IAP - Introduction à l'Algorithmique et à la Programmation

T03 – Commentaires
T05 – Entrées/sorties
T11 – Expressions
T13 – Classification des opérateurs
T14 – Les tables de vérité
T15 – Expressions booléennes et arithmétiques
T16 – Règle d'évaluation des expressions
T18 – Opérateurs
T19 – Associativité des opérateurs
T20 – Priorité des opérateurs
T21 – Exemple d'évaluation d'expression
T23 – Structures de contrôle
T25 – Conditionnelle (if... else...)
T32 – Sélection à choix multiple (switch)

Equipe pédagogique
Marie-José Caraty, Julien Rossit, Camille Kurtz,
Jacques Alès-Bianchetti, Eloi Keita

Cours n° 1

Les fondamentaux de la programmation impérative

Les éléments de langage de programmation

1. DOCUMENTATION

Commentaires (1/2)

```
/* @file Cours1a-Exo01.c */
#include "stdafx.h"
int main() {
   float b=0., c=0., delta=0.;
   printf("Resolution de l'equation x^2+bx+c = 0\n");
   printf("Entrez les valeurs de b et c : ");
   scanf("%f %f", &b, &c);
   delta = (b*b)-(4*c);
   if (delta<0.) // 2 racines imaginaires
      printf("Pas de racine reelle");
   else
      if (delta==0.) // 1 racine double
         printf("Une racine reelle : %f", -b/2);
      else // 2 racines reelles
         printf("Deux racines reelles : %f et %f",
            (-b-sqrt(delta))/2., (-b+sqrt(delta))/2.);
   system("pause"); return 0;
}
```


Scanned by CamScanner

Sommaire

1. **Structure de programme**

2. **Données**
   - Expressions

3. **Traitements**
   - Entrées-sorties
   - Structures de contrôle
     Conditionnelle
     Choix multiples

Données
Structures de contrôle
(conditionnelles)

1. DOCUMENTATION

Commentaires (2/2)

```
/* Je suis un commentaire sur une ou plusieurs lignes,
   je précède ce que je commente (sans ligne blanche) et suis
   indenté sur la ligne que je commente (ici un main) : même
   retrait de mon premier "/" que la déclaration du main */
int main() {
   ...
}

/* COM1 Je peux commenter une portion de code en utilisant
   une étiquette (par exemple COM1), je commente ici
   le problème et l'entrée des donnees du problème */
printf("Resolution de l'equation x^2+bx+c = 0\n");
printf("Entrez les valeurs de b et c : ");
scanf("%f %f", &b, &c);
/* FIN COM1 */

// Je suis un commentaire a priori de fin de ligne
float t; // temps d'exécution en s
// mais on m'utilise aussi en début de ligne
```


Scanned by CamScanner

Les entrées-sorties

Traitements Entrées/sorties

```
/* @file Cours1a-Exo01.c */
#include "stdafx.h"
int main() {
   float b=0., c=0., delta=0.;
   printf("Resolution de l'equation x^2+bx+c = 0\n");
   printf("Entrez les valeurs de b et c : ");
   scanf("%f %f", &b, &c);
   delta = (b*b)-(4*c);
   if (delta<0.) // 2 racines imaginaires
      printf("Pas de racine reelle");
   else
      if (delta==0.) // 1 racine double
         printf("Une racine reelle : %f", -b/2);
      else // 2 racines reelles
         printf("Deux racines reelles : %f et %f",
            (-b-sqrt(delta))/2., (-b+sqrt(delta))/2.);
   system("pause"); return 0;
}
```

opérateur de référencement → adresse dans la mémoire

BNF de description de format (2/5)

`%[flag][prefixe][width][.precision][length]specifieur`

`flag`	cadrage à droite par défaut, pour un cadrage à gauche (-)
`préfixe`	+ 0 _ _ espace
`width`	largeur du champ
`precision`	nombre de chiffres après la virgule
`length`	court (`h`), long (`l`)
`specifieur`	caractère (`c`), chaîne (`s`)
	entier: décimal (`d` ou `i`), octal (`o`), hexadécimal (`x` ou `X`)
	court (`hd`), long (`ld`)
	unsigned (`u`)
	réel: court (`f`), notation scientifique (`e` ou `E`), général (`g`)
	long (`%lf`), notation scientifique (`le` ou `lE`), général (`lg`)

Exemple `int n=12; printf("*%-+6d*", n); // affiche :*+12___*`

Scanned by CamScanner

Pseudo-code écrire(v)

2. **TRAITEMENTS** – Entrées/Sorties

Sortie à partir du flot standard (écran) (1/5)

Bibliothèque `stdio.h`

Prototype `int printf( const char* format, ... );`

Exemple :

```
// Affichage des variables r, e et c (reel, entier, caractere)
// sous la forme de l'égalite des variables et de leur valeur
printf("r=%f e2=%d c=%c", r, e*e, c);
```

- `"r=%f e2=%d c=%c"` : chaîne de format de sortie
- `r, e*e, c` : Variables/ expression correspondant aux formats %
- à chaque format % doit correspondre la variable/expression à afficher

Remarque **Le retour entier** de la fonction `printf` peut être récupéré

```
int n;
n=printf("r=%f e=%d c=%c", r, e-2, c);
```

`n` indique alors le nombre de caractères affichés

Pseudo-code lire(v)

2. **TRAITEMENTS** – Entrées/Sorties

Entrée à partir du flot standard (clavier) (3/5)

Bibliothèque `"stdio.h"`

Prototype `int scanf( const char* format, ... );`

`format` est la chaine de conversion des entrées

Exemple : Forme simple de la saisie (au clavier) des 3 variables r, e et c de type resp. réel, entier et caractère

```
scanf("%f %d %c", &r, &e, &c);
```

- `"%f %d %c"` : Chaîne de format de conversion
- `&r, &e, &c` : adresses des variables « à lire »
- à chaque format % doit correspondre l'adresse de la variable à saisir au clavier

Remarque Vérification des entrées par le retour entier de la fonction `scanf`

```
int n;
n=scanf("r=%f e=%d c=%c", &r, &e, &c);
```

`n` indique le nombre de variables lues (en cohérence avec les formats)

Scanned by CamScanner

Exemple d'entrées-sorties (4/5)

```
/* Cours1-Exo01.c */
#include "stdafx.h"
int main() {
    int m, n, e;
    char c;
    double r;
    printf("Entrez un caractere : ");
    m=scanf("%c", &c);
    printf("%d variable(s) lue(s), ", m);
    printf("caractere lu : %c \n", c);
    printf("Entrez un entier et un reel : ");
    m=scanf("%d %lf", &e, &r);
    n = printf("%d variables lues : e=%d, r=%lf\n", m, e, r);
    printf("%d caracteres affiches\n", n);
    printf("Test de formats (\"|\" pour encadrer l'affichage)\n");
             // \" pour afficher le caractère "
    printf("Entier cadre a droite sur 5 caracteres |n=%5d|\n", e);
    printf("Entier cadre a gauche sur 5 caracteres avec prefixe + si >0
            |n=%-+5d|\n", e);
    printf("Reel sur 10 chiffres avec 3 de precision |r=%10.3lf|\n", r);
    system("pause"); return 0;
}
```

à remplacer par les inclusions d'entête de bibliothèque

```
#include <stdio.h>
#include <stdlib.h>
```

stdio.h, pour les Entrées-Sorties
stdlib.h, pour la fonction system(...)

`system("pause");`
Commande de pause de la fonction `system()` (spécifique à une exécution sous Visual Studio, figer la console d'exécution jusqu'à la l'entrée d'un caractère)

Les expressions

```
/* @file Cours1a-Exo01.c */
#include "stdafx.h"
int main() {
   float b=0., c=0., delta=0.;
   printf("Resolution de l'equation x^2+bx+c=0\n");
   printf("Entrez les valeurs de b et c : ");
   scanf("%f %f", &b, &c);
   delta = (b*b)-(4*c);
   if (delta<0.) // 2 racines imaginaires
      printf("Pas de racine reelle");
   else
      if (delta==0.) // 1 racine double
         printf("Une racine reelle : %f", -b/2);
      else // 2 racines reelles
         printf("Deux racines reelles : %f et %f",
           (-b-sqrt(delta))/2., (-b+sqrt(delta))/2.);
   system("pause"); return 0;
}
```

Traitements
Expressions

Scanned by CamScanner

Exemple d'entrée-sortie (5/5)

Trace d'exécution

```
Entrez un caractere : W
1 variable(s) lue(s), caractere lu : W
Entrez un entier et un reel : 12 2.713743
2 variables lues : e=12, r=2.713743
36 caracteres affiches          Rem : le caractère "\n" est comptabilisé
Test de formats ("|" pour encadrer l'affichage)
Entier cadre a droite sur 5 caracteres |n=   12|
Entier cadre a gauche sur 5 caracteres avec prefixe + si >0 |n=+12  |
Reel sur 10 chiffres avec 3 de precision |r=     2.714|
```

Remarque A la lecture de plusieurs variables enchaînées dans un `scanf` les données saisies au clavier doivent être séparées par un « white space »

« white space »
un espace, un retour chariot ou une tabulation

Opérateurs et expressions

Opérateur

Symbole indiquant une opération à effectuer entre 1, 2 ou plusieurs opérandes et retournant un résultat typé dépendant de l'opérateur et du type des opérandes

Un opérateur est caractérisé par son arité (nombre d'arguments)
Lorsque l'arité est supérieure à 1, l'opération est en notation infixée (l'opérateur est situé entre les 2 opérandes)

Exemples : moins unaire : `-10`
moins binaire : `10-7`

Expression

Combinaison de littéraux (constantes de type numérique/caractère), de variables, de fonctions et d'opérateurs

L'expression exprime un calcul, une manipulation de caractères ou un test de données

Exemples
```
2+3*x+1
(a<b && a>0)||(a>b && a==0)
```


Scanned by CamScanner

Classification des opérateurs

Opérateur binaire modulo : reste de la division entière
10%2 vaut 0, 10%3 vaut 1

Arithmétiques +, -, *, /, % (modulo)
associés à une/des opérandes de type entier/réel/caractère
Résultat entier/réel/caractère

Relationnels <, >, <= (≤), >= (≥), == (=) , != (≠)
associés à deux opérandes de même type,
l'expression est booléenne
Résultat booléen (faux ou vrai en logique)
codé respectivement en langage C (0 ou 1)

Logiques && (∧), || (∨)
∧ et ∨ logiques associés à des opérandes booléennes,
l'expression est booléenne
Résultat booléen (faux ou vrai en logique)
entier codé respectivement en langage C (0 ou 1)

Expressions booléennes et arithmétiques

```
/* @file Cours1-Exo02.c */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int a=7, b=3, x=5, Exp2;
    printf("Pour a=7 et b=3 :\n");
    printf("%d<%d est evalue a %d\n", a, b, a<b);
    printf("%d>%d est evalue a %d\n", a, b, a>b);

    printf("Exp1 2+3*x+1=%d\n", 2+3*x+1);
    Exp2= ((a<b && a>0)||(a>b && a==0));
    printf("Exp2 (a<b && a>0)||(a>b && a==0) vaut %d\n", Exp2);
}
```

```
Pour a=7, b=3, x=5 :
7<3 est evalue a 0
7>3 est evalue a 1
T12. 2+3*x+1=18
T12. (a<b && a>0)||(a>b && a==0) vaut 0
```

expression booléenne fausse
expression booléenne vraie
expression arithmétique évaluée à 18
expression booléenne fausse

Scanned by CamScanner

Les tables de vérité

Opérateur	Opération
!	NON logique
&&	ET logique
||	OU logique
^	OU EXCLUSIF

x	y	! x	x && y	x || y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Règle d'évaluation des expressions (1/2)

Evaluation d'une expression

`2+3*x` // est évaluée comme `(2+3)*x` ? ou comme `2+(3*x)` ?

Règle d'évaluation

(i) Fondée sur l'ordre de priorité/précédence des opérateurs
(nombre arbitraire fixant pour deux priorités distinctes l'ordre d'évaluation)
l'expression correspondant à l'opérateur de plus forte priorité
est évaluée en premier

(ii) Fondée sur l'associativité de l'opérateur : de gauche (G) à droite (D) ou de D à G
Associativité gauche : a ♥ b ♥ c = (a ♥ b) ♥ c
Associativité droite : a ♥ b ♥ c = a ♥ (b ♥ c)

(iii) Règle d'évaluation en cas d'égalité de priorité : de gauche à droite
a ♥ b ♣ c // si ♥ et ♣ sont deux opérateurs de même priorité
(a ♥ b) ♣ c est évalué // ordre d'évaluation de gauche à droite : 1) ♥ et 2) ♣

(iv) Pour un opérateur (binaire) donné, l'ordre d'évaluation est (généralement)
celui du premier opérande puis du deuxième opérande

Scanned by CamScanner

Règle d'évaluation des expressions (2/2)

Remarque la règle d'évaluation des expressions n'est pas standard dans les langages de programmation

i) les priorités, ii) les règles d'associativité et iii) la règle d'évaluation des opérateurs binaires doivent être connues et apprises pour chaque langage

Parenthésage explicite

Le parenthésage dans une expression force l'ordre d'évaluation : les expressions entre parenthèses sont évaluées en premier

Bonne Pratique

N'utiliser que les priorités d'opérateurs les « plus usuelles »
Définir explicitement la priorité des opérations par le parenthésage approprié, même si elles sont inutiles

Associativité des opérateurs

	Associativité
`()` `[]` `->` `.`	gauche à droite

Opérateurs unaires
`!` `~` `-` `+` `*`
`&` `++` `--` `(type)` `sizeof`
	droite à gauche

`*` `/` `\%`	gauche à droite
`+` `-`	gauche à droite
`<<` `>>`	gauche à droite
`<` `<=` `>` `>=`	gauche à droite
`==` `!=`	gauche à droite
`&`	gauche à droite
`^`	gauche à droite
`|`	gauche à droite
`&&`	gauche à droite
`||`	gauche à droite
`? :` (conditionnelle)	droite à gauche
Affectations simples et composées `=` `+=` `*=` `...`	droite à gauche
`,` (virgule)	gauche à droite

http://users.polytech.unice.fr/~jpr/ANSI_C99/Documents/ANSI_C99-poly.pdf

Exemples de la règle d'associativité

L'opérateur `<` est associatif de G à D

`a < b < c` **eq.** `(a < b) < c`

L'opérateur d'affectation `=` est associatif de D à G

`a = b = c` **eq.** **`a = (b = c)`**

La valeur de c est ainsi affectée à b puis à a

Rem 3. L'affectation composée (%)
Définition de l'assignation composée
`a ♥= b;` est une contraction de `a = a ♥ b;`

♥ : opérateur arithmétique générique {+, -, *, /, %}

Scanned by CamScanner

3. TRAITEMENTS – Expressions

Opérateurs

≡ est équivalent à

Opérateur	Utilisation	Signification
`()`	`f(x1, x2, ...)`	appel de fonction
`[]`	`t[i]`	indexation
`->`	`p->champ`	sélection de champ de structure
`.`	`s.champ`	sélection de champ de structure
`!`	`!a`	négation logique
`~`	`~a`	complément à 1
`-`	`-a`	moins unaire
`+`	`+a`	plus unaire
`*`	`*p`	indirection
`&`	`&x`	adresse de
`++`	`x++ ou ++x`	post ou pré-incrémentation
`--`	`x-- ou --x`	post ou pré-décrémentation
`(type)`	`(type)x`	conversion explicite (*cast*)
`sizeof`	**`sizeof(x)`**	taille mémoire d'un objet
`*`	`a*b`	multiplication
`/`	`a/b`	division
`%`	`a%b`	modulo (reste de la division)
`+`	`a+b`	addition
`-`	`a-b`	soustraction
`<<`	`a<<b`	décalage gauche
`>>`	`a>>b`	décalage droit
`<`	`a<b`	inférieur
`<=`	`a<=b`	inférieur ou égal
`>`	`a>b`	supérieur
`>=`	`a>=b`	supérieur ou égal
`==`	`a==b`	égal
`!=`	`a!=b`	différent de
`&`	`i&j`	« et » bit à bit
`^`	`i^j`	« ou exclusif » bit à bit
`|`	`i|j`	« ou » bit à bit
`&&`	`a&&b`	« et » logique séquentiel
`||`	`a||b`	« ou » logique séquentiel
`? :`	`a?b:c`	expression conditionelle
`= += *= ...`	`a=b`	affectations
`,`	`x1,x2,...,xn`	évaluation séquentielle

Rem 1. Pré/post-incrémentation (`++i` et `i++`)

Incrémentation préfixe : `++i`

`++i ≡ i+=1 ≡ i=i+1`

si `i` vaut 5, `++i` vaut 6 et `i` vaut 6

Incrémentation postfixe : `i++`

`i++ ≡ tmp=i, i+=1, tmp`

`,` est l'opérateur d'évaluation séquentielle

`i++` vaut `tmp` (dernière expression évaluée)

si `i` vaut 5, `i++` vaut 5 et i vaut 6

http://users.polytech.unice.fr/~jpr/ANSI_C99/Documents/ANSI_C99-poly.pdf

3. TRAITEMENTS – Expressions

Priorité/précédence des opérateurs

		Associativité
15	`()` `[]` `->` `.`	gauche à droite

14	*Opérateurs unaires*
	`!` `~` `-` `+` `*`
	`&` `++` `--` `(type)` `sizeof`
		droite à gauche

13	`*` `/` `\%`	gauche à droite
12	`+` `-`	gauche à droite
11	`<<` `>>`	gauche à droite
10	`<` `<=` `>` `>=`	gauche à droite
09	`==` `!=`	gauche à droite
08	`&`	gauche à droite
07	`^`	gauche à droite
06	`|`	gauche à droite
05	`&&`	gauche à droite
04	`||`	gauche à droite
03	`? :` (conditionnelle)	droite à gauche
02	*Affectations simples et composées* `=` `+=` `*=` `...`	droite à gauche
01	`,` (virgule)	gauche à droite

Par ordre croissant de priorité
Le sens de la flèche indique une croissance
(15 est plus prioritaire que 12)

Principe d'évaluation

Dans une expression (sans parenthésage), l'opérateur de plus forte priorité est évalué en premier
En cas d'égalité de priorité, l'évaluation de l'expression se fait de gauche à droite

http://users.polytech.unice.fr/~jpr/ANSI_C99/Documents/ANSI_C99-poly.pdf

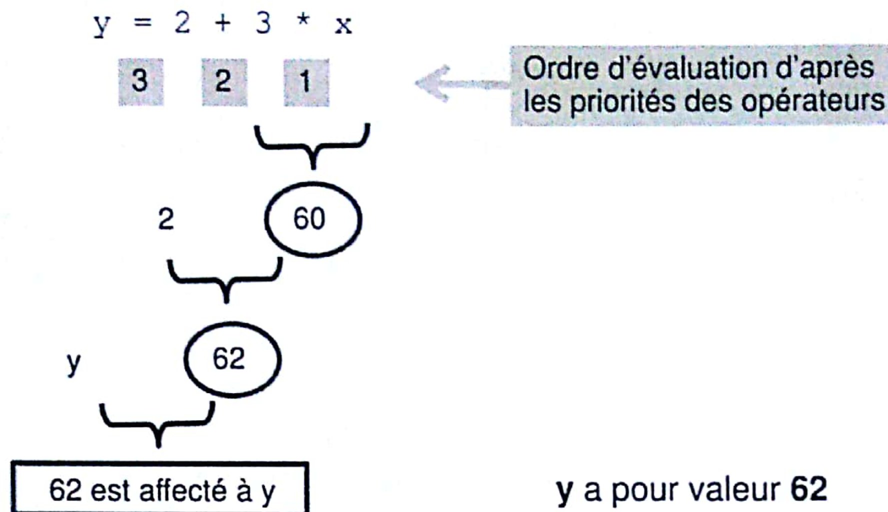
Scanned by CamScanner

Exemple d'évaluation des expressions

Opérateur	Priorité
*	13
+	12
=	2

Quel est l'ordre d'évaluation de l'expression ? `y = 2 + 3 * x`

Quelle est la valeur de y pour x=20



y a pour valeur **62**

Les structures de contrôle – Conditionnelle (if… else…)

```
/* @file Cours1a-Exo01.c */
#include "stdafx.h"
int main() {
   float b=0., c=0., delta=
   printf("Resolution de l'equation
   printf("Entrez les valeurs de b e
   scanf("%f %f", &b, &c);
   delta = (b*b)-(4*c);
   if (delta<0.) // 2 racines imaginaires
      printf("Pas de racine reelle");
   else
      if (delta==0.) // 1 racine double
         printf("Une racine reelle : %f", -b/2);
      else // 2 racines reelles
         printf("Deux racines reelles : %f et %f",
            (-b-sqrt(delta))/2., (-b+sqrt(delta))/2.);
   system("pause"); return 0;
}
```

Structures de contrôle

Branchements conditionnels

Scanned by CamScanner

3. **TRAITEMENTS** – Expressions

Vous savez tout, maintenant à vous !!! Préparation du TD1

Suivant les mêmes principes (cf. T. 21),
exercez-vous à évaluer les expressions suivantes :

1) (a==b && a<0) || (a<b && a>b) pour a=2 et b=10
2) (a==0) && (a<b && a>0) || (a>b && a==0) pour a=2 et b=16
3) 2 * 5 + 20 % 7 / 3 – 12
4) i+=a+=b pour i=4, a=1 et b=2

4. **STRUCTURES DE CONTRÔLE**

Rôle des structures de contrôle

Une structure de contrôle permet de modifier l'ordre séquentiel d'exécution des instructions d'un programme (flux d'exécution)

Faire exécuter des instructions
- en fonction de certaines conditions (les branchements)
- de façon répétitive (cf. les boucles, Cours 2)
- par appel de fonction (cf. les fonctions, Cours 3)

Parmi les structures de contrôle de type branchement (exprimée en pseudo-code)
- Branchement conditionnel avec alternative ou non
  si … alors … sinon … finSi
- Sélection à choix multiples
  suivant … cas … faire … finFaire

En pseudo-code: faire … finFaire
correspond à la notion de blocs d'instructions { … }

Scanned by CamScanner

<if-cond> ::= if (expr) B_instr [else B_instr]

Branchement conditionnel (1/7)

Permet d'exécuter des traitements selon certaines conditions (alternative de traitements)

if (<condition>) <Bloc_inst1> [else <Bloc_inst2>]

condition : expression
si condition est évaluée à vrai,
  le bloc d'instructions (Bloc_inst1) est exécuté
sinon (condition est évaluée à faux)
  le bloc d'instructions (Bloc_inst2) est exécuté

Pseudo-code
```
si (condition)
   alors
      Bloc_inst1
   sinon
      Bloc_inst2
fin-si
```

Remarque :

L'alternative `else` est optionnelle

Rappel :

Un bloc est une séquence d'instructions délimitée par { et }
Le bloc peut être vide ou restreint à une seule instruction (simple ou composée) et dans ce cas les délimiteurs peuvent être omis

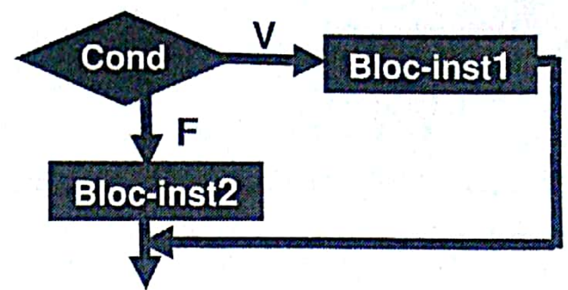
Branchement conditionnel – Exemple sans alternative (3/7)

```c
/* Cours1-Exo4.c */
#pragma warning(disable: 4996)
#include <stdio.h>
#include <stdlib.h>

// Calcul de la valeur absolue d'un entier
int main() {
   int x, valAbsolue;
   printf("Entier ? ");
   scanf("%d", &x);

   valAbsolue=x;
   if (x < 0)
      valAbsolue=-x;
   printf("Valeur absolue de %d : %d\n", x, valAbsolue);

   system("pause"); return 0;
}
```

```
Entier ? -7
Valeur absolue de -7 : 7

Entier ? 3
Valeur absolue de 3 : 3
```


Scanned by CamScanner

Branchement conditionnel



Contrôle d'exécution

```
   inst1;
   if (condition) {
      instV1;
      instV2;
   }
   else {
      instF1;
      instF2;
      instF3;
   }
α  inst3;
```

Flux d'exécution

inst1;
si condition est évaluée vraie
    instV1; instV2;
    // continuer en séquence en α
    inst3;

sinon // condition est évaluée fausse
    instF1; instF2; instF3;
    // continuer en séquence en α
    inst3;

Bonne Pratique

Respect de la règle d'indentation
Les instructions constitutives d'un bloc doivent être **indentées** (mises en retrait par tabulation) relativement à la structure de bloc ({ … })
Objectif : mettre en valeur le flux d'exécution
(séquencement des instructions à l'exécution)

Branchement conditionnel – Exemple avec alternative (4/7)

```
/* Cours1-Exo5.c */
#pragma warning(disable: 4996)
#include <stdio.h>
#include <stdlib.h>

// Calcul du minimum de deux entiers
int main() {
   int min, x, y;
   printf("Valeur des deux entiers ? ");
   scanf("%d %d", &x, &y);
   if (x<y)
      min=x;
   else
      min=y;
   printf("Le minimum de %d et %d est : %d\n", x, y, min);

   system("pause"); return 0;
}
```

```
Valeur des deux entiers ? 5 -3
Le minimum de 5 et -3 est : -3

Valeur des deux entiers ? 7 9
Le minimum de 7 et 9 est : 7
```


Scanned by CamScanner

Branchement conditionnel (5/7)

Imbrication de branchements conditionnels

```
inst1;
if (condition1) {
   inst11;
   …;
}
else if (condition2) {
         inst21;
         …;
     }
     else   if (condition3) {
               inst31;
               …;
            }
            else {
               instParDefaut;
            }
instSuivante;
```

Flux d'exécution ?
lorsqu'aucune condition n'est vérifiée

Remarque
Code illisible, sans une indentation correcte des bocs `if` et `else`

Branchement conditionnel (7/7)

Style de codage

```
if (condition)
{
  inst1;
  instr2;
}
else
{
  inst3;
}
inst4;
```

```
if (condition) {
  inst1;
  inst2;
}
else {
  inst3;
}
inst4;
```

```
if (condition) {
  inst1;
  inst2;
}
else
   inst3;
inst4;
```

Bonne Pratique

- Coder de façon homogène et respecter l'indentation (retrait des instructions)
- Une règle de codage pour la maintenance peut être de systématiquement délimiter un bloc par { et } (même dans le cas d'une instruction unique) : la structure de bloc est mise en place pour d'éventuels ajouts d'instructions

Scanned by CamScanner

Branchement conditionnel - Exemple avec imbrication (6/7)

```
/* Cours1-Exo6.c */
#pragma warning(disable: 4996)
#include <stdio.h>
#include <stdlib.h>

// Calcul du minimum de trois entiers
int main() {
    int min, x, y, z;
    printf("Valeur des trois entiers ? ");
    scanf("%d %d %d", &x, &y, &z);

    if (x<y && x<z)
        min=x;
    else
        if (y<z)
            min=y;
        else
            min=z;
    printf("Le minimum de %d, %d et %d est : %d\n",
            x, y, z, min);

    system("pause"); return 0;
}
```

```
Valeur des trois entiers ? 57 75 92
Le minimum de 57, 75 et 92 est : 57

Valeur des trois entiers ? 34 65 20
Le minimum de 34, 65 et 20 est : 20

Valeur des trois entiers ? 5 3 7
Le minimum de 5, 3 et 7 est : 3
```


Sélection à choix multiples (1/3)

Traitements à effectuer pour certaines valeurs (discrètes) d'une expression (de type entier/caractère)

```
switch (<expression>) {
        case <valeur> : [<instructions>]
        ...
        [default : [<instructions>]
}
```

expression : le discriminant de type entier ou caractère
valeur : un littéral du type de expression

Bonne Pratique

Lorsque le type de l'expression le permet,
préférer le `switch` à l'enchaînement de branchements conditionnels
dès que le niveau d'imbrication dépasse 2

Scanned by CamScanner

Sélection à choix multiples (2/3)

Contrôle d'exécution

```
inst1;
switch (expression) {
   case V1 : instV1a;
             instV1b;
             break;
   case V2 : instV2;
   case V3:  instV3;
   case V4:
   case V5:  instV5;
             break;
   default:  instDefault;
}
inst3;
```

Flux d'exécution

`inst1;`

si `expression` vaut `V1`
`instV1a; instV1b; inst3;`

si `expression` vaut `V2`
`instV2; instV3; instV5; inst3;`

si `expression` vaut à `V3`
`instV3;instV5; inst3;`

si `expression` vaut à `V4`
`instV5; inst3;`

si `expression` vaut `V5`
`instV5; inst3;`

si `expression` est différente de `V1`, `V2`, `V3`, `V4` et `V5`
`instDefault; inst3;`

Attention :

Ne pas oublier d'invoquer l'nstruction `break` pour interrompre l'exécution en séquence et provoquer la sortie du bloc `switch`

Bilan de ce que vous avez appris en cours

- Les commentaires
- Le principe des entrées-sorties standard
- Les expressions
  - Leur règle d'évaluation
  - La priorité des opérateurs
- Les structures de contrôle
  - Leur rôle
  - Instruction conditionnelle
  - Instruction à choix multiples

Au prochain cours…

La suite des structures de contrôle : les boucles

Scanned by CamScanner

Sélection à choix multiples - Exemple (3/3)

```
/* Cours1-Exo7.c */
#pragma warning(disable: 4996)
#include <stdio.h>
#include <stdlib.h>

// Oui ou non ?
int main() {
  char c;
  printf("Entrez un caractere, oui ou non (o/n) ? ");
  scanf("%c", &c);

  switch (c) {
    case 'o':
    case 'O': printf("Vous m'avez dit Oui!");
              break;
    case 'n':
    case 'N': printf("Vous m'avez dit Non!");
              break;
    default : printf("Ni oui, ni non. Peut-être ?");
  }

  system("pause"); return 0;
}
```

```
Entrez un caractere, oui ou non (o/n) ? O
Vous m'avez dit Oui!

Entrez un caractere, oui ou non (o/n) ? n
Vous m'avez dit Non!

Entrez un caractere, oui ou non (o/n) ? Y
Ni oui, ni non. Peut-etre ?
```


Scanned by CamScanner

Sommaire

1. **Structure de programme**

2. **Données**
   - Tableaux statiques
   - Chaînes de caractères
   - Type structuré/utilisateur

3. **Traitements**
   - Structures de contrôle
     Les instructions itératives

Données
Structures de contrôle (itératives)

3. TRAITEMENTS – Structures de contrôle

Boucle `for` – Boucle à itérations bornées (1/5)

for ([<for-init>] ; [<expr>] ; [<expr>]) <bloc-inst>

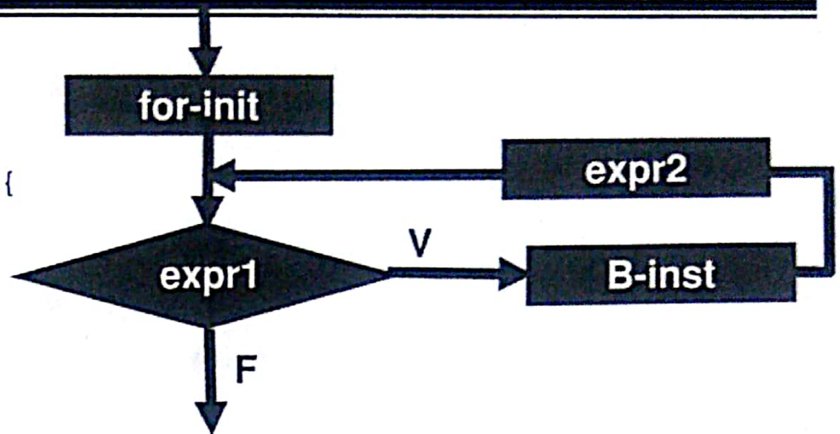
<for-init>	les instructions (d'initialisation) exécutées avant l'exécution de la boucle int i=0
<expr>	expression (condition), si l'expression est vraie, la boucle exécute son corps (bloc-inst) sinon elle s'arrête a=1
<expr>	les instructions exécutées à la fin de chaque itération (après exécution de bloc-inst) i++
<bloc-inst>	le boc des instructions exécutées à chaque itération choses à faire
Remarque	La BNF montre toute l'élaboration de la boucle `for`
Usuellement	la boucle `for` permet un traitement itératif contrôlé par un mécanisme de variable d'itération (compteur) et une condition d'arrêt

Scanned by CamScanner

Boucle for – Contrôle d'exécution (2/5)

Contrôle d'exécution

```
inst1;
for(for-init; expr1; expr2){
  B-inst;
}
inst2;
```



Flux d'exécution

inst1;
for-init
α si (expr1)
B-inst;
expr2
aller en α
finSi
inst3;

Remarques

Les 3 premiers éléments de la boucle sont optionnels

Sans la première expression (expr1), une constante ≠ 0 est considérée, la condition est toujours vraie

`for ( ; ; );` // est ainsi une boucle infinie

Boucle for – Pseudo-code (4/5)

Le pseudo-code de la boucle `for`
correspond aux itérations bornées et à l'utilisation d'un compteur

Pseudo-code (forme avec variable d'itération)

```
pour i allant de 1 à  n par pas de p faire
    bloc-inst
finFaire
```

Le pas est omis s'il vaut 1

```
pour i allant de 1 à  n faire
    bloc-inst
finFaire
```

Le pas peut être négatif

```
pour i allant de n à  1 par pas de -1 faire
    bloc-inst
finFaire
```

Remarque

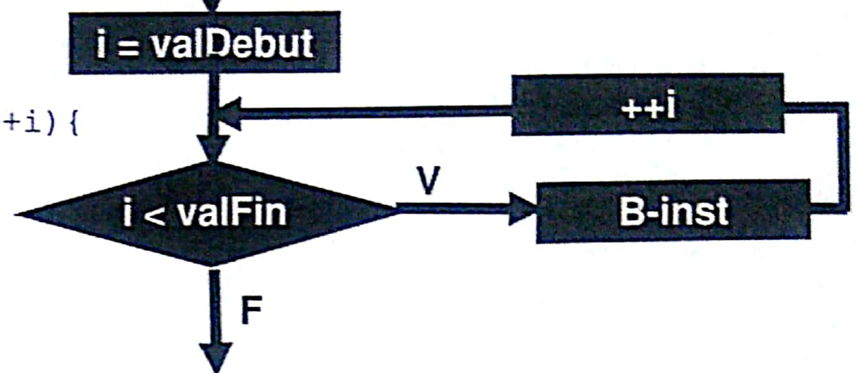
Boucle la plus adaptée lorsque le nombre d'itérations est connu

Scanned by CamScanner

Boucle `for` – Forme usuelle avec variable d'itération (3/5)

Contrôle d'exécution

```
inst1;
for(i=valDebut; i<valFin; ++i){
  B-inst;
}
inst3;
```



Flux d'exécution

inst1;
i ← valDebut
α si (i < valFin)
B-inst;
++i;
aller en α
finSi
inst3;

Remarques

(i) Ne jamais modifier la variable d'itération i (compteur) dans le corps de la boucle

(ii) Le nombre d'itérations est borné
dans notre cas de boucle où `i < valFin`,
si `ValFin` est strictement supérieur à `valDebut`,
le nombre d'itérations est `valFin-valDebut`
sinon aucune itération

Boucle `for` – Exemple (5/5)

Calculer la moyenne des 30 premiers entiers positifs

```
/* @file Cours2-Exo01.c */
#include "stdafx.h"

int main()
{
    int i;
    int nb;
    float cumul;
    for (i=1, nb=30, cumul=0.0; i<=nb; ++i) {
        cumul+=i; // eq. cumul=cumul+i;
    }
    printf("Moyenne des %d premiers entiers >0 %f\n", nb,
           cumul/nb);

    system("pause"); return 0;
}
```

Tests unitaires ?

`i=1, nb=30, i<=nb ,++i`
arrêt de boucle vérifié

Moyenne des 30 premiers entiers positifs 15.500000

Scanned by CamScanner

Boucles for imbriquées (1/2)

Une instruction de boucle peut être une boucle

Exemple :

Affichage attendu ?

```
pour i allant de 1 à 10 faire
   pour j allant de 0 à i-1 faire
      écrire('O')
   finFaire
   écrire('K')
finFaire
```

```
Dans la boucle (i), pour i=1
   La boucle interne (j) varie de 0 à 0,
   son bloc est exécuté 1 fois
     ("O" est affiché)
   Arrêt de la boucle (j)
   "K" est affiché
la boucle (i) continue de s'exécuter,
i=2
...
```


Boucle while – Itérations non bornées (1/3)

Permet un traitement itératif contrôlé par une condition d'itération a priori

`while (<expr>)` <bloc-inst>

Pseudo-code

```
tantQue (condition) faire
   bloc-inst
finFaire
```

Remarque :

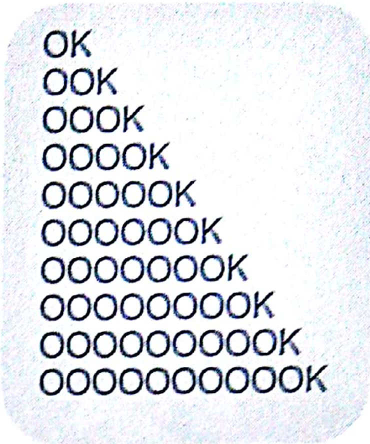
Boucle utilisée lorsque le nombre d'itérations n'est pas connu

L'exécution du bloc d'instruction est contrôlée par la condition
Si cette condition n'est pas satisfaite initialement, le bloc n'est pas exécuté

Scanned by CamScanner

Boucles for imbriquées – Exemple (2/2)

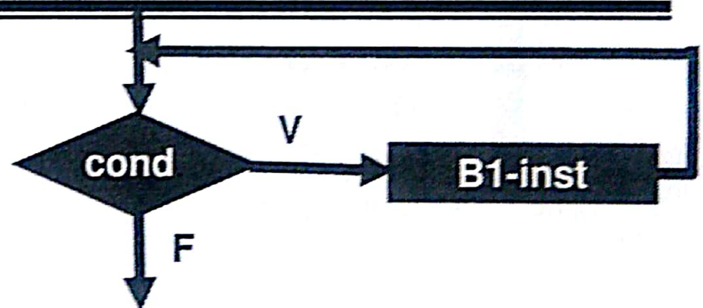
```
/* @file Cours2-Exo02.c */
#include "stdafx.h"

int main() {
   int i,j;
   for(i=1; i<=10; ++i) {
      for(j=0; j<i; ++j)
         printf("%c", 'O');
      printf("%c\n", 'K');
   }

   system("pause"); return 0;
}
```



Boucle while – Contrôle d'exécution (2/3)

Contrôle d'exécution

```
    inst1;
1    while (condition) {
α       instB1;
        instB2;
    }
    inst3;
```



Flux d'exécution

inst1;
1. si (condition est vraie) faire
   // continuer en séquence (en α)
   instB1; instB2; // fin de bloc
   aller en 1.
 fin faire
 sinon inst3;

Remarque :

Une instruction (au moins) de la boucle doit avoir un effet de bord sur (modifier) la condition et tendre vers la condition d'arrêt

Scanned by CamScanner

Boucle while – Exemple (3/3)

Et si i était initialisé à 0 avant la boucle ?

Déterminer le premier nombre entier n tel que la somme de 1 à n dépasse strictement 1000

L'entier n tel que 1+2+...+n > 1000 est : 45

```
/* @file Cours2-Exo03.c */
#include "stdafx.h"

int main() {
    int i, som;
    som=0;  1
    i=1;
    while (som <= 1000)  2
        som+=i; // eq. som=som+i;
        i=i+1;                        3
    }
    printf("n tel que 1+2+...+n > 1000 est : %d\n", i);

    system("pause"); return 0;
}
```

Principe du calcul incrémental
Une variable `som` *// somme demandée*
`som` *initialisée à* `0` *// élément neutre de l'addition*
À chaque itération : `i` *est additionnée à* `som`
`i` *est incrémenté de* `1`

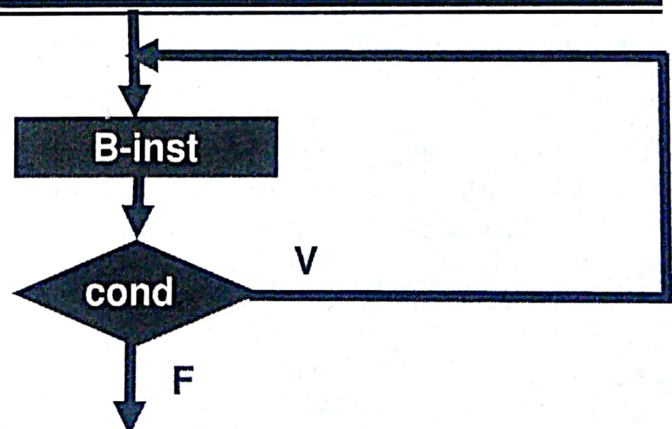
{1, 2 ,3} : arrêt de boucle vérifié

Test
On sait que 1+2+...+n= n.(n+1)/2
Vérifier :
n.(n+1)/2 > 1000 et n.(n-1)/2 <= 1000

Boucle do – Contrôle d'exécution (2/3)

Contrôle d'exécution

```
    inst1;
    do {
1       B-inst1;
        B-inst2;
    } while (condition);
    inst3;
```



Remarque :

Une instruction (au moins) de la boucle doit avoir un effet de bord sur la condition (la modifier) et tendre vers la condition d'arrêt

Flux d'exécution

inst1;
1 B-inst1; B-inst2; // fin de bloc
si (condition est vraie)
aller en 1
sinon inst3;

Scanned by CamScanner

Boucle do – Itérations non bornées (1/3)

Permet un traitement itératif contrôlé par une condition d'itération a postériori

```
do <bloc-inst> while (<expr>)
```

```
Pseudo-code
faire
    bloc-inst
tantQue (condition)
```

Remarque :

Boucle adaptée lorsque le nombre d'itérations n'est pas connu

Le bloc d'instruction est au moins exécuté une fois sans contrôle les autres itérations sont contrôlées par la condition

Boucle do – Exemple (3/3)

En combien d'années serez-vous millionnaire ? En partant d'un capital nul et en investissant chaque année 1000 euros à 57% d'intérêt.

```
/* @file Cours2-Exo04.c */
#include "stdafx.h"

int main() {
    double capital;
    short nbAnnees;
    capital=0.0;
    nbAnnees=0;
    do {
        capital=capital+1000.;
        capital=(1+0.57)*capital;
        nbAnnees++;
    } while (capital<1000000.);
    printf("Millionnaire dans %d ans!!!\n", nbAnnees);

    system("pause"); return 0;
}
```

```
Millionnaire dans 14 ans!!!
```

Reste à trouver l'investissement...

1, 2 ,3 : arrêt de boucle vérifié

Tests unitaires ?

Scanned by CamScanner

Flux d'exécution – (for et while) & (do et while)

D'après les flux d'exécution, la boucle for

```
for (<initialisation_s>; <condition>; <mise_s_à_jour>)
    <blocB>
```

peut s 'écrire sous la forme d'une boucle while

```
<initialisation_s>;
while (condition)
    <blocB>
    <mise_s_à_jour>
}
```

La boucle do

```
do
     <blocB>
while (<condition>)
```

peut s 'écrire sous la forme d'une boucle while

```
<blocB>
while (<condition>)
     <blocB>
```


Ruptures de séquence (1/2)

Instructions de rupture de séquence : `break et continue`

`break`

- Exclusivement dans une boucle ou une clause de sélection (case)
- Interrompt le flux d'exécution dans la boucle (la plus interne) ou l'alternative de sélection en provoquant un saut vers l'instruction suivant la structure de contrôle

`continue`

- Exclusivement dans une boucle
- Interrompt le flux d'exécution des instructions du bloc en provoquant un transfert d'exécution à l'itération suivante de la boucle
  Dans le cas d'une boucle for : mise à jour des variables d'itération et ré-évaluation de la condition d'arrêt

Bonne Pratique

Utiliser avec discernement `break` et `continue` (e.g., cas de recherche)

Scanned by CamScanner

Choix des boucles

La boucle `for` est à choisir pour les itérations bornées

Elle est conceptuellement élaborée, son écriture est compacte et ergonomique dans la localisation
(1) des initialisations d'entrée dans la boucle (au moins du compteur)
(2) de la condition d'itération
(3) de la mise à jour de fin d'itération (du compteur)
Le bloc `for` contient uniquement les instructions de traitement

La boucle `for` peut également être utilisées pour les itérations non bornées (cf. Transparent 34)

Les boucles `while` et `do` utilisées pour les itérations non bornées doivent être conçues par le programmeur avec la même rigueur
(1) des initialisations nécessaires avant l'entrée dans la boucle
(2) la mise à jour de fin d'itération dans le bloc instruction
(3) la mise à jour de fin d'itération doit faire converger la condition vers la condition d'arrêt de boucle
Le bloc instructions contient imbriquées : les instructions de traitement et de mise à jour de fin d'itération

Ruptures de séquence (2/2)

Exemples d'utilisation

```
while (true) {  //boucle infinie
  instB1;
  …
  if (condition)
     break;
  instB2;
  …
  }
inst3;
```

Code spaghetti ?
Utile dans des algorithmes de recherche

```
int borne=20;
for (int x=0; x<borne; ++x) {
  if (x%2)
     continue;
  // Traitements des entiers impairs
  instB1;
  …
}
```


Scanned by CamScanner

Retour sur les types de données

les types énumérés

et les pointeurs

avant de passer aux tableaux et à leur traitement

Type pointeur – Opérations élémentaires

Exemple pour une adresse mémoire sur 4 octets (processeur 32 bits) et un short sur 2 octets

Tout pointeur

(1) est une variable destinée à contenir une adresse mémoire

(2) est associé à un type d'objet

Déclaration du pointeur

L1 `short* pt_s;`

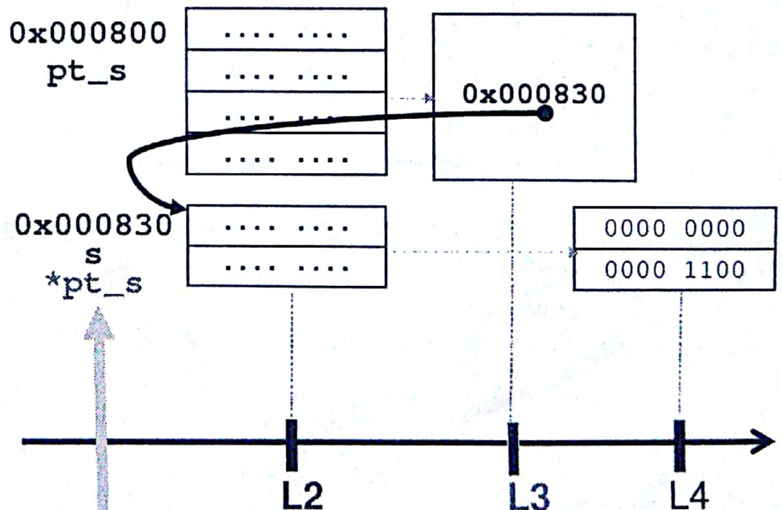
L2 `short s;`

(t1) Initialisation du pointeur

L3 `pt_s=&s;`

Déréférencement

L4 `*pt_s=12;`



*s et *pt_s sont deux noms qui accède au même emplacement mémoire*

Question
```
short m;
m=*pt_s;
Que vaut m ?
```


Scanned by CamScanner

Type énuméré – enum

Un type énuméré est un sous-type des entiers
Sa définition indique la liste des valeurs (constantes d'énumération) qu'une variable de ce type peut prendre

Déclaration d'un type énuméré

```
enum Color{BLUE, WHITE, RED};
```

Les constantes d'énumération ont des valeurs entières (à partir de 0) sauf si elles sont redéfinies

```
enum Color {BLUE, WHITE=3, RED, PINK=9};
// BLUE vaut 0, WHITE vaut 3, RED 4 et PINK 9
```

Déclaration et initialisation de variable

```
enum Color c; // le type est enum Color
c=BLUE;
```

Permet de déclarer une constante (d'énumération) et de lui affecter une valeur

```
enum {N=5};
```


Tableaux statiques – Déclaration (1/3)

Un tableau est une collection d'objets tous du même type

Déclaration

```
short tab[5]; // un tableau de 5 éléments entiers (codés sur 2 octets)
```

Bonne Pratique **Utiliser une constante pour définir la taille du tableau**

```
const N=5; // Constantes non autorisées en C90
```

On utilisera une variable d'énumération de valeur 5 (cf. T22)

```
enum {N=5};

short tab[N];
```

Représentation en mémoire (? : état de la mémoire)

tab	?	?	?	?	?
	0	1	2	3	4

Index du tableau : {0, .., 4}
accès au 4ème élément par `tab[3]`

Scanned by CamScanner

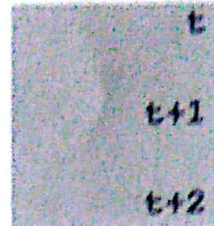
Schématiquement en mémoire

cf. TP
Affichage de tab (pointeur) et des 3 éléments du tableau (tab[i]) pour i allant de 0 à 2) et de leur adresse (tab, tab+1, tab+2)

A la déclaration d'un tableau de 3 entiers de type `short`

```
short t[3];
```

Zone contigüe en mémoire des 3 éléments du tableau (arithmétique des pointeurs)



	Adresses	Mémoire
		....
	0x00A126C	....
	0x00A126D	....
t	0x00A126E	....
	0x00A126F	....
t+1	0x00A1270	....
	0x00A1271	....
t+2	0x00A1272	....
	0x00A1273	....
	0x00A1274	....

t = &t = &t[0]
t est l'adresse du tableau implanté en mémoire **&t**

t est également l'adresse du 1er élément du tableau **&t[0]**
t+1 est l'adresse du 2ème élément du tableau **&t[1]**
t+2 est l'adresse du 3ème élément du tableau **&t[2]**
(arithmétique des pointeurs)

.... état (aléatoire) de la mémoire

Chaînes de caractères – Littéral (1/3)

Représentation interne d'une chaîne de caractères littérale

```
printf("Hello, world!");
```

H	e	l	l	o	,		W	o	r	l	d	!	\0
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Taille utile (13 octets)

Taille occupée en mémoire (14 octets)

Une chaîne est stockée dans un tableau de caractères

Attention

'a' est différent de "a"
'a' est un caractère stocké en mémoire sur un caractère
"a" est une chaîne de caractère stocké sur 2 caractères ('a' et '\0')

Scanned by CamScanner

3. RETOUR SUR LES TYPES – Données

Tableaux statiques – Exemple (3/3)

```
/* @file Cours2-Exo07.c */
#include "stdafx.h"

int main() {
    int i;
    enum {N=5}; // constante (d'enumeration) initialisee à 5
    int t1[N]={7, 2, 3, 6, 1}; // t1 initialisé par liste
    int t2[N];

    printf("Tableau t1 : ");
    for (i=0; i<N; ++i)
        printf("%d ", t1[i]);

    for (i=0; i<N; ++i) // t2 initialise par boucle
        t2[i]=i+1;

    printf("\nTableau t2 : ");
    for (i=0; i<N; ++i)
        printf("%d ", t2[i]);

    system("pause"); return 0;
}
```

```
Tableau t1 : 7 2 3 6 1
Tableau t2 : 1 2 3 4 5
```


3. RETOUR SUR LES TYPES – type natif tableau

Chaînes de caractères – `strlen()` et `sizeof()` (2/3)

A savoir

Une chaîne de caractères est stockée dans un tableau de caractères et est initialisée par un littéral chaîne

```
char s1[80] = "Hello!";
char s2[] = "Hello, you!";
```

`strlen(...)` fonction de la bibliothèque `<string.h>`
`strlen(s1)` longueur utile de la chaîne s1 (sans le délimiteur '\0')

`sizeof(...)` opérateur de C
donne la taille mémoire (en octets) d'une variable/type

`strlen(s1 )` vaut 6
`strlen(s2 )` vaut 11

`sizeof(s1)` vaut 80
`sizeof(s2)` vaut 12
`sizeof(int)` vaut 4

Scanned by CamScanner

Chaînes de caractères – Exemple (3/3)

```
/* @file Cours2-Exo07.c */
#include "stdafx.h"

int main()
{
    char s1[80]="Hello!"; // initialisation par un littéral chaîne
    char s2[] = "Hello, you!";

    printf("strlen(s1)=%d\n", strlen(s1));
    printf("sizeof(s1)=%d\n\n", sizeof(s1));

    printf("strlen(s2)=%d\n", strlen(s2));
    printf("sizeof(s2)=%d\n", sizeof(s2));

    system("pause"); return 0;
}
```

```
strlen(s1)=6
sizeof(s1)=80

strlen(s2)=11
sizeof(s2)=12
```

```
strlen() taille utile d'une chaîne
sizeof() taille mémoire occupée par une variable ou un type
```


Scanned by CamScanner

CONCLUSION

Bilan de ce que vous avez appris en cours

- Le rôle des itérations
- Les trois différents types de boucle en C, leur flux d'exécution et leurs ruptures de séquence
- L'importance de la boucle `for` pour les itérations bornées
- Ce qui doit guider le choix des boucles et les bonnes pratiques pour l'utilisation des boucles non bornées
- L'aide à la conception des boucles en vérifiant le critère d'arrêt
- Le type énuméré et le type pointeur pour l'introduction des tableaux
- La déclaration et la manipulation des tableaux statiques, et des chaînes de caractères ainsi que leur représentation mémoire

Au prochain cours…

La suite des traitements en programmation : les fonctions

Scanned by CamScanner

UNIVERSITÉ PARIS DESCARTES

IAP - Introduction à l'Algorithmique et à la Programmation

T04 – BNF définition de fonction
T06 – Contexte de déclaration
T07 – Contexte de définition
T08 – Contexte d'appel
T10 – Contexte d'exécution
T11 – Appel de fonction
T13 – Exemples d'exécution
T17 – Type structuré
T19 – typedef
T21 –Conception de fonction
T22 – Prototypage
T23 – Type retour structuré
T24 – Codage des types
T26 – Codage du corps
T28 – Documentation
T30 –Codage du main()
T31 – Structuration de code

Equipe pédagogique
Marie-José Caraty, Julien Rossit, Camille Kurtz,
Jacques Alès-Bianchetti, Eloi Keita

Cours n° 3

Les fondamentaux de la programmation impérative

Les fonctions (1/2)

1. FONCTIONS – Programme principal

Le main – La fonction point d'entrée d'une application

```
/* @file Cours1a-Exo01.c */
#include "stdafx.h"
int main() {
   float b=0., c=0., delta=0.;
   printf("Resolution de l'equation x^2          \n");
   printf("Entrez les valeurs de b et c    ");
   scanf("%f %f", &b, &c);
   delta = (b*b)-(4*c);
   if (delta<0.) // 2 racines imaginaires
      printf("Pas de racine reelle");
   else
      if (delta==0.) // 1 racine double
         printf("Une racine reelle : %f", -b/2);
      else // 2 racines reelles
         printf("Deux racines reelles : %f et %f",
            (-b-sqrt(delta))/2., (-b+sqrt(delta))/2.);
   system("pause"); return 0;
}
```


Scanned by CamScanner

Sommaire

Cf. Transp BNF et mécanisme d'exécution de la fonction

1. Structure de programme

2. Données

   Le type structuré

3. Traitements

   Les fonctions
   - Les différents contextes d'une fonction
   - Fonctions avec retour
   - Fonction sans retour
   - Phases de conception d'une fonction

Fonctions

1. FONCTIONS – Syntaxe

BNF de la définition d'une fonction

Terminologie

définition

```
<type_retour>|void  <id_fonction> ([<type> <id_arg>[,]])
     <bloc_instructions>
```

`<type_retour>`	type de la valeur renvoyée par la fonction dans ce cas, le bloc instruction contient au moins une instruction : `return <expression>;`
`void`	fonction sans retour de valeur
`<id_fonction >`	identificateur de la fonction
`<type>`	type de l'argument
`<id_arg>`	identificateur de l'argument
`<bloc_instructions>`	corps de la fonction

déclaration/ prototype

```
<type_retour>  <id_fonction> ([<type> <id_arg>[,]]);
```

corps

```
<bloc_instructions>
```


Scanned by CamScanner

Analyse de la BNF de définition de fonction

D'après la BNF

Une fonction peut avoir un retour ou non

- une valeur retournée par la fonction (`<type_retour>`)
- pas de valeur retournée (`void`)
  - appelée dans d'autres langages une procédure

Une fonction peut avoir

- des paramètres (appelés **paramètres formels**)

```
double leMin(double x, double y);
```

- aucun paramètre

```
void InfoServices();
```

Quatre contextes associés à la fonction

- Déclaration – Définition – Appel – Exécution

Contexte de définition de fonction (2/3)

La définition d'une fonction

- reprend son type et surtout définit son corps
- le bloc d'instructions qui sera exécuté à chaque appel
  - toute instruction est autorisée dont l'appel d'autres fonctions
  - et l'appel de la fonction elle-même (appel récursif*)
- les **paramètres formels** sont utilisés dans le corps de la fonction comme toute variable déclarée en début de bloc

Définition de la fonction retournant le minimum de deux réels

```
double leMin( double x, double y){
   if (x<y) return x;
   else return y;
}
```

Notion différée : Récursivité

Scanned by CamScanner

Contexte de déclaration de fonction (1/3)

Chaque fonction a un type déterminé par son prototype
(nombre, type de ses paramètres formels et de son retour)

Retour

Une seule valeur de retour, avec pour types autorisés
les types scalaires (types affectables)
entiers, réels, pointeurs, structures ou énumérés

Remarque Le type de retour ne peut pas être un tableau ni une chaîne de caractères

Paramètres formels

avec pour types autorisés : les types scalaires

Déclaration/Prototype de la fonction retournant le minimum de deux réels

```
double leMin(double x, double y);
```

Remarque Comme tout objet, la fonction doit être déclarée avant son utilisation.
Le prototype doit être situé dans un bloc dont la portée contient l'appel de la fonction

Contextes d'appel/invocation de fonction (3/3)

Un appel /invocation de fonction est syntaxiquement équivalent à une valeur de son type de retour.

Prototype de la fonction sqrt() `double sqrt(double x);`

```
double racine=(-b-sqrt(delta))/2.;
```

L'appel `srqt(delta)` est une valeur de type `double` (la racine carrée de delta)

Appel de la fonction `leMin()`

```
int main() {
   double a=20., b=10.;
   double res=leMin(7, leMin(a, b));
   printf( "Minimum de 7, 20 et 10 : %lf\n" , res));
   printf( "Minimum de 7 et Min(20, 10) : %lf\n",
           leMin(7, leMin(a, b)));

   system("pause"); return 0;
}
```


Scanned by CamScanner

Quel est le mécanisme d'exécution d'une fonction ?

- Le contexte d'exécution d'une fonction
  - Contexte mémoire
  - Mécanisme d'exécution
    - à l'appel
      - Mode de passage des paramètres effectifs aux paramètres formel
    - au retour

Appel de fonction avec des paramètres effectifs (2/2)

La fonction est appelée avec les paramètres effectifs qui correspondent en nombre et en type aux paramètres formels

Le contexte d'exécution est mis en place

Les paramètres effectifs* sont passés aux paramètres formels

- Le mode de passage est unique en Langage C
  mode par copie de valeur du paramètre effectif au paramètre formel
  (la valeur du paramètre effectif initialise le paramètre formel)

*un paramètre effectif est une expression
(variable, expression arithmétique, appel de fonction, ...) (cf. BNF)

(1) le paramètre effectif est tout d'abord évalué
(2) éventuellement converti dans le type du paramètre formel
(3) sa valeur est affectée au paramètre formel lui correspondant

Au retour de la fonction, la valeur de retour est transmise à l'appelant
L'espace mémoire alloué à l'appel est désalloué (par le système)

→ Les variables du contexte d'exécution ne sont plus accessibles

Scanned by CamScanner

Contexte d'exécution d'une fonction (1/2)

Toute fonction a son propre contexte d'exécution :
une zone mémoire allouée par le système (**la pile d'exécution**)
que le flux d'exécution de la fonction va modifier
(par les entrées, les affectations et autres appels de fonction)

C'est à l'appel de la fonction que cette zone mémoire est réservée pour implanter

(1) les paramètres formels (entre les "(" ")" suivant l'identificateur de la fonction)

(2) les variables locales déclarées en début de bloc (instructions de la fonction)

Contexte d'exécution du main () fonction (ici) sans paramètre formel

```
L10 int main(){
L11    float b=0., c=0., delta=0.;
       ......
       system("pause"); return 0;
    }
```

main	L10	L11	L12	...
b	?			
c	?			
delta	?			

t

2. FONCTIONS – Mécanisme d'exécution de fonction

Exemples d'exécution de programme avec appel de fonction

fonction sans retour

fonction avec retour

Scanned by CamScanner

? état de la mémoire (1/2)

Exécution de programme – Fonction sans paramètre et sans retour

```
L20 void f(){
L21   double pi=3.14;
L22   printf("pi=%lf\n", pi);
L23   return;
    }

L10 int main() {
L11   double pi=3.1;
L12   printf("pi=%lf\n", pi);
L13   f();
L14   printf("pi=%lf\n", pi);
L15   system("pause"); return 0;
    }
```

```
pi=3.100000
pi=3.140000
pi=3.100000
```

(1) Exécution de programme, appel du `main()`

pi=3.1

main	L10	L11	L12
pi	?	3.1	3.1

(2) L13 : appel de `f()`, interruption du `main()`

main	L10	L11	L12
pi	?	3.1	3.1

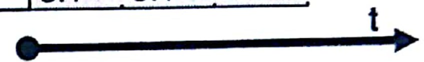
pi=3.14

f	L20	L21	L22	L23
pi	?	3.14	3.14	?

(3) L13 : retour d'exécution du `main()`

pi=3.1

main	L13	L14	L15
pi	3.1	3.1	?



Savoir faire

Simuler l'état mémoire (à base de schémas) à l'exécution d'un programme

QCM – Effet de bord ou non ?

(1) Une variable `h` déclarée dans un `main()` est-elle la même que la variable `h` déclarée dans une fonction `f()` ?

Oui/Non

(2) Si une variable `h` déclarée dans une fonction `f()` est modifiée dans le corps de `f()`, la variable `h` déclarée dans le `main()` est-elle modifiée ?

Oui/Non

(3) Si dans le corps d'une fonction, on modifie un paramètre formel `p`, le paramètre effectif `e` de l'appelant est-il modifié ?

Oui/Non

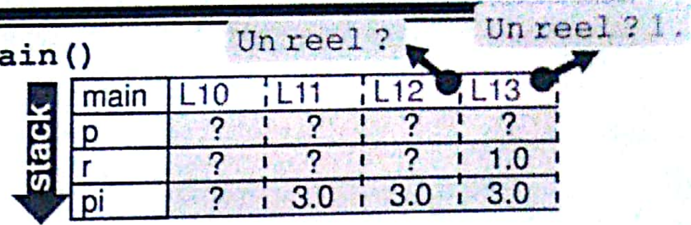
Effet de Bord **Modification d'une variable de l'appelant par la fonction**

Scanned by CamScanner

Exécution de programme – Fonction avec paramètre et avec retour (2/2)

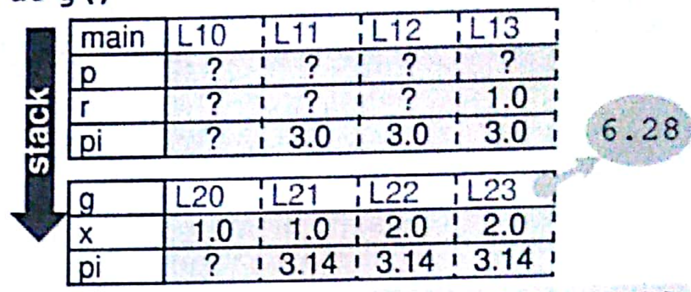
```
L20  double g(double x){
L21          double pi=3.14;
L22          x=2.*x;
L23          return pi*x;
     }
L10  int main() {
L11   double p, r, pi=3.;
L12   printf("Un reel ? ");
L13   scanf("%lf", &r);
L14   p=g(r);
L15   printf("p=g(%lf)=%lf r=%lf\n",
       r, p, r);
L16   system("pause"); return 0;
}
```

(1) Appel du main()



main	L10	L11	L12	L13
p	?	?	?	?
r	?	?	?	1.0
pi	?	3.0	3.0	3.0

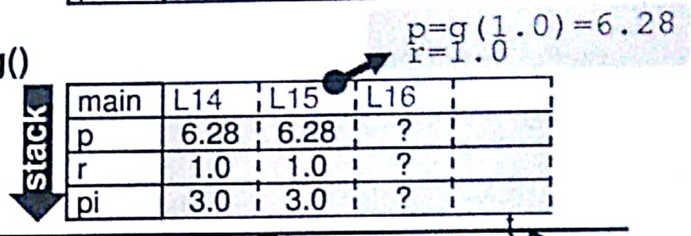
(2) L14 : appel de g()



main	L10	L11	L12	L13
p	?	?	?	?
r	?	?	?	1.0
pi	?	3.0	3.0	3.0

g	L20	L21	L22	L23
x	1.0	1.0	2.0	2.0
pi	?	3.14	3.14	3.14

(3) Retour de g()



main	L14	L15	L16	
p	6.28	6.28	?	
r	1.0	1.0	?	
pi	3.0	3.0	?	

```
Un reel ? 1.
p=g(1.000000)=6.280000 r=1.000000
Appuyez sur une touche pour continuer...
```


Coder une fonction résolvant toute équation du second degré du type $x^2+bx+c=0$

Problème — le retour de la fonction n'est pas unique plusieurs cas liés aux valeurs de b et c donnent des résultats différents

Retour sur les types

Le type structuré une solution à notre problème

Scanned by CamScanner

Type structuré/utilisateur – enregistrement (1/3)

Un enregistrement correspond à une structuration logique de données de type éventuellement hétérogène
Ces données sont aussi appelées des champs

Remarque :
On l'appelle également type utilisateur
un type (non standard) défini dans le source

Bonne Pratique programmation structurée

Structurer les instructions (fonctions)
Structurer les données (logiquement liées)

Alias de type pour faciliter l'écriture – typedef

Pour les types structurés (struct) ou énumérés (enum)

```
// Declaration dans un bloc B d'un type (structure) Point
typedef struct {
  int abs; // abscisse
  int ord; // ordonnee
} Point;
```

Accès aux champs de p par la notation pointée

```
// Declaration du point p dans B ou l'un de ses sous-blocs
Point p;
// Acces dans B ou l'un de ses sous-blocs
p.abs=4;
p.ord=7;
```

```
typedef unsigned char uchar;
uchar nbJours=365;

typedef int t[100] Vector // type Vector (tableau de 100 entiers)
Vector v;

typedef char t[21] Char20; // type chaine de 20 caractères (utiles)
Char20 s1, s2;
```


Scanned by CamScanner

Type structuré/utilisateur – struct (2/3)

Définition du type structuré Point dans un plan

```
struct Point {
   int abs; // abscisse
   int ord; // ordonnée
};
```

Déclaration d'une variable de type Point

```
struct Point p; // en C90
Point p; // normes postérieures
```

Accès aux champs de p par la notation pointée

```
   p.abs   // abscisse de p
   p.ord   // ordonnée de p
```

Déclaration d'une courbe (tableau de points)

```
Point courbe[10];
```

Accès aux coordonnées du 10ème point de la courbe

```
courbe[9].abs et courbe[9].ord
```


Revenons à notre problème

Coder une fonction résolvant toute équation du second degré du type $x^2 + bx + c = 0$

Objectif : prototyper cette fonction (1ère étape)

analyse de l'interface de la fonction

Scanned by CamScanner

Prototypage – Les bonnes questions à se poser

Analyse de l'interface de la fonction

Le type de la solution (2/2)

Analyse de l'algorithmique

Trois résultats possibles (suivant `delta`) que l'on code
zero : pas de racine réelle
une : une solution réelle (`r1`)
deux : deux solutions réelles (`r1` et `r2`)

Deux choix pour le stockage de la valeur de la/des solution(s)

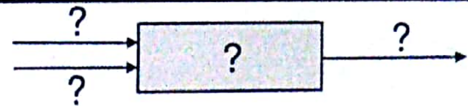
(1) deux variables (`x1` et `x2`)
 `r1` stocke la 1ère racine dans les cas 1 et 2
 `r2` stocke la 2ème racine dans le cas 2

(2) Un tableau de 2 réels
 `solution[0]` stocke `r1` , `solution[1]` stocke `r2`

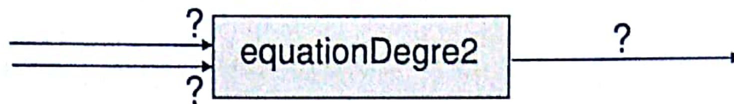
Scanned by CamScanner

Analyse de l'interface d'une fonction (1/2)

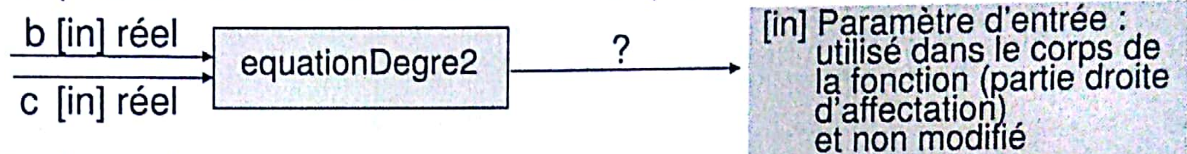
Schéma de l'interface



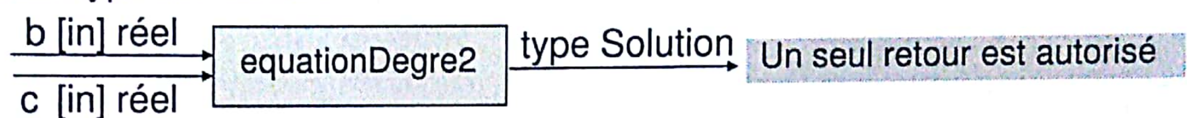
(1) Quel nom donner à la fonction ?



(2) De quelles données avons-nous besoin pour résoudre de problème ?



(3) Quel type de retour ?



Prototypage immédiat après le choix du type réel

```
Solution equationDegre2(double b, double c);
```


Codage des types – énuméré (Racine_s) et structuré (Solution) (1/4)

```
/* @file Cours3-Ex02.c*/

typedef enum {
   ZERO, // aucune racine réelle
   UNE, // une racine réelle
   DEUX // deux racines réelles
} Racine_s;

typedef struct {
   Racine_s nbRacines; // Nombre des racines de la solution
   double racines[2]; // Conteneur des racines
} Solution;

Solution equationDegre2(double b, double c);
```


Scanned by CamScanner

5. FONCTIONS – Codage de la fonction

Corps de la fonction – Algorithme de résolution

```c
/* @file Cours1a-Exo01.c */
#include "stdafx.h"
int main() {
   float b=0., c=0., delta=0.;
   printf("Resolution de l'equation x^2+bx+c = 0\n");
   printf("Entrez les valeurs de b et c : ");
   scanf("%f %f", &b, &c);
   delta = (b*b)-(4*c);
   if (delta<0.) // 2 racines imaginaires
      printf("Pas de racine reelle");
   else
      if (delta==0.) // 1 racine double
         printf("Une racine reelle : %f", -b/2);
      else // 2 racines reelles
         printf("Deux racines reelles : %f et %f",
            -b-sqrt(delta))/2., (-b+sqrt(delta))/2.);
   system("pause"); return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```


5. FONCTIONS – Codage de la fonction

Documentation de fonction

Faut-il documenter une fonction ?

Oui/Non

Si oui, pourquoi ?

sinon, pourquoi ?

Scanned by CamScanner

Codage de la fonction equationDegre2() (2/4)

```
Solution equationDegre2(double b, double c) {
   Solution res;
   double delta = (b*b)-(4*c);
   if (delta < 0.) {
      res.nbRacines=ZERO;
   }
   else if (delta==0.) {
         res.nbRacines=UNE;
         res.racines[0]= -b/2;
      }
      else {
         res.nbRacines=DEUX;
         res.racines[0]= (-b-sqrt(delta))/2.;
         res.racines[1]= (-b+sqrt(delta))/2.;
      }
   return res;
}
```


Documentation de fonction

La documentation des sources est obligatoire
- blocs et fonctions
  - pour la ré-utilisation (fonctions)
  - pour la maintenance (fonctions et partie de code « difficile »)

La documentation doit être ciblée et concise

Pour une fonction savoir décrire pour le programmeur
- le rôle de la fonction,
- la sémantique
  - des paramètres formels,
  - du retour du résultat
- le domaine de définition* de la fonction

Le langage naturel peut convenir si l'objectif de documentation est bien tenu

Normalisation de la documentation

Incontournable dans les gros projets industriels/recherche

Exemple : Javadoc* un formalisme à base d'étiquettes sémantiques

Notion différée* – **Le formalisme de la Javadoc**
~~**Précondition(s) d'une fonction et vérification d'appel**~~

Scanned by CamScanner

Structuration du code – Déclarations (types énuméré/strucuré)

```
/* @file Cours3-Ex02.c */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef enum {
   ZERO, // aucune racine réelle
   UNE, // une racine réelle
   DEUX // deux racines réelles
} Racine_s;

typedef struct {
   Racine_s nbRacines; // Nombre des raciness de la solution
   double racines[2]; // Conteneur des racines
} Solution;

Solution equationDegre2(double b, double c);
```

Avant le `main()` les déclarations usuelles
- Liste des définitions des types utilisés (énumérés et structurés)
- Liste des prototypes des fonctions utilisées (par le `main()` ou autres fonctions)

Scanned by CamScanner

Codage de la fonction `equationDegre2()` (3/4)

```
/* Retourne la solution de l'equation x^2+ax+b=0
 * format de la solution : le nombre de racines(nbRacines)
 * de valeur (ZERO, UNE, DEUX)
 * et le tableau (racines) stockant la racine (1er élément)
 * ou les racines (1er et 2ème élément) du polynome
 */
Solution equationDegre2(double b, double c) {
   Solution res;
   double delta = (b*b)-(4*c);
   if (delta < 0.) {
      res.nbRacines=ZERO;
   }
   else if (delta==0.) {
             res.nbRacines=UNE;
             res.racines[0]= -b/2;
         }
         else {
             res.nbRacines=DEUX;
             res.racines[0]= (-b-sqrt(delta))/2.;
             res.racines[1]= (-b+sqrt(delta))/2.;
         }
   return res;
}
```


Scanned by CamScanner

Codage de la fonction `main()` (4/4)

```
int main() {
   double b=0., c=0., delta=0.;
   Solution res;
   printf("Resolution de l'equation x^2+bx+c = 0\n");
   printf("Entrez les valeurs de b et c : " );
   scanf("%lf %lf", &b, &c);
   res=equationDegre2(b, c);
   switch (res.nbRacines) {
      case ZERO :    printf("Pas de racine reelle\n");
                     break;
      case UNE :     printf("Une racine double reelle %lf\n",
                        res.racines[0]);
                     break;
      case DEUX :    printf("Deux racines reelles %lf et %lf\n",
                        res.racines[0], res.racines[1]);
                     break;
      default :      printf("Erreur...");
                     break;
   }
   system("pause"); return 0;
}
```


Scanned by CamScanner

Structuration du code – le main() (2/3)

```
int main() {
   double b=0., c=0., delta=0.;
   Solution res;
   printf("Resolution de l'equation x^2+bx+c = 0\n");
   printf("Entrez les valeurs de b et c : " );
   scanf("%lf %lf", &b, &c);
   res=equationDegre2(b, c);
   switch (res.nbRacines) {
      case ZERO : printf("Pas de racine reelle\n");
                  break;
      case UNE :  printf("Une racine double reelle %lf\n",
                        res.racines[0]);
                  break;
      case DEUX : printf("Deux racines reelles %lf et %lf\n",
                        res.racines[0], res.racines[1]);
                  break;
      default :   printf("Erreur...");
                break;
   }
   system("pause"); return 0;
}
```

Autre exemple de structuration de code

Utilisation de plusieurs fonctions

Scanned by CamScanner

Structuration du code – la fonction `equationDegre2` (3/3)

```
/* Retourne la solution de l'équation x^2+ax+b=0
 * format de la solution : le nombre de racines(nbRacines)
 * de valeur (ZERO, UNE, DEUX)
 * et le tableau (racines) stockant la racine (1er élément)
 * ou les racines (1er et 2ème élément) du polynome
 */
Solution equationDegre2(double b, double c) {
   Solution res;
   double delta = (b*b)-(4*c);
   if (delta < 0.) {
      res.nbRacines=ZERO;
   }
   else if (delta==0.) {
         res.nbRacines=UNE;
         res.racines[0]= -b/2;
      }
      else {
         res.nbRacines=DEUX;
         res.racines[0]= (-b-sqrt(delta))/2.;
         res.racines[1]= (-b+sqrt(delta))/2.;
      }
   return res;
}
```

Après le main()
Liste de toutes les définitions de fonctions documentées utilisées par le main() ou autres fonctions

Choix 1) (1/2)

Structuration du code – Utilisation de plusieurs fonctions

```
Minimum de 7 et min(20.000000, 10.000000) : 7.000000
Maximum de 7 et max(20.000000, 10.000000) : 20.000000
```

```
/* @file Cours3-Ex03.c */
#include "stdafx.h"

double Min(double x, double y);
double Max(double x, double y);

int main() {
   double a=20., b=10.;
   double res=Max(a, b);
   printf("Minimum de 7 et min(%lf, %lf) : %lf\n", a, b, Min(7, Min(a,b)));
   printf("Maximum de 7 et max(%lf, %lf) : %lf\n", a, b, Max(7, res));
   system("pause"); return 0;
}

/**
 * Retour du minimum de deux réels x et y
 */
double Min(double x, double y){
   if (x<y) return x;
   else return y;
}

/**
 * Retour du maximum de deux réels x et y
 */
double Max(double x, double y){
   if (x>y) return x;
   else return y
}
```

Avant le main()
Liste de toutes les déclarations de fonctions utilisées par le `main()` ou autres fonctions

Après le main()
Liste de toutes les définitions de fonctions documentées utilisées par le main() ou autres fonctions

Scanned by CamScanner

Structuration du code – Utilisation de plusieurs fonctions

```
Minimum de 7 et min(20.000000, 10.000000) : 7.000000
Maximum de 7 et max(20.000000, 10.000000) : 20.000000
```

Avant le main()
Liste de toutes les définitions de fonctions documentées utilisées par le `main()` ou autres fonctions

```c
/* @file Cours3-Ex03.c */
#include "stdafx.h"

/**
 * Retour du minimum de deux réels x et y
 */
double Min(double x, double y){
   if (x<y) return x;
   else return y;
}

/**
 * Retour du maximum de deux réels x et y
 */
double Max(double x, double y){
   if (x>y) return x;
   else return y
}

int main() {
   double a=20., b=10.;
   double res=Max(a, b);
   printf("Minimum de 7 et min(%lf, %lf) : %lf\n", a, b, Min(7, Min(a,b)));
   printf("Maximum de 7 et max(%lf, %lf) : %lf\n", a, b, Max(7, res));
   system("pause"); return 0;
}
```


Scanned by CamScanner

CONCLUSION

Bilan de ce que vous avez appris en cours

- L'usage de la BNF pour définir une fonction
- Les trois contextes d'une fonction (déclaration, définition et appel)
- Le mécanisme d'exécution d'une fonction (contexte d'exécution, appel et passage par copie de la valeur des paramètres effectifs aux paramètres formels, exécution de programme)
- Retour sur les types (type structuré `struct`) et l'utilisation de `typedef`
- La conception de fonction avec une technique de prototypage
- Un exemple de conception de fonction (avec un type de retour structuré)
- La documentation des fonctions
- La structuration de code de programmes incluant une ou plusieurs fonctions

Au prochain cours…

De l'algorithmique de recherche d'information

Scanned by CamScanner

UNIVERSITE PARIS DESCARTES

IAP - Introduction à l'Algorithmique et à la Programmation

Equipe pédagogique
Marie-José Caraty, Julien Rossit, Camille Kurtz,
Jacques Alès-Bianchetti, Eloi Keita

T02 – Généralités
T05 – Expression des algorithmes
T06 – Notions fondamentales de la complexité
T09 – Algorithme de recherche linéaire
T15 – Algorithme de recherche dichotomique
T21 – Implémentation du calcul du nombre d'opérations fondamentales
T25 – Temps d'exécution
T27 – Premier bilan sur la complexité
T28 – Comparaison de temps d'exécution
T29 – Codage temps d'exécution Posix

Cours n° 4

Fondamentaux en algorithmique

Algorithmes de recherche

1. GENERALITES

Quelques dates historiques

Algorithme d'Euclide (-325, -265) – Calcul du PGCD

```
a, b, r : trois entiers
lire(a, b)
r ← a / b
tantQue (r <> 0) faire
   a ← b,
   b ← r,
   r ← a / b
finFaire
écrire(«PGCD = » , b)
```

Algorithme d'Archimède (-287, -212) – Approximation du nombre π

Algorithme d'Eratosthène (-276, -194) – Liste des nombres premiers

Algorithme de Zhang Cang (-206) – Calcul de l'aire d'un cercle

Scanned by CamScanner

1. GENERALITES
Algorithmique

Algorithme (Al-Khwârizmî 780-850)

- Description non ambiguë d'un nombre fini d'actions spécifiant une fonctionnalité d'un traitement automatique
- Diversité d'algorithmes pour une même fonctionnalité

Spécification d'un algorithme

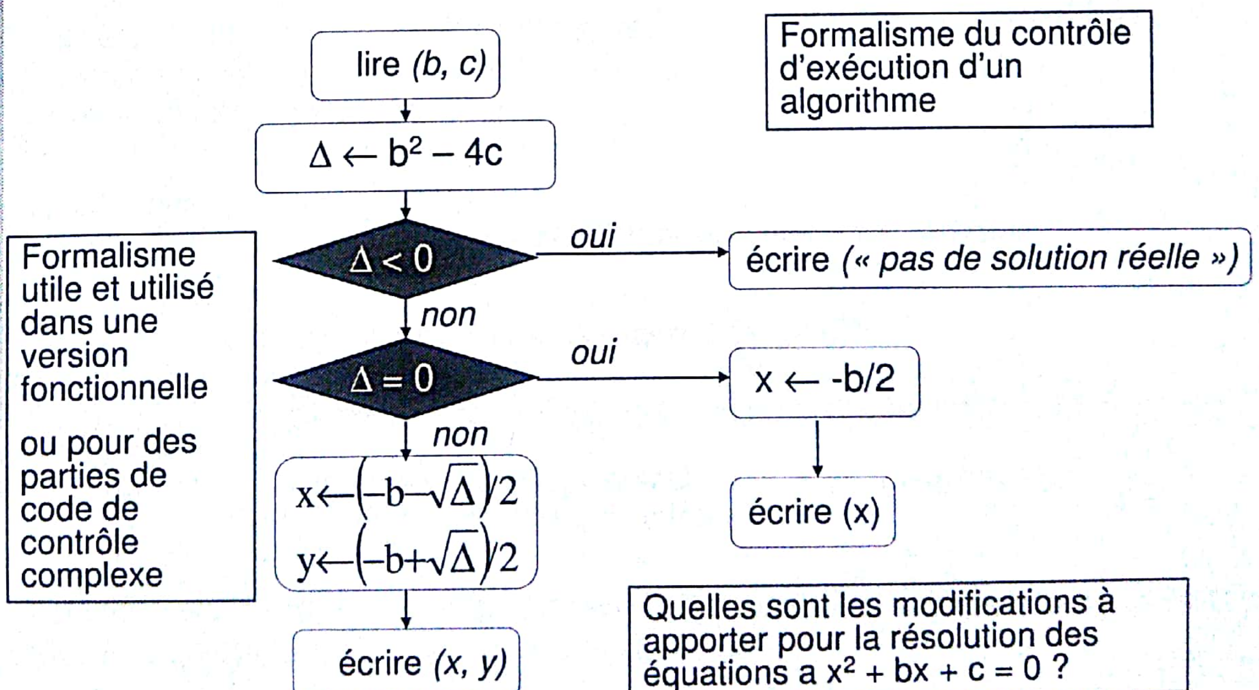
- Description de la fonctionnalité attendue
- Données d'entrée et de sortie
- Contraintes sur les données d'entrée (assertions sur les préconditions)

Expression d'un algorithme

- Description de l'algorithme en pseudo-langage PDL (Program Design Language) ou pseudo-code

Complexité d'un algorithme

2. EXPRESSION DES ALGORITHMES
Organigramme (Extrait du Cours 1)



Scanned by CamScanner

Pseudo-code (Extrait du Cours 1)

```
b, c : deux réels, lire(b, c)
delta←b²-4c
si delta<0 alors
   écrire(« pas de solution réelle »)
sinon
   si delta=0 alors
      x←-b/2
      écrire(x)
   sinon
      x←(-b-√delta)/2,  y←(-b+√delta)/2
      écrire(x, y)
   finSi
finSi
```

C'est la forme d'expression que nous retiendrons

Comparaison d'algorithmes

Comparaison de différents algorithmes ayant la même fonctionnalité (résolvant le même problème)

Indépendamment de la machine
- nombre de processeurs et architecture
- fréquence d'horloge et jeu d'instructions du processeur
- structure, taille et temps d'accès à la mémoire
- système d'exploitation

Indépendamment du langage de programmation
- efficacité du langage et du compilateur

Indépendamment des détails d'implémentation
- implémentation des structures de données (choix du conteneur : tableau, liste chaînée, liste à accès direct)

Propriété souhaitée

Algorithme $\mathcal{A}1$ demande moins de ressources que l'algorithme $\mathcal{A}2$,
- pour des données de taille suffisante,
- sur toutes les machines,
- quelque soit le langage de programmation,
- et les détails de l'implémentation

Scanned by CamScanner

Grandeurs physiques

Analyse des ressources de l'ordinateur utilisées pour exécuter l'implémentation d'un algorithme

- **Temps de calcul** pour exécuter les opérations
- **Occupation de la mémoire** pour contenir et manipuler les programmes ainsi que leurs données

Fonctions mathématiques approximant ces grandeurs physiques en fonction de la taille des données d'entrée

- Taille des données d'entrée / stockées en mémoire via un conteneur utilisé dans le programme
- Par exemple : le nombre d'éléments d'un tableau

Algorithme de complexité linéaire en temps
Temps de calcul proportionnel à la taille des données (n) approximé par $f(n)=n$

Algorithme de complexité quadratique en mémoire
Occupation mémoire proportionnelle au carré de la taille des données (n) approximée par $f(n)= n^2$

Opérations fondamentales

Dépendance des grandeurs physiques (temps de calcul et occupation mémoire) à la machine utilisée, au langage de programmation et aux détails d'implémentation

Notion d'opération fondamentale

- affectation et déplacement de données en mémoire centrale,
- comparaisons d'éléments,
- additions et multiplications flottantes,
- entrées/sorties sur mémoire de masse (disques)

Choix d'une ou de plusieurs opérations fondamentales selon l'algorithme

Exemple : Comparaisons de deux éléments pour les algorithmes de recherche en mémoire centrale

Fonction mathématique approximant le nombre d'opérations fondamentales en fonction de la taille des données d'entrée

Scanned by CamScanner

Un premier algorithme de recherche (linéaire)

Fonctionnalité

Chercher dans un tableau trié `t` d'entier un entier `e`
Retour : `vrai` ou `faux`

Principe de l'algorithme

Balayer le tableau `t` depuis son début jusqu'à sa fin jusqu'à trouver l'élément `e` (s'il existe)

Recherche linéaire – Exemple sur 10 éléments (1/3)

Application de l'algorithme sur un exemple (tableau de 10 éléments)

0	1	2	3	4	5	6	7	8	9
2	7	10	12	15	19	34	56	78	79

Trois cas de complexité usuellement considérés pour estimer la complexité d'un algorithme en fonction du nombre d'opérations fondamentales

Meilleur des cas Recherche la plus rapide de l'algorithme

Pire des cas Recherche la plus longue de l'algorithme

Cas moyen Recherche moyenne de l'algorithme

Scanned by CamScanner

`longueur(t)` renvoie la longueur du tableau `t`

4. ALGORITHME DE RECHERCHE

Recherche linéaire – Algorithme

Soit `t` un tableau d'entiers

Hypothèse : `t` est trié par ordre croissant

Problème : comment savoir si un entier `e` apparaît dans `t` ?

Un 1er algorithme

```
pour i allant de 0 à longueur(t)-1 faire
    si (t[i]=e) alors
        retourner vrai
    sinon
        si (t[i]>e) alors
            retourner faux
        finSi
    finSi
finPour
retourner faux
```

`longueur (t)` itérations au maximum

2 comparaisons par itération si l'élément n'est pas trouvé

Opération fondamentale de l'algorithme : la comparaison d'éléments

4. ALGORITHME DE RECHERCHE

Recherche linéaire – (2/3)

Exemple sur 10 éléments

```
pour i allant de 0 à longueur(t)-1
  faire
    si (t[i]=e) alors
        retourner vrai
    sinon
        si (t[i]>e) alors
            retourner faux
        finSi
    finSi
finFaire
retourner faux
```

Application de l'algorithme sur un exemple (tableau de10 éléments)

0	1	2	3	4	5	6	7	8	9
2	7	10	12	15	19	34	56	78	79

Meilleur des cas recherche de **2 => 1 comparaison**

Pire des cas recherche de **80** (l'élément n'est pas stocké dans le tableau) **=> 20 comparaisons**

Cas moyen Moyenne du nombre de comparaisons sur l'ensemble des éléments de l'exemple

recherche de 2 (1 comp.), recherche de 7 (3 comp.), recherche de 10 (5 comp.),
recherche de 12 (7 comp.), recherche de 15 (9 comp.), recherche de 19 (11 comp.),
recherche de 34 (13 comp.), recherche de 56 (15 comp.), recherche de 78 (17 comp.),
recherche de 79 (19 comp.).

(1 + 3 + 5 + 7 + 9 + 11 + 13 + 15 + 17 + 19) / 10= **10 comparaisons** en moyenne

Scanned by CamScanner

Recherche linéaire – Exemple sur 10 éléments (3/3)

Application de l'algorithme sur un exemple (tableau de10 éléments)

0	1	2	3	4	5	6	7	8	9
2	7	10	12	15	19	34	56	78	79

Meilleur des cas recherche de **2** ⇒ **1 comparaison**

Pire des cas recherche de **80** (l'élément n'est pas stocké dans le tableau) ⇒ **20 comparaisons**

Cas moyen Moyenne du nombre de comparaisons sur l'ensemble des éléments de l'exemple ⇒ **10 comparaisons** en moyenne

Recherche linéaire dans un tableau de taille n

Dans le pire des cas, il y aura **2*n** comparaisons

Dans le cas moyen, il y en aura la moitié, soit **n**

La complexité est une fonction linéaire de la taille des données (2*n et n)

Recherche dichotomique – Algorithme

Un 2ème algorithme

```
début←0, fin←longueur(t)-1
tantQue début≤fin faire
    milieu←(début+fin)/2
    si t[milieu]=e alors
        retourner vrai
    sinon
        si t[milieu]>e alors
            fin←milieu-1
        sinon
            début←milieu+1
        finSi
    finSi
finFaire
retourner faux
```


Scanned by CamScanner

Un deuxième algorithme de recherche (dichotomique)

Fonctionnalité

Chercher dans un tableau trié `t` d'entier un entier `e`
Retour : `vrai` ou `faux`

Principe de l'algorithme

Algorithme utilisé (de façon très approximative) lorsque l'on recherche d'un mot dans un dictionnaire

Recherche dichotomique – (1/6)

Exemple sur 10 éléments

```
début←0, fin←longueur(t)-1
  tantQue début≤fin faire
    milieu←(début+fin)/2
    si t[milieu]=e alors
      retourner vrai
    sinon
      si t[milieu]>e alors
        fin←milieu-1
      sinon
        début←milieu+1
      finSi
    finSi
  finFaire
retourner faux
```

Application de l'algorithme (10 éléments)

0	1	2	3	4	5	6	7	8	9
2	7	10	12	15	19	34	56	78	79

Meilleur des cas recherche de 15 ⇒ 1 comparaison

debut	fin	milieu
0	9	4

Itération 1

```
début←0, fin←longueur(t)-1
milieu←(début+fin)/2
```

0	1	2	3	4	5	6	7	8	9
2	7	10	12	15	19	34	56	78	79

↑ début (indice 0) ↑ milieu (indice 4) ↑ fin (indice 9)

Scanned by CamScanner

Recherche dichotomique – (2/6)

Exemple sur 10 éléments

```
début←0, fin←longueur(t)-1
  tantQue début≤fin faire
    milieu←(début+fin)/2
    si t[milieu]=e alors
      retourner vrai
    sinon
      si t[milieu]>e alors
        fin←milieu-1
      sinon
        début←milieu+1
      finSi
    finSi
  finFaire
retourner faux
```

Pire des cas recherche de 80

Itération 1

```
début←0, fin←longueur(t)-1
milieu←(début+fin)/2
t[milieu]<=80 ⇒ début←milieu+1
```

0	1	2	3	4	5	6	7	8	9
2	7	10	12	15	19	34	56	78	79

début (0), milieu (4), fin (9)

debut	fin	milieu
0	9	
5		

Itération 2

```
milieu←(début+fin)/2
t[milieu]<=80 ⇒ début←milieu+1
```

0	1	2	3	4	5	6	7	8	9
2	7	10	12	15	19	34	56	78	79

début (5), milieu (7), fin (9)

debut	fin	milieu
5	9	
8		

Recherche dichotomique – (4/6)

Exemple sur 10 éléments)

```
début←0, fin←longueur(t)-1
  tantQue début≤fin faire
    milieu←(début+fin)/2
    si t[milieu]=e alors
      retourner vrai
    sinon
      si t[milieu]>e alors
        fin←milieu-1
      sinon
        début←milieu+1
      finSi
    finSi
  finFaire
retourner faux
```

Itération 5

```
début>fin ⇒ Sortie de boucle
retour de faux
```

0	1	2	3	4	5	6	7	8	9
2	7	10	12	15	19	34	56	78	79

fin milieu (9), début (10)

debut	fin	milieu
10	9	

Pire des cas recherche de **80**

⇒ **8 comparaisons**

4 itérations avec 2 comparaisons par itération

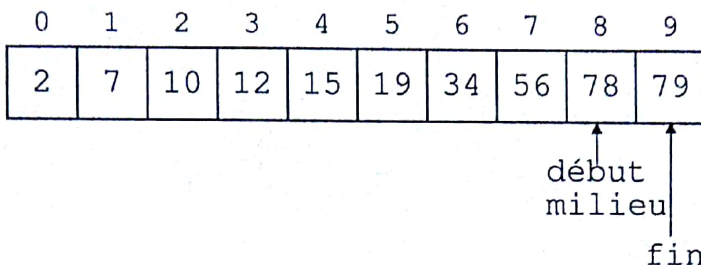
Scanned by CamScanner

4. ALGORITHME DE RECHERCHE
Recherche dichotomique – (3/6)
Exemple sur 10 éléments

```
début←0, fin←longueur(t)-1
  tantQue début≤fin faire
    milieu←(début+fin)/2
    si t[milieu]=e alors
      retourner vrai
    sinon
      si t[milieu]>e alors
        fin←milieu-1
      sinon
        début←milieu+1
      finSi
    finSi
  finFaire
retourner faux
```

Itération 3

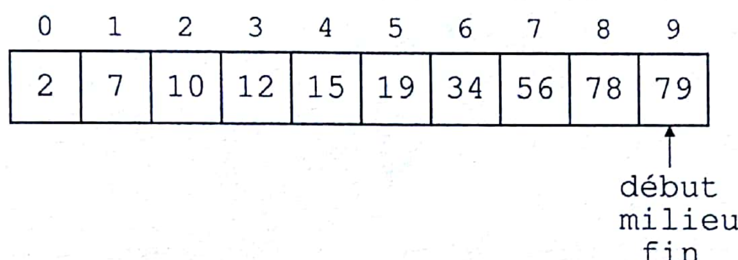
```
milieu←(debut+fin)/2
t[milieu]<=80 ⇒ début←milieu+1
```



debut	fin	milieu
8	9	
9		

Itération 4

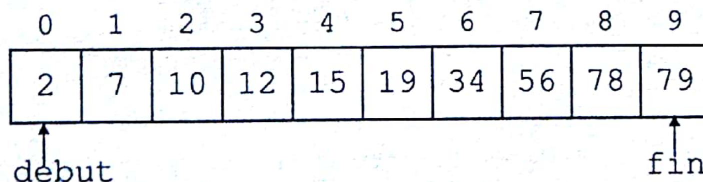
```
milieu←(début+fin)/2
t[milieu]<=80 ⇒ debut←milieu+1
```



debut	fin	milieu
9	9	
10		

4. ALGORITHME DE RECHERCHE
Recherche dichotomique – (5/6)
Exemple sur 10 éléments


```
début←0, fin←longueur(t)-1
  tantQue début≤fin faire
    milieu←(début+fin)/2
    si t[milieu]=e alors
      retourner vrai
    sinon
      si t[milieu]>e alors
        fin←milieu-1
      sinon
        début←milieu+1
      finSi
    finSi
  finFaire
retourner faux
```

debut	fin	milieu
0	9	

Même principe pour **le cas moyen :**
Moyenne du nombre de comparaisons nécessaire à la recherche sur l'ensemble des éléments de l'exemple

Codage pour le cas moyen, d'un programme de calcul du nombre de comparaisonspour la recherche d'un élément

Scanned by CamScanner

4. ALGORITHME DE RECHERCHE - Implémentation

Calcul du nombre de comparaisons pour la recherche d'un élément (1/3)

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

typedef enum {FALSE, TRUE} Boolean;

typedef struct {
    Boolean trouve; // FALSE(0) ou TRUE(1)
    int cpt;
} Solution;
```


4. ALGORITHME DE RECHERCHE - Implémentation

Calcul du nombre de comparaisons pour la recherche d'un élément (3/3)

```c
int main() {
   int i, som=0;
   int tab[10]={2, 7, 10, 12, 15, 19, 34, 56, 78, 79};
   Solution res;
   /* Cas moyen */
   for (i=0; i<10; ++i) {
      res=dichotomie(tab, 10, tab[i]);
      som=som+res.cpt;
      printf("Recherche t[%d]=%d : %d comparaisons et
      retour=%d\n", i , tab[i], res.cpt, res.trouve);
   }
   printf("Nombre moyen de recherche = %f\n", som/10.);

   /* Le pire des cas
   res=dichotomie(tab, 10, 80);

   printf("Le pire des cas - Recherche de %d : %d comparaisons
   et retour=%d\n", 80, res.cpt, res.retour);
   */
   system("pause"); return 0;
}
```


Scanned by CamScanner

Calcul du nombre de comparaisons pour la recherche d'un élément (2/3)

```
Solution dichotomie(int* t, int longueur, int e) {
   Solution r;
   int cpt=0;
   int debut=0, fin=longueur-1, milieu;
   assert(longueur>0);
   while (debut <= fin) {
      milieu=(debut+fin)/2;
      ++cpt;
      if (t[milieu]==e) {
         r.trouve=TRUE;
         r.cpt=cpt;
         return r;
      }
      else {
         ++cpt;
         if (t[milieu]>e)
            fin=milieu-1;
         else
            debut=milieu+1;
      }
   }
   r.trouve=FALSE;
   r.cpt=cpt;
   return r;
}
```


Recherche dichotomique – Exemple sur 10 éléments (6/6)

Cas moyen

Moyenne du nombre de comparaisons sur l'ensemble des éléments de l'exemple

	0	1	2	3	4	5	6	7	8	9
t	2	7	10	12	15	19	34	56	78	79
Nb de comparaisons	05	03	05	07	01	05	07	03	05	07

Recherche t[0]=2 : 5 comparaisons et retour=1
Recherche t[1]=7 : 3 comparaisons et retour=1
Recherche t[2]=10 : 5 comparaisons et retour=1
Recherche t[3]=12 : 7 comparaisons et retour=1
Recherche t[4]=15 : 1 comparaisons et retour=1
Recherche t[5]=19 : 5 comparaisons et retour=1
Recherche t[6]=34 : 7 comparaisons et retour=1
Recherche t[7]=56 : 3 comparaisons et retour=1
Recherche t[8]=78 : 5 comparaisons et retour=1
Recherche t[9]=79 : 7 comparaisons et retour=1

(5+3+5+7+1+5+7+3+5+7) / 10 =4.8 comparaisons en moyenne

Recherche dans un tableau de taille $n=2^k-1 \Rightarrow 2^k=n+1$, $k=\log_2(n+1)$
Dans le **pire des cas**, il y aura 2*k comparaisons = 2* $\log_2(n+1)$ comparaisons
Pour le **cas moyen**, le nombre de comparaisons est inférieur au pire des cas mais se rapproche du pire des cas (sans l'atteindre) quand n est grand

Scanned by CamScanner

Temps d'exécution en fonction de la taille du problème et de la complexité

Complexité

Taille du problème (n)	1	$log_2 n$	n	$n*log_2 n$	n^2	n^3	2^n
10^2	1 µs	7 µs	0,1 ms	0,7 ms	10 ms	1 s	4.10^7 Ga
10^3	1 µs	10 µs	1 ms	10 ms	1 s	17 mn	∞
10^4	1 µs	13 µs	10 ms	130 ms	1,7mn	12j	∞
10^5	1 µs	17 µs	0,1 s	1,7 s	2,8h	32 a	∞
10^6	1 µs	20 µs	1 s	20 s	12j	32 Ka	∞

Premier bilan sur la complexité

Un programme **doit** **être correct** et **avoir un temps d'exécution raisonnable**

La complexité algorithmique est relative à une ou plusieurs opérations fondamentales

- Complexité < n^2 ⇒ taille des données : quelconque
- Complexité entre n^2 et n^3 ⇒ taille des données : moyenne
- Complexité > n^3 ⇒ taille des données : petite

La performance des machines ne change pas l'efficacité de l'algorithme

Scanned by CamScanner

Complexité et problème/algorithme type

Complexité ↓

Complexité	Algorithme
$\log_2 n$	Recherche dichotomique
n	Recherche linéaire
$n*\log_2 n$	Tri rapide
n^2	Tri bulle
n^3	Multiplication de matrices
2^n	Recherche de chemin hamiltonien (probème du voyageur de commerce)

Comparaison de temps d'exécution

Une comparaison usuelle consiste à comparer deux algorithmes ($\mathcal{A}1$ et $\mathcal{A}2$) par leur temps d'exécution respectifs

Les systèmes actuels multi-tâches permettent une comparaison peu précise si l'on utilise un temps `Posix` obtenu par l'appel

```
((double)clock())/CLOCKS_PER_SEC;
```

où, `(double)clock()` représente le nombre de secondes écoulées depuis le 1er janvier 1970 à 00:00:00

Des appels systèmes permettent un calcul précis

en fonction des quanta alloués par le système pour l'exécution d'une tâche

Notion Différée Temps d'exécution d'une tâche

Scanned by CamScanner

Calcul du temps d'exécution (Posix) – Exemple (1/6)

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef enum {FALSE, TRUE} Boolean;

double heureCourante() {
   return ((double)clock()) / CLOCKS_PER_SEC;
}
```


Calcul du temps d'exécution (Posix) – Exemple (3/6)

```
Bool recherche_dichotomique(const int t[], int taille, int e) {
   int debut=0, fin=taille-1;
   assert(taille>=0);

   while (debut<=fin) {
      int milieu=(debut+fin)/2;
      if (t[milieu]==e)
         return TRUE;
      else
         if (t[milieu]>e)
            fin=milieu-1;
         else
            debut=milieu+1;
   }
   return FALSE;
}
```


Scanned by CamScanner

Calcul du temps d'exécution (Posix) – Exemple (2/6)

```
Bool recherche_lineaire(const int t[], int taille, int e) {
   int i;

   assert(taille>=0);
   for(i=0; i<taille; ++i) {
      if (t[i]==e)
         return TRUE;
      else
         if (t[i]>e)
            return FALSE;
   }
   return FALSE;
}
```


Calcul du temps d'exécution (Posix) – Exemple (4/6)

```
int main() {
   enum {TAILLE=100000, NB_RECH=TAILLE};

   int *tab, *rech;
   int i;
   double top, duree;

   /* Pour pouvoir être de très grande taille,
    * les tableaux doivent être alloués dynamiquement
    */
   tab = (int*) malloc(TAILLE*sizeof(int));
   rech = (int*) malloc(NB_RECH*sizeof(int));

   for(i=0; i<TAILLE; ++i)
      /* initialisation du Tableau par tous les nombres pairs
       * compris entre 0 et TAILLE*2 */
      tab[i]=2*i;

   /* Initialisation du générateur de nombres pseudo-aléatoires */
   srand((unsigned)time(NULL));
```


Scanned by CamScanner

Calcul du temps d'exécution (`Posix`) – Exemple (5/6)

```
for (i=0; i<NB_RECH; ++i)
    rech[i] = rand() % (TAILLE * 2); // des nombres aléatoires
                                      // entre 0 et TAILLE * 2
top = heureCourante();
for(i = 0; i < NB_RECH; ++i) {
   Boolean trouve = recherche_lineaire(tab, TAILLE, rech[i]);
   if ((trouve && rech[i]%2 == 1) || (!trouve && rech[i]%2 == 0)) {
      printf("BUG en lineaire!!!\n");
      return 1;
   }
}
duree = heureCourante() - top;
printf("temps de recherche lineaire : %.3fs\n", duree);
```


Bilan de ce que vous avez appris en cours

- La notion de complexité d'un algorithme est liée au nombre d'opérations fondamentales de l'algorithme
- Deux algorithmes de recherche sur des données triées, la recherche linéaire et la recherche dichotomique
- La complexité du meilleur cas, du pire cas et du cas moyen
- Le temps d'exécution en fonction de la taille du problème et de la complexité
- La problématique de la comparaison des algorithmes sur la base du temps d'exécution

Au prochain cours…

La suite des fonctions

Scanned by CamScanner

Calcul du temps d'exécution (Posix) – Exemple (6/6)

```
   top = heureCourante();
   for(i = 0; i < NB_RECH; ++i) {
      Boolean trouve = recherche_dichotomique(tab, TAILLE, rech[i]);
      if ((trouve && rech[i]%2 == 1) || (!trouve && rech[i]%2 == 0)) {
         printf("BUG en dichotomique!!!\n");
         return 1;
      }
   }
   duree = heureCourante() - top;
   printf("temps de recherche dichotomique : %.3fs\n", duree);

   free(rech);
   free(tab);

   system("pause");
   return 0;
}
```


Scanned by CamScanner

UNIVERSITÉ PARIS DESCARTES

IAP - Introduction à l'Algorithmique et à la Programmation

T02 – Fonction mathématique
T03 – Fonction informatique
T04 – Utilisation d'une bibliothèque
T11 – Contrat Concepteur-Programmeur
T12 – Type pointeur
T18 – Passage de paramètre de sortie
T21 – Choix du prototypage
T22 – Cas d'une saisie
T24 – Cas du swap
T26 – Fonctions et tableaux
T30 – Passage d'un tableau dans une fonction
T31 – Fonctions et chaînes de caractères
T35 – Passage d'une chaîne de caractères dans une fonction

Equipe pédagogique
Marie-José Caraty, Julien Rossit, Camille Kurtz, Jacques Alès-Bianchetti, Eloi Keita

Cours n° 5

Les fondamentaux de la programmation impérative

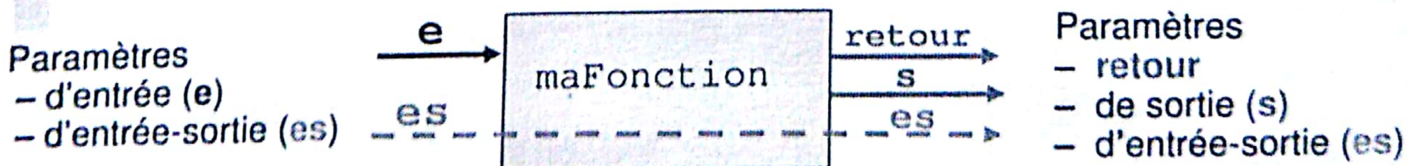
Les fonctions (2/2)

1. FONCTIONS – Généralités

Fonction informatique

Base de la programmation structurée
Portion de code conçue pour sa « ré-utilisation »
(rôle suffisamment « atomique », généralisation visée, ...)

Un concept plus étendu que les fonctions mathématiques
(diversité des paramètres)
Mais avec un point commun : le domaine de définition
qui est tel qu'il existe un résultat pour les paramètres d'entrée de la fonction



Impact Le domaine de définition de toute fonction programmée doit être précisé
Pour un appel sécurisé, l'utilisation sera contrôlée

Notion différée* – **Les paramètres d'entrées-sorties**

Scanned by CamScanner

Fonction mathématique – Domaine de définition

Soit la fonction f de $R \rightarrow R$

$$f(x) = \frac{\sqrt{x-1}}{x^2 - 4x + 4}$$

Domaine de définition (D_f) de la fonction f

$$D_f = \{x \in R \,/\, \exists r \in R \,/\, r = f(x)\}$$

est l'ensemble des éléments x de R tel que $f(x)$ existe (**calculable**)

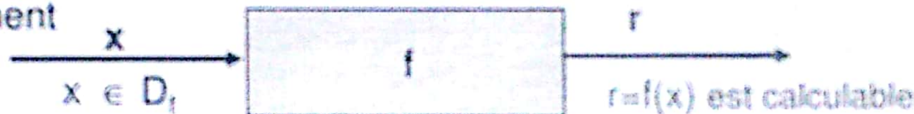
La racine de x-1 est définie pour $x \geq 1$

Le dénominateur (identité remarquable) $(x\text{-}2)^2$ s'annule pour x=2

$$D_f = [1; +\infty[- \{2\}$$

Toute étude/analyse de la fonction se fera dans D_f

Schématiquement



Utilisation d'une fonction de bibliothèque

Utilisation de la fonction `pow()` de la bibliothèque mathématique du langage C

Pour utiliser cette bibliothèque, inclure dans le `main()` son entête appelée "`math.h`" contenant les informations (prototypes) nécessaires à l'utilisation des fonctions mathématiques implémentées

```
#include <math.h>
```

Une bibliothèque est documentée

Linux manual
http://man7.org/linux/man-pages/man3/pow.3.html

Librairie standard du langage C
http://www.tutorialspoint.com/c_standard_library

Scanned by CamScanner

http://man7.org/linux/man-pages/man3/pow.3.html

Documentation – Fonction pow() (extrait) (1/2)

NAME
pow, powf, powl - power functions

Le nom de la fonction

SYNOPSIS
#include <math.h>

```
double pow(double x, double y);
...
```

Un résumé d'utilisation de la fonction
Le nom de la bibliothèque utilisée et à inclure dans le source

Le prototype permettant l'appel de la fonction

DESCRIPTION
The **pow()** function returns the value of x raised to the power of y.

Le rôle de la fonction
Documentation du retour de la fonction

RETURN VALUE
If x is a finite value less than 0, and y is a finite noninteger, a domain error occurs, and a NaN is returned.
If the result overflows, a range error occurs, and the functions return **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL**, ...

La valeur de retour
Description détaillée des cas où la fonction n'est pas calculable. Avec la particularité informatique de l'overflow lié au codage discret de l'information : un calcul conduisant à un dépassement de capacité de représentation 20 cas d'erreur décrits (overflow ou underflow)

Utilisation/appel de fonction – Fonction pow()

Appel de la fonction pow() pour calculer la racine carrée de delta

Prototype — identificateur de la fonction

```
double pow(double x, double y);
```

type de la valeur retournée par la fonction — Paramètres formels

Appel de la fonction
avec les **paramètres effectifs**
correspondant aux **paramètres formels** x et y
(correspondance en nombre, en type et dans l'ordre du prototype)

Scanned by CamScanner

Documentation – Fonction pow() (extrait) (2/2)

ERRORS

The following errors can occur:

Les erreurs possibles à l'utilisation

2 cas d'erreur pour des valeurs de x et y

$-3^{1/2}$

Domain error: x is negative, and y is a finite noninteger errno is set to EDOM. An invalid floating-point exception (FE_INVALID) is raised.

(1) x<0 et y un non entier fini

0^{-1}

Pole error: x is zero, and y is negative errno is set to ERANGE (but see BUGS). A divide-by-zero floating-point exception (FE_DIVBYZERO) is raised.

(2) x=0 et y est négatif (division par 0)

(1) et (2) valeurs pour lesquels il n'y a pas de solution (exclues du domaine de définition)

underflow

Range error: the result overflows errno is set to ERANGE. An overflow floating-point exception (FE_OVERFLOW) is raised.

overflow

Range error: the result underflows errno is set to ERANGE. An underflow floating-point exception (FE_UNDERFLOW) is raised.

2 cas d'erreur (overflow et underflow) liés au codage (numérique) de l'information en Informatique

Appel de fonction – Côté appelant

L'appelant est le `main()`
du programme de résolution de l'équation

```
printf("Deux racines reelles : %f et %f",
        -b-sqrt(delta))/2., (-b+sqrt(delta))/2.);
```

devient

```
printf("Deux racines reelles : %f et %f",
        -b-pow(delta, 0.5)/2., -b+pow(delta, 0.5)/2.);
```

La racine carrée est la puissance 1/2 =0.5

```
pow(delta, 0.5); // delta (pour x) et 0.5 (pour y)
```

```
double r; // memorise le résultat de la fonction
delta= b*b - 4.0*c;
r=pow(delta, 0.5); // delta (pour x) et 0.5 (pour y)
                   // ou delta (pour x) et 1./2.0 (pour y)
```


Scanned by CamScanner

Exemple – Utilisation de la fonction pow()

```
/* @file Cours-Exo01.c */
#include "stdafx.h"
#include <math.h>

int main() {
  double b=0., c=0., delta=0.;
  printf("Resolution de l'equation x^2+bx+c = 0\n");
  printf("Entrez les valeurs de b et c : ");
  scanf_s("%lf %lf", &b, &c);
  delta = (b*b)-(4*c);
  if (delta<0.)
    printf("Pas de racine reelle");
  else if (delta==0.)
      printf("Une racine reelle : %lf", -b/2);
      else
      printf("Deux racines reelles : %lf et %lf",
        (-b-pow(delta, 0.5))/2., (-b+pow(delta, 0.5))/2.);
  getchar(); return 0;
}
```

```
Resolution de l'equation x^2+bx+c = 0
Entrez les valeurs de b et c : 5 1
Deux racines reelles : -4.791288 et -0.208712
Cf. Cours1a T20
```


Contrat Concepteur-Programmeur

Documentation des fonctions

Rôle de la fonction
Paramètres formels
rôle des paramètres
mode de passage (`[in]` entrée, `[out]` sortie)

Paramètre de retour

Précondition(s)
domaine de définition de la fonction
domaine de validité des paramètres d'entrée

Corps de fonction

Associé aux préconditions, vérification des paramètres d'appel
utilisation de la fonction `assert()`

Scanned by CamScanner

Le rôle des bibliothèques informatique

Incontournable en Informatique
Permettent de démultiplier le développement des programmes

Qualité d'une bonne bibliothèque

Produit logiciel testé (sans trou de sécurité, ...)
Documentation des fonctionnalités
Rôle, prototype, tout paramètre et retour, domaine de définition, cas d'utilisation

Bonne Pratique

Documenter nos propres fonctions
Un code non documenté

Notion introduite – Précondition(s) d'une fonction et vérification

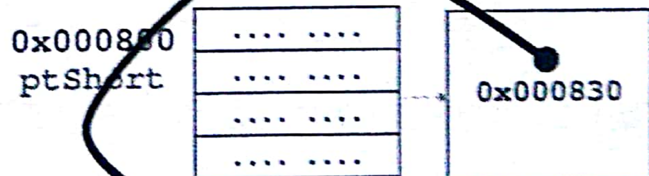
Rappel – Type pointeur

Exemple pour une adresse-mémoire sur 4 octets (processeur 32 bits) et un short sur 2 octets

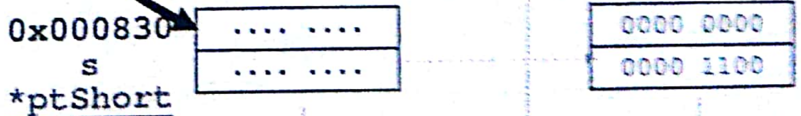
cf. Transp. 23 Cours 2

Un pointeur
(1) est une variable destinée à contenir une adresse mémoire
(2) est associé à un type d'objet

Déclaration du pointeur
`short* ptShort;` (t0)



`short s;` (t0)



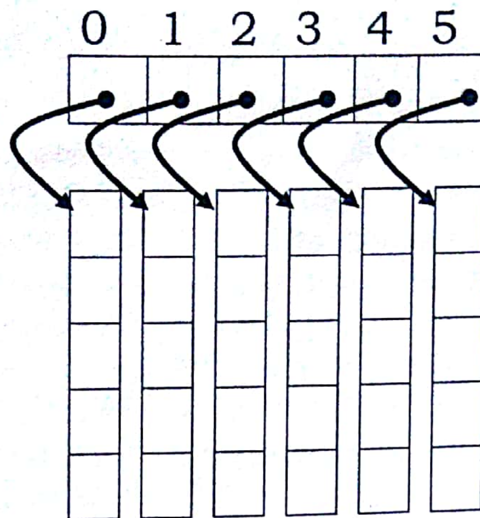
Initialisation du pointeur par référencement
& l'opérateur d'indirection – accès à l'adresse d'une variable
`ptShort=&s;` (t1)

Déréférencement
* l'opérateur de déréférencement – accès à l'objet référencé par le pointeur
`*ptShort=12;` (t2)

Scanned by CamScanner

Utilisation des pointeurs – Structures des données (1/2)

Enrichissement des structures de données
pour la modélisation



RAPPEL – Cours 3

Mécanisme et contexte d'exécution d'une fonction

- Contexte mémoire
- Mécanisme d'exécution
  - à l'appel
    - Mode de passage des paramètres effectifs aux paramètres formel
  - au retour

Scanned by CamScanner

2. POINTEURS ET UTILISATION

Utilisation des pointeurs – Passage de paramètres (2/2) de sortie dans une fonction

Principe

Utiliser un <u>paramètre formel</u> (`pFormel`) de <u>type pointeur</u> pour <u>modifier</u> le <u>paramètre effectif</u> (`pEffectif`) <u>dans l'appelant</u>

Prototype `void f(int* pFormel);`
Appel `int pEffectif;… f(&pEffectif);`

Le mode de passage par copie de la valeur du paramètre effectif dans le paramètre formel revient à l'affectation suivante

```
int* pFormel = &pEffectif; //pFormel pointe sur pEffectif
```

Dans le corps de la fonction `f`,
toute modification par déréférencement de `pFormel`
modifiera `pEffectif` dans son contexte d'exécution

Effet de bord

Modifier par déréférencement le paramètre formel dans le corps de la fonction revient à modifier le paramètre effectif
les modifications sont répercutées au contexte d'exécution de la fonction appelante

RAPPEL – Cours 3

2. FONCTIONS – Mécanisme d'exécution de fonction

Contexte d'exécution d'une fonction (1/2)

Toute fonction a son propre contexte d'exécution :
une zone mémoire allouée par le système (**pile d'exécution**)
que le flux d'exécution de la fonction va modifier
(par les entrées, les affectations et autres appels de fonction)

C'est à l'appel de la fonction que cette zone mémoire est réservée pour implanter

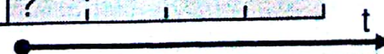
(1) les paramètres formels (entre les "(" ")" suivant l'identificateur de la fonction)

(2) les variables locales déclarées en début de bloc (instructions de la fonction)

Contexte d'exécution du main () fonction (ici) sans paramètre formel

```
L10 int main(){
L11    float b=0., c=0., delta=0.;
       ......
       system("pause"); return 0;
    }
```

main	L10	L11	L12	…
b	?			
c	?			
delta	?			



Scanned by CamScanner

2. **FONCTIONS** – Mécanisme d'exécution de fonction

Appel de fonction avec des paramètres effectifs (2/2)

La fonction est appelée avec les paramètres effectifs qui correspondent en nombre et en type aux paramètres formels

Le contexte d'exécution est mis en place

Les paramètres effectifs* sont passés aux paramètres formels

Le mode de passage est unique en Langage C
mode par **copie de valeur** du paramètre effectif au paramètre formel
(la valeur du paramètre effectif initialise le paramètre formel)
Rem : *un paramètre effectif est une expression
(variable, expression arithmétique, appel de fonction, ...) (cf. BNF)

(1) le paramètre effectif est tout d'abord évalué
(2) éventuellement converti dans le type du paramètre formel
(3) sa valeur est affectée au paramètre formel lui correspondant

Au retour de la fonction, la **valeur de retour (éventuelle) est transmise à l'appelant** et **l'espace mémoire alloué à l'appel est désalloué** (par le système)

Les variables du contexte d'exécution ne sont plus accessibles

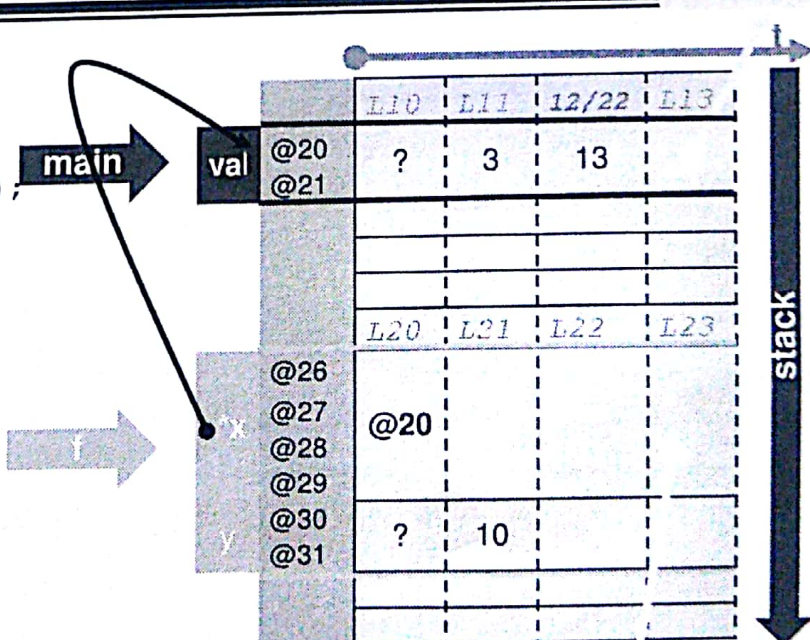
? état de la mémoire

3. **FONCTIONS** – Paramètres de sortie

Techniquement – Autre représentation de la pile d'exécution

val=13

```
L10 int main() {
L11         short val=3;
L12         f(&val);
L13         printf("val=%d\n", val);
L14         system("pause");
L15         return 0;
    }

L20 void f(short* x){
L21         short y=10;
L22         *x=*x+y; // *x+=y;
L23         printf("*x=%d\n", *x);
    }
```



*x=13

Dans f, après exécution de L23 :
fin d'exécution de f,
désallocation du contexte d'exécution de f
*x et y ne sont plus accessibles

Scanned by CamScanner

Techniquement – Fonction avec paramètre de sortie

(1) Appel du `main()`

```
L10 int main() {
L11  int val=3;
L12         f(&val);
L13         printf("val=%d\n", val);
L14         system("pause"); return 0;
    }
```

stack

main	L10	L11		
val	?	3		

(2) L12 : appel de `f()`

```
L20 void f(int* x){
L21         int y=10;
L22         *x=*x+y; // *x+=y;
L23         printf("*x=%d\n", *x);
    }
```

stack

@20

main	L10	L11	L12	
val	?	3	13	

f	L20	L21	L22	L23
*x	@20	@20	@20	@20
y	?	10		

*x=13

(3) Retour de f()

Après exécution de L23 :
fin d'exécution de f
désallocation du contexte d'exécution de f
*x et y ne sont plus accessibles

stack

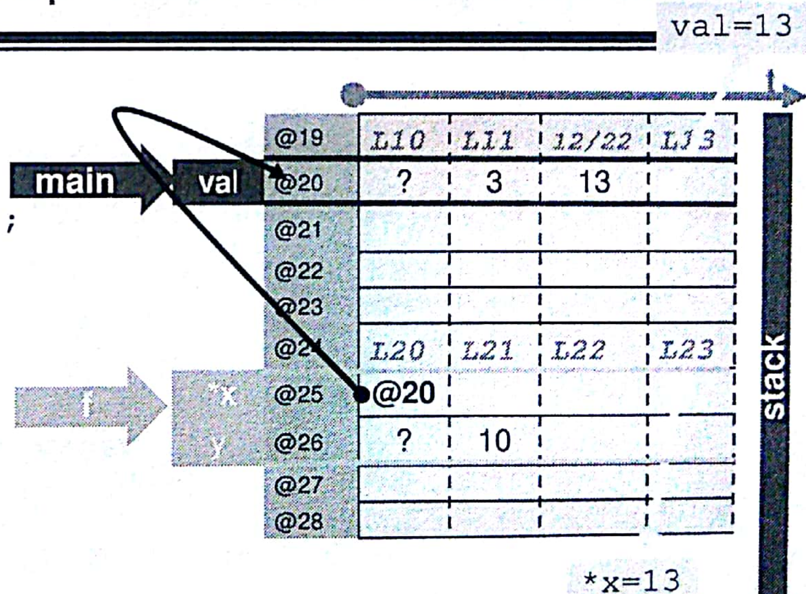
val=13

main	L13	L14		
val	13	?		

t

Techniquement – Autre représentation sans les tailles mémoire

```
L10 int main() {
L11         short val=3;
L12         f(&val);
L13         printf("val=%d\n", val);
L14         system("pause");
L15  return 0;
    }


L20 void f(short* x){
L21         short y=10;
L22         *x=*x+y; // *x+=y;
L23         printf("*x=%d\n", *x);
    }
```



Scanned by CamScanner

Choix du prototypage

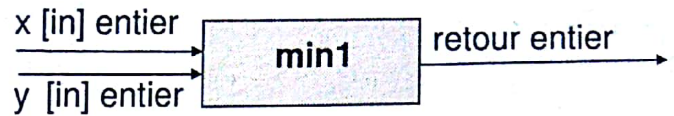
Plusieurs prototypes pour une même fonctionalité

Exemple : le minimum de deux entiers

```
min1(7, 3)=3
min2(7, 3)=3
```

Premier prototypage avec retour

```
int min1(int x, int y){
   if (x<y) return x;
   else return y;
```



Deuxième prototypage avec un paramètre de sortie

```
void min2(int x, int y, int* res){
   if (x<y)
        *res=x;
   else
        *res=y;
}
```

```
int main() {
   int x=7, y=3, res1, res2;
   res1= min1(x, y);
   printf("min1(%d, %d)=%d\n", x, y, res1);
   // ou printf("min1(%d, %d)=%d\n", x, y, min1(x, y));
   min2(x, y, &res2);
   printf("min2(%d, %d)=%d\n", x, y, res2);
   system("pause"); return 0;
}
```

Votre avis ?

Cas d'une saisie (2/2)

```
L20 void saisie(float* px) {
L21    printf("*px = %f\n", *px);
L22    printf("entrez un nombre :");
L23    scanf("%f", px);
L24    printf("*px = %f\n", *px);
L25    return;
    }
L10 int main() {
L11    float pi = 3.14;
L12    printf("pi = %f\n",pi);
L13    saisie(&pi);
L14    printf("pi = %f\n",pi);
L15    system("pause"); return 0;
    }
```

1. Exécution du programme (main)

`pi = 3.140000`

main	L11	L12
pi	3.14	3.14

2. Interruption d'exécution du main (L13) - appel de saisie

`entrez un nombre :`

main	L12	L13/23
pi (0x123)	3.14	3.14159

saisie	L20	L21	L22	L23	L24	L25
px	0x123	0x123	0x123	0x123	0x123	?

`*px = 3.140000`

`entrez un nombre : 3.14159`

`*px = 3.141590`

`pi = 3.141590`

main	L13	L14	L15
pi	3.14159	3.14159	?

Reg.0

float* px=π
px « pointe » sur pi

Scanned by CamScanner

Cas d'une saisie avec paramètre de sortie (1/2)

```
#pragma warning(disable: 4996)
#include <stdio.h>
#include <stdlib.h>

void saisie(float* px) {
  printf("*px = %f\n", *px);
  printf("entrez un nombre : ");
  scanf("%f", px);
  printf("*px = %f\n", *px);
  return;
}

int main() {
  float pi = 3.14;
  printf("pi = %f\n", pi);
  saisie(&pi);
  printf("pi = %f\n", pi);
  system("pause"); return 0;
}
```

```
pi = 3.140000
*px = 3.140000
entrez un nombre : 3.14159
*px = 3.141590
pi = 3.141590
```


Cas du `swap` – Deux paramètres d'entrée-sortie (1/2)

Exemple du `swap` échange du contenu de deux variables
deux résultats associés aux deux variables

Principe de résolution

Utiliser une variable temporaire
(pour stocker l'une des deux variables)

procéder à l'échange sur l'une des deux variables

mettre à jour l'autre variable (par la variable stockée)

Scanned by CamScanner

Cas du swap – Deux paramètres d'entrée-sortie (2/2)

```
#pragma warning(disable: 4996)
#include <stdio.h>
#include <stdlib.h>

void swap(float* u, float* v);

int main() {
    float x = 3.14, y = 9.81;
    printf("x=%.2f  y=%.2f\n", x, y);
    swap(&x, &y);
    printf("x=%.2f  y=%.2f\n", x, y);
}

void swap(float* u, float* v) {
     float aux = *u;
     *u = *v;
     *v =  aux;
}
```

```
x=3.14 y=9.81
x=9.81 y=3.14
```


RAPPEL

Tableaux statiques – Déclaration

Un tableau est une collection d'objets tous du même type

Déclaration de tableau statique

```
<type> <identificateur> [constante_entière]*
```

autant de [...] que de dimensions

```
short tab[5]; // un tableau de 5 entiers codes sur 2 octets
```

Bonne Pratique

Utiliser une constante pour définir la taille du tableau

```
const N=5; // Constantes non autorisées en C90
```

On utilisera une variable d'énumération de valeur 5

```
enum {N=5};
short tab[N];
```

tab	?	?	?	?	?
	0	1	2	3	4

Index du tableau : {0, .., 4}

accès au 4ème élément par `tab[3]`

Scanned by CamScanner

Fonctions et tableaux

Cas de passage d'un tableau
dans une fonction

RAPPEL

Schématiquement en mémoire

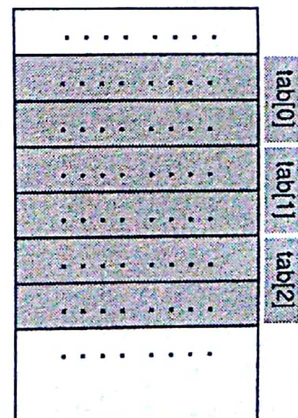
A la déclaration d'un tableau statique de 3 entiers de type `short`

```
short tab[3];
```

L'identificateur du tableau est l'adresse
du premier élément (1er octet)
alloué en mémoire (pile d'exécution)
le **type** de `tab` est un pointeur sur un `short`
c'est-à-dire **`short* tab;`**

	Adresses
tab	0x00A126E
	0x00A126F
tab+1	0x00A1270
	0x00A1271
tab+2	0x00A1272
	0x00A126B
	0x00A1274

Zone contigüe en mémoire
des 3 éléments du tableau
(arithmétique des pointeurs)



.... état (aléatoire) de la mémoire

Scanned by CamScanner

Cas du swap – Deux paramètres d'entrée-sortie (2/2)

```
#pragma warning(disable: 4996)
#include <stdio.h>
#include <stdlib.h>

void swap(float* u, float* v);

int main() {
   float x = 3.14, y = 9.81;
   printf("x=%.2f  y=%.2f\n", x, y);
   swap(&x, &y);
   printf("x=%.2f  y=%.2f\n", x, y);
}

void swap(float* u, float* v) {
     float aux = *u;
     *u = *v;
     *v =  aux;
}
```

```
x=3.14 y=9.81
x=9.81 y=3.14
```


RAPPEL

Tableaux statiques – Déclaration

Un tableau est une collection d'objets tous du même type

Déclaration de tableau statique

`<type> <identificateur> [constante_entière]*`

autant de [...] que de dimensions

```
short tab[5]; // un tableau de 5 entiers codes sur 2 octets
```

Bonne Pratique

Utiliser une constante pour définir la taille du tableau

```
const N=5; // Constantes non autorisées en C90
```

On utilisera une variable d'énumération de valeur 5

```
enum {N=5};
short tab[N];
```

tab	?	?	?	?	?
	0	1	2	3	4

Index du tableau : {0, .., 4}

accès au 4ème élément par `tab[3]`

Scanned by CamScanner

Fonctions et tableaux

Cas de passage d'un tableau

dans une fonction

RAPPEL

Schématiquement en mémoire

A la déclaration d'un tableau statique de 3 entiers de type `short`

```
short tab[3];
```

L'identificateur du tableau est l'adresse
du premier élément (1er octet)
alloué en mémoire (pile d'exécution)
le **type** de `tab` est un pointeur sur un `short`
c'est-à-dire **`short* tab;`**

Zone contigüe en mémoire des 3 éléments du tableau (arithmétique des pointeurs)

	Adresses	Mémoire	
		....	
tab	0x00A126E	....	tab[0]
	0x00A126F	....	
tab+1	0x00A1270	....	tab[1]
	0x00A1271	....	
tab+2	0x00A1272	....	tab[2]
	0x00A126B	....	
	0x00A1274	....	

.... état (aléatoire) de la mémoire

Scanned by CamScanner

Tableaux statiques – Exemple

```
#include <stdio.h>
#include <stdlib.h>

enum {TAILLE=5};

void initialiser(int tab[]) {     // ou int* tab
    for (int i=0; i<TAILLE; ++i)
        tab[i]=i+1;
}

void afficher(const int tab[]) { // ou const int* tab
    for (int i=0; i<TAILLE; ++i)
        printf("tab[%d]=%d ", i, tab[i]);
    printf("\n");
}

int main() {
    int t[TAILLE];
    initialiser(t);
    afficher(t);
    system("pause"); return 0;
}
```

Fonction d'initialisation d'un tableau de 5 entiers (de 1 à 5)
Fonction d'affichage du tableau

Dans le cas d'un paramètre d'entrée (type tableau ou structuré), on utilisera le modifieur `const` pour prévenir, par erreur du compilateur, toute modification dans le corps de la fonction

```
tab[0]=1 tab[1]=2 tab[2]=3 tab[3]=4 tab[4]=5
```


Fonctions et chaînes de caractères

Passage d'une chaîne de caractères

dans une fonction

Scanned by CamScanner

Passage de tableau dans une fonction

Rappel

`tab` est l 'adresse sur le premier élément du tableau

n'intègre donc aucune information sur la taille de tab

Pour traiter un tableau dans une fonction, deux paramètres sont à passer

(1) la variable tableau

(2) sa taille

Ou encore,

encapsuler le tableau et sa taille dans une structure `Tableau`

```
typedef struct {
   {enum n=20};
   int tab[n];
   int taille=n;
} Tableau;
```


RAPPEL

Chaînes de caractères – Littéral (1/3)

Représentation interne d'une chaîne de caractères littérale

```
printf("Hello, world!");
```

H	e	l	l	o	,		W	o	r	l	d	!	\0
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Taille utile (13 octets)

Taille occupée en mémoire (14 octets)

Une chaîne est stockée dans un tableau de caractères

Attention

'a' est différent de "a"

'a' est un caractère stocké en mémoire sur un caractère

"a" est une chaîne de caractère stockée sur 2 caractères ('a' et '\0')

Scanned by CamScanner

Chaînes de caractères – strlen() et sizeof() (2/3)

A savoir

Une chaîne de caractères est stockée dans un tableau de caractères et est initialisée par un littéral chaîne

```
char s1[80] = "Hello!";
char s2[] = "Hello, you!";
```

`strlen(...)` fonction de la bibliothèque `<string.h>`
`strlen(s1)` longueur utile de la chaîne s1 (sans le délimiteur '\0')

`sizeof(...)` opérateur de C
donne la taille mémoire (en octets) d'une variable/type

`strlen(s1)` vaut 6
`strlen(s2)` vaut 11

`sizeof(s1)` vaut 80
`sizeof(s2)` vaut 12
`sizeof(int)` vaut 4

Passage d'une chaîne de caractères dans une fonction

Une chaîne de caractères `c` est un tableau de caractères
Pour son passage comme paramètre formel d'une fonction
utiliser : `char* c` ou `char c[]`
Remarque : `'\0'` est le caractère délimiteur de fin de chaîne

La fonction `strlen(c)` donne la longueur « utile »
de la chaîne de caractères `c`

La chaîne de caractères (`c`) ne pose pas le problème
du passage de sa taille dans une fonction
en raison de son caractère délimiteur de fin de chaîne

Exemple : `void conversion(char* c);`

Scanned by CamScanner

RAPPEL

Chaînes de caractères – Exemple (3/3)

```
/* @file Cours2-Exo07.c */
#include "stdafx.h"

int main()
{
    char s1[80]="Hello!"; // initialisation par un littéral chaîne
    char s2[] = "Hello, you!";

    printf("strlen(s1)=%d\n", strlen(s1));
    printf("sizeof(s1)=%d\n\n", sizeof(s1));

    printf("strlen(s2)=%d\n", strlen(s2));
    printf("sizeof(s2)=%d\n", sizeof(s2));

    system("pause"); return 0;
}
```

```
strlen(s1)=6
sizeof(s1)=80

strlen(s2)=11
sizeof(s2)=12
```

```
strlen() : taille utile
sizeof(): taille mémoire occupée
```


Bilan de ce que vous avez appris en cours

- La notion de fonction mathématique et la notion de fonction informatique
  Leurs différences et leur point commun
- La documentation d'une bibliothèque et l'importance de la documentation de son domaine de définition
  La nécessité de donner le domaine de définition des fonctions informatiques (préconditions et leur vérification à l'utilisation)
- Le passage des paramètres de sortie par l'utilisation de pointeur et l'effet de bord qu'ils permettent
- Le passage d'un tableau et d'une chaîne de caractères dans une fonction

Au prochain cours…

La qualité de code et quelques spécificités des dossiers de développement logiciel

Scanned by CamScanner