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

TDs et TPs de IAP 2019-2020

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX DIRIGES – Semaine n°1

Thèmes

- Evaluation des expressions
- Algorithmes simples
- Conditionnelles

Préparation

Les exercices 4, 5, 6 et 7 sont à coder en langage C.

Exercice 1. Evaluation d'une expression

En suivant le principe d'évaluation des expressions décrit dans le cours (cf. transparents 20 et 21 du cours n°1)

Evaluez l'expression suivante : `2 * 5 + 20 % 7 / 3 - 12`

Rappel

L'opérateur modulo s'écrit `%` en C (Ex : *7%2 vaut 1*).

Remarque

L'opérateur division `/` est la division entière quand ses deux arguments sont entiers (Ex : 7/2 vaut 3).

Exercice 2. Exécution d'une séquence d'instructions

Soit `x`, `y` et `z` trois variables entières de type `int`.

Exécutez en séquence (l'une après l'autre) les 3 lignes d'instruction suivantes :

```
L1    x = 1;
L2    y = x++ + 5 * 3;
L3    z = ++x * 2;
```

Vous réécrirez les instructions en précisant l'ordre d'évaluation par un parenthésage explicite.

Combien valent `x`, y et `z` après chaque instruction. Schématisez l'état mémoire.

Exercice 3. Années bissextiles

Soit `a` une année du calendrier grégorien (donc postérieure à 1582). Ecrivez une expression booléenne qui indique s'il s'agit d'une année bissextile.

Rappels

Une année est bissextile si c'est un multiple de 4 sauf si c'est un multiple de 100 mais les multiples de 400 sont tout de même bissextiles.

Un nombre `n` est un multiple de `x` (non nul) si `n` modulo `x` est nul.

Exercice 4. Expression simple

Convertissez 100 dollars américains en euros avec un taux de conversion de 1.17 dollar pour 1 euro.

Affichez le résultat sous la forme « X dollars = Y euros », les montants en dollars et en euros sont à afficher sur 2 décimales.

Exercice 5. Nombre de jours du mois

Soit un entier `m` (de 1 à 12) représentant un mois de l'année et `a` une année du calendrier grégorien.

Grâce à une expression conditionnelle calculez `nbj` : le nombre de jours du mois `m` pour l'année `a`.

Exercice 6. Lendemain d'une date

Déterminez la date qui suit une date donnée sous la forme de trois entiers j, `m` et `a`. Vous supposerez que la variable `nbj` contient le nombre de jours du mois `m` pour l'année `a`.

N'utilisez pas de nouvelles variables pour la nouvelle date : modifiez directement `j`, `m` et `a` mais vous n'avez pas le droit à l'opérateur `++`.

Exercice 7. Polynôme de degré 1

Soit le polynôme `ax + b` où `a`, `b` et `x` sont réels.

Saisissez le polynôme (i.e. les valeurs de `a` et `b`).

Saisissez `x` et affichez la valeur du polynôme.

Résolvez l'équation `ax + b = 0` et affichez le résultat (il y a 3 cas selon que `a` ou `b` sont nuls ou non).

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX PRATIQUES – Semaine n°1

Thèmes

- Prise en main de Visual Studio 2017
- Expressions, types de donnés, algorithmes simples

Lors de première séance de TP, vous traiterez dans l'ordre les exercices du TP de la semaine.

Lors de la deuxième séance de TP et suivant votre avancement, vous devez continuer à passer sur machine les exercices de TP qui ne l'ont pas été en première séance. Une bonne pratique est de tester sur machine les exercices de TD. Cette semaine, vous passerez sur machine et testerez le code en langage C correspondant aux exercices 4, 5, 6 et 7 du TD.

Préambule. Prise en main de Visual Studio

Lancez Visual Studio 2017 (`démarrer -> Tous les programmes -> Environnement de développement -> Microsoft Visual 2017 -> Microsoft Visual 2017`).

Choisissez les paramètres d'environnement par défaut correspondant au `développement Visual C++`.

Créez un nouveau **projet** (`Fichier -> Nouveau -> Projet…`) :

- Modèle de projet : `Visual C++ -> Général`
- Type de projet : `Projet vide`
- Donnez un nom à votre projet. `TP1-Exo1 est` raisonnable.
- Changez son emplacement (les fichiers doivent être **sur votre dossier réseau** et `Z:\IAP` est la racine de ce dossier).
- Donnez un nom à la ***solution*** de votre projet (une *solution* permet de regrouper plusieurs projets). `IAP-Sem1 ou IAPs1` sont raisonnables.
- Laissez cochée la case « `Créer un nouveau répertoire pour la solution` ».

Lorsque vous créerez de nouveaux projets, Visual Studio vous donnera la possibilité de les ajouter à la solution courante. Nous vous recommandons de réunir tous les projets d'IAP de la même semaine dans une même solution.

Les programmes que vous écrirez seront composés (du moins les deux premières semaines) d'une unique fonction `main`. Vous pourrez au choix créer un projet par exercice ou bien faire un seul projet pour toute la séance et y ajouter un bloc d'instruction entre accolades { } pour chaque exercice. Chaque bloc dispose de ses propres variables : vous pouvez donc utiliser le même nom de variables dans plusieurs blocs. Après avoir testé un bloc, vous pourrez le masquer par un commentaire (`/*…*/`) pour éviter une réexécution ultérieure.
Avant de pouvoir programmer, vous devez créer un fichier C au sein de votre projet (`Projet -> Ajouter un nouvel élément…`). Choisissez le type de fichier `Fichier C++` et donnez-lui un nom **finissant impérativement par «.c»**. Comme vous aurez pour le moment un seul fichier par projet vous pouvez lui donner le même nom que le projet avec **«.c»** à la fin.

Une fois le programme écrit, compilez-le (`Générer -> <votre nom de projet>` ou tapez la touche `F7` ce qui génère toute la solution) et corrigez (si nécessaire) toutes les erreurs. Exécutez votre programme (`Déboguez -> Exécutez sans débogage` ou frappez la touche `F5`).

Exercice 1. Premier programme

Écrivez un programme qui affiche « Bonjour a tous » sur une ligne puis « Ca va ? » sur la ligne suivante.

Exercice 2. Types – taille et domaine de variation des données

Affichez pour chaque type élémentaire du langage C, la taille mémoire (en nombre d'octets) prise par une variable de ce type et le domaine de validité de ce type. Le domaine est caractérisé par une borne minimale et une borne maximale. Ces bornes sont définies (par des macro-instructions) dans le fichier d'entête `"limits.h"`, ajoutez l'entête correspondantes `#include" limits.h"` avant la fonction `main()`. Les constantes que vous devez employer pour chacun des types sont données dans le tableau ci-dessous.

Pour afficher la taille des types, vous utiliserez l'opérateur `sizeof(T)` (pour le type `T`). La borne minimale des types `unsigned` n'est pas définie dans `limits.h`, pourquoi ? Vous afficherez la borne à partir du littéral entier correspondant (constante entière). Pour les type `float` et `double`, vous afficherez simplement leur taille.

Type	*Borne min.*	*Borne max.*
`char`	`CHAR_MIN`	`CHAR_MAX`
`unsigned char`	*Non définie*	`UCHAR_MAX`
`short`	`SHRT_MIN`	`SHRT_MAX`
`unsigned short`	*Non définie*	`USHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`
`unsigned int`	*Non définie*	`UINT_MAX`
`long`	`LONG_MIN`	`LONG_MAX`
`unsigned long`	*Non définie*	`ULONG_MAX`

Type	*Borne min.*	*Borne max.*
`float`	*Non définies*	
`double`	*Non définies*	

Votre programme devra provoquer l'affichage suivant :

```
Types : coût-mémoire (en octets) et domaine de variation
char : 1 - Domaine : -128 .. 127
unsigned char : 1 - Domaine : 0 .. 255
short : 2 - Domaine : -32768 .. 32767
unsigned short : 2 - Domaine : 0 .. 65535
int : 4 - Domaine : -2147483648 .. 2147483647
unsigned int : 4 - Domaine : 0 .. 4294967295
long : 4 - Domaine : -2147483648 .. 2147483647
unsigned long : 4 - Domaine : 0 .. 4294967295
float : 4
double : 8
```

Note : Pour un bon affichage des domaines de variation, vous utiliserez les formats : `%u` pour les `unsigned int`, `%ld` pour les `long`, `%lu` pour les `unsigned long`.

Exercice 3. Switch

Un médecin de campagne peut être selon le jour de la semaine être : présent (du lundi à vendredi), en congé (le dimanche) ou d'astreinte (le samedi).

Selon le numéro du jour dans la semaine, affichez le statut du médecin ou un message d'erreur si le numéro du jour est incorrect. Le lundi est le jour 1.

Utilisez l'instruction switch.

Exercice 4. double et float sont différents

Lisez deux nombres réels (un `double` et un `float`) et affichez leur moyenne.

Indiquez `%f` pour saisir un `float` avec `scanf` et `%lf` pour saisir un `double` sous peine de récupérer une valeur totalement différente !

Par contre pour l'affichage avec `printf`, `%f` correspond à un double (mais un `float` est également correctement affiché).

Exercice 5. Conditionnelles et tests sur des réels

Affichez le plus grand parmi trois nombres réels. Au choix vous les saisirez ou les initialiserez avec différentes valeurs (et dans ce dernier cas vous recompilerez votre projet) pour tester votre programme sur plusieurs cas.

Utilisation de l'assertion pour les tests

Au lieu d'afficher le résultat vous pouvez écrire une assertion de la forme

```
assert(max == 5.2) ; // vérifie que max vaut bien 5.2
```

L'inclusion <`assert.h`> devra alors être ajoutée avant la fonction `main()`.

Une assertion interrompt l'exécution du programme avec un message d'erreur si elle est fausse.

Hélas, pour comparer deux réels il est vivement conseillé de prévoir une marge d'erreur et d'écrire plutôt :

```
double marge = 0.0001 ; //par exemple
```

$|max-5.2| < marge$ (1)

Rappel : $|x| = -x$ si $x < 0$ sinon x.

Exprimez (1) sous la forme d'une expression booléenne.

Implantation des exercices du TD et test des solutions

Implantez les exercices du TD en machine et testez-les dans la mesure du possible sur l'ensemble des valeurs (Jeu de Données de Test - JDT) représentant les différents cas des algorithmes.

Pour cela, plusieurs solutions :

a) Soit vous saisissez des valeurs différentes lors de plusieurs exécutions.

b) Les variables ne sont pas saisies mais initialisées (en dur) dans le code-source. Pour tester d'autres données de test, les variables seront initialisées différemment et le programme recompilé avant chaque exécution.

Dans ces deux premiers cas, la vérification se fera dans le code par un affichage des résultats aussi explicite que possible

c) La vérification pourra se faire par assertion (cf. exercice 5) sans besoin d'affichage. Ce type de test est appelé **test unitaire**.

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX DIRIGES – Semaine n°2

Thèmes

- Algorithmes
- Structures de contrôle itératives et conditionnelles

Préparation

Les exercices sont à préparer (en pseudo-code ou en langage C).

Exercice 1. La factorielle d'un nombre entier naturel

Ecrivez en pseudo-code un algorithme qui permet de calculer la factorielle d'un entier naturel *n*. Vous utiliserez dans une première version une boucle *tant que*. Vérifiez que votre boucle s'arrête.

> On rappelle que la factorielle de *n* (notée *n!*) est le produit des nombres entiers strictement positifs inférieurs ou égaux à *n*.
>
> $n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$

Codez un programme qui permet de calculer la factorielle d'un nombre positif que vous initialiserez dans le code à l'aide d'une boucle *for*. Vérifiez l'arrêt de la boucle.

Donnez un plan de tests unitaires.

Exercice 2. Les diviseurs d'un nombre

Affichez tous les diviseurs d'un nombre *n* en les séparant par un espace.

> On rappelle que le diviseur d d'un nombre n est tel que le reste de la division entière de n par d est nul.

Facultatif : Quelles sont les modifications à apporter au code pour afficher les diviseurs 3 par ligne. Indication : utilisez l'opérateur modulo.

Donnez un plan de tests unitaires.

Exercice 3. PGCD de deux nombres

Le PGCD (plus grand commun diviseur) de deux entiers naturels est le plus grand entier naturel qui les divise tous les deux.

Calculez le PGCD de deux nombres sachant que :

> PGCD(a, 0) = a ;
>
> Si $b \neq 0$ alors PGCD(a, b) = PGCD(b, a%b)

Donnez un plan de tests unitaires.

Exercice 4. Nombre premier

Un nombre *n* est premier s'il a deux diviseurs : 1 et *n* lui-même (1 n'est pas premier).

Déterminez si un nombre est premier en testant ses diviseurs potentiels et en remarquant que si un nombre n'est pas premier il a au moins un diviseur (autre que 1) plus petit ou égal à sa racine carrée.

Donnez un plan de tests unitaires.

Exercice 5. Eratosthène et les nombres premiers

Affichez les *n* premiers nombres premiers en utilisant l'algorithme dit du crible d'Eratosthène.

Ce dernier initialise un tableau *crible* de booléens à vrai (de l'indice 2 à l'indice n) puis teste chaque nombre *i* du tableau ainsi : si *crible(i)* est vrai alors *i* est premier et on barre (c.à.d. on marque à faux) tous les multiples de *i*. Il est inutile de barrer les multiples des nombres non premiers.

Remarquez par ailleurs que certains multiples de *i* ont déjà été testés s'ils sont aussi multiples d'un nombre plus petit que *i*. Il suffit donc de barrer les multiples de *i* à partir de *i*i*.

Donnez un plan de tests unitaires.

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX PRATIQUES – Semaine n°2

Thèmes

- Traitement des tableaux et chaînes de caractères
- Les structures de contrôle (itératives et conditionnelles)

Préparation

Les exercices sont à préparer et à coder en C.

Les tableaux d'entiers ou de réels peuvent être initialisés par des listes explicites (cf. T26 du cours 2) et les chaînes de caractères par des littéraux de chaîne (cf. T28 du cours 2).

Exercice 1. La table ASCII

Le code ASCII établit une correspondance entre un entier naturel (codé sur un octet) et un caractère. Le code ASCII établit cette correspondance sur la plage 0..127 ; de 128 à 255, on parle de code ASCII étendu. Les 31 premiers caractères ne sont pas affichables (caractères de contrôle). Ecrivez un programme en C qui affiche la table ASCII pour les entiers de 32 à 255.

Repérez les plages de contigüité de codage des chiffres et des lettres de l'alphabet (minuscules et majuscules).

- Vous afficherez un entier avec le format "%d" et un caractère avec le format "%c".
- Affichez la table avec 8 caractères par ligne sous le format suivant :

```
32->' '.. 33->'!'.. 34->'"'.. 35->'#'.. 36->'$'.. 37->'%'.. 38->'&'.. 39->'''..
```

Vous utiliserez le format "%3d" pour cadrer l'entier à droite sur 3 caractères (cf. Cours1, Transparent 7).

Exercice 2. Tableau trié

L'objectif est de vérifier si un tableau (*tab*) de taille (*size*) est trié par ordre strictement croissant.

∀ i / i≥0 et i<size-1, tab[i] < tab[i+1]

Première version

1) Déclarez un tableau `t` d'entiers de taille 4.

2) Lisez les 4 éléments du tableau à partir d'une entrée clavier.

3) Vérifiez par affichage le tableau entré suivant le format t= [e1, e2, e3, e4].

4) Recherche de l'indice du premier élément non trié par ordre strictement croissant sinon positionnez cet indice à la taille du tableau.

5) Affichage du résultat : affichez suivant le cas « Tableau trié par ordre strictement croissant » ou « Tableau trié par ordre strictement croissant jusqu'à l'indice » suivi de la valeur de l'indice.

6) Testez votre programme avec des jeux de données qui testent les différents résultats possibles correspondant au problème posé. Vérifiez entre autres le résultat obtenu dans le cas de deux éléments consécutifs égaux.

Deuxième version

Modifiez la phase 1) par une initialisation par liste (« en dur ») dans le programme avec les valeurs précédemment données pour la taille et les éléments du tableau.

Vérifiez les résultats des différents cas de test.

Exercice 3. « Egalité » de tableaux

Vérifiez que deux tableaux de même taille sont égaux, c'est-à-dire qu'ils ont les mêmes éléments aux mêmes indices. Testez les deux cas (égalité et différence).

Exercice 4. Occurrences dans un tableau

Remplacez toutes les occurrences d'un élément de valeur `v` donnée dans un tableau d'entiers par 0, affichez le tableau résultant.

Exercice 5. Ordre lexicographique

Comparez deux chaînes s1 et s2 et affichez un entier positif si s1>s2 dans l'ordre lexicographique, négatif si s1<s2 et nul sinon. Testez cette comparaison de chaines.

Exercice 6. Copie de chaîne (`strcpy()`)

Copiez une chaîne dans une autre chaîne. Testez votre programme en vérifiant votre résultat avec l'affichage du résultat de le fonction `strcpy` de la bibliothèque standard `sdtlib` des chaînes de caractères : `strcpy(s2, s1)` si votre chaîne originelle a pour nom `s1` et la chaîne de destination `s2`.

Exercice 7. Palindrome

Un mot (resp. une phrase) est un palindrome si on peut le (resp. la) lire indifféremment de gauche à droite et de droite à gauche. Vérifiez cette propriété pour une chaîne de caractères.

Exercice 8. Nombre d'occurrences dans un tableau

Comptez le nombre d'occurrences d'un réel dans un tableau. La comparaison entre réels peut être imprécise. *Indication : pour comparer deux réels il est vivement conseillé de prévoir une marge d'erreur. Par exemple pour tester l'égalité de r1 et r2 : |r1- r2| < marge.*

En langage C, la fonction fabs(r) donne la valeur absolue d'un réel (abs celle d'un entier). Pour calculer |r1- r2|, codez fabs(r1-r2).

Exercice 9. L'inverse d'une chaîne

Affichez l'inverse d'une chaîne de caractères sans modifier la chaîne d'origine. Faites le même exercice mais en modifiant la chaîne d'origine.

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX DIRIGES – Semaine n°3

Thèmes

- Les fonctions

Exercice 1. Utilisation et définition de fonction

1.1. Utilisation de fonction d'une bibliothèque

La fonction `power` de la librairie mathématique `math.h` calcule x^y pour deux réels `x` et `y`. Son prototype est le suivant :

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

Affichez dans un programme principal x^y pour les valeurs de x et y respectivement égales à 3 et 5. Format d'affichage : "3^5 = *valeur calculée*"

Vérifiez le résultat obtenu.

Remarque : Pour l'utilisation de toute fonction mathématique, inclure l'entête de la librairie (`#include <math.h>`).

1.2. Définition de fonction

A partir du programme principal de l'exercice 1 du TD2 donné ci-dessous, codez une fonction nommée `factorielle` qui calcule la factorielle d'un entier naturel `n` (notée `n!`). Ecrivez un programme principal qui affichera le résultat de la fonction pour les valeurs de `n` égales respectivement à `0`, `1`, `3` et `5` sous le format `3!=6` pour `n=3`.

```
int main() {
   unsigned int n, i, fact;

   printf("Calcul de n!, entrez n : ");
   scanf("%u", &n);
   for (i=0, fact=1; i<n; ++i)
     fact *= i+1;
   printf("%u! = %u\n", n, fact);
   system("pause"); return 0;
}
```


Exercice 2. Compter les appels à une fonction

Soit la fonction `badCompteur` suivante qui tente de compter le nombre de fois où elle est appelée. Le bloc d'instructions qui suit est extrait d'un programme principal qui appelle deux fois `badCompteur`. Quel est l'affichage après exécution du code ? Pourquoi ?

```
L20  unsigned int badCompteur() {
L21    unsigned int cpt = 0;
L22    ++cpt;
L23    return cpt;
     }
```

```
     int main() {
L30    unsigned int x;
L31    x = badCompteur();
L32    x = badCompteur();
L33    printf("x = %i\n", x);
L34    system("pause"); return 0;
     }
```

Exercice 3. Simuler l'exécution d'une fonction

Un programmeur débutant a écrit la fonction `badPermut` pour permuter la valeur de deux variables entières. Dans le `main` de son code source, il teste sa fonction. Déroulez à la main l'exécution de son programme ci-dessous, en schématisant l'état de la mémoire que vous ferez évoluer à chaque instruction. Indiquez ce qu'affiche le programme. Est-ce que la fonction `badPermut` se comporte comme prévu ? Pourquoi ? Modifiez le code en conséquence.

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

L20  void badPermut(int a, int b) {
L21    a += b;
L22    b = a - b;
L23    a -= b;
L24    printf("Dans badPermut apres permutation a = %i b = %i\n", a, b);
     }

L10  int main() {
L11    int x = 3, y = 5;
L12    printf("Avant badPermut x = %i y = %i\n", x, y);
L13    badPermut(x, y);
L14    printf("Apres badPermut x = %i y = %i\n", x, y);

L15    system("pause");
L16    return 0;
     }
```

Exercice 4. Division par soustractions successives

Prototypez, codez et testez une fonction qui renvoie la division entière (A/B) de deux entiers naturels A et B par soustractions successives (de B à A).

Exercice 5. Conjecture hongroise

Soit `h` la fonction sur les entiers naturels non nuls définie ainsi :
si n est pair alors `h(n) = n/2` sinon `h(n) = 3n+1`.

Codez cette fonction telle qu'elle est définie et vérifiez à l'aide d'assertions que `h(1) = 4`, `h(2) =1` et `h(3) =10`.

Soit `g` la fonction sur les entiers naturels non nuls définie ainsi :
`g(1)=0` et si `n>1 g(n) =1+g(h(n))`.

Codez la fonction `g` telle qu'elle est définie et vérifiez à l'aide d'assertions que `g(1)=0`, `g(2) =1` et `g(3) =7`.

Codez à présent la fonction `g` avec une boucle. Remarquez qu'elle compte simplement combien de fois il faut appliquer `h` pour obtenir `1`. Testez-la avec les mêmes valeurs que précédemment.

Peut-on être sûr que cette boucle se termine toujours ?

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX PRATIQUES – Semaine n°3

Thèmes

- Fonctions
- Débogueur

Préparation

Les exercices sont à préparer et à coder en C.

Exercice 1. Utilisation du débogueur

Un débogueur est un outil puissant que doit maîtriser tout développeur de logiciel. Cet outil permet d'exécuter pas à pas son programme ou bien de l'arrêter à des endroits précis et d'inspecter la valeur des variables en mémoire.

Comme son nom l'indique cet outil permet notamment de chercher à comprendre l'origine de bogues mais plus généralement il permet de comprendre finement l'exécution d'un programme.

Il est important toutefois de savoir comment utiliser cet outil **efficacement** car une exécution pas à pas exhaustive est le plus souvent fastidieuse et extrêmement longue.

Deux outils sont indispensables pour aller dans la plage d'exécution que l'on veut inspecter : les **points d'arrêts** et les **différentes sortes de pas à pas**. Enfin vous devez savoir comment inspecter la valeur d'une variable ou même d'une expression quelconque au fur et à mesure de l'exécution grâce à l'introduction d'**espions**.

Illustrons tout de suite ces concepts sur l'exercice 3 du TD3 : la fonction `badpermut` qui échouait à permuter la valeur de deux variables de la fonction `main` qui l'appelait.

Pour commencer, créez un projet `Exo1` dans votre solution `TP-Sem3` et ajoutez un nouveau fichier source `Exo1.c` dans lequel vous ferez un *copier/coller* du code de `badPermut` et de la fonction `main` donné ci-dessous (cf. COMMUN/Sem3/TP).

Introduisez un point d'arrêt juste avant la déclaration et initialisation de `x` et `y` dans la fonction `main`, en cliquant avec le bouton gauche de la souris dans la bande verticale grise qui borde le texte de votre programme sur la gauche (au niveau de la ligne de déclaration). Vous devez voir apparaître une **boule rouge** : c'est le point d'arrêt. Il force l'arrêt de l'exécution juste avant cette ligne.


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

void badPermut(int a, int b) {
      a += b;
      b = a - b;
      a -= b;
      printf("Dans badPermut apres permutation a = %i b = %i\n", a, b);
}

int main() {
      int x = 3, y = 5;
      printf("Avant badPermut x = %i y = %i\n", x, y);
      badPermut(x, y);
      printf("Apres badPermut x = %i y = %i\n", x, y);

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

A présent, vous pouvez lancer l'exécution du programme : soit en tapant **`F5`** (mais pas `CTRL + F5` qui exécute le programme sans déboguer et qui ignore donc les points d'arrêts) soit à partir du menu « Déboguer » /« Démarrez le débogage ».

L'exécution doit démarrer (vous voyez une console qui apparait) et s'arrêter presque aussitôt sur le point d'arrêt : une **flèche jaune** indique la position actuelle du compteur ordinal (ou pointeur d'instruction courante).

En bas du cadre qui affiche le code de votre programme vous devez voir apparaître deux fenêtres. En bas à gauche, la fenêtre « Automatique » représente les données de la pile d'exécution. Pour le moment vous voyez que les variables `x` et `y` ont des valeurs arbitraires (état de la mémoire) : elles ne sont pas encore initialisées car le point d'arrêt est placé juste avant leurs initialisations.

En bas à droite vous voyez le cadre qui représente la pile des appels de fonctions. En cliquant sur le bouton droit de la souris vous pouvez si nécessaire cocher ou décocher diverses informations selon ce qui vous intéresse. Par exemple vous pouvez décocher les « offsets d'octets » qui ne nous intéressent pas ici. Remarquez qu'au sommet de la pile est placé l'appel à la fonction `main`. En dessous, les appels précédents ne nous intéressent pas ici.

Au-dessus du cadre contenant le texte du programme, doit apparaître un groupe de boutons correspondant au menu déboguer. Si ce n'est pas le cas utilisez le menu « Déboguer » à la place ou les touches de fonctions correspondantes.

- Le **pas à pas détaillé** (**`F11`**) exécute toutes les instructions les unes après les autres. C'est le mode le plus lent. Utilisez le quand vous êtes arrivés à l'endroit qui vous intéresse.
- Le **pas à pas principal** (**`F10`**) est similaire mais il exécute sans débogage l'appel à une fonction.
- Enfin le **pas à pas sortant** (**`Majuscule + F11`**) permet de terminer l'exécution d'une fonction sans débogage, par exemple si vous y êtes entré par erreur ou si le pas à pas détaillé de la fonction ne vous intéresse plus.

Appuyez une fois sur le pas à pas détaillé. Vérifiez dans la fenêtre en bas à gauche que les valeurs de `x` et `y` ont bien été mises à jour. L'instruction suivante est un appel à `printf`. Comme vous ne voulez pas l'exécuter en détail utilisez

une fois le pas à pas principal pour sauter cet appel. Normalement dans la console vous pouvez voir l'effet de `printf`. Si vous vous êtes trompé et êtes entré dans l'exécution de `printf` vous verrez apparaître une fenêtre avec du code incompréhensible : pas de panique ! Tapez un pas à pas sortant (**`Majuscule + F11`**) pour sortir de cet appel à `printf`.

A présent vous devez avoir la flèche jaune qui pointe sur l'appel `badPermut(x,y)` dans le `main`. Tapez un pas à pas détaillé. Remarquez que la flèche jaune s'est bien déplacée au début de `badPermut` et que dans la fenêtre en bas à droite l'appel à `badPermut` a bien été empilé. Remarquez bien que la fenêtre automatique en bas à gauche indique à présent les variables locales à la fonction (paramètres formels et variables du bloc de la fonction) de `badPermut` : `a` et `b` (les paramètres formels). Mais les variables de `main` n'ont pas disparu. Double-cliquez sur l'appel à `main` dans la pile des appels et vous verrez à nouveau `x` et `y`. Double-cliquez ensuite sur l'appel à `badPermut` pour réafficher `a` et `b`. Exécutez pas à pas les trois premières instructions de `badPermut`. Constatez que les valeurs de `a` et `b` ont bien été permutées. Pas à pas principal pour sauter le `printf`. Encore une fois pour sortir de `badPermut`.

La flèche jaune est à présent à nouveau dans le `main`, toujours sur l'appel à `badPermut` (cela serait sans doute plus clair si la flèche était placée après l'appel). Notez bien qu'en bas à gauche la fenêtre automatique affiche à nouveau `x` et `y` qui n'ont pas été modifiées. En bas à droite la pile des appels ne contient plus l'appel à `badPermut`. En effet en sortant de la fonction tout son contexte d'exécution a été dépilé ce qui se voit aussi bien dans la pile des appels que dans le cadre « Automatique ».

Pour finir rapidement la fin de l'exécution : cliquez sur le « triangle vert » ou tapez `F5` ou bien « Déboguer » /« Démarrez le débogage ».

Exercice 2. Factorielle

Dans cet exercice nous allons comparer deux façons très différentes de coder la même fonction ce qui vous permettra aussi de mieux comprendre la notion de pile d'exécution.

Tout d'abord ajoutez à votre projet la fonction suivante (définie en TD).

```
unsigned int factorielle(unsigned int n) {
      unsigned int i, fact;
      for (i=0, fact=1; i<n; ++i)
            fact *= i+1;
      return fact;
}
```

Dans la fonction `main` appelez `factorielle(3)`. Mettez un point d'arrêt avant et exécutez pas à pas chaque instruction. Notez que `factorielle` est appelée une seule fois et que le corps de la boucle est parcouru plusieurs fois.

À présent ajoutez la fonction suivante à votre projet et appelez `factorec(3)` dans le `main`. La fonction récursive est fondée sur la définition récursive du factoriel (n !=n*(n-1)!) et s'appelle elle-même. Ôtez le point d'arrêt devant l'appel *factorielle(3)* et placez-en un devant l'appel *factorec(3).* Exécutez un pas à pas (détaillé).


```
unsigned int factorec(unsigned int n) {
      if (n==0)
            return 1;
      else return n*factorec(n-1);
}
```

Surveillez la pile des appels et notez que `factorec` apparait une, puis deux, puis trois, puis quatre fois avec des valeurs différentes pour `n`. C'est la même fonction mais le contexte d'exécution est différent à chaque fois.

Notez enfin qu'à chaque fois que la flèche jaune passe sur l'accolade de fin de la fonction, le contexte de cet appel est dépilé : on a donc quatre, puis trois, puis deux, puis un seul, puis aucun appel à `factorec` dans la pile des appels.

Exercice 3. Programmation descendante

Pour résoudre un problème informatique une bonne démarche consiste à le décomposer en fonctions et ainsi de suite pour chaque fonction jusqu'à ce que chacune d'elle soit facile à coder et ne dépasse pas quelques lignes de code.

Le principe du jeu est le suivant : un joueur choisit un nombre entier naturel (inférieur à une valeur maximale autorisée), l'autre joueur doit deviner ce nombre en un nombre minimum d'essais.

À chaque fois que le second joueur fait une proposition le premier répond « trop grand », « trop petit », ou « Nombre `x` découvert en n coups ».

Pour obtenir un entier naturel pseudo-aléatoire : inclure les entêtes `"stdlib.h"` et `"time.h"`. Vous initialiserez alors le générateur pseudo-aléatoire une seule fois en début de votre programme par l'instruction suivante. Sans cette initialisation vous obtiendrez toujours la même séquence pseudo-aléatoire à la relance de votre programme.

```
      srand(time(NULL));
```

Vous obtiendrez un nombre pseudo-aléatoire entre 0 et n-1 par l'expression :

```
      rand() % n
```

Vous pouvez développer trois versions de ce jeu, choisissez la ou les versions que vous développerez :

- humain contre humain : la valeur à deviner est saisie au clavier et la recherche de la valeur se fait au clavier.
- ordinateur contre humain : la valeur à deviner est choisie aléatoirement par l'ordinateur et la recherche de la valeur se fait au clavier.
- humain contre ordinateur : la valeur à deviner est saisie au clavier et la recherche de la valeur se fait par l'ordinateur. Dans cette version, vous pourrez régler « l'intelligence » de l'ordinateur à trois niveaux (par efficacité croissante) :
  - l'ordinateur choisit aléatoirement une valeur indépendamment des essais précédents,
  - l'ordinateur choisit une valeur aléatoirement plus grande ou plus petite en fonction des réponses précédentes,
  - l'ordinateur fait une dichotomie (en choisissant la valeur médiane).

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX DIRIGES – Semaine n°4

Thèmes

- Fonctions et paramètres
- Mode des paramètres
- Tableaux en tant que paramètres

Exercice 1. Facilité d'utilisation

Donnez les prototypes possibles pour une fonction rendant le résultat et le reste de la division entière de deux entiers naturels. Précisez le mode des paramètres de vos fonctions.

Écrivez un court programme donnant pour chaque prototype une invocation de la fonction. Désignez le prototype vous semblant le plus pratique du point de vue de celui qui invoque la fonction.

Exercice 2. Permutation

Prototypez et codez une fonction permutant le contenu de deux variables entières.

Écrivez un court programme invoquant cette fonction et dessinez un schéma de la mémoire correspondant à l'exécution de la fonction.

Exercice 3. Affichage

Pour gérer les notes d'un étudiant, la constante et le type qui suivent ont été définis :

```
// enum car constante employée pour dimensionner un tableau
enum {NB_EPREUVES = 10};

// une note par épreuve
typedef float Notes[NB_EPREUVES];
```

Écrivez une fonction affichant les notes d'un étudiant suivant le format :

```
[13.0,  4.0, 12.0,  5.0, 20.0, 16.0, 11.0,  8.0,  6.0,  0.0]
```


Exercice 4. Moyenne pondérée

La note moyenne d'un étudiant est calculée en prenant en compte le coefficient (un réel) de chaque épreuve (comme pour le baccalauréat ou les notes d'UE de votre DUT). Écrivez une fonction calculant la moyenne d'un étudiant. Vous noterez que le type `Notes` (cf. exercice 3) peut être employé pour stocker les coefficients des épreuves.

Exercice 5. Correction des copies

Pour simuler la correction des copies, il vous est demandé d'écrire une fonction affectant une note aléatoire à chacune des épreuves. Pour ce faire, les constantes suivantes sont définies :

```
const int NOTE_MIN = 0, NOTE_MAX = 20;
```

Notez que la fonction `rand()` de la bibliothèque standard retourne un nombre aléatoire compris entre `0` et `RAND_MAX` (qui vaut au moins `32767`).

Exercice 6. Modifier une note

Écrire une fonction permettant de modifier une note d'une épreuve donnée (désignée par sa position dans le tableau de note).

Exercice 7. Moyennes par épreuve d'une promotion

Les notes de la promotion sont stockées au moyen de la constante et du type qui suivent :

```
// enum car constante employée pour dimensionner un tableau
enum {NB_ETUDIANTS = 175};

// une note par épreuve par étudiant
typedef Notes TableauDeNotes[NB_ETUDIANTS];
```

Écrivez une fonction qui calcule les moyennes de la promotion à chacune des épreuves.

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX PRATIQUES – Semaine n°4

Thèmes

- Fonctions et paramètres
- Mode des paramètres
- Structure en tant que paramètres et retours de fonction

Présentation

Dans le cadre d'une application dédiée à l'industrie spatiale, il est nécessaire de manipuler des entiers de très grande taille (à terme des entiers compris entre -10^{100} et 10^{100}). Dans ce but, il a été décidé de définir les types et constantes suivants :

```
typedef enum {POSITIF, NEGATIF} Signe;

enum {NB_CHIFFRES = 100};

// Chaque case du tableau 'chiffres' contient un chiffre
// composant le nombre.
// Les unités sont à l'indice 0, les dizaines à l'indice 1, etc
typedef struct {
     Signe signe;
     unsigned char chiffres[NB_CHIFFRES];
} Nombre;
```

A terme, la valeur de la constante NB_CHIFFRES sera modifiée. Nous rappelons que les types structurés (à l'instar des types de base et à la différence des tableaux statiques) sont passées par valeur lorsqu'ils sont employés en tant que paramètre.

Exercice 1. Affichage

Écrivez une fonction affichant un nombre. Seuls les chiffres significatifs doivent être affichés.

Exercice 2. Zéro

Écrivez une fonction retournant un nombre égal à zéro. Faites un court programme invoquant cette fonction et dessinez le schéma mémoire correspondant à son exécution. Proposez une solution au problème que vous remarquez.

Exercice 3. Initialisation

Écrivez une fonction permettant d'initialiser un nombre à une valeur donnée (un `int`).

Exercice 4. Incrémentation

Écrivez une fonction incrémentant un nombre. Notez bien que l'incrémentation du nombre -1 provoque le changement de signe.

Exercice 5. Conversion en texte

La fonction d'affichage de l'exercice 1 est jugée peu pratique. Proposez une fonction convertissant un nombre en chaîne de caractères. Notez que tout nombre peut être représenté par une chaîne de longueur `NB_CHIFFRES + 2` (en comptant le signe et le caractère `'\0'` terminant la chaîne).

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX DIRIGES – Semaine n°5

Thèmes

- Choix des structures de donnée
- Conception d'algorithmes
- Complexité des algorithmes

Présentation

Nous faisons l'hypothèse que nous sommes dans un monde parfait dans lequel les devises « *les amis de mes amis sont mes amis* » et « *mes amis sont mes amis pour la vie* » ont été adoptées par chacun. De plus, seuls des immortels peuplent ce monde parfait. Le nombre d'habitants est connu et il n'évolue plus.

Lorsque deux habitants de ce monde se rencontrent, ils ont souvent à décider s'ils sont déjà amis ou pas. Pour répondre à cette question, chacun doit énumérer ses amis actuels afin de détecter s'ils ont des amis communs. Cette tâche (laborieuse et source d'erreur) est si fréquente qu'il a été décidé de proposer une solution informatique.

Le système que vous développerez pourra être interrogé par chacun pour savoir si deux personnes sont connues comme étant amies. Dans le cas contraire, il devra être possible d'enregistrer leur amitié nouvelle auprès du système.

Chaque habitant sera désigné dans le système par un numéro qui lui est propre (allant de 0 à n – 1 où n représente le nombre d'habitants). Le système devra être initialisé de façon à ce qu'aucun habitant n'ait d'ami.

Tous les algorithmes demandés seront exprimés en pseudo-code.

Exercice 1. Première analyse

Proposez une structuration de donnée permettant d'encoder la relation d'amitié liant les habitants. Indiquez dans la mesure du possible le coût (la complexité temporelle et spatiale) des deux fonctionnalités devant être assurées par le système. Pour ce faire vous devrez imaginer et exprimer (en pseudo-code) les algorithmes correspondants.

Exercice 2. Une structuration alternative

Une structuration des données bien connue pour ce problème permet d'obtenir des algorithmes de très faible complexité (quasi constante).

Le principe général consiste à désigner au sein de chaque groupe d'amis, un représentant (unique) du groupe. Comme initialement personne n'a d'ami, chaque habitant sera le représentant de son propre groupe.

La structure de données est constituée d'un simple tableau (nommé `amis`) contenant une case par habitant. L'indice d'une case correspond au numéro de l'habitant représenté par la case.

Dans chaque case du tableau est stocké l'indice d'une personne appartenant au même groupe d'amis (le représentant du groupe par exemple).

Initialement, chaque habitant est seul dans son groupe et en est le représentant. Donnez le tableau dans sa configuration initiale pour un monde composé de 4 habitants.

Donnez le pseudo-code de cette initialisation pour une population de n habitants.

Exercice 3. Premier algorithme

Un algorithme essentiel de cette structuration est la détermination du représentant du groupe d'amis pour une personne donnée. Les représentants de groupe sont les seuls pour lesquels la valeur de la case est égale à son indice. Pour une personne donnée, trouver le représentant du groupe auquel elle appartient revient à suivre la suite d'indices formée par les cases jusqu'à trouver un représentant. Par exemple, à partir de la situation suivante (avec une population de 6 habitants) :

	0	1	2	3	4	5
amis	0	0	1	3	0	3

déterminer le représentant du groupe auquel appartient 2 revient à parcourir les cases d'indice 2, 1 puis 0.

De ce tableau, nous pouvons déduire que 0 est le représentant du groupe auquel appartiennent 1 et 4, que 2 a le même représentant que 1 (et donc que c'est 0 qui représente ce groupe) et que 3 est le représentant du groupe auquel appartient 5. On peut en conclure que {0, 1, 2 et 4} ainsi que {3, 5} sont les deux groupes d'amis.

Nous verrons par la suite comment nous assurer que le tableau ne contienne pas de chaîne d'indices formant un circuit non élémentaire (ne passant jamais par un représentant de groupe).

Donnez le pseudo-code de l'algorithme déterminant le représentant d'un groupe auquel appartient une personne donnée.

Exercice 4. Algorithme complémentaire

Deux personnes sont des amis si leurs groupes respectifs ont le même représentant. L'algorithme est immédiat en se basant sur la solution de l'exercice précédent.

Lorsque deux personnes sont déclarées comme étant liées d'amitié alors que ce n'est pas encore le cas, le principe consiste à indiquer que le représentant du premier groupe est à présent le représentant du second groupe (ceci nous assure que le tableau ne contiendra jamais de circuit d'indice). Pour ce faire, il est suffisant d'affecter une seule case du tableau. Déterminez laquelle et donnez le pseudo-code de l'algorithme correspondant.

Exercice 5. Complexité

Imaginons un monde de 10 personnes. À partir de la situation initiale, les personnes 0 et 1, puis 1 et 2, … jusqu'à 8 et 9 sont successivement déclarées comme étant liées d'amitié. Donnez la situation finale du tableau et déduisez en la complexité dans le pire des cas des deux précédents algorithmes (cf. exercices 3 et 4).

Exercice 6 Première optimisation

L'idée est de profiter du calcul du représentant du groupe auquel appartient un habitant pour compresser le chemin d'indices parcouru. Cette compression permettra d'accélérer les futures recherches impliquant les habitants correspondants à ces indices.

Le principe consiste à mettre à jour chaque case visitée avec le contenu de la future case visitée. Par exemple, dans le tableau illustrant l'exercice 3, le calcul du représentant de 2 conduit à visiter les cases 2, 1 et 0. Lors de ce parcours, il est facile d'affecter à `amis[2]` la valeur de `amis[1]` puis à `amis[1]` celle de `amis[0]`. Il est facile de voir que les recherches devant visiter la case d'indice 2 sont à présent plus rapides.

Exercice 7. Une optimisation plus radicale

Une façon plus coûteuse mais plus efficace en terme de compression consiste à calculer la valeur du représentant puis à affecter cette valeur à toutes les cases impliquées dans la recherche. Donnez le pseudo-code de l'algorithme intégrant cette compression.

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX PRATIQUES – Semaine n°5

Thèmes

- Encodage de donnée
- Implémentation et optimisation d'un algorithme de plus court chemin

Présentation

À partir d'une position donnée dans un labyrinthe, un joueur doit se déplacer pour rejoindre une position d'arrivée. Le jeu est pimenté par le fait qu'il se déroule dans le noir et que le joueur peut de ce fait entrer en collision avec les murs.

Une première implémentation du jeu vous est proposée. Le labyrinthe est considéré comme étant carré. Le joueur indique successivement dans quelle direction il souhaite se déplacer et le programme lui indique si le déplacement a pu se faire ou pas. Le labyrinthe n'est jamais affiché car le joueur est dans le noir. Le programme s'arrête dès que le joueur atteint la position d'arrivée.

Exercice 1. Compréhension du programme fourni

Créez un projet et importez-y le programme fourni (`labyrinthe.c`). Lisez le code source en portant une attention particulière à la définition du type `Cellule` et des constantes `TAILLE`, `terrain`, `NB_DIR` et `dir`.

Corrigez le code de la fonction `estValide`. Cette fonction doit retourner la valeur `0` si les coordonnées passées en paramètre désignent une case en dehors du terrain ou une case contenant un mur. Dans le cas contraire, elle doit retourner une valeur différente de `0`. Actuellement, elle retourne systématiquement la valeur `0` interdisant de fait tout déplacement.

Compilez votre projet et jouez une partie.

Exercice 2. Calcul de la distance minimale - Structure de données

Le programme actuel juge de la qualité du joueur en comparant le nombre de déplacements qu'il a réalisé au nombre minimal requis pour atteindre le point d'arrivée. Pour l'instant, ce nombre minimal est renseigné en dur dans le programme. Notre objectif est de développer une fonction calculant automatiquement sa valeur en fonction du labyrinthe et des points de départ et d'arrivée.

Le principe consiste à calculer pour chaque case accessible sa distance minimale (en nombre de déplacements) par rapport au point de départ.

Initialement, seule la distance du point de départ à lui-même est connue (0 déplacement). L'étape suivante consiste à affecter la distance 1 à toutes les cases voisines du point de départ. L'étape suivante consiste à affecter la distance 2 à toutes les cases voisines de celles identifiées à l'étape précédente et ainsi de suite. Bien entendu, la distance d'une case ne doit être affectée que si sa valeur n'a pas été définie au préalable.

Proposez une structure de données permettant de mémoriser les données mises en œuvre par cet algorithme et donnez sa traduction en C.

Exercice 3. Calcul de la distance minimale - Algorithme

L'algorithme décrit informellement ci-dessus peut être résumé par le pseudo-code suivant :

```
initialiser chaque case comme étant non-visitée à l'exception de la
case de départ pour laquelle la distance est 0
dep ← 0
faire
        pour chaque case dont la distance est égale à dep faire
                pour toute case voisine non encore visitée faire
                        lui associer la distance (dep + 1)
                        la marquer comme étant visitée
                fin pour
        fin pour
        dep ← dep + 1
tant qu'on peut marquer au moins une nouvelle case
si la case d'arrivée a été visitée alors
        retourner sa distance
sinon
        retourner une valeur particulière
```

Implémentez une fonction retournant la distance minimale séparant deux points.

Exercice 4. Mise en œuvre

Changez l'initialisation de la constante `MIN_NB_DEP` par l'invocation de votre fonction. Vérifier que la valeur calculée est la bonne (6). Changez le point de départ pour la coordonnée `(0, TAILLE - 1)`. Vérifier que la valeur calculée est correcte (9).

Exercice 5. Finir au plus tôt

Transformez l'algorithme ci-dessus de manière à ce que le résultat soit retourné dès qu'il est connu.

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX DIRIGES – Semaine n°6

Thèmes

- Fonctions et paramètres
- Domaine de définition d'une fonction
  Préconditions et vérification d'utilisation d'une fonction

A partir de l'exercice 3 : reprise des exercices du TD 5 qui n'ont pas été étudiés en semaine 5.

Exercice 1. Préconditions

Prototypez puis codez chacune des fonctions suivantes. Vous passerez systématiquement la taille du tableau en paramètre. Un tableau de taille nulle est vide. Vous donnerez en commentaire les préconditions précisant le domaine de définition des fonctions et vous vérifierez dans le code l'utilisation de la fonction par les assertions correspondant aux préconditions. Vous introduirez le type booléen `bool`. Codez un `main()` qui initialise un tableau de taille 5 par initialisation explicite et qui appellera chacune des fonctions.

`nieme(…)` : renvoie le n^ième élément d'un tableau d'entiers (élément d'indice (`n-1`)).

`plusGrand(…)` : renvoie le plus grand élément d'un tableau d'entiers.

`indice(…)` : renvoie le premier indice `i` d'un tableau d'entiers tel que `t[i]=x` (où `x` est un entier), `-1` sinon.

`minore(…)` : renvoie TRUE si le réel x minore le tableau d'entiers considéré, FALSE sinon. `x` minore le tableau si `x` est plus petit ou égal à tous ses éléments.

Exercice 2. Date

Soit le type `Date` qui représente une date du calendrier grégorien.

```
typedef struct { int jour, mois, an } Date ;
```

Rappel : dans ce calendrier introduit en France fin 1582, seules les années à partir de 1583 sont considérées. Une année est bissextile si elle est multiple de `4` mais pas de `100` ou alors aussi de `400` (cf. TD1).

Codez les fonctions suivantes où : (1) `j` est le numéro d'un jour du mois `m` considéré, (2) `m` celui d'un mois, (3) `a` est une année du calendrier grégorien, et

(4) `d` représente une donnée de type `Date`. Vous n'oublierez pas de traiter les préconditions liées au domaine de définition des fonctions.

`dateValide(…)` : renvoie TRUE si le jour `j` existe pour le mois `m` de l'année `a`, `FALSE sinon` ;

`initDate(…)` : initialise la date `d` avec le jour `j`, le mois `m` et l'année `a`.

`nbJoursDuMois(…)` : renvoie le nombre de jours du mois `m` de l'année `a`.

Exercice 3. Moyenne pondérée

La note moyenne d'un étudiant est calculée en prenant en compte le coefficient (un réel) de chaque épreuve (comme pour le baccalauréat ou les notes d'UE de votre DUT). Écrivez une fonction calculant la moyenne d'un étudiant. Vous noterez que le type `Notes` peut être employé pour stocker les coefficients des épreuves.

Exercice 4. Correction des copies

Pour simuler la correction des copies, il vous est demandé d'écrire une fonction affectant une note aléatoire à chacune des épreuves. Pour ce faire, les constantes suivantes sont définies :

```
const int NOTE_MIN = 0, NOTE_MAX = 20;
```

Notez que la fonction `rand()` de la bibliothèque standard retourne un nombre aléatoire compris entre `0` et `RAND_MAX` (qui vaut au moins `32767`).

Exercice 5. Modifier une note

Écrire une fonction permettant de modifier une note d'une épreuve donnée (désignée par sa position dans le tableau de note).

Exercice 6. Moyennes par épreuve d'une promotion

Les notes de la promotion sont stockées au moyen de la constante et du type qui suivent :

```
// enum car constante employée pour dimensionner un tableau
enum {NB_ETUDIANTS = 175};

// une note par épreuve par étudiant
typedef Notes TableauDeNotes[NB_ETUDIANTS];
```

Écrivez une fonction qui calcule les moyennes de la promotion à chacune des épreuves.

UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année

IAP – Introduction à l'Algorithmique et à la Programmation

TRAVAUX DIRIGES – Semaine n°7

Thèmes

- Fonctions et paramètres
- Domaine de définition d'une fonction
  Préconditions et vérification d'utilisation d'une fonction

Exercice 1.

(1) Quel est l'affichage résultant de l'extrait de code suivant :

```
int i=2, k=3;
int* p1=&i, *p2=&k;
*p1 -= 4;
*p2 *= 3;
printf("%d %d\n", i, k);
```

(2) Soit le prototype de la fonction `fiche` :

```
void fiche(float *x, float *y, int i, char z, char r ) ;
```

- avec les déclarations suivantes :

```
float a, c ;
int j ;
char b, h ;
```

- et en supposant les variables `a`, `c`, `b`, `h` et `j` correctement initialisées, donnez le numéro de tous les appels corrects de la fonction fiche :

```
1) fiche (a, c; j ; b, h) ;
2) fiche (&a, &b, c, j, h) ;
3) fiche (&a, &c, 3, 'b', b) ;
4) fiche (a; j; b; h) ;
5) fiche (&a, &c, j, b, h) ;
```

Exercice 2.

(1) Documentez et déclarez une fonction (`f`) qui au nombre réel `x` donne pour résultat `1/(2-x)`. Donnez la/les précondition(s) de `f`.

(2) Test unitaire : donner l'assertion qui permet de vérifier que l'appel de la fonction pour `x=1.0` vaut bien `1.0`.

(3) Définissez la fonction. Si besoin, vérifiez l'utilisation de `f`.

(4) Testez la fonction dans un programme principal.

Exercice 3.

Soit le type Date suivant :

```
typedef struct {
    unsigned char jour;
    unsigned char mois;
    unsigned int an;
} Date;
```

Prototypez les fonctions initialiserDate, nbJoursDuMois et DateValide qui respectivement -initialise une date, -retourne le nombre de jour en fonction du mois (entier compris entre 1 et 12) et de l'année et -indique si une date est valide. Les codes ne sont pas demandés. Donnez les éventuelles préconditions des fonctions.
Rappel : une année est bissextile si l'année est divisible par 4 et non divisible par 100 ou si l'année est divisible par 400.

Vous appliquerez les règles d'optimisation pour le prototypage.

Rappel sur l'optimisation : il est usual de passer les paramètres de type structuré/utilisateur (de plus de 4 octets, taille d'une adresse) comme des pointeurs sur le type. De plus, les paramètres d'entrée seront passés comme constant pour empêcher toute affectation dans le corps de la fonction.