# Cours d'algorithmique

IUT Informatique première année
Olivier ROUSSEL
*olivier.roussel@iut-lens.univ-artois.fr*



## Une brève bibliographie


- Initiation à l'algorithmique et aux structures de données, vol. 1 et 2, *J. Courtin et I. Kowarski*, éd. Dunod.
- Algorithmique pour les BTS et IUT, vol 1 : les bases de la programmation, *A. Maunoury et K. Ben Sassi*, éd. Masson.
- C++, *G. Willms*, éd. Micro-application, collection PC-Poche.
- La programmation en Pascal, *Peter Grogno*, éd. Intereditions.
- Initiation à la programmation, *C. Delannoy*, éd. Eyrolles




## Le menu

- Notions de syntaxe
- Types de base
- Constantes
- Variables
- Entrées/Sorties
- Opérateurs
- Expressions
- Logique
- …



## Le menu (suite)

- …
- Les structures de contrôle
  - Séquence
  - Structures conditionnelles
  - Structures itératives (boucles)
- Procédures/Fonctions
- Définition de nouveaux types
  - Structures
  - Tableaux
- Pointeurs et allocation dynamique



## Notions sur les langages



## Les règles à respecter

- Dans toute langue (Français, Anglais,…), il y a des règles à respecter
  - L'orthographe des mots (**règles lexicales**)
    ensemble des règles qui permettent de savoir si un mot est correctement écrit ou pas
  - La grammaire (**règles syntaxiques**)
    ensemble des règles qui permettent de juxtaposer des mots de la langue pour former une phrase correcte
- Ces règles permettent de construire des phrases qui seront reconnaissables comme étant du français, de l'anglais ou une autre langue. Elle n'indiquent pas quel sens a la phrase. Les **règles sémantiques** permettent de définir le sens qu'aura une phrase.



## Pour se faire comprendre…

- Pour pouvoir être compris par un interlocuteur, il faut respecter toutes ces règles, quel que soit le langage ou la langue que l'on utilise (français, anglais, langage informatique…).



## Définition d'un langage

- Un langage informatique se définit par
  - Des règles lexicales :
    elles indiquent comment écrire les mots du langages et quels sont les mots qui ont un sens particulier (mots réservés)
  - Des règles syntaxiques :
    elles indiquent comment agencer les mots du langages pour former un programme correct
  - Des règles sémantiques :
    elles indiquent le sens, l'opération qu'effectuera l'agencement des phrases du programme.



## Les règles lexicales

- Les règles lexicales indiquent
  - Quels sont les mots qui ont un sens particulier dans le langage (mots réservés) et qu'on ne peut pas utiliser pour autre chose
  - De quelle manière écrire les noms que l'on peut donner aux divers composants d'un programme (identificateur)
  - De quelle manière écrire les informations de base que l'on manipule (nombre, mots,…)

# Exemple : les identificateurs

- Un identificateur est un nom que l'on donne à un élément du programme (afin de l'identifier).
- Dans de nombreux langages, un identificateur doit
  - Commencer par une lettre (de A à Z) ou le caractère souligné ( _ )
  - Ne contenir ensuite que des lettres, chiffres ou caractère souligné
  - Doit être différent de tout mot réservé
- Exemples : s, _s, Age, TauxTVA, taux_tva, s56
- Contre exemples : 2m, taux-tva, taille&poids
- Certains langages font la différence entre minuscules et majuscules (C,C++,Java). Dans ce cas, Ident et ident sont deux identificateurs différents.



# Les règles syntaxiques

- Les règles syntaxiques se décrivent par une grammaire. Les grammaires utilisées en informatique sont beaucoup plus simples que celles utilisées en français. On peut les décrire en utilisant la notation BNF (Backus-Naur form)



# Grammaire BNF

- Une grammaire BNF se décrit à l'aide de deux types de symboles
  - Les symboles terminaux : ils représentent directement un mot du langage. On les soulignera pour les distinguer.
  - Les symboles non terminaux : ils représentent un nom que l'on donne à une partie de phrase
- On utilise
  - une flèche → pour indiquer comment décomposer un symbole non terminal
  - la barre verticale | pour indiquer les différentes décompositions possibles



# Exemple de grammaire française

- phrase → sujet verbe complément
  *une phrase se compose d'un sujet suivi d'un verbe puis d'un complément*
- sujet → je | tu | il | groupe_nominal
  *le sujet est soit le mot « je », soit le mot « tu », soit le mot « il », soit un groupe nominal*
- groupe_nominal → article nom
  | article adjectif nom
  *un groupe nominal est un article suivi d'un nom ou bien un article suivi d'un adjectif et d'un nom*



# Utilisation

- Dans la suite du cours, on utilisera
  - Des exemples pour présenter les quelques règles lexicales importantes
  - La notation BNF pour décrire les règles syntaxiques (parfois simplifiées)
  - Des explications en français pour les règles sémantiques
- Rappel : les notions seront présentées en pseudo-langage et C++ (parfois aussi en Java ou C)



# Les types



# Notion de type

- Un programme manipule des informations pour calculer un résultat tout comme une recette de cuisine manipule des ingrédients pour obtenir un produit fini.
- Il existe différentes sortes d'informations qui ont des caractéristiques différentes et ne se manipulent pas de la même manière, tout comme il y a des différences entre les ingrédients d'une recette (liquides, poudres, solides,…)



# Type

- Un type est le nom que l'on donne à une catégorie d'informations. Le type d'une information détermine
  - La manière dont elle va être représentée dans la mémoire de l'ordinateur (place occupée,…) et dans le source du programme
  - Les opérations que l'on peut effectuer sur ces informations (ex: addition possible sur des nombres, mais pas sur du texte)



# Comparaison en cuisine

*Le type bouteille d'eau*

*Le type bouteille de vin*

*Le type boîte à œufs*

*Le type casserole*

*Un exemplaire de casserole (on parlera d'une instance de casserole)*

*Un autre exemplaire*

# Les types de base

- Les langages définissent des types de base pour représenter les informations les plus courantes. On verra par la suite que le programmeur peut définir de nouveaux types à partir des types de base (ex: définition du type complexe en utilisant le type réel)



# Liste des types de base

- Les types de base les plus courants sont :
  - Booléen
  - Caractère
  - Entier
  - Réel (simple ou double précision)
  - Chaîne de caractères (peut aussi être considéré comme un type composé)



# Le type booléen

- type des informations qui n'ont que deux valeurs possibles (vrai/faux)
- Les opérations possibles sur des valeurs de ce type sont les opérations logiques (et, ou, non,…)
- Exemple : l'information « un CDROM se trouve dans le lecteur » est soit vraie, soit fausse



# Le type caractère

- Type des informations qui contiennent un caractère **et un seul** (lettre, chiffre, symbole de ponctuation, autres caractères imprimables et caractères de contrôle)
- Les opérations possibles sur des valeurs de ce type restent limités (obtention du code ascii du caractère, ajout à une chaîne de caractère, obtention du caractère précédent ou suivant)
- Exemple : le caractère « A », le caractère « a », le caractère « _ », le caractère « ( », le caractère « Ctrl-M »



# Le type entier

- Type des informations qui contiennent un nombre entier (pas de chiffres après la virgule)
- Les langages proposent souvent différent types entiers qui se distinguent par l'intervalle de nombres qu'ils peuvent contenir (et donc la place qu'ils occupent en mémoire)
- Les opérations possibles sur des valeurs de ce type contiennent bien sûr les opérations arithmétiques classiques (+,-,*,/)
- Exemple : le nombre 26



# Le type réel

- Type des informations qui contiennent une représentation approchée d'un nombre réel
- Les opérations possibles sur des valeurs de ce type contiennent bien sûr les opérations arithmétiques classiques (+,-,*,/) et les fonctions mathématiques usuelles
- Exemple : le nombre 3.14159265



# Le type chaîne de caractères

- Type des variables qui contiennent une suite de caractères (mot, suite de mots,…)
- Les opérations possibles sur des valeurs de ce type sont diverses : concaténation (mise bout à bout de deux chaînes), extraction d'une partie de la chaîne, …
- Exemple : la chaîne « Le chat de ma grand-mère »



# Nom des types

| En pseudo-langage | En C++ |
|---|---|
| booléen | bool |
| caractère | char |
| entier | int (et d'autres) |
| réel | float *ou* double |
| chaîne | string |



# Les constantes

# Les constantes

- Une constante est une information qui est directement incluse dans le programme et qui va être transformée. Les constantes jouent le même rôle que les denrées manipulées par la recette de cuisine
- Il existe deux types de constantes
  - Constantes littérales
  - Constantes nommées



# Constantes littérales

- Une constante littérale est une information que l'on place directement dans le programme quand on en a besoin (et que l'on va répéter chaque fois qu'on l'utilise).
- Selon le type de l'information, on les note de différentes manières.



# Constantes booléenne

- Notation en pseudo-langage
  - vrai
  - faux
- Notation en C++ (en minuscules)
  - true
  - false



# Constantes caractère

- En pseudo-langage comme en C++, les constantes caractère se notent entre guillemets simples. Un seul caractère doit se trouver entre ces guillemets !
- Exemples : 'A'
- Certains caractères se notent en C++ de manière spécifique grâce à une séquence d'échappement. Le symbole antislash (\) annonce que le(s) prochain(s) symbole(s) doi(ven)t être interprété(s) spécialement. L'ensemble de ces symboles ne représente qu'un seul caractère.



# Des caractères spéciaux en C++

- '\n' représente le caractère saut de ligne (newline)
- '\r' représente le caractère retour chariot (return)
- '\t' représente le caractère tabulation
- '\' ' représente le caractère guillemet simple
- '\" ' représente le caractère guillemet double
- '\\' représente le caractère antislash
- '\b' représente le caractère backspace
- '\f' représente le caractère form feed
- '\40' représente le caractère de code ascii octal 40 (32 en décimal)
- '\x20' représente le caractère de code ascii hexadécimal 20 (32 en décimal)



# Constantes entières

- Notation décimale habituelle (base 10) :
  5
  +5
  -5
- Notation octale en C/C++/Java (base 8) :
  023
  *le zéro initial indique qu'il s'agit d'un nombre en octal*
- Notation hexadécimale en C/C++/Java (base 16) :
  0x13
  *le 0x initial indique qu'il s'agit d'un nombre en hexadécimal*



# Constantes réelles

- Notation habituelle anglaise (le point remplace la virgule) :
  1
  1.5
  -10.567
- Notation ingénieur
  -5.698E-12



# Constantes chaînes de caractères

- Les chaînes de caractères se notent entre guillemets doubles. Elles peuvent contenir des caractères spéciaux.
  "Bonjour"
  "Bonjour tout le monde\n"
  "Des 'guillemets simples'."
  "Des \"guillemets doubles\"."
- Ne pas confondre un caractère 'A' avec une chaîne qui ne contient qu'un seul caractère "A"



# Les constantes nommées

- Les constantes présentées jusqu'ici étaient des constantes littérales. Quand on utilise plusieurs fois la même valeur avec la même signification, il est plus pratique de donner un nom à la valeur.
- Le nom donné à la valeur doit être un identificateur valide.
- Le type de la constante doit être précisé dans la déclaration
- Dès que l'ordinateur rencontre le nom d'une constante (après la déclaration), il la remplace par sa valeur
- Les constantes doivent être déclarées avant les variables

## Déclaration de constantes en PL

- BNF en pseudo-langage
  déclarations_constantes →
  constantes liste_décl_constantes
  liste_décl_constantes →
  déclaration_une_constante
  | déclaration_une_constante liste_décl_constantes
  déclaration_une_constante →
  identificateur : type = valeur
- Exemple :
  constantes
  pi : réel = 3.14159265
  message : chaîne = "Bienvenue"



## Déclaration de constantes en C++

- BNF en C++
  liste_décl_constantes →
  déclaration_une_constante
  | déclaration_une_constante liste_décl_constantes
  déclaration_une_constante →
  const type identificateur = valeur ;
- Exemple :
  const float pi = 3.14159265 ;
  const string message = "Bienvenue";



## Constante littérale ou nommée

- On utilise une constante littérale lorsque l'on n'utilise l'information qu'à un seul endroit du programme et qu'on ne risque pas de la changer dans une future version du programme.
- On **doit utiliser** une constante nommée dès qu'on utilise la même information (avec le même sens) à deux endroits différents du programme (par exemple un taux de TVA), ou bien qu'on risque de changer la valeur dans une future version du programme.



## Utilité des constantes

- Intérêt :
  - Il est plus facile d'utiliser le nom pi que de répéter 3.14159265
  - si la valeur change (TVA par ex.), on ne doit faire qu'une seule modification dans le programme
  - L'utilisation de constantes facilite la compréhension du programme (le nombre 800 ne signifie rien à priori tandis qu'une constante largeur_fenetre valant 800 sera parfaitement claire)



## Constance des constantes

- Une fois définie, une **constante ne peut plus changer** de valeur au cours de l'exécution du programme (d'où le nom de constante).
- S'il faut toutefois modifier la valeur (ex: taux de TVA), il faut arrêter le programme, modifier le source, recompiler et réexécuter le programme



## Les variables



## Les variables

- Dans une recette, il faut des récipients pour manipuler les denrées.
- Pour manipuler et transformer des informations, un programme a besoin de récipients : ce sont les variables.
- Une variable contient une information d'un certain type.
- Le contenu d'une variable peut être modifié à volonté.
- Quand l'ordinateur rencontre le nom d'une variable dans une expression à calculer, il le remplace par le contenu de cette variable.



## Déclaration de variable

- Avant de pouvoir utiliser une variable, il faut la déclarer, c'est à dire demander à l'ordinateur de créer le « récipient » du type indiqué et lui donner un nom pour le distinguer des autres.
- Les variables que l'on utilise tout au long du programme (variables globales) se définissent avant le début du programme (juste après les constantes éventuelles)



## Déclaration de variables en PL

- BNF en pseudo-langage
  déclarations_variables →
  variables liste_décl_variables
  liste_décl_variables →
  déclaration_variable
  | déclaration_variable liste_décl_variables
  déclaration_variable →
  liste_identificateurs : type
  liste_identificateur →
  identificateur | identificateur , liste_identificateur
- Exemple :
  variables
  i,j,k : entier
  prénom : chaine

# Déclaration de variables en C++

- BNF en C++
  liste_décl_variables →
  déclaration_variable
  | déclaration_variable liste_décl_variables
  déclaration_une_variable →
  type liste_identificateur ;
- Exemple :
  int i,j,k;
  string prenom;



# Ordre

- On notera bien que
  - En pseudo langage, on met d'abord le nom de la variable et ensuite son type
  - Qu'en C/C++/Java, on met le type puis le nom de la variable
- On notera aussi que chaque déclaration (ou instruction) en C/C++/Java se termine par un point-virgule



# Initialisation

- Quand on déclare une variable, l'ordinateur crée un récipient mais ne donne aucune garantie sur le contenu du récipient. En fait, dans la plupart des langages, le contenu d'une variable nouvellement créée est aléatoire ou incohérent.
- Avant d'utiliser une variable, il faut être sûr de son contenu. Il faut donc, dès le début du programme, placer une valeur connue dans les variables. C'est l'initialisation (faire la vaisselle)



# Règle d'or

Le contenu d'une variable ne doit JAMAIS être utilisé avant qu'elle n'ait été initialisée.

- L'initialisation consiste à affecter une constante à la variable.



# L'affectation

- L'opération qui permet de placer une valeur dans une variable s'appelle l'affectation. Le contenu précédent de la variable est définitivement perdu.
- Cette opération s'effectue en 2 étapes
  1. On calcule la valeur à placer dans la variable en calculant le résultat d'une expression (à droite du symbole d'affectation)
  2. Le contenu de la variable affectée est alors **remplacé** par le résultat calculé.



# L'affectation en PL

- BNF :
  affectation →
  identificateur ← expression
- Exemple
  programme essai_affectation
  variables
  a : entier
  début
  a ← 1
  a ← a+1
  fin



# En pseudo-langage

- La flèche vers la gauche est le symbole de l'affectation en PL. Elle symbolise bien qu'on calcule le résultat de l'expression à la droite du symbole d'affection pour le ranger dans la variable à la gauche du symbole.

**Seul un nom de variable** peut se trouver du côté gauche du symbole d'affectation

- L'affectation a ← b+1 peut se lire « la variable a **reçoit** la valeur [de l'expression] b+1 »



# En C++

- Le principe est le même : on calcule le résultat de l'expression à la droite du symbole d'affection pour le ranger dans la variable à la gauche du symbole.

**Seul un nom de variable** peut se trouver du côté gauche du symbole d'affectation

- Seul change le symbole de l'affectation qui, en C,C++ et Java, est le signe =

- L'usage de ce symbole ne signifie en rien que l'on a affaire à une équation mathématique !!



# L'affectation en C++

- BNF :
  affectation →
  identificateur = expression ;
- Exemple

```
int a;
int main()
{
  a=1;
  a=a+1;
}
```

# Constitution d'un programme

Un premier schéma



# En pseudo-langage

- BNF
  programme →
  programme identificateur
  déclarations_constantes
  déclarations_variables
  début
  séquence_instructions
  fin



# Exemple en pseudo-langage

```
programme cercle
constantes
   pi : réel = 3.14159265
variables
   rayon, périmètre : réel
début
   rayon ←10
   périmètre ←2*pi*rayon
fin
```



# En C++

- BNF
  programme →
  liste_include
  déclarations_constantes
  déclarations_variables
  int main()
  {
  séquence_instructions
  }



# Exemple en C++

```
#include <iostream>
using namespace std;
const float pi=3.14159265;
float rayon,perimetre;
int main()
{
   rayon=10;
   perimetre=2*pi*rayon;
}
```



# Les entrées/sorties



# Dialogue avec l'utilisateur

- Un programme doit être en mesure d'interagir avec l'utilisateur pour
  - Lui demander certaines informations (ex: le rayon du cercle)
  - Lui communiquer certains résultats (ex: le périmètre du cercle)
- C'est le rôle des entrée/sorties. Le sujet étant vaste, on n'en présente que l'essentiel dans un premier temps.



# Opération d'affichage

- L'affichage consiste à communiquer sur l'écran de l'ordinateur une information qui peut être
  - Une constante
  - Le contenu d'une variable
  - Le résultat d'une expression
- En pseudo-langage, on écrira
  afficher("le périmètre vaut ",périmètre)
- L'instruction se nomme afficher, les éléments à afficher sont cités entre parenthèses et séparés par des virgules
- On pourrait aussi écrire
  afficher("le périmètre vaut ",2*pi*rayon)



# Opération de lecture

- La lecture consiste à lire une information depuis le clavier de l'ordinateur et à ranger cette information dans une variable
- En pseudo-langage, on écrira
  lire(rayon)
- L'instruction se nomme lire et l'on doit trouver une variable entre les parenthèses
- Le type de la variable détermine ce que l'on attend au clavier (entier, réel, chaîne,…)
- Une lecture peut servir d'initialisation de variable

## Guider l'utilisateur

- Avant toute opération de lecture, il faut informer l'utilisateur par un affichage qu'on attend qu'il saisisse une information (et préciser quelle est cette information). Sinon, il pensera que votre programme est planté ou tapera une autre information. On parle de message d'invite (prompt).
- De même, il faut que l'utilisateur sache ce que l'on affiche



## Exemple en pseudo-langage

```
programme cercle
constantes
   pi : réel = 3.14159265
variables
   rayon, périmètre : réel
début
   afficher("Entrez le rayon d'un cercle :")
   lire(rayon)
   périmètre ←2*pi*rayon
    afficher("le périmètre vaut ",périmètre)
fin
```



## En C++

- afficher(*info1*,*info2*,*info3*) se traduit par
  cout << *info1* << *info2* << *info3* ;
- lire(*variable*) se traduit par
  cin >> *variable*;
- cout représente l'écran et cin le clavier. Le « verbe » est l'opérateur << ou >>
- On remarquera que l'orientation du symbole (>> ou <<) indique dans quel sens transite l'information.
- Pour pouvoir effectuer des entrées/sorties, il faut spécifier au début du programme
  #include <iostream>
  using namespace std;



## Exemple en C++

```
#include <iostream>
using namespace std;
const float pi=3.14159265;
float rayon,perimetre;
int main()
{
   cout << "Entrez le rayon d'un cercle : ";
   cin >> rayon;
   perimetre=2*pi*rayon;
   cout << "le périmètre vaut " << perimetre << "\n";
}
```



## Modification de l'affichage

- Il existe selon les langages différents outils pour changer la manière d'afficher ou de lire des informations (exemple : affichage des nombres dans une autre base, alignement à gauche ou à droite,…)
- En C++, on utilise des manipulateurs pour modifier l'affichage ou la lecture (cf. document annexe)



## Les opérateurs



## Différents opérateurs

- Un opérateur est une fonction qui se note de manière particulière (exemple : l'addition). Les paramètres d'un opérateur sont appelé des opérandes. Un opérateur peut avoir un seul paramètre (opérateur unaire), ou deux (opérateurs binaires) voire plus.
- On distingue plusieurs notations :
  - Préfixe : ++a mais aussi -a
    l'opérateur est devant les opérandes
  - Infixe : a+b
    l'opérateur est placé entre ses opérandes
  - Suffixe : a++
    l'opérateur se trouve après les opérandes



## Importance des types

- Un même symbole peut représenter des opérateurs différents selon le type des opérandes auquel il est appliqué
- Ex:
  1+1 (résultat entier)
  1.5+3.5 (résultat réel)
  "ABC"+"DEF" (concaténation, résultat de type chaîne de caractères)



## Les opérateurs relationnels

- Les opérateurs relationnels servent à comparer des valeurs.
- Ce sont les tests les plus élémentaires dans les langages (c-à-d les expressions booléennes les plus simples après les variables booléennes)
- On ne peut bien sûr comparer que ce qui est comparable.
  - On peut en général vérifier l'égalité de deux éléments de même type
  - On peut bien sûr tester l'égalité et l'ordre sur les entiers et les réels
  - Le reste dépend souvent du langage utilisé. Certains langages (C++ par exemple) permettent de définir de nouvelles relations d'ordre sur de nouveaux types

# Les opérateurs relationnels

- On trouve bien sûr le test d'égalité et de différence (=,≠)
- Les opérations classiques de comparaison (<,>,≤,≥)
- Chacun de ces opérateurs renvoie soit vrai soit faux selon le résultat du test (résultat de type booléen). Exemple :
  2 < 3 est vrai
  3 < 3 est faux



# BNF (pseudo langage)

Expression_booléenne →
expression opérateur_relationnel expression
opérateur_relationnel →
≤ | ≥ | ≦ | ≧ | ⩵ | ≠

- Le résultat d'une expression booléenne (vrai ou faux) peut très bien être rangé dans une variable ou être affiché :
  b ← 2<3 *b doit être du type booléen*
  afficher(3=4)



# Notation

| En pseudo-langage | En C++ |
|---|---|
| < | < |
| > | > |
| ≤ | <= |
| ≥ | >= |
| = | = = |
| ≠ | != |

Le point d'exclamation en C++ représente la négation.
!= signifie donc « pas égal », c'est à dire différent



 = et = = 

- Attention ! En C++, l'opérateur d'affectation se note par un = et le test d'égalité se note par un double signe égal (= =) l'un à la suite de l'autre. **Il ne faut surtout pas confondre les deux.**
- L'opérateur d'affectation sert à placer une valeur dans une variable. À sa gauche, on doit trouver le nom d'une variable, à sa droite, l'expression calculant la valeur à placer dans la variable. L'opérateur d'affectation ne renvoie pas de résultat. Il constitue une instruction à part entière.
- Le test d'égalité sert à comparer deux valeurs. On doit trouver de chaque côté une expression calculant les valeurs à comparer. Le test d'égalité renvoie un résultat qui est soit vrai, soit faux. Le test d'égalité n'est pas une instruction



# Méfiance

- Attention ! Les calculs sur les réels sont toujours approchés. Il faut en tenir compte dans les comparaison.
- Exemple : 3*(1/3) a de fortes chances d'être différent de 1



# Les opérateurs logiques

- Les opérateurs logiques (et, ou, non,…) servent à combiner des tests élémentaires pour former des conditions plus complexes.
- Les opérandes de ces opérateurs sont forcément des valeurs de type booléen.
- Le résultat de ces opérateurs est de type booléen
- Plus de détails dans le chapitre sur la logique



# Les opérateurs mathématiques

- On retrouve bien sûr les opérateurs mathématiques classiques
  - Addition +
  - Soustraction –
  - Multiplication *
  - Division /
- Ces opérateurs s'appliquent à des nombres. Le type du résultat dépend du type des opérandes. Dès qu'un opérande est un réel, le résultat sera un réel.



# La division entière

- On veut parfois obtenir un quotient entier et un reste quand on effectue une division entre deux entiers
- On veut donc une relation de la forme
  dividende=diviseur*quotient+reste
  avec reste<quotient
- En pseudo langage, l'opérateur donnant le quotient de la division entière est **div**, l'opérateur donnant le reste de la division entière est **mod** (pour modulo).



# Exemple de division entière

| 23 | 5 |
|---|---|
| 3 | 4 |

- 23=4*5+3
- Si on veut obtenir le quotient entier, on écrit 23 div 5 (ce qui vaut 4)
- Si on veut obtenir le reste de la division entière, on écrit 23 mod 5 (ce qui vaut 3)
- Attention ! Le reste de la division est forcément compris dans l'intervalle [0,diviseur-1]

## Utilité de la division entière

- La division entière est extrêmement utilisée en informatique. Il faut penser à l'utiliser chaque fois qu'on a une répétition périodique
- Exemple : si le premier janvier est un lundi, à quel jour correspond le 23 janvier ?
  Réponse : 23-1 mod 7 = 1, ce sera un mardi !
- Dans ce genre de calcul, il faut toujours penser à se ramener à l'intervalle [0,diviseur-1], donc, dans l'exemple, coder lundi par 0, mardi par 1, … et dimanche par 6.



## En C++

- En C++, mod se note %  Exemple : 23 % 7 vaut 2
- En C++, div se note / (comme la division sur les réels). Exemple : 23 / 7 vaut 3
- Règle : quand le symbole / porte sur deux entiers, c'est une division entière. Quand l'un des opérandes est réel, c'est une division réelle.
  Exemples : 7/2 vaut 3
  7.0/2 vaut 3.5
  7/2.0 vaut 3.5
  7.0/2.0 vaut 3.5

- Si on veut une division réelle de deux entiers, il faut forcer l'un des entiers à être considéré comme un réel (ajouter .0 à une constante ou utiliser un transtypage)



## Division/multiplication par deux

- L'ordinateur travaille en binaire. La multiplication et la division entière par deux sont des cas particuliers que l'ordinateur sait effectuer particulièrement facilement.
- En C++, x << y (x et y entiers) représente $x*2^y$
  En binaire, il s'agit d'un simple décalage à gauche (ajout de y zéros à droite du nombre)
- En C++, x >> y (x et y entiers) représente x div $2^y$
  En binaire, il s'agit d'un simple décalage à droite (suppression des y chiffres à droite du nombre)



## Opérations bit à bit

- On peut en C++ effectuer des opérations logiques simultanément sur chacun des bits de la représentation d'un nombre entier.
- Très utile quand on travaille sur une représentation binaire
- Exemples en C++ :
  0xA & 0x6 vaut 0x2 (1010 et bit à bit 0110=0010)
  0xA | 0x6 vaut 0xE (1010 ou bit à bit 0110=1110)
  0xA ^ 0x6 vaut 0xC (1010 xor bit à bit 0110=1100)
  ~0xA vaut 0x5 (complément à 1 de 1010=0101)
- Attention à ne pas confondre ces opérations avec le **et**, le **ou** et le **non** logiques sur des booléens. Les opérateurs bit à bit renvoient des **entiers**, les opérateurs logiques renvoient des **booléens**



## Fonctions mathématiques

- On peut utiliser les fonctions mathématiques classiques sin(x), cos(x),…,atan(x),…
- La racine carrée s'écrit racine(x) en PL et sqrt(x) en C++. Le log(x) de C++ est le log népérien.
- Pour utiliser ces fonctions en C++, il faut rajouter
  `#include <cmath>`
  et rajouter l'option –lm à la ligne de compilation
  `g++ -o prg prg.cc -lm`



## Transtypage

- Il existe un opérateur spécial pour transformer une valeur d'un certain type en une valeur d'un autre type : c'est la conversion de type ou transtypage (ou cast en C++)
- Cette opération ne marche que dans des cas très restreints (en général, uniquement entre réel et entier)
- Notation en PL : nouveautype(expression)
  Notation en C/C++ : (nouveautype)expression
- Exemple en PL : entier(4.5)
  Exemple en C/C++ : (float)4
- En C++, la transformation d'un entier en réel se fait en tronquant la partie décimale



## Transtypage (2)

- Exemples en C++ :
  `reel=1/(float)2;`
  `reel=((float)1)/2;`
  *attention à la priorité des opérateurs*
  *les parenthèses sont ici indispensables*
- En règle générale, le transtypage est une opération dangereuse qu'il ne faut utiliser qu'en toute connaissance de cause. Il faut donc la proscrire (sauf pour les conversions entre entier et réel)



## Incrémentation/décrémentation

- L'incrémentation consiste à ajouter 1 à la valeur d'une variable ; la décrémentation consiste à soustraire 1 à la valeur d'une variable.
- En pseudo langage, on pourra noter inc(a) au lieu de a ←a+1 et dec(a) au lieu de a ←a-1
- En C++, on note a++ et a--
- Il existe une différence subtile en C++ entre pré-(in/dé)crémentation et post-(in/dé)crémentation.



## Autres opérateurs

- Il existent bien sûr d'autres opérateurs qui varient d'un langage à l'autre.
- C++ est particulièrement fourni en opérateurs divers et variés et aux notations exotiques.

# Les expressions



# Expression

- Une expression représente un calcul que l'ordinateur doit mener pour obtenir une valeur (3+4 par exemple)
- Une expression représente donc un résultat calculé, une valeur qu'il faut utiliser en la rangeant dans une variable, en l'affichant ou en la passant à une procédure ou fonction.
- Cela n'a pas de sens de calculer un résultat pour le laisser tomber aussitôt. **Il faut utiliser la valeur calculée**



# Différence Expression/Instruction

- Une instruction est un ordre donné à la machine. L'instruction va permettre de réaliser une certaine action.
- Il ne faut pas confondre instruction et expression. Une expression représente une certaine valeur (calculée) tandis qu'une instruction ne représente aucune valeur.



# Ordre d'évaluation

- L'ordinateur doit savoir précisément dans quel ordre il doit effectuer les calculs de l'expression (ordre d'évaluation). Pour le préciser, il existe un certain nombre de règles qui permettent de fixer l'ordre d'évaluation d'une expression en évitant d'utiliser trop de parenthèses.



# Commutativité

- Un opérateur ayant deux opérandes est commutatif si l'on peut permuter ses opérandes sans que cela modifie le résultat
- Ex: a+b=b+a
- La plupart des opérateurs ne sont pas commutatifs
- En informatique, la machine effectue toujours les calculs dans le même ordre (mais on ne sait pas forcément lequel)



# Associativité (math.)

- Un opérateur est associatif si l'on peut changer l'ordre dans lequel on effectue les opérations sans modifier le résultat (c-à-d déplacer les parenthèses sans changer le résultat)
- Ex: (a+b)+c=a+(b+c)
- Quand on sait qu'un opérateur est associatif, on peut simplifier l'écriture en supprimant des parenthèses
- Ex: (a+b)+c=a+(b+c)=a+b+c



# « Associativité » (info.)

- En informatique, on parle aussi de règle d'associativité pour des opérateurs qui ne sont pas forcément associatifs au sens mathématique du terme. Une règle d'associativité permet simplement de préciser dans quel ordre les opérations seront effectuées en l'absence de parenthèses (afin de les supprimer pour faciliter la lecture)



# Associativité à droite et à gauche

- Un opérateur O est associatif à droite si la notation
  a O b O c
  signifie en fait
  a O (b O c)
- Un opérateur O est associatif à gauche si la notation
  a O b O c
  signifie en fait
  (a O b) O c
- En C++, les opérateurs unaires et d'affectation sont associatifs à droite. Tous les autres sont associatifs à gauche.



# Priorité des opérateurs

- Vous savez tous que la multiplication a priorité sur l'addition. Cela signifie, qu'en absence de parenthèses, on calcule d'abord les multiplications et ensuite seulement les additions.
- En informatique, il en va de même. Certains opérateurs sont plus prioritaires que d'autres. En l'absence de parenthèses, on commence par calculer le résultat des opérateurs les plus prioritaires.
- Si des opérateurs ont même priorité, on utilise les règles d'associativité
- Les parenthèses servent à rendre un calcul prioritaire

# Méfiance

- Dans certains langages, l'ordre d'évaluation entre opérateurs de même priorité peut ne pas être spécifié. Cela signifie qu'il ne faut pas faire d'hypothèse sur l'ordre des calculs sans quoi on écrit un programme qui sera faux dans certains cas.



# Exemple en C++

- Exemple de priorités, du plus prioritaire au moins prioritaire :
  +, <<, <, !=, &&
- Donc 1 << 2+3<6 && 1+2+3!=6
  s'évalue
  ((1 << (2+3))<6) && ((1+2+3)!=6)
- Conclusion : en cas de doute sur l'ordre d'évaluation, mettez des parenthèses (mais pas trop !)



# La logique



# Un peu « d'intelligence »

- Tout programme doit effectuer des tests pour adapter son comportement à son environnement.
- Il faut au moins être en mesure de décider si oui ou non (booléen) il faut exécuter certaines des instructions d'un programme
- Pour cela, il faut d'abord savoir exprimer sous quelle condition il faut exécuter ces instructions particulières



# Arbre de décision

- Une méthode particulièrement simple de « programmation » consiste à établir des arbres de décision.
- L'exécution part de la racine de l'arbre (en haut)
- Un nœud de l'arbre contient un test à effectuer.
- Selon le résultat du test, on poursuit l'exécution en suivant la branche qui est étiquetée par le résultat du test

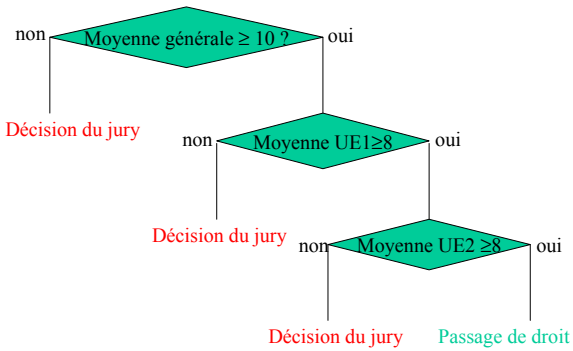


# Exemple: condition de passage en deuxième année

- Pour passer en deuxième année, il faut remplir les conditions suivantes
  - Avoir une moyenne générale supérieure ou égale à 10
  - Avoir une moyenne dans l'UE1 supérieure ou égale à 8
  - Avoir une moyenne dans l'UE2 supérieure ou égale à 8



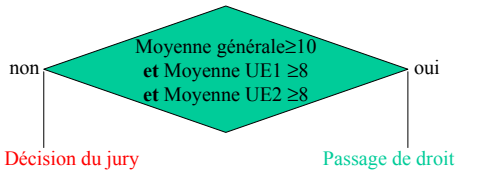
# Arbre de décision





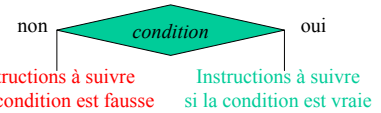

# Un deuxième arbre

- On peut simplifier l'arbre de décision précédent en utilisant une seule condition formée à partir des conditions élémentaires et de certains connecteurs logiques






# Traduction en pseudo-langage




- devient

si condition alors
    Instructions à suivre si la condition est vraie
sinon
    Instructions à suivre si la condition est fausse
fin si

# Les connecteurs logiques

- Un connecteur logique sert à créer des conditions composées à partir de conditions élémentaires (atomiques). Les expressions reliées par un connecteur logique sont forcément des expressions booléennes
- Il existe différents connecteurs logiques :
  - Très utilisés : Non, Et, Ou, Ou exclusif
  - Moins fréquents : Implication, Équivalence
- Le sens de ces connecteurs est précisément défini par des tables de vérité qui indiquent la valeur de l'expression logique en fonction des valeurs des opérandes



# La négation logique (NON)

- (**non** A) est vrai si et seulement si A est faux

Table de vérité :

| *A* | **non** *A* |
|---|---|
| faux | vrai |
| vrai | faux |



# La conjonction logique (ET)

- (A **et** B) est vrai si et seulement si A est vrai ainsi que B. Dès que l'un des deux est faux, (A **et** B) est faux.

Table de vérité :

| *A* | *B* | *A* **et** *B* |
|---|---|---|
| faux | faux | faux |
| faux | vrai | faux |
| vrai | faux | faux |
| vrai | vrai | vrai |



# La disjonction logique (OU)

- (A **ou** B) est faux si et seulement si A est faux ainsi que B. Dès que l'un des deux est vrai (éventuellement les deux), (A **ou** B) est vrai.

Table de vérité :

| *A* | *B* | *A* **ou** *B* |
|---|---|---|
| faux | faux | faux |
| faux | vrai | vrai |
| vrai | faux | vrai |
| vrai | vrai | vrai |



# Le ou exclusif (OUX)

- (A **oux** B) est faux si et seulement si A a la même valeur que B. Dès que l'un des deux est vrai tandis que l'autre est faux, (A **oux** B) est vrai. Autrement dit, il faut que l'un des deux opérandes soient vrais, mais pas les deux en même temps.

Table de vérité :

| *A* | *B* | *A* **ou** *B* |
|---|---|---|
| faux | faux | faux |
| faux | vrai | vrai |
| vrai | faux | vrai |
| vrai | vrai | faux |



# L'implication

- (A $\Rightarrow$ B) est faux si et seulement si A est vrai et B est faux. Il signifie : si A est vrai, alors B l'est aussi

Table de vérité :

| *A* | *B* | *A* **ou** *B* |
|---|---|---|
| faux | faux | vrai |
| faux | vrai | vrai |
| vrai | faux | faux |
| vrai | vrai | vrai |



# L'équivalence

- (A $\Leftrightarrow$ B) est vrai si et seulement si A a la même valeur que B. Dès que l'un des deux est vrai tandis que l'autre est faux, (A $\Leftrightarrow$ B) est faux.

Table de vérité :

| *A* | *B* | *A* **ou** *B* |
|---|---|---|
| faux | faux | vrai |
| faux | vrai | faux |
| vrai | faux | faux |
| vrai | vrai | vrai |



# Propriétés

- Commutativité :
  - a et b = b et a
  - a ou b = b ou a
  - a oux b = b oux a
- Associativité
  - (a et b) et c = a et (b et c)
  - (a ou b) ou c = a ou (b ou c)
  - (a oux b) oux c = a oux (b oux c)



# Propriétés

- Distributivité :
  - (a et b) ou c = (a ou c) et (b ou c)
  - (a ou b) et c = (a et c) ou (b et c)
- Élément neutre :
  - a et vrai = a
  - a ou faux = a
  - a oux faux = a

# Propriétés

- Élément absorbant :
  - a et faux = faux
  - a ou vrai = vrai
- Idempotence :
  - a ou a = a
  - a et a = a
- Négation
  - a et (non a) = faux
  - a ou (non a) = vrai
  - a oux (non a) = vrai
- Double négation
  - non(non a)=a



# Propriétés

- Lois de De Morgan
  - non (a et b) = (non a) ou (non b)
  - non (a ou b) = (non b) et (non a)
- Ou exclusif
  - a oux b = (a et non b) ou (b et non a)
    = (a ou b) et ((non a) ou (non b))
- Implication
  - a ⇒ b = (non a) ou b
- Équivalence
  - a ⇔ b = (a ⇒ b ) et (b ⇒ a)
    = ((non a) ou b) et ((non b) ou a)
    = (a et b) ou ((non a) et (non b))



# Diverses notations

Si on note vrai par 1 et faux par 0, le **et** se comporte comme une multiplication et le **ou** comme une addition avec la règle spéciale 1+1=1. D'où la notation algébrique et les priorités habituelles : le **non** est le plus prioritaire et le **et** est prioritaire sur le **ou**

| Français | Anglais | Logique | Algébrique |
|---|---|---|---|
| non | not | ¬ | $\overline{x}$ |
| et | and | ∧ | . |
| ou | or | ∨ | + |
| oux | xor | | ⊕ |



# Exemple de simplification

(non a ou b) et a et non (a et b)

(non a ou b) et a et (non a ou non b)

(non a ou (b et non b)) et a

(non a ou faux) et a

non a et a

faux



# Évaluation des expressions logiques

- Il existe deux modes d'évaluation des expressions logiques (c-à-d deux méthodes pour calculer leur valeur) :
  - Évaluation complète
  - Évaluation partielle
- Il est impératif de savoir quelle méthode est employée par le langage qu'on utilise pour prendre les précautions nécessaires



# Évaluation complète

Elle se déroule en deux étapes

1. Toutes les conditions élémentaires de l'expression sont évaluées une par une
2. La valeur de vérité de l'expression est alors calculée en utilisant les règles des connecteurs logiques



# Exemple d'évaluation complète

- Soit x une variable entière contenant 5

x>3 *vrai* ou x<2 *faux*

ou

*vrai*



# Problème avec l'évaluation complète

- Soit x une variable entière contenant 0

x≠0 *faux* et 1/x>2 *Erreur !*

Le résultat est ici incalculable !

Pourtant, faux et n'importe quoi donne forcément faux !



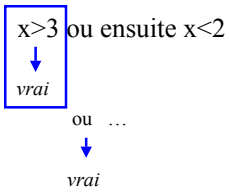
# Évaluation partielle

- Les conditions élémentaires de l'expression sont évaluées dans l'ordre de lecture de l'expression (de la gauche vers la droite). L'évaluation de l'expression s'arrête dès que l'on peut déterminer la valeur finale de l'expression (i.e. quand on rencontre un élément absorbant)
- On utilisera les notations
  - **et ensuite** (« and then » en Ada)
  - **ou ensuite** (« or else » en Ada)

  pour les opérateurs qui utilisent cette évaluation partielle

## Exemple d'évaluation partielle

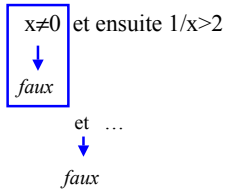
- Soit x une variable entière contenant 5






## Autre exemple d'évaluation partielle

- Soit x une variable entière contenant 0






## Évaluation partielle

- Avec une évaluation partielle, les opérateurs logiques ne sont plus commutatifs !
- Par exemple, x≠0 et ensuite 1/x>2 est une expression toujours évaluable tandis que 1/x>2 et ensuite x≠0 ne peut pas être calculé quand x vaut 0
- Quand une condition protège l'évaluation de calculs impossibles dans certains cas, elle doit être placée **devant** !



## Transformation (1/2)

- Quand on a besoin d'une évaluation partielle dans un langage qui n'offre que l'évaluation complète, il faut utiliser une structure conditionnelle
- Exemple : b←x≠0 et ensuite 1/x>2 devient

```
si x≠0 alors
    b ← 1/x>2
sinon
    b ← faux
fin si
```



## Transformation (2/2)

- Quand on a besoin d'une évaluation complète dans un langage qui n'offre que l'évaluation partielle, il faut décomposer les calculs et stocker les évaluations intermédiaires dans des variables temporaires
- Exemple : b←x≠0 et f(x)>2 devient

```
tmp ← f(x)>2
b ← x≠0 et ensuite tmp
```



## Comparaison des deux évaluations

- Inconvénients de l'évaluation complète :
  - On fait des calculs pour rien
  - Cela complique parfois les tests (cf. la division par zéro)
- Avantages de l'évaluation complète :
  - Elle permet d'utiliser dans l'expression des fonctions ayant des effets de bord. Comme c'est une très mauvaise technique de programmation, ce n'est pas un avantage.



## En C++

- Le langage C++ ne propose que le **et ensuite** et le **ou ensuite**
- Notations :

| Pseudo-langage | C++ |
|---|---|
| non | ! |
| et ensuite | && |
| ou ensuite | \|\| |

- Ne pas confondre le et logique (&&) qui est une fonction de B×B→B avec le et bit à bit (&) qui est une fonction de N×N→N. Idem pour le ou !



## Les structures de contrôle



## Instruction

- Une instruction est un ordre donné à l'ordinateur. L'affectation et les opérations d'entrée/sortie sont des instructions. Il existe bien sûr bien d'autres instructions qui existent en standard dans les langages
- Les expressions (arithmétiques, booléennes,…) ne sont pas des instructions!
- Pour délimiter les instructions, il faut
  - **Soit terminer chaque instruction par un point-virgule (C,C++,Java)**
  - Soit séparer les instructions par des points-virgules (Pascal)
  - Soit passer à la ligne (pseudo-langage)

# Structure de contrôle

- Une structure de contrôle va permettre d'enchaîner l'exécution de plusieurs instructions en déterminant comment et dans quel ordre elles doivent s'exécuter



# La séquence

- Dans une séquence d'instructions, les commandes sont exécutées les unes à la suite des autres, dans l'ordre où elles apparaissent dans le programme
- Ex:

```
Deplacer(1,2)
Deplacer(1,3)
Deplacer(2,1)
```

↓ Ordre d'exécution



# La séquence

- BNF:
  séquence_instructions→ ∅
  | instruction séquence_instructions
- Autrement dit, dans certains cas, une séquence peut être vide (ce qui revient à ne rien faire)



# Le bloc d'intructions

- Un bloc d'instruction est en fait une séquence d'instructions qui sont regroupées pour ne former qu'une seule instruction
- Exemple en pseudo langage

```
début
   Deplacer(1,2)
   Deplacer(1,3)
   Deplacer(2,1)
fin
```



# Le bloc d'instructions

- BNF en pseudo-langage:
  bloc_instructions→
  début séquence_instructions fin
  instruction →bloc_instructions
- Un bloc d'instructions est considéré comme une seule (grosse) instruction.
- Dans certains langages, certaines structures de contrôle ne peuvent porter que sur une seule instruction. On utilise alors un bloc chaque fois qu'il faut agir sur plusieurs instructions (début et fin jouent le rôle de parenthèses).
- Le bloc peut aussi servir à définir des variables locales.



# Le bloc d'instructions en C++

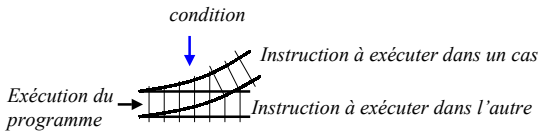
- BNF en C++:
  bloc_instructions→
  { séquence_instructions }
- Exemple en C++ :

```
{
   Deplacer(1,2);
   Deplacer(1,3);
   Deplacer(2,1);
}
```



# Structures conditionnelles

- Dans une structure conditionnelle, le résultat d'un test permet de choisir quelles instructions seront exécutées parmi un ensemble d'instructions prévues.
- Il s'agit d'un aiguillage






# La structure si : le si…fin si

- Le si…fin si permet de n'exécuter des instructions que lorsqu'une condition est vérifiée
- C'est donc un aiguillage qui a une voie correspondant à la condition vraie tandis que l'autre voie sert de raccourci pour sauter ces instructions là.
- Un si…fin si compte comme une seule instruction



# Le si…fin si en PL

- BNF :
  si_simple→
  si condition alors séquence_instructions fin si
  instruction →si_simple
- Exemple :

```
si b≠0 alors
   b←1/b
fin si
```

# Le si…fin si en C++

- BNF :
  si_simple→
  if ( condition ) une_seule_instruction
- Exemple :
  ```
  if (b!=0)
     b=1/b;
  ```
- Notez bien les parenthèses qui délimitent la condition et le fait que le if ne porte que sur une seule instruction



# Le si…fin si en C++

- Si plusieurs instructions doivent dépendre du if, il faut utiliser forcément un bloc d'instructions
- Exemple :
  ```
  if (b!=0)
  {
     b++;
     b=1/b;
  }
  ```



# Le si…sinon…fin si

- Le si…sinon…fin si permet de choisir entre deux séquences d'instructions celle qui doit être exécutée.
- C'est donc un aiguillage à deux voies avec sur chaque voie des instructions différentes. La condition détermine sur quelle voie va se poursuivre l'exécution du programme.
- Un si…sinon..fin si compte comme une seule instruction



# Le si…sinon..fin si en PL

- BNF :
  si_sinon→
  si condition alors
  séquence_instructions
  sinon
  séquence_instructions
  fin si
  *Exécuté quand la condition est vraie*
  *Exécuté quand la condition est fausse*
- Exemple :
  ```
  si a>b alors
     a←a-b
  sinon
     b←b-a
  fin si
  ```



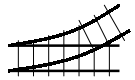
# Le si…sinon..fin si en C++

- BNF :
  si_sinon→
  if ( condition )
  une_seule_instruction
  else
  une_seule_instruction
  instruction →si_sinon
  *Exécuté quand la condition est vraie*
  *Exécuté quand la condition est fausse*
- Exemple :
  ```
  if (a>b)
     a=a-b;
  else
     b=b-a;
  ```



# Plus de deux cas ?

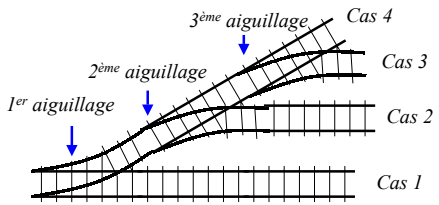
- Comment faire avec un aiguillage à deux voies (si…sinon…fin si) pour distinguer plus de deux cas ?





# Plus de deux cas

- Solution : on imbrique plusieurs aiguillages, comme dans une gare de triage






# Exemple

```
si a=b alors
   afficher("Les nombres sont égaux\n")
sinon
   si a>b alors
      afficher("a est plus grand que b\n")
   sinon
      afficher("b est plus grand que a\n")
   fin si
fin si
```

- Il faut passer des espaces pour aligner les si…sinon.. fin si et bien repérer qui dépend de quoi : c'est l'indentation

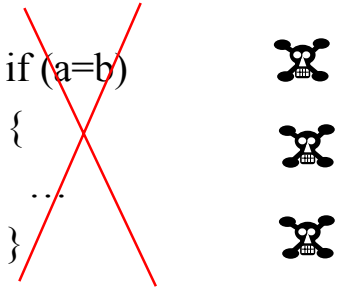


# Imbrication de si

- On peut imbriquer autant de si que l'on veut.
- Attention en C++, le else se rapporte toujours au dernier si rencontré. Dans certains cas, il faut donc rajouter des accolades. Exemple :

```
if (a!=b)
{
     if (a>b)
          b=b-a;
}
else
     a=0;
```

## Attention






## Éviter les maladresses

```
si a=b alors
  …
sinon
  si a≠b et b-a>3 alors
    …
  fin si
fin si
```

*Test inutile*



## Éviter les maladresses

```
si i>j alors
  b ← vrai          →   b ← i>j
sinon
  b ← faux
fin si
```



## Éviter les maladresses

```
si i>j alors
  b ← faux          →   b ← non i>j
sinon
  b ← vrai
fin si
```



## Éviter les maladresses

```
si i>j alors                si i>j alors
  seq1              →         seq1
fin si                      sinon
                              seq2
si i≤j alors                fin si
  seq2
fin si
```



## La structure selon

- Lorsque le nombre de cas à tester devient important, il devient pénible et illisible d'assembler les aiguillages du type si…sinon…fin si.
- La structure de contrôle selon est une commodité qui remplace une suite de si…sinon…fin si imbriqués
- Elle permet d'améliorer grandement la lisibilité du programme.
- On l'appelle aussi parfois structure « cas » (d'après son nom en Pascal)



## Fonctionnement du selon

- La structure selon fonctionne en trois étapes
  1. L'expression qui suit le selon est évaluée (on calcule sa valeur)
  2. La valeur de l'expression permet de sélectionner l'alternative qui doit être appliquée
  3. Les instructions de cette alternative sont exécutées



## Exemple de selon

```
selon NumeroDuJourDeLaSemaine choisir
  quand 1 faire
      afficher("lundi")
      fin
  quand 2 faire
      afficher("mardi")
      fin
  quand 3 faire
      afficher("mercredi")
      fin
  …
fin selon
```

*L'expression*

*Valeur possible de l'expression*

*une alternative*



## Comparaison avec le si

```
si NumeroDuJourDeLaSemaine=1 alors
   afficher("lundi")
sinon
   si NumeroDuJourDeLaSemaine=2 alors
        afficher("mardi")
   sinon
        si NumeroDuJourDeLaSemaine=3 alors
             afficher("mercredi")
        sinon
             …
        fin si
   fin si
fin si
```

- Le selon est plus clair et évite des erreurs

# Syntaxe du selon

selon →
    selon expression choisir
        liste_alternatives
        défaut
    fin selon
liste_alternatives →alternative
        | alternative liste_alternatives
alternative →
    quand condition_selon faire
        séquence_instructions
        fin
défaut →∅
    | par défaut faire séquence_instructions fin
instruction →selon



# Restrictions

- La plupart des langages imposent de fortes restrictions pour des raisons d'efficacité :
  - L'expression qui gouverne le selon doit être de type scalaire (c'est à dire d'un type assimilable à un entier). **Les réels sont donc interdits**.
  - Les conditions du selon sont souvent très restreintes : en général, on se limite à un test d'égalité entre la valeur indiquée et la valeur de l'expression, au mieux à un test d'appartenance à un ensemble de valeurs



# En pseudo-langage

- On se restreindra pour les conditions du selon aux possibilités offertes par Ada :
  - Un test d'égalité entre l'expression et une valeur
  - Un test d'appartenance de l'expression à un ensemble de valeurs
  - Un test d'appartenance de l'expression à un intervalle de valeurs
- Ces deux derniers tests peuvent toujours se réécrire sous la forme d'une suite de tests d'égalité



# Exemple

```
selon NumeroDuJourDeLaSemaine choisir
   quand 1 | 5 faire          ← Appartenance à un ensemble de valeurs
        afficher("début ou fin de semaine")
        fin
   quand 3 faire              ← Test d'égalité
        afficher("milieu de semaine")
        fin
   quand 6 à 7 faire          ← Appartenance à un intervalle de valeurs
        afficher("week-end")
        fin
   …
fin selon
```



# Syntaxe des conditions du selon

condition_selon → constante
        | liste_valeurs
        | intervalle_valeurs
liste_valeurs → constante
        | constante | liste_valeurs
intervalle_valeurs →constante à constante



# Alternative par défaut

- Il est possible de définir une alternative par défaut qui n'est exécutée que si la valeur de l'expression du selon ne correspond à aucune autre alternative.
- Cette alternative par défaut doit intervenir en dernier



# Exemple

```
selon NumeroDuJourDeLaSemaine choisir
   quand 1 | 5 faire
        afficher("début ou fin de semaine")
        fin
   quand 6 à 7 faire
        afficher("week-end")
        fin
   par défaut faire           ← Alternative pour tous les cas non prévus
        afficher("jour quelconque")
        fin
fin selon
```



# Règles d'utilisation

- Les conditions des alternatives doivent être **mutuellement exclusives**, ce qui signifie que pour une valeur de l'expression, on ne doit pouvoir exécuter que l'une des alternatives (pas deux)
- Il convient d'être **exhaustif**, c'est à dire de prévoir une alternative pour chaque valeur possible de l'expression (en utilisant l'alternative par défaut si nécessaire). C'est d'ailleurs obligatoire dans certains langages (Ada).



# En C++

- Le C++ (comme C et Java) est très limité en ce qui concerne la condition que l'on peut placer dans une alternative : il n'accepte que le test d'égalité
- On peut par contre, par répétition, obtenir l'équivalent d'un test d'appartenance à un ensemble de valeurs

## En C++

```
switch (NumeroDuJourDeLaSemaine)
{
    case 2:
          cout << "jour de l'algo";
          break;
    case 1:
    case 5:
          cout << "début ou fin de semaine";
          break;
    case 6:
    case 7:
          cout << "week-end" ;
          break;
    default:
          cout << "jour quelconque";
          break;
}
```

*Ne pas oublier les break !!*



## En C++

- Le selon se traduit par switch
- La condition doit être entre parenthèses
- Les alternatives sont entre accolades
- Les valeurs possibles sont indiquées par le mot case (n'oubliez pas le deux-points)
- Chaque alternative doit se terminer par un break
- L'alternative par défaut se note default:



## Les boucles (itérations)

- Il arrive très souvent dans un programme que l'on doive répéter l'exécution de certaines instructions
  - Soit un nombre de fois défini à l'avance
  - Soit jusqu'à ce qu'une condition change de valeur
- La force de l'ordinateur est qu'il ne se lasse pas de répéter les opérations et qu'il le fait vite
- Une exécution des instructions répétées est appelée itération (une itération=un seul passage)



## La boucle *pour*

- Elle s'utilise quand on connaît à l'avance le nombre de fois où l'on veut répéter la ou les instructions
- Il faut systématiquement l'utiliser chaque fois que l'on connaît le nombre de répétitions avant de démarrer la boucle (et on l'utilise uniquement dans ces cas là)



## Le compteur de la boucle *pour*

- Un compteur permet de distinguer les itérations les unes des autres
- En général, ce compteur est la propriété exclusive de la boucle *pour* et ne doit pas être modifié
- On peut toujours utiliser la valeur du compteur dans des expressions (mais on ne doit pas modifier sa valeur)
- En général, ce compteur disparaît à la fin de la boucle. On ne peut donc pas utiliser la valeur du compteur à la sortie de la boucle !
- Le nom du compteur doit être un identificateur valide



## Exemple en pseudo-langage

```
pour i de 1 à 5 faire
        afficher(i)
fin pour
```

- Cet exemple va répéter l'instruction d'affichage 5 fois, c'est à dire pour toutes les valeurs de i comprises entre 1 et 5, dans l'ordre. Il va donc afficher les nombres entiers de 1 à 5.



## Syntaxe du pour en PL

boucle_pour →
  pour nom_compteur de valeur_initiale
      à valeur_finale sens faire
    séquence_instructions
  fin pour
sens →∅ | croissant | décroissant
instruction →boucle_pour



## Fonctionnement du *pour*

- La séquence d'instructions est répétée pour chaque valeur possible du compteur comprise entre la valeur initiale et la valeur finale (bornes comprises)
- Les bornes de la boucle sont normalement calculées au tout début de la boucle
- Les bornes doivent être de même type et être de type scalaire (i.e. facilement assimilables à des entiers). Cela exclut en particulier les réels ! Le compteur sera bien sur de même type que les bornes.



## Sens de la boucle

- Les valeurs que prend successivement le compteur sont énumérées dans l'ordre spécifié par le sens de la boucle : dans le sens décroissant si l'on indique le mot clef décroissant et dans le sens croissant si l'on indique croissant ou si l'on ne spécifie rien (utilisation la plus courante).

## Boucle décroissante

pour i de 5 à 1 décroissant faire
    afficher(i)
fin pour

- Cet exemple va afficher les nombres entiers dans l'ordre décroissant, de 5 à 1.



## Ordre des valeurs

- Si une boucle croissante indique une valeur initiale strictement plus grande que la valeur finale, la séquence d'instructions n'est pas exécutée
- Exemple :

pour i de 6 à 5 faire
    afficher(i)
fin pour

n'affichera rien



## Ordre des valeurs (2)

- Si une boucle décroissante indique une valeur initiale strictement plus petite que la valeur finale, la séquence d'instructions n'est pas exécutée
- Exemple :

pour i de 5 à 6 décroissant faire
    afficher(i)
fin pour

n'affichera rien



## La boucle *pour* en C++

pour i de 1 à 5 faire
    afficher(i)
fin pour

se traduit par

```
for(int i=1;i<=5;i++)
    cout << i;
```



## Syntaxe d'un pour croissant en C++

boucle_for →
  for( type_compteur nom_compteur = val_init ;
    nom_compteur <= valeur_finale ;
    nom_compteur++)
      une_seule_instruction
instruction →boucle_for



## Syntaxe d'un pour décroissant en C++

boucle_for →
  for( type_compteur nom_compteur = val_init ;
    nom_compteur >= valeur_finale ;
    nom_compteur--)
      une_seule_instruction
instruction →boucle_for



## Remarques sur le for de C++

- En fait, en C++, la boucle *for* se compose de trois éléments séparés par des points-virgules :
  - l'initialisation : on précise le type du compteur, puis le nom du compteur, puis l'opérateur d'affectation et enfin la valeur initiale
  - le test de continuation : dès que le test devient faux, la boucle s'arrête. Pour une boucle croissante, le test doit être de la forme compteur<=valeur finale. Pour une boucle décroissante, il faut utiliser compteur>=valeur_finale. On peut utiliser les autres opérateurs de comparaison (<,>,!=,= =). Si le test est faux au tout début de la boucle, la séquence qui dépend du for n'est pas exécutée.
  - Le passage à la valeur suivante : compteur++ pour une boucle croissante et compteur-- pour une boucle décroissante



## Remarques sur le for de C++

- Comme pour le *if*, le *for* ne porte que sur une seule instruction. Il faut donc utiliser des accolades si l'on veut répéter plusieurs instructions
- Tout comme pour le *if*, il ne faut surtout pas mettre de point-virgule entre la ligne *for* et l'instruction (sinon, l'instruction répétée est l'instruction vide délimitée par le point virgule !)
- L'instruction *for* de C++ offre de multiples possibilités qu'on ne citera pas ici (voir l'équivalence avec le while)



## Exemples

- 
```
for(int i=1;i<=10;i++)
    cout << "*" << i;
```
affichera *1*2*3*4*5*6*7*8*9*10
- 
```
for(int i=10;i>=1;i--)
    cout << "*" << i;
```
affichera *10*9*8*7*6*5*4*3*2*1
- 
```
for(char c='A';c<='Z';c++)
    cout << c;
```
affichera ABCDEFGHIJKLMNOPQRSTUVWXYZ

## Exemples

- for(char c='Z';c<='A';c++)
  cout << c;

n'affichera rien

- for(char c='A';c<='A';c++)
  cout << c;

affichera A

- for(char c='A';c<'D';c++)
  cout << c;

affichera ABC



## Pas d'une boucle

- Certains langages proposent des boucles où le compteur varie de plus de une unité à chaque itération. Le nombre qui est ajouté au compteur à chaque itération est le pas de la boucle.
- En PL, on écrira
  pour i de 2 à 10 par pas de 2 faire

pour énumérer les nombres pairs <=10



## Pas d'une boucle (2)

- Si un langage ne permet pas de gérer le pas, il faut utiliser une boucle avec un compteur variant de 1 en 1 et l'utiliser pour calculer un nouveau compteur
- Exemple :
  pour i de 3 à 11 par pas de 2 faire
  afficher(i)
  fin pour

devient

pour nouveau de 1 à 5 faire
i←2*nouveau+1
afficher(i)
fin pour

*À simplifier*



## Pas d'une boucle en C++

- Boucles croissantes :
  pour i de *inf* à *sup* par pas de *pas* faire

se traduit par
for(int i=*inf* ; i<=*sup* ; i+=*pas*)

- Boucles décroissantes :
  pour i de *sup* à *inf* décroissant
  par pas de *pas* faire

se traduit par
for(int i=*sup* ; i>=*inf* ; i-=*pas*)



## La boucle pour chaque

- Cette boucle existe rarement dans les langages procéduraux classiques, elle est par contre fréquente dans les langages de script.
- Elle permet de faire prendre successivement au compteur les valeurs énumérées dans une liste.
- Ce type de boucle permet d'effectuer des énumérations sur des types non scalaires (chaîne de caractères par exemple)
- Elle ne peut être utilisée que si l'on connaît juste avant le début de la boucle le nombre de répétitions à effectuer.



## Syntaxe du pour chaque en PL

boucle_pour_chaque →
pour chaque nom_compteur dans
liste_valeurs faire
séquence_instructions
fin pour chaque

liste_valeurs→∅ | valeur liste_valeurs

instruction →boucle_pour_chaque



## Fonctionnement du pour chaque

1. La liste des valeurs est évaluée une seule fois, juste avant que la boucle ne commence
2. La séquence d'instructions est alors exécutée pour chaque valeur de la liste, dans l'ordre de leur énumération. Si la liste est vide, la séquence n'est pas du tout exécutée.



## Exemple en PL

afficher("les moyens de transport à votre disposition sont : ")

pour chaque m dans "avion" "bateau" "train" faire
afficher(m)

fin pour chaque



## Solution de remplacement

- À défaut de boucle pour chaque, on peut utiliser une boucle pour classique pour peu que l'on dispose de la notion de liste
- Exemple :

```
moyens←liste("avion","bateau","train")
pour i de premier_indice(moyens)
        à dernier_indice(moyens) faire
    afficher(élément(moyens,i))
fin pour
```

## En C++

- C++ ne propose pas directement la boucle *pour chaque*.
- Une forme de boucle *pour chaque* existe néanmoins pour les éléments d'un conteneur (notion objet).



## Les autres boucles

- Quand on ne connaît pas le nombre d'itérations à effectuer, il n'est pas possible d'utiliser une boucle *pour* (ni d'ailleurs une boucle *pour chaque*). Il faut alors utiliser une forme plus générale de boucle
  - Boucle *tant que*
  - Boucle *répéter…tant que*
  - Boucle *répéter…jusque*



## La boucle *tant que*

- Cette boucle répète la séquence d'instructions tant qu'une condition est vérifiée
- La condition est vérifiée avant l'exécution des instructions
- Si la condition est fausse avant l'exécution de la boucle, les instructions ne seront pas exécutées.



## Exemple en PL

```
afficher("les puissances de 2 inférieures à
   1000 sont : ")
n←2
tant que n<=1000 faire
   afficher(n)
   n ←n*2
fin
```



## Syntaxe du tant que en PL

```
boucle_tant_que →
   tant que condition faire
        séquence_instructions
   fin tant que
instruction →boucle_tant_que
```



## Votre responsabilité

- Le programmeur doit garantir que :
  - La condition du *tant que* a une valeur déterminée (en général vrai) avant le début de la boucle (vérifier l'initialisation)
  - La condition doit à un moment donnée devenir fausse pour que la boucle s'arrête et que l'on passe à la suite du programme. La séquence d'instructions doit donc à un moment donné modifier l'état du programme pour que cette condition devienne fausse.
    Si la condition reste toujours vraie, on a une boucle infinie (plantage ?)



## Syntaxe du *tant que* en C++

- Le *tant que* se traduit par *while* en C++. Comme d'habitude, la condition est indiquée entre parenthèses et la boucle ne porte que sur une seule instruction

```
boucle_while →
   while ( condition )
        une_seule_instruction
instruction →boucle_while
```



## Exemple en C++

```
cout <<"les puissances de 2 inférieures à 1000 sont : ";
n=2;
while (n<=1000)
{
   cout << n;
   n =n*2;
}
```



## Remarque

- Les structures *tant que* et *si…alors…fin si* sont les seules structures de contrôle réellement indispensables pour pouvoir écrire n'importe quel programme.
- Toutes les autres structures sont des facilités qui permettent de simplifier l'écriture des programmes.

## Exemple de substitution en C++

- La boucle for de C++ est en fait un simple while déguisé :

for(*initialisation*;*condition*;*suivant*)
    *instruction*;
correspond en fait (presque) directement à
    *initialisation*;
    while (*condition*)
    {
        *instruction*;
        *suivant*;
    }



## Exemple de substitution en C++

- for(int i=1;i<=5;i++)
    cout << i;

est donc (presque) équivalent à

```
i=1;
while (i<=5)
{
    cout << i;
    i++;
}
```



## La boucle *répéter tant que*

- La boucle *répéter…tant que* est une variante de la boucle *tant que* où l'on n'effectue le test qu'à la fin de l'itération.
- On trouve ce type de boucle dans les langages de la famille C (C,C++,Java)
- La condition est une condition de continuation : elle doit être vraie pour que l'on poursuive l'exécution de la boucle
- **Les instructions sont exécutées au moins une fois**



## Exemple de *répéter tant que* en PL

```
répéter
   afficher("Voulez-vous continuer (O/N) ?")
   lire(car)
tant que car≠'O' et car ≠'N' fin
/* on répète la lecture tant que l'utilisateur n'a
   pas tapé la lettre 'O' ou la lettre 'N' */
```



## Syntaxe du *répéter tant que* en PL

```
boucle_répéter_tant_que →
   répéter
      séquence_instructions
   tant que condition fin
instruction →boucle_répéter_tant_que
```



## Syntaxe du *répéter tant que* en C++

- Le *répéter…tant que* se traduit par *do…while* en C++. Comme d'habitude, la condition est indiquée entre parenthèses et la boucle ne porte que sur une seule instruction

```
boucle_do_while →
   do
      une_seule_instruction
   while ( condition )
instruction →boucle_do_while
```



## Exemple de *répéter tant que* en C++

```
do
{
   cout << "Voulez-vous continuer (O/N) ?";
   cin >> car;
}
while (car!='O' && car!='N');
/* on répète la lecture tant que l'utilisateur n'a pas
   tapé la lettre 'O' ou la lettre 'N' */
```



## La boucle *répéter jusque*

- La boucle *répéter…jusque* est une variante de la boucle *répéter… tant que.* Le test permettant de savoir s'il faut continuer à répéter les instructions est encore effectué à la fin de l'itération, mais cette fois-ci, on spécifie une condition d'arrêt. Le test doit donc être faux pour continuer à boucler (contraire du *répéter tant que*).
- On trouve ce type de boucle dans les langages de la famille Pascal (Pascal, Ada)
- **Les instructions sont exécutées au moins une fois**



## Exemple de *répéter jusque* en PL

```
répéter
   afficher("Voulez-vous continuer (O/N) ?")
   lire(car)
jusque car='O' ou car='N' fin
/* on répète la lecture jusqu'à ce que
   l'utilisateur tape la lettre 'O' ou la lettre 'N'
   */
```

# Syntaxe du *répéter jusque* en PL

boucle_répéter_jusque →
  répéter
    séquence_instructions
  jusque condition fin
instruction →boucle_répéter_jusque



# En C++

- Le *répéter…jusque* n'existe pas en C++. Ce n'est absolument pas pénalisant car un *répéter…jusque* se transforme en *répéter…tant que* (ou inversement) en prenant la négation de la condition.

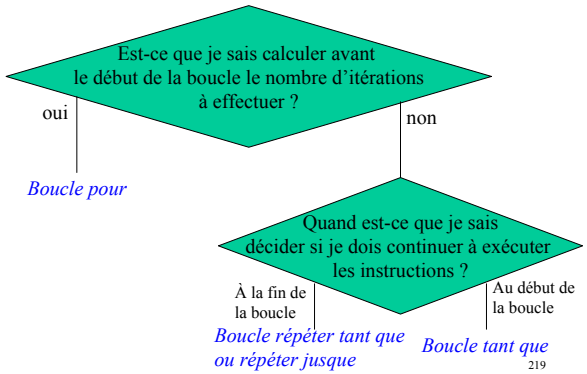
répéter
  *instructions*
tant que *condition* fin

⟷

répéter
  *instructions*
jusque **non** *condition* fin

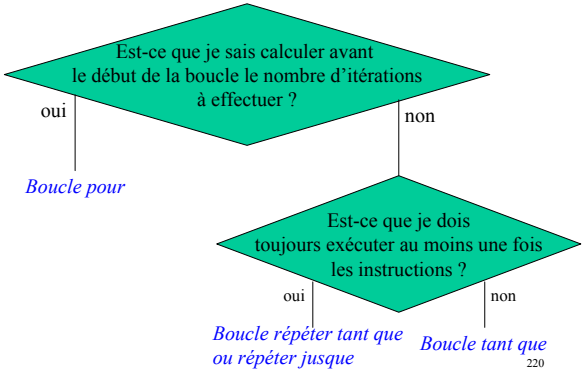


# Comment choisir la bonne boucle ?






# Questionnaire équivalent






# Indentation

- Pour faciliter la lecture d'un programme, il faut que la présentation mette en relief les structures de contrôle.
- Pour cela, on décale les instructions d'un certain nombre d'espaces (2 par exemple) pour mettre en évidence qu'une instruction dépend d'une autre : c'est l'indentation



# Règles d'indentation

1. Mettre une instruction par ligne
2. Les instructions d'une même séquence doivent être alignées à gauche
3. Les divers mots-clefs d'une même structure doivent être alignés (ex: si… sinon…fin si)
4. Les instructions dont l'exécution dépend d'une structure de contrôle doivent être décalées de deux espaces vers la droite par rapport à cette structure

**Voir les exemples d'indentation dans les transparents précédents**



# Commentaires

- Des commentaires doivent être rajoutés dans le programme pour expliquer en français ce que font les différentes parties du programme
- Un commentaire ne doit pas paraphraser le code
- En C++, il y a deux types de commentaires
  - Commentaire d'une seule ligne
    débute par // et se termine à la fin de la ligne
  - Commentaires sur plusieurs lignes
    commence par /* et se termine par */



# Les procédures et fonctions



# Pour quoi faire ?

- Dans un programme, il arrive souvent que l'on effectue le même traitement en plusieurs endroits. Cela amène des répétitions qui sont
  - Lourdes (on doit recopier les même instructions à plusieurs endroits)
  - Dangereuses (bug du copier-coller)
  - La cause de difficultés à lire le programme
- Pour éviter ces problèmes, on regroupe dans une procédure ou une fonction les traitements (instructions) que l'on effectue en plusieurs points du programme (factorisation).

# Procédure

- Une procédure est le regroupement d'instructions auquel on donne un nom (qui doit être un identificateur)
- Le traitement effectué par la procédure peut être modifié par un certain nombre de paramètres.
- Une fois le nom de la procédure associé au traitement, il suffit de citer le nom de la procédure (en indiquant aussi les éventuels paramètres) pour exécuter les instructions qui composent cette procédure.
- L'appel de la procédure est une instruction comme une autre.



# Fonction

- Une fonction est très similaire à une procédure. Il s'agit là aussi de regrouper des instructions en donnant un nom à ce regroupement. La différence est que les instructions qui composent la fonction doivent servir à calculer un résultat.
- Une fonction correspond en fait à la notion mathématique de fonction. À partir de certaines informations (les paramètres), on calcule un résultat.
- Comme en mathématiques, l'appel de la fonction « f(x) » représente en fait le résultat calculé par cette fonction. Quand l'ordinateur rencontre un tel appel, il exécute les instructions pour calculer le résultat et le substituer à l'appel de la fonction. Une fonction sera donc forcément utilisée dans une expression.



# Les procédures et fonctions

- Il y a deux étapes dans l'utilisation des proc./fonc.
  1. La définition
  2. L'appel



# La définition

- Elle consiste
  - à donner le nom de la proc./fonc.,
  - à indiquer les paramètres sur lesquels elle va travailler
  - et à indiquer l'ensemble des instructions qu'elle va effectuer
- Lors de la définition, l'ordinateur enregistre ce que la proc./fonct. doit faire mais il n'exécute aucune instruction.



# L'appel

- Il consiste à demander l'exécution des instructions de la proc./fonc. sur un ensemble de données que l'on fournit en paramètres



# Ordre

- Les proc./fonc. se définissent au début du programme, après les constantes et variables globales.
- Une proc./fonc. doit être définie avant d'être appelée (ou tout du moins déclarée dans le cas de récursivité croisée) : on ne doit pas invoquer un nom sans avoir défini ce nom au préalable



# Paramètre formel

- Quand on définit une proc./fonct., on ne peut pas savoir à l'avance avec quels paramètres elle va être appelée.
- Dans la définition de la proc./fonct., on utilise donc un nom qui désigne le paramètre (l'information) que l'on va recevoir. C'est le **paramètre formel**
- Ce nom n'est utilisé que dans la définition de la proc./fonct.



# Paramètre formel (2)

- Le paramètre formel joue un peu le rôle d'une variable muette en mathématiques. Tout comme en mathématiques, le nom choisi n'a aucune importance à l'extérieur de la proc./fonc. Il faut néanmoins choisir un nom évocateur.

*Le choix de ce nom n'a aucune importance*

$$\sum_{i=1}^{3} i$$



# Paramètre formel (3)

- Pour chaque paramètre formel, il faut indiquer
  - Son nom
  - Son type
  - Son mode de passage (voir plus loin) qui, en pseudo, sera spécifié par
    - soit Donnée (D)
    - soit Résultat (R)
    - soit Donnée/Résultat (D/R)

# Paramètre formel (4)

- En pseudo-langage, les paramètres formel sont séparés par des point-virgules
  - procedure P(*paramètre_formel _1* **:** *mode_passage type* **;** *paramètre_formel_2* **:** *mode_passage type*)
    début … fin
- Quand des paramètres ont même type et même mode de passage, on peut factoriser en séparant les noms de paramètres par des virgules

  procedure P(*paramètre_formel _1* **,** *paramètre_formel_2* **:** *mode_passage* **:** *type*)
  début … fin



# Paramètre formel (5)

- En C++, les paramètres formels sont séparés par des virgules et l'on doit indiquer le type devant chaque paramètre (on ne peut pas regrouper des paramètres formels de même type et mode de passage). Le mode de passage se traduira par la présence ou l'absence d'un & devant le nom du paramètre
  - void P(*type mode_passage paramètre_formel_1***,** *type mode_passage paramètre_formel_2*)
    { … }



# Paramètre effectif

- Quand on appelle une proc./fonct., on lui transmet les véritables informations sur lesquelles elle doit travailler : ce sont les **paramètres effectifs**
- D'un appel à l'autre de la proc./fonct. les paramètres effectifs utilisés seront vraisemblablement différents



# Paramètre effectif (2)

- Lors de l'appel de la proc./fonct., on donne les paramètres effectifs en les séparant simplement par des virgules
  - Deplacer(1,2)
  - Deplacer(1,n)



# Les procédures

- En pseudo-langage, elle se définit comme-ceci

  procedure *nom_procedure*
  (*liste_de_paramètres_formels*)
  *déclarations_locales*
  début
  *séquence_instructions*
  fin



# Les procédures (2)

*Pas de point-virgule*

- En C++, elle se définit comme-ceci

  void *nom_procedure*
  (*liste_de_paramètres_formels*)
  {
  *déclarations_locales*
  *séquence_instructions*
  }



# Les procédures (3)

- Il peut arriver qu'une procédure n'ait pas besoin de paramètre. Dans ce cas, on ne note rien entre les parenthèses (mais on laisse ces parenthèses)
  - void ToutInitialiser()
    { … }



# Appel de procédure

- Une fois la procédure définie, on peut l'appeler (i.e. demander son exécution) à partir du programme principal ou d'une autre procédure. Pour cela, on donne simplement le nom de la procédure, suivi des paramètres effectifs entre parenthèses. Une procédure est une instruction, il faut donc mettre un point-virgule en C++. Ex: Deplacer(1,2);



# Les déclarations locales

- Une procédure ou une fonction peut être considérée comme un programme en miniature qui est vu de l'extérieur comme une boîte noire. Il est donc normal que l'on puisse définir des constantes, variables, procédures et fonctions qui soient propres à cette boîte noire. Ce sont les déclarations locales. Cela permet de rendre la proc./fonct. indépendante du reste du programme et facilite sa réutilisation.

# Les déclarations locales (2)

- Attention, en C/C++, les déclarations locales ne peuvent comporter que des constantes et variables (pas de procédure ni de fonction)



# Les fonctions

- En pseudo-langage, elle se définit comme-ceci
  fonction *nom_fonction*
  (*liste_de_paramètres_formels*)
  retourne *type_résultat*
  *déclarations_locales*
  début
  *séquence_instructions*
  retourner *résultat_calculé*
  fin



# Les fonctions (2)

*Pas de point-virgule*

- En C++, elle se définit comme-ceci
  *type_résultat nom_fonction*
  (*liste_de_paramètres_formels*)
  {
  *déclarations_locales*
  *séquence_instructions*
  return *résultat_calculé* ;
  }



# Les fonctions (3)

- Il est indispensable de préciser le type d'information renvoyée par la fonction. Certains langages imposent des restrictions sur le type que peut renvoyer une fonction.
- Il est également indispensable de renvoyer le résultat calculé par la fonction à la fin de celle-ci.
- Il faut toujours s'assurer que la fonction renvoie un résultat dans tous les cas de figure.



# Les fonctions (4)

- Certains langages (dont C++) permettent de retourner le résultat de la fonction dès qu'il est obtenu (éventuellement avant la fin de la fonction). Dans ce cas, l'instruction retourner met immédiatement fin à l'exécution de la fonction.



# Appel de fonction

- Le résultat d'une fonction doit normalement apparaître dans une expression (mathématique, logique, relationnelle,…) et le résultat de cette fonction doit être utilisé (sinon, à quoi bon l'avoir calculé ?)
  Ex:
  n=factorielle(5);
  n=factorielle(5)+pow(2,5);
  TransfererTas(1,NumeroAUtiliser(x));



# Le cas de main()

- En C++, int main() est bien sûr une fonction qui peut avoir des paramètres (les arguments fournis au programme sur la ligne de commande) mais qui surtout renvoie un entier qui représente un code d'erreur (0 si tout s'est bien passé, une valeur de 1 à 255 sinon)
- Un programme correct doit donc contenir un return 0; à la fin de *main*



# Le passage de paramètres



# Différents modes de passage

- Il existe différentes manières de passer des paramètres effectifs à une proc./fonc.pour qu'elle puisse les manipuler au travers des paramètres formels
  - Par copie (Ada)
  - Par valeur (C,C++,…)
  - Par référence (C++,…)

# Le passage par copie

- Dans ce mode qui est utilisé en Ada, on considère les paramètres formels comme des récipients (variables) intermédiaires qui servent à recevoir les paramètres effectifs qui seront manipulés par la proc./fonc.
- Au début et à la fin de la proc./fonc., il y a ou pas une opération de copie entre ces récipients intermédiaires et les paramètres effectifs
- Cela est spécifié par le programmeur quand il indique le mode de passage : donnée, résultat ou donnée/résultat

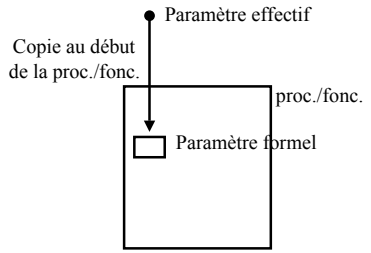


# Le passage par copie en donnée

- Le paramètre effectif est copié dans le paramètre formel au début de la proc./fonc. Les informations sont transmises dans un seul sens : du programme vers la proc./fonc.
- Aucune restriction sur le paramètre effectif qui peut être une constante, une variable, une expression mathématique, l'appel à une fonction,…
- Les éventuelles modifications du paramètre formel dans la proc./fonc. ne sont pas retransmises au paramètre effectif

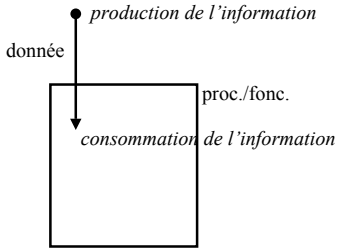


# Le passage par copie en donnée






# Le passage par copie en donnée



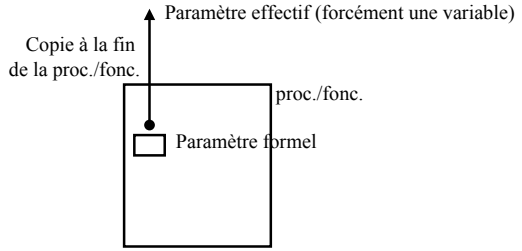



# Le passage par copie en résultat

- Les informations sont transmises dans un seul sens : de la proc./fonc. vers le programme.
- À la fin de la proc./fonc., le paramètre formel est recopié dans le paramètre effectif par une opération d'affectation.
- De ce fait, le paramètre effectif doit nécessairement être une variable

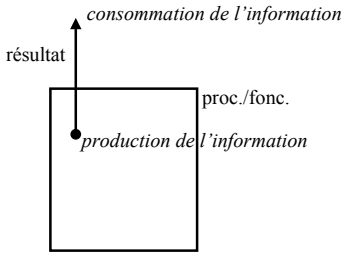


# Le passage par copie en résultat






# Le passage par copie en résultat



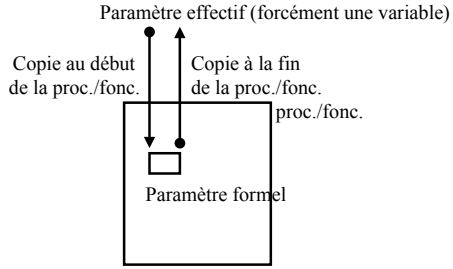



# Le passage par copie en donnée/résultat

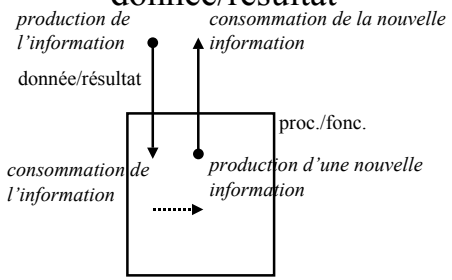
- C'est la combinaison des deux modes précédents. Les informations transitent dans les deux sens : du prg vers la proc./fonc. au début de celle-ci et de la proc./fonc. vers le prg à la fin de l'appel.
- Le paramètre effectif est copié dans le paramètre formel au début de la proc./fonc.
- Le paramètre formel est copié dans le paramètre effectif à la fin de la proc./fonc.
- Le paramètre effectif doit nécessairement être une variable.



# Le passage par copie en donnée/résultat



# Le passage par copie en donnée/résultat



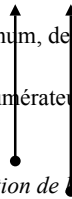



# Le choix du mode de passage

- Si la proc./fonc. utilise l'information que peut contenir le paramètre, le mode de passage doit contenir D
- Si la proc./fonc. modifie le paramètre et s'il faut que cette modification soit reportée sur le paramètre effectif, le mode de passage doit contenir R
- Si la réponse aux deux questions est positive, le mode de passage doit être D/R



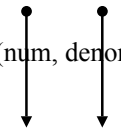
# Exemples : procédure de lecture



```
procedure LireFraction(num, denom : resultat entier)
debut
   afficher(" Entrez le numérateur puis le
   dénominateur ")
   lire (num)
   lire(denom) // production de l'information
   //  (l'utilisateur la tape au clavier, on ne
   //  consomme aucune information)
fin
```



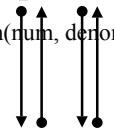
# Exemples : procédure d'affichage



```
procedure AfficherFraction(num, denom :
   donnée entier)
debut
   afficher (num, "/" ,denom)
   // consommation de l'information
fin
```



# Exemples : simplification



```
procedure SimplifierFraction(num, denom : donnée
   résultat entier)
variables
   diviseur:entier
debut
   diviseur←PGCD(num,denum)
   num←num/diviseur
   denom←denom/diviseur
   // modification de l'information (consommation
   puis production d'une nouvelle information)
fin
```



- Une modification des paramètres formels ne signifie pas forcément qu'il faille les passer en donnée/résultat !



```
fonction PGCD(a,b:donnée entier) retourne entier
début
   // ici, on se permet de modifier le paramètre formel en
   // sachant que ces modifications ne seront pas reportées
   // sur le paramètre effectif
   tant que a≠b faire
      si a>b alors
            a ←a-b
      sinon
            b ←b-a
      fin si
   fin tant que
   retourner a
fin
```



# Le mode de passage par valeur

- C'est le nom donné en Pascal, C, C++, Java,… au mode de passage par copie en donnée.
- En C++, on indique qu'un paramètre doit être passé par valeur en ne mettant rien entre le type et le nom du paramètre formel.
- Ex: void Affiche(int num, int denom)



# Le mode de passage par référence

- Ce mode correspond presque au mode de passage par copie en donnée/résultat. La différence est qu'il n'y a pas de récipient intermédiaire et que la proc./fonc manipule directement le paramètre effectif. Seules des variables peuvent être passées par référence.
- En C++, le passage par référence se note en mettant un & entre le type du paramètre formel et le nom du paramètre
- Ex: void Simplifie(int &num, int &denom)



# Correspondance

- Dans les langages qui ne disposent pas du mode de passage par copie, on utilisera la correspondance suivante

| | |
|---|---|
| Donnée | Par valeur |
| Résultat | Par référence |
| Donnée/résultat | Par référence |

# Transcription de fonctions mathématiques

- Une fonction mathématique définie par
  *nom :* $E1 \times E2 \times E3 \rightarrow E$
  $(x,y,z) \rightarrow r$
  se retranscrit en C++ par

```
typeDeE nom(typeDeE1 x, typeDeE2 y, typeDeE3 z)
{
    calcul de r
    return r ;
}
```



# Exemple

factorielle : $N \rightarrow N$
$n \rightarrow n!$

```
unsigned int factorielle(unsigned int n)
{
  int produit;
  produit=1;
  for(int i=2;i<=n;i++)
      produit=produit*i;
  return produit;
}
```



# Exemple

CarreParfait : $N \rightarrow B$
$n \rightarrow$ vrai ssi n est le carré d'un entier

```
bool CarreParfait(int n)
{
  int r; // racine entière de n
  r=(int)(sqrt(n)+0.5);
  return r*r= =n;
}
```



# Passage de paramètres

- Il ne faut pas confondre le passage de paramètre qui sert à obtenir une information du programme ou à lui en transmettre une avec les opérations de lecture et d'affichage (cin/cout) qui servent à échanger des informations avec l'utilisateur devant l'ordinateur.



# Fonction ou procédure

- On utilise une fonction quand on veut que le résultat produit soit utilisé en lieu et place de l'appel à la fonction (souvent à l'intérieur d'une expression mathématique).
- On utilise une procédure quand il n'y a pas de résultat à produire où quand on préfère transmettre le résultat par l'intermédiaire des paramètres en résultat ou donnée/résultat.



# Les variables locales



# Un programme en miniature

- Une proc./fonc. peut être considérée comme une boîte noire ou encore un programme en miniature. Cette boîte noire doit être la moins dépendante possible du reste du programme pour pouvoir être réutilisée d'un programme à l'autre.
- La proc./fonc. doit donc utiliser ses propres variables pour effectuer ses calculs.



# Variables locales

- Les variables qui sont déclarées au début d'une proc./fonc. sont appelées des variables locales.
- Elles n'existent qu'à partir du moment où la proc./fonc. commence son exécution et disparaissent dès qu'elle se termine.
- Les variables locales ne peuvent être utilisées qu'à l'intérieur de la proc./fonc.



# Variables globales

- Les variables qui sont déclarées en dehors de toute proc./fonc. sont des variables globales qui existent depuis le lancement du programme jusqu'à sa terminaison.
- Les variables globales peuvent être utilisées partout dans le programme.

# Comparaison culinaire

- Les variables globales sont comparables à des récipients qui sont partagés par les différents cuisiniers (les proc./fonc.) chargés de mener à bien une recette (le prg)
- Les variables locales sont comparables à des récipients jetables que le cuisinier met en place dès qu'il s'installe et qu'il jette quand il a fini de travailler



# Comparaison culinaire (2)

- Le cuisinier qui travaille avec des récipients jetables est sûr d'être le seul à les utiliser et peut donc en faire ce qu'il veut.
- Par contre, un cuisinier qui utilise les récipients partagés (var. globales) ne peut jamais être sûr d'être le seul à les utiliser. Dans ce cas, le chef (le programmeur) doit s'assurer que les cuisiniers utilisent les récipients partagés de manière coordonnée



# Utilité des variables locales.

- Il convient d'utiliser le plus souvent possible des variables locales de manière à éviter les interactions entre les diverses proc./fonc. (effets de bord).
- Si on ne le fait pas, on risque de se retrouver dans une situation où deux cuisiniers (proc./fonc.) utilisent un même récipient (var. globale) chacun leur tour, l'un pour faire un gâteau au chocolat, l'autre pour faire du pot-au-feu. Catastrophe garantie !



# Effet de bord

- On appelle effet de bord la modification par une proc./fonc. d'une variable globale qui n'est pas passée en paramètre. C'est donc un effet caché de la boîte noire.
- Il convient de s'interdire le plus souvent possible les effets de bord car ils rendent impossible la réutilisation des proc./fonc. et sont générateurs de bogues difficiles à identifier.



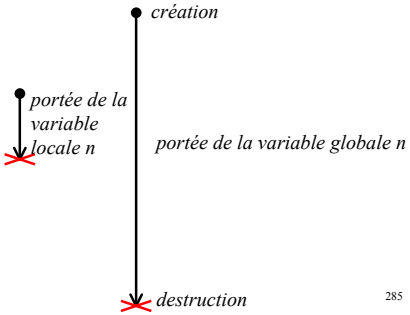
# Portée d'une variable

- La portée d'une variable représente la durée de vie de cette variable : elle s'étend de la création de la variable (sa déclaration) jusqu'à l'endroit où cette variable est détruite.
- Une variable globale n'est détruite qu'à la fin du programme
- Une variable locale est détruite à la fin de la proc./fonc.



# Exemple de portée

```
int n;
void P(int a)
{
   int n;
   n=a+2;
}
int main()
{
   n=2;
   P(n);
}
```






# Les boucles for

- Quand on écrit une boucle for sous la forme

```
for(int i=1;i<=10;i++)
{
      …
}
```

On déclare en fait une variable locale à la boucle qui n'est utilisable que dans la boucle et qui est détruite dès la fin de celle-ci.



# Portée dans une boucle for

```
for(int i=1;i<=10;i++)
{
      …
}
```



# Boucle for sans variable locale

- On peut écrire une boucle pour sans déclarer de variable locale à cette boucle pour servir de compteur. Dans ce cas, il faut bien avoir conscience que la variable n'est plus la propriété de la boucle et peut-être manipulée par d'autres parties du programme.

# Exemple de boucle for sans variable locale

```
int i;
…
for(i=1;i<=10;i++)
{
                …
}
// i peut être réutilisé. À la sortie de la boucle,
// i vaut 11
```



# Visibilité d'une variable

- Puisqu'une proc./fonc. est un programme en miniature, rien n'interdit à une variable locale de porter le même nom qu'une variable globale.
- Les définitions locales l'emportent sur les définitions globales.
- Une variable locale masque une variable globale de même nom et donc, à l'intérieur de la proc./fonc., c'est la variable locale qui sera manipulée quand on indiquera son nom plutôt que la variable globale.

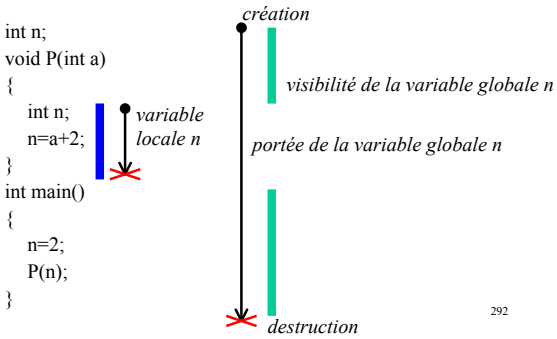


# Visibilité

- On dit qu'une variable est visible lorsque c'est elle qui sera manipulée par le programme lorsqu'on donne son nom (et pas une autre variable de même nom déclarée ailleurs dans le programme)
- Une variable ne peut être visible que quand elle existe, c'est à dire à l'intérieur de sa portée
- Une variable masquée n'est pas visible

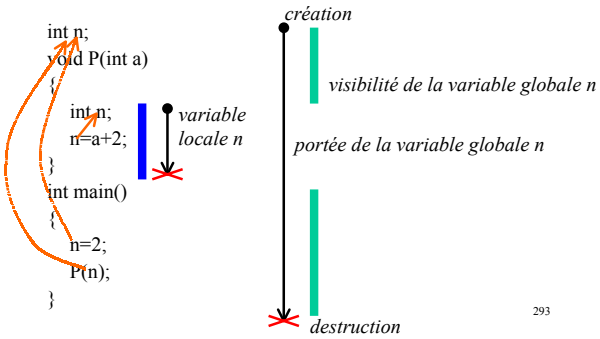


# Exemple de visibilité






# Exemple de visibilité






# Variable locale et paramètre

- Une variable locale ne peut pas porter le même nom qu'un paramètre (dans une même boîte noire, cela n'a pas de sens de donner le même nom à deux éléments différents)



# Pile d'exécution

- L'ordinateur doit s'y retrouver dans la gestion des variables locales. Pour cela, il utilise une pile d'exécution.
- Les variables globales sont complètement à part et sont stockées dans des endroits bien connus de la mémoire (segment de données).

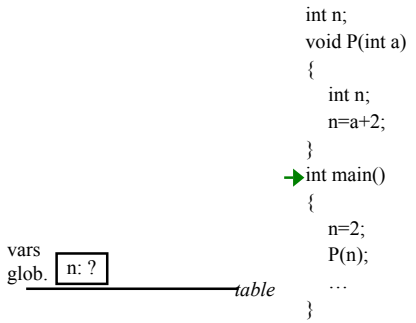


# Pile d'exécution

- Quand une proc./fonc. démarre, elle pose un plateau (frame) sur la pile d'exécution et dépose sur ce plateau ses récipients (var. locales).
- Un cuisinier travaille toujours avec les récipients qui **sont visibles** et qui lui sont le plus directement accessibles.
- Quand une proc/fonc se termine, les variables locales sur le plateau au dessus sont jetées avec le plateau.



# Exemple



# Exemple

```
int n;
void P(int a)
{
   int n;
   n=a+2;
}
int main()
{
   n=2;
   P(n);
   …
}
```

vars glob. n: 2 — *table*



# Exemple

```
int n;
void P(int a)
{
   int n;
   n=a+2;
}
int main()
{
   n=2;
   P(n);
   …
}
```

a: — *plateau*

vars glob. n: 2 — *table*



# Exemple

```
int n;
void P(int a)
{
   int n;
   n=a+2;
}
int main()
{
   n=2;
   P(n);
   …
}
```

a: 2 — *plateau*

vars glob. n: 2 — *table*



# Exemple

```
int n;
void P(int a)
{
   int n;
   n=a+2;
}
int main()
{
   n=2;
   P(n);
   …
}
```

n: ? a: 2 — *plateau*

vars glob. n: 2 cette var. est masquée — *table*



# Exemple

```
int n;
void P(int a)
{
   int n;
   n=a+2;
}
int main()
{
   n=2;
   P(n);
   …
}
```

n: 4 a: 2 — *plateau*

vars glob. n: 2 cette var. est masquée — *table*



# Exemple

```
int n;
void P(int a)
{
   int n;
   n=a+2;
}
int main()
{
   n=2;
   P(n);
   …
}
```

n: 4 a: 2 — *plateau*

vars glob. n: 2 cette var. est masquée — *table*



# Exemple

```
int n;
void P(int a)
{
   int n;
   n=a+2;
}
int main()
{
   n=2;
   P(n);
   …
}
```

vars glob. n: 2 — *table*



# Attention

- La visibilité d'une variable se décide lors de la compilation, c'est à dire lors de la traduction du programme en langage machine. Elle ne change plus par la suite.
- Cela simplifie beaucoup la compréhension des programmes !
- Il faut bien avoir conscience que même si un nom est utilisé à un moment donné sur plusieurs plateaux, ce sont les **règles de visibilité** qui permettent de savoir quel objet sera manipulé.



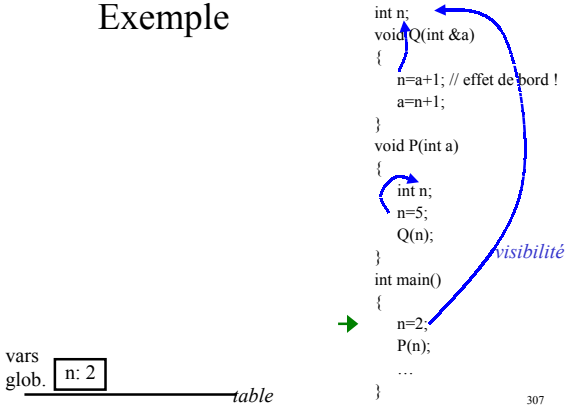
# Exemple

```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

*visibilité*

vars glob. n: ? — *table*

# Exemple

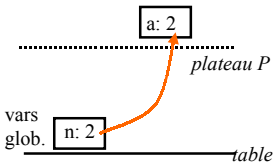



```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

visibilité



# Exemple

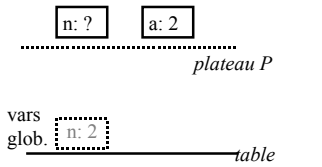



```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

visibilité



# Exemple

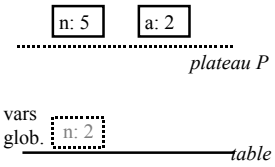



```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

visibilité



# Exemple

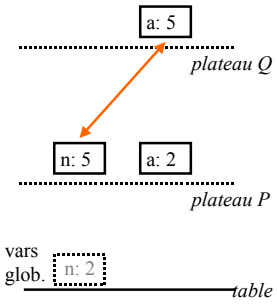



```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

visibilité



# Exemple

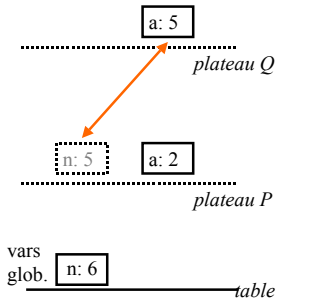



```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

visibilité



# Exemple

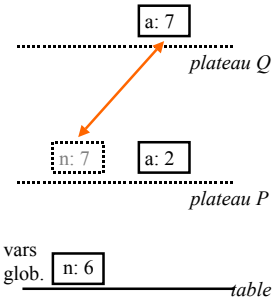



```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

visibilité



# Exemple

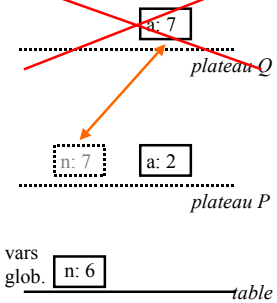



```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

visibilité



# Exemple

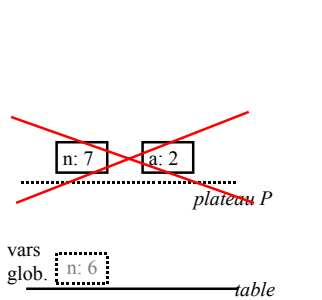



```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

visibilité



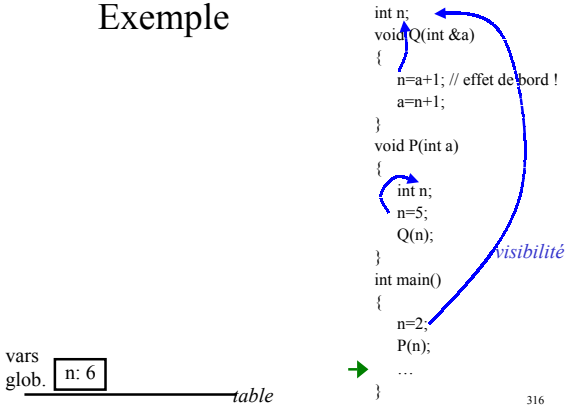
# Exemple




```
int n;
void Q(int &a)
{
   n=a+1; // effet de bord !
   a=n+1;
}
void P(int a)
{
   int n;
   n=5;
   Q(n);
}
int main()
{
   n=2;
   P(n);
   …
}
```

visibilité

# Exemple






# Le « plateau »

- Le plateau contient des informations supplémentaires. En particulier, il mémorise l'endroit où doit se poursuivre l'exécution du programme une fois que la proc./fonc. se termine. Cela n'est visible que quand on fait de l'assembleur.



# Taille de la pile

- La pile a une taille limitée qui peut éventuellement être paramétrée (typiquement de l'ordre de qq Ko à qq Mo, paramètrable par la commande limit sous Unix)
- Les grosses variables (tableaux,…) doivent donc parfois être déclarées globales pour que la pile ne déborde pas.



# Les types définis par le programmeur



# Pourquoi ?

- Les types de base (bool, char, int, float,…) se révèlent en général trop simples et trop limitatifs
- Le programmeur a la possibilité de construire de nouveaux types
- Les types se définissent normalement après les constantes et avant les variables.
- On distingue les types simples qui contiennent une seule information des types composés qui en contiennent plusieurs



# Restriction de types existants

- La manière la plus simple de définir de nouveaux types est de se baser sur un type existant et de restreindre l'ensemble des valeurs
- Exemples en pseudo
  types
    octet = 0..255
    majuscule = 'A'..'Z'



# D'un langage à l'autre

- Tous les langages ne permettent pas les même restriction sur les types
- C++ est particulièrement pauvre de ce point de vue. En effet, il n'y est pas possible de définir un intervalle quelconque de valeurs pour une variable (impossible de définir un type ne pouvant contenir que des majuscules par exemple)



# Déclaration de type en pseudo

programme premier_type
constantes
  …
types
  *nom_du_type = définition_du_type*
variables
  …



# Déclaration de type en C++

- En C++, quand on veut définir un nouveau type T, on fait comme si on déclarait une variable dont le nom serait T et on rajoute le mot clef typedef devant la déclaration
- Exemple en C++
  typedef
    unsigned char octet;
- On peut ensuite déclarer des variables de type octet
    octet n;

# Les types énumérés

- Ils permettent d'utiliser des valeurs symboliques dans les programmes plutôt que des valeur numériques
- Très utile pour clarifier le source d'un programme
- Restriction : la plupart des langages ne gèrent pas du tout leur lecture au clavier ou leur affichage à l'écran



# Type énuméré en pseudo

```
types
    jour=(lundi,mardi,mercredi,jeudi,vendredi,
          samedi,dimanche)
variables
    aujourdhui,demain,hier:jour
debut
    aujourdhui ← mardi
    demain ← successeur(aujourdhui)
    hier ← predecesseur(aujourdhui)
    // le comportement de succ. et pred. n'est pas
    // défini pour les extrêmes
fin
```



# Type énuméré en C++

```
enum jour {lundi,mardi,mercredi,jeudi,vendredi,
           samedi,dimanche};
int main()
{
    jour aujourdhui,demain,hier;
    aujourdhui=mardi;
    demain=aujourdhui+1;
    hier=aujourdhui-1;
}
```



# Les types composés



# Les structures

- Une structure, aussi appelée enregistrement ou record, est le regroupement de plusieurs variables qui n'ont pas le même type ou qui ne représentent pas la même information (ne jouent pas le même rôle).
- Exemple en pseudo

```
types
    personne = structure
                    nom, prenom : chaine
                    age : entier
               fin structure
```

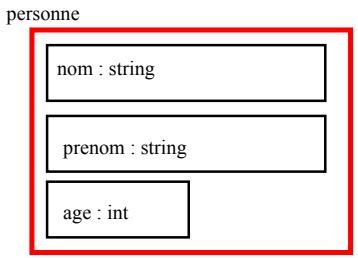


# Exemple en C++

```
typedef
    struct
    {
        string nom,prenom;
        int age;
    } personne;
```



# Illustration






# Exemple : le type complexe

- En pseudo :

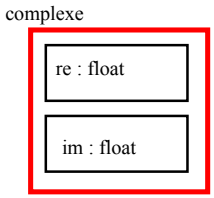
```
types
    complexe = structure
                    re,im:reel
               fin structure
```

- En C++ :

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



# Illustration



## Affectation et comparaison de structures

- On peut déclarer des variables de type structure et effectuer des affectations entre elles sans aucun problème.
- Par contre, de manière générale, les comparaisons entre structures (égalité, relation d'ordre) ne sont pas possibles (N.B: C++ permet de le faire en définissant des opérateurs spécifiques)



## Les champs

- Chaque composante d'une structure est appelé un champ. Le type complexe a donc deux champs : re et im
- Une variable de type structure forme un tout qui regroupe l'ensemble des champs.



## Accès aux champs

- Pour accéder aux champs de la structure, on utilise la notation pointée

  *nom_var_structure*.*nom_champ*

  Ex: complexe c;
  c.re=0;
  c.im=0;



## Exemple

```
complexe c,zero,i;
zero.re=0;
zero.im=0;
i.re=0;
i.im=1;
c=i;
```



## Imbrication de structures

- Une structure peut contenir une autre structure. Quand un champ est une structure, on peut recommencer à mettre un point et le nom d'un champ pour aboutir aux informations (voir exemple)

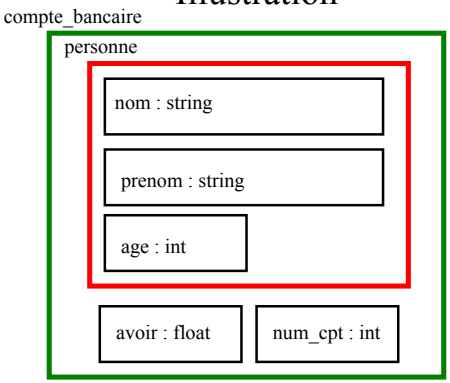


## Exemple

```
typedef
    struct
    {
        personne client;
        float avoir;
        int numero_compte;
    } compte_bancaire;
```



## Illustration



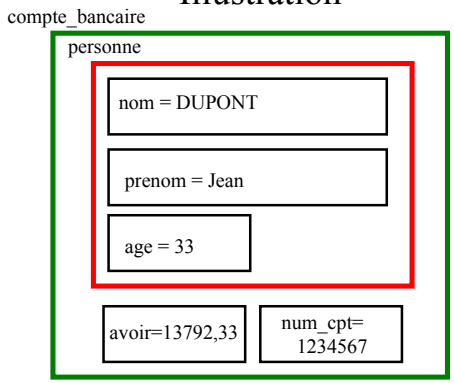



## Exemple (suite)

```
int main()
{
    compte c;
    c.client.nom="DUPONT";
    c.client.prenom="Jean";
    c.client.age=33;
    c.avoir=13792.33;
    c.numero_compte=1234567;
}
```



## Illustration



# Intérêt d'une structure

- L'intérêt d'une structure est que l'ensemble des champs forme un tout que l'on peut transmettre comme paramètre ou recevoir comme résultat d'une fonction (sauf dans certains langages)



# Exemple

```
complexe addition(complexe c1, complexe c2)
{
    complexe s;
    s.re=c1.re+c2.re;
    s.im=c1.im+c2.im;
    return s;
}
int main()
{
    complexe a,b,c;
    …
    a=addition(b,c);
}
```



# Note sur C++

- En C++, il suffirait d'appeler la fonction d'addition *operator* + pour pouvoir écrire a=b+c; avec a,b,c complexes.



# Les tableaux

- Un tableau est le regroupement de plusieurs variables qui sont **nécessairement** du même type et qui seront normalement utilisées de la même manière.
- Les variables du tableau sont repérées par un numéro appelé **indice** (tableau à une dimension) ou bien par plusieurs indices (tableau à plusieurs dimensions)



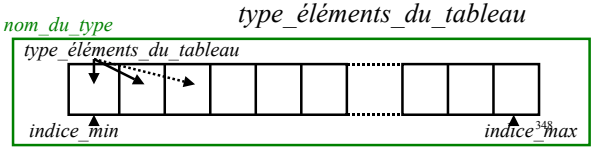
# Indice dans un tableau

- L'indice utilisé pour identifier les variables qui composent un tableau doit être du type entier ou, plus généralement, d'un type scalaire (i.e. directement assimilable à un entier)
- Les indices utilisés sont nécessairement consécutifs. Quand on déclare un tableau on donne donc uniquement l'indice minimum et l'indice maximum (ainsi que le type des éléments du tableau).



# Tableau à une dimension en pseudo

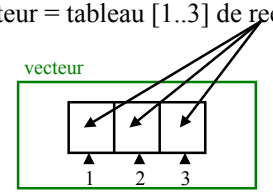
- En pseudo, on définira un type tableau comme ceci :
  types
  *nom_du_type* = tableau [*indice_min*.. *indice_max*] de *type_éléments_du_tableau*






# Exemple à une dimension en pseudo

- types
  vecteur = tableau [1..3] de reel

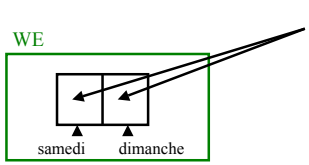



Ce tableau regroupe 3 réels numérotés de 1 à 3



# Autre exemple

- types
  WE = tableau [samedi..dimanche] de reel




Ce tableau regroupe 2 réels numérotés de samedi à dimanche (devant être définis dans un type énuméré)



# Accès aux éléments d'un tableau

- Pour accéder à un élément du tableau, il suffit de donner le nom du tableau et de mettre entre crochets l'indice de l'élément que l'on veut manipuler
  *nom_tableau*[*indice*]
- L'indice doit **impérativement** être compris entre les bornes min et max données lors de la déclaration du tableau. Si ce n'est pas le cas, votre programme plante.

## Exemple d'accès en pseudo

variables
    v:vecteur
debut
    v[1]←0
    v[2] ←v[1]
    v[3] ←v[2]
fin



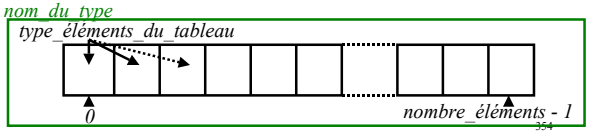
## Les indices en C/C++/Java

- Dans ces langages, le programmeur ne peut pas choisir le plus petit indice du tableau, qui est toujours égal à 0.
- On indique uniquement le nombre *n* d'éléments qui seront dans le tableau.
- Ces éléments seront donc indicés de 0 à *n-1*
- Il est de ce fait courant d'utiliser des boucles de la forme for(int i=0;i<*n*;i++)



## Tableau à une dimension en C/C++

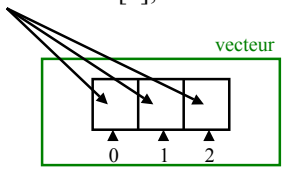
- En C/C++, on définira un type tableau comme ceci :
  typedef
    *type_éléments_du_tableau*
      *nom_du_type*[*nombre_éléments*];






## Exemple à une dimension en C/C++

- typedef
    float vecteur[3];

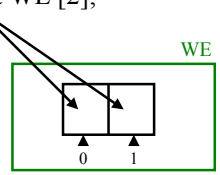



Ce tableau regroupe 3 réels numérotés de 0 à 2



## Autre exemple

- types
    float WE [2];




Ce tableau regroupe 2 réels numérotés de 0 à 1
(il n'est pas possible ici d'utiliser samedi et dimanche comme indice)



## Exemple d'accès en C/C++

```
int main()
{
    vecteur v;

    v[0]=0;
    v[1]=v[0];
    v[2]=v[1];
}
```



## Autre forme de déclaration

- En C/C++, on peut déclarer une variable (ou un paramètre) de type tableau sans définir de type
  Ex:
    int tab[5]; // définit une variable tab
      // qui est un tableau de 5 entiers
- Cette possibilité doit être utilisée avec discernement



## Et si l'indice est en dehors des bornes ?

- Certains langages (Ada, Pascal, Java…) vont s'apercevoir de l'erreur et arrêteront le programme.
- D'autres langages (C,C++) ne font aucune vérification pour aller le plus vite possible. Ils supposent que le programmeur (*vous !*) est une personne intelligente qui a réfléchi à ce qu'elle écrivait. Que se passe-t-il si tel n'est pas le cas ?



## Débordement de tableau en C/C++

- Dans le meilleur des cas, l'utilisation d'un indice en dehors des limites du tableau provoque l'accès à une partie de la mémoire qui n'appartient pas au programme (segmentation violation ⇒ core dump). Le débogueur permet de retrouver facilement l'endroit où s'est produit cette erreur.

# Débordement de tableau en C/C++ (2)

- Dans le pire des cas, l'utilisation d'un indice en dehors des limites du tableau provoque l'accès à une partie de la mémoire qui n'appartient pas au tableau, mais appartient malgré tout au reste du programme. On se retrouve alors en train de manipuler une autre variable du programme (ou un autre bout de variable) sans même s'en rendre compte. Ce genre d'erreur est excessivement difficile à retrouver.



# Exemples d'erreurs

```
int tab[2];
float r;
int i;
int main()
{
    i=tab[-1]; // génèrera probablement (?) une
               // segmentation fault
    tab[2]=0; // va probablement (?) modifier de
              // manière aléatoire la variable r au
              // point de la rendre incohérente.
}
```



# Morale

- Il **vous** appartient (en tant qu'être supérieurement intelligent) de **garantir** que l'indice que vous utilisez pour accéder à un élément du tableau est toujours compris dans les bornes du tableau.
- Sinon…



# Passage des tableaux en paramètres

- Un tableau a très souvent une taille qui n'est pas négligeable. Des opérations de copie de tableau peuvent très vite se révéler coûteuses.
- De ce fait, il n'est pas rare que les langages traitent les tableaux de manière spécifique pour le passage de paramètre.
- En C/C++, un paramètre formel de type tableau représente directement le paramètre effectif qui a été transmis (aucune opération de copie) : cela revient à un passage par référence systématique



# En résumé

- En C/C++, un tableau est donc toujours passé par référence.
- Un tableau ne peut pas être passé par valeur (si ce n'est en l'incluant dans une structure)
- Il ne faut donc pas mettre de & pour un paramètre formel de type tableau que l'on voudrait passer en résultat ou donnée/résultat.



# Quelques détails

- En fait, en C/C++, le nom d'un tableau désigne l'adresse dans la mémoire de la première case du tableau.
- Quand on transmet par valeur un tableau, on transmet en fait par valeur l'adresse de la première case du paramètre effectif, ce qui revient à un passage par référence.



# Exemple

```
void RAZ(int tab[5])
{
    // le passage par valeur du tableau est en fait
    // équivalent à un passage par référence
    for(int i=0;i<5;i++)
        tab[i]=0;
}
int main()
{
    int tableau[5];
    RAZ(tableau);
    // tous les éléments de tableau sont maintenant à 0
}
```



# Comparaison et affectation de tableau

- Comme le nom d'un tableau représente la position en mémoire du premier élément du tableau, comparer deux tableaux ou tenter une affectation entre tableaux n'a pas de sens en C/C++.

```
int t1[5],t2[5];
if (t1= =t2) // accepté par le compilateur
             // mais toujours faux.
if (t1<t2) // accepté par le compilateur
           // mais sans signification
t1=t2; // normalement rejeté par le compilateur
```



# Les gros tableaux

- Les très gros tableaux ne doivent pas être déclarés en variable locale car la place disponible dans la pile d'exécution est en général limitée
- De trop gros tableaux doivent donc être déclarées en variable globale (en gérant les risques associés au partage des variables globales)

## Tableau comme résultat de fonction

- Il ne faut pas renvoyer comme résultat d'une fonction un tableau déclaré en variable locale. En effet, on renvoie dans ce cas l'adresse du premier élément du tableau qui est une variable locale qui sera détruite dès la fin de la fonction. Autrement dit, on renvoie dans ce cas une information qui sera immédiatement périmée.



## Tableaux à plusieurs dimensions

- Il est possible d'indexer les éléments d'un tableau par plus d'un indice. On dit alors qu'on a un tableau à plusieurs dimensions.
- Il est courant d'avoir des tableaux à 2 dimensions (matrice,…) voire 3 dimensions.



## Exemples en pseudo

```
types
    matrice = tableau[1..3,1..3] de reel
/* un tableau de 9 réels indicés de 1 à 3 sur la
première et sur la deuxième dimension */
    EmploiDuTemps = tableau
        [lundi..dimanche,8..20] de chaine
/* un tableau de chaînes indexé par les jours de la
semaine et par les heures de la journée */
```



## Exemples d'accès en pseudo

```
variables
    m : matrice
    edt : EmploiDuTemps;
debut
    m[1,2] ←0
    edt[mardi,16] ←  " Cours d'algo "
fin
```



## Tableau multidimensionnels en C/C++

- En C/C++, il n'existe pas à proprement parler de tableau à plusieurs dimensions. On utilise à la place des tableaux de tableaux (ce qui revient au même)
- C/C++ transforme ce tableau en tableau unidimensionnel en juxtaposant les tableaux les uns à la suite des autres.



## Exemples en C/C++

```
typedef
    float matrice[3][3];
/* un tableau de 3 tableaux de 3 réels (indexation
de 0 à 2 sur chaque dimension) */
typedef
    string EmploiDuTemps[7][20-8+1];
/* un tableau de 7 tableaux de 13 créneaux
horaires */
```



## Exemples d'accès en C/C++

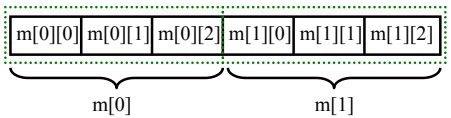
```
int main()
{
    matrice m;
    EmploiDuTemps edt;
    m[1][2]=0;
    edt[1][8]= " Cours d'algo" // le mardi a 16H !
}
```

*Surtout pas de virgule !!*



## Représentation en mémoire

float m[2][3];




Un tableau de 2 tableaux de 3 réels. Les éléments sont stockés consécutivement en mémoire.



## Quelle convention ?

- Quelle convention pour représenter une matrice ?
  - ligne,colonne (comme en math.)
  - ou bien x,y (ce qui revient à dire colonne, ligne)
- À vous de choisir la convention la plus adaptée à votre problème. Dans tous les cas, il faut préciser le choix que vous avez fait (en commentaire) en indiquant clairement ce que représente chaque dimension du tableau.

# Tableaux de taille quelconque en paramètre

- En C/C++, on peut omettre de spécifier la **première** dimension d'un tableau dans la spécification d'un paramètre formel de manière à accepter n'importe quelle taille de tableau sur cette **première** dimension (les tailles sur les autres dimensions *doivent* être fixées)
- Il appartient au programmeur de se débrouiller pour connaître la taille effective du tableau pour ne pas en sortir (on peut utiliser un autre paramètre qui contiendra la taille par exemple)



# Exemple

```
void affiche(int tab[][3], int taille)
// accès par tab[ligne][colonne]
{
    for(int y=0;y<taille;y++)
    {
        for(int x=0;x<3;x++)
            cout << tab[y][x];
        cout << endl;
    }
}
```



# Exemple (2)

```
int main()
{
    int t1[5][3];
    int t2[3][3];
    …
    affiche(t1,5);
    affiche(t2,3);
}
```



# Initialiseur de tableau

- Les variables peuvent être initialisées dès leur déclaration
  int n=0;
- Pour initialiser un tableau, on liste entre accolades ses élements séparés par des virgules
  int tab[3]={0,1,2};
- On peut laisser l'ordinateur compter le nombre d'éléments de l'initialiseur sur la **première** dimension en omettant de spécifier la **première** dimension du tableau. La taille du tableau est alors définie par le nombre d'éléments dans le tableau.
  int tab[][3]={{0,1,2},{3,4,5},{6,7,8}};



# Initialiseur de structure

- On peut utiliser le même notation pour initialiser directement des structures
  Exemple :

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

complexe zero={0,0},i={0,1};
```



# Tableau ou structure ?

- On utilise un tableau pour des informations homogènes (de par leur type et leur utilisation)
- On utilise une structure pour des informations hétérogènes (de par leur type ou leur utilisation)



# Mélange de tableaux et de structures

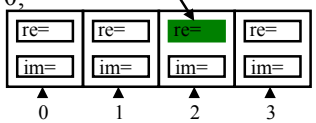
- On peut bien sûr créer
  - des tableaux de structures
  - des structures contenant des tableaux
  - des structures contenant des tableaux de structures
  - des tableaux de structures contenant des tableaux
  - des tableaux de structures contenant des tableaux de structures
  - …



# Exemple

- tableau de complexes

```
typedef
    struct {float re,im;} Complexe;
typedef
    Complexe TableauComplexes[4];
TableauComplexes t;
t[2].re=0;
```

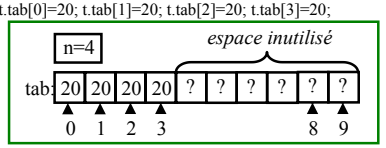





# Exemple

- Un « tableau » de taille « variable »

```
const int TailleMax=10;
typedef
    struct
    {
        int n; // nombre de réels effectivement
               // stockés dans le tableau (<=TailleMax)
        float tab[TailleMax];
    } TableauTailleVariable;
TableauTailleVariable t;
t.n=4; t.tab[0]=20; t.tab[1]=20; t.tab[2]=20; t.tab[3]=20;
```



# Un tableau dans une structure

- Quand un paramètre formel est une structure contenant un tableau, ce paramètre peut être passé par valeur tout à fait normalement et le tableau qu'il contient sera également passé par valeur (il fait partie de la structure).
- Il est toujours possible de faire des affectations entre structures même si elles contiennent des tableaux. La comparaison entre structures reste bien sûr impossible.



# Exemples d'algorithmes sur les tableaux

- On utilisera les définitions suivantes :

```
const int MAX=20;
typedef
    ... element;
    // mettre par exemple float à la place de ...
typedef
    element tableau[MAX];
```



# Somme des éléments

- element doit être un type pour lequel l'addition est définie

```
element Somme(tableau t, int debut, int fin)
// somme des éléments dans le sous tableau [debut,fin[
// version itérative
{
 element somme;

 somme=0; // élément neutre de l'addition

 for(int i=debut;i<fin;i++)
  somme=somme+t[i];

 return somme;
}
```



# Plus grand élément

```
int PlusGrand(tableau t, int debut, int fin)
// recherche de l'indice du plus grand élément dans le sous tableau
    [debut,fin[
// version itérative
{
 element max;
 int indicemax;
 max=t[debut];
 indicemax=debut;

 for(int i=debut+1;i<fin;i++)
  if (t[i]>max)
  {
   max=t[i];
   indicemax=i;
  }
 return indicemax;
}
```



## Recherche d'un élément dans un tableau non trié

```
int recherche_lineaire1(tableau t, element e)
{
 int i=0;
 bool trouve=false;

 while (i<MAX && !trouve)
  if (t[i]= =e)
   trouve=true;
  else
   i++;

 if (trouve)
  return i;
 else
  return -1;
}
```



## Recherche d'un élément dans un tableau non trié

```
int recherche_lineaire2(tableau t, element e)
{
 int i;

 i=0;

 while (i<MAX && t[i]!=e)
  i++;

 if (i!=MAX)
  return i;
 else
  return -1;
}
```



## Recherche d'un élément dans un tableau non trié

```
int recherche_lineaire3(tableau t, element e)
{
 for(int i=0;i<MAX;i++)
  if (t[i]= =e)
   return i;

 return -1;
}
```



# Recherche linéaire dans un tableau trié

```
int recherche_lineaire_trie(tableau t, int debut, int fin, element e)
// recherche dans un tableau [debut,fin[ trié par ordre croissant
// version itérative
{
 int i=debut;

 while (i<fin && t[i]<e)
  i++;

 if (i<fin && t[i]= =e)
  return i;
 else
  return -1;
}
```



## recherche dichotomique dans un tableau trié

```
int recherche_dichotomique(tableau t, int debut, int fin, element e)
// recherche par dichotomie dans [debut,fin[, version itérative
{
 int milieu;  bool trouve=false;

 while (debut<fin && !trouve)
 // tant que l'intervalle n'est pas vide et que je n'ai pas trouvé
 {
  milieu=(debut+fin)/2; // c'est une division entière
  if (t[milieu]= =e)
   trouve=true;
  else
   if (e<t[milieu])
    fin=milieu;
   else
    debut=milieu+1;
 }
…/…
```

recherche dichotomique dans un tableau trié (suite)

```
…/…
if (trouve)
  return milieu;
else
  return -1;
}
```



# Les chaînes de caractères

- Fondamentalement, une chaîne de caractères est un tableau de caractères qui doit pouvoir contenir des mots ou phrases de tailles différentes.
- Le tableau utilisé doit être suffisamment grand pour pouvoir contenir la plus grande phrase que l'on veut conserver.
- La phrase est toujours conservée dans les premières cases du tableau.
- Les dernières cases ne seront pas utilisées chaque fois que la phrase est plus courte que la taille maximale.



# Les chaînes de caractères (2)

- Il est possible d'accéder à une chaîne de caractères s, caractère par caractère, en utilisant la notation de tableau s[i]. Chaque caractère peut être manipulé et modifié individuellement.
- Il existe deux grandes représentations des chaînes:
  - à la C
  - à la Pascal

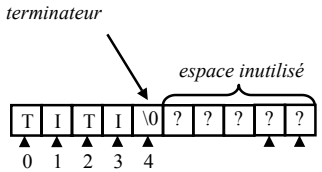


# Représentation à la C

- Une chaîne est un simple tableau de caractère. La fin de la partie utilisée du tableau est marquée par un caractère spécial appelé terminateur (caractère '\0' de code ascii 0). Ce terminateur occupe la première case du tableau qui n'est pas occupée par la phrase. Il ne peut pas figurer dans la phrase. Cette technique permet de gérer des chaînes de longueur quelconque.



# Exemple de chaîne à la C






# Déclaration de chaîne à la C

- char chaine[taille]
- char chaine[];
- char *chaine;
- char chaine[]="La chaine";
- char *chaine="La chaine";

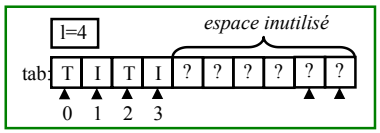


# Représentation à la Pascal

- Une chaîne est une structure regroupant
  - un (petit) entier représentant le nombre de caractères présents dans la chaîne (sa longueur)
  - un tableau contenant les caractères
- On peut alors utiliser n'importe quel caractère dans la chaîne mais le nombre de caractères peut être limité par la taille du (petit) entier et du tableau.



# Exemple de chaîne à la Pascal






# Déclaration de chaîne à la Pascal

- const int TAILLE_MAX=100;
  typedef
  struct
  {
  int longueur;
  char chaine[TAILLE_MAX];
  } chaine_pascal;

# En C++

- C++ propose les chaînes de C ainsi que le type string qui fonctionne sur le principe de Pascal mais ne limite pas la taille des chaînes (redimensionnement automatique du tableau)
- Les opérations sur les chaînes sont aussi facilitée avec ce type string, c'est donc celui que nous utiliserons. (voir feuille de résumé)
- Le type string est en fait un *objet*



# Brève présentation de la notion d'objet

- Un objet est le regroupement de données (comme dans une structure) et des procédures ou fonctions qui manipulent ces données.
- On utilise le terme de classe (au lieu de structure) pour ce regroupement.



# Avantage des objets

- L'avantage d'un objet est qu'il permet de regrouper tout ce qui permet de travailler sur un certain type.
- Un objet doit prévoir des méthodes d'initialisation et de finalisation (constructeur et destructeur)
- On peut contrôler l'accès aux données de l'objet et interdire des modifications non prévues (notion d'encapsulation)
- On peut construire un objet à partir d'un autre (héritage)
- On peut obtenir une vue homogène d'objets hétérogènes (polymorphisme)
- …



# Exemple d'objet en C++

```
class Compteur
{
  private:
     int cpt;
  public:
    // un constructeur par défaut
    Compteur( ) {cpt=0;}
    // un autre constructeur
    Compteur(int n) {cpt=n;}
    // des méthodes de l'objet
    void ajouter(int n) { cpt=cpt+n; }
    int valeur( ) { return cpt; }
};
```



# Appel des méthodes

- Les procédures et fonctions définies pour un objet (on les appelle des méthodes) appartiennent à cet objet. Il faut donc, comme pour une structure, utiliser la notation pointée à partir d'une variable pour utiliser ces méthodes.
- L'objet à partir duquel on effectue cet appel est implicitement transmis en paramètre à la méthode (paramètre this)



# Exemple en C++

```
Compteur c1; // un compteur initialisé à 0 par défaut
Compteur c2(10); // un compteur initialisé à 10

c1.ajouter(5); // modifier c1 en lui ajoutant 5

c2.ajouter(c1.valeur( ));
        // modifier c2 en lui ajoutant la valeur de c1

cout << c2.valeur();
```



# Les string de C++

- Comme les chaînes de C++ (string) sont des objets, il faut également utiliser la notation pointée pour utiliser les méthodes qu'elles fournissent.
- Exemple :

```
string s="Un test";
cout << s << " contient " << s.length( )
        << " caractères\n";
```



# Les pointeurs



# Les pointeurs

- Les pointeurs sont l'une des notions les plus puissantes en informatique. Comme tout outil puissant, ils présentent, quand ils sont mal utilisés, des dangers qu'il ne faut pas sous-estimer.
- Même dans les langages où ils sont cachés (Java), les pointeurs sont à la base de toute manipulation évoluée.

# Rappel d'architecture

- La mémoire d'un ordinateur classique est composée d'une succession d'octets, chacun étant repéré par un numéro que l'on appelle adresse.
- On pourrait représenter la mémoire comme ceci

```
const int taille_memoire=256*1024*1024; // 256  Mo
typedef unsigned char octet;
typedef octet memoire[taille_memoire];
```

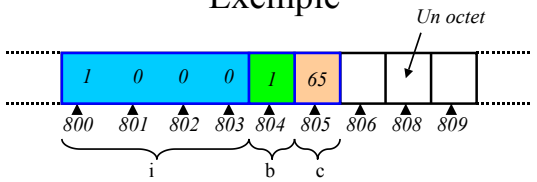


# Les variables en mémoire

- Toute variable est stockée en mémoire dans un certain nombre d'octets consécutifs. Pour tout connaître d'une variable, il faut savoir
  - Son type : il détermine combien de place la variable occupe et où sont rangées les différentes informations qu'elle peut contenir
  - La position du premier octet qu'elle occupe dans la mémoire de l'ordinateur : c'est ce qu'on appelle son adresse ou encore le *pointeur* sur cette variable



# Exemple




```
int i=1;
bool b= true;
char c='A';
```

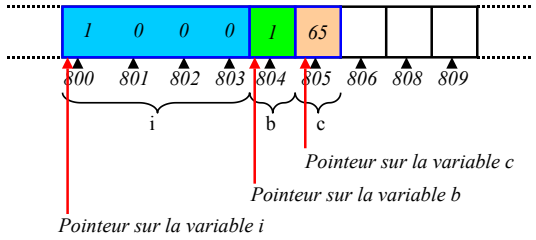


# Variations dans les représentations

- Attention, d'une machine à l'autre ou d'un compilateur à l'autre, il peut y avoir des différences :
  - dans la taille des entiers
  - dans la représentation des entiers (MSB en premier (big endian) ou LSB en premier (little endian))
  - dans la représentation des booléens (en C, 0 représente false, toute autre valeur représente true)
  - …



# Exemple de pointeurs






# Pointeur

- Un pointeur représente la position en mémoire d'une variable. Plus précisément, c'est la position en mémoire du premier octet de cette variable. Pour pouvoir être utilisable, il faut connaître le type de la variable qui se trouve au bout du pointeur.
- On représentera souvent un pointeur sous la forme d'une flèche qui "pointe" sur la variable.
- Un pointeur est stocké dans une variable



# Type pointeur en pseudo

- Pour définir un type pointeur sur une variable de type T, on écrit :

```
types
    pointeurSurT = ^T
```

- Le chapeau devant un nom de type se lit "pointeur sur"

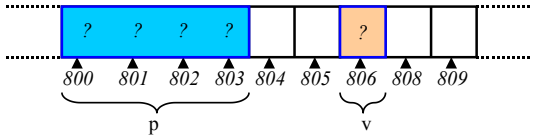


# Variable pointeur en pseudo

```
variables
    p : pointeurSurT // p va pointer sur une
    // variable de type T. On pourrait écrire
    // directement p : ^T
    v : T
debut
    p ← adresse(v); // p pointe sur la variable v
    p^ ←5 // la variable pointée par p reçoit la valeur 5
fin
```

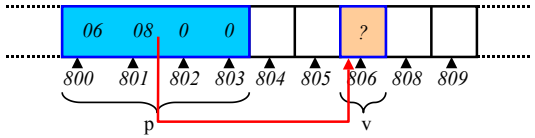


# Illustration




→ p ← adresse(v); // p pointe sur la variable v
p^ ←5 // la variable pointée par p reçoit la valeur 5

# Illustration

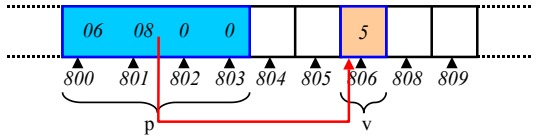



p ← adresse(v); // p pointe sur la variable v

p^ ←5 // la variable pointée par p reçoit la valeur 5



# Illustration




p ← adresse(v); // p pointe sur la variable v

p^ ←5 // la variable pointée par p reçoit la valeur 5



# En C++

```
typedef
   T *pointeurSurT; // type d'un pointeur sur une
                    // variable de type T
T v; // v est une variable de type T
pointeurSurT p; // p est une variable qui contient un
        // pointeur sur une variable de type T
…
p=&v; // p reçoit l'adresse de v (p pointe sur v)
*p=5; // la variable pointée par p reçoit la valeur 5
```



# En C++, sans utiliser de type intermédiaire

```
T v;    // v est une variable de type T
T *p;   // p est une variable qui contient un
        // pointeur sur une variable de type T
…
p=&v; // p reçoit l'adresse de v (p pointe sur v)
*p=5; // la variable pointée par p reçoit la valeur 5
```



# En C++

- On parlera du type T * pour désigner un pointeur sur une variable de type T.
- Attention ! Il faut considérer que l'étoile colle au nom de la variable et non pas au nom du type. En effet,
  int *p,v;
  signifie qu'on définit une variable p de type int * et une variable v de type int. On aurait pu écrire aussi
  int v,*p;
  pour définir les mêmes variables



# Déréférence

- L'opération qui consiste à aller rechercher la variable qui se trouve au bout d'un pointeur (comme dans *p) s'appelle une déréférence.



# Pointeur nul

- Quand on veut indiquer qu'un pointeur ne contient l'adresse d'aucune variable, on l'initialise avec une valeur spéciale. En pseudo, on l'appelle nil, en C++, on l'appelle NULL.
- p=NULL; signifie p ne contient plus l'adresse d'une variable
- Il est interdit de déréférencer un pointeur NULL (segmentation fault en C/C++ ou null pointer exception en Java)



# Taille d'une variable

- Il arrive parfois qu'on ait besoin de connaître le nombre d'octets nécessaires pour stocker une variable v de type T
- En C/C++, on dispose de l'opérateur sizeof pour connaître cette information. Exemples :
  sizeof(T) *ou bien*
  sizeof(v)



# Passage par adresse en C

- Dans le langage C, le passage par référence n'existe pas. Quand on veut transmettre un paramètre en donnée/résultat, il faut utiliser un passage par adresse.
- Le passage par adresse consiste à passer (par valeur), un pointeur sur le paramètre effectif (qui est donc forcément une variable)
- Quand on veut utiliser le paramètre, il faut systématiquement déréfencer le pointeur.

# Exemple de passage par adresse en C

```
void Echanger(int *pA, int *pB) // D/R, D/R
{
   int tmp;
   tmp=*pA;
   *pA=*pB;
   *pB=tmp
}

int main( )
{
   int a=2,b=3;
   Echanger(&a,&b);
}
```

*On passe l'adresse du paramètre effectif*

*Quand on veut utiliser le paramètre, il faut* déréférencer *le pointeur*

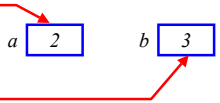
*On passe le pointeur sur la variable qui est le paramètre*



# Exemple de passage par adresse en C

```
void Echanger(int *pA, int *pB) // D/R, D/R
{
   int tmp;
   tmp=*pA;
   *pA=*pB;
   *pB=tmp
}

int main( )
{
   int a=2,b=3;
   Echanger(&a,&b);
}
```

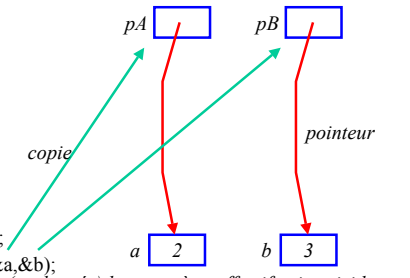





# Exemple de passage par adresse en C

```
void Echanger(int *pA, int *pB) // D/R, D/R
{
   int tmp;
   tmp=*pA;
   *pA=*pB;
   *pB=tmp
}

int main( )
{
   int a=2,b=3;
   Echanger(&a,&b);
}
```



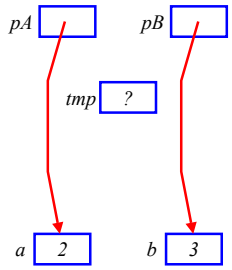

*On copie (en donnée) le paramètre effectif qui est ici la constante représentant l'adresse de la variable dans le paramètre formel*



# Exemple de passage par adresse en C

```
void Echanger(int *pA, int *pB) // D/R, D/R
{
   int tmp;
   tmp=*pA;
   *pA=*pB;
   *pB=tmp
}

int main( )
{
   int a=2,b=3;
   Echanger(&a,&b);
}
```

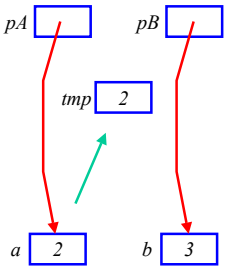





# Exemple de passage par adresse en C

```
void Echanger(int *pA, int *pB) // D/R, D/R
{
   int tmp;
   tmp=*pA;
   *pA=*pB;
   *pB=tmp
}

int main( )
{
   int a=2,b=3;
   Echanger(&a,&b);
}
```

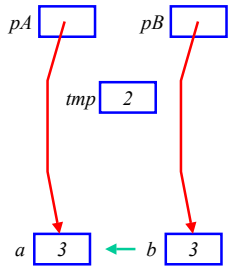





# Exemple de passage par adresse en C

```
void Echanger(int *pA, int *pB) // D/R, D/R
{
   int tmp;
   tmp=*pA;
   *pA=*pB;
   *pB=tmp
}

int main( )
{
   int a=2,b=3;
   Echanger(&a,&b);
}
```

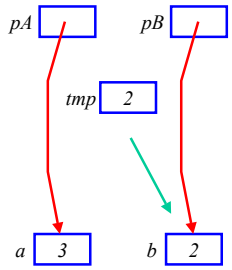





# Exemple de passage par adresse en C

```
void Echanger(int *pA, int *pB) // D/R, D/R
{
   int tmp;
   tmp=*pA;
   *pA=*pB;
   *pB=tmp
}

int main( )
{
   int a=2,b=3;
   Echanger(&a,&b);
}
```






# Exemple de passage par adresse en C

```
void Echanger(int *pA, int *pB) // D/R, D/R
{
   int tmp;
   tmp=*pA;
   *pA=*pB;
   *pB=tmp
}

int main( )
{
   int a=2,b=3;
   Echanger(&a,&b);
}
```




# Passage par référence en C++

- Le passage par référence en C++ fonctionne exactement de la même manière. La seule différence est que c'est le compilateur qui se charge de transmettre un pointeur sur le paramètre effectif et de déréférencer systématiquement le pointeur passé en paramètre formel.
- Cela simplifie beaucoup les notations et évite les erreurs !

## Comparaison C/C++

```
void Echanger(int *pA, int *pB)
// D/R, D/R
{
   int tmp;
   tmp=*pA;
   *pA=*pB;
   *pB=tmp
}
```

*Langage C*

```
int main( )
{
   int a=2,b=3;
   Echanger(&a,&b);
}
```

```
void Echanger(int &A, int &B)
// D/R, D/R
{
   int tmp;
   tmp=A;
   A=B;
   B=tmp
}
```

*Langage C++*

```
int main( )
{
   int a=2,b=3;
   Echanger(a,b);
}
```

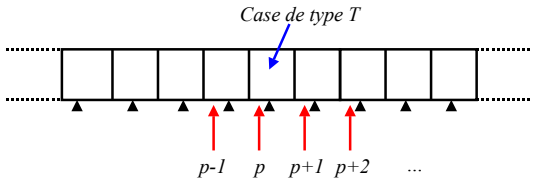


## Arithmétique sur les pointeurs

- Soit p un pointeur sur un type T et n un entier
- L'arithmétique sur les pointeurs consiste à faire comme si p pointait sur une case d'un tableau d'éléments de type T (tableau à une dimension), que ce soit le cas ou pas.



## Arithmétique sur les pointeurs




- L'écriture p+n représente la case du tableau située n cases après la case pointée par p.
- Quand n est négatif, il s'agit d'une case située avant p dans le tableau.



## Arithmétique sur les pointeurs détail

- Attention ! Écrire p+n ne signifie en aucun cas ajouter n à l'adresse contenu dans le pointeur p. Cela signifie en fait ajouter n*sizeof(T) à l'adresse mémoire contenue dans le pointeur p (de type pointeur sur T).



## Arithmétique sur les pointeurs

- On utilisera souvent :

```
p++; // pour passer à la case suivante d'un tableau
p=p+n; // pour avancer de n cases (n>0)
p--; // pour passer à la case précédente d'un tableau
p=p-n; // pour reculer de n cases (n>0)
```

- Après ces opérations, il faut être sûr que l'on ne pointe pas n'importe où dans la mémoire !!
- Si p1 et p2 sont tous les deux des pointeurs sur des cases d'un même tableau, on pourra écrire p2-p1 pour savoir combien de cases les séparent.



## Pointeurs et tableaux en C/C++

- On rappelle que le nom d'un tableau représente en fait le pointeur sur la première case du tableau.
- Dans int tab[5], tab est en fait un pointeur constant de type (int *) sur la première case du tableau.
- On peut donc écrire, lorsque p est du type (int *)

```
p=tab; // ce qui revient à p=&tab[0];
p=tab+1; // ce qui revient à p=&tab[1];
p=tab+2; // ce qui revient à p=&tab[2];
…
```



## Pointeurs et tableaux en C/C++

- En fait, quand tab est un tableau, la notation permettant d'accéder à une case du tableau

```
tab[n]
```

est en fait équivalente à

```
*(tab + n)
```



## Pointeurs et tableaux en C/C++

- Comme le nom d'un tableau représente en fait le pointeur sur la première case du tableau, on a en fait systématiquement un passage par adresse des variables de type tableau dans les proc./fonc. Cela revient à un passage par référence.

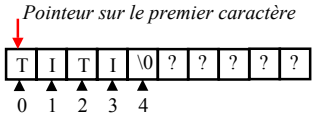


## Pointeurs et tableaux en C/C++

- Quand on veut manipuler un tableau sans préciser le nombre de cases qu'il contient, on utilisera un pointeur sur la première case de ce tableau. Les notations int tab[] et int *tab sont équivalentes. Attention ! Ces notations ne peuvent être utilisées que pour des paramètres formels ou pour l'allocation de variables dynamiques !

# Exemple : fonctions de manipulation de chaînes à la C

- Rappel :
  une chaîne "à la C" est un tableau de caractères qui contient les caractères de la chaîne. La fin de la chaîne est marquée par un terminateur (caractère de code ascii 0)
- Très souvent, on utilise le type char * pour ces chaînes de caractères (on mémorise le pointeur sur le premier caractère de cette chaîne)






# Manipulation de chaînes à la C : calcul de la longueur

```
int strlen(char *s) // retourne la longueur de la chaîne s
{
   int longueur=0;
   while (*s!=0)
   {
        s++;
        longueur++;
   }
   return longueur;
}
```



# Manipulation de chaînes à la C : calcul de la longueur (variante)

```
int strlen(char *s) // retourne la longueur de la chaîne s
{
   char *debut=s;
   while (*s!=0)
        s++;

   return s-debut;
}
```



# Manipulation de chaînes à la C : recherche d'un caractère

```
char *strchr(char *s, char c)
// retourne un pointeur sur la première occurrence du
// caractère c dans la chaîne s (ou NULL si non trouvé)
{
   while (*s!=0 && *s!=c)
        s++;

   if (*s= = 0)
        s=NULL;

   return s;
}
```



# Manipulation de chaînes à la C : recherche d'un caractère

```
char *strrchr(char *s, char c)
// retourne un pointeur sur la dernière occurrence du
// caractère c dans la chaîne s (ou NULL si non trouvé)
{
   char *p=NULL;
   while (*s!=0)
   {
        if (*s= =c)
          p=s;
        s++;
   }
   return p;
}
```



# Manipulation de chaînes à la C : comparaison de deux chaînes

```
int strcmp(char *s1, char *s2)
// retourne 0 si s1 est la même chaîne que s2 ou bien un nombre
// négatif (resp. positif) si s1 est plus petite (resp. plus grande)
// que s2 selon l'ordre du dictionnaire (ordre lexicographique)
{
   while (*s1!=0 && *s1= =*s2)
   {
        s1++; // sauter les caractères identiques
        s2++;
   }
   return (int)*s1-(int)*s2;
}
```



# Pointeurs et structures en C/C++

- Pour accéder à un champ v d'une structure pointée par p, il faut écrire
  (*p).v
  ce qui signifie déréférencer p pour obtenir la variable structure qui se trouve au bout du pointeur et ouvrir cette structure pour accéder au champ v. Les parenthèses sont nécessaires (cf. priorité des opérateurs * et .) car *p.v signifie "obtenir la variable qui se trouve au bout du pointeur stocké dans le champ v de la variable p"
- Une notation équivalente mais plus claire consiste à écrire
  p->v
  elle signifie "obtenir le champ v de la structure pointée par p"



# Pointeur sans type réservé aux pros !

- Dans des cas très particuliers, on veut mémoriser une adresse mémoire sans se souvenir du type d'information sur lequel on pointe.
- Dans ce cas, on utilise le type void * qui signifie "pointeur sur n'importe quoi".
- Il n'est pas possible de déréférencer un pointeur void * (puisqu'on ne sait pas ce qu'il y a au bout)
- On peut par contre convertir une information void * en un pointeur sur un type T quelconque (sous la seule responsabilité du programmeur : dangereux !)
- Cette possibilité est intéressante pour écrire des fonctions génériques en C. En C++, il convient d'utiliser d'autres techniques beaucoup plus sûres (polymorphisme ou templates)



# Conversions entre types pointeurs réservé aux pros !

- On peut demander à ce qu'un pointeur p sur un type T soit en fait considéré comme étant un pointeur sur un type U. Cela s'écrit avec un cast
  (U *)p
- Le programmeur a la seule responsabilité de ce type d'opération. Il doit s'assurer que cette conversion a un sens ce qui implique de connaître précisément la représentation mémoire des variables.

## Conversions entre types pointeurs réservé aux pros !

```
typedef
   struct
   {
        int x,y;
   } point;
point a={1,2};
point *p=&a;
*(int *)p=3; // revient à ranger 3 dans p->x
*(((int *)p)+1)=3; // revient à ranger 3 dans p->y
```



## Allocation dynamique



## Allocation dynamique

- L'un des premiers usages des pointeurs est de pouvoir créer des variables en cours d'exécution du programme.
- Jusqu'ici, les variables que l'on avait utilisées étaient soit globales (existant du début jusqu'à la fin du programme), soit locales (existant uniquement à l'intérieur d'une proc./fonc.). Dans les deux cas, il faut prévoir l'utilisation d'une variable avant la compilation du programme. On parle d'allocation statique.



## Allocation dynamique (2)

- Dans certains cas, on ne peut pas savoir avant la compilation du programme combien de variables vont être nécessaires ou quelle sera la taille de tableau dont on aura besoin.
- Dans ces cas là, il est nécessaire de créer les variables pendant l'exécution du programme. On parle d'allocation dynamique.



## Identifiant des variables dynamiques

- Quand on crée une variable dynamique, il faut bien sûr que le système nous fournisse un identifiant de cette variable pour qu'on puisse la distinguer des autres variables.
- Cet identifiant est le pointeur sur la variable créée.
- Pour une variable statique, l'identifiant de la variable que l'on manipule est son nom mais il faut savoir que le compilateur remplace systématiquement le nom de la variable par le pointeur sur cette variable.



## Allocation dynamique 1ère étape : création

- Pour créer une variable de type T au cours de l'exécution d'un programme, il faut disposer d'une variable *p* de type pointeur sur T et écrire
  - En pseudo :
    p←nouveau T
  - En C++ :
    p=new T;
- La variable p contient alors le pointeur sur la variable nouvellement créée



## Allocation dynamique 2ème étape : utilisation

- Une fois la variable créée, on la manipule directement par l'intermédiaire de son pointeur
  - En pseudo :
    p^
  - En C++ :
    *p



## Allocation dynamique 3ème étape : destruction

- Si à un moment donnée, la variable allouée n'a plus d'utilité, on peut la supprimer et libérer la place qu'elle occupait.
  - En pseudo :
    détruire p
  - En C++ :
    delete p;
- Il faut bien sûr que p soit un pointeur sur une variable allouée dynamiquement (pas question de détruire une variable statique)
- Une fois la variable détruite, on ne doit plus l'utiliser sous aucun prétexte. Il peut être prudent d'affecter NULL à p



## Remarques sur la destruction

- Une variable allouée dynamiquement existe de sa création jusqu'au moment où on la détruit. Elle n'est en aucun cas supprimée quand on sort de la proc./fonc. où elle a été créée.
- Il faut faire attention à ne pas perdre le pointeur que l'on a obtenu sur une variable dynamique car sinon il sera impossible de la détruire et l'on consommera de la mémoire pour rien (memory leak)
- Certains langages (Java) se chargent de détruire automatiquement les variables dynamiques qui ne sont plus accessibles (ramasse-miettes/garbage collector). Cela présente des avantages et des inconvénients.

# Opérateur new

- L'opérateur new de C++ crée donc une nouvelle variable du type T qui lui est indiqué.
- Il renvoie un pointeur sur cette variable créée, donc il renvoie une information de type T * (pointeur sur T)
- S'il n'y a plus assez de mémoire, l'opérateur new générera une erreur.



# Contrôlez vos types !

- Une méthode simple pour éviter de nombreuses erreurs et de vérifier le type de ce que l'on écrit et de s'assurer qu'on ne mélange pas des informations de type différents !
- Exemple :

```
int *p=new int; // OK, je range un int *
                // dans un int *
*p=5; // OK, je range un entier dans un entier
p=5; // FAUX, je range un entier dans
     //  un pointeur sur entier !!
```



# Allocation de tableaux à une dimension

- L'allocation de tableau à une dimension utilise une syntaxe particulière en C++
- Allouer un tableau de 10 entiers

```
int *tab=new int[10];
```

on obtient un pointeur sur la première case

- Accéder à la case i du tableau

```
tab[i]=…; // équivalent à *(tab+i)=…
```

- Supprimer le tableau

```
delete [ ] tab; // ne pas oublier les crochets
```



# Manipulation de chaînes à la C : duplication

```
char *strdup(char *s)
{
    int longueur=strlen(s);
    char *copie= new char [longueur+1]; // laisser de la place pour le terminateur
    char *p=copie;

    while (*s!=0)
    {
        *p=*s;
        p++;
        s++;
    }

    *p=0;
    return copie;
}
```



# Pointeur sur pointeur …

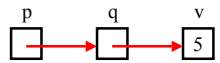
- Si p est un pointeur sur une variable de type T, rien n'interdit de mémoriser l'adresse de ce pointeur p dans un autre pointeur q.
- q sera alors un pointeur sur p qui est lui même un pointeur sur une variable de type T. Donc, q est un pointeur sur pointeur sur T.
- On dit alors en C++ que q est du type T ** (pointeur de pointeur sur T)



# Double déréférence

- Quand p est un pointeur sur un pointeur sur une variable v, il faut à partir de p suivre deux flèches pour arriver à la variable v. Il faut donc faire une double déréférence (deux étoiles) pour arriver à la variable.
- Ex:

```
int v,*q,**p;
// p est un pointeur sur un pointeur sur un entier
q=&v;
p=&q;
**p=5;
```






# Allocation de tableaux à plusieurs dimensions

- Il n'est pas directement possible d'allouer un tableau à plusieurs dimensions. En effet, pour utiliser un tableau à plusieurs dimensions, il faut connaître l'adresse de ce tableau ainsi que *les tailles des différentes dimensions du tableau*. Or, ces dernières informations ne peuvent pas être stockées dans un pointeur.



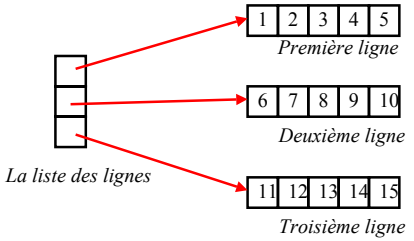
# Allocation de tableaux à plusieurs dimensions

- Quand on a besoin d'un tableau dynamique à plusieurs dimensions, il faut procéder de manière indirecte
  - en considérant qu'un tableau à deux dimensions est une succession de tableaux à une dimension dont on garde les adresses dans un tableau,
  - en considérant qu'un tableau à trois dimensions est une succession de tableaux à deux dimensions dont on garde les adresses dans un tableau,
  - et ainsi de suite…

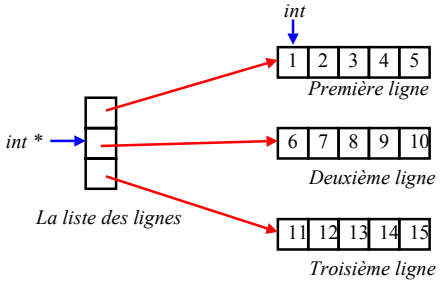


# Allocation de tableaux à 2 dimensions

*Exemple : je veux créer une "matrice" d'entiers de 5 colonnes et 3 lignes. Je crée donc dynamiquement trois lignes (tableau à une dimension) et je mémorise l'adresse de ces tableaux dans un quatrième tableau*



# Allocation de tableaux à 2 dimensions



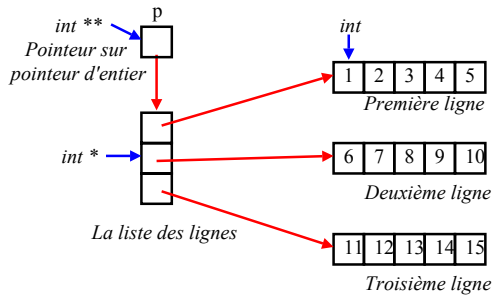



# Allocation de tableaux à 2 dimensions

- Le tableau contenant la "liste" des lignes doit souvent être alloué dynamiquement.
- Ce tableau est un tableau de pointeurs sur des entiers. L'adresse de la première case de ce tableau est un pointeur sur une case de type pointeur sur entier. Autrement dit, le pointeur sur la liste est du type pointeur sur pointeur d'entier (type int **) !
- Pour créer cette liste, je dois donc avoir une variable p de type int ** et appeler new pour créer un tableau d'éléments de type int *



# Allocation de tableaux à 2 dimensions






# Allocation de tableaux à 2 dimensions

- La création s'effectue donc en plusieurs étapes :
  1. Déclaration du pointeur p :
     int **p;
  2. Création de la liste des lignes :
     p=new int *[nbre_lignes];
  3. Création de chaque ligne :
     for(int i=0;i<nbre_lignes;i++)
       p[i]=new int[nbre_colonnes];

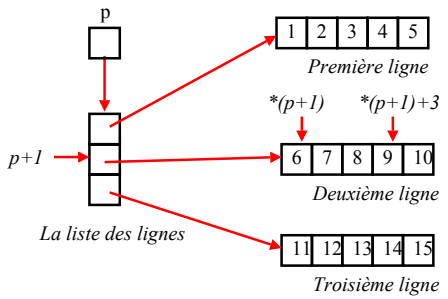


# Utilisation

- Par bonheur, une fois la matrice allouée dynamiquement, on peut accéder à ses éléments avec la notation très classique
  p[ligne][colonne]
- En effet, il faut se souvenir que p[i] signifie en fait *(p+i). Donc p[ligne][colonne] veut dire
  *(*(p+ligne)+colonne)



# Exemple : p[1][3] *(*(p+1)+3)






# Destruction du tableau à deux dimensions

- Quand on veut détruire le tableau, il ne faut surtout pas scier la branche sur laquelle on est assis ! Autrement dit, il ne faut pas commencer par détruire la liste des lignes car ensuite il sera impossible de détruire les lignes.
- Il faut procéder dans l'ordre inverse de la création

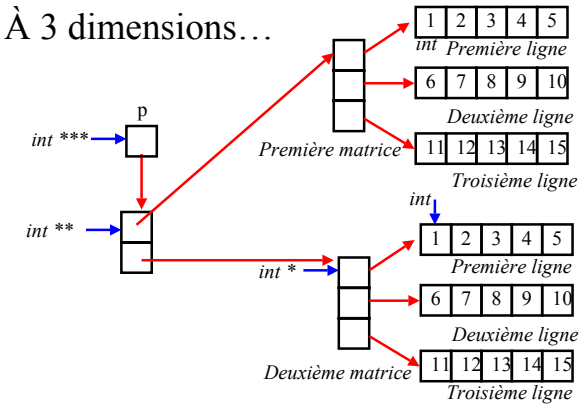


# Destruction du tableau à deux dimensions

1. Destruction des lignes, une par une :
   for(int i=0;i<nbre_lignes;i++)
     delete [] p[i];
2. Destruction de la liste de lignes :
   delete [] p;
3. [optionnel] Par prudence, marquer p comme ne pointant plus sur quoi que ce soit
   p=NULL;



# À 3 dimensions…



# Différents types d'allocation

- En C++, on dispose de différents opérateurs pour allouer ou libérer de la mémoire :
  - new/delete pour les variables simples
  - new [] /delete [] pour les tableaux
  - malloc/free en C
- Il est important de noter qu'on ne peut pas mélanger les opérateurs de création et de destruction. Il est donc interdit de libérer par delete un tableau créé par new [] ou de libérer par free une variable créée par new ou de libérer par delete une variable créée par malloc, …



# Pointeurs et fichiers

- Il n'est pas possible de sauvegarder un pointeur dans un fichier.
- En effet, un pointeur représente la position d'une variable en mémoire. La prochaine fois qu'on exécutera le programme, la variable sera vraisemblablement à un autre position. Le pointeur qu'on aurait sauvegardé dans un fichier ne correspondra plus à rien.



# Les fichiers

Voir poly.



# Notion de fichier

- Un fichier est une séquence d'informations
- Une tête de lecture/écriture permet d'accéder à ces informations, les unes après les autres.
- Quand la tête lit ou écrit une information, elle passe automatiquement à l'information suivante.
- Un fichier a un début et une fin
- Un fichier peut être vide



# Types d'accès

- Accès séquentiel :
  pour accéder à l'information numéro n, il faut lire les n-1 informations qui précédent (ex: bande magnétique)
- Accès aléatoire :
  on peut accéder directement à l'information qui nous intéresse (ex: tableau)



# Fichier texte

- Ils contiennent des caractères
- Avantages :
  - Faciles à visualiser et à modifier
  - Portables
- Inconvénients :
  - Peu efficaces
  - Accès aléatoire quasi impossible



# Fichier binaire

- Ils contiennent des données codées en binaires
- Avantages :
  - Plus efficaces
  - Permet un accès aléatoire
- Inconvénients :
  - Non portables
  - Difficiles à visualiser et modifier



# Les fichiers standards de C++

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

- cin : entrée standard (en général le clavier)
- cout : sortie standard (en général l'écran)
- cerr : sortie d'erreur (en général l'écran)



# Les étapes de manipulation d'un fichier

1. Déclarer une variable qui va mémoriser entre autres la position de la tête dans le fichier
2. Ouvrir le fichier : associer cette variable à un fichier du système de fichier (disque dur ou autre)
3. Lire ou/et écrire en vérifiant systématiquement que l'on n'a pas atteint la fin du fichier ou qu'un autre problème n'est pas survenu
4. Refermer le fichier

# Notions de base sur la compilation

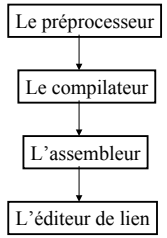


# Le rôle du compilateur

- Le compilateur se charge de traduire un programme écrit dans un langage évolué en langage machine directement compréhensible par le processeur.
- Cette traduction s'effectue en général en plusieurs passes



# Les différentes étapes pour un programme C/C++






# Le préprocesseur

- Le préprocesseur est une sorte de traitement de texte qui effectue des manipulation simple sur le source du programme. Il remplace certains mots par d'autres, inclut des fichiers dans le programme source, supprime des parties du programme qui ne doivent pas être compilées.
- Chaque commande du préprocesseur commence par le symbole #
- Le préprocesseur transforme le programme C/C++ en un programme où ne figurent plus que des instructions C/C++ (plus de #…)



# Les commandes du préprocesseur

- #include <fic.h>
  *inclure le fichier* fic.h *qui se trouve dans les répertoires habituels du compilateur (/usr/include,…)*
- #include "fic.h"
  *inclure le fichier* fic.h *qui se trouve dans le répertoire courant ou dans un répertoire que l'on précise avec l'option –I (exemple : -I/usr/include)*



# Les commandes du préprocesseur

- #define *mot substitut*
  *remplace partout dans le source le mot* mot *par l'expression* substitut*. Seul moyen de définir des constantes en C. Exemple : #define PI 3.14*
- #define *symbole*
  *définit un symbole sans lui donner de valeur. Utilisé pour la compilation conditionnelle*
- Il existe aussi d'autres utilisations de define pour définir des macros (sortes de fonctions simples). Exemple : #define max(a,b) (((a)>(b))?(a):(b))
  Ces macros sont dangereuses et en général obsolètes en C++ (remplacées par des fonctions inline)



# Compilation conditionnelle

- La compilation conditionnelle consiste à choisir quelles parties du programme doivent être effectivement compilées. Elle permet d'écrire en un seul source plusieurs variantes d'un programme. Cela permet par exemple d'écrire des programmes qui peuvent être compilés aussi bien sous Unix que sous Windows.



# Compilation conditionnelle (2)

- #ifdef *symbole*
  …
  #endif
  *ne compile le code (…) que si le symbole a été défini par un #define*
- #ifdef *symbole*
  …
  #else
  …
  #endif
- #ifndef *symbole : compilation uniquement si le symbole n'est pas défini*



# Appel du préprocesseur

- Le préprocesseur peut être appelé indépendamment par la commande cpp (C PréProcesseur) ou par g++ -E
- Le résultat reste un fichier C ou C++

# Le compilateur

- Il transforme le source C/C++ en langage d'assemblage. Le langage d'assemblage est une représentation du langage machine qui reste compréhensible par un humain.
- g++ -S permet d'obtenir la traduction en assembleur (fichier .s)



# L'assembleur

- L'assembleur traduit en langage machine le langage d'assemblage. Il s'agit d'une traduction relativement facile.
- L'assembleur génère un fichier objet (fichier .o) qui contient la représentation en langage machine des fonctions que vous avez écrites
- g++ -c permet d'obtenir le fichier objet à partir du programme source C/C++



# L'éditeur de lien

- Le fichier objet ne contient que le code des fonctions que vous avez écrites. Il ne contient pas le code des fonctions du systèmes qui ont été écrites pour vous (lecture/affichage, fonctions mathématiques,…)
- Ces fonctions se trouvent dans des librairies
- Le rôle de l'éditeur de lien est de réunir toutes les fonctions nécessaires à l'exécution du programme dans un fichier exécutable



# Appel de l'éditeur de lien

- L'éditeur de lien s'appelle par g++ -o *nom_executable liste_de_fichiers_objets liste_de_librairies*
- Certaines librairies sont utilisées par défaut, d'autres doivent être précisées
- On précise une librairie en utilisant l'option –l suivie immédiatement du nom de la librairie. Le fichier contenant la librairie se nomme lib*nom*.* et se trouve souvent dans /lib, /usr/lib ou /usr/X11/lib



# Quelques librairies

- -lstdc++ : librairie standard C++ utilisée par défaut par g++
- -lm : librairie mathématique
- -lX11 : librairie graphique (nécessite quelques connaissances)
- Si les librairies ne sont pas à un emplacement standard, il faut utiliser l'option –L pour indiquer leur répertoire. Ex: -L/home/user/lib



# Édition statique ou dynamique

- Il existe deux types de librairies :
  - Les librairies statiques (fichiers lib*.a)
  - Les librairies dynamiques (fichiers lib*.so, DLL sous Windows)
- Un exécutable compilé statiquement contient toutes les fonctions dont il a besoin. Il est donc complètement indépendant mais est beaucoup plus gros. Plusieurs programmes statiques chargés en mémoire auront leurs propres librairies.
- Un exécutable compilé dynamiquement ne contient pas les librairies systèmes. Elles sont chargée au démarrage du programme. L'exécutable n'est donc plus indépendant, mais il est (beaucoup) plus petit.



# Programmation modulaire

- Quand on écrit un gros programme, il faut décomposer le source en différents modules. Chaque module regroupe les fonctions et variables qui gèrent une partie du problème.
  Exemple : pour un navigateur internet, on écrira un module pour afficher les pages HTML, un module pour afficher les images, un module pour gérer les menus,…



# Module

- Un module est simplement le regroupement de fonctions et variables apparentées dans une même entité.
- La notion de module en C/C++ est assez sommaire. Elle consiste à écrire un fichier .cc qui contient les définitions des fonctions et les déclarations de variables. Un fichier .h va contenir les déclarations de fonctions et de variables. Ce fichier .h devra être inclus (#include) dans les autres modules pour pouvoir en utiliser les fonctions.



# Schéma de fichier .h

- Pour éviter d'avoir plusieurs fois les mêmes définitions si un même fichier .h est inclus plusieurs fois, on utilise le modèle suivant:
  #ifndef _NOMFICHIER_H_
  #define _NOMFICHIER_H_
  …
  *déclarations du fichier .h*
  …
  #endif

# Variables partagées

- Les variables globales qui doivent être utilisées par plusieurs modules doivent être
  - Définies dans le fichier .cc du module
    ex: int NumVersion;
  - Déclarées dans le fichier .h du module
    ex: extern int NumVersion;



# Automatisation de la compilation

- Quand un programme se compose de plusieurs fichiers sources, il est assez lent de tout recompiler lorsque l'on effectue une modification sur un seul fichier. On souhaite n'avoir à recompiler que le fichier qui a été modifié.
- Pour cela, on utilise un Makefile



# Le makefile

- Un fichier Makefile (c'est le nom du fichier) contient des règles qui disent :
  Pour obtenir un fichier *but* , il faut utiliser des fichiers *src1, src2,…*et lancer la commande *commande* qui se charge de créer le *but* à partir des fichiers *src.*
- Ce genre de règle s'écrit
  *but* : *src1 src2 src3*
  ⟶| *commande*
  La commande DOIT être précédée d'une tabulation en début de ligne



# Exemple de fichier Makefile

```
CFLAGS=-g
LDFLAGS=-g
fractale : calcul.o graphique.o
⟶| g++ $(LDFLAGS) -o fractale calcul.o fractale.o
calcul.o : calcul.cc calcul.h graphique.h
⟶| g++ $(CFLAGS) –c calcul.cc
graphique.o : graphique.cc graphique.h
⟶| g++ $(CFLAGS) –c graphique.cc
```



# Lancement de la compilation

- Une fois le fichier Makefile écrit, il suffit de taper make pour lancer la compilation et générer le premier but du Makefile.
- Pour générer un but autre que le premier, il faut indiquer son nom en paramètre quand on lance make.
- Quand un fichier est modifié, le programme make ne lance que les commandes nécessaires pour recréer le but. Il ne recompile pas les sources qui n'ont pas été changés (bien sûr, il faut toujours faire l'édition de lien)
- make se fonde sur la date des fichiers pour prendre sa décision. Si un fichier *src* est plus récent que le fichier *but*, il faut reconstruire le *but*