

PROGRAMMATION C (ÉLÉMENTS DE BASE)



I.AZAIEZ GCR1 2023

1

Introduction (1/7)

2

□ **Principes et conseils :**

- ▣ la programmation ne requiert ni grande théorie, ni connaissances encyclopédiques. Les concepts utilisés sont rudimentaires mais c'est leur mise en œuvre qui est délicate. S'il n'y avait qu'un seul conseil à donner, ce serait la règle des trois "P" :

- Programmer
- Programmer
- Programmer

2

Introduction

(2/7)

3

Les algorithmes/les programmes sont utilisés pour résoudre une grande variété de problèmes concrets dans de nombreux domaines. Voici quelques exemples concrets de la manière dont les algorithmes sont appliqués pour résoudre des problèmes du quotidien :

Recherche sur Internet :

- Les moteurs de recherche tels que Google utilisent des algorithmes sophistiqués pour classer et trier des milliards de pages Web afin de fournir des résultats pertinents en réponse aux requêtes des utilisateurs.

Navigation GPS :

- Les algorithmes de navigation GPS calculent les itinéraires les plus courts ou les plus rapides entre deux points en utilisant des données cartographiques, des informations sur la circulation en temps réel et des algorithmes de recherche de chemin.

3

Introduction

(3/7)

4

Recommandation de produits :

- Les sites de commerce électronique comme Amazon et Netflix utilisent des algorithmes de recommandation pour suggérer des produits ou des contenus aux utilisateurs en fonction de leur historique d'achat ou de visionnage, ainsi que des modèles de recommandation collaboratifs.

Analyse de données :

- Dans le domaine de la science des données, les algorithmes sont utilisés pour extraire des informations utiles à partir de grands ensembles de données. Par exemple, l'algorithme de régression linéaire peut être utilisé pour prédire une valeur en fonction de variables indépendantes.

4

Introduction

(4/7)

5

Gestion des réseaux :

- Les algorithmes de routage sont utilisés pour gérer le trafic sur les réseaux informatiques, en déterminant la meilleure façon de faire passer les données d'un point à un autre tout en évitant les congestions.

Planification d'horaires :

- Les algorithmes de planification sont utilisés dans la gestion du temps, comme l'établissement de plannings de travail, la planification de vols, et même la création d'emplois du temps scolaires.

Santé :

- Les algorithmes sont utilisés dans l'analyse médicale, telle que la détection de maladies à partir d'images médicales (imagerie par résonance magnétique, tomodensitométrie), ainsi que dans la recherche de médicaments et la modélisation des épidémies.

Optimisation :

- Les algorithmes d'optimisation sont utilisés pour résoudre des problèmes d'optimisation, tels que la minimisation des coûts de production, la maximisation des bénéfices, la conception de réseaux de distribution efficaces, etc.

5

Introduction

(5/7)

6

Jeux vidéo :

- Les jeux vidéo utilisent des algorithmes pour simuler des mondes virtuels, gérer l'intelligence artificielle des personnages non joueurs, calculer des mouvements physiques réalistes, et bien plus encore.

Robotique :

- Les robots utilisent des algorithmes pour planifier leurs mouvements, éviter les obstacles, prendre des décisions en temps réel et accomplir des tâches spécifiques dans des environnements variés.

Cryptographie :

- Les algorithmes de chiffrement sont utilisés pour sécuriser les communications en ligne, les transactions financières et les données sensibles.

6

Introduction

(6/7)

7

- Quelques conseils de base :
 - ▣ **S'amuser** : C'est une évidence en matière de pédagogie.
 - ▣ **Bricoler** : Ce que nous voulons dire par là, c'est qu'il ne faut pas hésiter à tâtonner, tester, fouiller, faire, défaire, casser, etc. L'ordinateur est un outil expérimental.
 - ▣ **Faire volontairement des erreurs** : Provoquer les erreurs pendant la phase d'apprentissage pour mieux les connaître est le meilleur moyen de comprendre beaucoup de choses et aussi de repérer ces erreurs quand elles ne seront plus volontaires.
 - ▣ **Rester (le) maître** de la machine et de son programme.
 - ▣ **Debugger** : Un outil essentiel pour comprendre ce qui se passe dans un programme

7

Introduction

(7/7)

8

- Pour savoir ce qu'un ordinateur sait vraiment faire, il faut commencer par son organe principal : le micro-processeur.
 - ▣ Il sait exécuter une suite ordonnée d'instructions.
 - ▣ Il possède un petit nombre de mémoires internes appelées registres.
 - ▣ Il dialogue avec le monde extérieur via de la mémoire en plus grande quantité que ses registres.
 - ▣ Cette mémoire contient, sous forme de nombres, les instructions à exécuter et les données sur lesquelles travailler.
 - ▣ Les instructions sont typiquement :
 - Lire ou écrire un nombre dans un registre ou en mémoire.
 - Effectuer des calculs simples : addition, multiplication, etc.
 - Tester ou comparer des valeurs et décider éventuellement de sauter à une autre partie de la suite d'instructions.

8

La compilation

9



- La traduction en code natif ou en « byte code » d'un programme s'appelle la compilation.



- Un langage compilé est alors à opposer à un langage interprété. Dans le cas du C, C++ et de la plupart des langages compilés (VB.NET, C#, etc.)



- En résumé, la production du fichier exécutable se fait de la façon suivante :
 - Compilation : fichier source ! fichier objet.
 - Link : fichier objet + autres fichiers objets + bibliothèque standard ou autres ! fichier exécutable.

9

L'environnement de programmation

10

- L'environnement de programmation est le logiciel permettant de programmer.
- Sous Windows, l'extension (le suffixe) sert à se repérer les types de fichier :
 - ▣ Un fichier source C++ se terminera par « .cpp »
 - ▣ Un fichier objet sera en « .obj »
 - ▣ Un fichier exécutable en « .exe »
- Nous verrons aussi plus loin dans le cours : Les "en-tête" C++ ou headers servant à être inclus dans un fichier source : fichiers « .h »

10

Plan du cours	
11	
Chapitre 1	• Spécificités du Langage C
Chapitre 2	• Les variables scalaires et les opérateurs
Chapitre 3	• Les structures de contrôles
Chapitre 4	• Les fonctions d'entrées/sorties
Chapitre 5	• Les structures de données
Chapitre 6	• Types de données personnalisés
Chapitre 7	• Structurations des programmes C
Chapitre 8	• Les pointeurs

11

Chapitre 1	
Spécificités du Langage C	
12	
Généralité sur C	
• Il existe plusieurs compilateurs C sous différents systèmes d'exploitation : Windows, Linux. Ces compilateurs sont connus sous les noms suivants : Microsoft Visual studio, C++ Builder, Dev C++, Turbo C++, ...	
Conventions lexicales	
• On distingue :	
• Identificateurs :	
• Un identificateur est une séquence (ou suite) de lettres, de chiffre ou blanc souligné () il ne doit pas commencer par un chiffre :	
• Lettre = a, ...z ou A, ... Z, les lettres accentués ne sont pas admise.	
• Chiffre : 0...9	

12

Chapitre 1

Spécificités du Langage C



13

Conventions lexicales

- **Exemple :** `i`, `compteur`, `nb_car`
 - Attention : le langage C distingue entre les lettres majuscules et minuscules. Ainsi, `a` et `A` sont 2 identificateurs différents.
- **Les mots clefs ou clés :**
 - Dans un langage de programmation (ici C) un mot clé à une signification particulière.
 - Exp.: `if ... else`, `switch`, `while`, `do`, `for` → (voir chap3)
 - `auto`, `register`, `static`, `extern`.
- Les mots clés sont donnés en minuscules et respectant la définition d'un identificateur.

13

Chapitre 1

Spécificités du Langage C



14

Conventions lexicales

- **Les constantes :**
 - Les constantes de type caractère. Un caractère peut être :
 - une lettre minuscule : `a ... z`
 - une lettre majuscule : `A ... Z`
 - un chiffre : `0 ... 9`
 - ou autre : `{}, (), ?, !, ; $, &, / ...`
- Il existe 256 caractères codifiés de 0 à 255 selon la codification ASCII.

14

Chapitre 1

Spécificités du Langage C



15

Conventions lexicales

- Dans le langage C, un caractère est encadré (ou engendré) entre deux apostrophes. **Exp. :** 'a' '#' '0'
- A ne pas confondre : 'a' est une constante de type caractère et a est un identificateur
- **Constante entière :** Constante décimale (base 10) : il s'agit d'une séquence de chiffres ne commençant pas par zéro. **Exp. :** -123 ou 123
- **Constante octale (base 8) :** il s'agit d'une séquence de chiffres (0 à 7) qui commence par zéro. **Exp. :** 0123
- **Constante hexadécimale (base 16) :** il s'agit d'une séquence de chiffres, de lettres (A → 10, B → 11, ..., F → 15)
- **Constantes réelles :** -12.3 12.3 .123 Forme simple /12.3 E+15 Forme flottantes

15

Chapitre 1

Spécificités du Langage C



16

Conventions lexicales

- **Les commentaires**
- Un commentaire est une suite de caractère qui commence par /* et se termine par */
 - **Ex :** /* ceci est un commentaire se tient sur une seule ligne */
 - Ou // ceci est un commentaire se tient sur une seule ligne
 - /* voici un commentaire qui se
 - tient sur plusieurs lignes */
- En C, les commentaires ne peuvent pas être imbriqués
 - /* /* */ */
- **Les chaînes de caractères :**
 - Une chaîne de caractères et une suite de caractères encadrée entre deux guillemets.
 - **Ex :** "Bonjour !" ; " Titre" ; "GCR1"

16

Chapitre 2

Les variables scalaires et les opérateurs



17

Une variable en C doit être
pré-déclarée avant toute
utilisation.

On distingue :

Les variables
scalaires ou simples.

Les variables non
scalaires ou structurée
(voir chapitre 4)

17

Chapitre 2

Les variables scalaires et les opérateurs



18

Déclaration d'une variable simple

- **Syntaxe générale :**

- type_scalaire identificateurs 1, ..., identificateur n ;

Explication

- Identificateur i (i allant de 1 à n avec n fini) sont des identificateurs au sens du langage C. La virgule est obligatoire, il s'agit d'un séparateur.

18

Chapitre 2

Les variables scalaires et les opérateurs



19

- Le langage C offre plusieurs types scalaires, dont voici la liste :

Nom	Taille	Domaine de valeur
Char	1 octet	-128 à 127
int	2 ou 4 octets	Selon implémentation
short	2 octets	- 32768 à 32767
long ou long int	4 octets	-2147483648 à 2147483647
unsigned char	1 octet	0 à 255
unsigned short	2 octets	0 à 65 535
unsigned ou unsigned int	2 ou 4 octets	Selon implémentation
unsigned long	4 octets	0 à 4 294 967 295
Type à virgule flottante		
float	4 octets	$3.4 \cdot 10^{-38}$ à $3.4 \cdot 10^{38}$
double	8 octets	$1.7 \cdot 10^{-308}$ à $1.7 \cdot 10^{308}$
long double	16 octets	$3.4 \cdot 10^{-4932}$ à $3.4 \cdot 10^{4932}$

19

Chapitre 2

Les variables scalaires et les opérateurs



20

Initialisation d'une variable scalaire à la déclaration

- `unsigned i = 0 ;`
- `float delta = 3.2834;`
- `int x = 0, y , z ;`
 - Les variables y et z peuvent posséder des valeurs indéterminées.

En effet le langage C n'offre pas un mécanisme permettant d'initialiser par défaut (ou d'une façon automatique) toutes les catégories des variables.

20

Chapitre 2

Les variables scalaires et les opérateurs



21

Les opérateurs

- Les opérateurs arithmétiques
- Les opérateurs logiques
- Les opérateurs bit à bit
- Les opérateurs de décalage
- L'opérateur d'affectation
- Les opérateurs d'Incrémentation/de décrémentation
- L'opérateur conditionnel

21

Chapitre 2

Les variables scalaires et les opérateurs



22

Les opérateurs arithmétiques

- + * - / et % (reste de la division entière)
- + * - / % sont des opérateurs binaires. C'est-à-dire ayant la forme suivante :
- **op1 op op2** avec op1 est l'opérande gauche, op est un opérateur arithmétique et op2 est l'opérande droite
 - **Remarque :**
 - + et - sont également des opérateurs, unaire.
 - **Exemple :**
 - int x, y ;
 - x * y → binaire
 - x - y → binaire
 - -y → unaire

22

Chapitre 2

Les variables scalaires et les opérateurs



23

Les opérateurs relationnels ou de comparaison

- Ils sont tous des opérateurs binaires on distingue :
 - $<$, $>$, \leq , \geq , $=$ (égalité), \neq ()
- Le résultat d'un opérateur relationnel est un entier un tel résultat peut être :
 - 0 vaut dire faux
 - 1 vaut dire vrai
- Les opérateurs relationnels sont applicables sur les variables de type scalaire.

23

Chapitre 2

Les variables scalaires et les opérateurs



24

Les opérateurs logiques

- Ils sont en nombre de trois :
 - ET (AND), OU (OR) et NON (NOT)
- **&& (AND Logique):** Il s'agit d'un opérateur binaire ayant la forme suivante $Op_1 \&\& Op_2$.
 - Le résultat de cette expression ($Op_1 \&\& Op_2$) est vrai (1) si les deux opérandes sont évalués à vrai (1). Si le premier opérande (Op_1) est évalué à faux (0) alors le second opérande n'est pas évalué car le résultat est déjà connu (0).
- **|| (OR logique):** Il s'agit d'un opérateur binaire ayant la forme suivante $Op_1 || Op_2$.
 - Le résultat de cette expression ($Op_1 || Op_2$) est faux ssi Op_1 et Op_2 sont faux. Si le premier opérande est évalué à vrai (1) alors le second opérande n'est pas évalué car le résultat est déjà connu (1).

24

Chapitre 2

Les variables scalaires et les opérateurs



25

➤ Exemple d'opérateurs logiques

- **! (not logique)** Il s'agit d'un opérateur unaire ayant la forme suivante : **! op**

- ▣ Si op est évalué à vrai (1) alors ! op est faux (0).

- **Règle de De Morgan**

$$!(op1 \ \&\& \ op2) \rightarrow (!op1) \ || \ (!op2)$$

$$!(op2 \ || \ op2) \rightarrow (!op1) \ \&\& \ (!op2)$$

- **Illustration :**

$$!(((i \geq 1) \ \&\& \ (i \leq 100)) \ || \ (i == 200))$$

$$\rightarrow ((i < 1 \ || \ (i > 100)) \ \&\& \ (i \neq 200))$$

25

Chapitre 2

Les variables scalaires et les opérateurs



26

Les opérateurs bits à bits

- Bit = 0/1 la plus petite information mémorisable dans la mémoire de l'ordinateur. Ces opérateurs bit à bit sont applicables sur des entiers (int, unsigned, char, long) qui sont considérés comme une séquence de bits. Les opérateurs de bits exécutent les opérateurs logiques ET (&), OU (!), OU exclusif (^) et NON (-) sur tous les bits, pris un à un, de leurs opérandes entiers.

bit1	bit2	- bit1	bit1 & bit2	bit1 ! bit2	bit1 ^ bit2
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

26

Chapitre 2

Les variables scalaires et les opérateurs



27

L'opérateur d'affectation

- **Syntaxe générale :**
 - variable = expression ;
- **Explication :**
 - Variable est une variable scalaire précédemment déclarée
 - = est le symbole affectation

L'expression peut être :

- Une Constante
- Une variable
- Un mélange de constante et de variable avec des Opérateurs (arithmétique, relationnels, logique, Bit à Bit ou de décalage)

27

Chapitre 2

Les variables scalaires et les opérateurs



28

Incrémentation / décrémentation

- **Incrémentation : (incrémenter)**

$$x = x + 1; \leftrightarrow x += 1; \leftrightarrow x++ \leftrightarrow ++x;$$
 - ++ Opérateur d'incrément il s'agit d'un opérateur unaire, il exige un seul opérande.
 - $X+1 \rightarrow$ Écriture infixée
 - $X++ \rightarrow$ Post fixée (ou Post incrémentation)
 - $++X \rightarrow$ Préfixé (ou Pré incrémentation)
- **Décrémentation : (décrémenter)**
 - Il s'agit d'un opérateur unaire, il exige un seul opérande.

$$x = x - 1; \leftrightarrow x -= 1; \leftrightarrow x-- \leftrightarrow --x;$$

28

Chapitre 2

Les variables scalaires et les opérateurs



29

L'opérateur conditionnel

- Il s'agit d'un opérateur **ternaire** c'est-à-dire il met en relation **trois expressions**.
- Un tel opérateur a la forme suivante :
 - Si $v1 > v2 \leftarrow \text{exp1}$
 - Alors
 - $v = v1 \leftarrow \text{exp2}$
 - Sinon
 - $v = v2 \leftarrow \text{exp3}$
- Si l'exp1 a rendu une valeur non nulle alors exp2 est évaluée sinon (valeur nulle rendue par exp1) exp3 est évaluée (c'est le résultat de l'opérateur ternaire).

29

Chapitre 3

Les structures de contrôles



30

Ce sont des constructions fournis par C permettant de contrôler le déroulement des programmes pendant qu'ils s'exécutent.

On distingue :

Schémas conditionnels

Schémas itératifs ou les boucles

L'instruction break, ...

30

Chapitre 3

Les structures de contrôles



31

- **Schémas conditionnels**
- On distingue : l'alternance et la sélection
 - ▣ L'alternance ou l'instruction test if
 - ▣ Objectif: choisir un traitement parmi deux traitements prévus
 - ▣ Syntaxe générale :


```
if ( exp)
    inst 1 ;
[ else
    inst2 ; ]
inst3;
```
- **Explication :**
 - ▣ if et else sont deux mots réservés
 - Tout ce qui figure entre [] est optionnel
 - ▣ Instruction i (1) peut être simple ou composée.
 - ▣ exp peut être de type entier ou réel.

31

Chapitre 3

Les structures de contrôles



32

- Le langage de programmation C offre une autre forme (else) de l'instruction if, dont voici sa syntaxe générale :

```
if (exp 1)
    inst 1 ;
else if ( exp 2)
    inst 2;
...
else if ( exp n)
    inst n
else
    inst n+1;
inst n+2;
```

if et else imbriquées

32

Chapitre 3

Les structures de contrôles



33

- la sélection ou branchement à choix multiple ou instruction **switch** : Elle permet de sélectionner un traitement parmi plusieurs traitements prévus en fonction de la valeur rendue par une expression.

- Syntaxe générale :

```
switch (expression)
{
    case const_1 : traitement 1 ;
                    [break ;]
    case const_2 : traitement 2 ;
                    [break ;]
    ...
    case const_N : traitement N ;
                    [break ;]
    [ default      : <traitement N+1>;]
                    [break ;] }
```

33

Chapitre 3

Les structures de contrôles



34

- la sélection ou branchement à choix multiple ou instruction **switch** :

- Explication :

- switch, case et default sont des mots clés ou réservés.
 - expression doit être de type entier ou caractère.
 - Traitement i (i 1 → n+1) est une suite d'instruction simple.
 - Tout ce qui figure entre [] est optionnel
 - break est une instruction simple fournie par C permettant de sortir de l'instruction switch.


- Sémantique ou effet :

- Elle permet de sélectionner un traitement (soit traitement i) en fonction de la valeur de l'expression. Le cas exprimé par défaut traduit les autres cas.

34

Chapitre 3

Les structures de contrôles



35

Les schémas itératifs ou les boucles

Un schéma itératif ou boucle permet d'exprimer un traitement qui se répète n fois, ce dernier est souvent fini.


Le langage C offre trois constructions itératives ou boucle :

while, do, for.

35

Chapitre 3

Les structures de contrôles



36

- Schéma while
 - ▣ Syntaxe générale :

```
while ( expression )  
    instructions ;
```
 - ▣ Explications :
 - while est un mot réservé
 - Expression peut être de type entier, réel ou caractère.
 - Instruction peut être simple ou composée.

36

Chapitre 3

Les structures de contrôles



37

- **Schéma do**
- Contrairement à la structure **while** la boucle **do ... while** teste sa condition de sortie après exécution des instructions du corps de la boucle.
- **Syntaxe générale :**

```
do {
    instruction(s) ;
} while (expression) ;
```
- **Explication :**
 - ▣ do est un mot clé,
 - ▣ instruction peut être simple ou composée appelée corps,
 - ▣ expression peut être de type réel, entier ou caractère.
- **Sémantique ou effet :**
 - ▣ Faire instruction jusqu'à l'expression soit nulle.
 - ▣ Par rapport au schéma while, le corps du schéma do est exécuté au moins 1 fois.

37

Chapitre 3

Les structures de contrôles



38

- **Schéma for**
 - ▣ **Syntaxe générale**

```
for ( instruction 1 ; expression ; instruction 2 )
    instruction 3 ;
```
 - ▣ **Explication :**
 - for est un mot clé
 - inst 1 est une inst simple
 - inst 2 est une inst simple
 - inst 3 peut être simple ou composée
 - Expression idem que while et do

38

Chapitre 3

Les structures de contrôles



39

- Pourquoi le découpage inst 1, inst 2, et inst 3 proposé par for ?
 - ▣ inst 1 : doit fixée les instructions initiales pour démarrer un traitement itératif.
 - ▣ expression : condition de bouclage
 - ▣ inst 2 : exprime le traitement permettant d'agir progressivement sur l'expression si on souhaite une boucle finie.
 - ▣ inst 3 : exprime le traitement qui va se répéter.
- Conditions initiales :


```
x = 0 ;
i = 0 ; } → inst 1
```
- agir sur l'expression


```
i++ ; → inst 2
```
- Traitement itératif


```
x = x + 3 ; → inst 3
```

39

Chapitre 3

Les structures de contrôles



40

- Architecture simplifiée d'un programme écrit en C

int main () → en tête

```
{
  PD : partie déclarative
  PE : partie exécutive
  return 0;
}
```


- Exemple :

```
int main( )
{ unsigned i;
  float j ;
  int sortie = 0 ;
  i = 0 ;
  j = 1.0 ;
  while (i < 30){
    j = j * 1.5 ;
    if (j >= 75.5)
      break ;
    i++ ;
  }
  sortie ++ ;
  return 0;
} // fin main
```

40

Chapitre 4

Les fonctions d'entrées/sorties




41

- Ce sont des fonctions formées par C et regroupés dans un fichier appelé "stdio.h" permettant d'échanger des données entre le programme et l'extérieur.
- Un programme vis à vis de l'utilisateur consomme des données et délivre des résultats.
- Les données nécessaires du fonctionnement d'un programme peuvent provenir du clavier.
- Les résultats produits par un programme, peuvent être affichés sur l'écran (message de sortie standard)
- Pour y parvenir, le langage fournit :
 - des fonctions d'entrées/sorties non formatées.
 - des fonctions d'entrées/sorties formatées

41

Chapitre 4

Les fonctions d'entrées/sorties



42

- Les fonctions d'entrées/sorties non formatées
 - ▣ Entrée non formatée
 - Syntaxe :


```
int getchar( )
```
 - ▣ Explication :
 - Elle lit et retourne un caractère provenant du clavier.
 - Exception : cette fonction `getchar()` rend une valeur constante équivalente EOF (qui vaut -1) si une erreur a survenue.

42

Chapitre 4

Les fonctions d'entrées/sorties



43

□ Exemple

- ▣ Ecrire un programme C permettant de lire un texte tapé au clavier et de compter les chiffres, les lettres (maj et mins) et les caractères spéciaux tels que retour à la ligne (nouvelle ligne) et tabulation. Un texte est une suite de caractères qui se termine par un caractère indiquant la fin. Un tel caractère est '#'.

43

Chapitre 4

Les fonctions d'entrées/sorties



44

□ Sortie non formatée

▣ Syntaxe

```
putchar ( int c ) ;
```

▣ Explication

- Elle affiche sur l'écran le contenu de "c". En réalité le caractère qui correspond au code mémorisé dans c.

▣ Exemple

- écrire un programme qui affiche tous les alphabets minuscules et majuscules.

44

Chapitre 4

Les fonctions d'entrées/sorties



45

- Les fonctions d'entrées/sorties formatées
 - ▣ Formaté signifie ici contrôlé la forme et le format des données.
 - ▣ Format est un mélange de chaînes de caractères écrites telles quelles et des indications de conversion introduites par le caractère % suivi d'une indication de tailles, d'un signe - optionnel indiquant un cadrage (par défaut à droite) et d'une modification de taille (lettre l ou L) pour les entiers long ou double, puis le type de la donnée pour conversion.

45

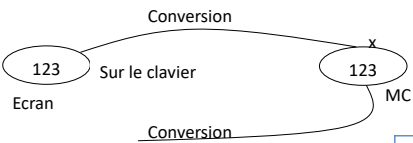
Chapitre 4

Les fonctions d'entrées/sorties



46

- Les fonctions d'entrées/sorties formatées



Les types de données
formés par C

Format	Type de données
% d	Entier décimal
% u	Décimal non signé (unsigned)
% x, X	Entier Hexadécimal
% c	Caractère (char)
% f	Réel (float)
% s	Chaîne de caractère
% p	Pointeur

46

Chapitre 4

Les fonctions d'entrées/sorties



47

□ Les fonctions d'entrées/sorties formatées

▣ Sortie formatée

Pour l'affichage formaté, on utilise la fonction `printf`.

```
printf ("format", [exp 1, ..., exp n]) ;
```

```
printf ("expression") ;
```

▣ Exemple :

```
printf ("%d plus %d donne %d ", 1000, i, i+1000) ;
```

```
printf(" bonjour GCR1 ");
```

47

Chapitre 4

Les fonctions d'entrées/sorties



48

□ Les fonctions d'entrées/sorties formatées

▣ Entrée formatée

La fonction *scanf* autorise la saisie formatée de données (depuis le clavier).

▣ Exemple

```
scanf ("%d %d %d", &a, &b, &c);
```

□ Application

- ▣ Ecrire un programme C permettant l'échange des valeurs de deux variables `x` et `y`. Prévoir la lecture et l'affichage des données.

48

49

Chapitre 5

Les structures de données

49

Chapitre 5

Les structures de données



50

- Le langage C dispose de deux types de données complexes : **les tableaux et les structures**.
- La différence entre les deux provient de la nature des éléments qu'on peut y ranger. Un tableau ne contient que des données identiques, alors qu'une structure peut être composée à partir d'éléments dissemblables.
- Ces structures de données, sont appelés type structuré ou type non scalaire. Le type des éléments du tableau peut être n'importe quel type du langage C.

50

Chapitre 5

Les structures de données



51

□ Les tableaux

▣ Indexation

- On identifie chaque élément du tableau par un nombre spécifique. Appelé indice ou index. L'index de cet élément étant différent de tous les autres index. L'index doit être entier positif (y compris 0).
- L'indexation commence toujours par 0 (et non pas par 1).

▣ Remarque :

- [index max] = nbre d'éléments du tableau-1

▣ Initialisation à la définition

- Syntaxe: tableau à une dimension :

<type> <nom du tableau> [<nbre d'éléments >] = {K1, K2,.....,Kn};

51

Chapitre 5

Les structures de données



52

□ Les tableaux

- ▣ Opération d'E/S avec les éléments des tableaux
- ▣ Exemple

```
printf ("%d \t %d \t %d \t %d \t %d ",v[0] ,v[1] ,v[2] ,v[3] ,v[4]);
```

Par une boucle.

```
for (k=0 ; k<5 ; k++)
{
    printf (" v[%d] \t ", k);
    scanf ("%d ", & v[k]);
}
```

52

Chapitre 5

Les structures de données



53

□ Tableaux à plusieurs dimensions

▣ Syntaxe

■ `<type> <nom du tableau> [e1] [e2].....[en];`

▣ Exemple

■ `int mat [3][4];`

▣ Initialisation

■ A la définition

`int mat [3][4] = { {1,2,3,4},{5,6,7,8},{9,10,11,12}};`

53

Chapitre 5

Les structures de données



54

□ Tableaux à plusieurs dimensions

▣ Initialisation par une boucle

```
for (i = 0; i < 3 ; i ++ )  
    for ( j = 0 ; j < 4 ; j ++ )  
    { printf (" mat[%d][%d]", i,j);  
      scanf ("%d", &mat[i][j]);  
    }
```

54

Chapitre 5

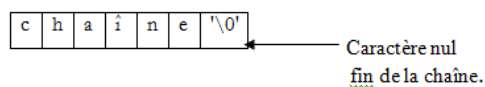
Les structures de données



55

□ Chaînes de caractères (String)

- Un tableau char doit toujours être créé avec un élément (au moins) de plus que le nombre de caractère du string à mémoriser.



- Les strings doivent se terminer par un caractère spécial. Ce caractère matérialisant la fin d'un string est le caractère nul '\0'.

55

Chapitre 5

Les structures de données



56

□ Chaînes de caractères (String)

■ Opérations de chaînes

- **strcmp** : Comparaison de chaîne de caractères
Syntaxe : `strcmp (<adresse_tableau1>, <adresse_tableau2>);`
- **strcat** : concaténation de deux chaînes de caractères
Syntaxe : `strcat (<adresse_tableau1>, <adresse_tableau2>);`
- **strlen** : longueur d'une chaîne de caractères
Syntaxe : `strlen(<adresse_tableau>);`
- **strcpy** : copiage d'une chaîne dans une autre
Syntaxe : `strcpy (<adresse_tableau1>, <adresse_tableau2>);`

NB. Ces fonctions nécessitent : `# include <string.h>`

56

Chapitre 5

Les structures de données



57

- Application : tri d'un tableau
 - ▣ On désigne par "tri" l'opération qui consiste à ordonner un ensemble d'éléments suivant un ordre croissant ou décroissant. Il existe plusieurs algorithmes de tri tels que : le tri bulle, le tri par insertion, le tri par sélection (par extraction simple), ...
 - ▣ Tri bulle : Le principe du tri bulle est de comparer deux à deux les éléments (tab[i] et tab [i+1]) consécutifs d'un tableau et d'effectuer une permutation si $tab[i] > tab[i+1]$. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

57

Chapitre 5

Les structures de données



58

Algorithme	<pre>VARIABLE permut : Booleen REPETER permut = FAUX POUR i VARIANT DE 1 à N-1 FAIRE SI a[i] > a[i+1] ALORS echanger a[i] et a[i+1] permut = VRAI FIN SI FIN POUR TANT QUE permut = VRAI</pre>
Programme C	Ecrie le programme C

58

Chapitre 5

Les structures de données

59

□ Exercice 1 :

```
#include<stdio.h>

void main()
{
    int i, j ;
    int k[3][4]={ {1,2,3},{4,5,6,7},{9,1}} ;
    for( i=0 ; i<4 ; i++)
        {for(j=0 ; j<3 ; j++)
            printf("k[%d][%d] \t", i, j) ;
        }
}
```

Question :
Corrigez le programme pour qu'il affiche la matrice sous cette forme :

1	2	3	0
4	5	6	7
9	1	0	0

59

Chapitre 5

Les structures de données

60

□ Exercice 2 :

- Ecrire un programme en C qui compte le nombre de chacune des lettres de l'alphabet et des chiffres d'un texte entré au clavier. Pour simplifier, on ne tiendra compte que des minuscules.
- Le programme devra accepter un nombre quelconque de lignes.

60

Chapitre 5

Les structures de données

61

□ Les structures

▣ Syntaxe

```
struct <nom_Structure>
{
    < type champ 1> < Nom_champ 1>;
    < type champ 2> < Nom_champ 2>;
    ....
    < type champ n> < Nom_champ n>;
};
ou [id 1], [id 2], ..., [id n];
```

struct

client

```
{
    char Nom [20];
    int cp;
    char ville;
};
```

Mot clé pour définir une structure

Nom du type de la structure

Structure dont le type est appelé "client"

Champ de la structure

61

Chapitre 5

Les structures de données

62

□ Les structures

▣ Opération sur les variables structurées

```
struct livre
{
    [ struct personne auteur; ]
    char nom_auteur [12];
    char titre [20];
    int an;
} ;
```

Ou encore

```
struct personne
{
    char prénom [15];
    char nom [15];
};
```

62

l.azaiez

31

Chapitre 5

Les structures de données



63

□ Les structures

▣ Accès aux champs d'une structure :

■ Syntaxe

<Nom _ Variable>.< Nom _ champ>;

▣ Les champs de la variable structurée l sont donc accessibles via les noms : l.nom_auteur, l.titre, l.an.

▣ Exemple

l.an = 1993.

strcpy (l.nom_auteur, "Molaire");

▣ Initialisation

Struct livre l = {"Dupond ", "progC", 2012};

63

Chapitre 5

Les structures de données



64

□ Tableau de structure

struct livre l [10];

l[0]	nom_auteur	titre	an	ville
l[1]	nom_auteur	titre	an	ville
l[i]
l[9]	nom_auteur	titre	an	ville

} Tableau bidimensionnel

□ Pour accéder à un certain champ d'un quelconque élément du tableau :

▣ Syntaxe :

< nom _ tableau > [index] .< nom _ champ >;

64

65

Chapitre 6

Types de données personnalisés

65

Chapitre 6

Types de données personnalisés



66

- Vous, permettent de définir vos propres types de données qui viendront s'ajouter aux types prédéfinis en C.
- Les deux mots clé:
 - ▣ `typedef`
 - ▣ `enum`
- Le mot `typedef` autorise simplement la définition de nouvelles appellations pour des types de données déjà existants. Avec `enum`, en revanche, on peut réellement définir un nouveau type de données.

66

Chapitre 6

Types de données personnalisés



67

▣ typedef

▣ Syntaxe :

```
typedef <type> <nom_Remplacement1>, <nom_Remplacement 2>, ..,<N>
```

- ▣ Permet de définir un ou plusieurs autres noms en tant que synonymes d'un type donné.

▣ Exemple :

```
struct AdresseClient a ; → typedef struct AdresseClient adcl;
```

- ▣ adcl définit comme synonyme de type "struct AdresseClient".

- ▣ Par la suite, la définition de variable :

```
adcl a; ⇔ struct AdresseClient a ;
```

67

Chapitre 6

Types de données personnalisés



68

▣ typedef

- ▣ En ce qui concerne les structures, on peut définir un type personnalisé en même temps qu'on déclare la structure.

```
typedef struct AdresseClient
{
    char rue [20];
    int code;
    char ville[30];
} adcl ;
```

- ▣ Ne définit pas une variable du nom "adcl" mais un **type de structure** nommée "adcl" qui pourra être utilisé comme **synonyme** de type **struct AdresseClient**.

68

Chapitre 6

Types de données personnalisés



69

□ Enum

- ▣ Ce nom dérive de l'anglais "enumerate" qui signifie "énumérer".

▣ Syntaxe :

```
enum nom_des_types_énumération
{
    nom 1,
    nom 2,
    ....
    nom n,
};
```

69

Chapitre 6

Types de données personnalisés



70

□ Enum

▣ Exemple

Mot clé → `enum` `Continents` ← Nom de type énumération

```
{ Afrique,
  Amerique,
  Asie,
  Europe,
}; ou } conti ;
```

← Liste des valeurs

- ▣ Définit une **variable** nommée "**conti**" dont le **type** est "**enum continents**".
- ▣ Cette variable ne peut prendre qu'une des valeurs contenues dans la liste spécifiée lors de la déclaration.

70

71

Chapitre 7

Structurations des programmes C

71

Chapitre 7

Structurations des programmes C



72

Un programme écrit en C comporte en général les éléments suivants :

- Constants, types et variables.
- Le programme principal ou la fonction appelé main.
- Les sous programmes : fonctions et procédure
- Les macro-instructions (# include, # define).

A propos des constantes C:

- Nommée
- Non nommée : 0.314, -3, 'a'.

72

Chapitre 7

Structurations des programmes C



73

- Pour définir une constante nommée, le langage C offre une macro-instruction ayant la forme suivante:

define **identification** **valeur**

- **Exemple :**

```
# define n 10
void main
{ int tab[n]
  ...
}
```

73

Chapitre 7

Structurations des programmes C



74

- **Portée des variables**
- Le langage C offre des moyens au programmeur permettant de contrôler la portée d'une variable, sa durée de vie et éventuellement son implantation en mémoire. Une variable en C doit être déclarée avant toute utilisation.
 - **Syntaxe générale :**
[Classe_d'allocation] type id1 , id2, ..., idn;
// id_i sont des variables
 - **Explication :**
 - Classe_d'allocation peut être **auto**, **register**, **static** ou **externe**. Ces derniers sont des mots réservés.
 - Type peut être simple ou structuré.

74

Chapitre 7

Structurations des programmes C



75

- De point de vue de la portée d'une variable, on distingue :
 - ▣ Variables locale (auto, register ou static)
 - ▣ Variables globales (static ou extern).
- Du point de vue de la durée de vie d'une variable, on distingue :
 - ▣ les variables provisoires (auto ou register).
 - ▣ les variables permanentes (static ou extern).

75

Chapitre 7

Structurations des programmes C



76

- Variables locales
- Ces variables ont une portée réduite au bloc où elles sont déclarées ou définies. Par bloc on distingue :
 - ▣ Bloc procédure (main ou sous – programme).
 - ▣ Bloc d'instruction.

76

Chapitre 7

Structurations des programmes C



77

□ Variables locales

- ▣ On dit que la variable *i* a une portée réduite au bloc où elle a été définie.
- ▣ Concernant la durée de vie d'une variable locale est égale à celle du bloc où elle a été définie.
- ▣ La durée de vie d'un bloc est le temps nécessaire à son exécution.
- ▣ Une variable locale est par défaut de classe d'allocation auto.

77

Chapitre 7

Structurations des programmes C



78

□ Variables locales static

- ▣ Une variable locale **static** correspond, plus ou moins, à une variable **globale** accessible uniquement dans le contexte d'une fonction donnée. Mais sa durée de vie correspond bien à une variable globale : en conséquence son état est maintenu entre chaque appel à la fonction ayant définie la variable locale statique. Voici un petit exemple d'utilisation.

78

Chapitre 7

Structurations des programmes C



79

□ Les variables globales

- Les variables globales sont déclarées en dehors de la fonction main et également en dehors de sous-programme.
- Une variable globale est visible pour tous les sous-programmes (y compris main) qui viennent textuellement après cette variable.

79

Chapitre 7

Structurations des programmes C



80

□ Les variables globales

- Le langage de programmation C donne la possibilité de répartir physiquement un programme sur plusieurs fichiers
- Une variable globale peut être :
 - Soit de portée réduite au fichier où elle a été déclarée. Elle aura la classe d'allocation **static**
 - Soit de portée étendue sur tout le programme. Elle aura la classe d'allocation **extern**.
- Une variable globale est une variable permanente qui a une durée de vie égale à celle du programme c'est-à-dire de main.

80

Chapitre 7

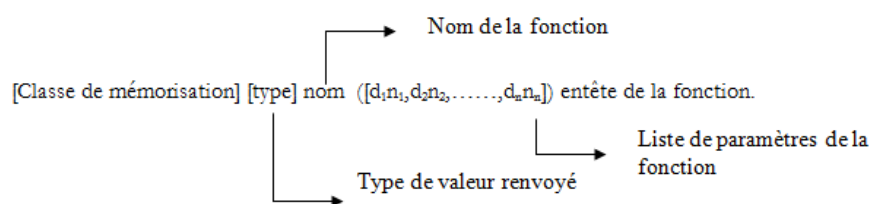
Structurations des programmes C



81

□ Les sous programmes

- ▣ On distingue : les fonctions et les procédures.
- ▣ Syntaxe générale :



81

Chapitre 7

Structurations des programmes C



82

□ Les sous programmes

□ Procédure :

`void <Nom_Procédure> ([paramètres formels]) {PD PE}`

□ Fonction :

```
<type_de_retour> <Nom_Fonction> ([paramètres
formels])
{
    [Déclaration des variables]
    [Partie exécutive : une séquence d'instruction]
}
```

82

Chapitre 7

Structurations des programmes C



83

□ Les sous programmes

▣ A propos de l'instruction return :

- Cette instruction est formée par C. Elle met fin à l'exécution d'un sous programme (procédure ou fonction)

▣ Syntaxe générale

(1) `return (expression);`

Ou

(2) `return;`

- La variable (1) est utilisée au sein de la partie exécutée d'une fonction. Ainsi l'expression doit être de même type que la fonction.
- La variable (2) peut être utilisée au sein de la partie exécutive d'une procédure.

83

Chapitre 7

Structurations des programmes C



84

□ Les sous programmes

- ▣ Dans la littérature concernant le langage C, une procédure est appelée également une fonction de type void.
- ▣ L'imbrication ou l'emboîtement des sous programmes n'est pas autorisée en C.
- ▣ À l'intérieur d'un sous programme on ne peut pas définir un sous programme.

84

Chapitre 7

Structurations des programmes C



85

□ Appels des sous-programmes

- L'appel d'un sous-programme prend la forme suivante :

`Nom_sous_programme (p1,p2,...,pn);`

// Avec p_i ($i = 1, \dots, n$) sont des paramètres effectifs.

- Lors de l'appel d'un sous-programme, on distingue:

□ Deux environnements

- Environnement appelant (exemple main)
- Environnement appelé (exemple fonction `f()`)

85

Chapitre 7

Structurations des programmes C



86

□ Remarque

- Les paramètres formels ont une portée réduite au sous programme où ils sont définis.
- L'appel d'un sous-programme suppose sa définition.
- Un sous-programme est défini une et une seule fois. Il peut être appelé (ou utilisée ou activé) au tant de fois (0 à n) que l'application exige.
- Les paramètres (qui sont des exemples) fournis lors de l'appel d'un sous-programme sont dites paramètres effectifs.

86

Chapitre 7

Structurations des programmes C



87

Les fonctions récursives : n!

```
int fact1(int value ) {
    if ( value <= 1 )
        return 1;
    //Appels récursifs
    return value*fact1(value-1);
}
```

fonction factorielle en version itérative

```
int fact(int value ) {
    int result = 1;
    while ( value > 1 ) {
        result *= value;
        value --;
    }
    return result;
}
```

```
int main() {
    printf( "3! = %d\n", fact( 3 ) );
    printf( "3! = %d\n", fact1( 6 ) );
    return 0; }
```

87

Chapitre 7

Structurations des programmes C



88

□ Les fonctions réutilisables

- Le but est de factoriser le code qui peut être réutiliser un grand nombre de fois au travers de votre (vos) programme(s).
- Exemple :
 - Définition d'une fonction d'élévation à une puissance données.

88

Chapitre 7

Structurations des programmes C



Fonctions réutilisables

```

1 // puiss.h
2 #ifndef maFonction_h
3 #define maFonction_h
4
5 int puiss( int value, unsigned int pow ){
6     if ( pow == 0 ) return 1;
7     if ( pow == 1 ) return value;
8     int accumulator = 1;
9     while( pow > 0 ) {
10         accumulator *= value;
11         pow--;
12     }
13     return accumulator;
14 }
15 #endif /* maFonction_h */

```

Main pour le test

```

16 int main() {
17
18     int result = puiss( 2, 3 );
19     printf( "2^3 == %d\n", result );
20     return 0;
21 }

```

89

Chapitre 7

Structurations des programmes C



90

- Le mot clé **extern** s'utilise traditionnellement lors d'une déclaration de donnée dans un fichier d'entête (.h). Il permet d'indiquer au compilateur que la donnée existe bien et quel est son type, mais précise surtout au compilateur de ne pas réserver un emplacement mémoire maintenant et résoudre cet emplacement mémoire plus tard : durant de la phase d'édition des liens.
- Nous allons envisager un petit exemple basé sur trois fichiers :
 - ▣ un fichier d'entête (un .h),
 - ▣ deux fichiers d'implémentation (deux .c).

90

Chapitre 7

Structurations des programmes C



91

Entête (.h)

```

9 #ifndef commun_h
10 #define commun_h
11 extern int globalCounter;
12 // La variable est déclarée
13 //mais non réservée en mémoire
14
15 void doSomething(void);
16
17 #endif /* commun_h */

```

Implémentation (.c)

```

8 #include <stdio.h>
9 #include "imp.h"
10 // On réserve la mémoire
11 //pour la variable globale et on l'initialise.
12 int globalCounter = 0;
13 // On implémente la fonction doSomething
14 void doSomething() {
15     globalCounter ++;
16     printf( "doSomething invoked\n" );
17 }

```

```

int main() {
    printf( "Counter == %d\n", globalCounter );
    // Affiche 0
    doSomething();
    printf( "Counter == %d\n", globalCounter );
    return 0;
}

```

91

92

Chapitre 8

Les pointeurs

92

Chapitre 8: Les pointeurs



93

□ Introduction (1/2)

- ▣ Dans nos précédant programmes, l'accès à une variable (plus précisément à son contenu ou à sa valeur) se faisait par l'intermédiaire de son nom.
- ▣ Si par exemple on devait affecter à une variable "**a**" la valeur d'une autre variable "**b**", on y arrivait par l'instruction **a = b** ; dans laquelle un nom de variable figurait des deux côtés de l'opérateur d'affectation « = »

93

Chapitre 8: Les pointeurs



94

□ Introduction (2/2)

- ▣ Une variable n'est donc rien d'autre qu'une zone mémoire portant un certain nom défini par le programmeur.
- ▣ Mais on peut aussi choisir un chemin d'accès indirect par le biais de l'adresse de la variable. Pour ce la, on utilise ce qu'on appelle un **pointeur**.

94

Chapitre 8: Les pointeurs

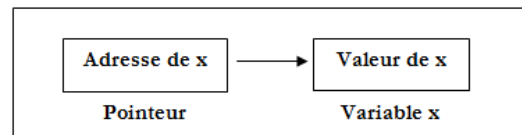


95

□ Définition (1/2)

- ▣ Un pointeur (pointer) est une donnée constante ou variable qui mémorise l'adresse d'une variable. On dit que le pointeur « **pointe** » vers la variable concernée, ce la via l'adresse de la variable.
- ▣ On dit aussi que le pointeur fait **référence à la variable**.

Mémoire



Variable pointeur
mémorise l'adresse d'une
donnée

95

Chapitre 8: Les pointeurs



96

□ Définition (2/2)

- ▣ Il existe aussi des pointeurs constants (adresse constante)
- ▣ **Exemple** : les noms des tableaux sont des pointeurs constants équivalents à l'adresse (de début) du tableau concerné
- ▣ **Variables et adresses** :


```

int x ;
&x ← adresse de x dans la mémoire en Hexadécimal.
int tab[10] ;
tab ← adresse du premier élément du tableau (&tab[0])
      
```

96

Chapitre 8: Les pointeurs



97

□ Définition de variables pointeurs

▣ Syntaxe :

< type-données> * < nom-pointeur> ;

▣ Exemple :

int * pi ; ← la variable "pi" est un pointeur vers une donnée de type int.

char * pc ; ← pc un pointeur vers un "char".

■ Peut contenir l'adresse d'une donnée de type "char".

■ pc à le type "char*"

97

Chapitre 8: Les pointeurs



98

□ Définition de variables pointeurs

▣ Initialisation :

int x ; // variable ordinaire

int * pi ; // variable pointeur

float *pf ;

float y ;

▣ L'opérateur unaire & appliqué à une variable permet d'obtenir l'adresse de cette variable.

■ Soit : pi = &x ; pi est un pointeur vers int, initialisé avec l'adresse de la variable x ;

▣ Affichage : printf ("% x",&x) ; // Hexadécimal

Ou "%d" //decimal

98

Chapitre 8: Les pointeurs



99

□ Accès indirect aux variables

- ▣ Pour pouvoir accéder, via un pointeur, à une autre donnée, on a besoin de l'opérateur "*" l'opérateur unaire "*" : appelé opérateur d'indirection (opérateur de référence).

- ▣ **Remarque** : L'utilisation de l'opérateur d'indirection "*" doit être ici strictement différenciée de son utilisation dans les définitions de variables pointeurs.

▣ Exemple :

1. `x=3;`
2. `pi=&x ;`
3. `y =*pi ; ⇔ y=x;`

99

Chapitre 8: Les pointeurs



100

□ Les opérateurs définis sur les pointeurs

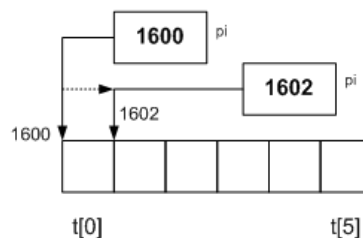
▣ Addition

- Soit : `int t [6] ;`
`int * pi ;`

- Alors : `pi =& t[0] ;` // pi pointe sur le premier élément du tableau 't' vers la variable "int" t[0]

L'instruction :

- ```
pi=pi + 1 ;
// l'adresse dans
//pi est passée
//de 1600 à 1602
```



100

## Chapitre 8: Les pointeurs



101

### □ Incrémentation et décrémentation

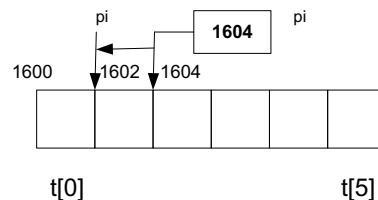
#### ▣ Incrémentation :

Soit : `int i[10];`  
`pi = & i[0];` avec `int * pi;`  
`pi = pi + 1 ;` ⇔ `pi ++ ;` ou `++ pi ;`  
`(*pi) ++ ;` // On augmente de 1 la valeur de la  
//donnée pointée actuellement par pi.

#### ▣ Décrémentation :

Soustraction :

`pi = pi - 1 ;` ⇔ `pi -- ;` ou `-- pi ;`



101

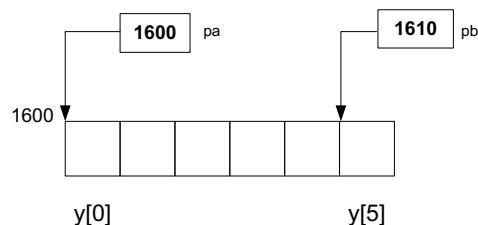
## Chapitre 8: Les pointeurs



102

### □ Comparaison

Soit : `short y[6];`  
`short * pa = &y[0], * pb = &y[5];`



`if (pa == pb) { // ("les deux pointeurs pointent vers la même donnée ") ; }`  
nous pouvons utilisé aussi : `<=`, `>=`, `!=`

102

## Chapitre 8: Les pointeurs



103

### □ Les tableaux et les pointeurs

Soit :     `int t [10] ;`  
           `int * p ;`  
           `int i ;`

- En C, le nom du tableau (ici `t`) est considéré comme un pointeur qui pointe sur le 1<sup>er</sup> élément du tableau.
- Le compilateur C transforme automatiquement une référence à un élément du tableau (`t[i]`) à une expression de type pointeur en observant les règles suivantes :

`t[0] ⇔ * t ;`

`t[i] ⇔ * ( t + i ) ;` Parcours des tableaux avec les pointeurs?

`t[i] ⇔ * ( & t [i] ) ;`

103

## Chapitre 8: Les pointeurs



104

### □ Les pointeurs et les chaînes de caractères

```
char * message = "bonjour ";
while (* message)
{
 printf ("%s", * message) ;
 message ++ ;
}
```

- On définit ici une variable pointeur "message" de type `char*` à la quelle on affecte, comme valeur initiale l'adresse de la chaîne "bonjour" ;
- Soit : `char ch[20] = "programmation" ;` écrire un programme C qui calcule la longueur de la chaîne de caractère "ch" de deux manières.

104

## Chapitre 8: Les pointeurs



105

### □ Allocation dynamique de tableaux

- Un tableau statique a une dimension bien définie à l'avance et possède en outre un nom, à l'aide duquel on peut accéder aux divers éléments du tableau.
- Un tableau dynamique, en revanche, a une dimension variable et ne possède pas de nom par lequel aurait accès aux éléments du tableau. Désormais, on utilise un pointeur qui pointe vers des données du type des valeurs saisies.
- Fonctions prédéfinies pour la gestion dynamique de la mémoire. Nous ferons appel à "stdlib.h" dans nos programmes.

105

## Chapitre 8: Les pointeurs



106

### □ Allocation de mémoire avec malloc

#### ■ Syntaxe :

`<pointeur> = malloc (< taille> ) ; // taille en octet`

- Retourne l'adresse du bloc de mémoire réservé rangée dans un pointeur. Et on peut via ce pointeur accéder au bloc de mémoire alloué.

#### ■ Exemple :

- `double * tab ;`
- `tab = malloc (400) ; // allocation d'un bloc mémoire de  
//400 octets et rangement de son adresse  
//dans un pointeur double.`

**tab est un tableau de 50 éléments de type double**

106

## Chapitre 8: Les pointeurs



107

### □ Allocation dynamique de tableaux

#### ▣ Gestion des erreurs :

- Si malloc n'arrive pas à réserver le bloc mémoire souhaité (pas assez d'espace mémoire) elle retourne le pointeur NULL.

```
if(tab==NULL)
 printf("Allocation demander impossible");
else
 { for(i=0;i<50;i++)
 tab[i]=(10*i); }
```

107

## Chapitre 8: Les pointeurs



108

### □ Allocation dynamique de tableaux

#### ▣ Si on manipule autre types de données :

#### ▣ **Exemple** : int occupent → 2 ou 4 octets selon la machine.

- Soit : 200 elem (2 octets) ou 100 elem (4 octets)

#### ▣ On résout le problème par l'opérateur "sizeof" pour 100 valeurs int :

On aura:

```
tab = malloc (100 * sizeof(int))
```

108

## Chapitre 8: Les pointeurs



109

### □ Allocation dynamique de tableaux

#### □ Conversion des pointeurs « void »

- La valeur que malloc renvoie au programme est un pointeur de type « void ». Un tel pointeur permet de mémoriser les adresses de données de tous types. Ce pendant, il est conseillé de convertir un pointeur de types indéterminé.

#### □ On aura :

```
tab = (double*) malloc (100 * sizeof (double))
```

#### □ Ou

```
tab = (int*) malloc (100 * sizeof(int)) ;
```

109

## Chapitre 8: Les pointeurs



110

### □ Allocation dynamique de tableaux

#### □ Libération de la mémoire avec « free »

- Elle permet de libère l'espace mémoire qui a été alloué par les fonctions malloc, et realloc.

#### □ Syntaxe :

```
free (<pointeur>) ;
```

#### □ Exemple :

```
free (tab) ;
```

110

## Chapitre 8: Les pointeurs



111

### □ Pointeurs et structures

- ▣ Les structures pouvant être gérées par des pointeurs.  
On définit les pointeurs vers des structures

Soit :

|                                                                  |                                                                                                                                                                                                                                                               |
|------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>struct article {     char name[20] ;     int num ; };</pre> | <pre>struct article *px; /*définit un pointeur px                     vers des données de type                     "struct article"*/ struct article x; px=&amp;x; // Ranger l'adresse de la variable       // structurée « x » dans le pointeur « px »</pre> |
|------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- ▣ x et px permettant d'accéder aux champs de la variable structurée « x »
  - x.name ; et x.num ; ⇔ (\*px).name; et (\*px).num ;

111

## Chapitre 8: Les pointeurs



112

### □ Pointeurs et structures

- ▣ Accès aux champs d'une structure pointée :
  - Syntaxe :  
(\* < pointeur>).<champ> ;
- ▣ Peut être remplacé par :
  - <pointeur>→<champ> ;
  - (\*px).name ⇔ px→name ;

112



## Chapitre 8: Les pointeurs



113

### □ Les pointeurs et les fonctions

#### □ Passage des paramètres par adresse (référence)

- Lorsqu'on veut qu'une fonction puisse modifier la valeur d'une donnée passée comme paramètre, il faut transmettre à la fonction non pas la valeur de l'objet concerné, mais son adresse.

#### □ Exemple : permutation des variables.

- La fonction appelée ne travaille plus sur une copie de l'objet transmis, mais sur l'objet lui-même (car la fonction reconnaît l'adresse). La fonction appelée range l'adresse transmise dans un paramètre formel approprié, donc dans un pointeur.

→ `permut (& a, &b) ;`

113

FIN



114