INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

Algorithmique et programmation 1
2020-2021

Julien OLIVIER Moncef HIDANE
{julien.olivier2,moncef.hidane}@insa-cvl.fr

INSA Centre Val de Loire

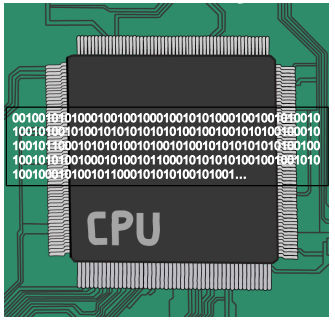
Séance 1

Langage machine

Un programme est une séquence d'*instructions* qui sera exécutée par le *microprocesseur* de l'ordinateur.

Le microprocesseur ne peut exécuter que des instructions écrites en *langage (code) machine*.

Le code machine est composé d'instructions et de données codées en *binaire*.



Le langage machine est très peu expressif : les temps de développement sont très longs et il est très facile de commettre des erreurs.

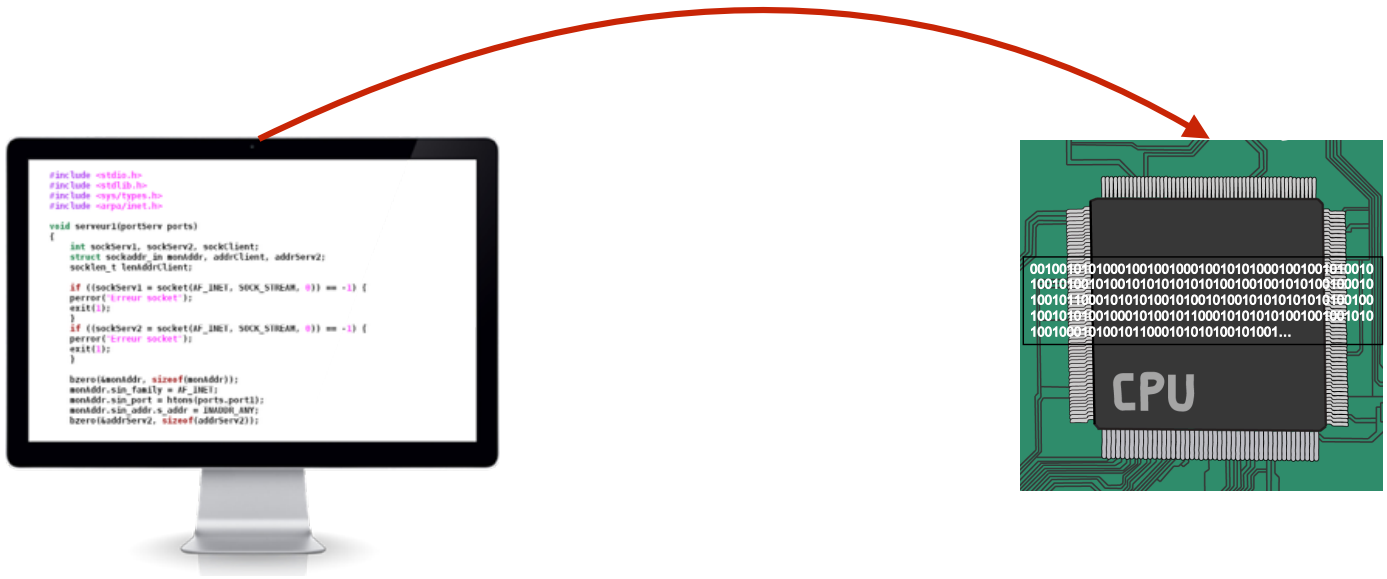
De plus, chaque processeur possède son propre langage machine : il faut tout réécrire dès que l'on change de microprocesseur !

Langages de Programmation

Les *langages de programmation* simplifient l'écriture des programmes en introduisant des *abstractions* comme les *variables*, les *types*, les *expressions*, les *fonctions*, ...

À l'aide d'un langage de programmation, on écrit du *code source* dans un ou plusieurs fichier(s). Ce code source est alors *traduit* en code machine par un programme spécialisé.

Compilateurs et interpréteurs



Dans un langage *interprété*, la phase de traduction est effectuée à chaque exécution du programme. Par exemple, Python est un langage interprété.

Dans un langage *compilé*, la phase de traduction est effectuée une seule fois et le résultat est stocké dans un fichier, que l'on appelle *fichier exécutable*. Le C et le C++ sont des langages compilés.

Algorithmes

Un algorithme est une suite d'étapes clairement définies, permettant de résoudre, dans tous les cas et de manière sûre, un problème posé.

Programmer, c'est mettre en oeuvre (on dit aussi *implémenter*) dans un langage de programmation donné, un algorithme donné, permettant de résoudre un problème posé.

2 points importants

1. La conception d'un algorithme précède son implémentation.
2. Un algorithme n'est pas lié à un langage de programmation particulier.

Un exemple d'algorithme

Résolution de l'équation $x^2 + bx + c = 0$ (dans $\mathbb{R}$).

Entrées : valeurs de b et de c

1 Calculer $\delta = b^2 - 4c$
2 Si $\delta < 0$
    Afficher le message : Aucune solution réelle.
    STOP
3 Si $\delta = 0$
    Calculer $x = -\frac{b}{2}$
    Afficher le message : Solution unique égale a x
    STOP
4 Calculer $x_1 = \frac{-b - \sqrt{\delta}}{2}$
5 Calculer $x_2 = \frac{-b + \sqrt{\delta}}{2}$
6 Afficher le message : Deux solutions x_1 et x_2

Comment programmer ?

```
1 Réfléchir à une solution du problème posé
2 Écrire le code source
3 Compiler le code source
4 Si la compilation a échoué (erreur syntaxique)
    Revenir à l'étape 2
  Sinon
    Tester le programme
    Si le programme est correct
      STOP
    Sinon (erreur sémantique)
      Revenir à l'étape 2 (voire l'étape 1 !!)
```

Pourquoi apprendre à programmer ?

“ The basis for education in the last millenium was ‘reading, writing, and arithmetic ;’ now it is reading, writing, and computing.”

Robert Sedgewick & Kevin Wayne
Introduction to Programming in Java

La langage C

- très populaire
- compilé
- apparu en 1972
- portable
- disposant d'une bibliothèque standard très riche (libc)
- souvent qualifié de bas niveau
- permet la maximisation de la performance
- a influencé beaucoup d'autres langages (C++, Java, PhP, C#, ...)

Les différentes normes du C

- norme ANSI en 1989 (C89 ou ANSI C)
- norme ISO en 1990 (C90)
- norme ISO en 1994 (C94) et 1996 (C95)
- norme ISO en 1999 (C99)
- norme ISO en 2011 (C11)

Objectifs du cours

- Apprendre à concevoir des algorithmes simples permettant de résoudre des problèmes simples.
- Apprendre les premières bases de la programmation en C.
- Apprendre à manipuler des données à l'aide des types du C.
- Apprendre à transcrire des algorithmes en langage C.

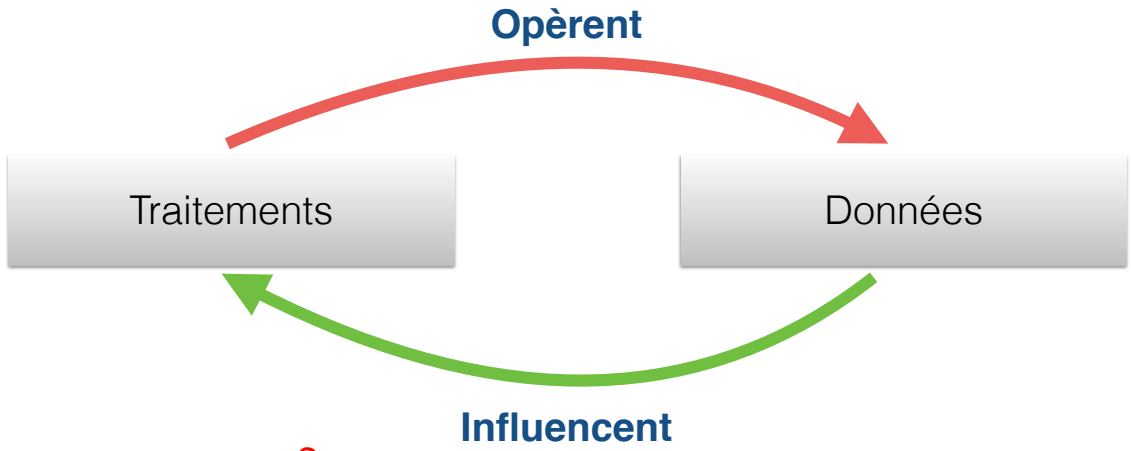
Déroulement du cours

- 5 CM de 1h20
- 1 TD de 1h20 + 4 TD de 2h40 (dont un sur machine)
- 3 TP de 2h40

Contenu du cours

- CM1 + CM2 : introduction, types, variables, expressions, entrées/sorties, alternatives. Présentiel.
- CM3 : tableaux, boucles. Distanciel en autonomie.
- CM4 : fonctions et récursivité. Distanciel en autonomie.
- CM5 : devoir surveillé d'1h
- CM6 : chaînes de caractères. Distanciel en autonomie.

Traitements et données en programmation



1 Calculer $\delta = b^2 - 4c$
2 Si $\delta < 0$
    Afficher : Aucune solution réelle.
    STOP
3 Si $\delta = 0$
    Calculer $x = -\frac{b}{2}$
    afficher : Solution unique égale a x
    STOP
4 Calculer $x_1 = \frac{-b - \sqrt{\delta}}{2}$
5 Calculer $x_2 = \frac{-b + \sqrt{\delta}}{2}$
6 Afficher : Deux solutions x_1 et x_2

Variables

En programmation, une *variable* est un moyen de stocker en mémoire une donnée et d'y associer un nom.

En C, une variable possède 3 caractéristiques

- son *identificateur*
- son *type*
- sa *valeur*

La *syntaxe* pour déclarer une variable en C est la suivante :

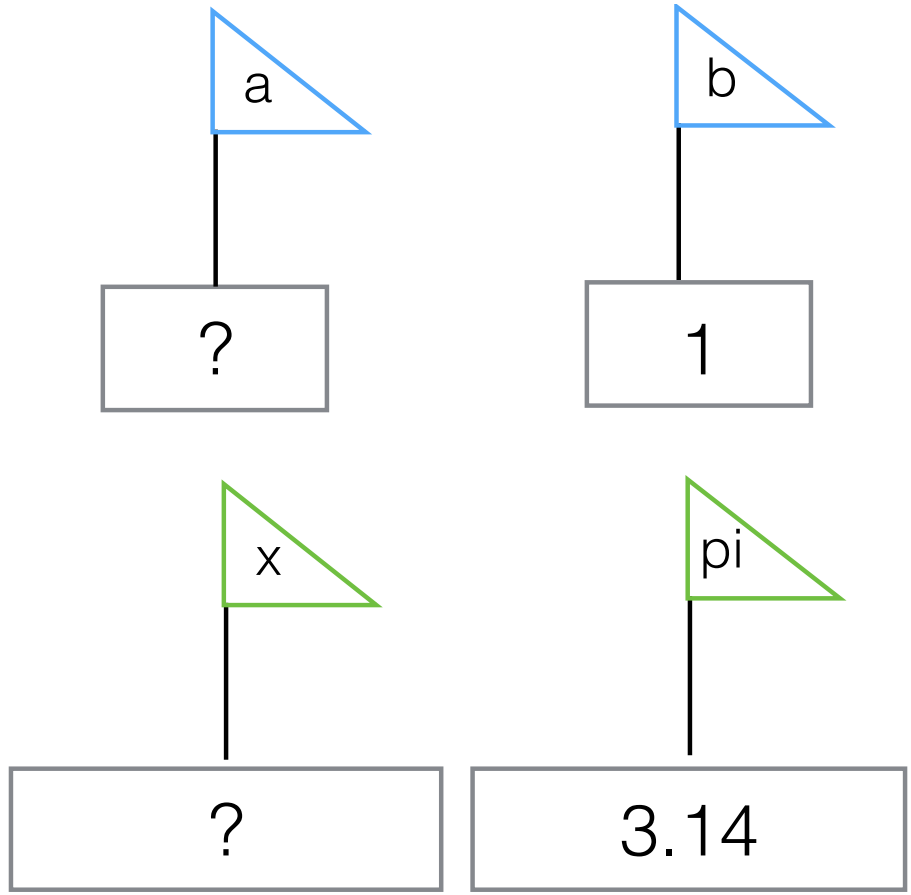
$$type\ indentificateur = valeur\_initiale\,;$$

ou

$$type\ indentificateur\,;$$

Exemples de déclarations

```
int a;

int b = 1;

double x;

double pi = 3.14;
```



Remarques sur la déclaration d'une variable

- En C, une variable doit toujours être déclarée avant d'être utilisée.
- La déclaration d'une variable réserve (en général) un emplacement en mémoire pour stocker cette variable.
- La valeur d'une variable non initialisée est indéterminée. Il ne faut donc jamais utiliser la valeur d'une variable non initialisée.
- Une fois défini, le type d'une variable ne peut plus changer.
- La taille qu'occupe une variable en mémoire dépend de son type.
- Il est possible d'écrire

```
int a, b;
int a = 1, b = 2;
double x, pi = 3.14;
```

Règles de nommage des variables

Les règles suivantes s'appliquent à l'identificateur d'une variable :

- il doit être constitué uniquement de lettres et de chiffres (donc pas d'espaces, par de !, . . .)
- les accents sont interdits
- le premier caractère doit être une lettre
- le caractère blanc souligné _ (tiret bas) est considéré comme une lettre
- un identificateur ne peut pas être un mot-clé du langage
- majuscules et minuscules sont autorisées mais ne sont pas équivalentes : Les noms `moyenne` et `Moyenne` désignent 2 variables différentes

Exemples de noms valides : `somme_valeurs`, `total`, `partieEntiere`
Exemples de noms non valides : `somme valeurs`, `total!`, `partie_entière`

Types de base

Les types de bases en C sont :

- `char` : pour les caractères
- `int` : pour les nombres entiers
- `float` : pour les nombres décimaux en précision simple
- `double` : pour les nombres décimaux en précision

- Le type `int` peut être précédé de `short` ou `long`
- Les types `char` et `int` peuvent être précédés de `signed` ou `unsigned`.
- Le type `double` peut être précédé de `long`.

Constantes littérales

Dans un code source, une valeur comme `42`, s'appelle une *constante littérale*.
Comme pour les variables, les constantes littérales ont un type.

- `42` et `-1` sont de type `int`
- `42.1`, `-1.0`, `42.`, `1e-3` sont de type `double`
- On utilise des guillemets simples pour écrire des constantes littérales représentant des caractères : `'a'`, `'?'`, `'0'`.

- Une `chaîne de caractères` est une séquence, éventuellement vide, de caractères.
- Une constante littérale de type chaîne de caractères s'écrit entre guillemets doubles :

  `"ceci est une chaine de caracteres"` ;
- On reviendra plus en détail sur les chaînes de caractères en CM 6.

Blocs

En C, un bloc (de code) est une partie d'un code source délimitée par une paire d'accolades.

```
{
   ...
}
```

En C, les blocs peuvent être imbriqués. Les instructions d'un même bloc doivent être alignées à gauche. Les instructions d'un sous-bloc doivent être décalées (on dit aussi *indentées*) par rapport à celles du bloc supérieur.

```
{
   int a = 1;
   {
      double x = 1.2, x_carre;
      x_carre = x * x;
   }
   a = a + 1;
}
```

Portée d'une variable

On dit d'une variable déclarée à l'intérieur d'un bloc qu'elle est *locale* (on dit aussi *automatique*).
On dit d'une variable déclarée en dehors de tout bloc qu'elle est *globale* (on dit aussi *externe*).

En C89, les déclarations des variables automatiques doivent figurer au début du bloc où elles sont déclarées.
Cette restriction est levée dans la version C99.

La portée d'une variable locale est limitée au bloc où elle est définie. Autrement dit, une variable locale n'existe que dans le bloc de code où elle est déclarée. Notons cependant qu'un bloc peut contenir des sous-blocs et qu'ainsi une variable déclarée dans un bloc est accessible dans tout sous-bloc.

Constantes symboliques

L'écriture de constantes littérales peut parfois obscurcir le code.
De plus, il est difficile de modifier une constante littérale partout où elle apparaît.

Le mécanisme suivant permet d'introduire des constantes symboliques :

```
#define nom texte de remplacement
```

À partir de cette ligne, toute occurrence de `nom` sera remplacée par `texte de remplacement`.

Par exemple

```
#define PI 3.14159265359
```

remplacera chaque occurrence de `PI` par `3.14159265359`.

Attention, une constante symbolique n'est pas une variable. Pour distinguer les 2, on écrit les noms des variables en minuscules et les noms des constantes symboliques en majuscules.

Opérateurs et expressions

- Dans un code source, les symboles +, -, =, ... s'appellent des *opérateurs*.
- Un opérateur s'applique à un ou plusieurs *opérande(s)*.
- Une *expression* est formée d'un ou plusieurs opérateur(s) et leur(s) opérande(s).

```
a + 1;
```

est une expression *simple* : elle est formée d'un seul opérateur (+) et de 2 opérandes (la variable a et la constante littérale 1).

```
a * 2 + 1;
```

est une expression *composée* : elle est constituée de deux opérateurs (+ et *) chacun ayant deux opérandes (a et 2 pour * et le résultat de a*2 et 1 pour +)

Toute expression en C a une valeur.

L'opérateur d'affectation

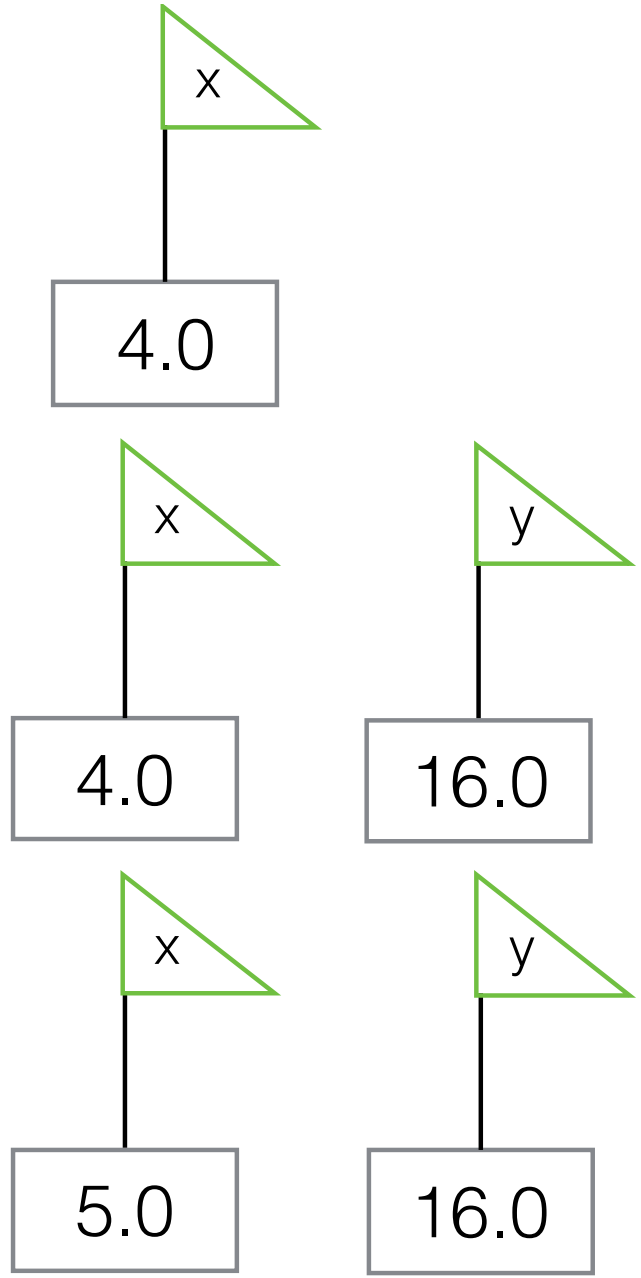
L'opérateur =, dit d'affectation, permet de modifier la valeur d'une variable au cours de l'exécution du programme.
Lors d'une affectation, l'expression se trouvant à droite de = est *évaluée*. La valeur qui en résulte est alors stockée dans la variable qui se trouve à gauche de =.

Attention, une affectation n'est pas une égalité au sens mathématique.
L'instruction

```
a = a + 1;
```

est correcte en C !

```
double x = 4.0;

double y = x * x;

x = x + 1;
```



Les opérateurs arithmétiques

Opérateur	Types	Fonction	Utilisation
$+$	tous	plus unaire	$+$a
$-$	tous	moins unaire	$-$a
$*$	tous	multiplication	a $*$ b
/	char, int	division euclidienne	a / b
%	char, int	reste de la division euclidienne	a % b
/	float, double	division réelle	a / b
$+$	tous	plus binaire	a $+$ b
$-$	tous	moins binaire	a $-$ b

Règles de priorité et d'associativité

L'expression suivante

```
a * 2 + 1
```

peut être interprétée de 2 manières différentes

```
(a * 2) + 1
```

ou

```
a * (2 + 1)
```

Les règles de priorité et d'associativité déterminent la sémantique d'une expression composée.

Règles de priorité

`*` est prioritaire devant `+`. On en déduit alors que

```
a * 2 + 1
```

est équivalent à

```
(a * 2) + 1
```

Règles de priorité et d'associativité

Règles d'associativité

L'expression

```
a / 2.0 * pi
```

est ambiguë car * et / ont la même priorité. Les règles d'associativité permettent dans ce cas de déterminer la sémantique.
On sait que * et / sont associatifs à gauche. Donc

```
a / 2.0 * pi ⟺ (a / 2.0) * pi
```

L'ensemble des règles de priorité et d'associativité des opérateurs du langage C peuvent être consulté ici :
`https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B`

Règles de priorité et d'associativité

- Au lieu d'apprendre toutes les règles de priorité et d'associativité, il vaut mieux utiliser les parenthèses.
- Les parenthèses permettent aussi de modifier la sémantique d'une expression. Par exemple

```
(x+2) * (y-1);
```

- Attention, les règles de priorité et d'associativité ne fournissent aucune indication quant à l'ordre d'évaluation des opérandes. Par exemple, dans l'expression

```
(x+2) * (y-1);
```

on ne sait pas laquelle des sous-expressions `x+2` ou `y-1` est évaluée en premier.

Entrées/Sorties classiques

- Les Entrées/Sorties sont les processus par lesquels un programme interagit avec l'utilisateur.
- Les plus simples sont l'affichage et la saisie d'informations à partir du terminal.
- Le langage C possède une bibliothèque standard (`stdio.h`) fournissant deux fonctions d'Entrées/Sorties :
  - `printf` pour l'affichage,
  - `scanf` pour la saisie.

Mais nous n'allons pas les utiliser (pour l'instant) !

Entrées/Sorties simplifiées

- Mise à disposition d'une bibliothèque interne à l'établissement `insaio.h`.
- Deux fonctions d'Entrée/Sortie (des macro-définitions en réalité) :
  - `AFFICHER` pour l'affichage,
  - `SAISIR` pour la saisie.

Pour le premier semestre, vous utiliserez `AFFICHER` et `SAISIR` pour implémenter les intéractions avec l'utilisateur du programme.

AFFICHER

- AFFICHER permet d'afficher sur le terminal jusqu'à neufs variables et/ou constantes littérales de type chaines de caractères.
- Pour afficher une chaine de caractères, on la passe en argument à la fonction AFFICHER.
- Pour afficher plusieurs chaines, on les sépare par des virgules.
- Rappel : neuf arguments maximum !

Premier exemple : un argument

```
AFFICHER("Voici mon premier affichage");
```

Deuxième exemple : quatre arguments

```
AFFICHER("Voici ","mon ", "deuxième ", "affichage ");
```

AFFICHER

- Afin de pouvoir mettre en forme les chaines de caractères contantes dans l'affichage, on utilise des caractères spéciaux appelés *séquences d'échappement*.
- Pour l'instant, seules deux nous intéressent :
  - `\n` pour aller à la ligne.
  - `\t` pour réaliser une tabulation horizontale.

Exemples

```
AFFICHER("Ligne 1","\n","Ligne 2","\n","Ligne 3","\n");
AFFICHER("Ligne 4\nLigne 5\nLigne 6\n");
AFFICHER("\n");
AFFICHER("1.2\t","2.56\t","5\n","3.658","\t","2","\t","10.56");
```

AFFICHER

- Pour afficher une variable, on la passe en argument à la fonction AFFICHER.
- Pour afficher plusieurs variables, on les sépare par des virgules.

Exemple

```
int a = 2;
float b = 5.3;
double c = 10.2;
AFFICHER(a);
AFFICHER(b,c);
```

AFFICHER

- Il est possible d'alterner entre les variables et les chaines de caractères constantes au sein d'un même appel à AFFICHER.

Exemple

```
int a = 2;
AFFICHER("la variable vaut : ",a,"\n ");
```

SAISIR

- Pour permettre à l'utilisateur du programme de saisir des valeurs pour une ou plusieurs variables, on utilise la fonction SAISIR.
- Pour saisir une variable, il faut la passer en argument à la fonction SAISIR.
- Lors de l'exécution de cette ligne du code source, le programme s'arrêtera et attendra que l'utilisateur saisisse une valeur.
- Une fois celle-ci saisie, elle sera affectée à la variable passée en argument à la fonction SAISIR.

Exemple

```
int a;
AFFICHER("Saisissez une valeur svp.\n");
SAISIR(a)
AFFICHER("Vous avez saisi la valeur ",a);
```

SAISIR

- Il est possible de demander la saisie de plusieurs variables au sein du même appel à SAISIR.

Exemple

```
float a,b;
double c;
AFFICHER("Saisissez trois valeurs réelles svp.\n");
SAISIR(a,b,c);
AFFICHER("Les variables saisies valent : ",a," ",b," ",c);
```

La fonction `main`

- La fonction `main` est le point d'entrée de tout programme écrit en C.
- Tout programme écrit en C doit contenir une unique fonction `main`.

```
int main()
{
  AFFICHER("Voici mon premier programme !\n");
  return 0;
}
```

Un exemple complet

```
#include <insaio.h>

 #define PI 3.1415

int main()
{

   double r, circ, aire;
   AFFICHER("Donnez le rayon d'un cercle.\n");
   SAISIR(r);
   circ = 2*PI*r;
   aire = PI*r*r;
   AFFICHER("La circonference d'un cercle de rayon ",r," est ",circ,".\n");
   AFFICHER("L'aire d'un cercle de rayon ",r," est",aire,".\n");


  return 0;
}
```

Expressions logiques

- Une *expression logique* est une *condition* qu'un programme informatique peut tester.
- C'est à l'aide d'expressions logiques que les données peuvent influencer les traitements.
- Par exemple, dans l'algorithme qui calcule les solutions d'une équation du second degré, les traitements que l'on effectue diffèrent selon le résultat de l'expression logique $\delta > 0$.
- Beaucoup de langages informatiques définissent un type booléen pouvant représenter 2 valeurs logiques : *vrai* et *faux*.
- En C ANSI, il n'existe pas de type booléen.

Opérateurs relationnels et d'égalité

En C, on construit des expressions logiques élémentaires à l'aide des *opérateurs relationnels* et des *opérateurs d'égalité*.

Symbole	Signification
$<$	strictement inférieur à
$>$	strictement supérieur à
$<=$	inférieur ou égal à
$>=$	supérieur ou égal à

Symbole	Signification
$==$	égal à
$!=$	différent de

Notez la différence entre l'opérateur == (test d'égalité) et l'opérateur = (affectation).

Remarques sur les opérateurs relationnels et d'égalité

- Les opérateurs relationnels et d'égalité
  - produisent la valeur entière 0 si l'expression est fausse et la valeur entière 1 sinon. Il n'existe pas de type booléen en C89.
  - peuvent être utilisés pour comparer des entiers (`char` et `int`) et des flottants (`float` et `double`)
  - ont une priorité inférieure à celle des opérateurs arithmétiques. Ainsi, l'expression `i + j < k - 1` est équivalente à `(i + j) < (k - 1)`
  - sont associatifs à gauche. Ainsi l'expression `i < j < k` est équivalente à `(i < j) < k`. Attention, ici se cache un bug !
- Les opérateurs d'égalité ont une priorité plus faible que celle des opérateurs relationnels. Ainsi, l'expression `i < j == j < k` est équivalente à `(i < j) == (j < k)`.
- Que vaut l'expression `(i >= j) + (i == j)` lorsque $i = 3$ et $j = 3$?

Opérateurs logiques

Les opérateurs logiques permettent de construire des expressions logiques complexes à partir d'expressions élémentaires.

Symbole	Signification
!	négation logique
&&	conjonction logique
||	disjonction logique

Les *tables de vérité* sont les suivantes :

P	!P
V	F
F	V

P	Q	P&&Q
V	V	V
V	F	F
F	V	F
F	F	F

P	Q	P||Q
V	V	V
V	F	V
F	V	V
F	F	F

Remarques sur les opérateurs logiques

- Les opérateurs logiques produisent, eux aussi, 0 (faux) ou 1 (vrai).
- Souvent, les opérandes des opérateurs logiques sont 0 ou 1 (ou des expressions dont la valeur est 0 ou 1). Cependant ceci n'est pas une obligation.
- Dans tous les cas, un opérande différent de zéro est considéré comme vrai et un opérande égal à zéro est considéré comme faux.
- L'évaluation d'une expression faisant intervenir l'opérateur `&&` ou `||` est *court-circuitée* : l'opérande de gauche est évalué en premier. L'opérande de droite n'est évalué que si le resultat reste indéterminé après la première évaluation. Par exemple, si la variable `i` est égale à 0, l'opérande de droite ne sera pas évalué dans l'expression

```
(i != 0) && (j / i > 0)
```

Structures de contrôle

- Une *structure de contrôle* est une commande qui contrôle l'*ordre* dans lequel les différentes instructions d'un programme sont exécutées.
- On appelle *flot d'exécution* l'enchaînement des instructions d'un programme.
- On peut représenter le flot d'exécution à l'aide d'un *graphe de flot de contrôle* ou *algorigramme*.

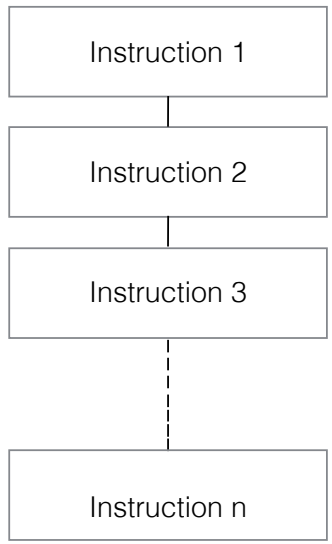


Figure – *

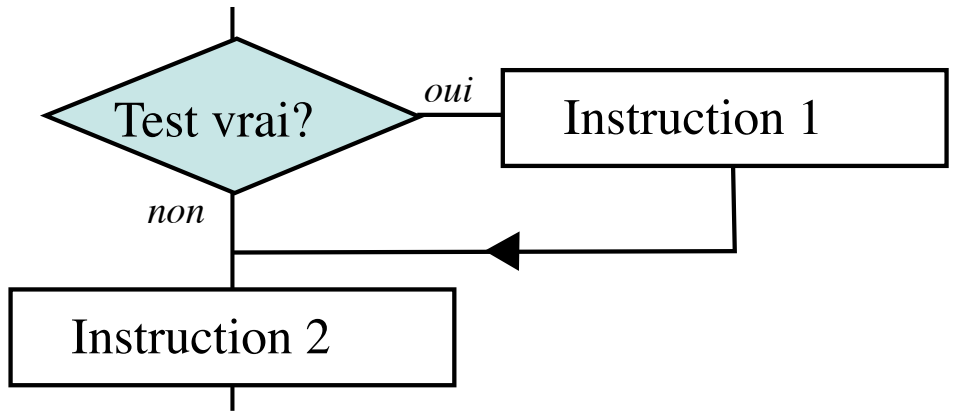
Exemple d'un *flot séquentiel*

Alternatives

- Les alternatives sont des structures de contrôle particulières. Elles effectuent un test logique sur une condition. Selon le résultat de ce test, un ensemble spécifique d'instructions est exécuté.
- Ainsi, les alternatives permettent de concevoir des programmes dont le flot d'exécution n'est plus séquentiel.
- De manière générale, on peut distinguer 4 types d'alternatives :
  - test si
  - test si sinon
  - test sinon si
  - test selon

Test si

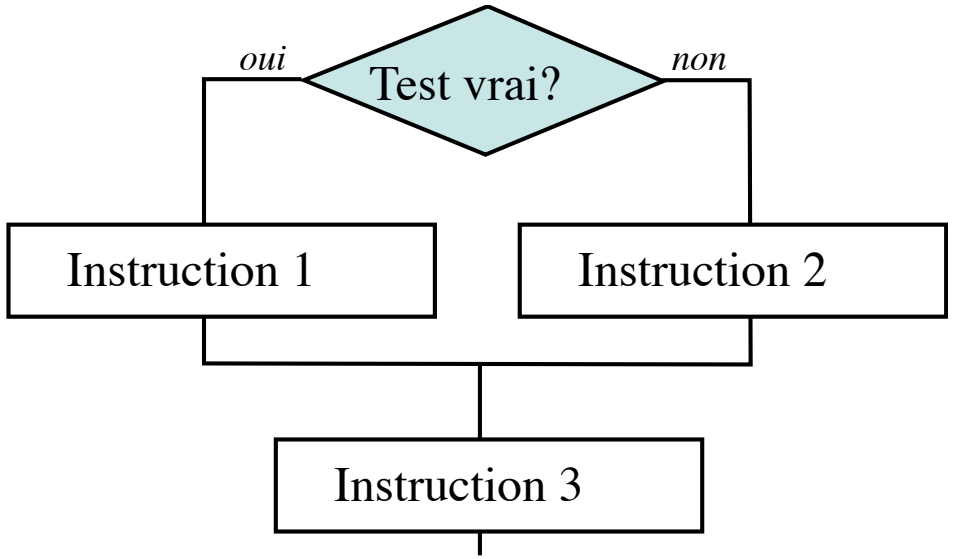
```
SI Test
   Instruction 1
FIN SI
Instruction 2
```



Remarque : `Instruction 1` peut comporter plusieurs instructions.

Test si sinon

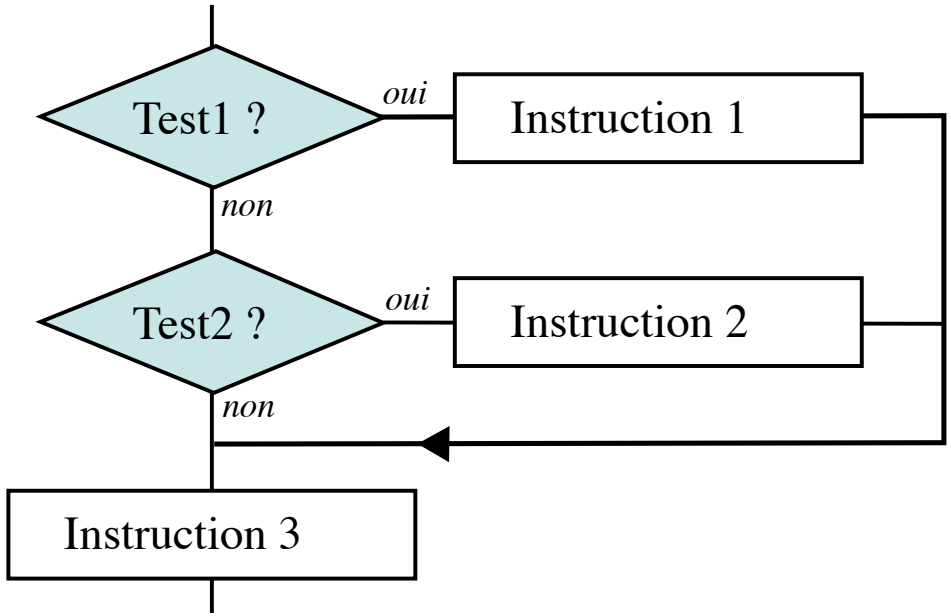
```
SI Test
   Instruction 1
SINON
   Instruction 2
FIN SI
Instruction 3
```



Remarque : `Instruction 1` et `Instruction 2` peuvent comporter plusieurs instructions chacune.

Test sinon si

```
SI Test1
   Instruction 1
SINONSI Test2
   Instruction 2
FIN SI
Instruction 3
```



Remarques :

1. On peut avoir plusieurs `SINONSI`.
2. `Instruction 1` et `Instruction 2` peuvent comporter plusieurs instructions chacune.

Test selon

```
SELON Variable 1
CAS Valeur 1:
   Instruction 1
CAS Valeur 2:
   Instruction 2
FIN SELON
Instruction 3
```

Remarques :

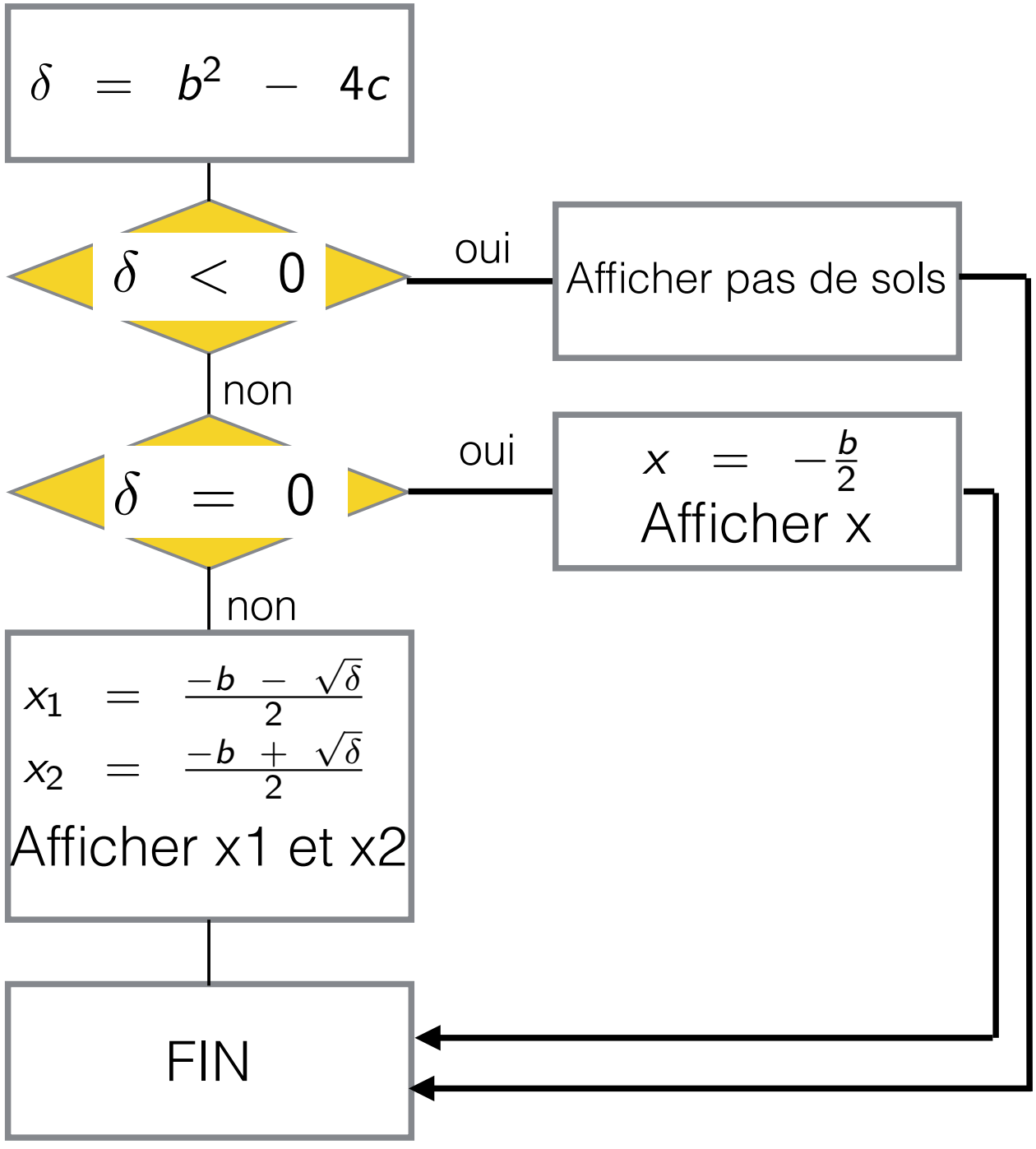
1. On peut avoir plusieurs `CAS`.
2. On peut avoir un CAS par défaut.
3. `Instruction 1` et `Instruction 2` peuvent comporter plusieurs instructions chacune.

Une version structurée de l'algorithme de résolution d'une équation du second degré

Résolution de l'équation $x^2 + bx + c = 0$ (dans $\mathbb{R}$).

Entrées : valeurs de b et de c

1 Calculer $\delta = b^2 - 4c$
2 Si $\delta < 0$
    Afficher le message : Aucune solution réelle.
3 Sinon Si $\delta = 0$
    Calculer $x = -\frac{b}{2}$
    Afficher le message : Solution unique égale a x
4 Sinon
    Calculer $x_1 = \frac{-b - \sqrt{\delta}}{2}$
    Calculer $x_2 = \frac{-b + \sqrt{\delta}}{2}$
    Afficher le message : Deux solutions x_1 et x_2


$\delta = b^2 - 4c$
$\delta < 0$
oui
Afficher pas de sols
non
$\delta = 0$
oui
$x = -\frac{b}{2}$
Afficher x
non
$x_1 = \frac{-b - \sqrt{\delta}}{2}$
$x_2 = \frac{-b + \sqrt{\delta}}{2}$
Afficher x1 et x2
FIN

Instruction `if`

- En C, l'instruction `if` permet d'implémenter les test si.
- La syntaxe est la suivante

  `if ( expression ) instruction`

- Lorsqu'une instruction `if` est exécutée, l'expression qui se trouve entre les parenthèses est *évaluée*. Si la valeur obtenue est différente de zéro (ce que le C interprète comme étant vrai), `instruction 1` est exécutée ; sinon, on passe directement aux instructions suivantes (s'il y en a).

```c
/* Attention a ne pas ecrire line_cnt = MAX_VAL */
if (line_cnt == MAX_VAL)
   ligne_num = 0;

/* Attention a ne pas ecrire 0 <= i < n */
if (0 <= i && i < n )
   AFFICHER(i," est entre 0 et ",n);
```

Instructions composées

- L'instruction se trouvant dans le corps d'un `if` peut être composée.
- Une instruction composée est un *bloc* de la forme

  `{ instruction }`
- L'instruction `if` fournit donc le 2ème exemple de bloc utile. Quel est le premier ?

```
if (fichier_vide != 1)
{
  ligne_cnt = 0;
  page_num++;
}
```

Attention, une variable déclarée à l'intérieur d'un bloc n'est visible qu'au sein de ce bloc. C'est la notion de portée vue en CM1.

```c
if (a == b)
{
    int i = 0; /* Variable locale definie dans le corps d'un if */
    i++;
}
/*
 * On n'a plus acces a la variable i ici !
 */
```

La clause else

- L'instruction if permet également d'implémenter un test si sinon, via la clause else.
- La syntaxe est la suivante

  `if ( expression ) instruction else instruction`
- L'instruction se trouvant après else n'est exécutée que si la condition est fausse.
- L'instruction du else peut être composée.

```
if (i > j)
   max = i; /* Pas besoin d'accolades car instruction simple */
else
{ /*
   * Ici on a besoin d'accolades car il s'agit d'une instruction
  composee.
   */
   max = j;
   AFFICHER("Dans la clause else\n");
}
```

Test `sinon si` en C

- La langage C ne définit pas de clause `elseif` (en un seul mot).
- On peut simuler un test `sinon si` en mettant une instruction `if/else` dans une clause `else`

```
if (var == 1)
   AFFICHER("vaut 1\n");
else
  if (var == 2)
     AFFICHER("vaut 2\n");
  else
      if (var == 3)
         AFFICHER("vaut 3\n");
      else
         AFFICHER("ne vaut ni 1, ni 2, ni 3\n");
```

Test `sinon si` en C

Il est conseillé d'écrire le code précédent de la manière suivante

```
if (var == 1)
   AFFICHER("vaut 1\n");
else if (var == 2)
   AFFICHER("vaut 2\n");
else if (var == 3)
   AFFICHER("vaut 3\n");
else
   AFFICHER("ne vaut ni 1, ni 2, ni 3\n");
```

Il faut bien comprendre que les deux exemples de code précédents sont strictement identiques. Seule la présentation du code change. Il n'y a pas de mot clé `elseif` en C.

Une remarque sur l'imbrication des `if/else`

En C, une clause `else` est associée au `if` le plus interne n'ayant pas de `else`.

```
if (y != 0)
  if (x != 0)
    result = x / y ;
else /* Malgre l'indentation, ce else apparatient a if (x != 0) */
  AFFICHER("Erreur : division par zero\n");
```

La bonne indentation est

```
if (y != 0)
  if (x != 0)
    result = x / y ;
  else
  AFFICHER("Erreur : division par zero\n");
```

Instruction switch

- L'instruction switch du C permet d'implémenter un test selon.
- La syntaxe est la suivante :

```
switch ( expression )
{
    case constant-expression : instructions
    ...
    case constant-expression : instructions
    default : instructions
}
```

- Chaque case doit correspondre à une expression constante entière. Par exemple une constante littérale comme 5 ou une expression comme 5+10, mais pas une expression comme n + 10.
- Il est interdit d'utiliser 2 labels identiques dans 2 cases différents.
- La clause default est optionnelle.
- Les instructions qui suivent un case se terminent généralement par le mot clé break.

Instruction switch

Le programme suivant affiche le message suivant

b

```
int nombre = 2;

switch (nombre)
{
    case 1 :
    AFFICHER("a\n");
    break;
    case 2 :
    AFFICHER("b\n");
    break;
    case 3 :
    AFFICHER("c\n");
    break;
    default :
    AFFICHER("Ni a, ni b, ni c\n");
}
```

Le mot-clé break

Le programme suivant affiche le message suivant

```
b
c
Ni a, ni b, ni c
```

```
int nombre = 2;

switch (nombre)
{
    case 1 :
    AFFICHER("a\n");
    break;
    case 2 :
    AFFICHER("b\n");
    case 3 :
    AFFICHER("c\n");
    default :
    AFFICHER("Ni a, ni b, ni c\n");
}
```

Un exemple utile d'omission de break

```
switch (nombre)
{
    case 1 :
    case 2 :
    case 3 :
    AFFICHER("a ou b ou c\n");
    break;
    default :
    AFFICHER("Ni a, ni b, ni c\n");
}
```

```
/* Le meme programme en plus compact */
switch (nombre)
{
    case 1 : case 2 : case 3 :
    AFFICHER("a ou b ou c\n");
    break;
    default :
    AFFICHER("Ni a, ni b, ni c\n");
}
```

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

Algorithmique et programmation 1
2020-2021

Moncef HIDANE Julien OLIVIER
{moncef.hidane,julien.olivier2}@insa-cvl.fr

INSA Centre Val de Loire

Séance 3
Tableaux et boucles

Tableaux

- Un tableau est une structure de données constituée d'un ensemble d'éléments, chacun étant identifié par (au moins) un indice.
- Un tableau doit être stocké de sorte à pouvoir accéder "rapidement" à un élément à partir de son (ses) indice(s).
- En C, un tableau unidimensionnel est un type composé permettant de stocker plusieurs données de même type et d'y référer par un unique identificateur.
- La syntaxe pour déclarer un tableau est

  `type identificateur[<taille>] ;`
- La syntaxe pour accéder à un élément du tableau est

  `identificateur[i]`

```c
#define PI 3.14159265359

int main()
{
  double rayons[3];
  double  aires[3];

  rayons[0] = 1.2;   /* L'indice du 1er element est 0. */
  rayons[1] = 17.4;
  rayons[2] = 37.0;  /* L'indice du dernier element est 2. */

  aires[0] = PI*rayons[0]*rayons[0];
  aires[1] = PI*rayons[1]*rayons[1];
  aires[2] = PI*rayons[2]*rayons[2];

  return 0;
}
```

```c
#define PI 3.14159265359
#define TAILLE 3

int main()
{
  double rayons[TAILLE];
  double  aires[TAILLE];

  rayons[0] = 1.2;  /* L'indice du 1er element est 0. */
  rayons[1] = 17.4;
  rayons[2] = 37.0; /* L'indice du dernier element est 2. */

  aires[0] = PI*rayons[0]*rayons[0];
  aires[1] = PI*rayons[1]*rayons[1];
  aires[2] = PI*rayons[2]*rayons[2];

  return 0;
}
```

```c
#define PI 3.14159265359

int main()
{

  /* Initialisation explicite.
     Pas besoin d'indiquer la taille.
  */
  double rayons[] = {1.2, 17.4, 37.0};

  double aires[3];

  aires[0] = PI*rayons[0]*rayons[0];
  aires[1] = PI*rayons[1]*rayons[1];
  aires[2] = PI*rayons[2]*rayons[2];

  return 0;
}
```

```c
int main()
{
  /*
   * Initialisation explicite a partir d'une suite de caracteres.
   */
  char hello1[] = {'H', 'e', 'l', 'l', 'o', '!'};

  /*
   * Initialisation explicite a partir d'une chaine de caracteres.
   */
  char hello2[] = "Hello!";

  /*
   * Attention, malgre les apparences, hello1 et hello2
   * ne contiennent pas la meme chose.
   */

  return 0;
}
```

ATTENTION : n'utilisez pas l'opérateur d'affectation pour copier des tableaux

```
1  int tab1[] = {1, 2, 3};
2  int tab2[] = {4, 5, 6};
3
4
5  /*
6   * Ne compile pas !
7   */
8  /* tab2 = tab1; */
9
10 /* Il faut copier a la main. */
11 tab2[0] = tab1[0];
12 tab2[1] = tab1[1];
13 tab2[2] = tab1[2];
```

ATTENTION : n'utilisez pas une variable comme taille de tableau

```c
/* Ne compile pas (en C89). */
/* int taille = 10; */
/* int tab1[taille]; */

/* Il faut ecrire */
/* int tab1[10]; */

/* Ou ecrire */
#define TAILLE 10
int tab1[TAILLE];
```

Tableaux multidimensionnels

- La déclaration suivante crée un tableau bidimensionnel

```
int m[3][5];
```

- On peut penser au tableau `m` comme ayant 3 lignes et 5 colonnes (attention, les indices commencent toujours à 0).

m[0][0]	m[0][1]	m[0][2]	m[0][3]	m[0][4]
m[1][0]	m[1][1]	m[1][2]	m[1][3]	m[1][4]
m[2][0]	m[2][1]	m[2][2]	m[2][3]	m[2][4]

Tableaux multidimensionnels

- En réalité, le C ne stocke pas un tableau bidimensionnel comme une matrice, mais comme un tableau unidimensionnel résultant de la *concaténation* des lignes de la matrice correspondante :

m[0][0]	m[0][1]	m[0][2]	m[0][3]	m[0][4]	**m[1][0]**	m[1][1]	m[1][2]	m[1][3]	m[1][4]	**m[2][0]**	m[2][1]	m[2][2]	m[2][3]	m[2][4]

Boucles

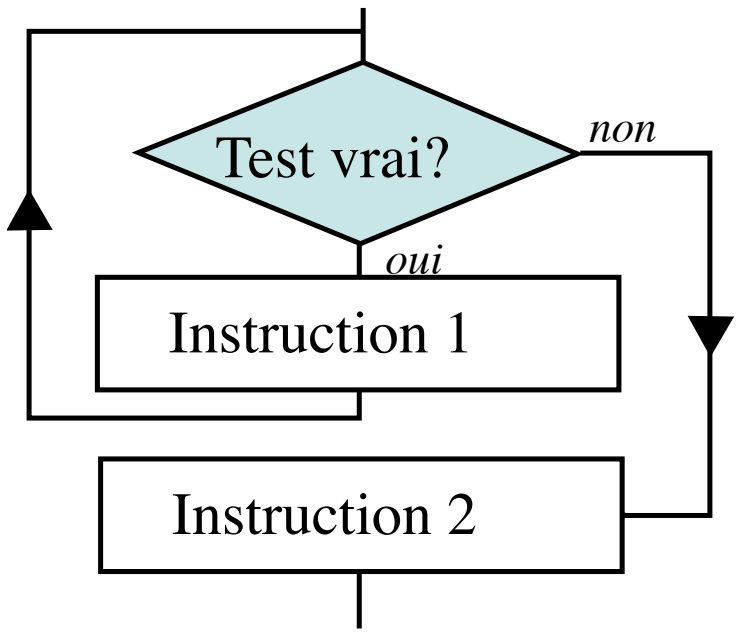
- Une boucle est une instruction qui exécute de manière répétée un ensemble d'instructions, appelé *corps de la boucle*.
- Chaque exécution du corps de la boucle s'appelle une *itération*.
- En C, toutes les boucles ont des *expressions de contrôle* : à chaque itération (au début ou à la fin), cette expression est évaluée ; si le résultat est différent de zéro (vrai en C), le corps de la boucle est exécuté ; sinon on sort de la boucle.
- Le C fournit trois instructions pour itérer : `while`, `do` et `for`.

Boucle `while`

- C'est la boucle la plus simple et la plus essentielle en C.
- La syntaxe est donnée par

```
while (condition) { instructions }
```

- Comme pour les alternatives, la condition est considérée comme vraie si et seulement si sa valeur est non nulle.
- L'algorigramme est le suivant



Le code suivant calcule la plus petite puissance de 2 supérieure ou égale à nombre `n`

```
i = 1;
while (i < n)
{
   i = i * 2;
}
```

Voici une *trace d'exécution* pour `n = 10`

i = 1 ;	i vaut 1
i < n ?	Oui ; exécuter le corps de la boucle
i = i * 2 ;	i vaut 2
i < n ?	Oui ; exécuter le corps de la boucle
i = i * 2 ;	i vaut 4
i < n ?	Oui ; exécuter le corps de la boucle
i = i * 2 ;	i vaut 8
i < n ?	Oui ; exécuter le corps de la boucle
i = i * 2 ;	i vaut 16
i < n ?	Non ; on sort de la boucle

Le code suivant affiche un compte à rebours

```
i = 5;
while (i > 0)
{
  AFFICHER("T - ", i, "\n");
  i--; /* l'effet est equivalent a i = i - 1; */
}
AFFICHER("T\n");
```

Voici ce qu'affiche ce programme

T - 5
T - 4
T - 3
T - 2
T - 1
T

Un exemple complet

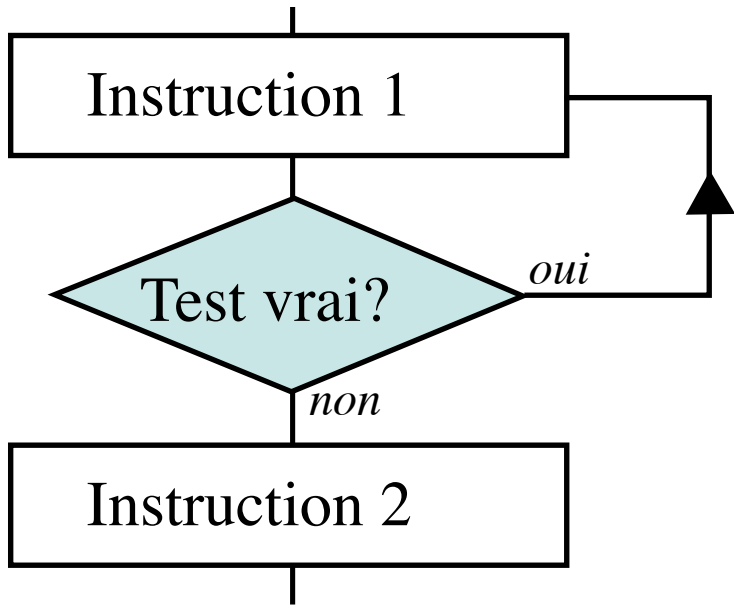
```
/* Somme d'une suite de nombres */
int main()
{
  int n, somme = 0;

  /* Notez le \ dans la ligne suivante. */
  AFFICHER("Ce programme calcule la somme d'une suite de nombres\
  entiers.\n");
  AFFICHER("Entrer un nombre (0 pour terminer) : ");

  SAISIR(n);
  while (n != 0)
  {
    somme += n; /* Equivalent a somme = somme + n; */
    SAISIR(n);
  }
  AFFICHER("La somme est egale a : ", somme, "\n");

  return 0;
}
```

La boucle do

- La boucle `do` est très similaire à la boucle `while`.
- Dans une boucle `do`, l'expression de contrôle est testée *à la fin* de chaque exécution du corps de la boucle.
- En particulier, les instructions du corps d'une boucle `do` sont exécutées au moins une fois.
- La syntaxe est la suivante

```
do { instructions } while ( condition );
```

- L'algorigramme est le suivant



Le code suivant affiche un compte à rebours en utilisant une boucle do.

```
i = 5;

do {
  AFFICHER("T - ", i, "\n");
  i--;
} while (i > 0) ; /* Notez bien point-virgule a la fin du while. */
AFFICHER("T\n");
```

Voici ce qu'affiche ce programme

T - 5
T - 4
T - 3
T - 2
T - 1
T

La boucle `for`

- La boucle `for` est souvent (mais pas seulement) utilisée pour écrire des boucles contenant des variables de comptage.
- La syntaxe est la suivante

  ```
  for (expr1; expr2; expr3) instruction
  ```

- `expr1` correspond à une expression d'initialisation. Elle est exécutée une seule fois au début.
- `expr2` correspond à l'expression de contrôle. Elle est exécutée avant chaque itération. On procède à une itération si et seulement si `expr2` est vraie.
- `exp3` est une expression exécutée à la fin de chaque itération, généralement pour modifier l'état d'une variable figurant dans l'expression de contrôle.

Le code suivant affiche un compte à rebours en utilisant une boucle `for`.

```
for (i = 5; i > 0; i--)
{
  AFFICHER("T ", i, "\n");
}
AFFICHER("T\n");
```

Voici une *trace d'exécution*

i = 5 ;	i vaut 5
i > 0 ?	Oui ; afficher T - i
i-- ;	i vaut 4
i > 0 ?	Oui ; afficher T - i
i-- ;	i vaut 3
i > 0 ?	Oui ; afficher T - i
i-- ;	i vaut 2
i > 0 ?	Oui ; afficher T - i
i-- ;	i vaut 1
i > 0 ?	Oui ; afficher T - i
i-- ;	i vaut 0

Lien entre boucle `for` et boucle `while`

Sauf à de rares exceptions, la boucle `for` suivante

```
for (expr1; expr2; expr3) { instructions }
```

peut être transformée en la boucle `while` suivante

```
expr1; while(expr2) { instructions; expr3; }
```

Le mot-clé `break`

- On a vu que le mot-clé `break` permet de sortir d'une instruction `switch`.
- Le mot-clé `break` permet aussi de sortir d'une boucle `while`, `do` ou `for`.
- `break` est utile pour écrire des boucles dont le point de sortie se trouve éventuellement au milieu, et non pas forcément au début ou à la fin.
- L'instruction `break` permet de sortir du `while`, `do`, `for` ou `switch` le plus interne.

Le programme suivant permet de savoir si un nombre entier `n` est premier ou pas.

```
int d;
for (d = 2; d < n; d++)
{
  if (n % d == 0)
  {
      break;
  }
}
if (d < n)
{
  AFFICHER(n, " est divisible par ", d, "\n");
}
else
{
  AFFICHER(n, " est premier\n");
}
```

Le programme précédent peut être écrit sans `break` en utilisant une `instruction nulle`.

```
int d;
for (d = 2; (d < n) && (n % d != 0); d++)
{
  /*
   * Notez le point virgule qui suit.
   * Il s'agit d'une instruction nulle.
   * Tout se passe dans l'en-tete du for
   */
  ;
}
if (d < n)
{
  AFFICHER(n, " est divisible par ", d, "\n");
}
else
{
  AFFICHER(n, " est premier\n");
}
```

Attention, le `break` du programme suivant fait sortir du `switch` et non pas du `while`

```
while (...)
{
  switch(...)
  {
    ...
    break;
    ...
  }
}
```

Le mot-clé `continue`

- L'utilisation du mot-clé `continue` permet de passer à l'itération suivante.
- Autrement dit, si le programme rencontre un `continue` à l'itération n, il passe directement à l'itération $n + 1$ et ceci même si plusieurs instructions figurent dans le code-source après `continue`.
- Attention, le mot-clé `continue` n'a rien à faire dans un `switch`. Son utilisation est réservée au boucles.

Le programme suivant calcule la somme de 10 nombres entiers non nuls saisis au clavier.

```
int n = 0, sum = 0, i;
while (n < 10)
{
  SAISIR(i);
  if (i == 0)
  {
      continue;
  }
  sum += i;
  n++;
}
```

En règle générale, il faut essayer de minimiser l'utilisation des `break` et surtout celle de `continue` dans les boucles que vous écrivez.

Retour sur l'équivalence `for`/`while`

On avait dit que, sauf dans rares occasions, les boucles `for` et `while` sont équivalentes. Le programme suivant fournit un exemple où les 2 boucles ne sont pas équivalentes. Pourquoi ?

```
int n = 0, sum = 0, i;
while (n < 10)
{
  SAISIR(i);
  if (i == 0)
  {
      continue;
  }
  sum += i;
  n++;
}
```

```
int sum = 0, n, i;
for (n = 0; n < 10; n++)
{
  SAISIR(i);
  if (i == 0)
  {
      continue;
  }
  sum += i;
}
```

- Les boucles sont particulièrement adaptées à l'utilisation des tableaux.
- Lorsque le nombre d'éléments d'un tableau est connu, on privilégiera une boucle *for*.
- Lorsque le nombre d'éléments n'est pas connu, et qu'une valeur particulière indique la fin du tableau, on s'orientera vers une boucle *while*. C'est en particulier le cas des chaînes de caractères que l'on abordera plus tard dans le cours.

Exemple

Saisie d'un tableau d'entiers

```
#include <insaio.h>
#define TAILLE 5
int main ()
{
  int i;
  int tab[TAILLE];

  AFFICHER("Saisissez les ",TAILLE," valeurs du tableau.\n");
  for(i=0;i<TAILLE;i++)
  {
    AFFICHER("Saisissez la valeur ", i+1, ": ");
    SAISIR(tab[i]);
  }

  return 0;
}
```

Exemple

Affichage d'un tableau d'entiers

```c
#include <insaio.h>
#define TAILLE 5
int main ()
{
  int i;
  int tab[TAILLE];

  /* Saisie du tableau */

  for(i=0;i<TAILLE;i++)
  {
    AFFICHER("[", i+1, "]\t", tab[i], "\n");
  }

  return 0;
}
```

Exemple

Calculer la valeur minimale d'un tableau

```
int main() {

  double tab[] = {1, 2, -1, 5};

  double min = tab[0];
  int i;

  for (i = 1; i < 4; i++)
  {
    if (tab[i] < min)
    {
       min = tab[i];
    }
  }

  AFFICHER(min);

  return 0;
}
```

Exemple

Affichage inverse des éléments d'un tableau

```
# define TAILLE 5
int main() {

  double tab[TAILLE];
  int i;
  for (i = 0; i < TAILLE; i++)
  {
    SAISIR(tab[i]);
  }

  for (i = TAILLE-1; i >= 0; i--)
  {
    AFFICHER(" ", tab[i]);
  }
  AFFICHER("\n");

  return 0;
}
```

INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
CENTRE VAL DE LOIRE

Algorithmique et programmation 1
2020-2021

Moncef HIDANE Julien OLIVIER
{moncef.hidane,julien.olivier2}@insa-cvl.fr

INSA Centre Val de Loire

Séance 4
Les fonctions

Généralités

Définition

- En C, une fonction est un ensemble d'instructions possédant un nom.
- Une fonction manipule des **données** à l'aide d'**instructions** afin de produire un **résultat**.

- Les données initiales de la fonctions sont appelées ses **paramètres**.
- Le résultat produit par la fonction est appelé **valeur retournée**.

Intérêts

Les fonctions ont deux intérêts :

1. elles permettent de découper un programme en plusieurs parties ;
2. elles sont réutilisables.

Syntaxe et vocabulaire

Syntaxe

La forme générale pour définir une fonction est

```
type-retourné nom-fonction(<parametres>)
{
  declarations
  instructions
}
```

Vocabulaire

1. La ligne `type-retourné nom-fonction(<parametres>)` s'appelle *l'en-tête* de la fonction.
2. Le *corps* d'une fonction est l'ensemble des déclarations et instructions se trouvant à l'intérieur des accolades.

Marche à suivre pour écrire une fonction :

1. Identifier les données initiales qu'elle va manipuler
   $\rightarrow$ les paramètres.
2. Déterminer si elle va produire un résultat
   $\rightarrow$ type de la valeur retournée.
3. Rédiger l'en-tête de la fonction.
4. Réfléchir à l'algorithme permettant, à partir des paramètres, d'obtenir le résultat voulu.
5. Implémenter cet algorithme
   $\rightarrow$ le corps de la fonction.

Une fonction peut ne pas produire de valeur résultat. Le type de la valeur retournée est alors *void*.

Exemple : étapes 1 et 2

Fonction qui produit la moyenne de deux réels

1. Donnée initiales : deux nombres réels.
   → Deux paramètres de type *float*
2. Valeur produite en sortie : moyenne de nombres réels
   → Valeur retournée de type *float*

Fonction qui produit le quotient réel de deux entiers

1. Donnée initiales : deux nombres entiers.
   → Deux paramètres de type *int*
2. Valeur produite en sortie : quotient réel de nombres entiers
   → Valeur retournée de type *float*

Fonction qui **produit** 1 si un nombre entier appartient à un intervalle défini par deux entiers et 0 sinon

1. Donnée initiales : trois nombres entiers (le nombre à tester + les bornes de l'intervalle).
   → Trois paramètres de type *int*
2. Valeur produite en sortie : 1 ou 0
   → Valeur retournée de type *int*

Fonction qui **affiche** si un nombre entier appartient à un intervalle défini par deux entiers

1. Donnée initiales : trois nombres entiers (le nombre à tester + les bornes de l'intervalle).
   → Trois paramètres de type *int*
2. Valeur produite en sortie : **AUCUNE**
   → Valeur retournée de type *void*

- Lorsqu'un tableau est paramètre d'une fonction, celle-ci n'a aucun moyen de connaître sa taille.
- Il est donc nécessaire de la préciser à l'aide d'un deuxième paramètre.

Fonction qui affiche un tableau de réels

1. Donnée initiales : un tableaux de réels.
   $\rightarrow$ Deux paramètres : un tableau de *float*, un entier de type *int* correspondant à la taille du tableau.
2. Valeur produite en sortie : **AUCUNE**
   $\rightarrow$ Valeur retournée de type *void*

En-tête d'une fonction

- Les étapes 1 et 2 nous permettent décrire l'en-tête de la fonction. Syntaxe : `type-retourné nom-fonction(<parametres>)`
- On choisit un nom pour la fonction le plus explicite possible.
- On choisit un nom pour chaque paramètre.
- Chaque paramètre est précédé de son type.
- Les différents paramètres sont séparés par des virgules.
- Les règles de nommage des fonctions et des paramètres sont identiques à celles des variables (pas de caractères spéciaux, pas d'espace, etc.).
- On écrira `void` dans la liste des paramètres si une fonction n'a pas de paramètres.

Exemple : étape 3

Fonction qui produit la moyenne de deux réels

1. Deux paramètres de type *float*
2. Valeur retournée de type *float*
3. `float moyenne(float x, float y)`

Fonction qui produit le quotient réel de deux entiers

1. Deux paramètres de type *int*
2. Valeur retournée de type *float*
3. `float division(int p1, int p2)`

Fonction qui **produit** 1 si un nombre entier appartient à un intervalle défini par deux entiers et 0 sinon

1. Trois paramètres de type *int*
2. Valeur retournée de type *int*
3. `int intervalle(int min, int max, int x)`

Fonction qui **affiche** si un nombre entier appartient à un intervalle défini par deux entiers

1. Trois paramètres de type *int*
2. Valeur retournée de type *void*
3. `void affiche_intervalle(int min, int max, int x)`

Remarque sur l'utilisation des tableaux

- Losqu'un paramètre est un tableau, il est inutile d'indiquer sa taille dans les crochets (sauf dans le cas des tableaux multi-dimensionnels).

Fonction qui affiche un tableau de réels

1. Deux paramètres : un tableau de *float*, un entier de type *int*
2. Valeur retournée de type *void*
3. `void affiche_tableau(float tab[], int taille)`

Corps d'une fonction

Généralités

- Le corps d'une fonction est l'implémentation de l'algorithme permettant de réaliser les opérations voulues.
- On peut y déclarer de nouvelles variables
- On peut y utiliser tout type d'instructions du langage C.
- Lorsque la fonction produit une valeur, on utilisera le mot-clé `return` pour réaliser cette tâche.
  Syntaxe : `return <expression>;`

Corps d'une fonction

Le mot-clé `return`

- `return` permet de produire une valeur en sortie de la fonction.
- Le type de cette valeur doit coïncider avec le type précisé dans l'en-tête de la fonction
- lorsque le programme rencontre le mot-clé `return`, l'exécution de la fonction est immédiatement stoppée. Le programme poursuit alors au niveau de l'instruction d'appel.

Important

- Au départ, une fonction ne connait que la valeur de ses paramètres.
- Il faut donc que l'algorithme implémenté soit capable de réaliser les opérations voulue uniquement avec ces valeurs (cela n'interdit pas la déclaration et l'utilisation de nouvelles variables).

Exemples : étapes 4 et 5

Fonction qui produit la moyenne de deux réels

```
1  float moyenne(float x, float y)
2  {
3      return (x+y)/2;
4  }
```

Fonction qui produit le quotient réel de deux entiers

```
1  float division(int p1, int p2)
2  {
3      float tmp = p1;
4      return tmp/p2;
5  }
```

Fonction qui **produit** 1 si un nombre entier appartient à un intervalle défini par deux entiers et 0 sinon

```
1  int intervalle(int min, int max, int x)
2  {
3      if(x>min && x<max)
4          return 1;
5      else          /* le else n'est pas obligatoire */
6          return 0;
7
8  }
```

Fonction qui **affiche** si un nombre entier appartient à un intervalle

```
1     void affiche_intervalle(int min, int max, int x)
2    {
3            if(x>min && x<max)
4            {
5                AFFICHER("La variable est dans l'intervalle");
6            }
7            else
8            {
9                AFFICHER("La variable n'est pas dans l'intervalle");
10           }
11    }
```

Fonction qui affiche un tableau de réels

```
1  void affiche_tableau(float tab[], int taille)
2  {
3      int i;
4      for(i=0;i<taille;i++)
5      {
6          AFFICHER("[",i+1,"]\t",tab[i],"\n");
7      }
8  }
```

Bonus : une fonction bien élevée

```
1  void bonjour(void)
2  {
3      AFFICHER("Bonjour\n");
4  }
```

Retour sur le `return`

- Une fonction non `void` qui arrive à sa fin sans rencontrer de `return` aura une valeur de retour indéterminée. Il est donc important de vérifier qu'un `return` est présent dans tous les branchements d'une fonction non `void`.
- Une fonction void peut utiliser le mot-clé `return`. Auquel cas, aucune expression ne devra être mentionnée après return.

```
1       void affiche_positif(int i)
2       {
3           if (i < 0)
4               return;
5           AFFICHER(i);
6           /* On n'a pas besoin de else car si on arrive à
7           AFFICHER c'est que la condition du if est fausse. */
8       }
```

Portée des variables

- Rappel : au départ, une fonction ne connaît que la valeur de ses paramètres.
- Toute variable déclarée dans la fonction respecte la règle des blocs de codes :
  $\rightarrow$ elle n'a d'existence que dans le bloc de code défini par la fonction.
- Bien entendu, on peut définir des sous-blocs dans le corps d'une fonction (alternatives, boucles, etc.).

- Lorsque l'on souhaite appeler une fonction, il faut obligatoirement préciser une valeur pour chaque paramètre. Ces valeurs sont appelées les **arguments** de la fonction.
- La fonction est alors exécutée et les paramètres de celle-ci prennent la valeur des arguments.
- Une fonction produit (généralement) un résultat. L'**appel à une fonction** est donc une **expression** dont la valeur correspond à celle produite par la fonction.
- Si l'on souhaite exploiter le résultat de la fonction, il faut exploiter cette expression.
- La valeur produite pourra donc être utiliser de plusieurs manières : affectation à une variable, affichage, test dans une condition etc.

Fonction qui produit la moyenne de deux réels

```
1   float moyenne(float x, float y)
2   {...}
3   int main()
4   {
5       float a,b,resultat;
6   /* Les 3 appels suivants sont équivalents */
7       resultat = moyenne(5,2.3);
8       AFFICHER("moyenne de deux réels : ",resultat);
9
10      a=5;
11      b=2.3;
12      resultat = moyenne(a,b);
13      AFFICHER("moyenne de deux réels : ",resultat);
14
15      AFFICHER("moyenne de deux réels : ",moyenne(a,b));
16
17      return 0;
18  }
```

Fonction qui produit le quotient réel de deux entiers

```
1        float moyenne(float x, float y){...}
2        float division(int p1, int p2){...}
3        int main()
4        {
5            float resultat;
6            int x,y;
7        /* Les 2 appels suivants sont équivalents */
8            resultat = division(10,3);
9            AFFICHER("10/3 = : ",resultat);
10
11           x=10;
12           y=3;
13           resultat = division(x,y);
14           AFFICHER("10/3 = : ",resultat);
15   /*  Ici on utilise la valeur produite par moyenne() comme argument
16       de division(). Comme moyenne retourne un réel, seule sa partie
17       entière sera prise en argument de division() */
18           AFFICHER("10/4 = : ",division(x,moyenne(5,3)));
19
20           return 0;
21       }
```

Fonction intervalle

```
1  int intervalle(int min, int max, int x){...}
2  int main()
3  {
4      int inf=0,sup=10,var,resultat;
5      SAISIR(var);
6      /* Version 1*/
7      resultat = intervalle(inf,sup,var);
8      if(resultat == 0){
9          AFFICHER("La variable est dans l'intervalle");
10     }
11     else{
12         AFFICHER("La variable n'est pas dans l'intervalle");
13     }
14     /* Version 2 : sans variable intermédiaire*/
15     if(intervalle(inf,sup,var) == 0){
16         AFFICHER("La variable est dans l'intervalle");
17     }
18     else{
19         AFFICHER("La variable n'est pas dans l'intervalle");
20     }
21     return 0;
22 }
```

Les fonction qui ne produisent pas de résultats sont utilisées comme des **instructions** et non des expressions.

Fonction affiche_intervalle

```
1  void affiche_intervalle(int min, int max, int x){...}
2  int main()
3  {
4      int inf=0,sup=10,var,resultat;
5      SAISIR(var);
6      affiche_intervalle(inf,sup,var);
7      return 0;
8  }
```

- Lorsque que l'on passe un tableau en argument d'une fonction, on précise simplement son nom (sans les crochets).
- Nous avons vu qu'une fonction ne connaissait pas la taille d'un tableau et qu'il était nécessaire de la considérer comme un deuxième paramètre.
- Cette propriété permet également de ne faire travailler une fonction que sur une partie du tableau

Fonction qui affiche un tableau de réels

```
1  void affiche_tableau(float tab[], int taille){...}
2  int main()
3  {
4      float tab1={20, 5.2, 3.1}, tab2 = {6, 7.5, 1,23, 12, 25};
5
6      affiche_tableau(tab1,3); /* On affiche tab1 en entier*/
7
8      affiche_tableau(tab2,5); /* On affiche tab2 en entier*/
9
10     affiche_tableau(tab1,2); /* On n'affiche que les 2 premières
11                              valeurs de tab1 */
12     affiche_tableau(tab2,3); /* On n'affiche que les 3 premières
13                              valeurs de tab2 */
14     /* L'instruction suivante est incorrecte */
15      affiche_tableau(tab1,10); /* On affiche trop de valeurs !*/
16     return 0;
17 }
```

Paramètres et arguments

Nous avons rencontrés les termes *paramètre* et *argument*. Nous allons à présent préciser la différence entre les 2.

Paramètre

Un paramètre apparaît dans la définition d'une fonction. Il s'agit d'un nom qui sera associé à une variable qui sera créée lorsque la fonction sera appelée. Dit autrement, tant que le fonction n'est pas appelée, les paramètres n'existent pas.

Argument

Un argument est une expression qui apparaît dans l'appel d'une fonction.

```
float m = moyenne(1+2, x/y);
/* Les arguments sont 2 expressions arithmetiques. */
```

Passage d'arguments

Définition

Le *passage d'arguments* est le mécanisme par lequel les paramètres sont créés à partir des arguments fournis lors de l'appel de la fonction.

Passage par valeur

En C, le passage d'arguments se fait toujours par valeur : lors de l'appel d'une fonction, les paramètres sont définis et initialisés avec la valeur des arguments associés.
Dit autrement, les paramètres sont des copies indépendantes des arguments.

Un point fondamental

À la fin d'un appel de fonction, les paramètres ainsi que les variables locales de la fonction sont détruits.

Inconvénients du passage par valeur

Le passage par valeur a également des inconvénients. Par exemple, il est impossible (avec nos connaissances actuelles en C) d'écrire une fonction permettant d'échanger la valeur de 2 variables.

Exercice

Qu'affiche le programme suivant ? Pourquoi .

```
void echange(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
int main()
{
    int a = 1, b = 2;
    echange(a,b);
    AFFICHER("a = ",a," b = ",b);
    return 0;
}
```

Arguments et tableaux

Un point très important

- Les tableaux sont un cas particulier du passage d'arguments
- Le tableau paramètre d'un fontion n'est pas une copie du tableau passé en argument mais le même !
- Ainsi, une fonction peut modifier les éléments du tableau passé en argument.
- Il s'agit d'un cas particulier. Nous apprendrons pourquoi au S2.

Arguments et tableaux

```
void incremente_tableau(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
            a[i]=a[i]+1;
}
# define TAILLE 5
int main()
{
    int i, b[] = {0, 1, 2, 3, 4};
    for(i = 0; i < TAILLE; i++){
        AFFICHER(b[i],"\t");
    }
    AFFICHER("\n");
    incremente_tableau(b,TAILLE);
    for(i = 0; i < TAILLE; i++){
         AFFICHER(b[i],"\t");
    }
    return 0;
}
```

Fonction appelante

- Lorsqu'une fonction est appelée, le contrôle du programme est transféré vers cette fonction.
- Les éventuels paramètres et variables locales sont créées et les diverses instructions exécutées.
- La fonction s'arrête si une des 2 conditions suivantes est vérifiée
  1. toutes les instructions de la fonction ont été exécutées ;
  2. Un `return` est rencontré.
- À ce moment le contrôle est rendu à l'environnement appelant (qui est forcément une fonction) qu'on appelle *fonction appelante*.

La fonction `main()`

- La fonction `main()` est particulière : c'est le point d'entrée de tout programme écrit en C.
- Dit autrement, les instructions machine exécutées par un programme correspondent aux déclarations et instructions se trouvant dans `main()`.
- Ainsi, il n'est pas possible de générer un exécutable à partir d'un code source C si ce dernier ne définit pas de fonction `main()`.
- Attention, une est une seule fonction `main()` doit être définie.
- L'instruction `return` dans `main()` termine l'exécution du programme.
- La valeur retournée par `main()` est particulière : il s'agit d'un code que certains système d'exploitation peuvent consulter lorsque le programme a cessé de fonctionner. La valeur `0` indique que l'exécution du programme s'est bien déroulée. Toute valeur différente de `0` sert à indiquer une exécution anormale.

Déclaration d'une fonction

Jusqu'à présent, nous avons toujours *défini* nos fonctions avant de les utiliser. Reprenons l'exemple 3 :

```
1   void bonjour(void)
2   {
3       AFFICHER("Bonjour\n");
4   }
5   int main()
6   {
7       bonjour();
8
9       return 0;
10  }
```

La *définition* de la fonction `bonjour()` se fait entre les lignes 1 et 4. L'appel se fait ligne 7.

En C, on n'est pas obligé de définir une fonction avant de l'utiliser. Il suffit de la *déclarer*.

```c
/* Ce qui suit est la declaration de la fonction */
void bonjour(void);

int main()
{
    bonjour();

    return 0;
}
/* Ce qui suit est la definition de la fonction */
void bonjour(void)
{
    AFFICHER("Bonjour\n");
}
```

- La déclaration d'une fonction sert à indiquer au compilateur le type de retour ainsi que le nombre et le type des éventuels paramètres.
- Ces informations sont appelés *prototype* (ou *signature*) de la fonction.
- Bien entendu, le prototype d'une fonction doit coïncider avec l'en-tête de sa définition.
- La bibliothèque standard utilise très souvent le mécanisme de déclaration. Par exemple le fichier `math.h` rassemble les *déclarations* de toutes les fonctions mathématiques (srt,pow,etc.).

Un point important

Une fonction peut être déclarée plusieurs fois (du moment que les déclarations sont identiques).
Cependant, une fonction ne peut être définie qu'une unique fois.

INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
CENTRE VAL DE LOIRE

Algorithmique et programmation 1

2020-2021

Moncef HIDANE Julien OLIVIER
{moncef.hidane,julien.olivier2}@insa-cvl.fr

INSA Centre Val de Loire

Caractères et chaînes de caractères

Le type char

- On utilise le type `char` pour déclarer des variables destinées à contenir des caractères.
- On peut initialiser des variables de type `char` à partir d'un seul caractère. Exemple

```
char c = 'a'; /* le caractere a */
c = 'A';      /* le caractere A */
c = '7';      /* le caractere 7 et non pas le nombre 7*/
char c = ' '; /* le caractere espace */
```

- Nous avons également vu en CM1 que l'on pouvait utiliser des séquences d'échappement pour les caractères. Par exemple

```
c = '\n';  /* nouvelle ligne */
c = '\t';  /* tabulation */
c = '\a';  /* ring bell */
```

Comment sont représentés les caractères en C ?

- En C, les caractères sont représentés comme des entiers de petite taille.
- Chaque caractère correspond à un code entier particulier.
- Une table de correspondance est ainsi établie entre un ensemble de caractères et un ensemble de nombres.
- C'est à partir de cette table que les différents caractères disponibles sont codés.

Comment sont représentés les caractères en C ?

- Supposons que la table utilisée soit la table ASCII (plus de détails dans la diapo suivante). Dans cette table, la valeur 65 correspond au caractère `A`. L'instruction suivante

  ```
  char var = 'A';
  AFFICHER(var);
  ```

  affichera A à l'écran, alors que l'instruction

  ```
  AFFICHER('A');
  ```

  affichera 65.
- Si l'on souhaite afficher le code entier correspondant à une variable de type char, il est nécessaire de passer par une variable entière. Ce comportement est lié à la macro AFFICHER (nous verrons au 2ème semestre qu'il y a plus simple !).

```c
char var = 'A';
int x = var;
AFFICHER(x); /* La valeur 65 sera affichée à l'écran */
AFFICHER(var) /* Le caractère A sera affiché à l'écran */
```

La table ASCII

- Le standard du langage C n'impose pas le choix d'une table de correspondance particulière.
- L'une des tables les plus répandues aujourd'hui est la table ASCII.
- Cette table utilise des entiers compris entre 0 et 127 pour coder 128 caractères.

Attention à la portabilité

- Comme le standard ne garantit pas l'utilisation de la table ASCII, il faut éviter de faire cette hypothèse quand vous écrivez vos programmes.
- En particulier, il est inutile d'apprendre par coeur cette table (pensez à la parcourir au moins une fois !).

La table ASCII

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Opérer sur des charactères

- Le fait que les variables de type `char` correspondent à des entiers a plusieurs implications.
- En particulier, lorsqu'un `char` apparaît comme opérande d'un opérateur (arithmétique ou de comparaison par exemple), c'est la valeur entière correspondante qui intervient.

```
/* On suppose que la table ASCII est utilisee. */
char ch;
ch = 65;     /* ch est egale a 'A' */
ch = ch + 1; /* ch est egale a 'B' */
ch++;        /* ch est egale a 'C' */
```

Opérer sur des caractères

Exercice 1 : que fait ce programme ?

```
/* On suppose que la table ASCII est utilisee. */
char ch;
SAISIR(ch);
if ('a' <= ch && ch <= 'z')
    ch = ch - 'a' + 'A';
```

Exercice 2 : à quoi correspondent les itérations de cette boucle ?

```
/* On suppose que la table ASCII est utilisee. */
for (ch = 'A'; ch <= 'Z'; ch++)    ....
```

Exercice 3

Quelles sont les hypothèses précises que nous avons utilisées sur la table de correspondance entre caractères et entiers ?

1 Les caractères en C

2 Les chaînes de caractères en C

Chaînes littérales

Définition

Une chaîne littérale est une suite de caractères entre des guillemets doubles :

```
"Ceci est chaine de caracteres ! "
```

- Pour l'instant, nous n'avons utilisé ces chaînes littérales que comme arguments de la macro `AFFICHER`.
- Les séquences d'échappement peuvent naturellement apparaître dans une chaîne littérale. Par exemple, l'affichage de la chaîne : `"Bonjour,\ntout le monde !"` sera

```
Bonjour
tout le monde !
```

Chaînes littérales trop longues

- Si une chaîne littérale est trop longue pour être écrite sur une même ligne, il faut la couper. Pour cela, on peut utiliser un caractère \ (anti-slash).
- Exemple :

```
AFFICHER("Everything should be made as simple as \
possible, but not simpler");
```

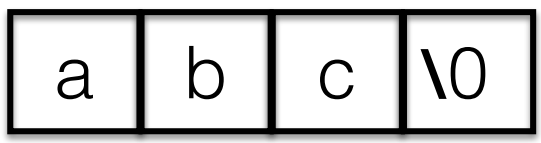
- Attention à l'indentation, elle sera prise en compte dans la chaîne. lorsque vous utilisez \.
- Pour couper des chaînes trop longues, il est préférable d'utiliser la syntaxe suivante :

```
AFFICHER("Everything should be made as simple as"
"possible, but not simpler").
```

Ici, comme les deux chaînes sont adjacentes, le compilateur les *concatène* (les met bout à bout) automatiquement.

Comment sont représentées les chaînes littérales ?

- Le C stocke les chaînes littérales comme des tableaux de caractères.
- Plus précisément, lorsqu'un compilateur C rencontre une chaîne littérale de taille n, il alloue une zone mémoire permettant de contenir $n + 1$ caractères.
- Pourquoi $n + 1$ et pas n ? Le C insère toujours un caractère terminaison, dit caractère nul. La représentation littérale de ce caractère est `'\0'`.
- Par exemple, la chaîne `"abc"` est représentée en mémoire sous la forme suivante



- Attention à la différence entre `'\0'` et `'0'`.

Une remarque

Ne confondez les constantes littérales de type caractère et les chaînes littérales.

- 'a' est une constante littérale de type caractère.
- "a" est une chaîne littérale ; elle contient deux caractères : 'a' et '\0'.

Variables de type chaîne de caractères

- Dans beaucoup de langages de programmation, il existe un type `string` permettant de représenter des chaînes de caractères.
- En C, il n'existe pas de type `string`.
- En C, tout tableau de caractères peut être considéré comme une chaîne de caractères. Le caractère nul ('\0') joue alors le rôle de caractère de terminaison.
- Pour initialiser une variable de type chaîne de caractères on pourra écrire

```
/* 8 a cause du caractere de terminaison */
char date1[8] = "Juin 14";
/* Pas besoin d'indiquer la taille. */
char date2[] = "Juin 14";
/* Moins pratique mais possible. */
char date3[] = {'J', 'u', 'i', 'n',  ' ', '1, '4', '\0'};
```

Afficher des chaînes de caractères

- Les chaînes de caractères sont interprétables par la macro AFFICHER (vous affichez des chaînes de caractères constantes depuis le premier cours !).
- AFFICHER parcours toute la chaîne à partir du premier caractère et affiche chaque caractère tant qu'elle ne rencontre pas le caractère de terminaison.
- Il faut donc s'assurer que ce dernier est présent !

```
char str1[] = "Hello";
char str2[] = "Hel\0lo";
char str3[] = {'H','e','l','l','o'}; /* bug ici */
AFFICHER(str1,"\n"); /* "Hello" sera affiché */
AFFICHER(str2,"\n"); /* "Hel" sera affiché */
AFFICHER(str3); /* comportement anarchique incoming ! */
```

Saisir des chaînes de caractères

- La macro SAISIR accepte des chaînes des caractères variables en argument.
- Lors de l'exécution de l'instruction, l'utilisateur saisira la chaîne de caractères au clavier.
- Le caractère de terminaison sera ajouté automatiquement.
- Il faut s'assurer que la chaîne dans laquelle on souhaite stocker les caractères saisis est suffisamment grande !
- Si l'utilisateur saisit plus de caractères que ce que peut stocker la chaîne, cela peut entraîner un arrêt brutal du programme.
- Rappel : une chaîne de taille *n* ne permet la saisie que de *n-1* caractères (il faut garder une place pour caractère de terminaison).

```
char str[8];
SAISIR(str);
AFFICHER(str);
```

Accès aux éléments d'une chaîne

- Une chaîne de caractères étant un tableau particulier de caractères, on utilisera l'opérateur [] pour accéder à ses éléments.

```
1  int compter_espace(char s[])
2  {
3      int cnt = 0, i = 0;
4
5      while (s[i] != '\0')
6      {
7          if (s[i] == ' ')
8          {
9              cnt++;
10         }
11         i++;
12     }
13     return cnt;
14 }
```

Utilisation de la bibliothèque standard

- La bibliothèque standard du C fournit un ensemble de fonctions permettant de manipuler des chaînes de caractères.
- Le fichier `string.h` centralise les déclarations de ces fonctions (attention, il n'existe pas de type string en C). Il faudra l'inclure si l'on souhaite les utiliser.
- `string.h` déclare plusieurs fonctions. Celles qui vont nous intéresser dans un premier temps sont
  - `strcpy`
  - `strlen`
  - `strcat`
  - `strcmp`

La fonction `strcpy` (string copy)

- Souvenez-vous que l'opérateur d'affectation ne permet pas de copier des tableaux. En particulier, il ne peut pas être utilisé pour copier des chaînes de caractères.
- Pour copier des chaînes, on peut utiliser la fonction `strcpy`. Par exemple, l'appel suivant

  ```
  strcpy(str1, str2);
  ```

  copie les caractères de `str2` jusqu'au caractère nul inclus dans `str1`.
- Attention, `strcpy` n'a pas de moyen de vérifier que `str1` est suffisamment grand pour contenir tous les caractères de `str2`.
- Dans le cas où la taille de `str1` est inférieure à celle de `str2`, un comportement indéfini se produit.
- Donc, toujours vérifier les tailles avant d'appeler `strcpy`.

La fonction `strncpy`

- La fonction `strncpy` permet de pallier le problème précédent.
- L'appel suivant

  ```
  strncpy(str1, str2, n)
  ```

  copie `n` caractères de `str2` dans `str1`.
- Un point important : `strncpy` n'insère pas automatiquement le caractère de terminaison.
- Corollaire du point précédent : si `n` est trop petit, `strncpy` ne copiera pas le caractère de terminaison. Cela peut être une source de bug !

La fonction `strlen` (string length)

- La fonction `strlen` retourne la taille d'une chaîne de caractères.
- Attention : la taille ici n'est pas celle du tableau sous-jacent.
- Il s'agit du nombre de caractères se trouvant avant le caractère nul (`'\0'`).

```
int len;
len = strlen("abc");      /* len vaut 3 */
len = strlen("");         /* len vaut 0 */
len = strlen("ab\0c");    /* len vaut 2 */
```

La fonction `strcat` (string concatenate)

- La fonction `strcat` permet de concaténer (mettre bout à bout) deux chaînes de caractères.
- L'appel

  `strcat(str1, str2)`

  concatène `str1` et `str2` dans cet ordre (`str1` puis `str2`). Le résultat est stocké dans `str1`.

```
/* contient a, b, c, d, \0, ?, ?, ? */
char str1[8] = "abcd";
/* contient z, y, x, \0 */
char str2[4] = "zyx";
/* str1 contient a, b, c, d, z, y, x, \0 */
strcat(str1, str2);
```

La fonction `strncat`

- ATTENTION : quand on utilise `strcat` il faut que `str1` soit suffisamment grand.
- La fonction `strncat` permet de contrôler le nombre de caractères que l'on ajoute en fin.
- L'appel suivant

  ```
  strncat(str1, str2, n)
  ```

  concatène les `n` premiers caractères de `str2` dans `str1`.
- `strncat` insère le caractère de terminaison (s'il y a suffisamment de place).

La fonction `strcmp` (string compare)

- La fonction `strcmp` permet de comparer deux chaînes de caractères.
- Le critère de comparaison utilisé par `strcmp` est celui de l'ordre lexicographique (ordre du dictionnaire).
- `strcmp` retourne une distance entre les deux chaînes passées en argument.
- L'appel suivant

  ```
  strcmp(str1, str2)
  ```

  produit
  - une valeur strictement négative si `str1` < `str2`
  - 0 si `str1` et `str2` sont identiques
  - une valeur strictement positive si `str1` > `str2`
- À quoi correspond le test suivant ?

  ```
  if (!strcmp(str1, str2))
  ```