Introduction à l'algorithmie et au langage C

Florent BERNARD, Brice COLOMBIER

<florent.bernard@univ-st-etienne.fr>

<b.colombier@univ-st-etienne.fr>

télécom saint-étienne
école d'ingénieurs nouvelles technologies

INSTITUT UNIVERSITAIRE DE TECHNOLOGIE SAINT-ÉTIENNE

Université Jean Monnet, Saint-Etienne

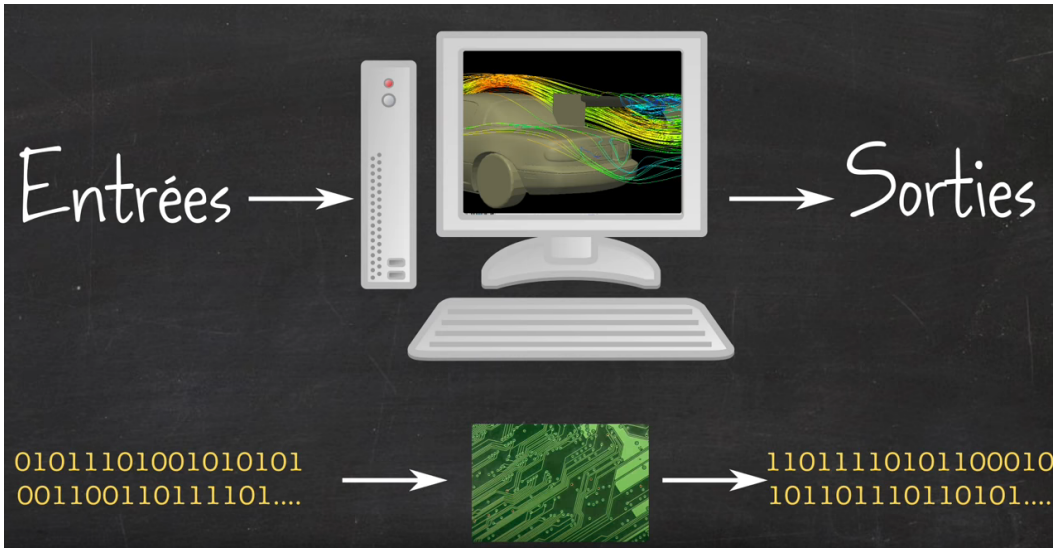
CiTiSE1

Année 2025 - 2026

Sommaire

L'informatique en GEII et les matières associées

- Ordinateur (au sens large [1]) manipule de l'**information binaire**



- Informations de types différents (Nombres, films, musiques, fichiers textes, navigation internet, jeux,...) $\longrightarrow$ Information binaire

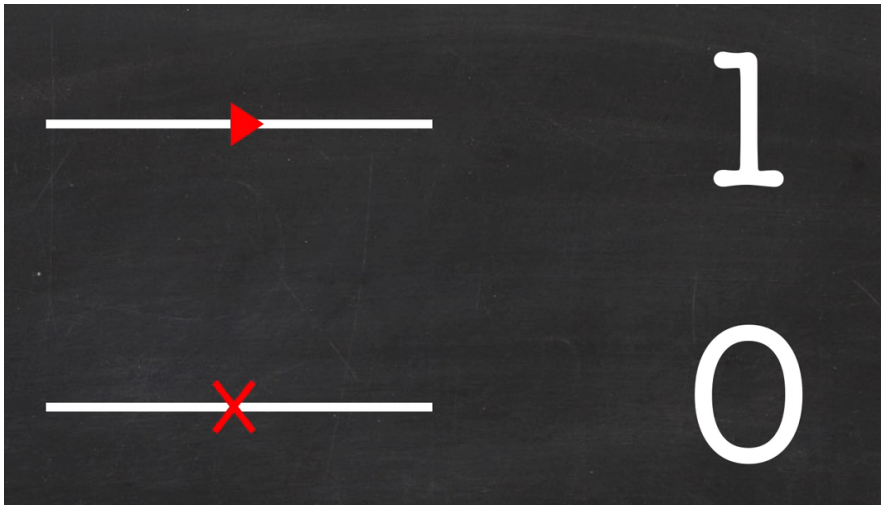
Codage de l'information

Notion de **codage de l'information** qui sera abordée (en particulier) en Auto1 (Ex SIN1), Info1, Info2.

1. on parlera de systèmes à microprocesseurs

L'informatique en GEII et les matières associées

- Information binaire est un ensemble de 0 et de 1, utilisée dans les ordinateurs car facile à représenter d'un point de vue électrique



- Opérateurs de base manipulant cette information : **les portes logiques** (Auto1 (Ex SIN1))
- Portes logiques fabriquées à l'aide de **transistors** (cours de SAé (Ex R1 et ER1) ainsi que Elen1 (Ex SE1))

L'informatique en GEII et les matières associées

- Le cours d'Info1 suppose que le système informatique [2] existe et se concentre sur sa **programmation**.
- Le cours d'Auto1 (Ex SIN1) suppose que les portes logiques existent et s'intéresse à la **conception/réalisation** de systèmes logiques (c'est à dire de système manipulant de l'information binaire) comme un microprocesseur par exemple.
- Les cours de SAé (Ex R1 et ER1) supposent que le transistor existe et proposent la **conception/réalisation** de circuits électroniques analogiques à partir de ces transitors (en particulier la réalisation des portes logiques).
- Le cours d'Elen1 (Ex SE1) étudie les **lois fondamentales de l'électronique** nécessaires à la réalisation de systèmes électroniques.

Chaîne de connaissance

Toutes ces matières **complémentaires** sont donc très importantes pour avoir une compréhension fine du système que vous allez au final programmer.

2. L'architecture d'un tel système sera étudiée au S2 via le cours d'info2

Organisation

- Objectifs :
  1. Etre capable de proposer un découpage fonctionnel (ensemble d'algorithmes) répondant à un problème donné
  2. Etre capable de traduire ces algorithmes en langage C et de détecter/corriger les erreurs de programmation
  3. Etre capable de proposer un ensemble de tests permettant de s'assurer que le programme répond au cahier des charges donné.
- Méthodes de travail :
  - Brouillon + crayon papier + gomme
  - Essayer = progresser
  - Se tromper = progresser
  - Poser des questions pendant les séances (cours/TD) = progresser
  - Faire des exercices chez soi = progresser
  - Consulter, en autonomie, des ressources complémentaires (livres, sites internet) = progresser
- Contact : <florent.bernard@univ-st-etienne.fr>
  - Site Manufacture, Laboratoire Hubert Curien, Bureau F140
  - Site Métare, IUT département GEII, Bureau E113

Déroulement du cours : CM

- CM (∼1h/semaine) : 14 séances en présentiel, classe entière (amphi) : F. Bernard

Préparation des séances de CM

- Vidéo de cours à regarder chez vous en dehors des heures de cours **ET** impérativement avant l'amphi concerné.
- Exercices [Ch] du fascicule d'exercices sont des exercices d'application **directe** du cours et **ne seront pas traités en TD**. Ils sont **obligatoirement** à travailler[a] en autonomie chez soi puisque une correction tapée est fournie dans le fascicule d'exercices.
- Séance de Questions/réponses en début de séance (sur la vidéo/exercices Ch)
- Séance de QCM (10-15min) en fin d'amphi : évaluation en présentiel (notée).
  Objectif : vérifier que les notions sont **travaillées** et **comprises**.

---

a. Travailler = chercher dans un premier temps sans l'aide du cours, puis avec l'aide du cours si blocage, puis à l'aide de la correction tapée si blocage periste.
Travailler ≠ lire la correction tapée

Déroulement du cours : TD

- 28h TD : 16 séances d'1h30 et 2 séances de 2h
  - IUTA : B. Colombier
  - IUTB : F. Bernard
  - IUTC : F. Bernard

Précision sur les TDs

- Le TD reprend des points importants du cours et propose de les travailler à travers des exercices à chercher en séance ($\sim$ 1h30) puis d'une séance sur l'autre.
- Possibilité d'apporter son propre ordinateur portable avec les outils de développement **installés et fonctionnels** et maîtrisés par l'étudiant.
- Un TD est **toujours** (sauf mention explicite du contraire) à terminer d'une séance sur l'autre sur une feuille séparée sur laquelle est indiquée votre nom et votre groupe de TD. Cette feuille est susceptible d'être ramassée pour évaluation.

BUT GEII : Compétences visées et apprentissages critiques

Compétences

Concevoir Vérifier

Apprentissages critiques	
Niveau 1 de la compétence 1	Niveau 1 de la compétence 2
➢ Produire une analyse fonctionnelle d'un système (**C1-N1-AC1**) ➢ Réaliser un prototype pour des solutions techniques matériel et/ou logiciel (**C1-N1-AC2**)	➢ Appliquer une procédure d'essai (**C2-N1-AC1**) ➢ Identifier un dysfonctionnement (**C2-N1-AC2**) ➢ Décrire les effets d'un dysfonctionnement (**C2-N1-AC3**)

Evaluation

CP 1 examen à mi-parcours (avant les vacances de la Toussaint)

CP 1 examen terminal (fin décembre/début janvier)

CC des QCMs réguliers et notés en amphis

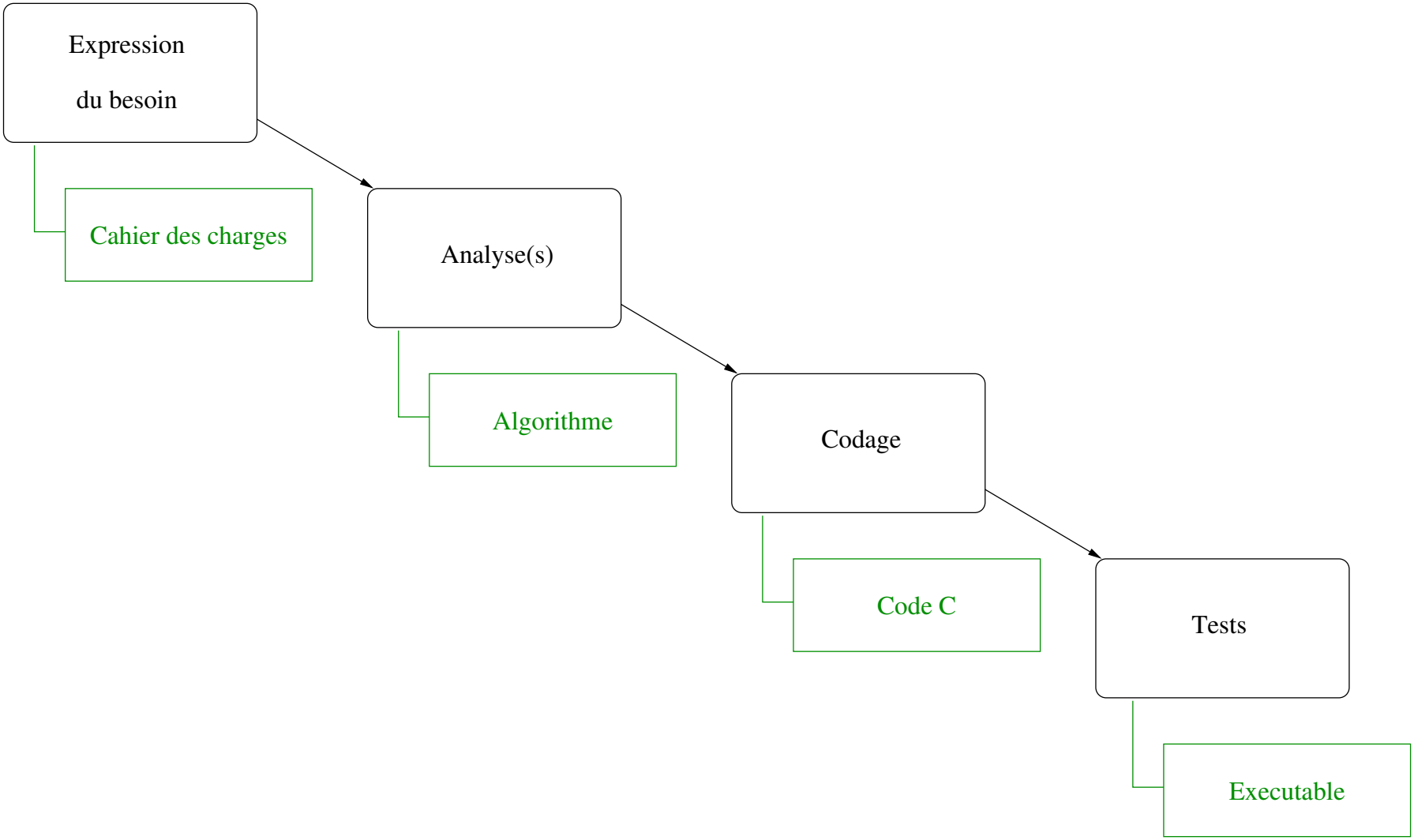
CC des séances de TD récupérées et notées.

CC des séances faites sur machine dont le code C sera à déposer sur l'ENT (Moodle) pour correction et évaluation. [3]

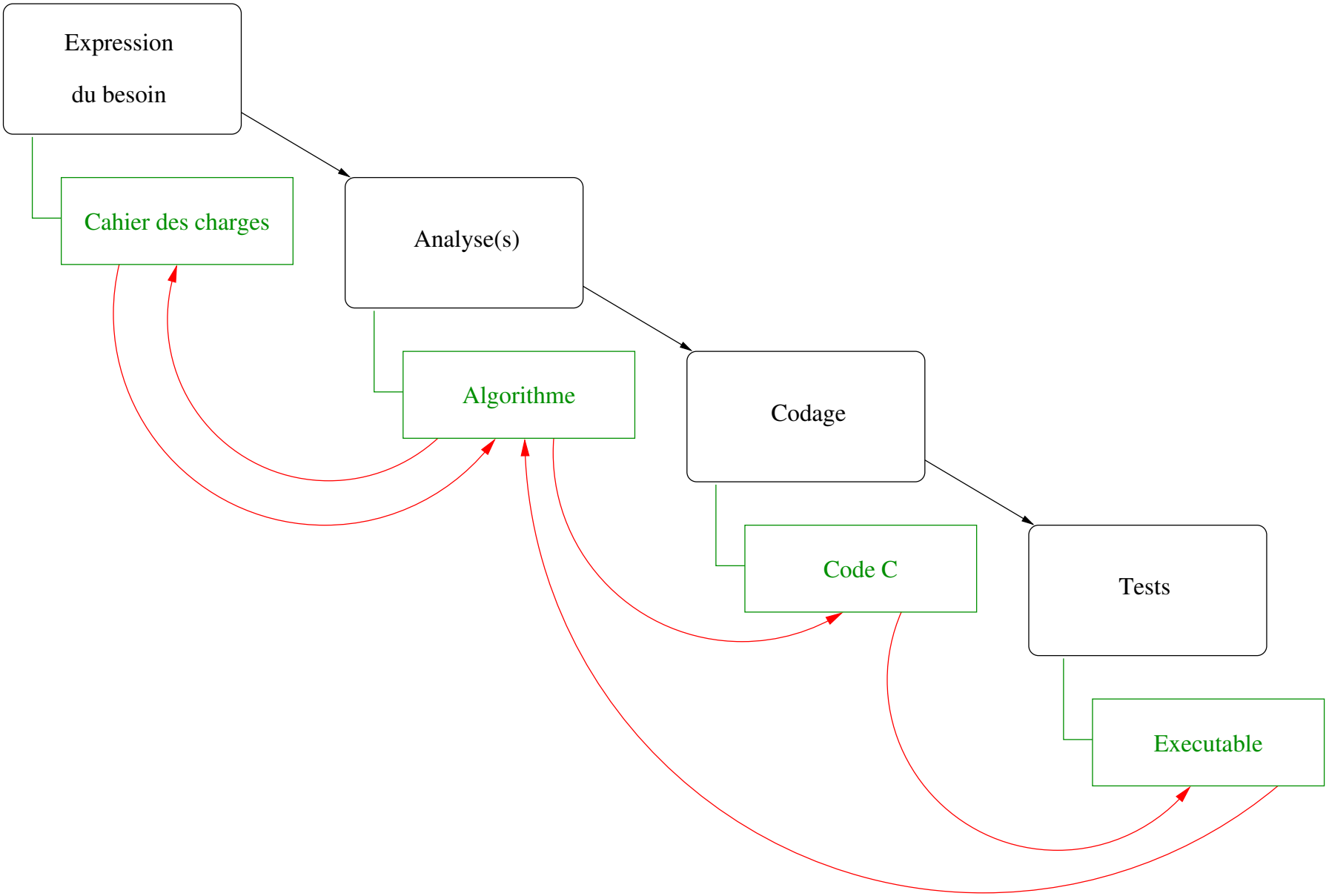
P Un projet info ? (en cours de discussion)

3. CP : contrôle partiel, CC : contrôle continu, P : Projet

Du besoin à la solution



Du besoin à la solution



Objectif final

- Obtenir de la machine qu'elle effectue un travail à notre place
- Problème : expliquer à la machine comment elle doit s'y prendre
  - Comment le lui dire ?
  - Comment lui apprendre à résoudre le problème ?
  - Comment être sûr qu'elle fera ce travail aussi bien que nous ?
  - . . .mieux que nous ?

Exemple

- Enoncé : Préparation d'un gateau de semoule
  A quel besoin cela répond ? Comment s'y prendre pour satisfaire le besoin ?
- Une solution :
  - Ingrédients :
    - Lait (1L)
    - Sucre en poudre (100g)
    - Semoule (125g)
  - Préparation :
    - Faire bouillir le lait avec le sucre
    - Verser la semoule
    - Laisser cuire jusqu'à épaississement (5-10 min)
    - Lorsque la préparation se détache de la casserole, enlever du feu
    - Verser dans un plat en pyrex
    - Laisser refroidir jusqu'à température ambiante
    - Placer 1h00 au frigo.
- Limitations : différences Homme-Machine ("faire bouillir", "jusqu'à épaississement", "se détache", ingrédients manquants : thermomètre, chronomètre, casserole, gazinière/plaques électriques, plat en pyrex, frigo)

Exercice : rigueur et précision

- L'Homme peut combler le manque d'information par son intelligence, son expérience et son bon sens (grâce à son cerveau)
- Remarque très importante : la machine est dépourvue de cerveau !

Dessine moi un carré [TD]

En suivant le modèle précédent (ingrédients+préparation), mais en étant beaucoup plus précis, proposez une suite d'actions à effectuer dans un ordre donné pour permettre à une personne quelconque de dessiner un carré.

Sommaire

Intuition d'algorithme

- Chapitre précédent : première intuition d'algorithme

Intuition

$$\text{Ingrédients} \xrightarrow{\text{suite d'actions}} \text{Produit fini}$$

- La notion d'algorithme ne vous est pas étrangère :

$$\text{Briques de légo} \xrightarrow{\text{Suite de dessins}} \text{Camion de pompiers}$$

$$\text{Meuble en kit} \xrightarrow{\text{Notice de montage}} \text{Etagère}$$

$$\underbrace{\text{Farine, oeufs, chocolat, etc...}}_{\text{Données d'entrée}} \quad \underbrace{\xrightarrow{\text{Recette}}}_{\text{Algorithme}} \quad \underbrace{\text{Forêt noire}}_{\text{Sortie/Résultat}}$$

Définition

Algorithme

Un *algorithme* est une **suite d'instructions** à appliquer dans un **ordre** spécifique, à un nombre **fini de données**, pour obtenir un **résultat**.

Remarque

L'algorithme doit être défini $\underbrace{\text{le plus précisément possible}}_{\text{précision}}$ et

$\underbrace{\text{englober tous les cas particuliers}}_{\text{rigueur}}$

Notes importantes

Note 1

Si les résultats escomptés ne sont pas obtenus, c'est le plus souvent **la conception de l'algorithme** qui est à remettre en question et non pas la machine.

Note 2

Un algorithme doit être compréhensible par n'importe quelle autre personne et doit donc être **indépendant** d'un langage de programmation. Cependant, afin d'être rigoureux et de ne pas laisser de place à une mauvaise interprétation, il est courant d'adopter certaines règles (avec un peu de souplesse)
$\Rightarrow$ Nécessité d'un certain **formalisme**.

Canevas général d'un algorithme

Dans ce module Info1, un algorithme devra se présenter sous la forme suivante :

Canevas

Lexique

{**Déclaration** des variables, constantes, fonctions utilisées}

Algorithme

Début

{Liste d'instructions permettant la résolution du problème posé}

Fin

Commentaires

Il est important et **demandé** de commenter les algorithmes pour améliorer leur lisibilité et leur compréhension. Les commentaires ne sont pas des instructions à proprement parler.
En algorithmie, les commentaires s'écriront entre **accolades**.

Présentation du lexique

Définition

Le *lexique* contient la **déclaration** des outils nécessaires à la résolution d'un problème. Parmi ces outils on distinguera :

- **Les variables** : Elles associent un nom à une valeur d'un type donné et peuvent changer au cours de l'algorithme.
  Rôle : stocker des informations (type à préciser) utilisées au cours d'un algorithme et qui peuvent changer.
- **Les constantes** : Elles associent un nom à une valeur **fixée une fois pour toute** d'un type donné.
  Rôle : stocker une information une fois pour toute (ex : taux de conversion, constante mathématique, ...). **Intérêt ?**
- **Les fonctions** : Ce sont d'autres algorithmes, existants ou que vous avez créés (avec des variables, des constantes, des fonctions, ...) disposant de données d'entrées et retournant une donnée en sortie.
  Rôle : réutilisation de solutions existantes dans un nouvel algorithme.
  Notion de **découpage fonctionnel** étudié dans le chapitre 3.

Types d'information

- Rappel : il y a deux **classes** d'information (variable et constante) qui peuvent stocker une information d'un certain **type**.
- Les différents **types** d'information en question sont :
  - **les entiers** (positifs ou négatifs : $\mathbb{Z}$) : par exemple -3, 4, 0, -20, ...
  - **les réels** ($\mathbb{R}$) : par exemple 3.45, -5.87, 1.0, ...
  - **les caractères** : par exemple 'a', 'z', 'e', 'r', 't', 'y', '1', '0', '9', ...(cf : table ASCII)
  - **les chaînes de caractères** (qui seront en fait des tableaux de caractères) : "toto", "saisissez une valeur", ...
  - **les booléens** : pouvant prendre les valeurs TRUE ou FALSE (VRAI ou FAUX). Ils sont très utilisés dans les tests. Parallèle à faire avec le cours de SIN1 et de Maths (Logique).
  - **les adresses** : une variable contenant une adresse (on appelle cela *un pointeur*) permet de repérer à quel emplacement mémoire une valeur est stockée. L'utilisation des pointeurs (massive en POO (S3)) sera abordée ultérieurement (dans le cours d'Info2 au S2).

Cohérence des types (ou **respect** des types)

Une variable ou une constante d'un certain type peut stocker seulement une valeur du **même type**.

Exemple

- Lexique pour le calcul de la surface d'un disque en fonction du rayon ($S = \pi R^2$)
- Deux **classes** de données interviennent dans cette formule :
  - une **variable** *rayon* qui sera saisie par l'utilisateur de la formule et qui doit être de **type réel**.
  - une **constante** *PI* qui a une valeur bien déterminée et fixée une fois pour toute et qui est un nombre **réel**.
- Les noms de variables et de constantes doivent être **pertinents et représentatifs** de l'information qui est mémorisée.
- Le lexique sera de la forme :

  Lexique :

  PI : la constante réelle := 3.14159265

  rayon : un réel

Analyse de l'exemple

Convention

Par convention, pour différencier facilement les constantes et les variables dans un algorithme (et plus tard dans un code C), on notera les **constantes en MAJUSCULES** et les variables en minuscule).

L'objectif est d'améliorer la **lisibilité** et la **clarté** des algorithmes.

Initialisation

Dans la ligne *PI* : la constante réelle := 3.14159265, il faut remarquer le **:=**3.14159265, cette étape, appelée *Initialisation*, est indispensable quand on travaille avec des constantes [a].

a. Comme on ne peut pas modifier la valeur d'une constante par la suite, il convient de lui donner une valeur par défaut lors de la déclaration.

Recommandation très importante

Mise en garde : séparateurs

Attention en informatique (et donc en algorithmie) les virgules, les espaces ont un rôle bien spécial de séparation des données.
Aussi vous ferez attention à bien noter les nombres décimaux avec un point et pas une virgule.

Exemple

3,14159265 est vu en informatique comme deux entiers (3 et 14159265) séparés par une virgule.
Pour écrire le nombre décimal correspondant à une approximation de π, il faut écrire : 3.14159265

Conclusion

De manière générale, éviter d'utiliser des noms de variables ou de fonctions ou de fichiers comportant des caractères spéciaux, des accents, des espaces, des virgules.

Généralités

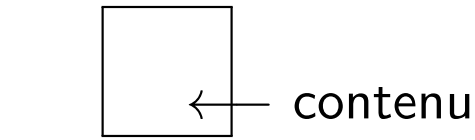
Jusqu'à présent nous avons introduit :

- deux **classes** d'information : constante et variable
- plusieurs **types** d'information s'appliquant aussi bien aux constantes qu'aux variables

Il faut à présent se doter d'**opérateurs de base** pour pouvoir effectuer des **traitements** sur ces variables.
Un algorithme basique sera donc une suite d'instructions utilisant des opérateurs de base.

L'affectation : ⟵

- Une variable d'un certain type permet de stocker une information du **même type**.
- Une variable est repérée/étiquetée par son nom et contient une information.



- Changer le contenu d'une variable est une opération de base qui s'appelle **l'affectation**.
- Syntaxe :
  `nom_variable` ⟵ `nouvelle_valeur`

Notes

1. L'ancien contenu de la variable est **perdu** après l'affectation !
2. Le **type** de `nom_variable` doit impérativement être le même que celui de `nouvelle_valeur`

Addition : +

- L'opérateur d'addition permet de faire l'addition entre le contenu de deux variables ou constantes
- Syntaxe : `nom_variable1 + nom_variable2`
- Cette opération retourne un résultat (la somme des contenus des deux variables).
- Ce résultat peut[4] être **affecté** à une troisième variable (appelons `somme` cette variable) :
  `somme`⟵`nom_variable1 + nom_variable2`

Types des opérandes et type du résultat de l'addition

1. Si les deux opérandes sont de type **entier** alors le résultat est de type **entier**.
2. Si une des deux opérandes (ou les deux) sont de type **réel** alors le résultat est de type **réel**.

4. et en pratique doit !

Soustraction : -

- L'opérateur de soustraction permet de faire la soustraction entre le contenu de deux variables ou constantes
- Syntaxe : `nom_variable1 - nom_variable2`
- Cette opération retourne un résultat (la différence des contenus des deux variables) qui peut éventuellement être négatif.
- Ce résultat peut[5] être **affecté** à une troisième variable (appelons `difference` cette variable) :
  `difference`⟵`nom_variable1 - nom_variable2`

Types des opérandes et type du résultat de la soustraction

1. Si les deux opérandes sont de type **entier** alors le résultat est de type **entier**.
2. Si une des deux opérandes (ou les deux) sont de type **réel** alors le résultat est de type **réel**.

5. et en pratique doit !

Multiplication : *

- L'opérateur de multiplication permet de faire la multiplication entre le contenu de deux variables ou constantes
- Syntaxe : `nom_variable1 * nom_variable2`
- Cette opération retourne un résultat (le produit des contenus des deux variables).
- Ce résultat peut[6] être **affecté** à une troisième variable (appelons `produit` cette variable) :
  `produit`⟵`nom_variable1 * nom_variable2`

Types des opérandes et type du résultat de la multiplication

1. Si les deux opérandes sont de type **entier** alors le résultat est de type **entier**.
2. Si une des deux opérandes (ou les deux) sont de type **réel** alors le résultat est de type **réel**.

6. et en pratique doit !

Division **réelle** : /

- L'opérateur de division **réelle** permet de faire la division du contenu d'une variable ou constante par celui d'une autre variable ou constante **non nulle** !
- Syntaxe : `nom_variable1 / nom_variable2`
- Cette opération retourne un résultat (la valeur de la fraction) qui est **nécessairement** de type **réel**.
- Ce résultat peut [7] être **affecté** à une troisième variable (appelons `frac` cette variable) :
  `frac`⟵`nom_variable1 / nom_variable2`
- Quel doit nécessairement être le type de `frac` ?

7. et en pratique doit !

Division euclidienne : opérateurs `div` et `reste`

Division euclidienne sur les entiers relatifs

Soient a et b deux entiers avec $b \neq 0$, on peut trouver un **unique** couple d'**entiers** (q, r) avec $r \geq 0$ tel que :

- $a = bq + r$
- $r < |b|$

L'entier q s'appelle le **quotient** et l'entier r le **reste** dans la division euclidienne de a par b.

- Avec les notations précédentes on peut définir les opérateurs `div` et `reste` qui opèrent sur des **entiers**.
  - Pour deux **entiers** a et b, on définit l'opération a `div` b qui a pour résultat l'entier q quotient de a par b.
  - Pour deux **entiers** a et b, on définit l'opération a `reste` b qui a pour résultat l'entier r reste de la division de a par b.

Opérateur `div`

- L'opérateur `div` permet d'obtenir le quotient dans la division du contenu d'une variable ou constante entière par celui d'une autre variable ou constante entière et **non nulle** !
- Syntaxe : `nom_variable1 div nom_variable2`
- Cette opération retourne un résultat de type **entier** (le quotient des contenus des deux variables).
- Ce résultat peut[8] être **affecté** à une troisième variable qui doit **nécessairement** être de type **entier** (appelons là `quotient`) : `quotient`$\leftarrow$`nom_variable1 div nom_variable2`

Exemple

- `43 div 7` doit retourner 6
- `1 div 2` doit retourner 0
- `5.7 div 2` n'est pas une opération acceptable car `5.7` n'est pas un entier.

8. et en pratique doit !

Opérateur `reste` (appelé aussi modulo)

- L'opérateur `reste` permet d'obtenir le reste dans la division du contenu d'une variable ou constante entière par celui d'une autre variable ou constante entière et **non nulle** !
- Syntaxe : `nom_variable1 reste nom_variable2`
- Cette opération retourne un résultat de type **entier** (le reste dans la division euclidienne des contenus des deux variables).
- Ce résultat peut [9] être **affecté** à une troisième variable qui doit **nécessairement** être de type **entier** (appelons là `res` [10]) : `res`⟵`nom_variable1 reste nom_variable2`

Exemple

- `43 reste 7` doit retourner 1 ($43=7\times6+1$)
- `1 reste 2` doit retourner 1 ($1=2\times0+1$)
- `5.7 reste 2` n'est pas une opération acceptable car `5.7` n'est pas un entier.

9. et en pratique doit !
10. Important : on ne peut pas appeler `reste` cette variable. `reste` est un nom **réservé** à l'opérateur qui permet d'obtenir le reste dans la division euclidienne

Priorité des opérateurs de base

- Les priorités sont les mêmes qu'en mathématiques sur les opérations d'addition/soustraction et multiplication/division.
- Comme en mathématique, on utilise des parenthèses pour "forcer" un calcul à être effectué en priorité ou pour lever une ambiguïté.
- L'affectation est toujours effectuée en dernier.

Exemple : une affectation

Si on effectue l'instruction suivante portant sur des variables a et b de type réel :
$a \leftarrow a * b + 3 * a - 1/(b * b + 1)$
Cette instruction est interprétée (en rajoutant des parenthèses superflues) comme :
$a \leftarrow (((a * b) + (3 * a)) - (1/((b * b) + 1)))$

- Comprendre la priorité des opérateurs est indispensable pour ne pas surcharger de parenthèses superflues vos instructions et ainsi améliorer la lisibilité du code.

Utilisateur et Concepteur : deux points de vue

1. Le concepteur de l'algorithme (ou le programmeur) qui apporte son intelligence à la machine (qui en est dépourvue).
2. L'utilisateur de la machine qui l'utilise pour résoudre son problème en faisant confiance à celui qui l'a programmée.

Communication Homme/Machine

Pour qu'un dialogue puisse avoir lieu entre la machine et l'utilisateur, il faut que le programmeur ait prévu des outils permettant à la machine de demander et stocker des informations à l'utilisateur et à l'utilisateur de visualiser des résultats.

On se place toujours du **point de vue du programmeur de la machine**. Il dispose de deux outils (on verra plus tard qu'il s'agit de fonctions) :

- `Lire()` dont le rôle est de récupérer une valeur saisie par l'utilisateur et de la stocker en mémoire.
- `Ecrire()` dont le rôle est d'afficher un message à l'utilisateur.

Exemple simple

- Saisie et affichage d'une valeur entrée par l'utilisateur

Lexique :

memo : un réel

Algorithme :

Début

Ecrire("Saisissez un réel")

Lire(*memo*) {stocke la valeur saisie par l'utilisateur dans la variable *memo*}

{C'est bien la machine qui **lit/récupère** la valeur saisie par l'utilisateur }

Ecrire("Vous avez saisi le réel ",*memo*)

{C'est bien la machine qui **écrit/affiche** à l'utilisateur la valeur stockée dans sa variable *memo*}

Fin

Un exemple complet

- Proposez un algorithme qui permet de demander à un utilisateur la valeur d'un rayon qui calcule et affiche la surface d'un disque en fonction du rayon ($S = \pi R^2$)
- Une solution :

Lexique :

PI : la constante réelle := 3.14159265
rayon, *surface* : deux réels

Algorithme :

Début
Ecrire("Saisissez une valeur de rayon (en cm) pour votre disque")
Lire(*rayon*) {stocke le rayon saisi dans la variable *rayon*}
surface ⟵ *PI*∗*rayon*∗*rayon* {Seule instruction de **Traitement**}
Ecrire("La surface du disque de rayon ",*rayon*,"cm est de ",*surface*,"cm^2")
Fin

Autres opérateurs

Il est possible d'utiliser en algorithmie d'autres opérateurs mathématiques (avec la notation mathématique), comme les fonctions racine (carrée, cubique, etc...), puissance, trigonométriques, etc...

- racine carrée : $\sqrt{\cdot}$
- racine cubique : $\sqrt[3]{\cdot}$
- fonction puissance : $(\cdot)^{\alpha}$ où $\alpha \in \mathbb{R}$
- fonctions trigonométriques : $\cos(\cdot)$, $\sin(\cdot)$, $\tan(\cdot)$
- ...

Attention

Lorsqu'on traduira ces opérateurs mathématiques dans le langage qui nous intéresse, il y aura des précautions à prendre :

- Vérifier que les opérateurs mathématiques sont bien définis dans le langage
- Être très rigoureux sur la syntaxe (traduction de l'algorithmie au langage)

Un exemple complet (suite)

- En utilisant l'opérateur d'élévation au carré, on peut écrire :

Lexique :

PI : la constante réelle := 3.14159265
rayon, *surface* : deux réels

Algorithme :

Début
Ecrire("Saisissez une valeur de rayon (en cm) pour votre disque")
Lire(*rayon*)
surface $\longleftarrow$ $PI*rayon^2$ {au lieu de $PI*rayon*rayon$}
Ecrire("La surface du disque de rayon ",*rayon*,"cm est de ",*surface*,"cm^2")
Fin

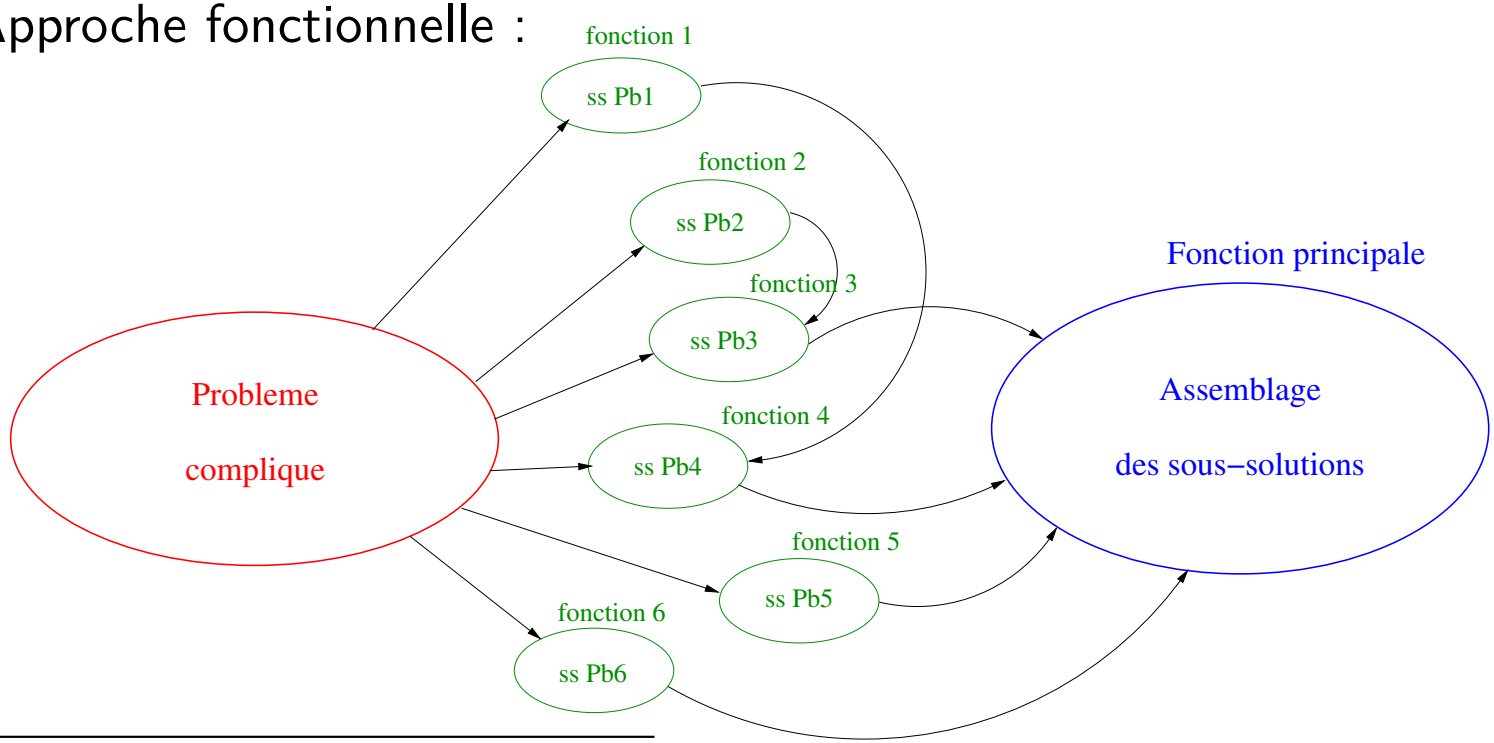
Problèmes/limitations

- Si on veut réutiliser cet algorithme pour calculer le volume d'un cylindre, on aimerait juste pouvoir calculer la surface de sa base sans devoir forcément l'afficher...
- On ne veut pas non plus avoir à copier/coller systématiquement la partie du code permettant de calculer la surface. Pourquoi ?

Sommaire

Motivations

- Besoin de structurer/découper des algorithmes complexes en sous-algorithmes plus simples [11].
- Besoin d'associer une tâche (fonctionnalité) spécifique à un algorithme simple et efficace.
- Réutilisabilité de fonctionnalités existantes ou développées précédemment par le programmeur lui même.
- Approche fonctionnelle :



11. approche : "diviser pour régner"

Qu'est ce qu'une fonction ?

- Visualisation graphique :

- Analogie avec les maths.

$$\begin{array}{rccc} f: & \mathbb{R} & \longrightarrow & \mathbb{R} \\ & x & \longmapsto & 2x+3 \end{array}$$

Les informations importantes sont :

- **Nom** de la fonction : f
- Domaine de définition de la fonction (ici $\mathbb{R}$) : c'est le **type du (ou des) paramètre(s) d'entrée**
- Image de la fonction (ici $\mathbb{R}$) : c'est le **type de l'élément retourné**
- **Définition générale** (en fonction d'un paramètre qu'on peut noter x ici) de la fonction : $f(x) = 2x + 3$
- **Utilisation(s) particulière(s)** :
  - pour $x = 5$, $f(x)$ **retourne** la valeur $13 (= 2 * 5 + 3)$
  - pour $x = -1.5$, $f(x)$ **retourne** la valeur $0 (= 2 * (-1.5) + 3)$
  - ...

Définition : Fonction

Définition

En informatique, une fonction est un algorithme particulier agissant sur un **nombre fini** de **paramètres** d'entrée et **retournant, au plus, un seul** type d'information [a].

a. parler de paramètre de sortie n'a pas de sens !

Bonnes pratiques

- Avant de créer une fonction, il faudra préciser (dans un commentaire) le **rôle** de la fonction ainsi que ses **paramètres d'entrée** et son **type de retour** ;
- Donner un **nom explicite** à la fonction par rapport à son rôle
- Ce nom commencera par une **majuscule**
  Exemple : `Surface` désignera une fonction tandis que `surface` représentera une variable.

Structure d'un algorithme comportant des fonctions

Pour résoudre un problème suivant **une approche fonctionnelle**, il faut :

- un algorithme (Lexique/algorithme) pour **définir** chaque fonction.
- un algorithme principal (Lexique/algorithme) permettant de **déclarer** (Lexique) et d'**utiliser/appeler** (Algorithme) les fonctions pour résoudre le problème posé.

Terminologie : local vs principal

L'algorithme permettant de décrire une fonction aura un lexique dit **local** et un algorithme dit **local** à la fonction.
Tout ce qui est déclaré dans le lexique local **n'est connu et utilisable que par la fonction** et est donc **inconnu** du lexique principal.

L'algorithme permettant d'**utiliser/appeler** toutes ou une partie de ces fonctions sera dit **principal**.
Le lexique de l'algorithme principal sera lui aussi qualifié de **principal** et contiendra les **déclarations** [a] des fonctions ainsi que les déclarations des **variables/constantes** utilisées dans l'algorithme principal.

Les **définitions** des fonctions seront faites en dehors du lexique principal et algorithme principal et peuvent donc être utilisées dans n'importe quel autre résolution de problème [b].

---

a. cf : slide suivant
b. ce qui donnera lieu à un nouveau programme qui aura un et un seul algorithme principal

Des maths à l'algorithmie

Maths	Infos importantes	Traduction algorithmique
f : $\mathbb{R} \longrightarrow \mathbb{R}$	Nom de la fonction Entrée de type réel Sortie de type réel	Lexique :{principal} {**Déclaration = Prototype** de la fonction f} {Ne pas oublier Rôle, Entrée, Sortie} f : la fonction (x : 1 réel)→1 réel {Déclarations d'autres variables/constantes} xp, resultat : 2 réels Algorithme :{principal} Début Ecrire("Saisir un réel") Lire(xp) resultat←—f(xp) {**utilisation** de la fonction} Ecrire("f(",xp,")=",resultat) Fin
$x \longmapsto 2x + 3$	Nom du paramètre d'entrée : x Définition $f(x) = 2x + 3$	{**Définition** de la fonction f} f : la fonction (x : 1 réel)→1 réel Lexique :{Local à f} res : 1 réel Algorithme :{Local à f} Début res←—2*x + 3 **Retourner** res Fin

Résumé

- Une fonction est dédiée à la réalisation d'un problème donné : calculs, conversions, affichage, saisie...
- Elle contient une suite d'instructions permettant de résoudre ce problème
- C'est donc un algorithme particulier qui peut contenir toutes les instructions et structures [12] qu'on étudiera dans les chapitres suivants.
- 3 **étapes** sont nécessaires pour l'utilisation d'une fonction :
  1. **Déclaration ou prototype** : dans le lexique principal
  2. **Appel** de la fonction : dans l'algorithme principal.
  3. **Définition** : en dehors du lexique principal et de l'algorithme principal

Important : concernant la définition d'une fonction

Pour construire le résultat retourné par une fonction, on se sert :

- de ses **paramètres d'entrée** ;
- des **variables/constantes** déclarées dans son lexique **local** et connues **uniquement** dans cette fonction

Remarque importante

Une fonction peut avoir **plusieurs paramètres d'entrée**, mais TOUJOURS **un seul** type de retour

12. `Si...Sinon...FinSi`, `Tant que...FinTantque`, etc...

Fonction : $x \rightarrow D\_atan(x) = \frac{1}{1+x^2}$ où x est un réel

- Déclarer, Définir et Utiliser *D_atan*

Lexique :{principal}
{Ne pas oublier Rôle, Entrée, Sortie}
D_atan : la fonction (x : 1 réel)→ 1 réel
xx : 1 réel
result : 1 réel

Algorithme :{principal}
Début
    Ecrire("Entrer 1 réel")
    Lire(xx)
    result←D_atan(xx) {Appel de la fonction}
    Ecrire("D_atan(",xx,")=",result)
Fin

{Définition, en dehors du lexique principal et de l'algorithme principal}
D_atan : la fonction (x : 1 réel)→ 1 réel
    Lexique :{Local à D_atan}
      res : 1 réel
    Algorithme :{Local à D_atan}
    Début
      res←1/(x*x+1)
      Retourner res
    Fin

Fonction : $t \rightarrow Cel2Far(t)$ où t est un réel

- Fonction qui prend une température en degrés Celsius et la convertit en degrés Fahrenheit [13]
- Déclarer, Définir et Utiliser *Cel2Far*

Lexique :{principal}
{Ne pas oublier Rôle, Entrées, Sortie}
`Cel2Far` : la fonction (t : 1 réel)→ 1 réel
cel : 1 réel
t_far : 1 réel
Algorithme :{principal}
Début
  Ecrire("Entrer 1 température en °C")
  Lire(cel)
  t_far←`Cel2Far`(cel) {Appel fonction}
  Ecrire(cel," °C = ", t_far," °F")
Fin

{Définition, en dehors du lexique principal et de l'algorithme principal}
`Cel2Far` : la fonction (t : 1 réel)→ 1 réel
  Lexique :{Local à `Cel2Far`}
    far : 1 réel
  Algorithme :{Local à `Cel2Far`}
  Début
    far←9/5*t+32 {Attention à la formule !}
    Retourner far
  Fin

13. Rappel : les degrés Celsius sont obtenus à partir des degrés Fahrenheit en ôtant 32 à ces derniers puis en multipliant par $\frac{5}{9}$.

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

Algorithme :{principal}
Début

Fin

Lexique :{Local à g}

Algorithme :{Local à g}
Début

Fin

Memoire utilisee par l'algorithme principal

L'algorithme principal n'a pas acces directement au contenu de x, y et z

Memoire utilisee par la fonction

La fonction n'a pas acces directement au contenu de xx, yy et zz

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}
g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Algorithme :{principal}
Début

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

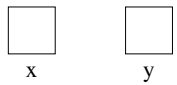
Algorithme :{Local à g}
Début

Fin

Memoire utilisee par l'algorithme principal

L'algorithme principal n'a pas acces directement au contenu de x, y et z

Memoire utilisee par la fonction



La fonction n'a pas acces directement au contenu de xx, yy et zz

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}
g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Algorithme :{principal}
Début

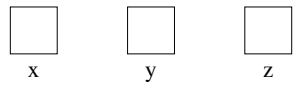
Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier
| Lexique :{Local à g}
| z : 1 entier
| Algorithme :{Local à g}
| Début
|
| Fin

Memoire utilisee par l'algorithme principal

L'algorithme principal n'a pas acces directement au contenu de x, y et z

Memoire utilisee par la fonction



La fonction n'a pas acces directement au contenu de xx, yy et zz

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}
g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Algorithme :{principal}
Début

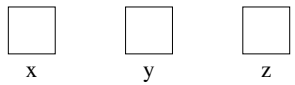
Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier
Lexique :{Local à g}
z : 1 entier
Algorithme :{Local à g}
Début
z←x*x-y
Retourner z
Fin

Memoire utilisee par l'algorithme principal

L'algorithme principal n'a pas acces directement au contenu de x, y et z

Memoire utilisee par la fonction



La fonction n'a pas acces directement au contenu de xx, yy et zz

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

Algorithme :{principal}

Début

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

Algorithme :{Local à g}

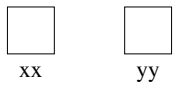
Début

z←x*x-y

Retourner z

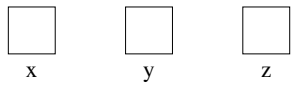
Fin

Memoire utilisee par l'algorithme principal



L'algorithme principal n'a pas acces directement au contenu de x, y et z

Memoire utilisee par la fonction



La fonction n'a pas acces directement au contenu de xx, yy et zz

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

zz : 1 entier

Algorithme :{principal}

Début

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

Algorithme :{Local à g}

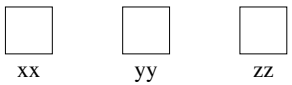
Début

z←x*x-y

Retourner z

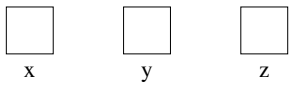
Fin

Memoire utilisee par l'algorithme principal



L'algorithme principal n'a pas acces directement au contenu de x, y et z

Memoire utilisee par la fonction



La fonction n'a pas acces directement au contenu de xx, yy et zz

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

zz : 1 entier

Algorithme :{principal}

Début

Ecrire("Entrer 2 entiers")

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

Algorithme :{Local à g}

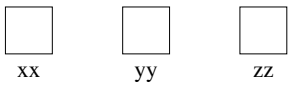
Début

z←x*x-y

Retourner z

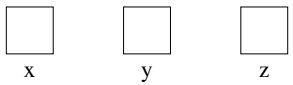
Fin

Memoire utilisee par l'algorithme principal



L'algorithme principal n'a pas acces directement au contenu de x, y et z

Memoire utilisee par la fonction



La fonction n'a pas acces directement au contenu de xx, yy et zz

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

zz : 1 entier

Algorithme :{principal}

Début

Ecrire("Entrer 2 entiers")

Lire(xx,yy)

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

Algorithme :{Local à g}

Début

z←x*x-y

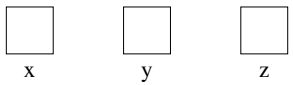
Retourner z

Fin

Memoire utilisee par l'algorithme principal

L'algorithme principal n'a pas acces directement au contenu de x, y et z

Memoire utilisee par la fonction



La fonction n'a pas acces directement au contenu de xx, yy et zz

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

zz : 1 entier

Algorithme :{principal}

Début

Ecrire("Entrer 2 entiers")

Lire(xx,yy)

g(xx,yy) {Appel de la fonction}

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

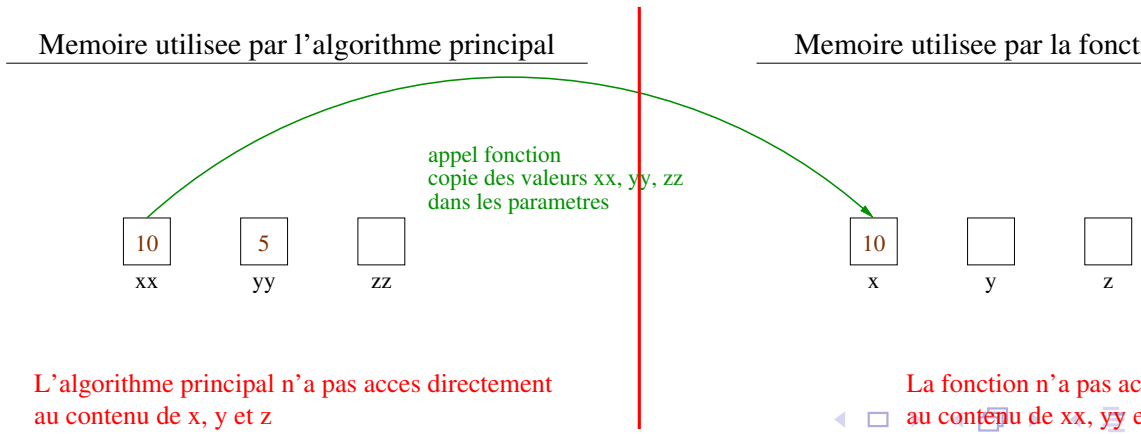
Algorithme :{Local à g}

Début

z←x*x-y

Retourner z

Fin



Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

zz : 1 entier

Algorithme :{principal}

Début

Ecrire("Entrer 2 entiers")

Lire(xx,yy)

g(xx,yy) {Appel de la fonction}

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

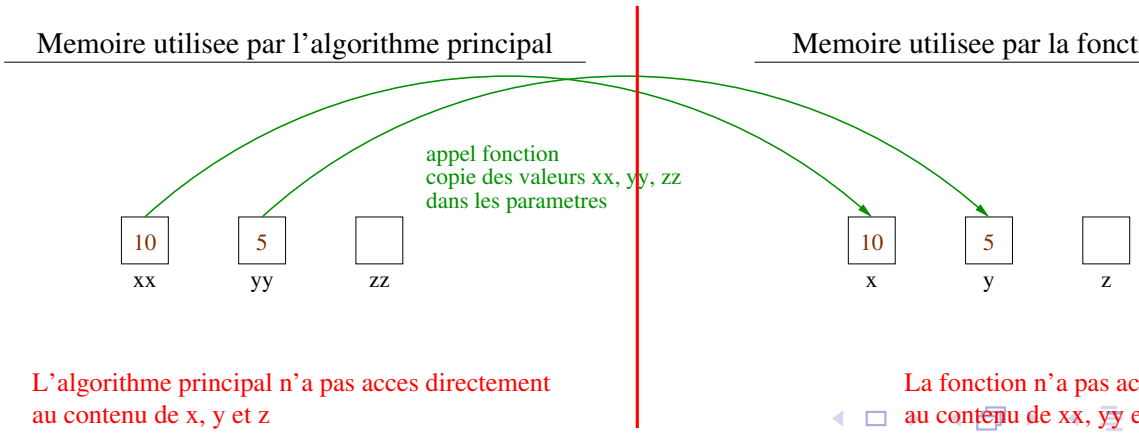
Algorithme :{Local à g}

Début

z←x*x-y

Retourner z

Fin



Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

zz : 1 entier

Algorithme :{principal}

Début

Ecrire("Entrer 2 entiers")

Lire(xx,yy)

g(xx,yy) {Appel de la fonction}

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

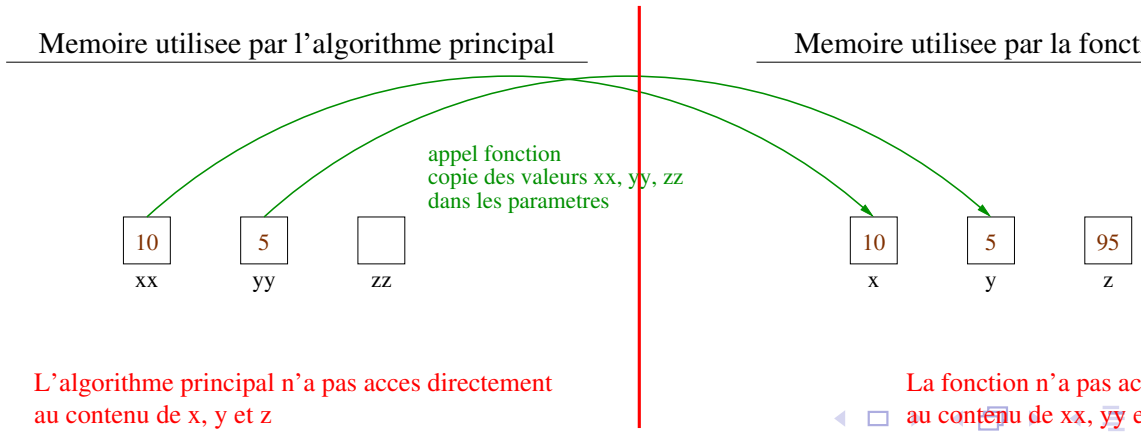
Algorithme :{Local à g}

Début

z←x*x-y

Retourner z

Fin



Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

zz : 1 entier

Algorithme :{principal}

Début

Ecrire("Entrer 2 entiers")

Lire(xx,yy)

g(xx,yy) {Appel de la fonction}

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

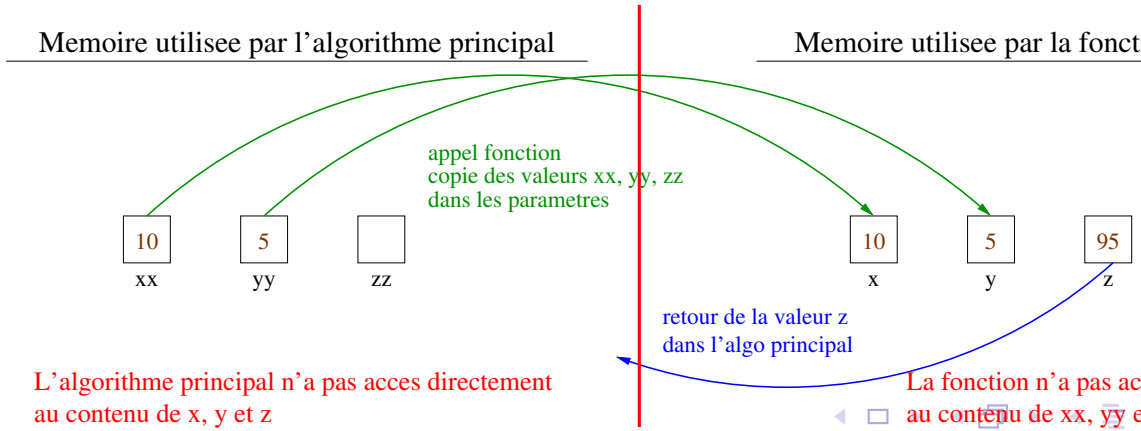
Algorithme :{Local à g}

Début

z←x*x-y

Retourner z

Fin



Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

zz : 1 entier

Algorithme :{principal}

Début

Ecrire("Entrer 2 entiers")

Lire(xx,yy)

zz←g(xx,yy) {Appel de la fonction}

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

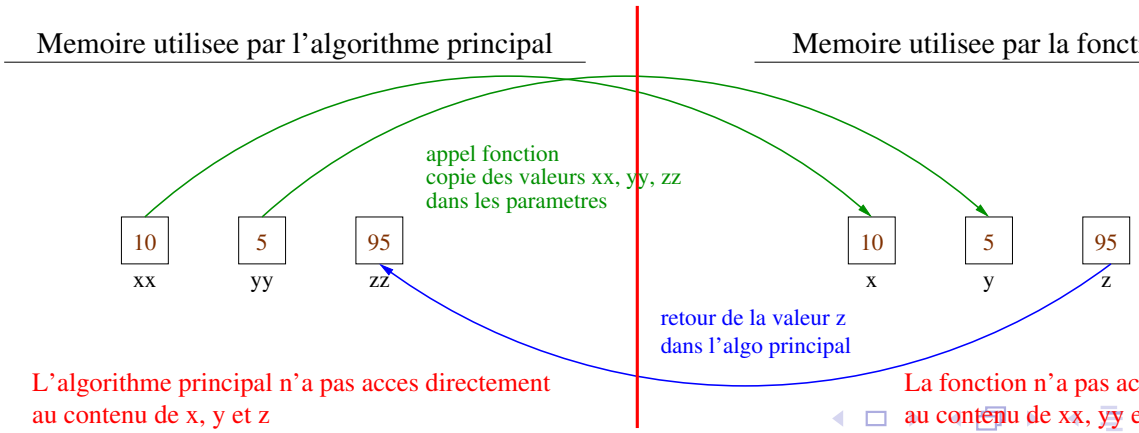
Algorithme :{Local à g}

Début

z←x*x-y

Retourner z

Fin



Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et $y \in \mathbb{Z}$. Déclarer, Définir et Utiliser g

Lexique :{principal}

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

xx,yy : 2 entiers

zz : 1 entier

Algorithme :{principal}

Début

Ecrire("Entrer 2 entiers")

Lire(xx,yy)

zz←g(xx,yy) {Appel de la fonction}

Ecrire("g(",xx,",",yy,")=",zz)

Fin

g : la fonction (x : 1 entier, y : 1 entier)→ 1 entier

Lexique :{Local à g}

z : 1 entier

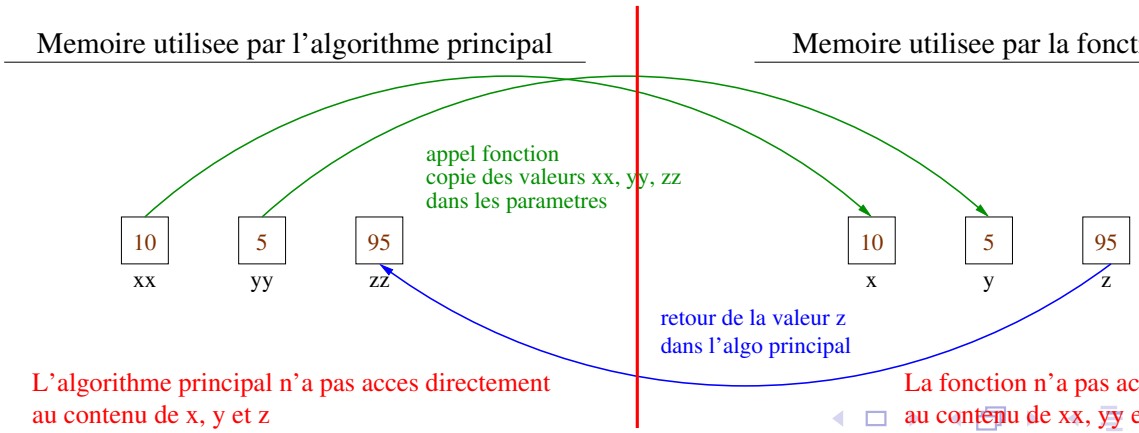
Algorithme :{Local à g}

Début

z←x*x-y

Retourner z

Fin



Une fonction qui ne renvoie rien

- C'est notamment le cas des fonctions d'affichage. Par exemple :

Lexique :{principal}
`Affiche_note` : la fonction (note : 1 réel, bareme : 1 réel)→ **vide**
eton,emerab : 2 réel
Algorithme :{principal}
Début
  Ecrire("Entrer votre note puis le bareme")
  Lire(eton,emerab)
  `Affiche_note`(eton,emerab) {Appel de la fonction}
Fin

`Affiche_note` : la fonction (note : 1 réel, bareme : 1 réel)→ **vide**
  Lexique :{Local à `Affiche_note`}
    vide
  Algorithme :{Local à `Affiche_note`}
  Début
    Ecrire("Vous avez ", note, "/",bareme)
  Fin
  {Noter l'absence de l'instruction **Retourner**}

Une fonction qui ne renvoie rien

- C'est notamment le cas des fonctions d'affichage. Par exemple :

Lexique :{principal}
`Affiche_note` : la fonction (note : 1 réel, bareme : 1 réel)→ **vide**

note,bareme : 2 réel ≠note,bareme $\begin{cases} \text{note,bareme} \in \text{lexique local} \\ \text{note,bareme} \in \text{lexique principal} \end{cases}$

Algorithme :{principal}
Début
  Ecrire("Entrer votre note puis le bareme")
  Lire(note,bareme)
  `Affiche_note`(note,bareme) {Appel de la fonction}
Fin

`Affiche_note` : la fonction (note : 1 réel, bareme : 1 réel)→ **vide**
  Lexique :{Local à `Affiche_note`}
    vide
  Algorithme :{Local à `Affiche_note`}
  Début
    Ecrire("Vous avez ", note, "/",bareme)
  Fin
  {Noter l'absence de l'instruction **Retourner**}

Une fonction sans paramètre

- C'est le cas des fonctions de saisie. Par exemple pour définir la fonction Saisir_note :

Saisir_Note : la fonction (**vide**)→ 1 réel

Lexique :{Local à Saisir_Note}

note : 1 réel

Algorithme :{Local à Saisir_Note}

```
Début
    Ecrire("Saisir une note : ")
    Lire(note)
    Retourner(note)
Fin
```

Une fonction qui ne renvoie rien et ne prend pas de paramètre

- Cas très particulier des fonctions qui permettent d'afficher des messages constants et qui n'attendent aucune saisie de l'utilisateur.
- Exemples :
  - Présentation d'un mode d'emploi pour un programme (pages de manuel par exemple)
  - Messages d'erreur suite à une entrée erronée (note non comprise entre 0 et 20 par exemple)
  - Message de Bienvenue

Exemple célèbre

Lexique :{principal}
`Hello` : la fonction (**vide**)→ **vide**

Algorithme :{principal}
Début
  `Hello()`
Fin

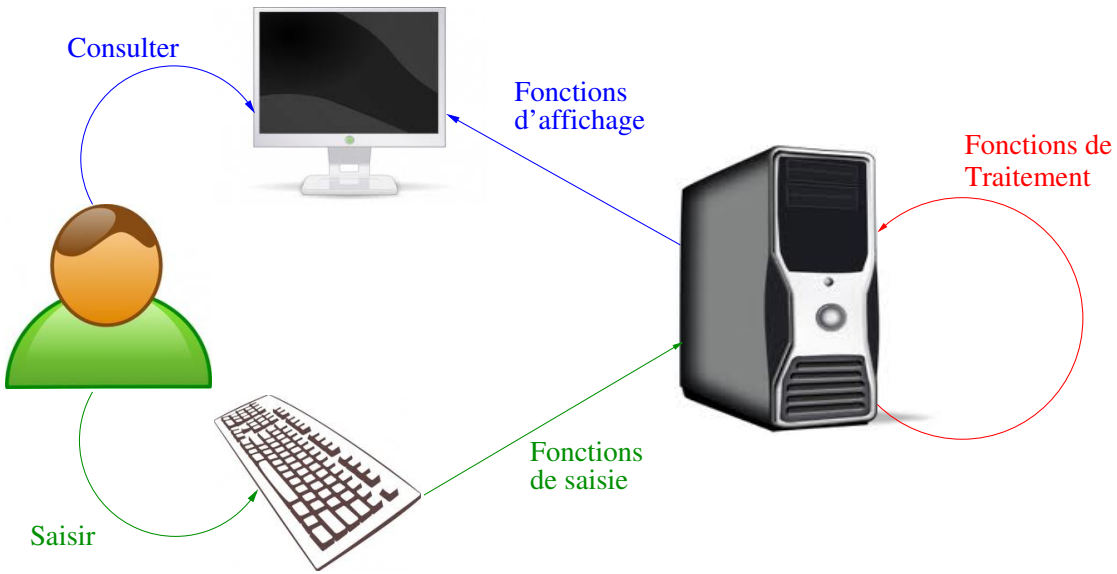
`Hello` : la fonction (**vide**)→ **vide**
  Lexique :{Local à `Hello`}
    vide
  Algorithme :{Local à `Hello`}
  Début
    Ecrire("Hello World !")
  Fin

Remarque importante

Notez bien la présence de `()` au niveau de l'appel à la fonction `Hello` même s'il n'y a pas de paramètre.
Une instruction `Hello` sans `()` engendrerait une erreur, puisque `Hello` désignerait une variable (et pas une fonction) qui n'aurait pas été déclarée dans le lexique.

Affichage/saisie et traitement

- Les fonctions d'**affichage** ont pour rôle d'afficher des informations à l'utilisateur : messages informatifs, contenu de variables
- Les fonctions de **saisie** permettent de demander à l'utilisateur de saisir des données qui seront stockées dans la machine sous forme de variables
- Les fonctions de **traitement** ont pour rôle d'effectuer des calculs avec un ou plusieurs paramètres d'entrée et de délivrer un résultat sous la forme d'un type de retour



Bilan

Observations

- On **sous-traite** le travail à effectuer à différentes fonctions.
  L'algorithme principal est **simple et facile** à lire.
  C'est l'intérêt du **découpage fonctionnel**, l'algorithme principal (chef d'orchestre du système) contient juste l'assemblage des appels aux différentes **fonctions de traitement** et permet de faire l'interface avec l'utilisateur par le biais des fonctions de **saisie et d'affichage**.
- Les noms des paramètres de la fonction ne servent que pour la **définition** de la fonction. Ils ne sont pas connus dans l'algorithme principal [a].
- Lors de l'**utilisation** de la fonction, on lui donne des valeurs de paramètres qui ont **le même type** que ceux utilisés pour la **déclaration** de la fonction.
- Une fonction qui ne renvoie rien est appelée une **procédure**.

a. Il est possible (mais dangereux au départ) de donner le même nom au paramètre d'une fonction et à une autre variable du lexique principal (cf : fonction `Bienvenue`)

Précision sur la notion de découpage fonctionnel

Dans les exercices, on vous demandera souvent de « proposer un découpage fonctionnel pour résoudre le problème . . . », il est important de comprendre ce qu'on attend par là pour que vous ne fassiez pas de hors sujet :

Travail attendu lors d'une demande de découpage fonctionnel

1. Déclarer (avec R,E,S) les fonctions qui seront utilisées pour résoudre le problème.
   Où ? Dans le Lexique principal
2. Compléter le lexique principal avec les variables utilisées dans l'algorithme principal, puis résoudre le problème en écrivant l'algorithme principal qui appellera les différentes fonctions du découpage.
3. Définir les fonctions du découpage qui ne seraient pas encore définies ou indiquer où on peut trouver la définition des autres fonctions (références vers d'autres exercices).
   Où ? En dehors du lexique principal et algorithme principal (souvent après).

Principe fondamental

Séparation des fonctions d'affichage et de traitement

Il est **indispensable** de séparer :

- les fonctions d'**affichage/saisie** permettant à l'homme et la machine de s'échanger des informations
- des fonctions de **traitement** qui utilisent et génèrent uniquement des informations présentes dans la mémoire de la machine (ie : les variables)

Exemple

Proposer un découpage fonctionnel pour le problème suivant : "demander à un utilisateur de saisir le rayon d'un disque et lui calculer puis afficher le périmètre et la surface du disque"

Une proposition de solution

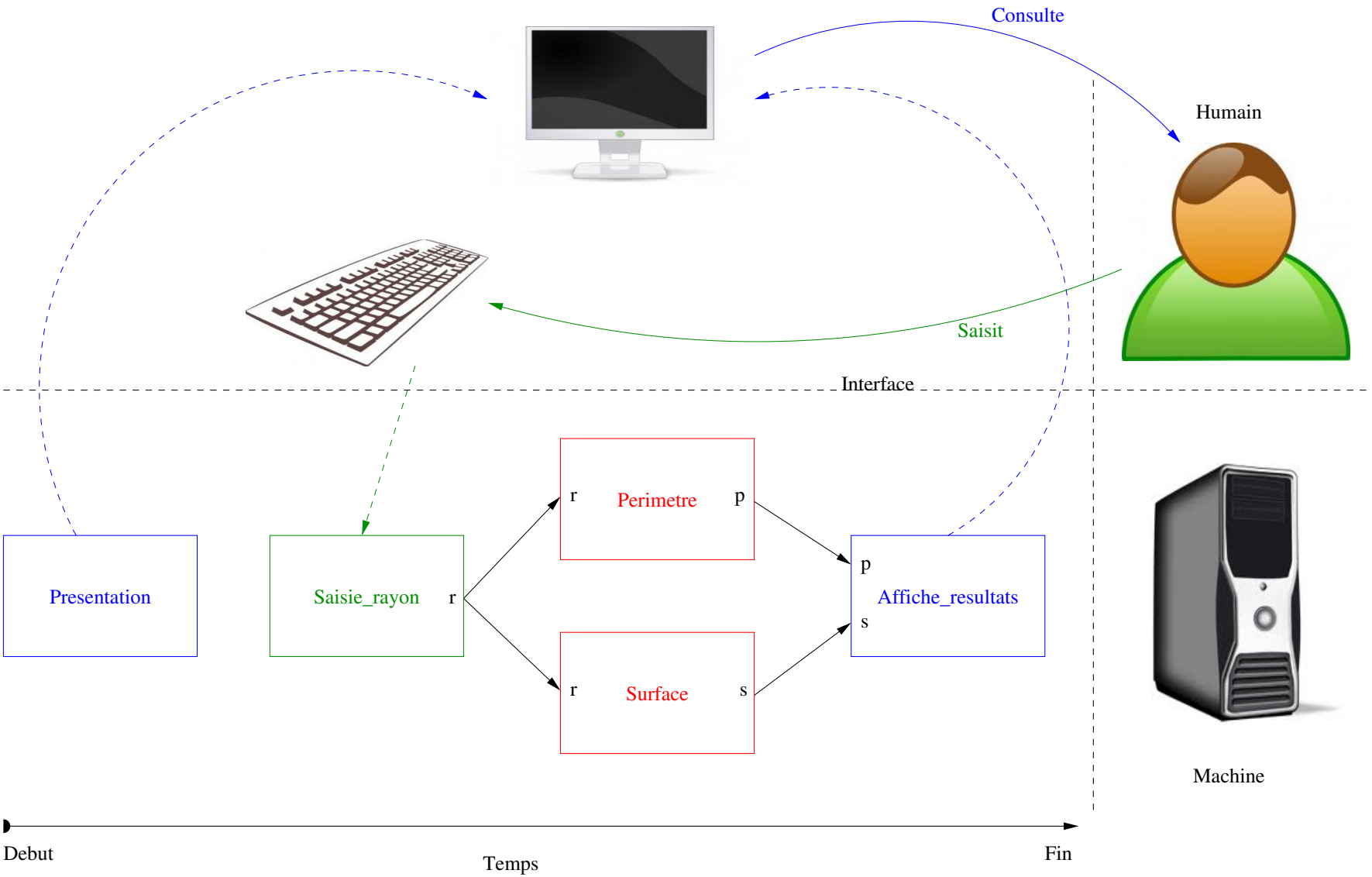
On identifie les fonctions suivantes :

1. `Presentation`, rôle : expliquer ce que fait le programme, entrée : vide, sortie : vide
2. `Saisie_rayon`, rôle : demande à l'utilisateur d'entrer un rayon, entrée : vide, sortie : 1 réel (le rayon)
3. `Perimetre`, rôle : calcule le périmètre d'un cercle de rayon r, entrée : 1 réel (le rayon), sortie : 1 réel (le périmètre)
4. `Surface`, rôle : calcule la surface d'un disque de rayon r, entrée : 1 réel (le rayon), sortie : 1 réel (la surface)
5. `Affiche_resultats`, rôle : afficher les valeurs du périmètre et de la surface, entrée : 2 réels (périmètre et surface), sortie : vide

Exercice [Ch] (TD) : Résolution du problème

1. Proposer lexique principal permettant de déclarer ces fonctions,
2. Proposer l'algorithme principal permettant de les utiliser pour répondre au problème posé.
3. Proposer ensuite un algorithme permettant de définir chacune de ces fonctions.

Représentation graphique du découpage fonctionnel



Sommaire

Introduction

- Machine disposant d'un processeur vu principalement comme une unité de calcul $(+, -, \times, \div)$ [14]
- Jeu d'instructions élémentaires codé en langage machine binaire (suite de 0 et de 1) dépendant de chaque processeur.
- Comment dire à la machine d'exécuter un ensemble d'instructions pour un processeur donné ?
  2 solutions :
  1. Connaître le jeu d'instructions de chaque processeur et leur codage en binaire ;
  2. Ecrire ces instructions dans un langage général [15], **normalisé** puis **traduire** ces instructions en langage machine.
- Deuxième solution évidemment préférable
- Langage utilisé dans la suite de ce cours : langage C et C++.

14. Plus de détails sur l'architecture d'un système à microprocesseur au S2 (Info2)
15. Indépendant d'un processeur donné

Petit historique du langage C

- Apparu en 1972 : Laboratoire Bell, en même temps qu'UNIX
- Développé par Dennis Ritchie puis popularisé et amélioré par Brian Kernighan (1978) ⇒ première **norme** C K&R [16]
- Normalisation successives : ANSI C (1989) puis reprise par l'ISO en 1990.
- Evolution de la norme ISO en 1999 (C99) et en 2011 (C11)

Dans ce cours

- Utilisation du C ANSI
- Utilisation de quelques fonctionnalités pratiques du C++ :
  - Communications avec l'utilisateur : `cout`, `cin`, au lieu de `scanf` et `printf`
  - Traduction du type booléen (qui n'existe pas en C) en C++

16. Kernighan, B.W. & Ritchie D.M., *The C Programming Language* Prentice Hall, Eaglewood Cliffs NJ, 1978.

Développement de code en langage C

- 3 outils nécessaires :
  - **Editeur de texte** : permet de taper le code C, il en existe avec coloration syntaxique pour faire ressortir les mots clés du langage.
  - **Compilateur** : son rôle est de transformer le code tapé en langage C en un code compréhensible par le processeur qui va exécuter le programme.
    Il analyse le code tapé et détecte les erreurs de syntaxe ne correspondant pas à la norme ANSI.
  - **Débogueur** : permet de tester le programme et de détecter les problèmes lors de l'exécution du programme.

Important :

Les deux derniers outils détectent les erreurs, ils ne les corrigent pas !

- Souvent ces outils sont regroupés dans un seul, on parle d'**IDE** (Integrated Developement Environment).
  Dans ce module, nous utiliserons Microsoft Visual Studio Community. [17]
  Vous êtes bien évidemment libre d'utiliser d'autres IDE (`Code::Blocks`, ...) ou encore les outils séparemment (`notepad++`/`emacs`, `gcc`, `gdb`, ...).
  Le plus important est que vous pratiquiez en autonomie le plus régulièrement possible.

Avant de commencer

- Méthode de travail :
  1. **Analyse** du problème posé (cerveau)
  2. Proposition d'**algorithmes** (cerveau)
  3. **Traduction** de l'algorithme en C (cerveau + "dictionnaire")
  4. **Compilation** du code ⇒ création d'un fichier (dit fichier objet) contenant des instructions compréhensibles par la machine (IDE + cerveau)
  5. Génération de l'**exécutable** (IDE)
  6. **Débogage** de l'exécutable (IDE+cerveau)
- On étudiera **systématiquement** les principes algorithmiques avant d'étudier leurs traductions en langage C.

Rappel

Si les résultats escomptés ne sont pas obtenus, c'est le plus souvent la **conception de l'algorithme** et/ou la traduction en C qui est à remettre en question et non pas la machine.

Remarque

Malgré "l'aide" de l'IDE, le **cerveau** sert encore $5/6^e$ du temps !

Contenu minimal d'un code en C

```
#include<...>
#include"..."
...
int main()
{
    //Commentaires
return 0;
}
```

Directives du préprocesseur

Fonction principale

Remarques

- Les directives du type `#include<...>` ou `#include"..."` permettent d'inclure des fichiers contenant les **déclarations** de fonctions existantes ou de vos propres fonctions. Ce ne sont pas des instructions à proprement parler.
- La présence d'une fonction principale `main()` est **indispensable et obligatoire** [a].
- Le `return 0;` (conforme à la norme ISO) placé à la fin du programme principal renvoie l'entier 0 et indique donc que toutes les instructions précédant cette dernière ont bien été exécutées.
- Toutes les instructions se terminent par un point-virgule ;

a. Sans cette fonction, la génération de l'exécutable ne peut pas se faire.

Remarque importante

Conseil pratique

D'un point de vue pratique il est fortement conseillé d'écrire le **squelette** général du code ou des structures (alternatives, itératives) algorithmiques qu'on étudiera par la suite.
Ceci permet de compiler le code "à vide" et **le plus régulièrement possible** afin de vérifier qu'il n'y a pas d'oubli d'accolades ou autres éléments de syntaxe avant que le code ne contienne 30000 lignes qui n'ont jamais été compilées.

Mauvais Exemple

```
#include<iostream>
using namespace std;
int main()
{
    return 0;
}//risque d'oubli de l'accolade fermante
```

Remarque importante

Conseil pratique

D'un point de vue pratique il est fortement conseillé d'écrire le **squelette** général du code ou des structures (alternatives, itératives) algorithmiques qu'on étudiera par la suite.
Ceci permet de compiler le code "à vide" et **le plus régulièrement possible** afin de vérifier qu'il n'y a pas d'oubli d'accolades ou autres éléments de syntaxe avant que le code ne contienne 30000 lignes qui n'ont jamais été compilées.

Mauvais Exemple

```
#include<iostream>
using namespace std;
int main()
{
    return 0; //ne compile pas à cet instant
}//risque d'oubli de l'accolade fermante
```

Remarque importante

Conseil pratique

D'un point de vue pratique il est fortement conseillé d'écrire le **squelette** général du code ou des structures (alternatives, itératives) algorithmiques qu'on étudiera par la suite.
Ceci permet de compiler le code "à vide" et **le plus régulièrement possible** afin de vérifier qu'il n'y a pas d'oubli d'accolades ou autres éléments de syntaxe avant que le code ne contienne 30000 lignes qui n'ont jamais été compilées.

Mauvais Exemple

```
#include<iostream>
using namespace std;
int main()
{
    return 0;
}//risque d'oubli de l'accolade fermante
```

Remarque importante

Conseil pratique

D'un point de vue pratique il est fortement conseillé d'écrire le **squelette** général du code ou des structures (alternatives, itératives) algorithmiques qu'on étudiera par la suite.
Ceci permet de compiler le code "à vide" et **le plus régulièrement possible** afin de vérifier qu'il n'y a pas d'oubli d'accolades ou autres éléments de syntaxe avant que le code ne contienne 30000 lignes qui n'ont jamais été compilées.

Conseils

```
#include<iostream>
using namespace std;
int main()
{
}    return 0;
}
```

Remarque importante

Conseil pratique

D'un point de vue pratique il est fortement conseillé d'écrire le **squelette** général du code ou des structures (alternatives, itératives) algorithmiques qu'on étudiera par la suite.
Ceci permet de compiler le code "à vide" et **le plus régulièrement possible** afin de vérifier qu'il n'y a pas d'oubli d'accolades ou autres éléments de syntaxe avant que le code ne contienne 30000 lignes qui n'ont jamais été compilées.

Conseils

```
#include<iostream>
using namespace std;
int main()
{
    return 0;
}
```

Remarque importante

Conseil pratique

D'un point de vue pratique il est fortement conseillé d'écrire le **squelette** général du code ou des structures (alternatives, itératives) algorithmiques qu'on étudiera par la suite.
Ceci permet de compiler le code "à vide" et **le plus régulièrement possible** afin de vérifier qu'il n'y a pas d'oubli d'accolades ou autres éléments de syntaxe avant que le code ne contienne 30000 lignes qui n'ont jamais été compilées.

Conseils

```
#include<iostream>
using namespace std;
int main()
{
    return 0;
}
```

Exemple célèbre et commentaires

- En langage C

```
#include<stdio.h>  //Pour printf() et scanf()
                                 Ecrire()   Lire()
int main()
{//Entrée dans la fonction principale
   printf("Hello World!\n");
return 0;
}//Sortie de la fonction principale
```

- En langage C (et en utilisant cin et cout du C++)

```
#include<iostream>  //Pour   cout    et   cin
                                Ecrire()   Lire()
using namespace std;
int main()
{//Entrée dans la fonction principale
   cout<<"Hello World!"<<     endl      ;
                              saut de ligne
return 0;
}//Sortie de la fonction principale
```

Traduction des parties lexique principal et algorithme principal

```
#include<...>
...
...
```

Inclusion des fichiers d'en-tête contenant les déclarations des fonctions **en dehors du** `main()`
Déclarations **globales**

```
int main()
{
//Lexique principal
   //Déclaration des variables
   /*Déclaration/initialisation
   des constantes*/
//Algorithme principal
   //Début
    //Instructions
   //Fin
return 0;
}
```

Fonction principale

Types de base et déclarations de variables en C

Pour déclarer une variable, la syntaxe sera de la forme :

```
<nom_type> <nom_variable>;
```

- `<nom_type>` peut correspondre à un des types suivants :
  - **les entiers signés** : `int` sur 16 bits ou 32 bits
  - **les entiers non signés** : `unsigned int` sur 16 bits ou 32 bits
  - **les réels** : `float` sur 32 bits ou `double` sur 64 bits
  - **les caractères** : `char` [18]
  - **les chaînes de caractères** : Voir le chapitre sur les Tableaux (les chaînes de caractères sont en fait des tableaux de caractères)
  - **les booléens** : ce type n'existe pas en C [19].
    La valeur 0 est considérée comme ayant la valeur FAUX.
    N'importe quelle valeur $\neq 0$ est considérée comme VRAI.
  - **les adresses** : ce sont les "fameux" pointeurs auxquels on consacrera une partie à la fin de ce semestre.
- Il suffit de rajouter le mot clé `const` avant le type pour déclarer une constante. Ne pas oublier de l'initialiser ! (Ex : `const char C='a';`)

18. pouvant être considérés comme des entiers signés sur 8 bits ou non signés si on utilise `unsigned char` (parallèle à faire avec le cours de SIN1)
19. Il existe un type booléen en C++ : `bool`

Exemples

- Traduisez le lexique suivant en C :

Lexique :

x,y,z : 3 réel
n,m : 2 entier
rayon, surface : 2 entier
c : 1 caractère
PI : la constante réelle :=3.14159265

```
//Une solution:
   float x,y,z;
   int n,m;
   unsigned int rayon,surface;
   char c;
   const double PI=3.14159265;
```

Remarque :

On distingue en C/C++ le signe des entiers alors qu'en algorithmie ce n'est pas nécessaire.
⇒ analyser les valeurs prises par les variables de type entier afin de les traduire en C avec le bon type `int` ou `unsigned int` [a]

a. Pas comme Youtube à l'époque de Gangnam style

Les opérateurs de base en C

Nom	Algorithmie	Langage C
Affectation	$\longleftarrow$	`=`
Addition	`+`	`+`
Soustraction	`-`	`-`
Multiplication	`*`	`*`
Division réelle	`/`	`/` [1]
Division entière	`div`	`/` [1]
Reste	`reste`	`%`
Afficher un message à l'utilisateur	`Ecrire()`	`cout` [2]
Récupérer saisie de l'utilisateur	`Lire()`	`cin` [2]

Les pièges à éviter (95% des étudiants feront l'erreur) :

- Affectation qui se traduit avec un symbole = à ne pas confondre avec l'opérateur logique d'égalité utilisé en algorithmie.
- Le symbole de division identique pour la division réelle et entière. Comparer 1/2 et 1.0/2 en Algorithmie puis en C.

1. Attention, même traduction mais usage complétement différent...
2. Venant du C++ mais offrant un usage beaucoup plus souple que `printf` et `scanf`

Exemple complet (algo)

- Proposez un algorithme qui permet de demander à un utilisateur la valeur d'un rayon qui calcule et affiche la surface d'un disque en fonction du rayon ($S = \pi R^2$)
- Une solution :

Lexique :

PI : la constante réelle := 3.14159265
r, s : deux réels

Algorithme :

Début
  Ecrire("Saisissez une valeur de rayon (en cm) pour votre disque")
  Lire(r)
  $s \longleftarrow PI*r*r$
  Ecrire("La surface du disque de rayon ",r,"cm est de ",s,"cm^2")
Fin

Exemple complet (traduit en C)

```
#include<iostream> /*pour communiquer avec l'utilisateur
                     cin et cout*/

using namespace std;
//pour éviter d'écrire std::cin et std::cout

 int main()
 {
 //Lexique
  const float PI=3.14159265;
  float rayon,surface;

 //Algorithme
  //Début
   cout<<"Saisissez une valeur de rayon (en cm)"<<endl;
   cin>>rayon; //Notez le sens différent des chevrons
   surface=PI*rayon*rayon;
   cout<<"La surface du disque de rayon "<<rayon<<"cm est de "
<<surface<<"cm2"<<endl;
  //Fin
return 0;
}
```

- Si on souhaite utiliser la fonction puissance comment faire ?

Exemple complet (traduit en C)

```
#include<iostream>
#include<math.h> // pour utiliser la fonction pow()
using namespace std;

 int main()
 {
 //Lexique
  const float PI=3.14159265;
  float rayon,surface;

 //Algorithme
    cout<<"Saisissez une valeur de rayon (en cm)"<<endl;
    cin>>rayon; //Notez le sens différent des chevrons
    surface=PI*pow(rayon,2);
    cout<<"La surface du disque de rayon "<<rayon<<"cm est de "
<<surface<<"cm2"<<endl;
 return 0;
 }
```

Remarque

On n'a pas besoin de savoir comment est définie la fonction `pow`, mais de connaître son rôle, $\underbrace{\text{ses entrées (dans l'ordre !) et son type de retour}}_{\text{son prototype}}$

Précisions sur l'utilisation de cin et cout

Algo	C/C++
Ecrire("rayon=",r,"cm")	`cout<<''rayon=''<<r<<''cm''<<endl;`
Lire(a)	`cin>>a;`

Lire(a,b)	`cin>>a;//correspond à Lire(a)`
	`cin>>b;//puis à Lire(b)`
	`//ou encore`
	`cin>>a>>b;//traduction de Lire(a,b)`

Sens des Chevrons

On se place du côté de la machine, cout correspond à la sortie standard c'est à dire l'écran de la machine tandis que cin correspond à l'entrée standard, c'est à dire le clavier.
La machine envoie des informations à l'écran :
`cout<< ''information importante'';`
Le clavier envoie des informations à la machine (qu'elle mémorise dans une variable) : `cin>>a;`

Sommaire

Rappels

- Nous avons vu l'intérêt d'utiliser des fonctions et de réaliser un découpage fonctionnel
- Une fonction est dédiée à la réalisation d'un problème donné.
- Elle contient une suite d'instructions permettant de résoudre ce problème.
- C'est donc un algorithme qui peut contenir toutes les instructions et structures qu'on étudiera dans les chapitres suivants.
- 3 **étapes** sont nécessaires pour l'utilisation d'une fonction :
  1. **Déclaration ou prototype**
  2. **Définition**
  3. **Utilisation/Appel** de la fonction

Remarque importante

Une fonction peut avoir **plusieurs paramètres d'entrée**, mais n'a qu'**un seul** type de retour.

Rappel d'algorithmie et correspondance en langage C

En algorithmie	En langage C

{**Déclaration** de la fonction f}
f : la fonction (x : 1 réel)→1 réel
	`//Déclaration de f`
	`float f(float x);`
	`//ou plus simplement`
	`float f(float);`

{**Définition** de la fonction f}
f : la fonction (x : 1 réel)→1 réel
Lexique :{Local à f}
res : 1 réel
Algorithme :{Local à f}
Début
res⟵2*x + 3
Retourner res
Fin
	`//Définition de f`
	`float f(float x)`
	`{ //Lexique`
	`float res;`
	
	`//Algo`
	`res=2*x+3;`
	`return res;`
	`}`

- La **déclaration** et la **définition** de la fonction se font en dehors du `main()` (dans des fichiers spécifiques)
- L'**utilisation/appel** de la fonction se fait dans le `main()` ou dans une autre fonction (déclarée et définie par la suite).

Déclaration d'une fonction

But d'une déclaration

Indique à la machine qu'il **existe** une fonction d'un **nom donné** qui prend *n* **paramètres** en entrée et produit **un seul** paramètre en sortie.
Lors de la **déclaration**, la machine a juste besoin de connaître les **types** des paramètres d'entrée et le **type** retourné et pas forcément les noms de ces informations.
On parle de **prototype** (ou de déclaration) de la fonction.

Où et quand déclarer une fonction ?

La déclaration d'une fonction se fait **avant sa définition** dans un fichier spécifique appelé fichier d'en-tête (*header* en anglais, d'où l'extension du fichier `.h`).
On créera donc un fichier `nom_fichier.h` [a] dans lequel on effectuera les **déclarations** de nos fonctions.

a. Le `nom_fichier` doit être lié aux fonctions qui le composent.

Utilisation du fichier `nom_fichier.h`

- Ce fichier **ne se compile pas**
- Il doit être inclus (`#include"nom_fichier.h"` [a]) dans tous les fichiers **définissant ou utilisant** les fonctions qui y sont déclarées.

a. Agit comme un copier/coller du contenu du fichier

Définition d'une fonction

But de la définition

Permet de décrire comment à partir des paramètres donnés en entrée de la fonction, le résultat est fabriqué.
Lors de la **définition**, il est **nécessaire** de connaître le **nom** des paramètres puisque on va les utiliser pour décrire le résultat.

Où et quand définir une fonction ?

La définition d'une fonction se fait **après sa déclaration** [a] dans un fichier, appelé *fichier source*, contenant uniquement les définitions des autres fonctions (autre que la fonction principale `main()`).
On créera donc un fichier `nom_fichier.cpp` [b] dans lequel on effectuera les **définitions** de nos fonctions autre que la fonction `main()`.

---

a. il convient donc d'inclure (`#include"..."`) le fichier contenant les déclarations des fonctions en en-tête de fichier

b. Le nom du fichier n'a aucune importance, l'extension `.cpp` si.

Utilisation/Appel d'une fonction

On distingue deux cas :

- Utilisation dans la fonction principale `main()` ;

Fichier `main.cpp`

Dans ce premier cas, on souhaite appeler les fonctions dans le `main()`. On créera donc un fichier `main.cpp` dans lequel :

1. On inclura les fichiers contenant les déclarations des fonctions qu'on souhaite appeler (`#include"..."` ou `#include<...>`)
2. On définira la fonction `main()` (sa présence est indispensable) qui contiendra, en particulier, les appels aux fonctions qu'on souhaite utiliser

Le fichier `main.cpp` ne doit donc contenir que les déclarations nécessaires aux appels des fonctions qu'on souhaite utiliser et la définition de la fonction principale `main()`.

- Utilisation dans une autre fonction.

Dans le fichier `nom_fichier.cpp`

Dans ce second cas, il suffit simplement de veiller à ce que la fonction qu'on souhaite appeler ait été **déclarée avant** la définition de la fonction appelante.

Exemple simple mais complet (1/3)

Exercice

Déclarer et définir la fonction f définie pour $x \in \mathbb{R}$ par $f(x) = 2x + 3$. Puis utiliser les traductions en C++ des fonctions Lire() et Ecrire() pour demander à l'utilisateur d'entrer une valeur et lui afficher le résultat retourné par la fonction.

Rappel : 3 étapes à suivre,

1. Création et édition du fichier `exemple.h` contenant la déclaration de la fonction f
2. Création et édition du fichier `exemple.cpp` contenant la définition de la fonction f
3. Création et édition du fichier `main.cpp` contenant la définition de la fonction principale dans laquelle on appelle la fonction f ainsi que les fonctions d'entrée/sortie (Lire() et Ecrire()).

Exemple simple mais complet (2/3)

exemple.h

```
//Déclaration de f
float f(float x);
/*Nom du paramètre
facultatif mais
conseillé*/
```

exemple.cpp

```
/*Inclusion du fichier
contenant la déclaration
de f */
#include"exemple.h"

//Définition de f
float f(float x)
{
  //Lexique local à f
  float res;
  //Algorithme local à f
  res=2*x+3;
  return res;
}
```

main.cpp

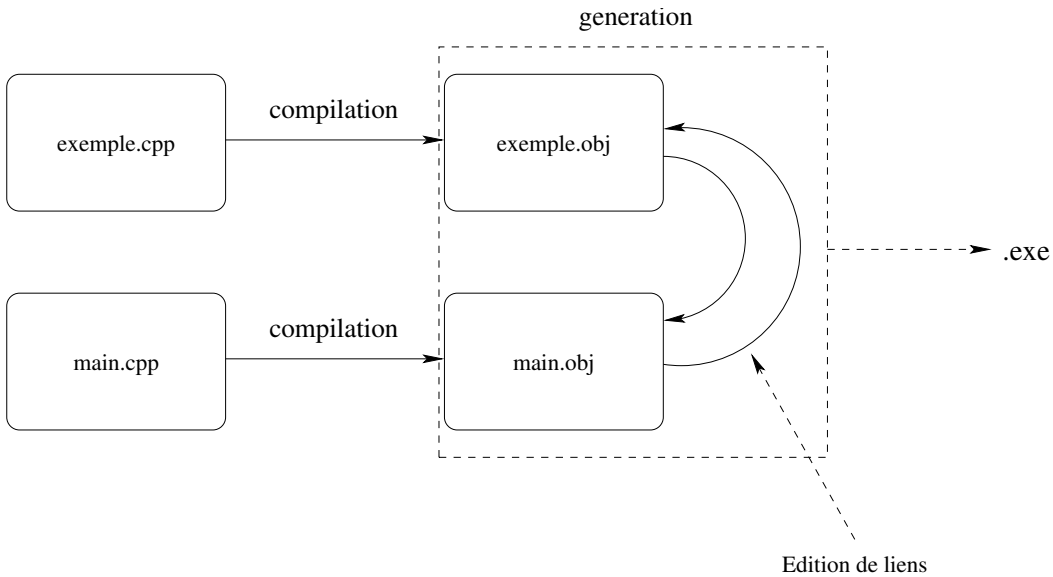
```
/*Inclusion des fichiers
contenant les déclarations
de f et de cin, cout*/
#include<iostream>
using namespace std;
#include"exemple.h"

//fonction principale
int main()
{
  //Lexique principal
  float x_in,r;
  //Algorithme principal
  cout<<"Entrer x: ";
  cin>>x_in;
  r=f(x_in);
  cout<<"f("<<x_in<<")="<<r;
  return 0;
}
```

Commentaires

- Pour la **déclaration** : seulement besoin de connaître les **types**
- Pour la **définition** : besoin du **type et du nom**
- Pour l'**appel** à la fonction `f` dans le `main`, aurait-on pu nommer les variables `x` et `res` au lieu de `x_in` et `r` ? Pourquoi ?

Exemple simple mais complet (3/3)

- Une fois les fichiers créés et édités (on verra comment le faire sous visual studio), il faut les compiler
- Rappel : Le rôle de la **compilation** est de **traduire** un code en langage C en un code, dit **objet** (extension `.obj`), compréhensible par la machine.
  - le fichier `exemple.h` ne **se compile pas**
  - les fichiers `exemple.cpp` et `main.cpp` se compilent **indépendament** l'un de l'autre.
- Il reste à faire le **lien** entre la définition de la fonction en langage machine dans le fichier `exemple.obj` et son appel dans le fichier `main.obj` pour fabriquer l'**unique** fichier **exécutable** qui permettra de lancer et tester le programme.



- Qu'affiche ce programme lorsqu'on l'exécute ?

Fonction : $x \rightarrow d\_atan(x) = \frac{1}{1+x^2}$ où x est un réel

- Reprendre l'algorithme proposé pour cette fonction et le traduire en C en reprenant la même structure de fichiers que dans l'exemple complet.

fonctions.h

```
//Déclaration
float D_atan(float x);
```

fonctions.cpp

```
//Fichier contenant la déclaration de d_atan
#include"fonctions.h"

//Définition
float D_atan(float x)
{
  //Lexique local
  float res;
  //Algorithme local
  res=1/(1+x*x);
  return res;
}
```

main.cpp

```
/*Fichiers contenant les déclarations
de d_atan et de cin, cout*/
#include<iostream>
using namespace std;
#include"fonctions.h"

//fonction principale
int main()
{
  //Lexique principal
  float x_in,r;
  //Algorithme principal
  cout<<"Entrer x: ";
  cin>>x_in;
  r=D_atan(x_in);
  cout<<"D_atan("<<x_in<<")="<<r;
  return 0;
}
```

Fonction : $t \rightarrow Cel2Far(t)$ où t est un réel

- De même pour la fonction `Cel2Far`

conversions.h

```
//Déclaration
float Cel2Far(float t);
```

conversions.cpp

```
//Fichier contenant la déclaration de Cel2Far
#include"conversions.h"

//Définition
float Cel2Far(float t)
{
  //Lexique local
  float res;
  //Algorithme local
  res=9.0/5*t+32; //Pourquoi 9.0 et pas 9 ?
  return res;
}
```

main.cpp

```
/*Fichiers contenant les déclarations
de Cel2Far et de cin, cout*/
#include<iostream>
using namespace std;
#include"conversions.h"

//fonction principale
int main()
{
  //Lexique principal
  float cel,far;
  //Algorithme principal
  cout<<"Température en °C: ";
  cin>>cel;
  far=Cel2Far(cel);
  cout<<"correspond à "<<far<<"°F";
  return 0;
}
```

Fonction : $(x, y) \rightarrow g(x, y) = x^2 - y$ où x et y sont des entiers

- De même pour la fonction g

fonctions.h

```
//Déclaration
int g(int x,int y);

/*Pourquoi int et pas
unsigned int ? */
```

fonctions.cpp

```
//Fichier contenant la déclaration de g
#include"fonctions.h"

//Définition
int g(int x,int y)
{
  //Lexique local
  int res;
  //Algorithme local
  res=x*x-y;
  return res;
}
```

main.cpp

```
/*Fichiers contenant les déclarations
de g et de cin, cout*/
#include<iostream>
using namespace std;
#include"fonctions.h"

//fonction principale
int main()
{
  //Lexique principal
  int x_in,y_in,r;
  //Algorithme principal
  cout<<"Entrer x puis y: ";
  cin>>x_in>>y_in;
  r=g(x_in,y_in);
  cout<<"g("<<x_in<<","<<y_in<<")="<<r;
  return 0;
}
```

Une fonction qui ne renvoie rien : La fonction d'affichage Affiche_note vue précedemment

affiche.h

```
//Déclaration
void
Affiche_note(float note,
float bareme);

/*Mais pas
void
Affiche_note(float note, bareme);
*/
```

affiche.cpp

```
//Fichier contenant la déclaration de Affiche_note
#include<iostream> //Car cout est utilisé
using namespace std;
#include"affiche.h"

//Définition
void Affiche_note(float note, float bareme)
{
  //Lexique local (vide ici)
  //Algorithme local
  cout<<"Vous avez "<<note<<"/"<<bareme;
}
```

main.cpp

```
/*Fichiers contenant les déclarations
de Affiche_note et de cin, cout*/
#include<iostream>
using namespace std;
#include"affiche.h"

//fonction principale
int main()
{
  //Lexique principal
  float eton,emerab;
  //Algorithme principal
  cout<<"Entrer votre note: ";
  cin>>eton;
  cout<<"puis le bareme: ";
  cin>>emerab;
  Affiche_note(eton,emerab);
  return 0;
}
```

- Aurait-on pu utiliser les noms de variables note et bareme en lieu et place de eton et emerab dans le fichier main.cpp ? Pourquoi ?

Une fonction sans paramètre : fonction de saisie

saisie.h

```
//Déclaration
float Saisir_note(void);
```

saisie.cpp

```
//Fichier contenant la déclaration de Saisir_note
#include<iostream> //car cin et cout utilisés ici.
using namespace std;
#include"saisie.h"

//Définition
float Saisir_note(void)
{
  //Lexique local
  float note;
  //Algorithme local
  cout<<"Saisir une note ";
  cin>>note;
  return note;
}
```

main.cpp

```
/*Fichier contenant la déclaration
de Saisir_note*/
#include"saisie.h"

//fonction principale
int main()
{
  //Lexique principal
  float eton;
  //Algorithme principal
  eton=Saisir_note();
  return 0;
}
```

- Pourquoi n'y a-t-il pas de `#include<iostream>` dans le `main.cpp` ?
- Aurait-on pu remplacer `eton` par `note` ? Pourquoi ?

Une fonction qui ne renvoie rien et ne prend pas de paramètre

- Cas très particulier des fonctions qui permettent d'afficher des messages constants et qui n'attendent aucune saisie de l'utilisateur.

affiche.h

```
//Déclaration
void And_Now(void);

/*
Attention!
Mais surtout pas:
And_Now(); //Pourquoi?
*/
```

affiche.cpp

```
//Fichier contenant la déclaration de And_Now
#include<iostream> //car cout utilisé ici.
using namespace std;
#include"affiche.h"

//Définition
void And_Now(void)
{
  //Lexique local (vide ici)
  //Algorithme local
  cout<<"Nous allons pouvoir étudier, à présent
toutes les structures algorithmiques et leurs
traductions en C qu'on pourra trouver dans les
fonctions ou plus généralement dans tout algorithme";
}
```

main.cpp

```
/*Pas de #include<iostream>
car pas d'appel à cin ni cout dans la
fonction main()*/
//Fichier contenant la déclaration de And_Now
#include"affiche.h"

//fonction principale
int main()
{
   //Lexique principal vide ici
   //Algorithme principal
   And_Now();
   return 0;
 }
```

Sommaire

Introduction

Notion de choix

Dans les algorithmes précédents toutes les instructions sont exécutées, les unes à la suite des autres, dans l'ordre où elles apparaissent.
Cependant, on aimerait pouvoir ajouter des tests permettant d'effectuer une suite d'instructions plutôt qu'une autre (p. ex : lors d'une division, lors d'une saisie de note, déclenchement d'une alarme, ...)
On va donc **choisir** une séquence d'instructions à réaliser en fonction du résultat d'un test (VRAI ou FAUX), c'est à dire en fonction de la valeur d'une variable **booléenne**.

Choix mutuellement exclusifs

Il est très important de noter qu'on ne considèrera que **deux** issues possibles pour le résultat du test : VRAI ou FAUX qui sont mutuellement exclusives.

Instruction Si : le modèle

Canevas

Lexique :

...

Algorithme :

Début

```
    ...
    Si condition
      Alors
        instruction01
        instruction02
        ...
      Sinon
        instruction11
        instruction12
        ...
    FinSi
    ...
```

Fin

Instruction Si : le modèle

Canevas

Lexique :

...

Algorithme :

Début

...

Si condition

Alors

instruction01

instruction02

...

Sinon

instruction11

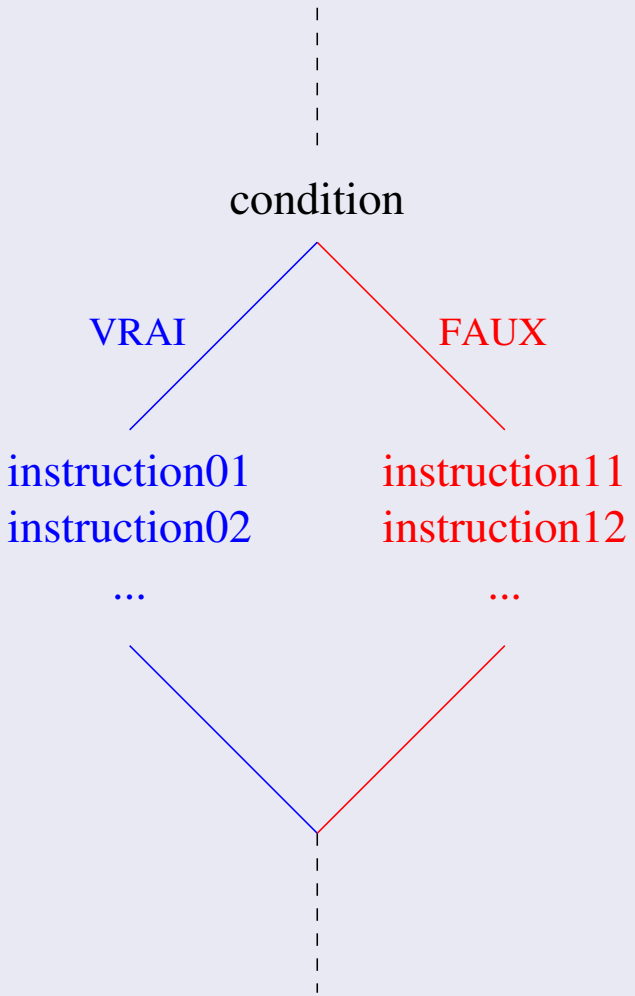
instruction12

...

FinSi

...

Fin



Commentaires

Bon sens

- Si `condition` est VRAI, alors `condition` n'est pas FAUX
- Si `condition` est FAUX, alors `condition` n'est pas VRAI
  $\Rightarrow$ il n'y a pas de place pour le peut-être !
  On se trouve **SOIT** dans une branche, **SOIT** dans l'autre, donc ni dans aucune, ni dans les deux.
- Une fois qu'on a effectué les instructions d'une des deux branches, l'instruction `Si` prend fin (mot clé : `FinSi`) et on poursuit les instructions de l'algorithme.
- Les instructions à l'intérieur d'une structure `Si` peuvent contenir d'autres instructions `Si` (**Imbrication** des `Si` $\rightarrow$ TD)

Lien avec le cours de SIN1

- Le fait d'avoir uniquement deux états possibles pour `condition` doit vous faire penser à une notion vue en SIN1 : laquelle ?

Opérateurs logiques

- `condition` peut être vue comme une variable logique (ou variable booléenne)
- Besoin d'opérateurs spécifiques : **opérateurs logiques** agissant sur les variables logiques et **opérateurs de comparaison** agissant sur des variables non booléennes (p. ex : entier, réel, caractère, chaîne de caractère).

Opérateurs de comparaison		Opérateurs logiques	
$=$	égalité entre deux variables	ET	cf : SIN1
$\neq$	inégalité entre deux variables		
$<$	"est strictement plus petit que"	OU	cf : SIN1
$\leq$	"est plus petit ou égal à"		
$>$	"est strictement plus grand que"	NON	cf : SIN1
$\geq$	"est plus grand ou égal à"		

- Important : tous ces opérateurs retournent une valeur **booléenne** : VRAI ou FAUX

Exemple : affichage d'un message conditionnel

- Ecrire un algorithme pour la fonction dont le rôle est d'afficher le message "passage autorisé" si la note passée en paramètre est supérieure ou égale à 10 et "redoublement" sinon.
- Proposer un algorithme principal permettant de tester cette fonction.

Exemple : affichage d'un message conditionnel

- Ecrire un algorithme pour la fonction dont le rôle est d'afficher le message "passage autorisé" si la note passée en paramètre est supérieure ou égale à 10 et "redoublement" sinon.
- Proposer un algorithme principal permettant de tester cette fonction.

```
Passoured : la fonction(n : 1 réel) → vide
  Lexique local
    vide {ici, on travaille directement sur le paramètre fourni}
  Algorithme local
  Début
     Si n≥ 10
       Alors
         Ecrire("passage autorisé")
       Sinon
         Ecrire("redoublement")
     FinSi
  Fin
```

```
Lexique principal :
Passoured : la fonction(n : 1 réel) → vide
note : 1 réel
Algorithme principal :
Début
      Ecrire("Saisissez une note")
      Lire(note)
      Passoured(note)
Fin
```

Cas particulier : le Sinon est facultatif

Canevas

Lexique :

...

Algorithme :

Début

```
    ...
    Si condition
      Alors
        instruction01
        instruction02
        ...
    FinSi
    ...
```

Fin

Cas particulier : le Sinon est facultatif

Canevas

Lexique :

...

Algorithme :

Début

...

Si condition

Alors

instruction01

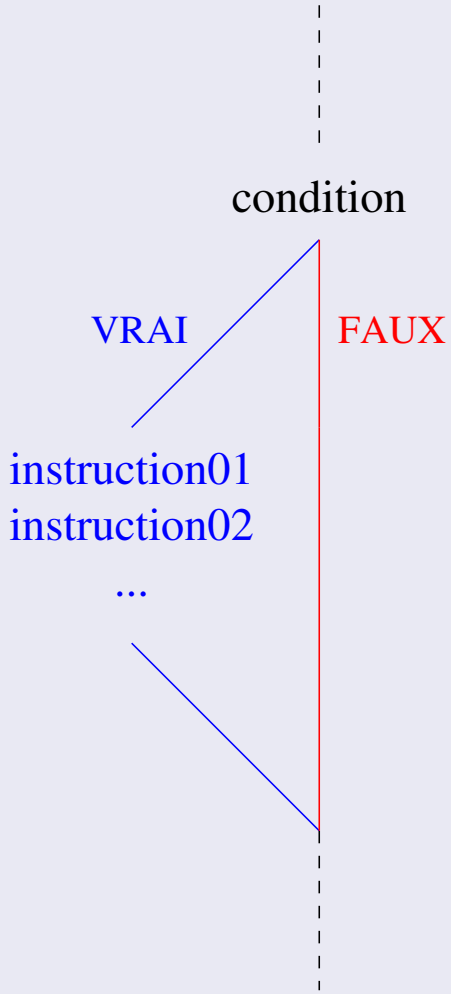
instruction02

...

FinSi

...

Fin



Exemple

- Ecrire un algorithme pour la fonction dont le rôle est d'afficher le message "Risque de Verglas" si la température passée en paramètre est strictement inférieure à 2°C
- Proposer un algorithme principal permettant de tester cette fonction.

Exemple

- Ecrire un algorithme pour la fonction dont le rôle est d'afficher le message "Risque de Verglas" si la température passée en paramètre est strictement inférieure à 2°C
- Proposer un algorithme principal permettant de tester cette fonction.

```
Verglas : la fonction(t : 1 réel) → vide
  Lexique local
    vide
  Algorithme local
  Début
     Si t< 2
       Alors
         Ecrire("Risque de Verglas !")
     FinSi
  Fin
```

```
Lexique principal :
Verglas : la fonction(t : 1 réel) → vide
temp : 1 réel
Algorithme principal :
Début
      Ecrire("Saisissez une température")
      Lire(temp)
      Verglas(temp)
Fin
```

- De nombreux autres exemples et exercices seront donnés en TD.

Introduction

La composition `Selon` : contexte

On rencontre une telle structure lorsqu'on doit prendre une décision en fonction (**selon**) **de la valeur** que peut prendre une variable. [a]

---

a. Le selon est utilisé en général lorsque la variable peut prendre **strictement plus que 2** valeurs. Dans le cas contraire on préfèrera utiliser une structure Si...FinSi.

Exemples

- Gestion d'un menu
- Choix d'opérations (simulation de calculatrice)
- Correspondance chiffres 1-7 et nom du jour de la semaine
- ...

Intérêt

Eviter les imbrications trop complexes de `Si...Sinon...FinSi` lorsqu'il n'y a pas de notion de priorité dans les conditions à tester.

Instruction `Selon` : le modèle

Canevas

Lexique :

...

Algorithme :

Début

```
    ...
    {On suppose var connue (paramètre ou Lexique)}
    Selon var
      val1 : instr11
             instr12
             ...
      val2 : instr21
             instr22
             ...
      ...
    FinSelon
    ...
```

Fin

Instruction `Selon` : le modèle

Canevas

Lexique :

...

Algorithme :

Début

...

{On suppose `var` connue (paramètre ou Lexique)}

Selon var

val1 : instr11

instr12

...

val2 : instr21

instr22

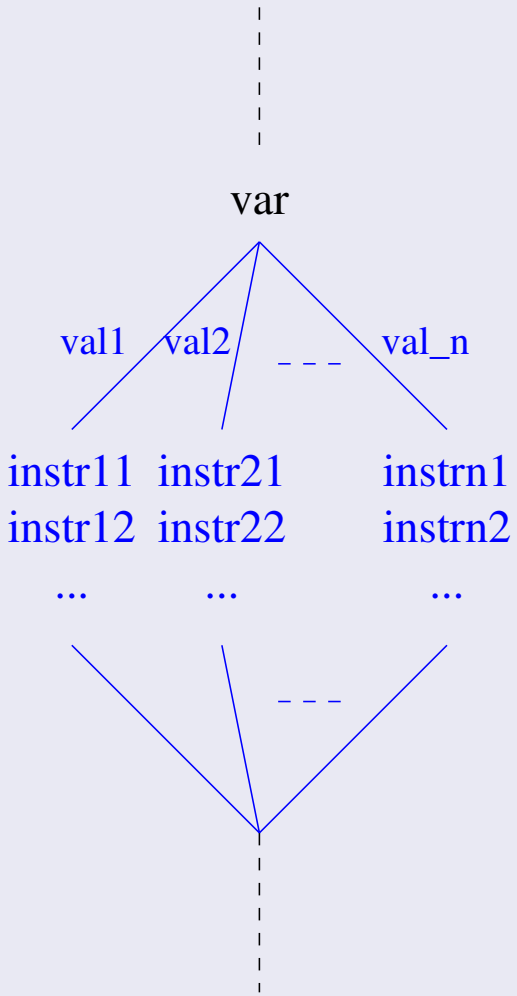
...

...

FinSelon

...

Fin



Principe de fonctionnement

- On teste d'abord si la variable `var` a pour valeur `val1`
  Si c'est le cas, on effectue l'ensemble des instructions correspondant à ce cas puis on quitte la structure `Selon`
- Dans le cas contraire, on passe à `val2` et on procède de la même manière.
- etc ...
- Si aucune valeur ne correspond aux cas prévus, on quitte la structure `Selon` sans avoir effectué d'instruction.

Cohérence des types

Il est indispensable que les valeurs (`val1`, `val2`, etc...) soient du **même type** que la variable testée.

Comparaison implicite

Le fait que la variable soit comparée aux différentes valeurs possibles est **implicite**.
On n'écrit JAMAIS d'opération de comparaison à la place de `val1`, `val2`, etc...

Exemple

- Ecrire un algorithme pour la fonction dont le rôle est d'afficher une appréciation selon le grade entre 'A' et 'F' passé en paramètre :
  - grade A : Excellent
  - grade B : Bien
  - grade C : Assez Bien
  - grade D : Passable
  - grade E : Insuffisant
  - grade F : Inacceptable

Exemple

- Ecrire un algorithme pour la fonction dont le rôle est d'afficher une appréciation selon le grade entre 'A' et 'F' passé en paramètre :
  - grade A : Excellent
  - grade B : Bien
  - grade C : Assez Bien
  - grade D : Passable
  - grade E : Insuffisant
  - grade F : Inacceptable

```
Apprec : la fonction(g : 1 caractère) → vide
  Lexique local
    vide
  Algorithme local
  Début
    Selon g
      'A' : Ecrire("Excellent")
      'B' : Ecrire("Bien")
      'C' : Ecrire("Assez Bien")
      'D' : Ecrire("Passable")
      'E' : Ecrire("Insuffisant")
      'F' : Ecrire("Inacceptable")
    FinSelon
  Fin
```

```
Lexique principal :
Apprec : la fonction(g : 1 caractère) → vide
grade : 1 caractère
Algorithme principal :
Début
      Ecrire("Saisissez un grade entre A et F")
      Lire(grade)
      Apprec(grade)
Fin
```

La structure Selon ...autrement

- Dans l'exemple précédent on suppose que l'utilisateur va entrer un grade acceptable : 'A', 'B', 'C', 'D', 'E' ou 'F' (Notion de confiance)
- Que se passe-t-il si ce n'est pas le cas ?

La structure `Selon ...autrement`

- Dans l'exemple précédent on suppose que l'utilisateur va entrer un grade acceptable : 'A', 'B', 'C', 'D', 'E' ou 'F' (Notion de confiance)
- Que se passe-t-il si ce n'est pas le cas ?
- Il est prévu un cas par défaut appelé `autrement` qui est positionné avant le `FinSelon`
- Ce cas est atteint uniquement lorsque la valeur de la variable est différente de tous les cas prévus dans le `Selon`

Instruction `Selon ...autrement` : le modèle

Canevas

Lexique :

...

Algorithme :

Début

```
    ...
   {On suppose var connue}
    Selon var
      val1 : instr11
             instr12
             ...
      val2 : instr21
             instr22
             ...
      autrement : instructionA1
                  instructionA2
                  ...
     FinSelon
    ...
```

Fin

Instruction Selon ...autrement : le modèle

Canevas

Lexique :

...

Algorithme :

Début

...

{On suppose var connue}

Selon var

val1 : instr11

instr12

...

val2 : instr21

instr22

...

autrement : instructionA1

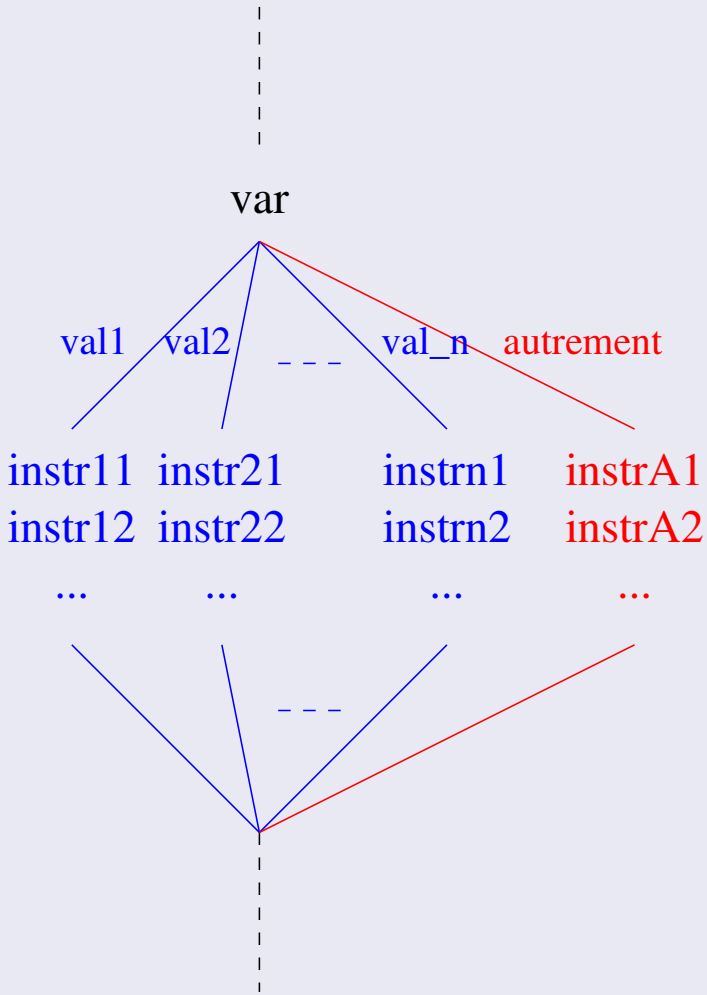
instructionA2

...

FinSelon

...

Fin



Exemple

- On reprend l'exemple précédent et on veut rajouter un message d'erreur si la saisie du grade n'est pas correcte.
(Seul l'algorithme local de la fonction apprec est modifié, le reste est inchangé ! c'est un des intérêts d'une approche fonctionnelle.)

Algorithme local :

Début
    Selon g
      'A' : Ecrire("Excellent")
      'B' : Ecrire("Bien")
      'C' : Ecrire("Assez Bien")
      'D' : Ecrire("Passable")
      'E' : Ecrire("Insuffisant")
      'F' : Ecrire("Inacceptable")
      autrement : Ecrire("Erreur de saisie : le grade doit être compris entre 'A' et 'F' ")
    FinSelon
Fin

Sommaire

Instruction Si : le modèle algo et le codage en C

Canevas

Lexique :

...

Algorithme :

Début

```
    ...
    Si condition
      Alors
        instruction01
        instruction02
        ...
      Sinon
        instruction11
        instruction12
        ...
    FinSi
    ...
```

Fin

```
//Lexique:
...

//Algorithme:
//Début
    ...
    if (condition)
    { //Alors
        instruction01;
        instruction02;
    }
    else
    { //Sinon
        instruction11;
        instruction12;
    }//FinSi
    ...
//Fin
```

Opérateurs de comparaison

- `condition` peut être vue comme une variable logique (ou variable booléenne)
- En C, pas de type booléen : FAUX correspond à la valeur 0 et VRAI correspond à n'importe quelle valeur différente de 0.
- Besoin d'opérateurs spécifiques : **opérateurs de comparaison** agissant sur des variables non booléennes (p. ex : entier, réel, caractère, chaîne de caractères).

Opérateurs de comparaison		
Nom	Algorithmie	C
égalité entre deux variables	$=$	==
inégalité entre deux variables	$\neq$	!=
"est strictement plus petit que"	$<$	<
"est plus petit ou égal à"	$\leq$	<=
"est strictement plus grand que"	$>$	>
"est plus grand ou égal à"	$\geq$	>=

- Important : tous ces opérateurs retournent une valeur ($\neq 0$: VRAI ou 0 : FAUX)

Opérateurs logiques

- `condition` peut être vue comme une variable logique (ou variable booléenne)
- En C, pas de type booléen : FAUX correspond à la valeur 0 et VRAI correspond à n'importe quelle valeur différente de 0.
- Besoin d'opérateurs spécifiques : **opérateurs logiques** agissant sur les variables logiques.

Opérateurs logiques		
Nom	Algorithmie	C
ET Logique	ET	&&
OU Logique	OU	||
NON Logique	NON	!

- Important : tous ces opérateurs retournent une valeur ($\neq 0$: VRAI ou 0 : FAUX)

Instruction Si : Exemple, utilisation directe dans un main

Lexique :

a,b : 2 entier

Algorithme :

Début
  Ecrire("Entrez 2 entiers")
  Lire(a,b)
  Si a<b
    Alors
      Ecrire("croissant")
    Sinon
      Ecrire("décroissant")
  FinSi
  Ecrire("bye !")
Fin

```
#include<iostream>

using namespace std;

int main()
{
 int a,b;
 cout<<"Entrez 2 entiers\n";
 cin>>a>>b;
 if (a<b)
 {
  cout<<"croissant"<<endl;
 }
 else
 {
  cout<<"décroissant"<<endl;
 }
 cout<<"bye!"<<endl;
 return 0;
}
```

Exemple : affichage d'un message conditionnel

- Traduire l'algorithme pour la fonction dont le rôle était d'afficher le message "passage autorisé" si la note passée en paramètre est supérieure ou égale à 10 et "redoublement" sinon.
- Proposer la traduction de l'algorithme principal qui a permis de tester cette fonction.

Exemple : affichage d'un message conditionnel (correction)

scolarite.h

```
//Déclaration
void
Passeoured(float);
```

scolarite.cpp

```
#include<iostream> //Car cout est utilisé
using namespace std;
//Fichier contenant la déclaration de passeoured
#include"scolarite.h"
//Définition
void Passeoured(float n)
{
  //Lexique local (vide ici)
  //Algorithme local
  if(n>=10)
  {
    cout<<"passage autorisé"<<endl;
  }
  else
  {
    cout<<"redoublement"<<endl;
  }
}
```

main.cpp

```
/*Fichiers contenant les déclarations
de passeoured et de cin, cout*/
#include<iostream>
using namespace std;
#include"scolarite.h"
//fonction principale
int main()
{
  //Lexique principal
  float note;
  //Algorithme principal
  cout<<"Saisissez une note: ";
  cin>>note;
  Passeoured(note);
  return 0;
}
```

Cas particulier : le Sinon est facultatif

Canevas

Lexique :

...

Algorithme :

Début

```
    ...
    Si condition
      Alors
        instruction01
        instruction02
        ...
    FinSi
    ...
```

Fin

```
//Lexique:
...

//Algorithme:
//Début
    ...
    if (condition)
    { //Alors
        instruction01;
        instruction02;
    }//FinSi
    ...
//Fin
```

Exemple

- Traduire l'algorithme pour la fonction dont le rôle était d'afficher le message "Risque de Verglas" si la température passée en paramètre est strictement inférieure à 2°C.
- Proposer la traduction de l'algorithme principal qui a permis de tester cette fonction.

Exemple (correction)

temperature.h

```
//Déclaration
void Verglas(float);
```

temperature.cpp

```
#include<iostream> //Car cout est utilisé
using namespace std;
//Fichier contenant la déclaration de verglas
#include"temperature.h"
//Définition
void Verglas(float t)
{
  //Lexique local (vide ici)
  //Algorithme local
  if(t<2)
  {
    cout<<"Risque de Verglas!"<<endl;
  }
}
```

main.cpp

```
/*Fichiers contenant les déclarations
de verglas et de cin, cout*/
#include<iostream>
using namespace std;
#include"temperature.h"
//fonction principale
int main()
{
  //Lexique principal
  float temp;
  //Algorithme principal
  cout<<"Saisissez une température: ";
  cin>>temp;
  Verglas(temp);
  return 0;
}
```

Instruction Selon ...autrement : le modèle

Lexique :

...

Algorithme :

Début

```
    ...
    Selon var

      val1 : instr11
             instr12
             ...

      val2 : instr21
             instr22
             ...

      ...
      autrement : instrA1
                  instrA2
    FinSelon
    ...
```

Fin

```
//Lexique:
...
//Algorithme:
//Début
    ...
    switch(var)
    {
      case val1: instr11;
                 instr12;
                 ...
                 break;
      case val2: instr21;
                 instr22;
                 ...
                 break;
      ...
      default: instrA1;
               instrA2;
    }
   ...
//Fin
```

Principe de fonctionnement

- On teste d'abord si la variable `var` a pour valeur `val1`
  Si c'est le cas, on effectue l'ensemble des instructions correspondant à ce cas puis on quitte la structure `switch` grâce à l'instruction `break;` qui est **indispensable**
- Dans le cas contraire, on passe à `val2` et on procède de la même manière.
- etc ...
- Si aucune valeur ne correspond aux cas prévus, on entre dans le cas `default`. Ce cas étant présent à la fin de la structure `switch`, l'instruction `break;` n'est pas nécessaire.

Très important !

- Si l'instruction `break;` est omise dans un des cas, toutes les instructions suivantes correspondant aux autres valeurs possibles pour la variable seront exécutée, même si la variable est différente de ces valeurs. En particulier, celles du cas `default` seront aussi exécutées...
- La variable `var` doit forcément avoir un type **entier** ou **caractère**.

Exemple

- On reprend l'exemple des grades/appréciation avec un message d'erreur si la saisie du grade n'est pas correcte. Traduire l'algorithme de la fonction `apprec` et l'algorithme principal en C.

Exemple (correction)

scolarite.h

```
//Déclaration
void Apprec(char);
```

scolarite.cpp

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

/*Fichier contenant la
déclaration de apprec*/
#include"scolarite.h"

//Définition
void Apprec(char g)
{
  //Lexique local (vide ici)
  //Algorithme local
  switch(g)
  {
    case 'A': cout<<"Excellent"<<endl;
              break;
    case 'B': cout<<"Bien"<<endl;
              break;
    case 'C': cout<<"Assez Bien"<<endl;
              break;
    case 'D': cout<<"Passable"<<endl;
              break;
    case 'E': cout<<"Insuffisant"<<endl;
              break;
    case 'F': cout<<"Inacceptable"<<endl;
              break;
    default: cout<<"Erreur de saisie"<<endl;
  }
}
```

main.cpp

```
/*Fichiers contenant les déclarations
de apprec et de cin, cout*/
#include<iostream>
using namespace std;
#include"scolarite.h"
//fonction principale
int main()
{
  //Lexique principal
  char grade;
  //Algorithme principal
  cout<<"Saisissez un grade entre A et F: ";
  cin>>grade;
  Apprec(grade);
  return 0;
}
```

Quelques erreurs classiques à trouver et à éviter

- Erreurs avec l'instruction `if`

Ce que le programmeur a écrit	Ce qu'il aurait dû écrire
`if(a>b) ;` `  a=b;`	

Ce que le programmeur a écrit	Ce qu'il aurait dû écrire

`if(a>b)`
`    if(x>y) x=y;`
`else`
`    ...`	

- Erreur avec l'instruction `switch`

Ce que le programmeur a écrit	Ce qu'il aurait dû écrire

`switch(a)`
`{`
`  case 1: b=2;`
`  default: b=3;`
`}`	

Sommaire

Motivations

- Contrôle sur la saisie de l'utilisateur
  - note entre 0 et 20
  - mois entre 1 et 12
  - rayon>0
  - ...

  Idée : test avec une structure Si
  Problème : réalisé seulement une fois.
- Répétition d'une tâche donnée
  - Calculs répétitifs
    p.ex : afficher les carrés des 50 premiers entiers, ou encore calculer les termes successifs d'une suite définie par récurrence.
  - Plus tard : parcours et manipulation d'un tableau ou d'une chaîne de caractères.

La boucle Tant que

Structure générale d'une boucle Tant que

Lexique :

...

Algorithme :

Début

:

{une ou plusieurs actions d'initialisation}

Tant que condition

instruction01

instruction02

:

{Modification de la condition}

FinTantque

:

Fin

Fonctionnement de la boucle

- Action(s) d'initialisation : leur rôle est de déterminer une valeur booléenne (VRAI ou FAUX) initiale pour `condition`.
  - Si cette valeur initiale est FAUX, on passe directement au **FinTantque**. Dans ce cas, les instructions de la boucle ne sont pas du tout exécutées.
  - Si cette valeur initiale est VRAI, on entre dans la boucle
- Entrée dans la boucle (cas où `condition`=VRAI) :
  - Les instructions sont exécutées **séquentiellement**
  - Parmis ces instructions, une instruction **doit** modifier la `condition` afin qu'elle soit de nouveau évaluée.
    (elle peut rester VRAI ou devenir FAUX)
  - Une fois **toutes** les instructions de la boucle exécutées, `condition` est de nouveau évaluée.
    - Si elle est VRAI, on recommence toutes les instructions au début de la boucle
    - Si elle est FAUX, on passe au **FinTantque** ce qui correspond à une sortie de la boucle.

Un premier exemple

Affichage des 50 premiers entiers naturels

Lexique :

i : 1 entier

Algorithme :

Début

    i⟵0
    Tant que i<50
      Ecrire("entier n° : ",i)
      i⟵i+1
    FinTantque

Fin

- "Dérouler" l'algorithme pour les premières valeurs de `i`
- Quel est le dernier entier affiché ?
- Que peut-on modifier pour afficher les entiers de 0 à 50 ?
- Que faut-il changer pour afficher les 50 premiers entiers pairs ?

Exemple :

Que fait cet algorithme ?

Lexique :

s, x : 2 réels i : 1 entier

Algorithme :

```
Début
    s⟵0
    i⟵0
    Tant que i<6
      Ecrire("Saisir un réel")
      Lire(x)
      s⟵s+x
      i⟵i+1
      Ecrire("s=",s)
    FinTantque
Fin
```

Expliquer ce qui se passerait si on supprimait :

- l'instruction i⟵0
- l'instruction i⟵i+1
- l'instruction s⟵0
- l'instruction s⟵s+x

Les deux instructions Ecrire() vous semblent-elles bien placées ?

Exercice (à chercher)

1. Proposer l'algorithme d'une fonction dont le rôle est de calculer et de retourner la somme des `n` premiers entiers naturels, `n` étant le paramètre d'entrée de la fonction.
2. Proposer l'algorithme principal permettant d'appeler cette fonction pour la tester.

La boucle Faire...Tant que

Structure générale d'une boucle Faire...Tant que

Lexique :

...

Algorithme :

Début

:

{action(s) d'initialisation (optionnelle(s))}

Faire

instruction01

instruction02

:

{Modification de la condition}

Tant que condition

FinFaireTantque

:

Fin

Fonctionnement de la boucle

- Action(s) d'initialisation : optionnelle(s) ici.
  S'il n'y en a pas, on entre tout de même dans la boucle.

Différence fondamentale avec la boucle `Tant que`

Contrairement à la boucle `Tant que`, la boucle `Faire ...Tant que` est exécutée **au moins une fois**.

- Entrée dans la boucle (au moins une fois) :
  - Les instructions sont exécutées séquentiellement
  - Parmis ces instructions, une instruction **doit** modifier la `condition` afin qu'elle soit évaluée. (elle peut être VRAI ou FAUX)
  - Une fois **toutes** les instructions de la boucle exécutées, `condition` est évaluée.
    - Si elle est VRAI, on recommence toutes les instructions au début de la boucle
    - Si elle est FAUX, on passe au **FinFaireTantque** ce qui correspond à une sortie de la boucle.
- Principale utilité de cette boucle : saisie avec vérification.

Exemple : saisie de note entre 0 et 20

Proposer l'algorithme d'une fonction dont le rôle est de faire saisir une note à l'utilisateur en contrôlant qu'elle est bien entre 0 et 20.

Avec une boucle `Faire...Tant que`

Lexique {Principal} :

`Saisie_note_0_20` : la fonction(vide)→ 1 réel

Lexique {local} :

note : 1 réel

Algorithme {local} :

```
Début
   Faire
      Ecrire("Saisir une note entre 0 et 20")
      Lire(note)
    Tant que note<0 OU note>20
   FinFaireTantque
   Retourner note
Fin
```

Exemple : saisie de note entre 0 et 20

Même exercice avec une boucle `Tant que`

Lexique {Principal} :

`Saisie_note_0_20` : la fonction(vide)→ 1 réel

Lexique {local} :

note : 1 réel

Algorithme {local} :

```
Début
  Ecrire("Saisir une note entre 0 et 20")
  Lire(note) {initialisation}
  Tant que note<0 OU note>20
    Ecrire("Erreur de saisie")
    Ecrire("Merci de saisir une note entre 0 et 20")
    Lire(note) {Modification de la condition}
  FinTantque
  Retourner note
Fin
```

Exemple : saisie de note entre 0 et 20

Même exercice avec une boucle `Tant que`

Lexique {Principal} :
`Saisie_note_0_20` : la fonction(vide)→ 1 réel
  Lexique {local} :
  note : 1 réel

  Algorithme {local} :
  Début
    Ecrire("Saisir une note entre 0 et 20") {Demande initiale}
    Lire(note) {initialisation}
    Tant que note<0 OU note>20
      Ecrire("Erreur de saisie") {Message d'erreur}
      Ecrire("Merci de saisir une note entre 0 et 20")
      Lire(note) {Modification de la condition}
    FinTantque
    Retourner note
  Fin

Utilité : distinguer la Demande initiale de l'erreur de saisie.

Pièges à éviter

- Une boucle, il faut en sortir ! Voici deux exemples de ce qu'il faut éviter [20]. Expliquer pourquoi.

Exemple1

```
val1←—2
val2←—3
Tant que val1<10
    val2←—val2*val1
FinTantque
```

Exemple2 : affichage des 10 premiers entiers naturels impairs (2 erreurs)

```
i←—0
Faire
    n←—2*i+1
    Ecrire("Le ",i+1,"e entier naturel impair est : ", n)
    i←—i+1
    Tant que n≠10
FinFaireTantque
```

20. on suppose que les lexiques de ces deux algorithmes sont corrects.

De l'énoncé au choix de la bonne boucle

Exercices (à chercher)

Trouver la boucle adaptée à la résolution de chacun des problèmes suivants :

- Problème 1 : Afficher le carré des valeurs saisies tant qu'on ne saisit pas la valeur 0
- Problème 2 : Saisir des données et s'arrêter dès que leur somme dépasse 500
- Problème 3 : Saisir des données tant que leur somme ne dépasse pas un seuil donné.

Sommaire

Motivation

- Contrôle sur la saisie de l'utilisateur
  - note entre 0 et 20
  - mois entre 1 et 12
  - rayon>0
  - ...

  Idée : test avec une structure Si
  Problème : réalisé seulement une fois.
- Répétition d'une tâche donnée
  - Calculs répétitifs
    p.ex : afficher les carrés des 50 premiers entiers
    calculs sur les suites définies par récurrence, etc...
  - Plus tard : parcours et manipulation d'un tableau ou chaîne de caractères.

La boucle while

Canevas

Lexique :

...

Algorithme :

Début

...

{Initialisation}

Tant que condition

instruction01

instruction02

...

{Modification de la condition}

FinTantque

...

Fin

```
//Lexique:
...

//Algorithme:
//Début
    ...
    //Initialisation
    while (condition)
    {
        instruction01;
        instruction02;
        ...
        /*modification
        de la condition*/
    }//FinTantque
    ...
//Fin
```

Exercices (à chercher)

Proposer la traduction de l'exercice suivant (fait dans le chapitre précédent)

1. Proposer l'algorithme d'une fonction dont le rôle est de calculer et de retourner la somme des `n` premiers entiers naturels, `n` étant le paramètre d'entrée de la fonction.
2. Proposer l'algorithme principal permettant d'appeler cette fonction pour la tester.

Autre façon de traduire le Tant que : la boucle for

Canevas

for($\underbrace{\quad}_{\textit{initialisation}}$; $\underbrace{\quad}_{\textit{condition}}$; $\underbrace{\quad}_{\textit{modification}}$ **)**

```
{
    instruction01;
    instruction02;
    ...
}//FinTantque
```

Utilité : lorsque le nombre d'itérations est connu à l'avance

```
/*Afficher les carrés des 100 premiers entiers naturels:*/
unsigned int i;
for(i=0;i<100;i=i+1)
{
   cout«i*i;
}
```

Incrémentation et boucle `for`

Remarques importantes

- L'instruction `i=i+1;` peut également se traduire : `i++;`
- Il est possible de déclarer la variable de boucle `i` directement dans le `for` :

```
for(unsigned int i=0;i<100;i++)
{
   cout<<i*i<<endl;
}
```

  Dans ce cas la variable `i` n'existe que dans le `for`, on dira que la **portée** de la variable `i` est locale à la boucle `for`.
  Une instruction du type `i=0;` en dehors du `for` renverra un message d'erreur comme quoi la variable `i` n'a pas été déclarée.
- Bien que toutes les boucles `Tant que` puisse se traduire sous la forme d'un `for`, on utilisera cette structure principalement lorsque le nombre d'itérations est connu à l'avance. (Manipulation d'un tableau principalement).

La boucle do...while

Canevas

Lexique :

...

Algorithme :

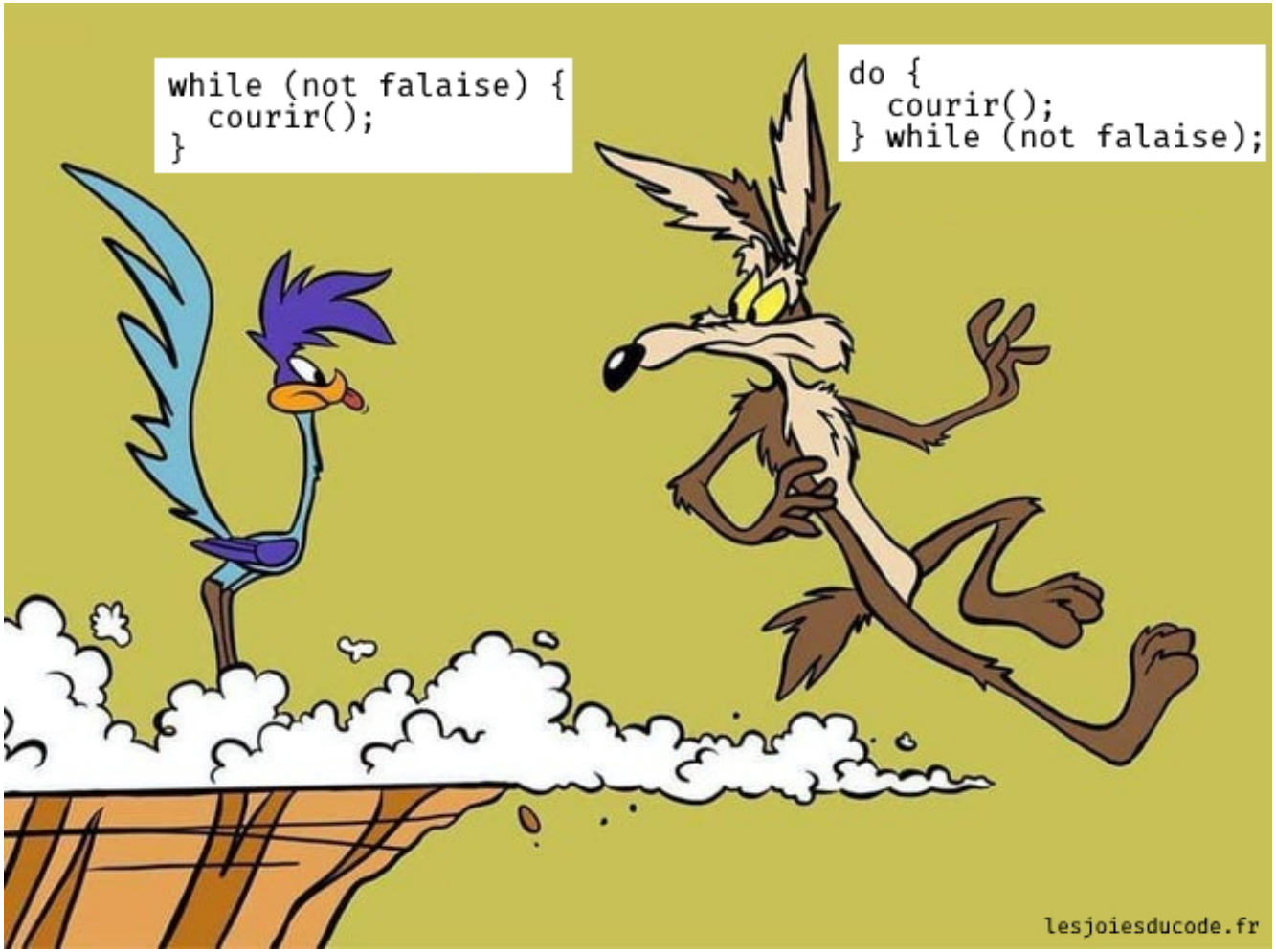
Début

```
    ...
    {Initialisation optionnelle}
    Faire
        instruction01
        instruction02
        ...
        {modification
        de la condition}
        Tantque condition
    FinFaireTantque
    ...
```

Fin

```
//Lexique:
...

//Algorithme:
//Début
    ...
    //Initialisation opt.
    do
    {
        instruction01;
        instruction02;
        ...
        /*modification
        de la condition*/
    }while(condition);
    ...
//Fin
```

Différence fondamentale entre les deux boucles



Merci à Alexandre Basset (CiTiSE 2016-2019)

Exemple : saisie de note entre 0 et 20

Traduisez l'algorithme définissant la fonction `Saisie_note_0_20` en C.

Avec une boucle `Faire...Tant que`

Lexique {Principal} :

`Saisie_note_0_20` : la fonction(vide)→ 1 réel

Lexique {local} :

note : 1 réel

Algorithme {local} :

Début

Faire

Ecrire("Saisir une note entre 0 et 20")

Lire(note)

Tant que note<0 OU note>20

FinFaireTantque

Retourner note

Fin

Traduction avec exemple d'utilisation dans la fonction main()

mes_fonctions.h

```
/*
R: demande à l'utilisateur de saisir une note entre 0 et 20
et lui repose la question tant qu'elle n'est pas dans le bon rang
E: vide
S: 1 réel correspondant à la note entre 0 et 20
*/
float Saisie_note_0_20(void);
```

mes_fonctions.cpp

```
#include<iostream>
using namespace std;
#include"mes_fonctions.h"
float Saisie_note_0_20(void)
{
  float note;
  do
  {
    cout<<"Saisir une note entre 0 et 20"<<endl;
    cin>>note;
  }while(note<0 || note>20);//Ne pas oublier le ; ici
  return note;
}
```

main.cpp

```
#include<iostream>
using namespace std;
#include"mes_fonctions.h"

int main()
{
  float note1;
  note1=Saisie_note_0_20();
  cout<<note1<<" est bien entre 0 et 20"<<endl;
  return 0;
}
```

Quelques erreurs de débutants...

- Deux exemples de boucles infinies :

```
unsigned int i;
i=100;
while(i>=0)
{
  i--;
}
```

```
unsigned int i;
i=0;
while(i<100);
{
  i++;
}
```

- ...à comprendre et à corriger

Sommaire

Introduction

- Comment mémoriser dans un même "objet" plusieurs informations de **même type** ?

Mémoriser les 6 notes de maths d'un étudiant

Lexique :
n_math1 : 1 réel
n_math2 : 1 réel
n_math3 : 1 réel
n_math4 : 1 réel
n_math5 : 1 réel
n_math6 : 1 réel

{Très, très mauvais}

Lexique :
n_math1, n_math2, n_math3,
n_math4, n_math5, n_math6 : 6 réel

{un peu mieux mais lourd à manipuler}
{Comment faire pour mémoriser 500 variables de même type ? ?}

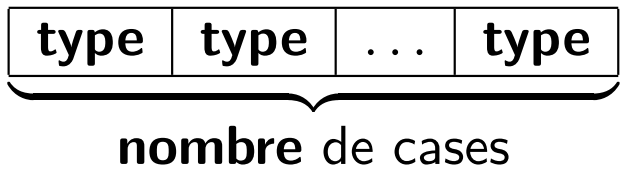
Tableaux : définition

- Pour mémoriser plusieurs informations de **même type** on utilise une **structure de données** appelée **tableau**.

Définition

Un **tableau** est une structure de données linéaire qui permet de **mémoriser/stocker** des données de **même type**.

- Il faut préciser deux informations importantes lorsqu'on veut utiliser un tableau :
  1. le **nombre** d'éléments à mémoriser
  2. le **type** de ces éléments [21]
- Représentation/Visualisation pratique d'un tableau :



Chaque case contient une information. Toutes les informations ont le même type.

21. Rappel : ce type doit être **le même** pour tous les éléments

Déclaration d'un tableau

- La déclaration d'un tableau se fait, comme toutes les déclarations, dans le lexique.

Modèle

Lexique :

`nom_tab` : 1 tableau de `TAILLE` `nom_type`

Déclaration d'un tableau permettant de stocker 48 notes de maths

Lexique :

$\underbrace{\texttt{notes\_maths}}_{\texttt{nom\_tab}}$: 1 tableau de $\underbrace{48}_{\texttt{TAILLE}}$ $\underbrace{\text{réel}}_{\texttt{nom\_type}}$

Recommandations

- Le nom du tableau devrait décrire ce que le tableau représente.
- La taille du tableau est **fixée une fois pour toute** et ne peut plus changer. Elle est donc souvent représentée par une **constante**.

Initialisation(s) d'un tableau

- Lors de la **déclaration** d'un tableau, on peut lui donner un ensemble de valeurs initiales. Cette action s'appelle **initialisation** du tableau.

Exemple : nombre de jours par mois

Lexique :

`nb_jours_mois` : 1 tableau de 12 entier :={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}

- Il est possible d'initialiser seulement les premiers éléments d'un tableau. Dans ce cas, les autres "cases" du tableau existent MAIS devront, comme toutes variables, être initialisées avant d'être utilisées.

Exemple : relevé de températures (3 premiers jours d'un mois)

Lexique :

`temperature_mensuelle` : 1 tableau de 31 réel :={5.5, 8, 15}

Initialisation d'un tableau constant

- Dans le cas où on ne veut pas modifier la valeur initiale du tableau, on peut déclarer ce tableau comme étant une constante.
- Il est alors **nécessaire** de l'initialiser.
- Ceci est important puisque les données contenues dans le tableau ne pourront pas être modifiées par la suite, elles sont en quelque sorte "protégées en écriture". (Voir la section sur les fonctions et tableaux)

Exemple : révolution des planètes du système solaire de la plus proche à la plus éloignée du soleil, en année terrestre

Lexique :

`REVOL_PLANETES` : La constante tableau de 8 réels :={0.2463169349, 0.6151831773, 1, 1.002509648, 11.8619682757, 29.4572122412, 84.019164585, 164.7674047379}

Notation indicielle

- Une fois le tableau déclaré, on connaît :
  - son nom : `nom_tableau`
  - son nombre de cases (taille du tableau) : `TAILLE`
  - le type des éléments stockés dans le tableau
- Les cases sont numérotées de 0 à `TAILLE` -1
- Chaque case du tableau peut être vue et utilisée comme une variable.
- Accès à la case d'indice `i` (entre 0 et `TAILLE` -1), on utilisera la notation **indicielle** générale suivante : `nom_tableau[i]`

Exemple : Quelles sont les instructions erronnées ?

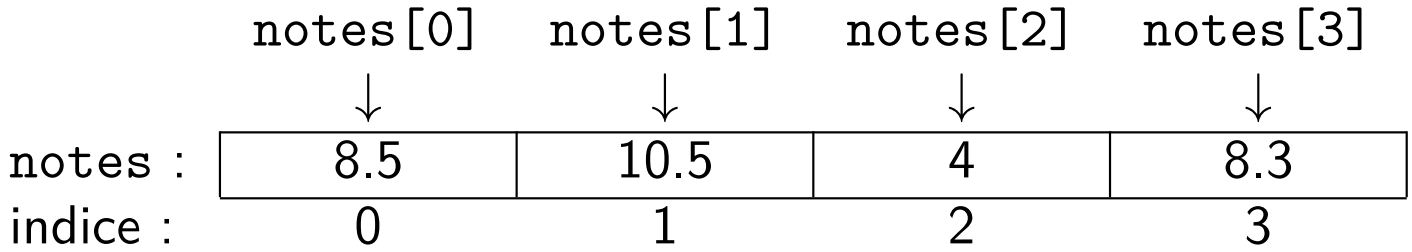
Lexique :
temp : 1 tableau de 10 réel

Algorithme :
Début
...

{1} tab[0]$\longleftarrow$ 4.5
{2} temp[0]$\longleftarrow$ -3.4
{3} temp[1]$\longleftarrow$ 4.5+temp[0]
{4} temp[9]$\longleftarrow$ $-$temp[1]
{5} temp[10]$\longleftarrow$8.4
...
Fin

Variable indice liée à un tableau

- Soit un tableau à 4 éléments de type réel nommé `notes`.



- L'indice, variant ici entre 0 et 3, permet de repérer chaque case du tableau.
- La plupart du temps, on déclarera l'indice associé à un tableau dans le lexique. Cet indice sera une **variable** de type **entier**.

Exemple

Lexique :

`notes_maths` : 1 tableau de 48 réel

`i` : 1 entier {indice du tableau variant ici de 0 à 47}

Remarque importante

Si un tableau `tab` a une taille `T`, il est **interdit** d'accéder aux cases `tab[i]` pour `i`< 0 ou `i`$\geq T$.

Exercices [Ch]

Consulter l'exercice 10.1 du fascicule.

Copie d'un tableau dans un autre

- Il peut arriver qu'on souhaite copier l'intégralité d'un tableau dans un autre.

Première (mauvaise) idée

Supposons qu'on ait un extrait du Lexique suivant :
tab1,tab2 : 2 tableaux de 4 entier
Il serait tentant dans l'algorithme d'avoir une instruction du type :
tab2$\leftarrow$ tab1
Le problème est qu'on ne peut pas **affecter globalement** un tableau dans un autre. Cette instruction est donc **invalide**.

- La **seule façon** de procéder est d'effectuer la copie **élément par élément** grâce à l'opération d'affectation sur des variables de même type.
- On utilisera une structure itérative pour effectuer la copie élément par élément tant qu'on n'est pas à la fin du tableau [22].

22. voir section 4 sur les fonctions et tableaux

Affichage des éléments d'un tableau

- Il peut arriver qu'on souhaite afficher l'intégralité d'un tableau.

Première (mauvaise) idée

Supposons qu'on ait un extrait du Lexique suivant :
tab1 : 1 tableau de 6 réels :={14.5, 15, 19.3, 9.7, 14.7, 13.8}
Il serait tentant dans l'algorithme d'avoir une instruction du type :
Ecrire(tab1)
Le problème est qu'on ne peut pas **afficher globalement** un tableau.
Cette instruction est donc **invalide**.

- La **seule façon** de procéder est d'effectuer l'affichage **élément par élément** grâce à l'instruction Ecrire(tab1[i]) pour chaque élément du tableau (Ici pour i allant de 0 à 5).
- On utilisera une structure itérative pour effectuer l'affichage du tableau élément par élément [23].

23. voir section 4 sur les fonctions et tableaux

Recommandations sur le bon usage d'un tableau

- Les accès aux éléments d'un tableau se font grâce à une **variable indice** de type **entier** qui varie entre **0** et **la taille du tableau moins 1**.
  On utilise la **notation indicielle** : `nom_tableau[i]`
- Pour affecter une valeur à une case d'un tableau on (lire "le programmeur") doit toujours vérifier :
  1. si c'est une case valide (indice dans le bon rang)
  2. si le type de la valeur est le même que le type des éléments du tableau. (respect des types)
- Ne pas confondre l'**initialisation** qui est une opération autorisée (seulement dans le lexique) avec l'**affectation globale** d'un tableau dans un autre qui est **strictement interdite**
- La **copie** d'un tableau dans un autre ne peut se faire que **élément par élément**.
- L'affichage d'un tableau **en globalité** n'est pas possible, il faut également l'afficher **élément par élément**.

Usage des tableaux dans les fonctions

- Il peut être intéressant d'avoir des fonctions permettant de manipuler des tableaux (Saisie et mémorisation de plusieurs informations, calculs de moyenne, affichage des éléments du tableau, recherche d'un minimum/maximum, tri d'un tableau, etc. . .)
- Une fonction peut tout à fait recevoir comme paramètres un ou plusieurs tableaux et accéder à leurs élements comme elle le ferait avec n'importe quel autre paramètre.
- Un premier point nouveau à retenir est le suivant :

Type de retour de la fonction

Une fonction ne peut **JAMAIS** retourner un tableau. Elle peut éventuellement retourner un de ses éléments, ou encore le résultat d'un calcul fait sur ses éléments (moyenne par exemple), mais jamais un tableau globalement.

- Nous verrons par la suite deux moyens (paramètres d'entrée/sortie et pointeurs) pour une fonction de "récupérer" les informations contenues dans le tableau.

Taille réelle et taille pratique du tableau

- Rappel : La taille **réelle** (c'est à dire la place réelle qu'il occupe en mémoire) d'un tableau est fixée une fois pour toute et ne peut plus changer.
  $\Rightarrow$ le tableau est souvent surdimensionné... et contient en général plus de cases que de valeurs utiles.
- Le nombre de valeurs utiles qu'il contient sera appelé sa **taille pratique**. Cette taille pratique sera utilisée dans les fonctions permettant de manipuler les tableaux.
- L'intérêt est d'optimiser le temps de parcours d'un tableau en ne parcourant que les cases "utiles" et pas tout le tableau.
- Cela permet également de ne pas accéder à des cases qui ne sont pas initialisées.

Exemple

Lexique :
tab : 1 tableau de 30 réels :={2.3,-6.5,3.4,-10.4}
{La taille réelle du tableau est de 30 alors que sa taille pratique est de 4.
Il n'est donc pas nécessaire de parcourir les 30 cases à chaque fois, mais seulement les (ici 4) cases renseignées.
Gain ici : $\approx$ 86.7% du temps de traitement (26/30).}

Paramètres d'entrée/sortie

Rappels :

- Nous avons vu l'intérêt de Rôle, Entrées, Sortie d'une fonction
- Nous avons vu que les paramètres d'entrées d'une fonction sont des copies de variables déclarées dans le lexique principal (voir slide 49)
- Que se passe-t-il si une fonction travaille sur un tableau de 10000 éléments ?

Passage par adresse

- Les tableaux sont très souvent de grande taille : la copie élément par élément prendrait énormément de temps lors d'un appel à une fonction travaillant sur un tableau.
- Les tableaux sont donc **TOUJOURS** passés **par adresse** à la fonction [a], c'est à dire que la fonction travaille **directement** sur le tableau qui a été déclaré dans le lexique principal.
- La fonction peut donc modifier les valeurs du tableau, le remplir, etc. . .**ET** cette modification est prise en compte dans l'algorithme principal (ou plus généralement dans la fonction appelante) ce qui peut être dangereux si on ne souhaite que consulter les valeurs du tableau.
- Ceci nous amène à définir la notion de **paramètres d'entrée/sortie** pour les tableaux que la fonction modifiera. Ce n'est pas une entrée à proprement parler, ni une sortie puisque la sortie correspond au type retourné par la fonction ET ce type là ne peut **JAMAIS** être un tableau.

---

a. On communique à la fonction l'adresse du tableau. Derrière le nom du tableau qu'on donne à la fonction se cache donc son adresse en mémoire

Paramètres d'entrée/sortie et déclaration

- On ne précisera pas la taille **réelle** des tableaux dans les fonctions. En revanche, on passera un paramètre supplémentaire représentant la taille **pratique** du tableau. Comme dans l'exemple ci-dessous :

Déclaration d'une fonction permettant de copier un tableau de réels dans un autre

{R : Copier un tableau dans un autre élément par élément lorsque la copie est possible (=> les tailles réelles des tableaux ont été comparées avant l'appel de cette fonction)

E : 1 tableau de réels : le tableau à copier (tab_ini), 1 entier : le nombre d'éléments à copier (taille pratique)

E/S : **1 tableau de réels : la copie du tableau (tab_out)**

S : **vide**}

Copie_tab : la fonction(tab_ini : 1 tableau de réels, taille : 1 entier, tab_out : 1 tableau de réels)→ vide

- De même, proposer les prototypes avec R, E, E/S, S des fonctions suivantes :
  - Affichage d'un tableau élément par élément
  - Saisie de valeurs dans un tableau (voir exercice Exercice 10.2 du fascicule)

Les algorithmes de ces fonctions seront étudiés en TD.

Tableaux à plusieurs dimensions

- Un tableau peut contenir un certain nombre d'éléments de même type, ce type pouvant être lui aussi un tableau.
- On peut donc créer des tableaux de tableaux.

Déclaration et initialisation de tableaux à deux dimensions

- Le principe est que chaque ligne est un élément du tableau. Chaque ligne est donc un tableau de 10 réels (correspondant aux 10 colonnes)
  Lexique :
  matrice : 1 tableau de $\underbrace{6}_{\text{lignes}}$ tableau de $\underbrace{10}_{\text{colonnes}}$ réels
- Initialisation :
  matrice2 : 1 tableau de $\underbrace{3}_{\text{lignes}}$ tableau de $\underbrace{4}_{\text{colonnes}}$ entiers :={{2, 3, 5, 7}, {11, 13, 17, 19}, {23, 29, 31, 37}}

Tableaux à plusieurs dimensions et fonctions

- Utilisation dans les fonctions : on a vu qu'il n'était pas nécessaire de préciser le nombre d'éléments d'un tableau passé en paramètre d'une fonction.
- Pour les tableaux à deux (ou plus) dimensions, ceci n'est vrai **que** pour le nombre de lignes. Il **FAUT** préciser **impérativement** le nombre maximal de colonnes.
- En général, on aura également en paramètre les nombres pratiques de lignes et de colonnes.

Exemple : prototype d'une fonction d'affichage de tableau 2D

{R : Affiche les éléments d'un tableau de réels à deux dimensions, ligne par ligne

E : 1 tableau de tableau de 100 réels : le tableau à afficher avec un nombre de lignes quelconque et un nombre de colonnes limité à 100 (tab_2d)
1 entier correspondant au nombre pratique de lignes (nb_lignes)
1 entier correspondant au nombre pratique (≤ 100) de colonnes (nb_colonnes)

E/S : vide (la fonction ne modifie pas le tableau)

S : vide }

Affiche_tab2D : la fonction(tab2d : 1 tableau de tableau de 100 réels, nb_lignes : 1 entier, nb_colonnes : 1 entier)$\rightarrow$ vide

Chaînes de caractères et tableaux

- Le type `chaîne de caractère` est un type de base. Il a été utilisé jusque là pour afficher des messages (phrases à l'utilisateur) et comprend l'ensemble des caractères que l'on souhaite afficher.
- Une chaîne de caractère peut donc aussi être vue comme un **tableau de caractères**.

Chaine de caractères, tableau de caractères et caractère de fin de chaîne

- Ce tableau contient l'ensemble des caractères de la chaîne + un caractère spécial appelé caractère de fin de chaîne et noté `FCH`
- Ce caractère spécial (à déclarer dans le lexique) est fortement utilisé dans toutes les fonctions de manipulation des chaînes de caractères.
- Principe général : tant qu'on ne rencontre pas le caractère de fin de chaîne, on poursuit le traitement.

Exemple

mois : Un tableau de 10 caractères :={'s','e','p','t','e','m','b','r','e',FCH}
mois2 : Une chaîne de caractères :="septembre"
La chaîne de caractère "septembre" contenant 9 caractères peut donc être vue comme le tableau de **9+1=10** caractères.
L'accès aux éléments de la chaîne se fait comme pour les tableaux, en utilisant la notation indicielle : mois2[i] ou mois[i] représentent ici la même information.

Déclarations d'une chaîne de caractères, accès aux caractères

- La façon naturelle de déclarer une chaîne de caractères est la suivante :

Exemple : déclaration d'une chaîne de caractères

Lexique :

mois : 1 `chaîne de caractères` :="septembre"

- Actuellement, nous ne pouvons pas déclarer une chaîne sans l'initialiser. Il faudra aborder la notion de pointeur de caractères pour se rendre compte qu'une chaîne de caractères est en fait un pointeur et qu'il faut lui allouer de l'espace mémoire.
- Chaînes de caractères et tableaux de caractères sont liés, en particulier, l'accès aux éléments d'une chaîne peut donc se faire de la même façon que l'accès aux éléments d'un tableau : notation indicielle.

Limitations dans l'utilisation des chaînes de caractères

Instructions valides	Instructions invalides
Ecrire(mois) Ecrire(mois[3])	Lire(mois) {pas pratique quand on demande le nom de quelqu'un} Lire(mois[3])

- L'utilisation de tableaux de caractères (ayant une taille maximale fixée) sera bien plus pratique dans un premier temps.

Déclaration d'un tableau de caractères, lien avec les chaînes de caractères

Exemple : déclaration équivalente utilisant un tableau

Lexique :

mois : 1 tableau de 10 caractères :={'s', 'e', 'p', 't', 'e', 'm', 'b', 'r', 'e', FCH}

- Les instructions : Ecrire(mois[3]), Ecrire(mois[8]), Ecrire(mois[9]), Ecrire(mois) sont (comme précédemment) valides. Qu'affichent-elles ?
- Mais en plus :
  - Lire(mois), Lire(mois[3]) seront tout à fait acceptable (à condition qu'on n'excède pas le nombre de caractères maximal du tableau **en comptant le caractère de fin de chaîne**).
  - Attention, une instruction du type : mois←"octobre" reste interdite. Pourquoi ?

Attention ! Ne mélangeons pas tout !

- Un tableau de caractères (avec un caractère de fin de chaîne) peut être vu comme une chaîne de caractères mais offrant une souplesse d'utilisation plus importante.
- Ceci n'est vrai que pour les tableaux de caractères ! !
- Ecrire(tab1d), Lire(tab1d) pour des tableaux autre que des tableaux de caractères ne fonctionne toujours pas et est à proscrire !

Tableaux à plusieurs dimensions

- Un tableau peut aussi contenir des tableaux

Extrait de Lexique

mois : 1 tableau de 12 tableau de 10 caractère :=
{{‘j’,‘a’,‘n’,‘v’,‘i’,‘e’,‘r’,FCH},{‘f’,‘e’,‘v’,‘r’,‘i’,‘e’,‘r’,FCH},{‘m’,‘a’,‘r’,‘s’,FCH},
{‘a’,‘v’,‘r’,‘i’,‘l’,FCH},{‘m’,‘a’,‘i’,FCH},{‘j’,‘u’,‘i’,‘n’,FCH},
{‘j’,‘u’,‘i’,‘l’,‘l’,‘e’,‘t’,FCH},{‘a’,‘o’,‘û’,‘t’,FCH},
{‘s’,‘e’,‘p’,‘t’,‘e’,‘m’,‘b’,‘r’,‘e’,FCH},{‘o’,‘c’,‘t’,‘o’,‘b’,‘r’,‘e’,FCH},
{‘n’,‘o’,‘v’,‘e’,‘m’,‘b’,‘r’,‘e’,FCH},{‘d’,‘é’,‘c’,‘e’,‘m’,‘b’,‘r’,‘e’,FCH}}

- Accès aux éléments en deux temps
  1. mois[0] est le tableau de 10 caractère correspondant ici à {‘j’,‘a’,‘n’,‘v’,‘i’,‘e’,‘r’,FCH}
  2. mois[0][3] est le 4^e élément du tableau mois[0], c’est à dire, le caractère ‘v’.
- Exercice : à quoi correspondent les éléments suivants : mois[4], mois[5][3], mois[3][4], mois[8][7] ?
- On peut imaginer des tableaux de tableaux de tableaux,...

Autre façon de "voir" les tableaux à deux dimensions

- Forme matricielle (ligne/colonne)

i\ j	0	1	2	3	4	5	6	7	8	9
0	'j'	'a'	'n'	'v'	'i'	'e'	'r'	FCH		
1	'f'	'e'	'v'	'r'	'i'	'e'	'r'	FCH		
2	'm'	'a'	'r'	's'	FCH					
3	'a'	'v'	'r'	'i'	'l'	FCH				
4	'm'	'a'	'i'	FCH						
5	'j'	'u'	'i'	'n'	FCH					
6	'j'	'u'	'i'	'l'	'l'	'e'	't'	FCH		
7	'a'	'o'	'û'	't'	FCH					
8	's'	'e'	'p'	't'	'e'	'm'	'b'	'r'	'e'	FCH
9	'o'	'c'	't'	'o'	'b'	'r'	'e'	FCH		
10	'n'	'o'	'v'	'e'	'm'	'b'	'r'	'e'	FCH	
11	'd'	'é'	'c'	'e'	'm'	'b'	'r'	'e'	FCH	

- mois[i] permet de repérer le tableau correspondant à un des 12 mois puis mois[i][j] donne accès au $(j+1)^e$ caractère du $(i+1)^e$ mois de l'année.

Sommaire

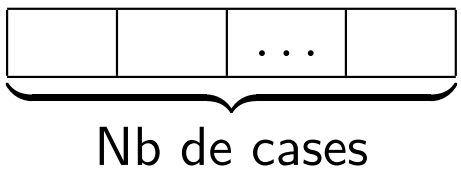
Tableau : définition

- Pour mémoriser plusieurs informations de **même type** on utilise une **structure de données** appelée **tableau**.

Définition

Un **tableau** est une structure de données linéaire qui permet de **mémoriser/stocker** des données de **même type**.

- Il faut préciser deux informations importantes lorsqu'on veut utiliser un tableau :
  1. le **nombre** d'éléments à mémoriser. Ce nombre est fixé une fois pour toute.
  2. le **type** de ces éléments [24]
- Représentation/Visualisation pratique d'un tableau :



24. Rappel : ce type doit être **le même** pour tous les éléments

Déclaration d'un tableau

- La déclaration d'un tableau se fait, comme toutes les déclarations, dans le lexique.

Modèle Algorithmique (déclaration dans le lexique)

`nom_tab` : 1 tableau de `TAILLE` `nom_type`

Rappel : la taille est fixée une fois pour toute et ne plus être modifiée !

Traduction en C

```
const unsigned int TAILLE=50;//initialisation à 50
nom_type nom_tab[TAILLE];
```

- Déclarer un tableau permettant de stocker 69 notes de maths.

Exemple : algorithmie et traduction

$\underbrace{\texttt{notes\_maths}}_{\texttt{nom\_tab}}$: 1 tableau de $\underbrace{69}_{\texttt{taille}}$ $\underbrace{\text{réel}}_{\texttt{nom\_type}}$

```
float notes_maths[69];
```

Remarques importantes sur la déclaration d'un tableau

- Un tableau a forcément une taille de type entier (positif)
- Un tableau a forcément une taille constante définie à la déclaration :

```
unsigned int n=10;
int tab1[n];
```

n'est pas acceptable. Il faut écrire :

```
const unsigned int N=10;
int tab1[N];
//ou plus simplement
int tab1[10];
```

Initialisation(s) d'un tableau

- Lors de la **déclaration** d'un tableau, on peut lui donner un ensemble de valeurs initiales. Cette action s'appelle **initialisation** du tableau.

Exemple : nombre de jours par mois

```
//Lexique
  unsigned int nb_jours_mois[12]={31, 28, 31, 30, 31, 30, 31, 31, 30,
31, 30, 31};
```

- Il est possible d'initialiser seulement les premiers éléments d'un tableau. Dans ce cas, les autres "cases" du tableau devront, comme toutes variables, être initialisées avant d'être utilisées.

Exemple : relevé de températures (3 premiers jours d'un mois)

```
//Lexique
  float temperatures_mensuelles[31]={5.5, 8, 15};
```

Initialisation d'un tableau constant

- Dans le cas où on ne veut pas modifier la valeur initiale du tableau, on peut déclarer ce tableau comme étant une constante.
- Il est alors **nécessaire** de l'initialiser.
- Ceci est important puisque les données contenues dans le tableau ne pourront pas être modifiées par la suite, elles sont en quelque sorte "protégées en écriture". (Voir la section sur les fonctions et tableaux)

Exemple : révolution des planètes du système solaire de la plus proche à la plus éloignée du soleil, en années terrestres

```
//Lexique
  const float REVOL_PLANETES[8]={0.2463169349, 0.6151831773, 1,
1.002509648, 11.8619682757, 29.4572122412, 84.019164585,
164.7674047379};
//Une instruction du type
REVOL_PLANETES[0]=0.25; //sera interdite
```

Notation indicielle

Exemple : corriger l'algorithme et le code C suivant

Lexique :

temp : 1 tableau de 10 réel

Algorithme :

Début

...

{1} tab[0]⟵ 4.5
{2} temp[0]⟵ −3.4
{3} temp[1]⟵ 4.5+temp[0]
{4} temp[9]⟵ −temp[1]
{5} temp[10]⟵8.4
...
Fin

```
int main()
{
 float temp[10];
tab[0]=4.5;
 temp[0]=-3.4;
 temp[1]=4.5+temp[0];
 temp[9]=-temp[1];
 temp[10]=8.4;
 return 0;
}
```

Variable indice liée à un tableau

- Cet indice (vu en algo) sera une **variable** de type **entier** (positif).
- Cet indice doit varier entre 0 et la taille du tableau -1.

Remarques importantes

Si un tableau `tab` a une taille `TAILLE`, il est **interdit** d'accéder aux cases `tab[i]` pour `i`<0 ou `i`$\geq$ *TAILLE*.
Attention : le programme peut tout à fait vous autoriser à accéder à une case qui n'est pas dans le tableau (si la mémoire est libre).
Conclusion : c'est au **programmeur** de veiller à ce que l'indice soit dans le bon rang.

Exemple

```
float notes_maths[69];
unsigned int i;
//indice du tableau devant varier entre 0 à 68
```

Exercices [Ch]

Traduire en C l'exercice corrigé 10.1 du fascicule.

Recommandations sur le bon usage d'un tableau

- Les accès aux éléments d'un tableau se font grâce à une **variable indice** de type **entier** qui varie entre **0** et **la taille du tableau moins 1**.
  On utilise la **notation indicielle** : `nom_tableau[i]`
- Pour affecter une valeur à une case d'un tableau on doit toujours vérifier :
  1. si c'est une case valide (indice dans le bon rang)
  2. si le type de la valeur est le même que le type des éléments du tableau. (respect des types)
- Ne pas confondre l'**initialisation** qui est une opération autorisée (seulement dans le lexique) avec l'**affectation globale** d'un tableau dans un autre qui est **strictement interdite**
- La **copie** d'un tableau dans un autre ne peut se faire que **élément par élément**.
- L'affichage d'un tableau **en globalité** n'est pas possible, il faut également l'afficher **élément par élément**.

Usage des tableaux dans les fonctions

- Il peut être intéressant d'avoir des fonctions permettant de manipuler des tableaux (Saisie et mémorisation de plusieurs informations, calculs de moyenne, affichage des éléments du tableau, recherche d'un minimum/maximum, tri d'un tableau, etc. . .)
- Une fonction peut tout à fait recevoir comme paramètres un ou plusieurs tableaux et accéder à leurs élements comme elle le ferait avec n'importe quel autre paramètre.
- On rappelle un point très important à retenir (si ce n'est déjà fait...) :

Type de retour de la fonction

Une fonction ne peut **JAMAIS** retourner un tableau. Elle peut éventuellement retourner un de ses éléments, ou encore le résultat d'un calcul fait sur ses éléments (moyenne par exemple), mais jamais un tableau globalement.

- Pour "retourner" un tableau, nous avons vu la notion de paramètres d'entrées/sortie qui permet à la fonction de modifier un tableau qui lui est passé en paramètres.

Rappels : Taille réelle et taille pratique du tableau

- La taille **réelle** (c'est à dire la place réelle qu'il occupe en mémoire) d'un tableau est fixée une fois pour toute et ne peut plus changer.
  $\Rightarrow$ le tableau est souvent surdimensionné... et contient en général plus de cases que de valeurs utiles.
- Le nombre de valeurs utiles qu'il contient sera appelé sa **taille pratique**. Cette taille pratique sera utilisée dans les fonctions permettant de manipuler les tableaux.
- L'intérêt est d'optimiser le temps de parcours d'un tableau en ne parcourant que les cases "utiles" et pas tout le tableau.

Exemple

Lexique :

```
  float tab[30]={2.3,-6.5,3.4,-10.4}
/*La taille réelle du tableau est de 30 alors que sa taille
pratique est de 4.
Il n'est donc pas nécessaire de parcourir les 30 cases à chaque
fois, mais seulement les (ici 4) cases renseignées.
Gain: ≈ 86.7% du temps de traitement (26/30).}*/
```

Paramètres d'entrée/sortie

Rappels :

- Nous avons vu l'intérêt de Rôle, Entrées, Sortie d'une fonction
- Nous avons vu que les paramètres d'entrées d'une fonction sont des copies de variables déclarées dans le lexique principal (voir slide 49)
- En revanche, nous avons vu dans le chapitre précédent qu'un tableau est souvent volumineux et est passé à la fonction par adresse.

Rappel : passage par adresse

- Les tableaux sont très souvent de grande taille : la copie élément par élément prendrait énormément de temps lors d'un appel à une fonction travaillant sur un tableau.
- Les tableaux sont donc **TOUJOURS** passés **par adresse** à la fonction, c'est à dire que la fonction travaille **directement** sur le tableau qui a été déclaré dans le lexique principal.
- La fonction peut donc modifier les valeurs du tableau, le remplir, etc...**ET** cette modification est prise en compte dans l'algorithme principal (ou plus généralement dans la fonction appelante)
- Ceci nous à amener à définir la notion de **paramètres d'entrée/sortie** pour les tableaux que la fonction modifiera. Ce n'est pas une entrée à proprement parler, ni une sortie puisque la sortie correspond au type retourné par la fonction ET ce type là ne peut **JAMAIS** être un tableau.

- Nous allons à présent voir comment traduire ces notions en C et en particulier distinguer les paramètres d'entrée des paramètres d'entrées/sortie.

Conséquences (Exemple)

Insertion d'un élément dans un tableau à un indice donné

mes_fonctions.h

```
void Insere_tab(float, unsigned int, float [], unsigned int);
//Il n'est pas nécessaire de préciser la taille réelle du tableau de réels
//Cette taille réelle est passée en paramètre via le unsigned int
```

mes_fonctions.cpp

```
#include"mes_fonctions.h"

void Insere_tab(float el,unsigned int pos, float tab[], unsigned int tr)
{
  if(pos<tr)//Si la position est valide (pourquoi pos>=0 forcément?)
  {
    tab[pos]=el;//on insère l'élément
  }//On pourrait ajouter une branche alternative avec un message d'erreur
}//Notez l'absence d'instruction return
```

main.cpp

```
#include<iostream>
using namespace std;
#include"mes_fonctions.h"

int main()
{
  float tableau[50];
  float x;
  unsigned int index;
  cout<<"Element à insérer:";
  cin>>x;
  cout<<"à quel indice (entre 0 et 49)?";
  cin>>index;
  Insere_tab(x,index,tableau,50);
  cout<<"L'élément inséré est:";//on suppose l'index
  cout<<tableau[index]<<endl;//dans le bon rang
  return 0;
}
```

Paramètres d'entrée/sortie et déclaration

- Pour distinguer un paramètre d'entrée **passé par adresse** (ne devant pas être modifié par la fonction) d'un paramètre d'entrée/sortie modifiable par la fonction, on utilisera le mot clé `const` devant le paramètre d'entrée.
- Dans ce contexte d'utilisation, le mot clé `const` ne signifie pas que le tableau est constant, il indique juste que, pendant la durée de l'appel à la fonction, le tableau est un paramètre d'entrée (lecture seule) et ne peut pas être modifié par la fonction.
  Il peut en revanche être modifié dans l'algorithme principal ou par une autre fonction qui l'utiliserait en paramètre d'entrée/sortie.

Déclaration d'une fonction permettant de copier un tableau de réels dans un autre

/*R : Copier un tableau dans un autre élément par élément (en supposant que la copie est possible, que la taille réelle du tableau recevant la copie soit plus grande ou égale à la taille du tableau à copier)

E : 1 tableau de réels : le tableau à copier (tab_ini), 1 entier : le nombre d'éléments à copier (nb)

E/S : **1 tableau de réels : la copie du tableau (tab_out)**

S : **vide***/

```
void Copie_tab(const float tab_ini[], unsigned int nb, float tab_out[]);
               \_____ para. d'entrée _____/            \____ para. d'E/S ____/
```

Exercices

Exercice 1

- De même, proposer les prototypes (déclarations) avec R, E, E/S, S des fonctions suivantes :
  - Affichage d'un tableau élément par élément
  - Saisie de valeurs dans un tableau (Traduire l'exercice 10.2 du fascicule en langage C)

  Les algorithmes de ces fonctions ont été étudiés en TD.

Exercice 2

Un programme doit demander à un utilisateur de saisir des valeurs entières dans un tableau `tab1`, puis de copier `tab1` dans un autre tableau d'éléments du même type `tab2` et enfin d'afficher les éléments de `tab1`.

- Donner la déclaration de `tab1` et `tab2` dans le programme principal
- Donner ensuite les appels aux fonctions de saisie, copie et affichage en respectant les prototypes précédents.

Tableaux à plusieurs dimensions

- Un tableau peut contenir un certain nombre d'éléments de même type, ce type pouvant être lui aussi un tableau.
- On peut donc créer des tableaux de tableaux, par exemple un tableau de tableaux de 10 réels.

Déclaration et initialisation de tableaux à deux dimensions

- Le principe est que chaque ligne est un élément du tableau. Chaque ligne est donc un tableau de 10 réels (correspondant aux 10 colonnes)

```
//Lexique
  float matrice    [6]      [10]
                nb. lignes nb. col.
```

- Avec initialisation :

```
  unsigned int matrice2[3][4]={{2, 3, 5, 7}, {11, 13, 17,
19}, {23, 29, 31, 37}}
//ou de manière équivalente mais moins claire...
  unsigned int matrice2[3][4]={2, 3, 5, 7, 11, 13, 17, 19,
23, 29, 31, 37}
```

Tableaux à plusieurs dimensions et fonctions

- Utilisation dans les fonctions : on a vu qu'il n'était pas nécessaire de préciser le nombre d'éléments d'un tableau passé en paramètre d'une fonction.
- Pour les tableaux à deux (ou plus) dimensions, ceci n'est vrai **que** pour le nombre de lignes. Il **FAUT** préciser **impérativement** le nombre maximal de colonnes.
- En général, on aura également en paramètre les nombres pratiques de lignes et de colonnes.

Exemple : prototype d'une fonction d'affichage de tableau 2D

/*R : Affiche les éléments d'un tableau de réels à deux dimensions, ligne par ligne

E : 1 tableau de tableau de 100 réels : le tableau à afficher avec un nombre de lignes quelconque et un nombre de colonnes limité à 100 (tab_2d)
1 entier correspondant au nombre pratique de lignes (nb_lignes)
1 entier correspondant au nombre pratique (≤ 100) de colonnes (nb_colonnes)

E/S : vide (la fonction ne modifie pas le tableau)

S : vide */

```
void Affiche_tab2D(const float tab2d[][100], unsigned int nb_lignes,
unsigned int nb_colonnes);
```

Chaînes de caractères et tableaux

- Dans le chapitre précédent, nous avons vu qu'une `chaîne de caractère` peut donc aussi être vue comme un **tableau de caractères** comportant tous les caractères de la chaîne + un caractère spécial appelé caractère de fin de chaîne que l'on a noté `FCH`.

Caractère de fin de chaîne en C

- Ce caractère spécial est normalisé en C, il n'est donc **pas nécessaire de le déclarer**.
  Le caractère de fin de chaîne en C se traduit par le caractère spécial `'\0'`
  C'est un caractère non imprimable.

Exemple

```
char mois[10]={'s','e','p','t','e','m','b','r','e','\0'};
char mois2[10]=''septembre'';
```

La chaîne de caractère "septembre" contenant 9 caractères peut donc être vue comme le tableau de **9+1=10** caractères.
L'accès aux éléments de la chaîne se fait comme pour les tableaux, en utilisant la notation indicielle : mois2[i] ou mois[i] représentent la même information.

Opérations **globales** permises sur des **tableaux de caractères**

- Un des intérêt de déclarer une chaîne de caractère sous forme d'un tableau de caractères (avec un nombre limité maximal de caractères), c'est qu'il est possible de faire des opérations globales sur ce tableau de caractères. On rappelle que ceci n'est pas vrai pour les tableaux d'autres types... attention donc à ne pas tout mélanger...

Exemples :

```
//déclare deux tableaux de 10000 caractères
char tab_car[10000], tab_car2[10000];
```

- Opérations permises :

```
//Saisie d'une chaîne globalement par l'utilisateur
cin>>tab_car;
//Affichage global d'une chaîne de caractères
cout<<tab_car;
```

- Opération toujours interdite :

```
//Copie globale d'une chaîne dans une autre
tab_car2=tab_car; //instruction INVALIDE
```

Manipulation des chaînes de caractères en C

Saisie d'une chaîne dans un programme principal.

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

int main()
{
  char nom[30];//pour ANDRIAMIARANTSOANAVALONA
  cout<<"Veuillez entrer votre nom: "<<endl;
  cin>>nom;//l'espace en mémoire est bien défini et surdimensionné
  cout<<"Bienvenue "<<nom<<endl;
  return 0;
}
```

Tableaux à plusieurs dimensions

- Un tableau peut aussi contenir des tableaux

```
char mois[12][10]={{'j','a','n','v','i','e','r','\0'},{'f','é','v','r','i',
'e','r','\0'},{'m','a','r','s','\0'},{'a','v','r','i','l','\0'},{'m','a',
'i','\0'},{'j','u','i','n','\0'},{'j','u','i','l','l','e','t','\0'},{'a',
'o','û','t','\0'},{'s','e','p','t','e','m','b','r','e','\0'},{'o','c','t',
'o','b','r','e','\0'},{'n','o','v','e','m','b','r','e','\0'},{'d','é','c',
'e','m','b','r','e','\0'}};
```

  Qu'on aurait aussi pu déclarer (et initialiser) comme :

```
char mois[12][10]={"janvier", "février", "mars", "avril", "mai", "juin",
"juillet", "août", "septembre", "octobre", "novembre", "décembre"};
```

- Accès aux éléments en deux temps
  1. mois[0] est le tableau de 10 caractères correspondant ici à {'j','a','n','v','i','e','r','\0'}
  2. mois[0][3] est le 4^e élément du tableau mois[0], c'est à dire, le caractère 'v'.
- On peut imaginer des tableaux de tableaux de tableaux,...

Sommaire

Motivation

Calcul de la distance entre deux points

Proposez un algorithme qui permet de calculer la distance euclidienne entre deux points : P(x_P, y_P) et Q(x_Q, y_Q).

Première idée (Lexique)

Lexique :

xp : 1 réel yp : 1 réel	Concerne uniquement le point P
xq : 1 réel yq : 1 réel	Concerne uniquement le point Q

Envie de regrouper les informations concernant un même objet mais...

Problème

...Il n'existe pas de type point...

Structure de l'information

- Objectif : mémoriser des informations **différentes** concernant un même objet.

Exemples

Personne :	nom	Bulletin :	note fr
	prénom		note maths
	adresse		note II1
	téléphone		...
	mail		
Cycliste :	dossard		
	poids		
	temps de ref		
	...		

- Structurer l'information pour mieux l'exploiter
  $\Rightarrow$ Création d'un nouveau type **structuré** composé de plusieurs informations pouvant être de **types différents**.
- Question : quel est le type de chacune des informations précédentes ?

Déclaration

- Elle se fait dans le Lexique comme toutes les déclarations

Modèle

Lexique :

```
nom_nouveau_type :   un type composé de
                     <
                       champ1 : 1 type
                       champ2 : 1 type
                       ...
                     >
```

{Le nouveau type est créé}

{Il faut maintenant déclarer une variable de ce type}

`nom_variable` : un `nom_nouveau_type`

Exemple

Exemple

Lexique :

{Création du type `point`}

`point` : un type composé de

<

x : 1 réel

y : 1 réel

>

{Déclaration d'une variable de type `point`}

`p` : un `point`

Erreur classique

Environ la moitié des étudiants s'arrête à la déclaration du nouveau type et oublie de déclarer une variable de ce type là.

- Comment affecter "une valeur" à `p` ??

Accès aux champs

- Intérêt : modification d'un champ (ex : modifier le temps de référence d'un coureur cycliste, le numéro de téléphone d'une personne, sans changer les autres informations...)
- Question : comment accéder aux différents champs d'une variable de type composé ?
- Réponse : besoin d'un nouvel opérateur •, appelé opérateur d'accès aux champs.

Opérateur d'accès aux champs

On considère une variable de type composé : `nom_variable` qui a comme champs : `champ_1`, `champ_2`, ..., `champ_n`
Pour accéder au `champ_k` de cette variable de type composé, on utilise la syntaxe suivante :

`nom_variable`•`champ_k`

Cette expression représente une variable **du même type** que le type de `champ_k`
On peut donc lui **affecter une valeur de même type**, se servir de sa valeur actuelle comme on le ferait pour n'importe quelle variable.

Exemple (à chercher)

1. Déclarer un nouveau type `complexe` permettant de représenter un nombre complexe
2. Déclarer une variable `z` de type `complexe`
3. Proposer :
   1. L'algorithme de la fonction permettant de saisir un complexe
   2. L'algorithme de la fonction permettant d'afficher un complexe sous forme algébrique [25]
   3. Un algorithme principal permettant d'appeler et tester ces deux fonctions.

25. la forme algébrique d'un complexe est : Re(z)+jIm(z) où j est tel que $j^2 = -1$

Affectation globale

- On a vu qu'une affectation d'une variable dans une autre est possible à condition que ces deux variables aient **le même type**.
- Il est naturel de se demander si c'est encore vrai pour le type composé qu'on vient de créer.

Affectation de variables de type composé

Il est possible d'affecter **globalement** une variable d'un type composé donné dans une autre variable du **même type**.
Cette affectation peut être vue (et en pratique est réalisée) comme une succession d'affectation **champ à champ** entre ces deux variables.

Exemple

On suppose le type `complexe` déjà déclaré et qu'on dispose de deux complexes z1 et z2. L'affectation de z2 dans z1 est autorisée (même type) et on a :

$$\text{z1} \leftarrow \text{z2} \Leftrightarrow \begin{cases} \text{z1.re} \leftarrow \text{z2.re} \\ \text{z1.im} \leftarrow \text{z2.im} \end{cases}$$

Comparaison globale

Comparaison globale de deux variables du même type composé

Contrairement à l'affectation globale, il faut être plus prudent avec la comparaison globale de deux variables du **même type composé**.
Si on peut autoriser la comparaison globale en algorithmie, il est en général faux de la pratiquer en programmation.
Deux stratégies possibles :

- L'utiliser en algorithmie et être très vigilant lors du passage à la programmation (déconseillée)
- Ne pas l'utiliser dès le départ et comparer **champ à champ** les variables du même type composé (recommandée).

Exercice (à chercher)

1. Déclarer un type point2d.
2. Déclarer une constante ORIGINE de type point2d de coordonnées (0,0).
3. Puis proposer un découpage fonctionnel (et les algorithmes correspondant) permettant à un utilisateur de saisir un point et d'afficher si ce point est l'origine ou d'afficher la distance à l'origine sinon.

Initialisation d'une constante ou variable de type composé

A partir d'un exemple

Lexique :

`point2d` : un type composé de
<
`x` : 1 réel
`y` : 1 réel
>

`ORIGINE` : La constante point2d :={0,0}

Règle pour l'initialisation

Pour **initialiser** une constante de type composé (ou même une variable), il suffit d'indiquer entre accolades les valeurs de chaque champs en les séparant par des virgules.

Et pour l'affectation ?

On ne peut pas procéder de même pour l'**affectation**.
Si `p` est une variable de type `point2d`, il est INTERDIT d'écrire : `p`←{1,2}.
Il faut procéder de manière habituelle en accédant aux champs. `p.x`←1 et `p.y`←2

Sommaire

Déclaration d'un type composé

Lexique :

nom_nouveau_type : un type composé de
<
champ1 : 1 type
champ2 : 1 type
...
>

nom_variable : un nom_nouveau_type

```
struct nom_nouveau_type
{
  type1 champ1;
  type2 champ2;
  ...
};

nom_nouveau_type nom_variable;
```

Exemple : création et utilisation d'un type point2d

```
//Déclaration d'un type point2d
struct point2d{
  float x;
  float y;
};

//Déclaration d'une variable de type point2d
point2d p;
```

- Cette déclaration est souvent réalisée dans un fichier d'en-tête. Par exemple, si on souhaite développer un ensemble de fonctions manipulant les nombres complexes, on pourra créer un fichier `complexe.h` dans lequel on trouvera :
  - la déclaration du nouveau type `complexe`
  - la déclaration de toutes les fonctions sur les complexes.

Opérateur d'accès aux champs

Opérateur d'accès aux champs

On considère une variable de type composé : `nom_variable` qui a comme champs : `champ_1`, `champ_2`, ..., `champ_n`
Pour accéder au `champ_k` de cette variable de type composé, on utilise la syntaxe suivante :

```
nom_variable.champ_k
```

Cette expression représente une variable **du même type** que le type de `champ_k`
On peut donc lui affecter une valeur de même type, se servir de sa valeur actuelle comme on le ferait pour n'importe quelle variable.

Exemple (à chercher)

1. Déclarer en C un nouveau type `complexe` permettant de représenter un nombre complexe
2. Déclarer en C une variable `z` de type `complexe`
3. Proposer la traduction en C de :
   1. L'algorithme de la fonction permettant de saisir un complexe
   2. L'algorithme de la fonction permettant d'afficher un complexe sous forme algébrique [26]
   3. Un algorithme principal permettant d'appeler et tester ces deux fonctions.

   Vous respecterez le découpage en fichiers (`.h`, `.cpp` et `main.cpp`) vu au chapitre sur les fonctions en C.

26. la forme algébrique d'un complexe est : Re(z)+iIm(z)

Rappel : utilisation pratique des structures

- Il est possible d'**affecter globalement** une variable de type composé dans une autre variable du **même** type composé.
  - Exemple :

```
complexe z1,z2;
z1.re=3.4;
z1.im=-2.5;
z2=z1; /* est une instruction autorisée qui copie
la variable z1 champ par champ dans z2*/
```

- Il n'est pas possible de **comparer globalement** des variables de type composé.

```
if(z1==z2)//est une instruction interdite
{
    ...
}
//Le compilateur renverra une erreur de compilation.

//Il faut comparer champ à champ:
if(z1.re==z2.re && z1.im==z2.im)
```

Exercice (à chercher)

1. Déclarer un type `point2d` en C.
2. Déclarer une constante `POINTREF` de type `point2d` de coordonnées que vous choisirez initialement.
3. Suivre le découpage fonctionnel vu au chapitre sur le type composé et traduire les algorithmes proposés permettant à un utilisateur de saisir un point et d'afficher si ce point est l'origine ou d'afficher la distance à l'origine sinon.

Sommaire

Introduction

- L'utilisation des tableaux permet de mémoriser un ensemble d'éléments de même type.
- L'inconvénient est qu'il faut que la taille soit **fixée une fois pour toute** lors de la déclaration du tableau.
  $\Rightarrow$ surdimensionnement du tableau ...
- Autre inconvénient, on aimerait pouvoir retourner un ensemble d'éléments saisis par l'utilisateur (avec une fonction de saisie de notes par exemple).
  Il n'est pas possible de retourner un tableau ...

Ce qu'on aimerait faire

Travailler directement sur des zones de la mémoire en utilisant le nombre de cases mémoires adapté à notre cas (et ne pas surdimensionner un tableau par exemple).

Mémoire et pointeur

- Le programmeur nomme les variables pour travailler dessus.
- Mais le processeur qui exécute le programme ne "voit" que des cases mémoires avec lesquelles il peut échanger des données.
- Le processeur ne connait donc pas le nom des variables mais seulement leur localisation dans la mémoire, c'est à dire leur **adresse mémoire**.
- Pour accéder directement à ces cases, il faut connaître leurs adresses et donc avoir une variable permettant de stocker cette information adresse.

Définition

Un **pointeur** est une variable permettant de stocker une adresse. Il permet donc de repérer (on dira pointer) une case mémoire.

Déclaration d'un pointeur et **Initialisation**

Remarque préliminaire

Un pointeur peut pointer vers des zones de la mémoire dans lesquelles sont stockés différents types d'information : des caractères (`char`), des entiers (`int`, `unsigned int`, ...), des réels (`float` ou `double`), des tableaux, des types composés, d'autres pointeurs, ...

- Pour déclarer un pointeur, on indique d'abord **le type** d'élément pointé suivi d'une étoile (*) suivie du nom du pointeur.
  Exemple : `int * p;` déclare un pointeur d'entier signé nommé `p`.
- Ce pointeur doit forcément "pointer" vers une zone de la mémoire, il faut donc impérativement **initialiser** le pointeur en lui indiquant l'adresse de la zone pointée.
  Comment indiquer au pointeur la zone en mémoire qu'il doit repérer ?
  Faut-il avoir connaissance de toutes les adresses en mémoire lorsqu'on programme ? Comment savoir celles qui sont libres de celles qui sont utilisées par un autre programme ?

Initialisation du pointeur : 1ere possibilité

- Indiquer au pointeur l'adresse d'une variable **existante** (qui a donc un emplacement mémoire bien connu par le système).

Exemple1 :

```
int a;//Déclare une variable a de type int stockée en mémoire
int * pi;//Déclare un pointeur d'entier
pi=&a;//L'opérateur & permet d'accéder à l'adresse d'une variable
```

Exemple2 :

```
float tab[50];//Déclare un tableau tab de 50 float
float * pf;//Déclare un pointeur de float
pf=&tab[0];//pointe sur la première case du tableau
//Mais aussi
pf=&tab[37]; //qui est aussi une case existante du tableau.
```

Attention

Opération interdite : `pf=&a` car `pf` est un pointeur vers des éléments de type réel tandis que a est un entier.
⇒ Respect/cohérence des types !

Initialisation du pointeur : 2e possibilité

- Utiliser une fonction d'allocation de mémoire.
  Leur rôle est de réserver un ensemble de cases mémoires **libres**.
  Elles retournent l'adresse de **la première case** de la zone allouée et permettent donc d'initialiser le pointeur.
  On parle d'**allocation dynamique** puisque, au cours de l'exécution du programme, on peut demander à l'utilisateur quelle est la taille de la zone mémoire qu'il veut utiliser contrairement aux tableaux qui ont une taille fixée une fois pour toute.

Terminologie

- Lorsqu'un tableau est déclaré avec une taille fixée un fois pour toute, on parle de **tableau statique**
- Lorsqu'une zone mémoire est allouée lors de l'exécution du programme (sans que sa taille soit connue au moment de la compilation), cette zone mémoire de cases contigües de même type est appelée **tableau dynamique**

Un tableau **dynamique** est donc un tableau dont la taille n'est pas connue à l'avance et qui peut changer au cours du programme.

Allocation dynamique (en C++)

Principe général

Comme il n'est pas possible de connaître à chaque instant les cases (contigües) libres dans la mémoire, on va utiliser une fonction d'allocation dynamique dont le rôle est de consulter le gestionnaire de mémoire pour déterminer quelle zone de n cases de même type peut être allouée. Si elle en trouve une qui répond à la demande, elle réserve cette zone (qui n'est alors plus disponible) et retourne l'adresse de sa première case qu'on peut stocker dans un pointeur.

- Pour allouer une zone mémoire en C++, on utilise l'opérateur d'allocation dynamique `new` :
  Exemple : `int * pi=new int[50];`
  Déclare un pointeur d'entier `pi` pointant vers la première case d'une zone de la mémoire contenant 50 cases contigües de type `int` ayant été allouée par l'opérateur `new`.

Accès aux éléments pointés : déréférencement du pointeur

- Une fois la zone allouée (ou repérée par un pointeur), il faut pouvoir accéder aux éléments.
- Un pointeur ne pointe, à un moment donné, que sur **un seul** élément de la zone.
- L'accès aux éléments de la zone se fait par une opération appelée **déréférencement** du pointeur.
- Pour accéder **au contenu** de la case pointée par le pointeur, on déréférence donc le pointeur. L'opération de déréférencement se fait de la façon suivante : `*nom_pointeur`
- Exemple :

```
float notes[10]={1.5,2,3,4,5.4,2.3,5.2,3.4,8.2,8.1};
float * p=&notes[3];//Déclaration du pointeur et initialisation
cout<<"adresse case pointée par p: "<<p<<endl;
cout<<"valeur stockée dans cette case: "<<*p<<endl;//déréférencement
//Cette dernière instruction affiche la valeur 4: contenu de notes[3]
```

Changement de case pointée

- Navigation dans la zone mémoire allouée [27].
  Le pointeur pointe sur la première case de la zone, pour changer de case il suffit d'ajouter ou retrancher à la valeur du pointeur, le nombre de cases dont on veut se déplacer.
- Exemple :

```
float notes[10]={1.5,2,3,4,5.4,2.3,5.2,3.4,8.2,8.1};
float * p=&notes[3];
cout<<"adresse case pointée par p: "<<p<<endl;
cout<<"valeur stockée dans cette case: "<<*p<<endl;
p=p+3;//se déplace de 3 cases vers la droite
//On pointe maintenant sur la case d'indice 6
cout<<"valeur stockée dans cette case: "<<*p<<endl;
//Cette dernière instruction affiche la valeur 5.2
//Si on ne veut pas modifier la valeur du pointeur
//on peut quand même lire les autres éléments
cout<<"valeur stockée dans cette case: "<<*(p-4)<<endl;
//Quelle valeur sera affichée?
//Quelle est la case pointée par p?
//Que signifierait: *p-4 au lieu de *(p-4)?
```

27. Rappel : la zone mémoire est constituée de cases contigües

Exemple d'allocation dynamique

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

int main()
{
  float * p_reel=NULL;//ou nullptr depuis 2011 (C++11)
  unsigned int n;
  cout<<"Combien d'éléments voulez-vous saisir?"<<endl;
  cin>>n;
  //Allocation d'un tableau dynamique de n cases
  //n est inconnu au moment de la compilation
  p_reel=new float[n];
  ...//Navigation dans la zone
  return 0;
}
```

Commentaires importants

- On ne pouvait pas faire ça avec des tableaux statiques !
- Que se passe-t-il à la fin du programme ? ?

Libération de la mémoire allouée

- La mémoire est partagée par plusieurs programmes.
- Chacun de ces programmes utilise un espace plus ou moins important en mémoire.

Attention

La manipulation directe de la mémoire est potentiellement dangereuse.
Les zones auxquelles on veut accéder doivent être **libres**.
Une fois qu'on n'a plus besoin de cette zone mémoire, il faut la **libérer** pour ne pas saturer la mémoire.
C'est **au développeur/programmeur=vous !** de veiller à la bonne utilisation de la mémoire.

- Il est donc indispensable de libérer l'espace alloué par `new`
- Fonction de libération de la mémoire : la fonction `delete`
  - Syntaxe : `delete [] nom_pointeur`
  - Exemple :

```
//allocation de 1500 octets
char * pc=new char[1500];
...//utilisation de la zone
delete [] pc;//libère la zone mémoire allouée.
```

Exemple d'allocation dynamique/libération mémoire

```
int main(){
  float a;
  float * p_reel=NULL;//ou nullptr depuis 2011 (C++11)
  unsigned int n;
  cout<<"Combien d'éléments voulez-vous saisir?"<<endl;
  cin>>n;
  p_reel=new float[n];//Allocation d'un tableau dynamique de n cases
  ...//Travail dans la zone
  delete [] p_reel;//libère la zone lorsqu'elle n'est plus utile
  //mais ne détruit pas le pointeur qui peut encore être utilisé
  p_reel=&a; //types ok
  *p_reel=4.3;//Que fait cette instruction?
  return 0;
}
```

Commentaires importants

- L'opérateur `delete` **ne supprime pas** le pointeur ! Il libère la zone allouée par l'opérateur `new`
- Lorsqu'un `new` est utilisé dans un programme, il faut qu'à la fin du programme, avant la destruction des pointeurs, les zones mémoires allouées aient été désallouées. Il faut donc autant de `delete` que de `new` à l'échelle d'un programme.

Lien entre tableaux et pointeurs

- Une zone allouée dynamiquement est une zone constituée d'un nombre arbitrairement grand de cases de même type et est repérée par un pointeur. (adresse de la première case de la zone).
- Il est donc possible de considérer un pointeur comme un tableau [28] et d'accéder aux différents éléments stockés en mémoire grâce à la notation indicielle (souvent plus commode).
- Exemple :

```
  int * p=nullptr;//initialisation du pointeur, toujours
  int t[10]={1,2,3,4,5,6,7,8,9,10};
  p=t;/*autorisée: initialise le pointeur
sur la première case du tableau*/
  p=&t[0]; //équivalent à l'instruction précédente
  t=p;//est une instruction INTERDITE!
  //Accès à l'élément d'indice 3
  //p[3] ou *(p+3) par le pointeur
  //ou t[3] par le tableau (c'est la même zone)
```

28. La réciproque est fausse.

Cas particulier des chaînes de caractères

- Une chaîne de caractères peut être vue comme un tableau de caractères (vu en algorithmie).
- En particulier, nous avons vu qu'on peut indifféremment écrire :
  `char str[8]="Bonjour";`
  ou `char * str="Bonjour";`
  Premier cas : allocation d'un tableau de 8 caractères initialisé avec les caractères `'B','o','n','j','o','u','r','\0'`
  Deuxième cas : allocation d'un tableau **constant** de 8 caractères initialisé avec les caractères `'B','o','n','j','o','u','r','\0'` **et** d'une variable de type pointeur contenant l'adresse de la première case de ce tableau. Problème dans ce cas, on ne peut pas modifier les cases de ce tableau...
- Attention à ne pas généraliser ce qui n'est possible que pour le type `char*` :

```
int tab[5]={1,2,3,4,5};//est autorisé
int * tab={1,2,3,4,5};//reste INTERDITE!
//Une Solution
int * tab1=tab;//Attention: que fait cette instruction?
```

Fonctions et pointeurs

Rappels

- Une fonction peut prendre plusieurs paramètres en entrée ou entrées/sorties mais ne peut retourner qu'**un seul type** d'information.
- Une fonction ne peut pas retourner un tableau (ce n'est pas un type mais une structure de donnée))

Avec les pointeurs on a accès à un type d'information qui peut contenir une adresse.

- Une fonction pourra donc retourner un pointeur !
- Une fonction pourra prendre des pointeurs en paramètre.

Intérêt/Danger/Solution

- Plutôt que de créer un type composé comportant un tableau qui impose à la fonction de retourner ce type composé et de copier tous les éléments du tableau (ce qui peut être long), on ne retourne ici que l'adresse de la zone en mémoire (les données y sont présentes, pas besoin de dupliquer la zone)
- En travaillant sur des adresses directement, on peut modifier les données situées en mémoire. Ceci peut être dangereux lorsque la donnée est en entrée seule. Dans ce cas, comme pour les tableaux on utilisera le mot clé `const` pour indiquer à la fonction qu'elle peut lire la donnée mais pas la modifier.

Exemple : Fonction retournant un pointeur

- Proposer une fonction permettant de demander à un utilisateur d'indiquer un nombre d'éléments à saisir puis de procéder à la saisie et qui renvoie un pointeur vers la zone mémoire contenant l'ensemble des informations saisies.
- Une solution (Extrait) :

```
float * Saisir_reels(void);//Déclaration
float * Saisir_reels(void) //Définition
{
  float * p=nullptr;
  unsigned int n,i;
  cout<<"Entrer le nombre de reels à saisir: ";
  cin>>n;
  p=new float[n];
  i=0;
  while(i<n)
  {
    cout<<"saisir le "<<i+1<<"e nombre: ";
    cin>>*(p+i);//ou cin>>p[i];
    i++;
  }
  return p;//A-t-on oublié de faire un delete? Pourquoi?
}
```

Passage de pointeurs en paramètres de la fonction

- Il est possible de passer des pointeurs en paramètres à une fonction.
- L'intérêt est de manipuler directement une variable en connaissant son adresse mémoire.
- Proposer le code en C++ d'une fonction `Echange` (RES, prototype et définition) permettant de faire l'échange du contenu de deux variables. Testez là dans un algorithme principal.