



IAP - Introduction à l'Algorithmique et à la Programmation

Equipe pédagogique

Marie-José Caraty, Denis Poitrenaud,

Julien Rossit, Camille Kurtz,

Jacques Alès-Bianchetti, Denis Jeanneau

Cours de IAP 2019-2020

IAP - Introduction à l'Algorithmique et à la Programmation

T03 – Organisation du cours
T04 – Objectifs du cours
T05 – Logiciel
T06 – Composants matériels
T08 – Du source à l'exécutable
T11 – Spécification des langages
T12 – Mots réservés
T13 – Notation BNF
T15 – Diagrammes de Conway
T16 – Tests des logiciels

Equipe pédagogique

Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau

Tous mes remerciements à Mikal Ziane
pour son concours à ce cours préliminaire à IAP

Cours préliminaire Les fondamentaux de la programmation impérative Le contexte de la programmation

1

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty 2019-2020

1. INTRODUCTION A L'ALGORITHMIQUE ET A LA PROGRAMMATION

Bibliographie

Kernighan, B. W., & Ritchie, D. M. (1988). The C programming language (Vol. 2). Englewood Cliffs: prentice-Hall.

Braquelaire, J.-P. (2000). Méthodologie de la programmation en C: bibliothèque standard-API POSIX. Dunod.

Granet, V. (2012). Mini manuel d'algorithmique et de programmation. Dunod.

Quelques sites :

<http://man7.org/linux/man-pages/>

http://www.tutorialspoint.com/c_standard_library/

<http://www.montefiore.ulg.ac.be/~dethier/montef/docs/langage-c.pdf> (BNF du C)

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

<http://picolibre.int-evry.fr/projects/svn/course/Index.pdf>

2

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty 2019-2020

1. INTRODUCTION A L'ALGORITHMIQUE ET A LA PROGRAMMATION

Organisation du cours

Equipe pédagogique

- Marie-José Caraty
- Denis Poitrenaud
- Julien Rossit
- Camille Kurtz
- Jacques Alès-Bianchetti
- Denis Jeanneau

Répertoire d'accès aux cours, TDs et TP :
COMMUN/DUT_1ere_annee/IAP

7 semaines de cours - par semaine : 1 cours, 1 TD et 2 TP

Contrôle des connaissances

- Un **projet**, recette la semaine du 21 octobre coef. **1.5**
- Un **DST**, la semaine du 4 novembre coef. **3**

Environnement de programmation

- Visual Studio 2019**: EDI (Environnement de Développement Intégré) propriété de Microsoft (**VB**, **C**, **C++**, **C#**,...)
- Installation à partir du site officiel (licence de site université Paris Descartes, cf. « *Install party* »)

3

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty 2019-2020

1. INTRODUCTION A L'ALGORITHMIQUE ET A LA PROGRAMMATION

Objectifs du cours

Les fondamentaux de la programmation impérative

Le paradigme de la programmation impérative est le plus répandu parmi les langages de programmation

Il est fondé sur les machines de Turing et de Von Neumann

Dans l'évolution des langages impératifs, les paradigmes de programmation structurée, modulaire, objet, orientée objet, ...

Algorithmique

Problème et méthode de résolution (algorithme)

Les bases : les algorithmes « naïfs » et « non naïfs »

L'apprentissage de la programmation

- avec le langage C (Ritchie, 1972) conçu à l'origine pour le développement de système d'exploitation
- le développement d'un projet suivant les principes préconisés en Génie Logiciel

4

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty 2019-2020

Le logiciel

Logiciel

l'ensemble des programmes, des ressources (données) et des procédures nécessaires au fonctionnement d'un système informatique

Omniprésent dans la vie courante

aéronautique, automobile, énergie, défense, télécommunications, robotique, économie, finance, relation client, réseaux sociaux,

Nécessité de savoir maîtriser la programmation

Les erreurs (bugs, mauvaise utilisation, trous de sécurité, mauvaise modélisation, ...) peuvent coûter cher, très cher...

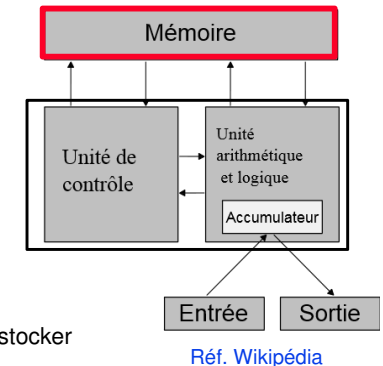
Terminologie : programme, code-source, code, source

un fichier texte dont le nom a une extension caractéristique du langage (.c en Langage C)

Composants matériels – Architecture de Von Neumann (1/2)

L'architecture de Von Neumann

- Mémoire centrale
- Processeur (UC et UAL)
- Périphériques d'entrée-sortie



Mémoire centrale

l'espace-mémoire permettant de mémoriser/stocker

- des données en cours de traitement
- le programme lui-même : des instructions qui à l'exécution modifieront **l'état de la mémoire** grâce à des affectations successives

Réf. Wikipédia

Composants matériels – Architecture de Von Neumann (2/2)

Processeur (microprocesseur)

C'est à travers lui que tout programme s'exécute

Il ne connaît qu'un seul langage, qui lui est propre : le langage-machine
Ce langage est constitué de l'ensemble des instructions élémentaires (instructions-machine) exécutables par le processeur

Le langage est exprimé sous un format binaire constitué de deux symboles élémentaires (bits – binary digits) notés par convention **0** et **1**

Usage Le nom donné à un langage-machine est celui du microprocesseur (qui le comprend)
Exemple, le 68000 est le langage-machine du microprocesseur Motorola 68000
56 instructions élémentaires qui combinées avec les modes d'adressage représente plus de 1000 instructions

Bus, ports d'entrées-sorties, périphériques

- l'échange (mémoire /processeur/ports)
- la sauvegarde des données (périphérique de stockage de masse)

Du source à l'exécutable – La machine de Turing (1/3)

Source et exécutable

Le source (fichier texte) est écrit en langage « de haut niveau » (e.g., Langage C) doit être traduit en langage-machine pour être exécuté

Deux types de traducteur

- Compilateur associé à un éditeur de liens
Deux phases : compilation et édition de lien d'un programme exécutable (.exe) en langage-machine
- Interpréteur : le source est analysé et exécuté à la demande par un programme appelé l'interpréteur qui vérifie la syntaxe et exécute le code

Exemples :

Le langage C est un langage compilé, Python est un langage interprété

Le premier prix Turing en 1966 a été attribué pour la construction de compilateurs à Alan J. Perlis (USA)

Prix attribué tous les ans pour une contribution technique majeure et durable dans le domaine informatique

Et le fameux test de Turing...

Compilateur (compiler)

Logiciel qui traduit le source en un code appelé le code-objet

Objectif

Produire un code-objet : i) correct, ii) optimisé (le code-objet doit s'exécuter le plus vite possible)

Etapas de la compilation

- Analyse lexicale : découpage du programme en lexèmes (mots du langage)
- Analyse syntaxique : vérification de la correction de la syntaxe du source
- Analyse sémantique : analyse des structures de données
- Transformation du source en code intermédiaire
- Application de techniques d'optimisation sur le code intermédiaire
- Transformation du code intermédiaire en code-objet, avec éventuellement l'insertion de données de débogage

Retour du compilateur de messages d'erreur/warning que le programmeur doit corriger dans le source (e.g., *variable non déclarée*, *type inconnu*, *;" attendu...*)

Editeur de liens (linker)

Seule une compilation sans erreur du source permet de passer à l'édition de lien

Son rôle

Construire une image mémoire contenant l'ensemble des sources du logiciel compilées séparément (modules/sous-programmes, fonctions de bibliothèques, ...)

A son terme (sans erreur)

l'exécutable (.exe) du source (.c) peut être exécuté

Spécification des langages de programmation

Langage de programmation

- Lexique (vocabulaire) incluant les mots réservés du langage
- Syntaxe (grammaire)
- Sémantique (sens)

Un langage de programmation est spécifié dans son manuel de référence

- précisément
- sans ambiguïté d'interprétation

Définition des aspects lexicaux et syntaxiques

Règles de grammaire hors-contexte (règles syntaxiques)

Deux formalismes usuels pour la description des règles syntaxiques

- Notation BNF (Backus Naur Form)
- Diagramme de Conway

Mots réservés du langage C – C90 (ISO/IEC 9899:1990)

C

| | | | | |
|----------|---------|--------|----------|--------|
| auto | break | case | char | const |
| continue | default | do | double | else |
| enum | extern | float | for | goto |
| if | int | long | register | return |
| short | signed | sizeof | static | struct |
| switch | typedef | union | unsigned | void |
| volatile | while | | | |

Identificateurs non autorisés dans le source

pour les noms de variable, de fonction et de type

Méta-langage adapté à la représentation des grammaires hors-contexte

- Méta-symboles

| | |
|-----|--|
| ::= | définition d'une catégorie syntaxique |
| <> | délimiteurs d'une catégorie (sans délimiteur : littéraux du langage) |
| | ou logique, |
| { } | répétition d'item (0 ou plusieurs fois) |
| [] | item optionnel (0 ou une fois) |

*Diverses extensions...
En gras, les littéraux
du langage*

- Symboles terminaux

Littéraux/mots/lexèmes du langage (mots réservés)

- Symboles non terminaux

Catégories syntaxiques décrivant le langage

Règle de formation d'un identificateur

Un identificateur est une séquence composée de lettres (minuscules ou majuscules, à l'exception des caractères accentués), de chiffres, du caractère de soulignement (_) ou du caractère dollar (\$) ; il ne doit pas commencer par un chiffre, les espaces ne sont pas autorisés.

Règle BNF (indépendante du contexte)

<identificateur> ::= <lettre> {<lettre> | <chiffre>}

*Catégories syntaxiques :
identificateur, lettre, chiffre*

<lettre> ::= **A|a|B|b|C|c|.....Z|z|\$_**

<chiffre> ::= **0|1|2|...|9**

Symboles terminaux en gras

Règle récursive équivalente

<identificateur> ::= <lettre> | <identificateur> <lettre> | <chiffre>

Exemples d'identificateurs :

Jhkjhj, JKK, nbClients, _items, // valides
2emeJoueur, catégorie, @adresse // invalides

Pour la BNF du Langage C

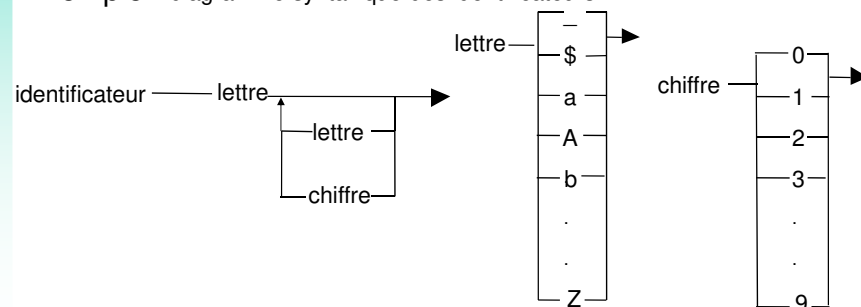
cf. <http://www.montefiore.ulg.ac.be/~dethier/montef/docs/langage-c.pdf>

Expression graphique des règles syntaxiques

Principe de formation d'une catégorie syntaxique

Tout chemin allant de l'origine du diagramme à son extrémité (fléchée) engendre une expression de la catégorie appartenant au langage

Exemple : diagramme syntaxique des identificateurs



Une particularité en informatique

On ne peut pas démontrer qu'un logiciel dédié à une application « fonctionne » (est sans erreur)

Recherches sur les preuves de programme...

Le test fonctionnel est utilisé, il consiste à exécuter un programme et vérifier son bon fonctionnement pour des données du problème appelées Jeu de Données de Test (JDT) :

les résultats observés sont bien les résultats attendus

Un bon test consiste à définir un ensemble de JDT qui « couvre » l'application et à vérifier le bon fonctionnement de l'application sur ces JDT

Une bonne couverture des tests est obtenue à partir d'une analyse partitionnelle des données liée aux résultats type de l'application et à la sélection d'un JDT représentatif de chacune des partitions

Suite en période Projet avec un développement agile...

IAP - Introduction à l'Algorithmique et à la Programmation

T19 – Point d'entrée d'une application
T20 – Unité de compilation
T22 – Blocs d'instructions et flot d'exécution
T24 – Notion de type
T25 – Types natifs
T26 – Types signés et non-signés
T27 – Domaine de variation
T29 – Représentation des caractères
T31 – Notion de variable
T32 – Déclaration de variable, sémantique et typage
T35 – Affectation de variable
T36 – Simulation d'exécution de programme
T38 – Portée des déclarations
T40 – Littéraux

Equipe pédagogique

*Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau*

Cours préliminaire

Les fondamentaux de la programmation impérative

Les éléments d'un langage de programmation

17

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty 2019-2020

Sommaire

1. Structure de programme

- Unité de compilation
- Point d'entrée d'une application

2. Données

- Types
- Variables
- Portée des déclarations

3. Traitements

Structure de programme
Données

18

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty 2019-2020

1. STRUCTURE DE PROGRAMME

Problème : Résolution de $x^2+bx+c=0$

Point d'entrée d'une application – La fonction main()

```
/* @file Cours1a-Exo01.c - Résolution  $x^2+bx+c=0$  */
#include "stdafx.h" // *1) Inclusion de bibliothèques
int main() {
    float b=0., c=0., delta=0.;
    printf("Résolution de l'équation  $x^2+bx+c = 0$ \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 réelle");
    else
        if (delta==0.) // 1 racine double
            printf("Une racine réelle : %f", -b/2);
        else // 2 racines réelles
            printf("Deux racines réelles : %f et %f",
                -b-sqrt(delta)/2., (-b+sqrt(delta))/2.);
    system("pause");
    return 0;
}
```

Structures de programme

19

DUT Informatique, 1^{ère} année – *1) "stdio.h", "stdlib.h" et "math.h"

1. STRUCTURE DE PROGRAMME

Unité de compilation : unité de base

(1/2)

Unité de compilation

Fichier source (fichier texte) compilable

Structure minimale d'un programme

Fichier source contenant un point d'entrée de l'application

- La fonction **main()** également appelée programme principal qui sera exécutée
- Le **corps** de la fonction est délimité par un **bloc** d'instructions repéré par deux accolades **{ }** et qui contient les instructions à exécuter. Chaque instruction se termine par **;**

Structure d'un programme

Un ensemble d'unités de compilation organisées en fichiers texte (fichiers physiques répertoriés dans le système de fichier) avec un seul point d'entrée

Notion différée – Compilation séparée

20

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty 2019-2020

```
/*@file lazyProgram.c */
int main() {
    return 0;
}
```

Texte inclus dans un fichier texte
unité de compilation « minimaliste » en C (**lazyProgram.c**)

```
int main() // Prototype de la fonction
```

La fonction **main** retourne un **int** au système :

```
{ 0    si le programme a été exécuté sans erreur
 ≠0    en cas d'erreur à l'exécution ( détectée avant l'exécution
                                     du return 0;)
```

Elle est syntaxiquement correcte (i) compilée sans erreur et (ii) exécutable

Le programme ne fait rien de plus que de renvoyer 0 au système
pour signifier que tout se passe très bien ...

```
Lazier... int main() { } // qui compile sans erreur !!!
```

Bloc

- Regroupe un certain nombre (éventuellement nul) d'instructions
- Délimité par les accolades "{" (début de bloc) et "}" (fin de bloc)
- Un bloc peut contenir d'autres blocs (imbriqués)

Exemples :

```
{ } ou ;           // bloc vide, sans instruction
{ i=i+1; }         // une instruction, les accolades peuvent être omises
{ float t=0.;      // bloc de deux instructions
  t=t+1;
}
```

Flot d'exécution

Les instructions d'un code source sont exécutées en séquence

- i) en partant de la première ligne de code
- ii) de « gauche » à « droite »
- iii) de « haut » en « bas »

```
/* @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 réelle");
    else
        if (delta==0.) // 1 racine double
            printf("Une racine réelle : %f", -b/2);
        else // 2 racines réelles
            printf("Deux racines réelles : %f et %f",
                -b-sqrt(delta)/2., (-b+sqrt(delta))/2.);
    system("pause"); return 0;
}
```

Données

Information sémantique associée à une donnée

Le type d'une donnée est le moyen de définir dans un langage informatique

- le nombre de bits utilisés pour sa représentation en mémoire (place mémoire)
- le type de codage utilisé pour sa représentation interne
- son domaine de variation (valeurs minimale et maximale)
- l'ensemble des opérations qui lui sont applicables

Remarque

La notion de type est puissante
à chaque type, un ensemble d'opérations peut être appliqué
aux variables déclarées de ce type

Par exemple : +, -, *, /, % pour les types entiers

Dans les Langages Objets, une classe est conceptuellement un type de données
pour laquelle on définit les opérations/méthodes qui s'appliquent à toute variable
déclarée de cette classe

n bits permettent de coder 2ⁿ valeurs distinctes

Ex. : Sur 1 octet = 8 bits, on peut coder 2⁸=256 valeurs distinctes

Types natifs du Langage C

(1/6)

Entre crochets, le nombre d'octets de la représentation interne

Types scalaires

| Type entier | | |
|----------------|-------|--|
| char | [1] | |
| unsigned char | [1] | |
| short | [2] | |
| unsigned short | [2] | |
| int | [4] | |
| unsigned int | [4] | |
| long | [4/8] | |
| unsigned long | [4/8] | |
| Types réels | | |
| float | [4] | |
| double | [8] | |

Types flottants

Remarque sur les types natifs (définis dans le langage)

Non standards en C et en C++ dépendent du micro-processeur, du système d'exploitation et du compilateur

Le modificateur **unsigned** désigne des positifs

cf. module M1101

Codage binaire et codage par complément à 2

Codage des flottants (norme ANSI/IEEE Std 754)

Types signés et non signés

(2/6)

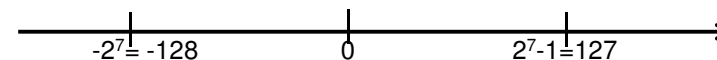
Exemple du type numérique `char` codé sur 1 octet (8 bits)

Les types signés (sans modificateur)

représentent les négatifs et les positifs en même proportion

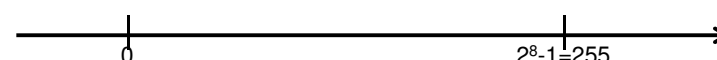
Codage de la représentation interne par complément à 2 (cf. module M1101)

Convention : 0 est positif

Types non signés (avec le modificateur **unsigned**)

ne représentent que les valeurs positives

Codage binaire de la représentation interne



Opérations arithmétiques autorisées (+, -, *, /, %)

Domaine de variation des types scalaires

(3/6)

Dynamique de codage des types entiers signés

| Taille (octet) | Valeurs min .. max | Ordre de grandeur |
|----------------|--|--|
| 1 | -128 .. 127 | |
| 2 | -32 768 .. +32 767 | |
| 4 | -2 147 483 648 .. +2 147 483 647 | ≅ 2 milliards (2 . 10 ⁹) |
| 8 | -9 223 372 036 854 775 808 .. +9 223 372 036 854 775 808 | ≅ 9 000 péta (10 ¹⁵) ≅ 9 exas (9 x 10 ¹⁸) |

Constantes (minimum et maximum) des domaines de variation des types sont définies dans le fichier d'entête `<limits.h>`

cf. TP semaine 1 – Domaine de variation des types natifs du langage

Types flottants

(4/6)

Précision des types flottants et valeurs absolues min et max

| Identificateur | Taille (octet) | Précision (chiffres) | Valeurs absolues min .. max |
|----------------|----------------|----------------------|---|
| float | 4 | 7 | 1.40299846 x 10 ⁻⁴⁵ .. 3.40282347 x 10 ³⁸ |
| double | 8 | 15 | 4.9406564584124654 x 10 ⁻³²⁴ .. 1.7976931348623160 x 10 ³⁰⁸ |

Valeurs données pour un codage en virgule flottante (norme IEEE 754) (cf. module M1101)

| Type | Taille (octet) | Nb valeurs distinctes | Norme |
|------|----------------|-----------------------|---|
| char | 1 | 256 | 128 codes ASCII 128 codes étendus Norme ASCII/ANSI (Latin1) |

Rem. ASCII/ANSI et Unicode sont deux normes pour coder les caractères (resp. sur 1 et 2 octets)

Unicode Codage de tous les systèmes d'écriture

Objectif Universalité de la Gestion Electronique des Documents (GED) et de l'édition de programmes
Inclut les 256 codes ASCII/ANSI (avec le 1^{er} octet nul)

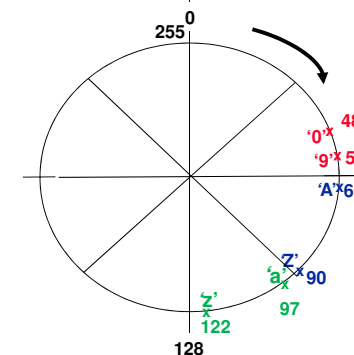
Environ 140 langues officielles usitées
Une cinquantaine d'écritures en usage
(alphabets, syllabaires, idéographiques, adjads)

Table ASCII – Caractères affichables (→ rang 127)

```

32->' ' 33->'!' 34->'"' 35->'#' 36->'$'
37->'%' 38->'&' 39->'(' 40->'(' 41->'('
42->'*' 43->'+' 44->'+' 45->'+' 46->'+'
47->'/' 48->'0' 49->'1' 50->'2' 51->'3'
52->'4' 53->'5' 54->'6' 55->'7' 56->'8'
57->'9' 58->':' 59->';' 60->'<' 61->'='
62->'>' 63->'?' 64->'@' 65->'A' 66->'B'
67->'C' 68->'D' 69->'E' 70->'F' 71->'G'
72->'H' 73->'I' 74->'J' 75->'K' 76->'L'
77->'M' 78->'N' 79->'O' 80->'P' 81->'Q'
82->'R' 83->'S' 84->'T' 85->'U' 86->'V'
87->'W' 88->'X' 89->'Y' 90->'Z' 91->'['
92->'\' 93->']' 94->'^' 95->'_' 96->'`'
97->'a' 98->'b' 99->'c' 100->'d' 101->'e'
102->'f' 103->'g' 104->'h' 105->'i' 106->'j'
107->'k' 108->'l' 109->'m' 110->'n' 111->'o'
112->'p' 113->'q' 114->'r' 115->'s' 116->'t'
117->'u' 118->'v' 119->'w' 120->'x' 121->'y'
122->'z' 123->'{' 124->'|' 125->'}' 126->'~'
127->'.'

```



Contiguïté de codage

- des chiffres de 0 à 9
- des majuscules de A à Z
- des minuscules de a à z

Moyen informatique de manipuler une donnée/information : associer la donnée à une variable

Une variable est une entité symbolique (nommée) représentant :

- (1) sur un plan conceptuel, une donnée
- (2) sur un plan informatique et géré par le compilateur, la réservation d'un emplacement libre en mémoire pour stocker la valeur de la variable
la place-mémoire réservée est fonction de son type (un ou plusieurs octets), elle est repérée par l'adresse du 1^{er} octet utilisé
- (3) la valeur de la variable est l'information codée et stockée à cet emplacement
- (4) l'affectation/assignation est l'opération (=) par laquelle une valeur est stockée/modifiée en mémoire

Le langage informatique comme tout autre langage comporte les aspects : lexical, syntaxique et sémantique.

La variable a le rôle sémantique de la donnée/information qu'elle représente.

Bonne Pratique

- Son rôle sémantique sera précisé par son identificateur
- Le choix de l'identificateur est important et nécessite un temps de réflexion

Bonne Pratique

L'identificateur doit être explicite et de longueur acceptable

Exemple de règle d'écriture des identificateurs :

Commencer par une minuscule, passage en majuscule en début de mot
nbClients, noCompteClient, tauxTVA, prixUnitaire

Les variables auxiliaires (calculs intermédiaires) : i, j, k, ..., tmp, aux

Déclaration de variable – Choix du type

(3/5)

Bonne Pratique

Choisir le typage d'une variable
le plus adapté à son domaine de variation

Importance

- Optimisation du stockage des données
- Définition des fonctions

Notion différée

Quel typage choisiriez-vous ?

pour l'âge des étudiants de la promotion ?

pour les températures enregistrées par une station météo ?

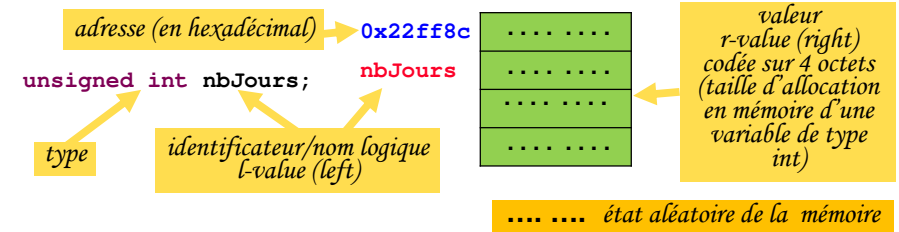
33

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty

Question

Déclaration de variable – Etat en mémoire

(4/5)



Les 4 attributs (caractéristiques) d'une variable

- identificateur (**nbJours**) représentant la donnée dans le source
- type définissant les modalités de représentation en mémoire (taille (4), codage (binaire) et les opérations applicables (+, -, *, /, %)
- adresse-mémoire (**0x22ff8c**) accessible par **&nbJours**
- sa valeur (**aléatoire tant que non affectée**)

Rem. En C, l'adresse mémoire d'une variable est accessible et la mémoire peut directement être lue et modifiée à partir de cette adresse

34

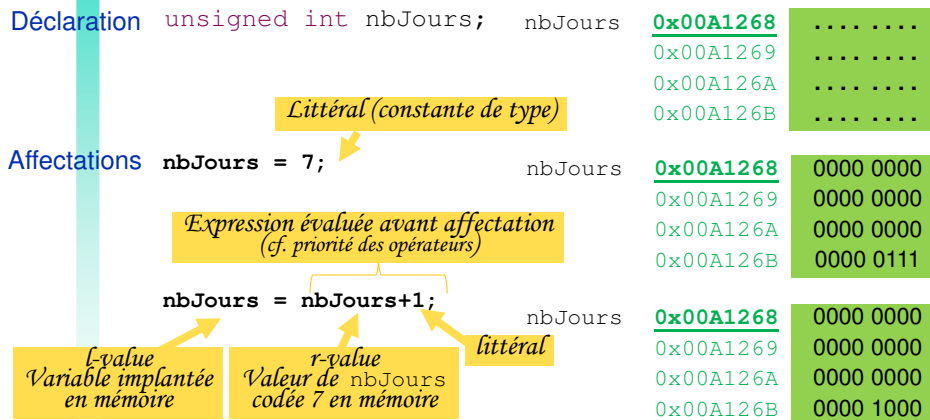
DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty

A savoir

Affectation de variable – Etat de la mémoire

(5/5)

<affectation> ::= <l-value> = <expression>
| <littéral> | <l-value>;



Littéraux et expressions sont des r-values
Ex: 0, 2.7, 'A', 'z', "Bonjour", 2*nbJours

Remarque : 2*Duree = nbJours; // **Erreur, Mess. Comp. : « non-lvalue in assignment »**

35

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty

A savoir

Exercice de base : simuler l'exécution de programme (1/2)

Extrait de code

```
1 char age;
2 float totalHT;
3 char birthday;
4 age=18;
5 birthday=age+1;
```

| Adresses | Mémoire |
|----------|--------------------------------|
| | |
| age | 0x00A1267 |
| totalHT | 0x00A1268 |
| | 0x00A1269 |
| | 0x00A126A |
| | 0x00A126B |
| | 0x00A126C |
| birthday | 0x00A126D |
| | 0x00A126E |
| | 0x00A126F |
| | 0x00A1270 |
| | 0x00A1271 |
| | |
| | état (aléatoire) de la mémoire |

36

DUT Informatique, 1^{ère} année – IAP – Cours Préliminaire – Marie-José Caraty

A savoir

Exercice de base : simuler l'exécution de programme (2/2)

Extrait de code

```

1 char age;
2 float totalHT;
3 char birthday;
4 age=18;
5 birthday=age+1;

```

Bonne Pratique

Initialiser toute variable avant son utilisation

- Warning signalé à la compilation si on utilise une variable non initialisée

| | Adresses | Mémoire | |
|----------|-----------|--------------------------------|---|
| age | 0x00A1267 | 0001 0010 | 4 |
| totalHT | 0x00A1268 | | |
| | 0x00A1269 | | |
| | 0x00A126A | | |
| | 0x00A126B | | |
| | 0x00A126C | | |
| birthday | 0x00A126D | 0001 0011 | 5 |
| | 0x00A126E | | |
| | 0x00A126F | | |
| | 0x00A1270 | | |
| | 0x00A1271 | | |
| | | état (aléatoire) de la mémoire | |

Etat final de la mémoire après exécution

Portée des déclarations

(1/2)

C'est la portion de code dans laquelle une variable est accessible/utilisable par son nom

Variables locales à un bloc

- Variables déclarées en début de bloc
- Elles ne sont accessibles qu'à l'intérieur de ce bloc

Variables globales

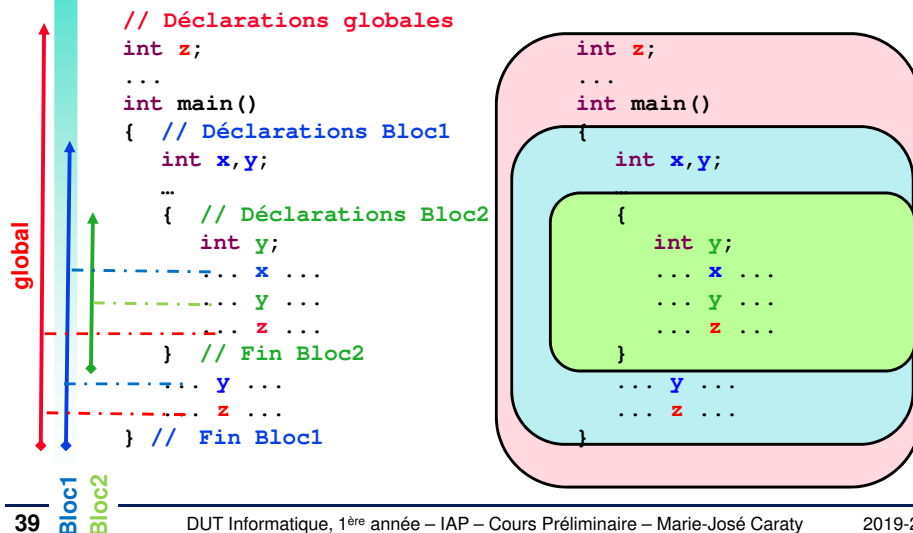
- Variables déclarées avant le `main()`
- Elles sont accessibles dans l'ensemble du programme

Bonne Pratique

Pour la sûreté du code (principe d'encapsulation), ne pas utiliser de variables globales

Visibilité et masquage des variables

(2/2)



Littéraux des types scalaires et flottants

(1/2)

Un littéral est constante

Entiers – Notations applicables aux variables de type entier

- Notation décimale
`i=0; i=-128; i=128; ...`
- Notation hexadécimale (0xH...)
Symboles hexadécimaux : {0..9, A, B, C, D, E, F} (minuscules autorisées)
`unsigned char nbMois=0xC; // unsigned char nbMois=12;`

Flottants – Notations applicables aux variables de type flottant

- Notation décimale
`x=39.276;`
`r=0.; s=.2; // r=0.0; s=0.2;`
- Notation exponentielle
`f = 24.3e-4; // f = 24.3*10-4 = 0.00243`
`g = 1.5123E3; // g = 1.5123*103 = 1512.3`

Exemples

```
char c, Q, quote, SautL;
c='b';
Q='?';
// saut de ligne
sautL='\n'
// caractère 'quote'
quote='\'';
// Utilisation de '\\' comme
// caractère d'échappement
```

| \ Code d'échappement | Description |
|----------------------|--------------------|
| \b | Retour arrière |
| \f | Saut de page |
| \n | Saut de ligne |
| \r | Retour chariot |
| \t | Tabulation |
| \' | Apostrophe (quote) |
| \" | Guillemet |
| \\ | Antislash |

Codes utiles à l'affichage

- Les caractéristiques de la programmation impérative
- Appréhender le manuel de référence d'un langage (BNF ou Conway)
- Les notions de base d'un langage impératif (Langage C)
- Point d'entrée d'une application d'un programme
- Unité de compilation
- Bloc d'instructions et flot d'exécution
- Type, variable
- Déclaration et affectation de variable
- Portée des déclarations
- Littéraux

Au prochain cours...

Les éléments d'un langage de programmation (suite)

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, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau

Cours n° 1

Les fondamentaux de la programmation impérative

Les éléments de langage de programmation

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

(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;
}
```

Commentaires

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 données 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
}
```

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;
}

```

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, caractère)
// sous la forme de l'égalité des variables et de leur valeur
printf("r=%f e2=%d c=%c", r, e*e, c);

```

chaîne de format
de sortieVariables/ 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

```

BNF de description de format

(2/5)

| | |
|--|---|
| <code>%[flag][prefixe][width][.precision][length]specifieur</code> | |
| flag | cadrage à droite par défaut, pour un cadrage à gauche (-) |
| préfixe | + 0 _ <code>_ espace</code> |
| 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__`

Entrée à partir du flot standard (clavier)

(3/5)

Bibliothèque `"stdio.h"` Norme C99Prototype `int scanf(const char* format, ...);`

format est la chaîne 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);
```

Chaîne de
format de
conversionadresses 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)

```

2. TRAITEMENTS – Entrées/Sorties

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

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

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

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)

2. TRAITEMENTS – Entrées/Sorties

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
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|

```

Rem : le caractère "\n" est comptabilisé

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

10

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

3. TRAITEMENT

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

11

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

3. TRAITEMENTS – Expressions

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

12

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

Classification des opérateurs

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)

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

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 |

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

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

- 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
- Fondée sur l'associativité de l'opérateur : de gauche (G) à droite (D) ou de D à G
Associativité gauche : $a \heartsuit b \heartsuit c = (a \heartsuit b) \heartsuit c$
Associativité droite : $a \heartsuit b \heartsuit c = a \heartsuit (b \heartsuit c)$
- Règle d'évaluation en cas d'égalité de priorité : de gauche à droite
 $a \heartsuit b \clubsuit c$ // si \heartsuit et \clubsuit sont deux opérateurs de même priorité
 $(a \heartsuit b) \clubsuit c$ est évalué // ordre d'évaluation de gauche à droite : 1) \heartsuit et 2) \clubsuit
- Pour un opérateur (binaire) donné, l'ordre d'évaluation est (généralement) celui du premier opérande puis du deuxième opérande

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 appries 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

17

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

Opérateurs

≡ est équivalent à

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

Incrémementation préfixe : ++i

++i ≡ i+=1 ≡ i=i+1

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

Incrémementation 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

| 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-incrémementation |
| -- | x-- ou --x | post ou pré-décrémementation |
| (type) | (type)a | 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 conditionnelle |
| = += *= ... | a=b | affectations |
| , | x1, x2, ..., xn | évaluation séquentielle |

18

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

Associativité des opérateurs

| | Associativité |
|-----------------------------------|-----------------|
| () [] -> . | gauche à droite |
| Opérateurs unaires | droite à gauche |
| ! ~ - + * | |
| & ++ -- (type) sizeof | |
| * / \% | 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 |

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 {+, -, *, /, %}

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

19

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

Priorité/précédence des opérateurs

| | Associativité |
|-----------------------------------|-----------------|
| 15 () [] -> . | gauche à droite |
| Opérateurs unaires | droite à gauche |
| 14 ! ~ - + * | |
| & ++ -- (type) sizeof | |
| 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 |
| Affectations simples et composées | droite à gauche |
| 02 = += *= ... | |
| 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

20

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

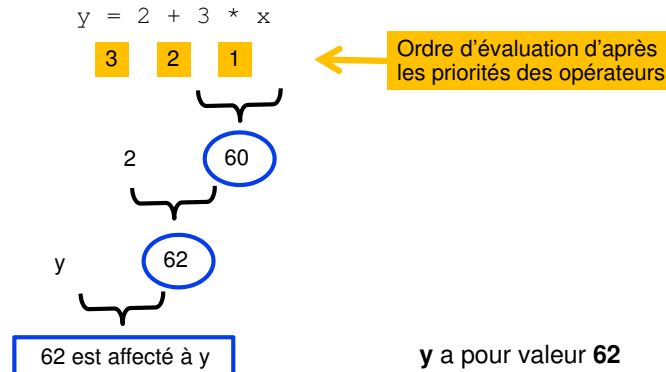
3. TRAITEMENTS – Expressions

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



21

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

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 $b=2$

22

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

4. STRUCTURES DE CONTRÔLE

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

```

/* @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\n");
    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

23

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

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 { ... }

24

DUT Informatique, 1^{ère} année – IAP – Cours 1 – Marie-José Caraty

2019-2020

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 :

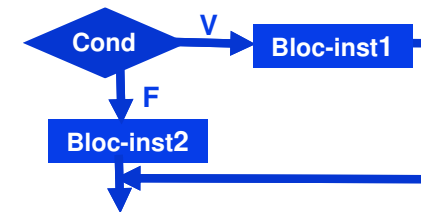
L'alternative else est optionnelle

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



Contrôle d'exécution

```

inst1;
if (condition) {
    instV1;
    instV2;
}
else {
    instF1;
    instF2;
    instF3;
}
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 (séquençement des instructions à l'exécution)

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;

Branchement conditionnel – Exemple sans alternative (3/7)

```

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

```

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

```

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

// 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;
}
  
```

Branchement conditionnel – Exemple avec alternative (4/7)

```

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

```

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
  
```

// 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;
}
  
```

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 blocs if et else

```

/* 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;
}

```

Evitez le warning de Microsoft
(scanf_s : non standard)

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

Style de codage

| | | |
|--|--|--|
| <pre> if (condition) { inst1; inst2; } else { inst3; } inst4; </pre> | <pre> if (condition) { inst1; inst2; } else { inst3; } inst4; </pre> | <pre> if (condition) { inst1; inst2; } else inst3; inst4; </pre> |
|--|--|--|

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

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

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;

```

Attention :

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

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;

```

```

/* 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-être ?

```

CONCLUSION

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

IAP - Introduction à l'Algorithmique et à la Programmation

Equipe pédagogique

Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau

T04 – Boucle `for`
T11 – Boucle `while`
T14 – Boucle `do ... while`
T18 – Choix des boucles
T19 – Ruptures de séquence
T22 – Type énuméré
T23 – Tableau statique
T25 – Chaîne de caractères
T28 – Type pointeur

Cours n° 2

Les fondamentaux de la programmation impérative

Les structures de contrôle (suite)

Sommaire

1. Structure de programme

2. Données

- Tableaux statiques
- Chaînes de caractères
- Type structuré/utilisateur

3. Traitements

- Structures de contrôle
Les instructions itératives



3. TRAITEMENTS – Structures de contrôle

Rôles des itérations/boucles

Les instructions de contrôle de type itératif permettent d'écrire un programme qui exécute plusieurs fois le même bloc d'instructions

Elles présentent le risque (comme d'autres structures de contrôle*) d'écrire un programme dont le temps d'exécution peut poser un problème au sens d'un temps trop long, voire « infini »

Trois instructions de type itératif sont définies en C

- à itérations bornées (boucle `for`), le nombre d'itérations est connu
- à itérations non bornées (boucles `while ...` et `do ... while`), le nombre d'itérations n'est pas connu

Notion différée* – Fonctions et récursivité

3. TRAITEMENTS – Structures de contrôle

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

for ([<for_init>] ; [<expr1>] ; [<expr2>]) <bloc_inst>

| | |
|-------------|---|
| <for_init> | les instructions (d'initialisation) exécutées avant l'exécution de la boucle |
| <expr1> | expression (condition), si l'expression est vraie, la boucle exécute son corps (bloc_inst) sinon elle s'arrête |
| <expr2> | les instructions exécutées à la fin de chaque itération (après exécution de bloc_inst) |
| <bloc_inst> | le bloc des instructions exécutées à chaque itération |

Remarque La BNF montre toute l'élaboration de la boucle `for`

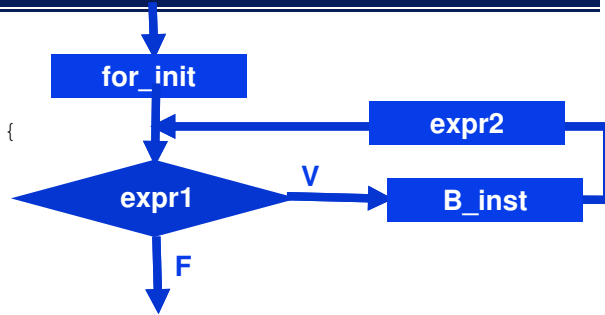
Usuellement la boucle `for` permet un traitement itératif contrôlé par un mécanisme de variable d'itération (compteur) et une condition d'arrêt

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
inst2;

```

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

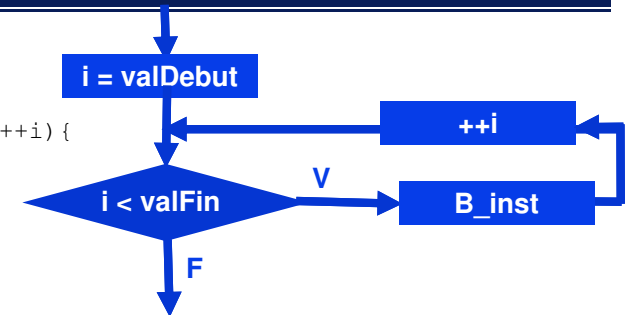
`for (; ;); // est ainsi une boucle infinie`

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

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

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;
}

```



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

Moyenne des 30 premiers entiers positifs 15.500000

Une instruction de boucle peut être une boucle

Exemple :

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

Affichage attendu ?

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,
pour i=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;
}
```

```
OK
OOK
OOOK
OOOOOK
OOOOOOK
OOOOOOOK
OOOOOOOOK
OOOOOOOOK
OOOOOOOOK
OOOOOOOOK
OOOOOOOOK
```

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

```
while (<expr>) <bloc_inst>
```

Pseudo-code

```
tantQue (condition) faire
    bloc_inst
finFaire
```

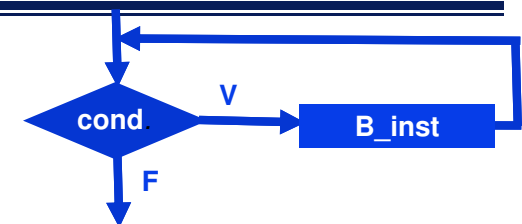
Remarque :

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

L'exécution du bloc d'instruction est contrôlée par la condition
si la condition est vraie, bloc_inst est exécuté
sinon le bloc n'est pas exécuté

Contrôle d'exécution

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



Flux d'exécution

```
inst1;
1. si (condition est vraie)
    // 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

3. TRAITEMENTS – Structures de contrôle

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

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

```
int main() {
    int i, som;
    som=0; 1
    i=1;
    while (som <= 1000) {
        som+=i; // eq. som=som+i; 2
        i=i+1; 3
    }
    printf("L'entier 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 $1+2+\dots+n = n.(n+1)/2$
Vérification :
 $(n-1).n/2 \leq 1000$ et $n.(n+1)/2 > 1000$
 $(44*45)/2=990$ et $(45*46)/2=1035$

L'entier n tel que $1+2+\dots+n > 1000$ est : 45

3. TRAITEMENTS – Structures de contrôle

Boucle do – Itérations non bornées

(1/3)

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

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'instructions est au moins exécuté une fois sans contrôle
les autres itérations sont contrôlées par la condition

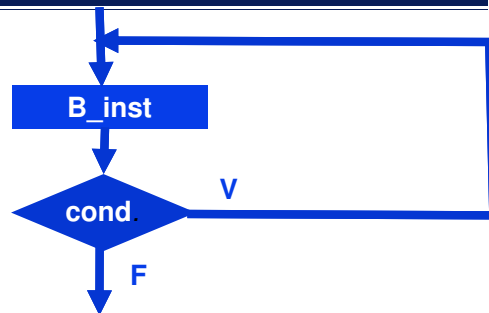
3. TRAITEMENTS – Structures de contrôle

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 (modifier la condition) 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;
```

3. TRAITEMENTS – Structures de contrôle

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; 1
    nbAnnees=0;
    do {
        capital=capital+1000.; 2
        capital=(1+0.57)*capital;
        nbAnnees++;
    } while (capital<1000000.); 3
    printf("Millionnaire dans %d ans!!!\n", nbAnnees);
    system("pause"); return 0;
}
```

1, 2, 3 : arrêt de boucle vérifié

Test ?

Reste à trouver l'investissement...

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>
```

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

Rem : Utilisation usuelle pour la manipulation des tableaux

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

(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)

Ruptures de séquence

(2/2)

Exemples d'utilisation

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

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

Code spaghetti ?

Utile dans des algorithmes de recherche et de traitement spécifiques

Retour sur les types de données

les types énumérés

les tableaux statiques et leur traitement

les chaînes de caractères

les pointeurs

21

DUT Informatique, 1^{ère} année – IAP – Cours 2 – Marie-José Caraty

2019-2020

3. RETOUR SUR LES TYPES – type natif tableau

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 : utile à la déclaration des tableaux

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

22

DUT Informatique, 1^{ère} année – IAP – Cours 2 – Marie-José Caraty

2019-2020

3. RETOUR SUR LES TYPES – Données

Tableaux statiques – Déclaration

(1/2)

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 dans notre version du C
```

On utilisera une constante 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 4^{ème} élément par `tab[3]`

Déclaration possible de constantes symboliques (traitées par le préprocesseur)

```
#define N 5 // à déclarer avant le main
```

23

DUT Informatique, 1^{ère} année – IAP – Cours 2 – Marie-José Caraty

2019-2020

3. RETOUR SUR LES TYPES – Données

Tableaux statiques – Exemple

(2/2)

```
/* @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 initialisé 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

24

DUT Informatique, 1^{ère} année – IAP – Cours 2 – Marie-José Caraty

2019-2020

Chaînes de caractères – Littéral

(1/3)

Une chaîne de caractères est stockée dans un tableau de caractères se terminant par le caractère délimiteur '\0'

```
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)

Attention

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

25

DUT Informatique, 1^{ère} année – IAP – Cours 2 – Marie-José Caraty

2019-2020

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

Déclaration d'une chaîne de caractères initialisée par un littéral chaîne

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

A savoir

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 sizeof(s1) vaut 80
 strlen(s2) vaut 11 sizeof(s2) vaut 12 ('0' inclus)

26

DUT Informatique, 1^{ère} année – IAP – Cours 2 – Marie-José Caraty

2019-2020

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!";

    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)=6
 sizeof(s2)=7

strlen() taille « utile » d'une chaîne (sans le délimiteur)
 sizeof() taille mémoire occupée par une variable ou un type
 (pour les chaînes de caractères, délimiteur '\0' inclus)

27

DUT Informatique, 1^{ère} année – IAP – Cours 2 – Marie-José Caraty

2019-2020

Type pointeur – Opérations élémentaires

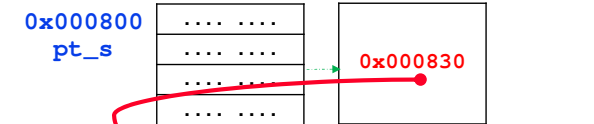
Exemple pour une adresse mémoire sur 4 octets (processeur 32 bits) et un short codé sur 2 octets

Tout pointeur

- (1) est une variable destinée à contenir une adresse mémoire (4 octets)
- (2) est associé à un type d'objet

Déclaration du pointeur

L1 short* pt_s;



L2 short s;

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 ?

28

DUT Informatique, 1^{ère} année – IAP – Cours 2 – Marie-José Caraty

Retour aux tableaux

cf. Vérification en 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 tab[3];
```

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

| Adresses | Mémoire |
|-----------|---------|
| | |
| 0x00A126C | |
| 0x00A126D | |
| 0x00A126E | |
| 0x00A126F | |
| 0x00A1270 | |
| 0x00A1271 | |
| 0x00A1272 | |
| 0x00A1273 | |
| 0x00A1274 | |
| | |

`tab = &tab = &tab[0]`

`tab` est l'adresse du tableau implanté en mémoire (`&tab`)

`tab` est également l'adresse du 1^{er} élément du tableau `&tab[0]`

`tab+1` est l'adresse du 2^{ème} élément du tableau `&tab[1]`

`tab+2` est l'adresse du 3^{ème} élément du tableau `&tab[2]` (arithmétique des pointeurs)

.... état (aléatoire) de la mémoire

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
- La déclaration, représentation mémoire et manipulation des tableaux statiques
- La déclaration, représentation mémoire et manipulation des chaînes de caractères
- Le type pointeur

Au prochain cours...

La suite des traitements en programmation : les fonctions

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, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau

Cours n° 3

Les fondamentaux de la programmation impérative

Les fonctions (1/2)

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 – 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 = 0\n");
    printf("Entrez les valeurs de b et c\n");
    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;
}
```

Fonction

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/ prototypage

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

corps

```
<bloc_instructions>
```

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 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

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

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 `sqrt(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;
}
```

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

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

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 dans le 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 qui lui correspond

Au **retour de la fonction**, la valeur de retour 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

Exemples d'exécution de programme
avec appel de fonction

fonction sans retour

fonction avec retour

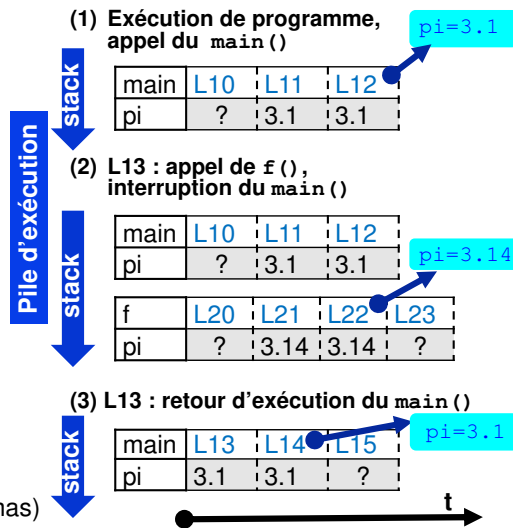
Exécution de programme – Fonction sans paramètre et sans retour

```

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;
}

```



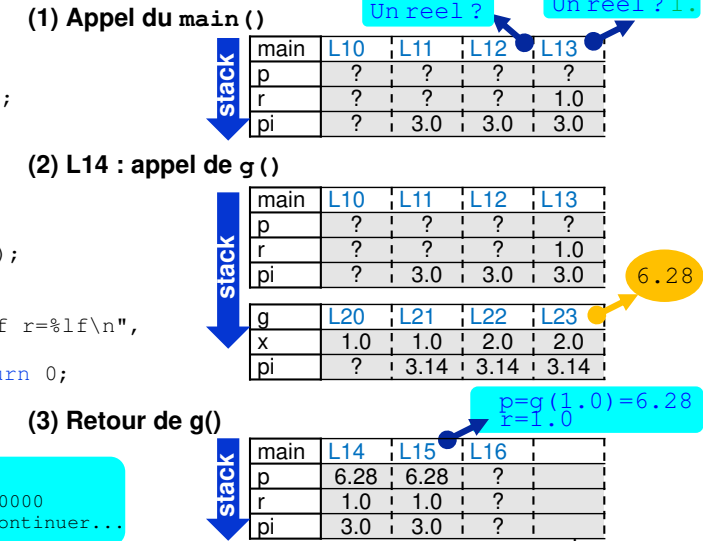
Exécution de programme – Fonction avec paramètre et avec retour

```

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",
L16           r, p, r);
L16     system("pause"); return 0;
}

```



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

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

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)

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 10^{ème} point de la courbe

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

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;
```

Revenons à notre problème

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

Objectif : prototyper cette fonction (1^{ère} étape)
analyse de l'interface de la fonction

Prototypage – Les bonnes questions à se poser

Analyse de l'interface de la fonction

1. Quel nom pour la fonction ?

3. Quel retour ? et de quel type ?

2. De quelles données a-t-on besoin pour résoudre le problème ?



Analyse de l'interface d'une fonction

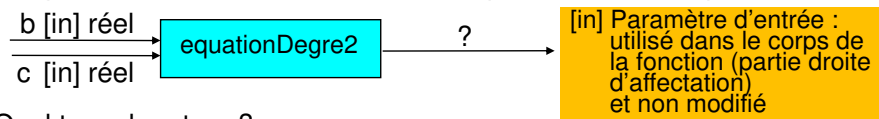
(1/2)

Schéma de l'interface

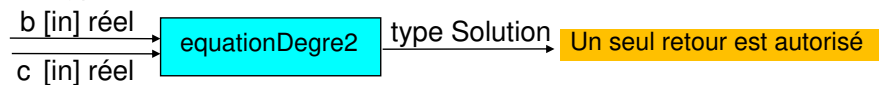
(1) Quel nom donner à la fonction ?



(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);

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 1^{ère} racine dans les cas 1 et 2`r2` stocke la 2^{ème} racine dans le cas 2

(2) Un tableau de 2 réels

`solution[0]` stocke `r1`, `solution[1]` stocke `r2`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);
```

Corps de la fonction – Algorithme de résolution

```

/* @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;
}

```

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

Faut-il documenter une fonction ?

Oui/Non

Si oui, pourquoi ?

sinon, pourquoi ?

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

5. FONCTIONS – Codage de la fonction

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

```
/* 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 2eme 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;
}
```

5. FONCTIONS – Codage de la fonction

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;
}
```

6. FONCTIONS – Structuration de programme

Structuration du code

Où mettre la fonction ?

Où déclarer le type structuré?

La notion de bloc

Déclaration au niveau global

6. FONCTIONS – Structuration de programme

Structuration du code – Déclarations (types énuméré/structuré) (1/3)

```
/* @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 racines 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)

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;
}

```

33

DUT Informatique, 1^{ère} année – IAP – Cours 3 – Marie-José Caraty

2019-2020

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 polynôme
 */
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

34

DUT Informatique, 1^{ère} année – IAP – Cours 3 – Marie-José Caraty

2019-2020

Autre exemple de structuration de code

Utilisation de plusieurs fonctions

35

DUT Informatique, 1^{ère} année – IAP – Cours 3 – Marie-José Caraty

2019-2020

Structuration du code – Utilisation de plusieurs fonctions

```

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

```

```

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

```

```

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

```

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

```

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;
}

```

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

```

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

```

36

DUT Informatique, 1^{ère} année – IAP – Cours 3 – Marie-José Caraty

2019-2020

Structuration du code – Utilisation de plusieurs fonctions

```

/* @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;
}

```

```

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

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

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, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau

Cours n° 4

Les fondamentaux de la programmation impérative

Les fonctions (2/2)

1. FONCTIONS – Généralités

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

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

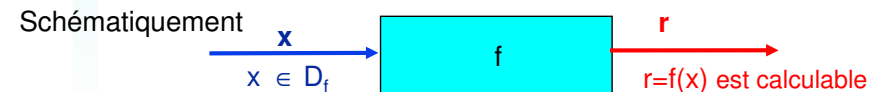
est l'ensemble des éléments x de R tel que $f(x)$ existe (**calculable**)

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

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

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

Toute étude/analyse de la fonction se fera dans D_f



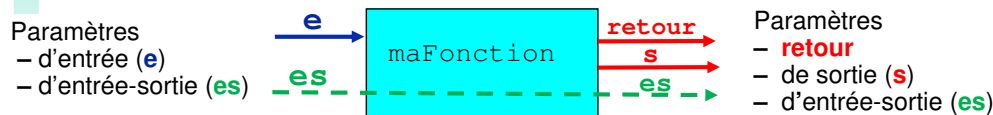
1. FONCTIONS – Généralités

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

1. FONCTIONS – Généralités

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

1. FONCTIONS – Généralités

Documentation – Fonction pow () (extrait)

(1/2)

NAME

pow, powf, powl - power functions

SYNOPSIS

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

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

DESCRIPTION

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

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**,

Le nom de la fonction

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

Le rôle de la fonction

Documentation du retour de la fonction

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)

5

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

1. FONCTIONS – Généralités

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.

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

overflow

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

6

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

1. FONCTIONS – Généralités

Utilisation/appel de fonction – Fonction pow ()

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

Prototype

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

identificateur de la fonction

Paramètres formels

type de la valeur retournée par la fonction

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)

7

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

1. FONCTIONS – Généralités

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; // mémorise 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)
```

8

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

1. FONCTIONS – Généralités

Exemple – Utilisation de la fonction pow()

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

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. Coursla T20

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;
}
```

9

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

1. FONCTIONS – Généralités

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

10

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

1. FONCTIONS – Généralités

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()

11

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

2. POINTEURS ET UTILISATION

Rappel – Type pointeur

Un pointeur

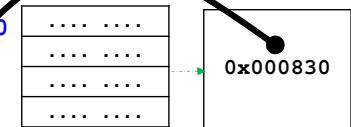
(1) est une variable destinée à contenir une adresse mémoire

(2) est associé à un type d'objet

Déclaration du pointeur

short* ptShort; (t0)

0x000800
ptShort



short s; (t0)

0x000830
s
*ptShort

0000 0000
0000 1100

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)



cf. Transp. 23 Cours 2

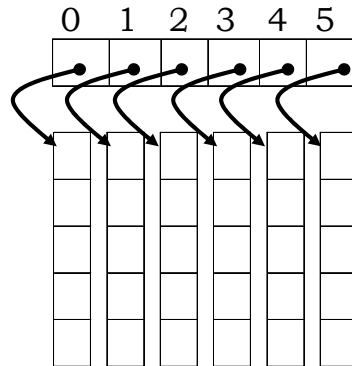
12

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

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

Enrichissement des structures de données pour la modélisation



13

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

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

Principe

Utiliser un paramètre formel (pFormel) de type pointeur pour modifier le paramètre effectif (pEffectif) dans l'appelant

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

14

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

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

15

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

Contexte d'exécution d'une fonction

(1/2)

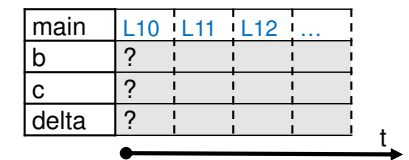
Toute fonction a son propre contexte d'exécution :
une zone mémoire allouée par le système (**pile d'exécution**)
que le flux d'exécution de la fonction va modifier
(par les entrées, les affectations et autres appels de fonction)

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;
}
```



16

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

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

17

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

3. FONCTIONS – Paramètres de sortie

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;
}
```

| | | | | |
|------|-----|-----|--|--|
| 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);
}
```

| | | | | |
|------|-----|-----|-----|-----|
| main | L10 | L11 | L12 | |
| val | ? | 3 | 13 | |
| f | L20 | L21 | L22 | L23 |
| *x | @20 | @20 | @20 | @20 |
| y | ? | 10 | | |

(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

| | | | | |
|------|-----|-----|--|--|
| main | L13 | L14 | | |
| val | 13 | ? | | |

18

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

3. FONCTIONS – Paramètres de sortie

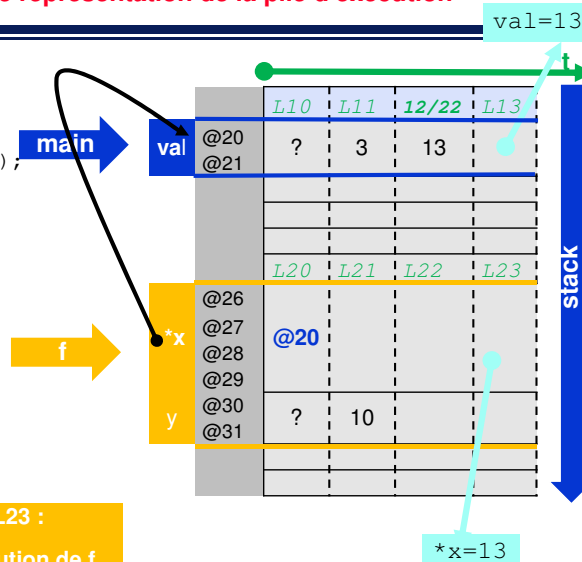
Techniquement – Autre représentation de la pile d'exécution

? état de la 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);
}
```

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



19

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

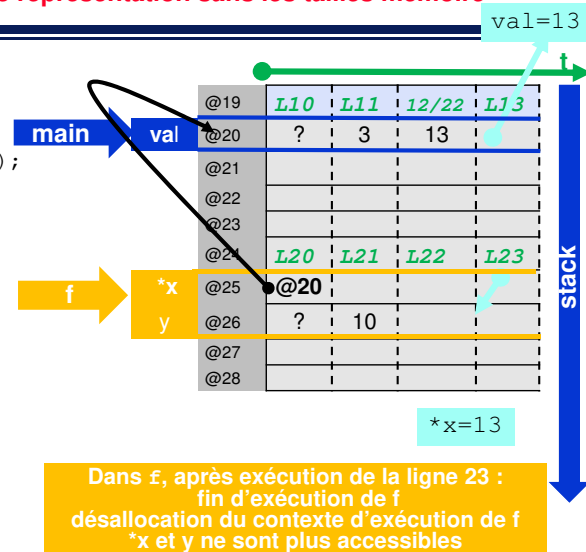
3. FONCTIONS – Paramètres de sortie

Techniquement – Autre représentation sans les tailles mémoire

? état de la 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);
}
```



Dans f, après exécution de la ligne 23 :
fin d'exécution de f,
désallocation du contexte d'exécution de f
*x et y ne sont plus accessibles

20

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

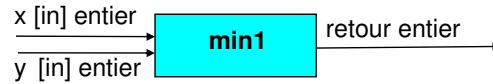
Choix du prototypage

Plusieurs prototypes pour une même fonctionnalité
Exemple : le minimum de deux entiers

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

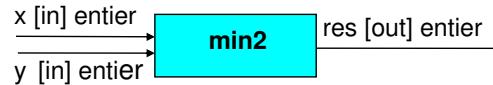
Premier prototypage avec retour

```
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;
}
```

Voire avis ?

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;
}
```

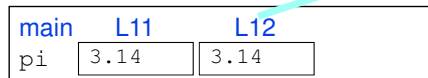
```
int main() {
    float pi = 3.14;
    printf("pi = %f\n", pi);
    saisie(&pi);
    printf("pi = %f\n", pi);
    system("pause"); return 0;
}
```

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

Cas d'une saisie (2/2)

1. Exécution du programme (main)

pi = 3.140000



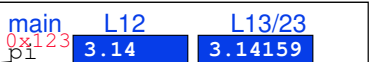
2. Interruption d'exécution du main

(L13) - appel de saisie

entrez un nombre :

entrez un nombre : 3.14159

Reg.0



*px = 3.140000

```
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;
}
```

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)

3. FONCTIONS – Paramètres de sortie

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
```

4. FONCTIONS – Paramètres de type Tableau

Fonctions et tableaux

Cas de passage d'un tableau
dans une fonction

4. FONCTIONS – Paramètres de type Tableau

Tableaux statiques – Déclaration

RAPPEL

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 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

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

| | | | | | |
|-----|---|---|---|---|---|
| tab | ? | ? | ? | ? | ? |
| | 0 | 1 | 2 | 3 | 4 |

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

2. DONNEES - Variables

Schématiquement en mémoire

RAPPEL

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 (1^{er} 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 | | tab[0] |
| tab+1 | | tab[1] |
| tab+2 | | tab[2] |
| 0x00A126E | | |
| 0x00A126F | | |
| 0x00A1270 | | |
| 0x00A1271 | | |
| 0x00A1272 | | |
| 0x00A126B | | |
| 0x00A1274 | | |

.... état (aléatoire) de la mémoire

3. RETOUR SUR LES TYPES – Données

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 modificateur
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
```

29

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

4. FONCTIONS – Paramètres de type Tableau

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;
```

30

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

4. FONCTIONS – Paramètres de type Tableau

Fonctions et chaînes de caractères

Passage d'une chaîne de caractères
dans une fonction

31

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

3. RETOUR SUR LES TYPES – type natif tableau

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')

32

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

3. RETOUR SUR LES TYPES – type natif tableau

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

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!";
```

A savoir

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      sizeof(s1) vaut 80
strlen(s2) vaut 11     sizeof(s2) vaut 12
                      sizeof(int) vaut 4
```

33

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

3. RETOUR SUR LES TYPES – type natif tableau

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
```

34

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

4. FONCTIONS – Paramètres de type Tableau

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);`

35

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

CONCLUSION

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...

Les algorithmes de recherche

36

DUT Informatique, 1^{ère} année – IAP – Cours 4 – Marie-José Caraty

2019-2020

IAP - Introduction à l'Algorithmique et à la Programmation

Équipe pédagogique

Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau

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° 5 Fondamentaux en algorithmique

Algorithmes de recherche

1. GENERALITES

Algorithmique

Algorithme (Al-Khawârizmî 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

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)
```

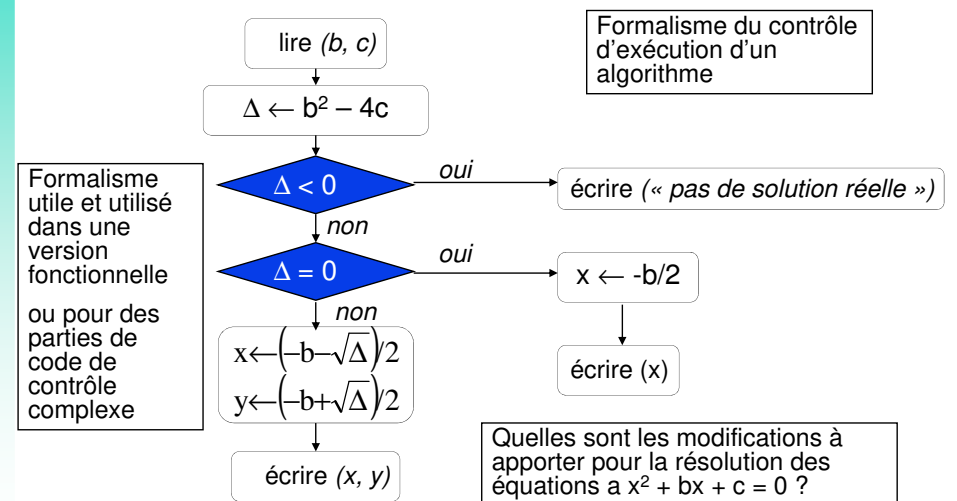
Algorithme d'Archimède (-287, -212) – Approximation du nombre π

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

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

2. EXPRESSION DES ALGORITHMES

Organigramme (Extrait du Cours 1)



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

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 $f(n) = n^2$

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

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

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 – 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 1^{er} 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
  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

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

Recherche linéaire – (2/3)
Exemple sur 10 éléments

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 |

```

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
    finPour
  finFaire
retourner faux
  
```

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

4. ALGORITHME DE RECHERCHE

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

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 |

Meilleur des cas recherche de **2** \Rightarrow **1 comparaison**

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

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

Recherche linéaire dans un tableau de taille n

Dans le pire des cas, il y aura $2 \cdot n$ comparaisons

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

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

4. ALGORITHME DE RECHERCHE

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

4. ALGORITHME DE RECHERCHE

Recherche dichotomique – Algorithme

Un 2^{ème} 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
    
```

4. ALGORITHME DE RECHERCHE

Recherche dichotomique – (1/6) Exemple sur 10 éléments

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 |

```

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
    
```

Meilleur des cas recherche de **15** \Rightarrow **1 comparaison**

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

| début | fin | milieu |
|-------|-----|--------|
| 0 | 9 | 4 |

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

↑
début
↑
milieu
↑
fin

4. ALGORITHME DE RECHERCHE

Recherche dichotomique – (2/6) Exemple sur 10 éléments

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 | | | | milieu | | | | | fin |

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 | milieu | | | | fin |

```
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
finFaire
retourner faux
```

| debut | fin | milieu |
|-------|-----|--------|
| 0 | 9 | 4 |
| 5 | | |

| debut | fin | milieu |
|-------|-----|--------|
| 5 | 9 | 7 |
| 8 | | |

4. ALGORITHME DE RECHERCHE

Recherche dichotomique – (3/6) Exemple sur 10 éléments

Itération 3

```
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 | milieu | fin |

Itération 4

```
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 | milieu | fin |

```
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
finFaire
retourner faux
```

| debut | fin | milieu |
|-------|-----|--------|
| 8 | 9 | 8 |
| 9 | | |

| debut | fin | milieu |
|-------|-----|--------|
| 9 | 9 | 9 |
| 10 | | |

4. ALGORITHME DE RECHERCHE

Recherche dichotomique – (4/6) Exemple sur 10 éléments

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 | début |

```
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
finFaire
retourner faux
```

| debut | fin | milieu |
|-------|-----|--------|
| 10 | 9 | 9 |
| | | |

Pire des cas recherche de 80

⇒ **8 comparaisons**

4 itérations avec 2 comparaisons par itération

4. ALGORITHME DE RECHERCHE

Recherche dichotomique – (5/6) Exemple sur 10 éléments

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

```
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
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 comparaisons pour la recherche d'un élément

Calcul du nombre de comparaisons pour la recherche d'un élément

(1/3)

```
#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;
```

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;
}
```

Calcul du nombre de comparaisons pour la recherche d'un élément

(3/3)

```
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.trouve);
    /*
    system("pause"); return 0;
    */
}
```

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 |
|---|---|----|----|----|----|----|----|----|----|
| 2 | 7 | 10 | 12 | 15 | 19 | 34 | 56 | 78 | 79 |

05 03 05 07 01 05 07 03 05 07

Nb de comparaisons

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

5. TEMPS D'EXECUTION

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

| | | Complexité | | | | | | |
|----------------------------|--------|------------|------------|--------|--------------------|-------|-------|-------------------|
| | | 1 | $\log_2 n$ | n | $n \cdot \log_2 n$ | n^2 | n^3 | 2^n |
| Taille du problème (n) | 10^2 | 1 μ s | 7 μ s | 0,1 ms | 0,7 ms | 10 ms | 1 s | $4 \cdot 10^7$ Ga |
| | 10^3 | 1 μ s | 10 μ s | 1 ms | 10 ms | 1 s | 17 mn | ∞ |
| | 10^4 | 1 μ s | 13 μ s | 10 ms | 130 ms | 1,7mn | 12j | ∞ |
| | 10^5 | 1 μ s | 17 μ s | 0,1 s | 1,7 s | 2,8h | 32 a | ∞ |
| | 10^6 | 1 μ s | 20 μ s | 1 s | 20 s | 12j | 32 Ka | ∞ |

5. TEMPS D'EXECUTION

Complexité et problème/algorithm type

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

5. TEMPS D'EXECUTION

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

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

5. TEMPS D'EXECUTION

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 1^{er} 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

5. TEMPS D'EXECUTION

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;
}
```

5. TEMPS D'EXECUTION

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;
}
```

5. TEMPS D'EXECUTION

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;
}
```

5. TEMPS D'EXECUTION

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));
}
```

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);

```

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;
}

```

CONCLUSION

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 la complexité
- La problématique de la comparaison des algorithmes sur la base du temps d'exécution

Au prochain cours...

La suite des fonctions



IAP - Introduction à l'Algorithmique et à la Programmation

Equipe pédagogique

*Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau*

Cours n°6

Projet IAP19-20
A venir...

http://www.tutorialspoint.com/c_standard_library/

IAP - Introduction à l'Algorithmique et à la Programmation

Equipe pédagogique

Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Denis Jeanneau

Cours n° 7

Les fondamentaux de la programmation impérative

Révisions

1. PROJET

Rappel des deadlines du projet

Date des recettes
la semaine du 21 octobre
en séance de IAP3 (2^{ème} TP de IAP)

Date limite de dépôt
(1) du Dossier de Développement Logiciel
au secrétariat le lundi 4 novembre

(2) de votre Archive Projet (format zip obligatoire)
dans le puits de votre groupe sur COMMUN/IAP1920
En cas de binômes de groupes différents, déposez votre archive
dans le puits du groupe du premier membre du binôme inscrit
sur le Dossier de Développement Logiciel)
Dossier de Développement Logiciel (version électronique, format pdf)
incluant le source du Sprint de plus haut niveau validé à la recette

1. REVISIONS

Le domaine des révisions

Rappel sur le type pointeur

Les fonctions

Interface et déclaration/prototypage

Les différents modes de passage de paramètre

Le domaine de définition de la fonction

Le corps d'une fonction

L'appel des fonctions

La spécificité du passage d'un paramètre tableau

1. REVISIONS

Les pointeurs – Déclaration, initialisation, indirection

L10 Réservez mémoire
des paramètres formels de la fonction (main)
et des variables locales à la fonction

```
→ L10 int main() {
L11 char a, b;
L12 char* p1, p2;
L13 a=88, b=12;
L14 p1=&a;
L15 p2=&b;
L16 printf("p1=%x *p1=%d\n ", p1, *p1);
L17 printf("p2=%x *p2=%d\n ", p2, *p2);
L18 *p1+=*p2+10;
L19 *p2=*p1*5;
L20 printf(" *p1=%d *p2=%d\n ",
L21 *p1, *p2);
L21 system("pause");
L22 return 0;
}
```

Adresses Mémoire

| | | |
|-----------|-----------|-------|
| | 0x00A126B | |
| a | 0x00A126C | |
| b | 0x00A126D | |
| p1 | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| p2 | 0x00A1272 | |
| | 0x00A1273 | |
| | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| | 0x00A1272 | |

1. REVISIONS

Les pointeurs – Déclaration, initialisation, indirection

L13 Initialisation des variables a et b

```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
L21         *p1, *p2);
L21   system("pause");
L22   return 0;
}
```

| | Adresses | Mémoire |
|----|-----------|---------|
| | 0x00A126C | |
| a | 0x00A126C | 88 |
| b | 0x00A126D | 12 |
| p1 | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| p2 | 0x00A1272 | |
| | 0x00A1273 | |
| | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| | 0x00A1272 | |

5

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

1. REVISIONS

Les pointeurs – Déclaration, initialisation, indirection

L14 Initialisation du pointeur p1

```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
L21         *p1, *p2);
L21   system("pause");
L22   return 0;
}
```

Déréférencement
2 noms pour une même zone mémoire

Référencement

| | Adresses | Mémoire |
|----|-----------|---------|
| | 0x00A126C | |
| | 0x00A126C | 88 |
| b | 0x00A126D | 12 |
| p1 | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| | 0x00A1272 | |
| | 0x00A1273 | |
| | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| | 0x00A1272 | |

6

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

1. REVISIONS

Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables a et b

```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
L21         *p1, *p2);
L21   system("pause");
L22   return 0;
}
```

| | Adresses | Mémoire |
|----|-----------|---------|
| | 0x00A126C | |
| | 0x00A126C | 88 |
| | 0x00A126D | 12 |
| p1 | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| | 0x00A1272 | |
| | 0x00A1273 | |
| | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| | 0x00A1272 | |

7

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

1. REVISIONS

Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables a et b

```
L10 int main() {
L11   char a, b;
L12   char* p1, p2;
L13   a=88, b=12;
L14   p1=&a;
L15   p2=&b;
L16   printf("p1=%x *p1=%d\n ", p1, *p1);
L17   printf("p2=%x *p2=%d\n ", p2, *p2);
L18   *p1+=*p2+10;
L19   *p2=*p1*5;
L20   printf("*p1=%d *p2=%d\n ",
L21         *p1, *p2);
L21   system("pause");
L22   return 0;
}
```

| | Adresses | Mémoire |
|----|-----------|---------|
| | 0x00A126C | |
| | 0x00A126C | 88 |
| | 0x00A126D | 12 |
| p1 | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| | 0x00A1272 | |
| | 0x00A1273 | |
| | 0x00A126E | |
| | 0x00A126F | |
| | 0x00A1270 | |
| | 0x00A1271 | |
| | 0x00A1272 | |

L16 p1=0x00A126C *p1=88
L17 p2=0x00A126D *p2=12

8

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

1. REVISIONS

Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables a et b

```
L10 int main() {
L11 char a, b;
L12 char* p1, p2;
L13 a=88, b=12;
L14 p1=&a;
L15 p2=&b;
L16 printf("p1=%x *p1=%d\n ", p1, *p1);
L17 printf("p2=%x *p2=%d\n ", p2, *p2);
L18 *p1+=*p2+10;
L19 *p2=*p1*5;
L20 printf("p1=%d *p2=%d\n ",
        *p1, *p2);
L21 system("pause");
L22 return 0;
}
```

| Adresses | Mémoire |
|-----------|---------|
| 0x00A126C | |
| 0x00A126C | 88 110 |
| 0x00A126D | 12 |
| 0x00A126E | |
| 0x00A126F | |
| 0x00A1270 | |
| 0x00A1271 | |
| 0x00A1272 | |
| 0x00A1273 | |
| 0x00A126E | |
| 0x00A126F | |
| 0x00A1270 | |
| 0x00A1271 | |
| 0x00A1272 | |

9

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

1. REVISIONS

Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables a et b

```
L10 int main() {
L11 int a, b;
L12 int *p1, *p2;
L13 a=88, b=12;
L14 p1=&a;
L15 p2=&b;
L16 printf("p1=%x *p1=%d\n ", p1, *p1);
L17 printf("p2=%x *p2=%d\n ", p2, *p2);
L18 *p1+=*p2+10;
L19 *p2=*p1*5;
L20 printf("p1=%d *p2=%d\n ",
        *p1, *p2);
L21 system("pause");
L22 return 0;
}
```

| Adresses | Mémoire |
|-----------|---------|
| 0x00A126C | |
| 0x00A126C | 88 110 |
| 0x00A126D | 12 550 |
| 0x00A126E | |
| 0x00A126F | |
| 0x00A1270 | |
| 0x00A1271 | |
| 0x00A1272 | |
| 0x00A1273 | |
| 0x00A126E | |
| 0x00A126F | |
| 0x00A1270 | |
| 0x00A1271 | |
| 0x00A1272 | |

10

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

1. REVISIONS

Les pointeurs – Déclaration, initialisation, indirection

L15 Initialisation des variables a et b

```
L10 int main() {
L11 char a, b;
L12 char* p1, p2;
L13 a=88, b=12;
L14 p1=&a;
L15 p2=&b;
L16 printf("p1=%x *p1=%d\n ", p1, *p1);
L17 printf("p2=%x *p2=%d\n ", p2, *p2);
L18 *p1+=*p2+10;
L19 *p2=*p1*5;
L20 printf("p1=%d *p2=%d\n ",
        *p1, *p2);
L21 system("pause");
L22 return 0;
}
```

| Adresses | Mémoire |
|-----------|---------|
| 0x00A126B | |
| 0x00A126C | 88 110 |
| 0x00A126D | 12 550 |
| 0x00A126E | |
| 0x00A126F | |
| 0x00A1270 | |
| 0x00A1271 | |
| 0x00A1272 | |
| 0x00A1273 | |
| 0x00A126E | |
| 0x00A126F | |
| 0x00A1270 | |
| 0x00A1271 | |
| 0x00A1272 | |

L20 *p1=110 *p2=550

11

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

1. REVISIONS

Les pointeurs – paramètres de fonctions

```
L22 void trio(char* x, char* y, char* z) {
L23     *x = *y + *z;
L24     *y = *z - *x;
L25     *z = *x * *y;
}

L10 int main() {
L11     char r=3, s=2, t=1;
L12     trio(&r, &s, &t);
L13     printf("%d %d %d\n", r, s, t);
L14     system("pause"); return 0;
}
```

| | L11 | L23 | L24 | L25 |
|------|------|-----|-----|-----|
| main | 3 | 3 | | |
| | 2 | | -2 | |
| | 1 | | | -6 |
| L22 | | | | |
| | @100 | | | |
| | @101 | | | |
| | @102 | | | |

L12 du main : la copie des adresses respectives de r, s et t dans x, y et z.
permet la modification des paramètres d'appel (effectifs) r, s et t.
En effet, $r \equiv *x$, $s \equiv *y$ et $t \equiv *z$

Affichage : 3 -2 -6

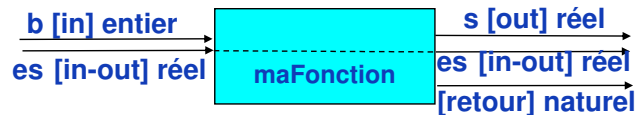
12

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

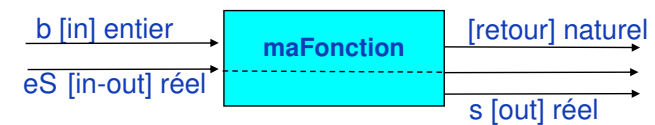
2019-2020

Prototypage – Les bonnes questions à se poser (1/2)

- (1) Quel nom pour la fonction ?
- (2) De quelles données a-t-on besoin pour résoudre le problème ?
Paramètres d'entrée [in]
 - utilisés pour le traitement,
 - caractéristique : dans le **corps de la fonction**,
en partie droite d'affectation et/ou condition
- (3) Quels sont les modes de transmission des résultats calculés par la fonction
 - (a) **transmis par retour [return]**
 - caractéristique : unique résultat, renvoyé par un `return`
 - (b) **transmis par paramètre de sortie [out]**
 - caractéristique : mode de passage par pointeur, utilisé dans le corps de la fonction en partie gauche d'affectation
 - (c) **transmis par paramètre d'entrée-sortie [in-out]**
 - caractéristique : utilisés pour le traitement et modifié dans le corps en partie droite et gauche d'affectation et/ou condition)



Prototypage – L'algorithme du prototypage (2/2)



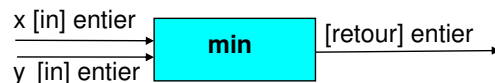
- (1) Y a-t-il un retour de la fonction ?
Oui : écrire le type du retour
Non : écrire `void`
- (2) Ecrire le nom de la fonction
- (3) Ecrire tous les autres paramètres formels entre parenthèses ([in], [in-out] et [out])
Pour tout paramètre [in] : le paramètre est passé par valeur, écrire le `type` et le `nom` du paramètre
Pour tout paramètre [out] ou [in-out] : le paramètre est passé par pointeur, écrire le `type` suivi de `*` et le `nom` du paramètre

Résultat de l'algorithme du prototypage :

```
unsigned int maFonction(int b, float* es, float* s);
```

Prototypage – avec paramètre de retour

Le minimum de deux entiers



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

```
int main() {
    int x=7, y=3, leMin;
    leMin=min(x, y);
}
```

- (1) `printf("min(%d, %d)=%d\n", x, y, leMin);`
- (2) `printf("min(%d, %d)=%d\n", x, y, min(x, y));`

```
min(7, 3)=3
min(7, 3)=3
```

Prototypage – avec paramètre de retour et documentation

Le minimum de deux entiers



```
/* Calcule le minimum de deux entiers
 * x [in] le premier argument
 * y [in] le deuxième argument
 * [retour] le minimum des deux entiers
 */
```

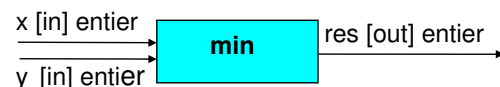
```
int min(int x, int y){
    if (x<y) return x;
    else return y;
}
```

```
int main() {
    int x=7, y=3, leMin;
    leMin=min(x, y);
    printf("min(%d, %d)=%d\n", x, y, leMin);
    printf("min(%d, %d)=%d\n", x, y, min(x, y));
    system("pause"); return 0;
}
```

```
min(7, 3)=3
min(7, 3)=3
```

Prototypage – avec paramètre de sortie (1/2)

Le minimum de deux entiers



Deuxième prototypage avec un paramètre de sortie

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

```
int main() {
    int x=7, y=3, r;
    min(x, y, &r);
    printf("min(%d, %d)=%d\n", x, y, r);
    system("pause"); return 0;
}
```

min(7, 3)=3

Prototypage – avec paramètre de sortie (1/2)

Le minimum de deux entiers



Deuxième prototypage avec un paramètre de sortie

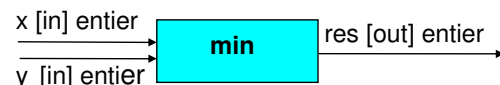
```
void min(int x, int y, int* res){
    if (x<y)
        *res=x;
    else
        *res=y;
}
```

```
int main() {
    int x=7, y=3, r;
    min(x, y, &r);
    printf("min(%d, %d)=%d\n", x, y, r);
    system("pause"); return 0;
}
```

min(7, 3)=3

Prototypage – avec paramètre de sortie et documentation (2/2)

Le minimum de deux entiers



Deuxième prototypage avec un paramètre de sortie

```
/* Calcule le minimum de deux entiers
 * x [in] le premier argument
 * y [in] le deuxième argument
 * res [out] le minimum des deux entiers
 */
```

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

```
int main() {
    int x=7, y=3, res;
    min(x, y, &res);
    printf("min(%d, %d)=%d\n", x, y, res);
    system("pause"); return 0;
}
```

Documentation
de fonction

min(7, 3)=3

Techniquement – Fonction avec paramètre de sortie

? état de la mémoire

(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;
}
```

| | | | | |
|------|-----|-----|-----|--|
| main | L10 | L11 | L12 | |
| val | ? | 3 | 3 | |

stack

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

| | | | | |
|------|------|------|------|------|
| main | L10 | L11 | L22 | |
| val | ? | 3 | 30 | |
| f | L20 | L21 | L22 | L23 |
| x | 0x20 | 0x20 | 0x20 | 0x20 |
| y | ? | 10 | | |

stack

*x=30

(2) L12 : appel de f()

*x est l'autre nom logique (dans la fonction f) de la variable de l'appelant (val) implantée en 0x20

| | | | | |
|------|-----|-----|--|--|
| main | L13 | L14 | | |
| val | 30 | ? | | |

stack

t

Documentation de source

```
/* main.c
 * Auteur : Marie-José Caraty
 * Date de création : 22/10/2018
 */
```

Le cartouche du programme

```
void min(int x, int y, int* res); // void min(int, int, int*);
```

```
int main() {
    int x=7, y=3, res;
    min(x, y, &res);
    printf("min(%d, %d)=%d\n", x, y, res);
    system("pause"); return 0;
}
```

Documentation des fonctions

```
/* Calcule le minimum de deux entiers
 * x [in] le premier argument
 * y [in] le deuxième argument
 * res [out] le minimum des deux entiers
 */
```

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

min(7, 3)=3

Précondition des fonctions

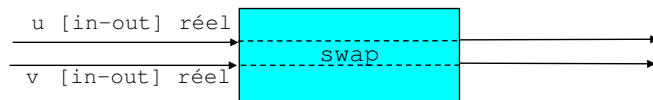
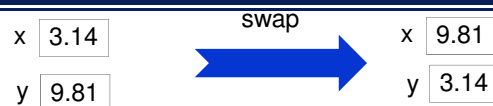
Déclarez une fonction (f) qui au nombre réel x donne pour résultat $1/(2-x)$
Documentez et codez la fonction f
Vous veillerez aux éventuelles préconditions de la fonction

```
/* Calcul de la fonction 1/(2-x)
 * x [in] l'argument
 * [retour] la valeur de la fonction
 * precondition : x différent de 2
 */
```

```
double f(double x) {
    assert(x!=2);
    return 1./(2-x);
}
```

Tout paramètre d'entrée peut faire l'objet d'une précondition

Cas du swap – Echange du contenu de deux variables paramètres d'entrée-sortie



```
/* Echange le contenu de deux variables
 * u [in-out] le premier argument
 * v [in-out] le deuxième argument
 */
```

```
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
```

```
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);
}
```

Tableaux statiques – Déclaration de tableau

(1/2)

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

Déclaration de tableau statique

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

autant de [...] que de dimensions

```
char tab[5]; // un tableau de 5 entiers codés sur 1 octet
// Attention : utilisation d'un nombre magique
```

Bonne Pratique

Utiliser une constante pour définir la taille du tableau

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

On utilisera une constante d'énumération de valeur 5

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

ou encore

```
#define N 5
```

```
short tab[N]={3, 5, 7, 2, 6}; //Initialisation par liste
```

| | | | | |
|---|---|---|---|---|
| 3 | 5 | 7 | 2 | 6 |
| 0 | 1 | 2 | 3 | 4 |

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

accès au 4^{ème} élément par tab[3]

Tableaux statiques – Déclaration et initialisation explicite (2/2)

On peut également initialiser un tableau par une liste explicite sans préciser sa taille

```
int tab[]={3, 5, 7, 2, 6};
```

Dans ce cas, on peut calculer sa taille (nombre d'éléments) par :

```
unsigned int taille = sizeof(tab) /sizeof(int);
```

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

```
char tab[5]; // un tableau de 5 entiers codés sur 1 octet
// Attention : utilisation d'un nombre magique
```

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}; // ou #define N 5
short tab[N]={3, 5, 7, 2, 6}; //Initialisation explicite
```

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

accès au 4^{ème} élément par tab[3]

| | | | | | |
|-----|---|---|---|---|---|
| tab | 3 | 5 | 7 | 2 | 6 |
| | 0 | 1 | 2 | 3 | 4 |

Schématiquement en mémoire

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

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

L'identificateur du tableau (tab) est l'adresse du premier élément du tableau alloué en mémoire
tab vaut 0x00A126E

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

| | | |
|---------------|-----------|--------|
| tab=&tab[0] | 0x00A126E | tab[0] |
| tab+1=&tab[1] | 0x00A1270 | tab[1] |
| tab+2=&tab[2] | 0x00A1272 | tab[2] |

Adresses Mémoire

| | |
|-----------|------|
| 0x00A126B | |
| 0x00A126C | |
| 0x00A126D | |
| 0x00A126E | |
| 0x00A126F | |
| 0x00A1270 | |
| 0x00A1271 | |
| 0x00A1272 | |
| 0x00A1273 | |
| 0x00A1274 | |

.... état (aléatoire) de la mémoire

Passage de tableau dans une fonction

Rappel

tab est l'adresse du premier élément du tableau

```
int* tab; // n'intègre aucune information sur la taille de tab
```

Pour traiter un tableau dans une fonction

- (1) passer la variable tableau
- (2) passer sa taille

Ou encore,

encapsuler le tableau et sa taille dans une structure Tableau

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

Passage de tableau dans une fonction – Passage du tableau et de sa taille

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

```
void initialiser(int* tab, unsigned int size) {
    int i;
    for (i=0; i<size; ++i)
        tab[i]=i+1;
}
```

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

```
int main() {
    enum {TAILLE=5};
    int t[TAILLE];
    initialiser(t, TAILLE);
    afficher(t, TAILLE);
    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 modificateur **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

Passage de tableau dans une fonction – Encapsulation de la taille dans un type structuré

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

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

void afficher(const Tableau* t) {
    unsigned int i;
    for (i=0; i<t->taille; ++i)
        printf(" %d", t->tab[i]);
}
```

```
void initialiser(Tableau* t) {
    unsigned int i;
    for (i=0; i<t->taille; ++i)
        t->tab[i]=i+1;
}
```

```
int main () {
    Tableau t;
    printf("Taille en octet de t : %d\n", sizeof(t));
    t.taille=N;
    initialiser(&t);
    afficher (&t);
    system("pause"); return 0;
}
```

Taille en octet de t : 84

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Passage de variable de type structuré dans une fonction

Optimisation

Pour tout paramètre formel d'entrée (mode [in]) de type structuré de taille supérieure à 4 octets, on **évitera** à l'appel de la fonction, la copie du paramètre effectif dans le paramètre formel en passant le paramètre par **pointeur**.

Sécurisation du paramètre d'entrée

On sécurisera le **tableau** passé en mode [in] (**paramètre d'entrée**) par le modificateur **const** pour empêcher toute modification par le corps de la fonction. Tout accès en écriture dans la fonction provoquera alors une erreur à la compilation.

```
void afficher (const Tableau* t) ;
void afficher(const int t[], unsigned int size);
void afficher(const int* t, unsigned int size);
```

Etude de la fonction minOf [6 points]

(1/5)

La fonction minOf a pour rôle de trouver dans un tableau d'entiers **tab** de taille **size** :

- la valeur minimum des éléments dans l'intervalle des index compris entre les indices **i** et **j** (inclus) avec **i < j**,
- et l'indice de ce minimum dans le tableau.

- (a) Pour les valeurs de **i** et **j** respectivement égales à 2 et 7, donnez la valeur du minimum et son indice dans le tableau suivant :

tab = [12, 2, 3, 1, 7, 24, -1, 94]

| | | | | | | | |
|----|---|---|---|---|----|----|----|
| 12 | 2 | 3 | 1 | 7 | 24 | -1 | 94 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

min : -1 et indice : 6

Etude de la fonction minOf

(2/5)

- (b) Déclarez la fonction `minOf` en utilisant une variable résultat de type structuré nommé `Solution`

```
typedef struct {
    int min;
    unsigned int index;
} Solution;
```

Réponse attendue

```
Solution minOf(const int* t, unsigned int size,
               unsigned int i, unsigned int j);
```

Etude de la fonction minOf

(3/5)

- (c) Donnez un extrait de code permettant de vérifier les assertions du test unitaire vérifiant le résultat de la fonction `minOf` pour l'exemple donné à la question (5.a).

```
int tab[]={12, 2, 3, 1, 7, 24, -1, 94}; //L1
unsigned int size=sizeof(tab)/sizeof(int); //L2
Solution s; //L3
s=minOf(tab, size, 2, 7); //L4
assert((s.min==-1) && (s.index=6));
```

- (d) Donnez l'assertion qui vérifie toutes les préconditions de la fonction `minOf`.

```
Solution minOf(const int* t, unsigned int size,
               unsigned int i, unsigned int j) {
    assert((size>0) && (i>=0) && (i<j) && (j<size));
    ...
}
```

Etude de la fonction minOf

(4/5)

- (c) Donnez un extrait de code permettant de vérifier les assertions du test unitaire vérifiant le résultat de la fonction `minOf` pour l'exemple donné à la question (5.a).

```
int tab[]={12, 2, 3, 1, 7, 24, -1, 94}; //L1
unsigned int size=sizeof(tab)/sizeof(int); //L2
Solution s; //L3
s=minOf(tab, size, 2, 7); //L4
assert((s.min==-1) && (s.index=6));
```

- (d) Donnez l'assertion qui vérifie toutes les préconditions de la fonction `minOf`.

```
assert((size>0) && (i>=0) && (i<j) && (j<size));
```

Etude de la fonction minOf

- (e) Définissez (codez) la fonction `minOf`.

```
Solution minOf(const int* t, unsigned int size,
               unsigned int i, unsigned int j) {
    assert((size>0) && (i>=0) && (i<j) && (j<size));
    Solution s;
    s.min= t[i];
    s.index=i;
    for (unsigned int k=i+1; k<=j; ++k) {
        if (t[k]<s.min) {
            s.min=t[k];
            s.index=k ;
        }
    }
    return s;
}
```

4. REVISIONS

Etude de la fonction minOf

- (f) Codez le programme principal qui affiche les résultats (minimum et index du minimum) de la fonction `minOf` appliquée pour les valeurs de `i=2` et `j=7` au tableau défini en (5.a).

Spécifications : le tableau sera initialisé par une liste explicite de ses éléments et sa taille calculée à partir de sa taille-mémoire.

```
int main() {
    int tab[]={12, 2, 3, 1, 7, 24, -1, 94}; //L1
    unsigned int size= sizeof(tab)/sizeof(int); //L2
    Solution s ; //L3
    s=minOf(tab, size, 2, 7); //L4
    printf("%d %d\n", s.min, s.index);
    system("pause"); return 0;
}
```

37

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

38

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

1. SPECIFICATIONS DE PROJET

Spécification – Ordonnancement (1/3)

Ordonnancement du source

- (1) **Cartouche du fichier source** (documentation du fichier physique)
- (2) **Les inclusions** des entêtes de bibliothèque (".h")
(guidées par l'utilisation d'une au moins de leurs fonctions dans le source)
- (3) **Déclaration des constantes** par directives du préprocesseur (#define)
- (4) Définition des **types structurés** avec **documentation** du type et des ses champs
(si leurs identificateurs ne sont pas suffisamment explicites)
- (5) Tous les **déclarations/prototypes** documenté(e)s des fonctions utilisées et codées dans le source
- (6) Le `main()`
- (7) La **définition documentée** de toutes les fonctions prototypées avant le `main()`

39

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

1. SPECIFICATIONS DE PROJET

Spécification – Documentation (2/3)

Cartouche de programme

Informations à donner sur le fichier source

- (1) le nom du fichier (fichier physique du système de fichier)
information nécessaire dans un contexte de compilation
séparée où plusieurs sources participent à une application
- (2) les auteurs du source
- (3) a minima, la date de création

Documentation des fonctions

- (1) rôle de la fonction
- (2) Liste (dans l'ordre du prototype) de tous les paramètres formels leur identificateur avec leur type [`in`], [`out`] et [`in-out`] et rôle et en dernier, en cas de retour [`retour`] avec son rôle

40

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

Spécification – Exemple

(3/3)

```

/* main.c
 * Auteur : Marie-José Caraty
 * Date de création : 07/10/2016
 */
// Toutes les déclarations de fonction (prototypes)
void min(int x, int y, int* res); // void min(int, int, int*);

int main() {
    int x=7, y=3, res;
    min(x, y, &res);
    printf("min(%d, %d)=%d\n", x, y, res);
    system("pause"); return 0;
}
// Toutes les définitions de fonction
/* Calcule le minimum de deux entiers
 * x [in] le premier argument
 * y [in] le deuxième argument
 * res [out] le minimum des deux entiers
 */
void min(int x, int y, int* res){
    if (x<y)
        *res=x;
    else
        *res=y;
}

```

Le cartouche

Documentation de fonction

min(7, 3)=3

41

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

Eléments de qualité d'un projet

Structuration des données

Définition de tous les types structurés (Equipe, Championnat, Rencontre, Calendrier)

pour mémoriser toutes les informations nécessaires aux traitements de l'application

Architecture logicielle

Analyse des fonctionnalités/fonctions

Conception des fonctions utiles (rôle et prototypage) de l'application

Documentation des sources

Cartouche, constantes, structures de données et leur champs, fonctions

Code

Notion différée : Architecture logicielle

42

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

Dossier de développement logiciel

☐ Respect des spécifications

Respect des spécifications données pour le dossier de développement logiciel

Respect du cahier des charges de présentation : présence dans le dossier
a) une page de garde indiquant le nom et le **groupe** des membres du binôme, l'objet du dossier, **b)** une table des matières de l'ensemble du dossier (incluant les annexes) avec la **pagination** de toutes les rubriques, la pagination est continue du début à la fin du dossier, **c)** présentation de l'application, **d)** organisation des tests, **e)** bilan de validation des tests, **f)** bilan de projet.

Brochez vos dossiers.

☐ Présentation de l'application

L'art de synthétiser le projet (taille donnée)

Rôle fonctionnel de l'application (ce que fait l'application), les entrées et sorties (résultats attendus) [spécification : 1 page, police 12]

✓ Structurations des données en mémoire

✓ Qualité des fonctions (pertinence des primitives)

43

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

Qualité de code

(1/2)

☐ Lisibilité du code

Code parfaitement indenté (tabulation)

Ligne de code limitée à 80 colonnes

☐ Documentation

Du fichier source (son nom, son/ses auteurs et a minima sa date de création) des types, des champs des types structurés, des variables représentant les données

Des fonctions :

rôle de la fonction,

rôle des paramètres formels, leur mode ([in], [out] et [in-out])

rôle du paramètre de retour éventuel

44

DUT Informatique, 1^{ère} année – IAP – Marie-José Caraty

2019-2020

❑ Absence de nombres magiques

Absence de littéraux, utilisation de constantes (d'énumération ou de macros) motivée par des changements possibles des constantes considérées.

❑ Longueur des fonctions

Le `main` doit être court (à très court, quelques lignes).

Un `main` est à un niveau macroscopique (constitué d'appels de fonction).

Un `main` trop long traduit un manque au niveau de l'analyse, des fonctions aurait dû être conçues.

Par exemple on n'y développe pas une interface (e.g., un menu).

Une fonction ne doit pas dépasser une vingtaine de lignes. Une fonction trop longue indique souvent un manque d'analyse fonctionnelle et s'accompagne de redondance : dans ce cas, on relit et on analyse le code en visant un niveau macroscopique (de l'algorithme) pour introduire les fonctions (à coder) et qui seront à appeler (la longueur du code diminuera).

❑ Code redondant

Introduire la/les fonction(s) qui généralisent les traitements.

❑ Tests

D'une manière générale, vous devez préciser votre stratégie de test : comment sont organisés vos tests.

Dans notre cas, les tests sont organisés en 5 Sprints

Un test comprend : un objectif (ce que l'on veut tester), un JDT (jeu de données de test), un résultat attendu (ou résultat de référence), un résultat (**trace d'exécution** de votre programme votre `run.txt`) et un **bilan de validation** (ce que valide le test).

Il faut rédiger cette partie (vérification de votre compréhension des tests et de leur sémantique).

Dans le rapport, on doit trouver une partie de « Bilan de validation des tests » où on synthétise tous les tests et ce qu'ils valident.

Des JDT « personnels » doivent servir à améliorer la couverture de test

❑ Bilan de projet

Retour d'expérience. Les difficultés rencontrées, ce qui est réussi, ce qui peut être amélioré

Objectif

En présence de votre enseignant, tester votre programme du Sprint de plus haut niveau (h , $1 \leq h \leq 5$) que vous avez atteint et validé sur un nouveau jeu de données conçu pour votre groupe n (`inhGn.txt`)

Le test se fera sur l'invite de commande par redirection des entrées et sorties de l'application.

La validation de votre application se fera par la comparaison des résultats de votre application (`run.txt`) avec le résultat de référence (`outhGn.txt`) que vous communiquera votre enseignant.

Vous utiliserez un comparateur de fichier texte (l'archive `jar (diff)` est mise à disposition sur COMMUN pour cette comparaison. Elle présente l'intérêt de préciser les lignes différentes (ajoutées ou omises).

Si aucune différence n'est constatée

votre application est validée pour le sprint `#h` testé

sinon il n'est pas validé et vous devez valider le sprint `#h-1`

Mise en œuvre de la recette

- Préparez un projet Visual de votre application sur une machine de votre choix (machine de l'IUT ou portable personnel). Chargez le source du sprint `#h` de plus haut niveau que vous avez validé. Votre enseignant vous communiquera le fichier `inSph.txt`. Exécutez votre programme avec la redirection des entrées-sorties de votre application (`inSph.txt` et `run.txt`)

- Vérification par votre enseignant de TP de la validité de votre solution

si votre fichier `run.txt` est identique à la solution de référence `outSph.txt` alors votre logiciel est accepté, le Sprint est validé

sinon le logiciel est refusé,

des corrections sont à envisager (5 minutes vous sont accordées)

Le délai passé, vous devez vous préparer à passer la recette du Sprint `#h-1`