt", "r");

  if(fp==NULL){
    printf("impossible d'ouvrir le fichier\n");
  }else{
    //manipulation du fichier

    fclose(fp);
  }
}//fin programme
```


Lecture

- La lecture du contenu d'un fichier peut se faire au moyen de la fonction `fscanf()`
  - lit caractère par caractère dans un fichier
  - fonctionne comme `scanf()` mais pour les fichiers

```
int fscanf(FILE*, const char*, ...);
```

Pointeur vers le fichier ouvert en lecture
format des caractères à lire (%d, %s, %f, ...)
Liste des variables pour stocker les caractères
Nombre de caractères lus

Lecture (2)

- Exemple
  - soit le fichier `toto.txt` formaté de la façon suivante:
    - ✓ chaque ligne dispose de 2 colonnes
      - espace séparant chaque colonne
      - chaque colonne contient une valeur réelle
    - ✓ le fichier comporte un nombre indéfini de lignes
  - problème
    - ✓ afficher à l'écran ligne par ligne le contenu du fichier

`toto.txt`

0.6434	5.6
3.1415	-4.3
2.7182	-6.23
0.6627	45.7
...	
0.6617	32.0

Lecture (3)

```
#include <stdio.h>
```
1. inclure stdio.h

```
int main(){
  FILE* fp = fopen("toto.txt", "r");
  float a, b;
```
2-3. descripteur de fichier + ouverture

vérification de l'ouverture

```
  if(fp==NULL)
    printf("impossible d'ouvrir le fichier\n");
  else{
    while(fscanf(fp, "%f %f", &a, &b)==2)
       printf("%f %f\n", a, b);

    fclose(fp);
  }
}//fin main()
```
4. manipulation en lecture

5. fermeture

Lecture (4)

- Que se passe-t-il avec le `fscanf()` ?

```
FILE* fp = fopen("toto.txt", "r");
//code ...
while(fscanf(fp, "%f %f", &a, &b)==2)
  printf("%f %f\n", a, b);
fclose(fp);
```

```
$>./programme
0.6434 5.6
3.1415 -4.3
2.7182 -6.23
0.6627 45.7
...
0.6617 32.0
$>
```

toto.txt

0.6434	5.6
3.1415	-4.3
2.7182	-6.23
0.6627	45.7
...	
0.6617	32.0

Ecriture

- L'écriture dans un fichier peut se faire au moyen de la fonction `fprintf()`
  - écrit caractère par caractère dans un fichier
  - fonctionne comme `printf()` mais pour les fichiers

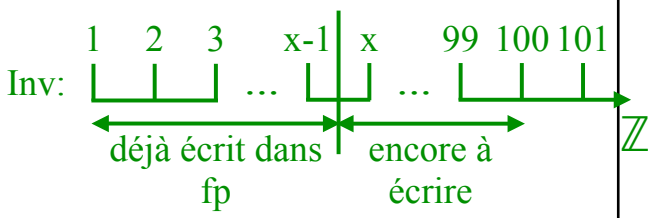
```
int fprintf(FILE*, const char*, ...);
```

même format que fscanf

Ecriture (2)

- Exemple

```
#include <stdio.h>

int main(){
  int x;
  FILE* fp = fopen("lulu.txt", "w");

  if(fp==NULL)
    printf("Impossible d'ouvrir le fichier!\n");
  else{
    for(x=1; x<=100; x++)
      fprintf(fp, "%d\n", x);

    fclose(fp);
  }
}//fin programme
```



Exercices

- Un fichier contient les descriptions d'article en vente. Chaque description se compose du code du produit (compris entre 0 et 99), du prix unitaire du produit (en €) et du nombre d'articles de ce produit en stock (≥ 0).
- Chaque ligne du fichier contient le code suivi du prix et du stock (un espace entre chaque). Le fichier contient, au maximum, 100 descriptions. Il est demandé:
  - de proposer une structure de données pour conserver en mémoire la description d'un produit
  - d'écrire un programme
    - ✓ qui charge le fichier des descriptions en mémoire via un tableau de structure
    - ✓ qui donne le prix d'un article en fonction de son code
    - ✓ qui détermine si un produit est toujours en stock, en fonction de son code

Exercices (2)

- qui diminue le stock d'un produit donné, en fonction de son code, d'un certain nombre d'unités
  - attention, cela doit se faire en fonction du stock disponible
- qui calcule la valeur du stock
  - somme du prix*quantité pour chaque produit
- qui sauve les produits dans un fichier (même format que le fichier de lecture)

- Le programme doit proposer un menu permettant de
  - consulter le prix d'un produit
  - consulter l'état du stock d'un produit
  - acheter un produit
  - connaître la valeur du stock
  - de quitter