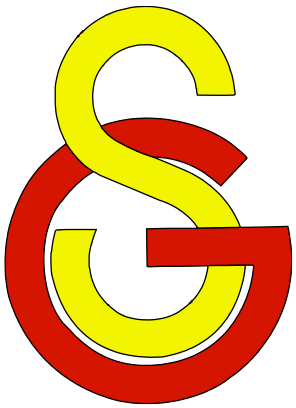
Algorithmique & programmation en langage C - vol.2

Damien Berthet, Vincent Labatut

Université Galatasaray
Faculté d'ingénierie et de technologie

Algorithmique &
programmation en langage C

Damien Berthet & Vincent Labatut



Sujets de travaux pratiques

Supports de cours – Volume 2
Période 2005-2014

 Damien Berthet & Vincent Labatut 2005-2014

Galatasaray Universitesi
Mühendislik ve Teknoloji Fakültesi
Çırağan Cad. No:36
Ortaköy 34349, İstanbul
Turquie

version 1
23/07/2014

Sommaire

SOMMAIRE

1 INFORMATIONS PRATIQUES

1.1 Conventions
1.2 Accès aux ressources

2 INSTALLATION ET CONFIGURATION

2.1 Java
2.1.1 Windows
2.1.2 Linux
2.1.3 Vérification
2.2 Eclipse
2.2.1 Installation
2.2.2 Utilisation
2.3 CDT
2.3.1 Compilateur C
2.3.2 C/C++ Development Tooling
2.3.3 Test de l'installation
2.3.4 Problèmes possibles
2.4 SDL
2.4.1 Installation
2.4.2 Configuration
2.4.3 Test de l'installation
2.4.4 Problèmes possibles

3 UTILISATION EN TP

3.1 Règles concernant les TP
3.1.1 Nommage
3.1.2 Programmation
3.1.3 Texte
3.1.4 Remise du travail
3.2 Création d'un projet
3.2.1 Nom du projet
3.2.2 Structure du projet
3.2.3 Importation de fichiers
3.3 Bibliothèque mathématique
3.4 Compilation
3.5 Exécution
3.5.1 Dans Eclipse
3.5.2 Hors d'Eclipse
3.6 Débogage
3.6.1 Perspective de débogage
3.6.2 Contrôle de l'exécution
3.6.3 Accès à la mémoire

4 SUJETS DE TP

01 Entrées-sorties
02 Types simples
03 Variables & opérateurs
04 Instructions de contrôle
05 Crible d'Érathostène
06 Chaînes de caractères
07 Tableaux multidimensionnels
08 Diagrammes textuels
09 Introduction à la SDL
10 Modification des couleurs
11 Passage de paramètres
12 Algorithme de Bresenham
13 Histogramme des couleurs
14 Propriétés arithmétiques
15 Algorithmes pour l'arithmétique
16 Bibliothèque Chaîne
17 Décomposition d'une permutation
18 Nombres binaires
19 Algorithme de Johnson
20 Manipulation de dates
21 Carrés latins
22 Représentation d'une promotion
23 Partition d'un entier
24 Rotation d'un carré
25 Zoom d'une image
26 Automates finis
27 Championnat de football
28 Floutage d'une image
29 Flous avancés
30 Gestion d'un lexique
31 Allocation dynamique
32 Générateur pseudo-aléatoire
33 Nombres hexadécimaux
34 Agenda téléphonique
35 Fichiers et arguments de programme
36 Diaporama
37 Stock d'une librairie
38 Automates cellulaires
39 Fonctions récursives
40 Approximations numériques
41 Figures fractales
42 Listes chaînées
43 Disques & guirlandes
44 Listes de caractères
45 Tapis de Sierpiński
46 Suite de Syracuse
47 Enveloppe d'un nuage de points
48 Marches de Graham & Jarvis
49 Analyse d'expressions
50 Conversion d'expressions
51 Détection de palindromes
52 Tours de Hanoï
53 Remplissage de formes
54 Parcours d'un labyrinthe
55 Génération d'un labyrinthe
56 Tri par dénombrement
57 Tri Cocktail
58 Représentation des tris
59 Tris sur listes
60 Représentation de l'ADN
61 Nombres de grande taille
62 Table de symboles
63 Plus longue sous-séquence commune
64 Arbres binaires

Ce document regroupe 64 sujets de travaux pratiques (TP) et d'examen, écrits pour différents enseignements d'algorithmique et de programmation en langage C donnés à la Faculté d'ingénierie de l'Université Galatasaray (Istanbul, Turquie), entre 2005 et 2014. Il s'agit du deuxième volume d'une série de 3 documents, comprenant également le support de cours (volume 1) et un recueil des corrigés de ces sujets (volume 3).

Les sujets d'examen ont été retravaillés pour prendre la forme de sujets de TP. Les 64 sujets proposés ont été ordonnés de manière à correspondre à la progression des concepts parallèlement étudiés en cours (cf. le volume 1). En ce qui concerne les concepts les plus simples, il est difficile de sortir des exercices assez classiques, d'autant plus que les étudiants ne disposent à ce moment-là du cours que d'un bagage technique très réduit. Cependant, nous avons tenté d'aborder des thèmes plus originaux dans les sujets venant plus tard. Nous nous sommes particulièrement attachés à proposer des exercices basés sur une approche graphique de l'algorithmique, grâce à l'utilisation de la bibliothèque SDL[1] (*Simple DirectMedia Layer*).

Le volume horaire d'un (ou même de deux) cours classique(s) ne permet bien entendu pas d'effectuer tous les TP proposés ici. Il faut remarquer que si certains sujets introduisent un concept nouveau, d'autres, au contraire, se concentrent sur l'approfondissement d'une ou plusieurs notions déjà utilisées. L'idée est plutôt, pour l'enseignant, de disposer d'un assortiment d'exercices divers, dans lequel il peut choisir ce dont il a besoin, en fonction des étudiants, de la progression effective et des objectifs de son cours. Pour les étudiants, il s'agit de proposer des exercices pouvant offrir une vision alternative à celle donnée dans le cours suivi, ou bien d'approfondir certains points vus en cours.

Malgré tout le soin apporté à la rédaction de ces sujets, il est probable que des erreurs s'y soient glissées. Merci de nous contacter afin de nous indiquer tout problème détecté dans ce document. Il faut également remarquer qu'il s'agit de TP donnés dans le cadre d'un cours d'introduction, aussi les notions abordées le sont parfois de façon simplifiée et/ou incomplète.

Le reste du document est organisé de la façon suivante. Nous présentons d'abord les conventions adoptées dans les sujets. Puis, nous décrivons les outils utilisés au cours des TP, à savoir GCC[2] pour la compilation, Eclipse[3] pour l'environnement de programmation, et la SDL pour la manipulation des graphismes. Nous expliquons comment les installer et donnons les rudiments de leur utilisation. Nous donnons ensuite une liste synthétique des sujets, en précisant notamment comment ils utilisent les différentes notions vues en cours. Enfin, le cœur du document est constitué des sujets de TP eux-mêmes.

Damien Berthet & Vincent Labatut
le 7 juillet 2014

[1] https://www.libsdl.org/
[2] http://gcc.gnu.org/
[3] http://www.eclipse.org/

1 Informations pratiques

1.1 Conventions

Afin de faciliter la compréhension des sujets, certaines conventions de mise en forme sont systématiquement employées dans les trois volumes.

Tout d'abord, les identificateurs (noms de fonctions, variables, constantes, etc.) sont indiqués en utilisant la police `Courier`.

La plupart des exercices consistent à écrire une fonction implémentant un traitement demandé. Nous essayons, dans la mesure du possible, de toujours donner un exemple de ce traitement. Quand celui-ci implique une interaction avec l'utilisateur (affichage ou saisie), le contenu de la console est représenté en utilisant `Courier` sur fond bleu. Lorsque l'utilisateur doit saisir des valeurs, celles-ci sont surlignées en jaune.

```
Entrez une valeur : 12
Vous avez entré la valeur 12.
```

Lorsque du code source est cité, nous employons un fond rouge. Si une partie du code source en particulier doit être désignée, elle est surlignée en vert.

```
int ma_function(int x)
{   int une_variable_notable;
    ...
```

1.2 Accès aux ressources

Certains sujets nécessitent d'utiliser/compléter des bibliothèques, ou bien d'accéder à des images. Ces données additionnelles sont disponibles à partir de la page du cours sur Kikencere[4]. La remise du travail se fait également sur Kikencere.

[4] http://kikencere.gsu.edu.tr/

2 Installation et configuration

Dans cette section, nous expliquons brièvement comment installer et configurer les différents outils utilisés en TP, à savoir :

- L'environnement d'exécution de Java, qui est nécessaire pour utiliser Eclipse ;
- Le logiciel Eclipse, qui sera notre environnement de programmation ;
- Un compilateur capable de traiter des programmes écrits en langage C ;
- La bibliothèque SDL que nous utiliserons pour dessiner à l'écran.

Notez que tous ces outils sont libres d'utilisation, et multiplateforme. Ils sont disponibles au moins pour les systèmes Windows, Unix/Linux et Apple. Cependant, dans nos explications, nous ne traitons que les deux premiers.

Chaque étape décrite ici suppose que l'étape précédente a été réalisée avec succès. Inutile de passer à l'étape suivante tant que l'étape courante n'est pas validée. Par exemple, ne tentez pas d'installer Eclipse tant que vous ne réussissez pas à faire fonctionner Java d'abord.

2.1 Java

Java est à la fois un langage de programmation et un environnement d'exécution, développés par Sun Microsystems à partir de 1995, puis Oracle depuis 2010. Le langage est orienté objet, et l'environnement d'exécution permet de créer des programmes de façon relativement indépendante de la plateforme.

2.1.1 Windows

- Allez sur le site de Sun Microsystems consacré à Java :
  http://java.sun.com/javase/downloads/index.jsp
- Téléchargez le dernier JDK (*Java Standard Edition Development Kit*) version SE (*Standard Edition*) pour Windows.
- Installez le JDK en exécutant le programme que vous venez de télécharger.
- On notera dorénavant `<Java>` le dossier d'installation du JDK.

2.1.2 Linux

- La plupart du temps, Java est déjà installé sous Linux (cela dépend de la distribution que vous utilisez).
- Vérifiez que :
  - C'est bien le JDK entier qui est installé, et non pas seulement le JRE (*Java Runtime Environment*).
  - Il s'agit bien de la dernière version disponible du JDK.
- Dans le cas contraire, vous devez installer le dernier JDK.
- **Remarque :** vous aurez généralement besoin d'être connecté en tant qu'administrateur.
- La procédure d'installation dépend de la distribution Linux que vous utilisez :
  - Ubuntu : vous pouvez utiliser le gestionnaire de packages Synaptic.
  - Fedora : vous pouvez utiliser le gestionnaire de packages Yellow Dog.
- Si vous ne pouvez pas utiliser de gestionnaire de packages, il faut réaliser la procédure manuellement :
  - Allez sur le site de Sun Microsystems consacré à Java :
    http://java.sun.com/javase/downloads/index.jsp
  - Téléchargez le dernier JDK (*Java Standard Edition Development Kit*) version SE (*Standard Edition*) pour Linux.
  - Déplacez l'archive auto-extractible dans le dossier où vous voulez installer Java, par exemple : `/usr/local`.
  - On notera dorénavant `<Java>` ce dossier d'installation.
  - Ouvrez un terminal OS (également appelé *console*) et rendez-vous dans `<Java>`, puis entrez les commandes :

```
> chmod +x jdk-<version>-linux-i586.bin
> ./jdk-<version>-linux-i586.bin
```

  - Vous devez bien sûr remplacer `<version>` par la version du JDK que vous êtes en train d'installer.
  - La licence d'utilisation va être affichée, tapez `yes` à la fin pour l'accepter.
  - Pour terminer l'installation, vous devez ensuite modifier et créer certaines variables d'environnement :

- Ouvrez votre fichier profil situé dans le dossier `$HOME`[5] : il s'agit généralement d'un fichier appelé `.profile` ou `.bashrc`.
- À la fin de ce fichier, rajoutez les lignes suivantes (en remplaçant les éléments entre `'<'` et `'>'` par les valeurs réelles) :

```
JAVA_HOME=<Java>/jdk<version>
export JAVA_HOME
CLASSPATH=.
export CLASSPATH
PATH=$JAVA_HOME/bin:$PATH
export PATH
```

- Enregistrez le fichier, déconnectez-vous, et connectez-vous sur le compte dont vous venez de modifier le profil.

2.1.3 Vérification

- Vérifiez que Java est bien installé :
  - Ouvrez un terminal OS.
  - Entrez la commande :

```
> java -version
```

  - Vous devez obtenir une réponse indiquant la version du JRE.
  - Entrez ensuite la commande :

```
> javac -version
```

  - Vous devez obtenir une réponse indiquant la version du JDK.

[5] Rappel : sous Unix ou Linux, `$HOME` représente le dossier personnel de l'utilisateur, i.e. : `/home/<utilisateur>` où `<utilisateur>` est l'identifiant de l'utilisateur.

2.2 Eclipse

Eclipse est un environnement de développement intégré (IDE) libre (open source), développé au sein d'une fondation rassemblant des acteurs importants du domaine, que ce soit des entreprises (IBM, Intel, Nokia, Google…) ou des institutions (universités, associations de promotion de l'open source…).

Initialement, Eclipse permet de programmer en Java, mais il est possible d'installer des extensions permettant d'utiliser d'autres langages.

Remarque : avant d'installer Eclipse, il est absolument nécessaire que vous vous assuriez que Java est correctement installé (cf. la section 2.1, décrivant l'installation de Java).

2.2.1 Installation

2.2.1.1 Windows

- Connectez-vous au site officiel d'Eclipse : http://www.eclipse.org.
- Téléchargez le dernier SDK (*Standard Development Kit*) version classique pour Windows.
- **Attention :** pour éviter d'éventuels problèmes, vous devez installer Eclipse dans un dossier dont le chemin ne comporte pas de caractère espace (`' '`).
- Créez un dossier `c:\Eclipse` qui sera utilisé pour l'installation d'Eclipse.
- Ce dossier sera dorénavant désigné sous le nom `<Eclipse>`.
- Décompressez l'archive dans le dossier `<Eclipse>`, de manière à ce que l'exécutable `eclipse.exe` se trouve directement dans `<Eclipse>`.

2.2.1.2 Linux

- Comme pour Java, certaines distributions (Ubuntu par exemple) incluent Eclipse :
  - Il suffit alors d'utiliser le gestionnaire de package (Synaptic ou Yellow Dog) pour installer Eclipse.
  - Vérifiez bien qu'il s'agit de la dernière version d'Eclipse disponible.
- Si vous ne disposez pas de gestionnaire de package :
  - Connectez-vous au site officiel d'Eclipse : http://www.eclipse.org.
  - Téléchargez le dernier SDK[1] (*Standard Development Kit*) version classique pour Linux.
  - Créez un dossier `eclipse` dans votre dossier personnel `$HOME`[6].
  - Ce dossier d'installation d'Eclipse `$HOME/eclipse` sera dorénavant désigné sous le nom `<Eclipse>`.
  - Décompressez l'archive dans le dossier `<Eclipse>`, de manière à ce que l'exécutable `eclipse` se trouve directement dans `<Eclipse>`.
- Par votre confort d'utilisation, vous pouvez créer un raccourci qui vous permettra de lancer facilement Eclipse. Par exemple, sous la GUI Gnome :
  - Faites un clic-droit sur le menu horizontal situé tout en haut du bureau.
  - Choisissez *Add to panel*.
  - Sélectionnez *Custom application launcher* puis cliquez sur *Add*.
  - Remplissez les champs :

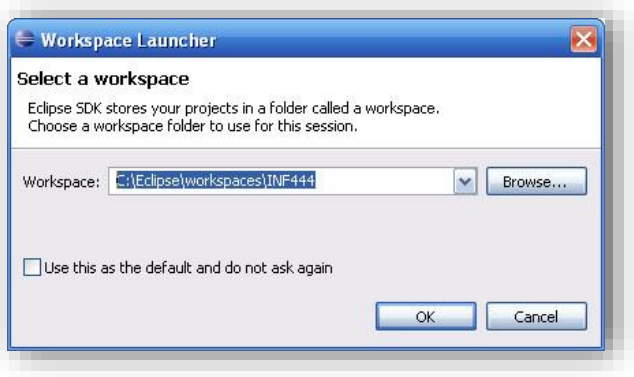
[6] Cf. l'installation de Java, section 0.

- *Name* : Eclipse
- *Command* : `<Eclipse>/eclipse`
- *Icon* : `<Eclipse>/icon.xpm`

2.2.2 Utilisation

2.2.2.1 Espace de travail

- Un espace de travail est un dossier destiné à contenir tous vos programmes concernant un projet ou un ensemble de projets liés. Un même utilisateur a en général besoin de plusieurs espaces de travail différents.
- Créez un dossier `<Eclipse>/workspaces` qui contiendra tous vos espaces de travail.
- **Attention :** le chemin de ce dossier ne doit surtout pas contenir de caractères espace (`' '`).
- Lancez Eclipse grâce à l'exécutable situé dans `<Eclipse>` ou au raccourci que vous avez créé.
- Au démarrage, Eclipse vous demande d'indiquer votre *Workspace* (espace de travail) : indiquez `<Eclipse>/workspaces/INFxxx` où `INFxxx` désigne le cours concerné (par exemple `INF202` pour le cours d'algorithmique et programmation II).

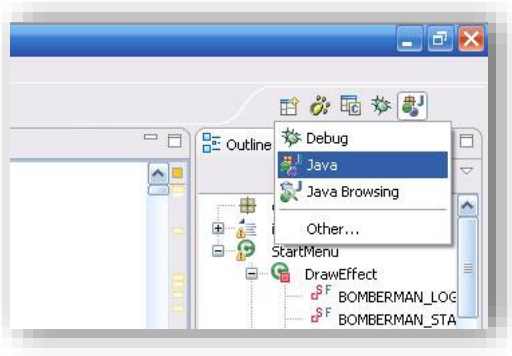


- **Attention :**
  - Si vous cochez la case *Use this as the default and do not ask again*, Eclipse ne vous demandera plus de choisir votre espace de travail au démarrage.
  - Il ne faut donc **surtout pas** cocher cette case.
  - Si vous cochez accidentellement cette case, vous pouvez rétablir l'option dans Eclipse en allant dans *Windows>Preferences>General>Startup and Shutdown>Prompt for workspace at startup*.
- Il est possible de changer l'espace de travail courant depuis Eclipse en utilisant *File>Switch Workspace*.

2.2.2.2 Perspective

- Eclipse est décomposé en différentes fenêtres qui offrent l'accès à différentes informations. Par défaut, on peut observer :
  - Le code source au centre.
  - Le navigateur de projet à gauche.
  - Le navigateur de classe à droite.
  - La console et différentes informations en bas.

- On appelle perspective la façon dont les différentes fenêtres d'Eclipse sont organisées. La perspective par défaut est la perspective Java, adaptée à ce langage de programmation.
- Mais Eclipse est un environnement ouvert, et il est possible de manipuler d'autres langages, en installant au préalable des extensions téléchargeables. Une Perspective différente peut être utilisée pour chaque langage.
- Mais il est également possible de changer de perspective en fonction de son activité. Par exemple, la perspective par défaut est une perspective dédiée à la programmation. Pour passer à une perspective dédiée au déboguage, il suffit d'utiliser le menu situé en haut à droite de la fenêtre principale.



2.2.2.3 Autres fonctionnalités

- Pour avoir un aperçu des différentes fonctionnalités d'Eclipse, référez-vous aux nombreux tutoriels disponibles sur le Web, comme par exemple :
  - http://help.eclipse.org/help32/index.jsp
  - http://eclipsetutorial.forge.os4os.org/in2.htm
  - http://www.eclipsetotale.com/articles/premierPas.html
  - http://jmdoudoux.developpez.com/java/eclipse/

2.3 CDT

CDT (*C/C++ Development Tooling*) est une extension pour Eclipse développée par la fondation Eclipse. Il s'agit d'un outil libre, mais professionnel, utilisé dans l'industrie. Pour l'utiliser, il faut disposer d'Eclipse et d'un compilateur C.

2.3.1 Compilateur C

Le compilateur C que nous allons utiliser pour programmer en C est gcc, qui a été développé pour Unix/Linux par GNU. Généralement, sous Linux ce compilateur est installé par défaut. Si ce n'est pas le cas, utilisez votre gestionnaire de package (par exemple Synaptic pour Ubuntu) pour l'installer.

2.3.1.1 Téléchargement

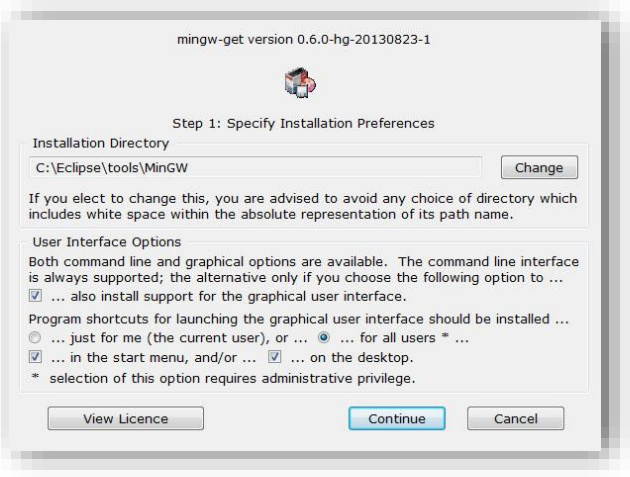
Sous Windows, il est nécessaire d'installer MinGW (collection d'outils permettant d'utiliser sous Windows des logiciels originalement développés pour Unix/Linux), MSys (complétant MinGW) et gdb (chargé du débogage). Les outils MSys et gdb sont maintenant inclus dans le package de MinGW, donc vous aurez seulement besoin de télécharger MinGW.

Pour cela :

- Allez sur la page Sourceforge de MinGW, qui est située à l'adresse http://sourceforge.net/project/showfiles.php?group_id=2435
- Téléchargez la dernière version stable.

2.3.1.2 Installation de MinGW

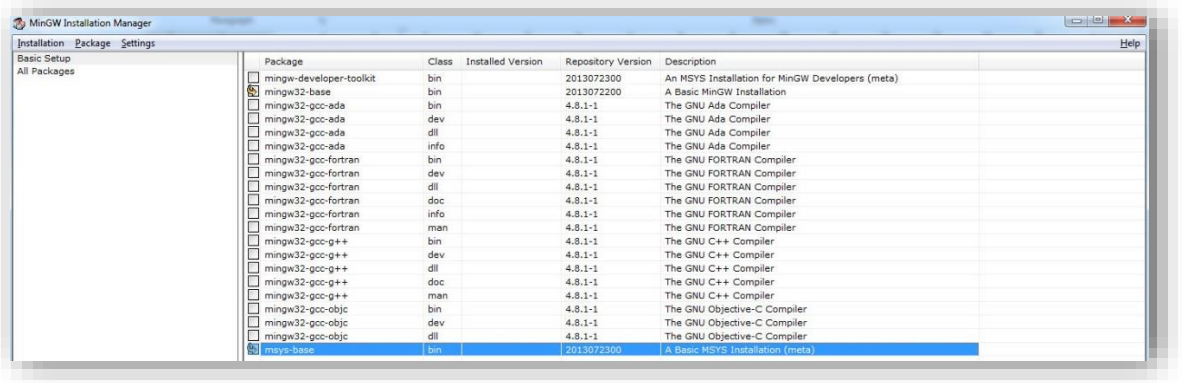
- Déclenchez l'installation de MinGW, de manière à faire apparaître la fenêtre suivante :



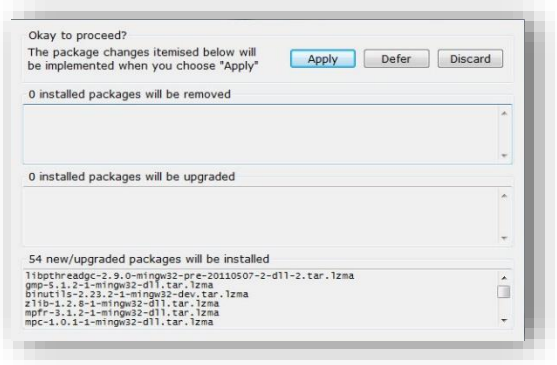
- Indiquez un dossier d'installation, de préférence dans Eclipse : `<Eclipse>`[7]`\tools\MinGW`. Ce dossier sera dorénavant noté `<MinGW>`.
- Pour les autres options, vous pouvez garder les valeurs par défaut. Cliquez ensuite sur *Continue* : le programme va charger les données nécessaires à l'installation.
- Une fois ce chargement terminé, cliquez à nouveau sur *Continue*.

[7] Dossier d'installation d'Eclipse, cf. la section 2.2 sur l'installation d'Eclipse.

- Un gestionnaire d'installation similaire à celui-ci-dessous va alors s'ouvrir. Sélectionnez uniquement les packages suivants (les autres sont inutiles pour nos TP de C) :
    - *mingw32-base* : compilateur et déboguer ;
    - *msys-base* : MSys.



- Cliquez sur *Installation* > *Apply Changes*, et la fenêtre suivante va s'ouvrir :

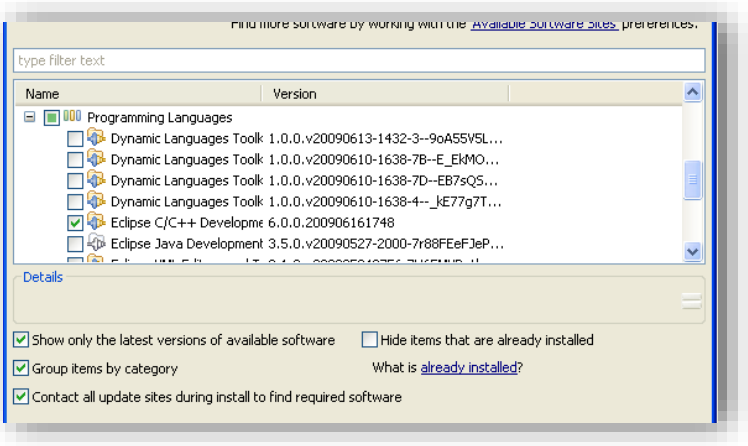


- Cliquez sur le bouton *Apply* pour poursuivre l'installation.
- Fermez la fenêtre une fois l'installation terminée, puis le gestionnaire d'installation.

2.3.2 C/C++ Development Tooling

2.3.2.1 Installation

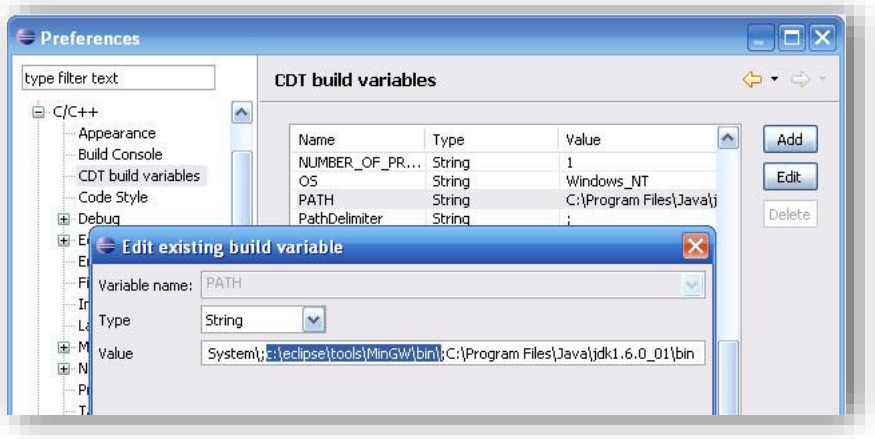
- Pour installer l'extension CDT (sous Windows et Linux), lancez d'abord Eclipse.
- **Remarque :** on suppose dans ce document que vous avez installé la toute dernière version d'Eclipse.
- Cliquez sur *Help*> *Install New Software*.
    - Choisissez tous les sites disponibles
    - Cochez *Programming languages* puis *C/C++ Development*
    - Cliquez sur *Next*



- Cliquez à nouveau sur *Next*.
- Acceptez la licence puis cliquez sur *Finish*.
- Eclipse va télécharger l'extension, puis l'installer (il vous sera éventuellement demandé de valider l'installation).
- Après l'installation de CDT, il est possible que vous deviez redémarrer Eclipse.

2.3.2.2 Configuration

- Après le redémarrage d'Eclipse, allez dans *Window>Preferences* :
  - Enregistrement automatique à la compilation : dans *C/C++>New CDT project wizard>Makefile project>Behaviour*, cochez la case *Build on resource save (Auto build)*.
  - Désactiver le correcteur orthographique : dans *General>Editors>Text Editors>Spelling*, décochez *Enabled spell checking*.
- La dernière étape concerne seulement les utilisateurs de Windows, elle consiste à vérifier que les chemins de MinGW, MSys et gdb sont bien présents dans la variable d'environnement `PATH` :
  - Allez dans *Window > Preferences > C/C++ > Build > Build variables* et cliquez sur *Show system variables* (en bas à gauche).
  - Cherchez la variable `PATH` et cliquez sur *Edit*.



  - Dans *Value*, vérifiez que les chemins `<MinGW>\bin` et `<MinGW>\msys\1.0\bin` sont biens présents (sinon rajoutez-les).
  - Cliquez sur *OK* (deux fois).

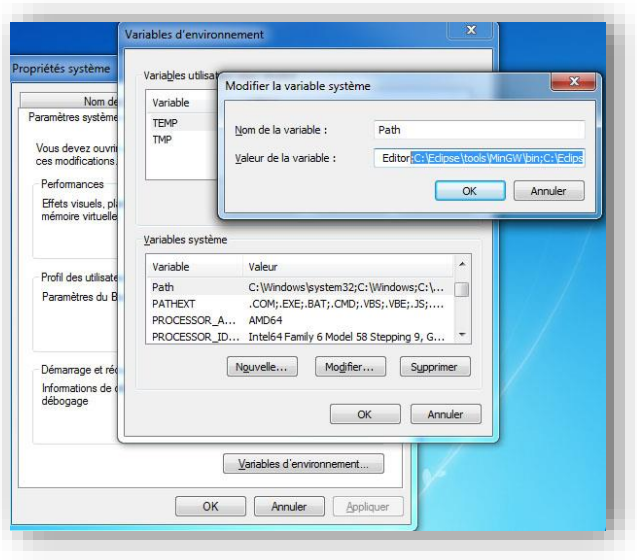
2.3.3 Test de l'installation

Pour vérifier si la CDT a été bien installée, créez, compilez et exécutez un programme C :

- Placez-vous dans la perspective C/C++
- Cliquez sur *File* > *New* > *C Project* (et non pas *C++ Project*, attention !)
- Une fenêtre apparait :
  - *Project name* : donnez un nom à votre projet (par exemple : `Test`).
  - *Project type* : sélectionnez *Executable* > *Hello World ANSI C Project*.
  - Cliquez sur le bouton *Finish*.
- Compilez le projet : *Project* > *Build Project*. En cas de problème, un message d'erreur apparaîtra dans la console d'Eclipse.
- Exécutez le programme : faites un clic-droit sur le projet (dans *Project Explorer*), puis *Run As* > *Local C/C++ Application*.
- Le texte `!!!Hello World!!!` Devrait apparaître dans la console. Sinon, c'est qu'il y a un problème d'installation.

2.3.4 Problèmes possibles

- `gcc not found` ou `make not found` :
  - Ce message est affiché lorsque le chemin vers MinGW et MSys n'est pas correctement configuré : vérifiez donc d'abord qu'il est valide.
  - Certaines versions d'Eclipse ont aussi des problèmes avec les chemins définis en interne. Il est alors possible de les définir en externe, i.e. dans Windows :
    - Ouvrez les propriétés du système :
      - Windows XP (et versions plus anciennes) :
        - Sur le bureau, faites un clic-droit sur le *Poste de travail*.
        - Cliquez sur *Propriétés*, puis sur l'onglet *Avancé*.
      - Windows Vista (et versions plus récentes) :
        - Allez dans le menu *Démarrer*, puis ouvrez le *Panneau de configuration*.
        - Cliquez sur *Système*, puis *Paramètres système avancés* (lien situé à gauche).



- La fenêtre *Propriétés système* ci-dessus apparait alors.

        - Cliquez sur le bouton *Variables d'environnement* : une nouvelle fenêtre judicieusement appelée *Variables d'environnement* va alors apparaître.
        - Dans la partie *Variables système*, sélectionnez `Path` puis cliquez sur le bouton *Modifier…*
        - Rajoutez les deux chemins manquants (cf. section 2.3.2.2), à la fin des chemins déjà présents, en les séparant par des points-virgules (`;`).
        - Cliquez 3 fois sur *OK* pour fermer ces fenêtres.
        - Redémarrez Windows pour appliquer les modifications.
- `Unresolved inclusion` :
    - Parfois, il est possible qu'Eclipse affiche cet avertissement quand vous incluez des bibliothèques fréquemment utilisées. Cela n'empêche pas l'exécution du programme, mais si vous voulez quand même supprimer l'avertissement :
        - Allez dans *Project>Properties>C/C++ General>Paths and Symbols*.
        - Ajoutez le chemin `<MinGW>\include` à *Include directories.*
- Erreur de compilation sans message associé :
    - Eclipse affiche parfois des erreurs de compilation au niveau du projet (par opposition aux fichiers contenus dans le projet).
    - Ces erreurs persistent lorsqu'on rafraichit le dossier et qu'on recompile.
    - Aucun message d'erreur n'est associé dans la console.
    - Vous pouvez alors tenter de faire un clic-droit sur le projet dans l'explorateur de dossier, puis *Index>Rebuild.*

2.4 SDL

Simple Directmedia Layer (SDL) est une bibliothèque opensource multiplateforme (Windows, Linux, MacOS…) permettant d'accéder à des fonctionnalités multimédia (graphismes, sons, contrôle…) depuis différents langages de programmation, et en particulier depuis le langage C. Dans le cadre de ce cours, nous utiliserons la SDL uniquement pour ses fonctionnalités graphiques. Ce document explique la marche à suivre pour utiliser la SDL avec Eclipse et CDT.

Remarque : on suppose ici que vous avez respecté la procédure des sections précédentes concernant l'installation de Java (2.1), Eclipse (2.2) et CDT (2.3).

2.4.1 Installation

2.4.1.1 Windows

- Allez à l'adresse http://www.libsdl.org, puis cliquez sur la version 1.2 dans la section *Downloads*.
- Dans *Development Libraries*, téléchargez la dernière version pour MinGW (pour Windows).
- Décompressez l'archive `SDL`, et copiez les dossiers `SDL\bin`, `SDL\include`, et `SDL\lib` dans le dossier `<MinGW>`[8].
- **Remarque :** attention de ne pas confondre `<MinGW>` et `<MinGW>\mingw32`.

2.4.1.2 Linux

- La méthode d'installation dépend de la distribution que vous utilisez, mais en règle générale il faudra disposer des droits d'administrateur.
- Sous Ubuntu, la SDL est disponible à partir du gestionnaire de packages *Synaptic*.
- Sous Fedora, vous pouvez utiliser Yellow Dog Updater, en tapant dans une console OS :

```
> yum install SDL-devel SDL_mixer-devel SDL_image-devel SDL_ttf-devel
```

- Pour les autres distributions, vous pouvez consulter la page suivante, qui contient quelques tutoriels concernant la SDL :
  http://lazyfoo.net/SDL_tutorials/lesson01/linux/index.php

2.4.2 Configuration

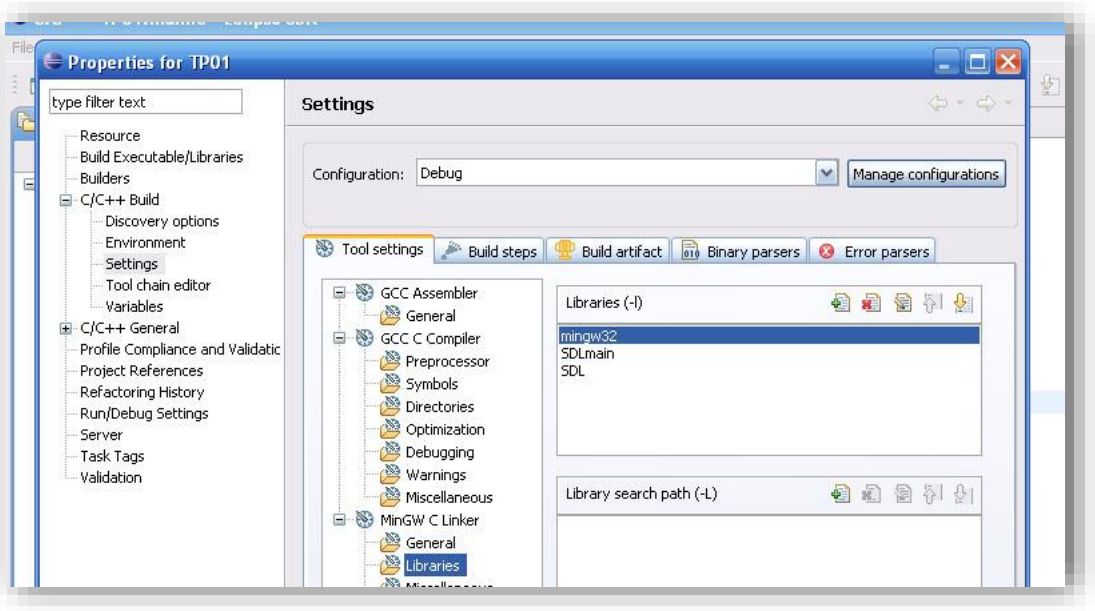
Par rapport à un projet textuel (n'utilisant pas la SDL), il est nécessaire d'effectuer certains réglages supplémentaires afin que le compilateur trouve la SDL. Ces réglages dépendent du système d'exploitation que vous utilisez.

2.4.2.1 Windows

- Créez tout d'abord un nouveau projet C.
- Dans les propriétés du projet, allez dans *C/C++ Build>Settings>Tool settings>MinGW C Linker>Libraries*.

[8] Dossier d'installation de MinGW, cf. la section 2.3 sur l'installation de CDT.

- Dans *Libraries (-l)* : rajoutez (en respectant impérativement cet ordre) les bibliothèques (simplement en entrant leurs noms) : `mingw32`, `SDLmain` et `SDL`.



2.4.2.2 Linux

- Dans les propriétés du projet, allez dans *C/C++ Build>Settings>Tool settings>GCC Linker>Libraries*.
- Dans *Libraries (-l)* : rajoutez `SDL` (simplement en entrant ce nom).

2.4.3 Test de l'installation

Afin de tester votre installation de la SDL, effectuez les actions suivantes :

- Assurez-vous que la CDT est correctement configurée, et qu'il est possible de compiler et d'exécuter un projet C classique (i.e. n'utilisant pas la SDL).
- **Remarque :** si vous ne savez pas comment faire, consultez la section 2.3 sur l'installation de la CDT.
- Créez un nouveau projet appelé `TestSDL`.
- Dans ce projet, créez un fichier `main.c`.
- Dans ce fichier, copiez-collez le code source suivant :

```
#include <SDL\SDL.h>

int main(int argc, char** argv)
{  // cree la fenetre graphique
   SDL_Init(SDL_INIT_VIDEO);
   atexit(SDL_Quit);
   SDL_SetVideoMode(200, 200, 32, SDL_SWSURFACE);
   SDL_WM_SetCaption("Test SDL", NULL);

   // attend une touche pour quitter
   SDL_Event event;
   do
   {  SDL_WaitEvent(&event);
   }
   while(event.type !=  SDL_QUIT && event.type != SDL_KEYDOWN);

   return 0;
}
```

- **Remarque :** pour Linux, la première ligne doit contenir un slash (`/`) à la place du backslash (`\`) utilisé pour Windows.

- Compilez et exécutez le projet.
- Une petite fenêtre noire devrait alors apparaître.
- Appuyez sur une touche pour la refermer.
- Si le programme ne compile pas, ou si la fenêtre n'apparait pas, c'est qu'il y a un problème soit dans l'installation de la SDL, soit dans la configuration de votre projet (en supposant que Java, Eclipse et la CDT soient correctement installés).

2.4.4 Problèmes possibles

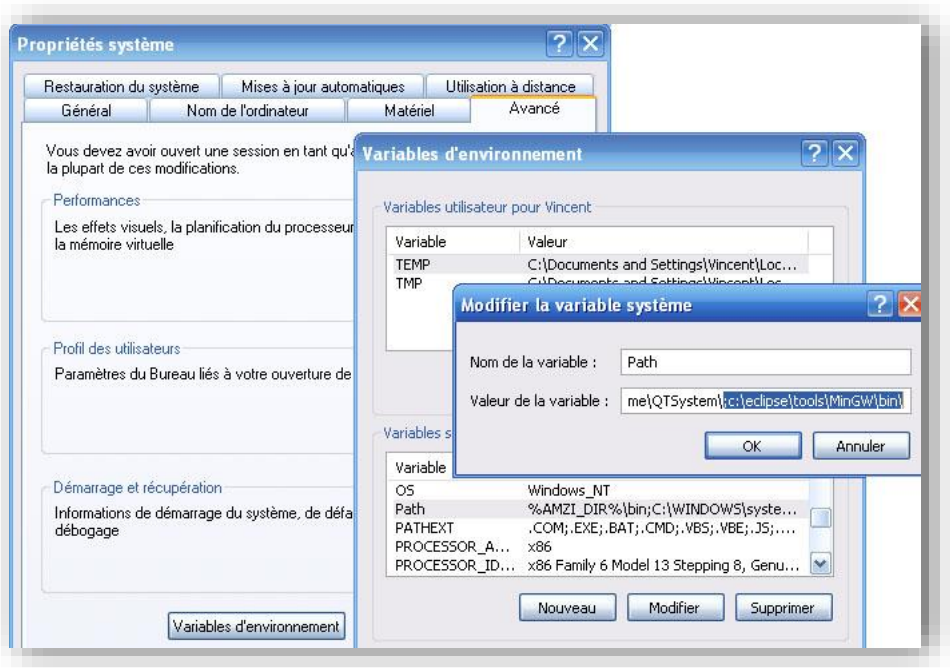
En fonction de votre configuration, il est possible que vous rencontriez différentes erreurs.

2.4.4.1 Erreur de compilation

- Le compilateur n'accède pas aux fichiers de la SDL.
- Concrètement : dans le fichier qui importe la SDL, la ligne d'importation est soulignée en rouge.
- Vérifiez que le caractère séparant `SDL` et `sdl.h` dans la ligne d'importation est bien :
  - pour Linux : `'/'` (slash).
  - pour Windows : `'\'` (backslash).
- Si ce n'est pas la cause de l'erreur, assurez-vous que vous avez **scrupuleusement** respecté la procédure d'installation indiquée dans ce document : le problème vient sûrement d'un chemin (path) erroné.

2.4.4.2 Erreur d'exécution

- Votre projet compile sans erreurs et la construction se termine normalement.
- Mais quand vous l'exécutez :
  - sous Eclipse : rien ne se produit.
  - sous la console de l'OS : vous obtenez l'erreur "`SDL.dll` est introuvable".
- La cause de ce problème peut être une mauvaise configuration de la variable d'environnement `PATH` de votre OS. Pour Windows :
  - Allez dans les *Propriétés du système* (*Panneau de configuration*>*Système*).
  - Sélectionnez l'onglet *Avancé* puis cliquez sur *Variables d'environnement*.
  - Dans *Variables Système*, sélectionnez `Path` et cliquez sur *Modifier*.
  - Après les valeurs déjà présentes, rajoutez (sans aucun espace `' '` et sans oublier le point-virgule `';'`) : `;<MinGW>\bin\`.
  - Remarque : vérifiez d'abord si ce chemin n'est pas déjà inclus dans `Path` : il ne sert à rien de l'ajouter une deuxième fois.


Propriétés système
Restauration du système
Mises à jour automatiques
Utilisation à distance
Général
Nom de l'ordinateur
Matériel
Avancé
Performances
Les effets visuels, la planification du processeur
la mémoire virtuelle
Profil des utilisateurs
Paramètres du Bureau liés à votre ouverture de
Démarrage et récupération
Informations de démarrage du système, de défa
débogage
Variables d'environnement
Variables d'environnement
Variables utilisateur pour Vincent
Variable
Valeur
TEMP
C:\Documents and Settings\Vincent\Loc...
TMP
Modifier la variable système
Nom de la variable :
Path
Valeur de la variable :
me\QTSystem\;c:\eclipse\tools\MinGW\bin\
OK
Annuler
Variable
OS
Windows_NT
Path
%AMZI_DIR%\bin;C:\WINDOWS\syste...
PATHEXT
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...
x86
PROCESSOR_ID...
x86 Family 6 Model 13 Stepping 8, Genu...
Nouveau
Modifier
Supprimer

3 Utilisation en TP

3.1 Règles concernant les TP

- Cette section contient un ensemble de règles valables pour tous les TP et pour tous les examens.
- Si vous ne respectez pas ces règles, votre travail ne sera pas corrigé et vous aurez la note zéro.
- Ces règles sont toujours valables, sauf si le sujet ou l'exercice le précise explicitement.

3.1.1 Nommage

- Pour un TP, vous devez impérativement donner à votre projet un nom de la forme suivante : `TPXX_NomPrénom`.
  - `XX` représente le numéro du TP, par exemple : `TP01` pour le premier TP (et non pas `TP1`).
  - `Nom` et `Prénom` représentent respectivement vos nom de famille et prénom. Vous ne devez surtout pas utiliser de caractères turcs ni français : pas d'accent, pas de cédille, etc. Par exemple, pour Can Emreoğlu, ça sera `EmreogluCan`.
  - Donc pour son premier TP, Can Emreoğlu utilisera le nom `TP01_EmreogluCan`.
- Pour l'examen de TP, vous devez donner à votre projet un nom de la forme `NomPrénom`.
- Quand des identificateurs (noms des fonctions, paramètres, constantes…) sont donnés dans le sujet, vous devez utiliser les mêmes noms dans vos programmes.

3.1.2 Programmation

- Votre code source doit être convenablement formaté et indenté. Eclipse permet de le faire automatiquement en faisant un clic-droit dans le code source et en sélectionnant *Source>Format*.
- Votre travail ne doit contenir aucune erreur de compilation.
- Vos fonctions doivent être placées :
  - Si rien n'est précisé : dans le fichier principal `main.c`.
  - Si le sujet le précise, dans une bibliothèque particulière.
  - Dans les deux cas, les fonction doivent être placées dans l'ordre des exercices du TP : d'abord la fonction de l'exercice 1, puis celle de l'exercice 2, etc., et la fonction `main` à la fin du programme.
- Votre fonction `main` doit contenir le code nécessaire au test de chacune des autres fonctions que vous avez écrites :
  - Chaque fonction doit être appelée avec des paramètres que vous aurez déterminés à l'avance.
  - Quand vous avez terminé et testé la fonction, vous devez placer l'appel ou (les appels) en commentaires avant de passer à la fonction suivante.
  - *exemple* : `maFonction1` est terminée, `maFonction2` est en cours de développement :

```
int maFonction1(int parametre)
{  // traitement de la fonction
}
```

```
int maFonction2(char parametre1, int parametre2)
{  // traitement de la fonction
}

int main()
{  int x,resultat;
   char c;
/*
   // maFonction1
   x = 0;
   temp = maFonction1(x);
   printf("maFonction1(%d)=%d\n",x,resultat);
   x = -1;
   temp = maFonction1(x);
   printf("maFonction1(%d)=%d\n",x,resultat);
   x = 255;
   temp = maFonction1(x);
   printf("maFonction1(%d)=%d\n",x,resultat);
*/

   // maFonction2
   ...
}
```

    - Le but est que les correcteurs puissent facilement tester votre travail.
- Les bibliothèques (listes, piles, files, etc.) éventuellement fournies avec un projet ne doivent pas être modifiées.

3.1.3 Texte

- Si la réponse à un exercice prend la forme d'un texte (et non pas d'un programme) :
    - Vous devez écrire cette réponse dans un fichier `exerciceX.txt` (pas de document Word, Open Office, ou autre), `X` étant le numéro de l'exercice concerné.
    - Ce fichier doit être placé dans la racine du projet, et pas ailleurs.
- Vous pouvez exceptionnellement utiliser du PDF, si jamais la question implique de rendre des schémas.

3.1.4 Remise du travail

- Vous devez rendre une archive au format ZIP ou RAR contenant votre travail.
- Cette archive doit être nommée comme votre projet.
- Elle doit contenir le dossier correspondant à votre projet Eclipse, c'est-à-dire le dossier du type `TPXX_NomPrénom` contenu dans votre workspace.
    - Attention de ne pas rendre un dossier contenant un dossier contenant... un dossier contenant votre projet.
    - Inversement, attention de ne pas rendre un sous-dossier de votre projet.
- Vous devez rendre un projet Eclipse : ne rendez pas seulement une archive contenant vos fichiers sources.
- Vous ne devez pas rendre autre chose qu'un projet Eclipse : pas de DevC++ ou autre.
- Le travail qui vous est demandé est un travail personnel : vous pouvez éventuellement réfléchir à plusieurs sur un problème, mais la programmation est individuelle.
- Si plusieurs étudiants présentent le même travail, ils seront tous sanctionnés. La sanction peut aller jusqu'à la note zéro pour tous les TP du semestre.

- Toute copie à partir des corrigés publiés les années précédentes sera également sanctionnée.

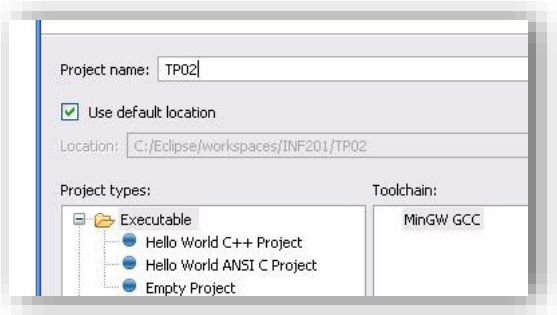
3.2 Création d'un projet

3.2.1 Nom du projet

- Sous Eclipse, allez dans *File>New>Project…*.
- Allez dans *C* et sélectionnez *C Project*.
- Dans *Project name*, donnez un nom à votre projet : cf. la section sur les règles des TP pour savoir quel nom vous devez donner à votre projet.
- Cliquez enfin sur *Finish* pour terminer la création du projet, qui va alors apparaître dans la fenêtre de gauche (appelée *Explorer*).

3.2.2 Structure du projet

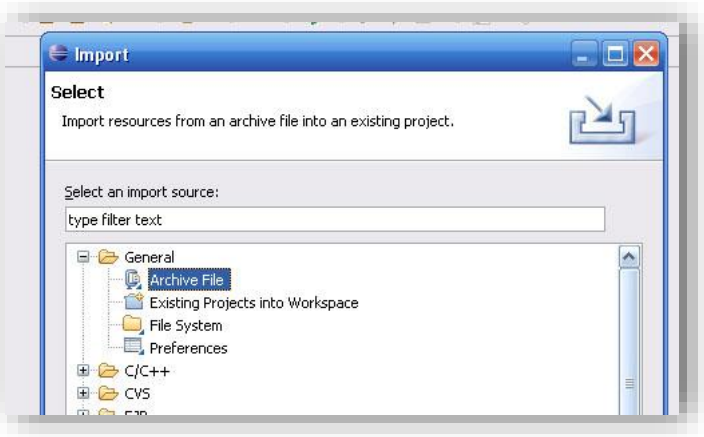
- Le nom du dossier contenant votre programme sera identique à celui de votre projet.
- Sélectionnez le mode *Executable* et le *Toolchain MinGW GCC*, puis cliquez sur *Finish*.



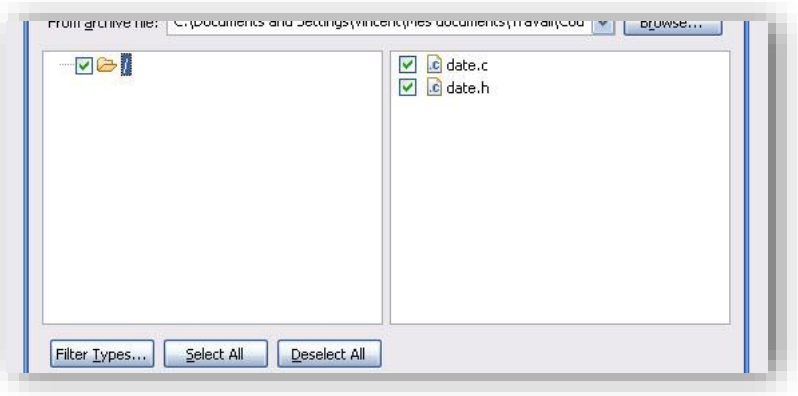
- Perspective :
  - Eclipse va éventuellement vous demander si vous voulez passer à la perspective C/C++, auquel cas il faut accepter.
  - Une perspective est une façon d'organiser les outils d'Eclipse adaptée à un langage de programmation donné.
  - La perspective par défaut est dédiée à Java.

3.2.3 Importation de fichiers

- Dans certains TP, vous devrez utiliser des fichiers fournis avec le sujet, en général sous la forme d'une archive (souvent un fichier ZIP).
- Pour pouvoir utiliser ces fichiers, vous devez les importer dans votre projet.
- Bien sûr, il faut d'abord avoir créé votre projet (en respectant les règles de nommage énoncées précédemment).
- Assurez-vous que votre projet est bien sélectionné dans l'explorateur (fenêtre de gauche), puis :
  - Cliquez dans le menu *File>Import* puis sur *General>Archive File*, et enfin sur *Next*.



- Utilisez le bouton *Browse* pour sélectionner l'archive à importer.
- Une fois que l'archive est choisie, son contenu va apparaître dans la partie gauche de la fenêtre : cochez les cases correspondantes aux fichiers et/ou dossiers que vous désirez importer (en général : tous !).



- Cliquez sur le bouton *Finish* pour effectuer l'importation.
- **Remarque :**
  - Si vous ne pouvez pas cliquer sur *Finish*, c'est que vous n'avez pas sélectionné de projet avant de démarrer l'importation.
  - Vous devez alors utiliser le bouton *Browse* pour sélectionner le projet adéquat.

- **Remarque :** l'importation copie toute la structure que vous sélectionnez dans l'archive, y compris l'arborescence des dossiers, donc faites attention à ce que vous sélectionnez.

3.3 Bibliothèque mathématique

Dans certains TP, il est nécessaire d'utiliser différentes fonctions appartenant à la bibliothèque mathématique : valeur absolue, fonctions trigonométriques, puissance, etc. Sous Windows, il suffit d'inclure cette bibliothèque dans le programme, grâce à la directive :

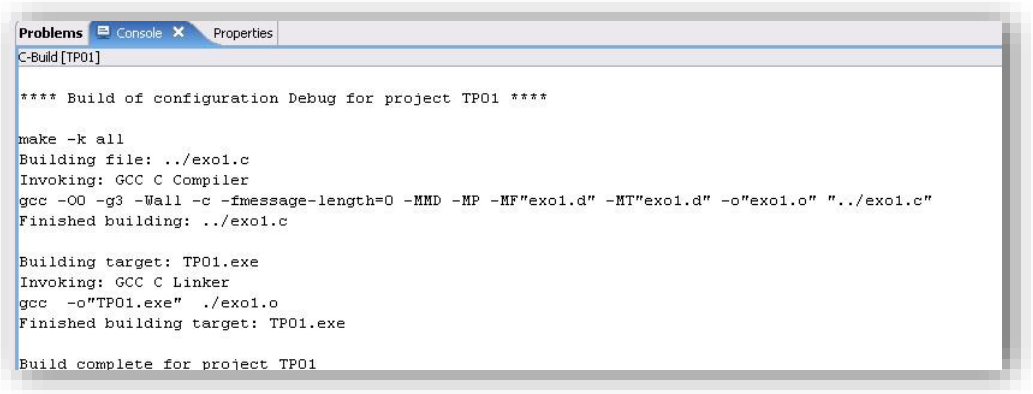
```
#include <math.h>
```

En revanche, sous Linux, il faut aussi réaliser cette inclusion mais elle ne suffit pas. Il est nécessaire de spécifier que cette bibliothèque est utilisée, en plus, dans les propriétés du projet. Pour cela, il faut réaliser les opérations suivantes :

- Dans les propriétés du projet, allez dans *C/C++ Build>Settings>Tool settings>GCC Linker>Libraries*.
- Dans *Libraries (-l)* : rajoutez `m` (juste cette lettre, en minuscule).

3.4 Compilation

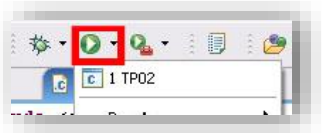
- Pour créer un fichier, effectuez un clic-droit sur le projet concerné (dans la fenêtre de gauche) et choisissez :
  - *New>Source File* si vous voulez créer un fichier source (`.c`).
  - *New>Header File* si vous voulez créer un fichier d'en-tête (`.h`)
- Vous devez préciser le nom de votre fichier, y compris son extension (`.c` ou `.h`). Par exemple : `exo1.c`.
- Si vous avez configuré Eclipse comme indiqué dans l'annexe sur l'installation, la compilation de votre fichier est réalisée automatiquement à chaque sauvegarde.
- Sinon, il faut compiler manuellement en utilisant le bouton situé en haut, ou bien en appuyant sur Control+B (pour *Build*).
- La console d'Eclipse montre alors le résultat de la construction de l'exécutable :



3.5 Exécution

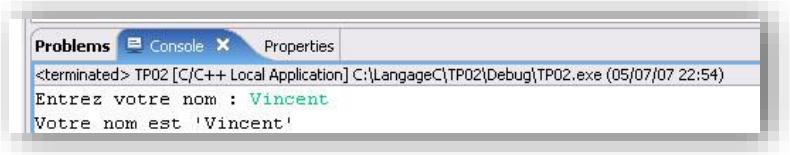
3.5.1 Dans Eclipse

- Si votre programme ne provoque pas d'erreur de compilation, vous pouvez l'exécuter depuis Eclipse, en utilisant la console d'Eclipse.
- Dans la fenêtre de gauche, faites un clic-droit sur votre projet et choisissez *Runs as…>Local C/C++ Application*.
- Une configuration d'exécution est automatiquement créée, et l'application est exécutée dans la console d'Eclipse située dans la fenêtre du bas.
- Une configuration d'exécution permet de paramétrer la façon dont un programme est exécuté.
- Si vous créez plusieurs configurations, elles apparaîtront dans le menu *Run*.



- Une fois que la configuration d'exécution est créée, il suffit de cliquer de nouveau sur le bouton *Run* de la barre d'outils pour ré-exécuter le même programme avec cette même configuration.
- Une configuration d'exécution existante peut être modifiée en cliquant sur *Open Run Dialog* dans le menu qui apparaît sous le bouton *Run*.
- En particulier, il est possible de spécifier des paramètres à passer au programme comme s'il était appelé depuis la ligne de commande (console DOS ou Linux), en utilisant l'onglet *Arguments*.

- Lorsque vous lancez une configuration d'exécution, votre programme est automatiquement exécuté dans la console d'Eclipse.
- Vous pouvez taper du texte dans la console si nécessaire, il apparaîtra dans une autre couleur que le texte affiché par le programme.



- **Attention :** la console Eclipse n'est pas bufférisée de la même façon que la console système : vous devez vider le buffer manuellement avec `fflush` pour afficher le résultat d'une opération manipulant le flux standard de sortie (`stdout`).

exemple :

```
printf("blablabla");
fflush(stdout);
```

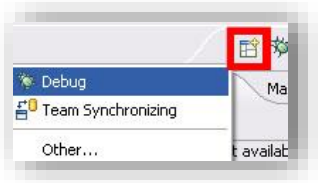
3.5.2 Hors d'Eclipse

- Il est également possible d'exécuter un programme depuis la console système, et non pas depuis celle d'Eclipse :
  - Ouvrez une console système (sous Windows : *Démarrer>Programmes>Accessoires>Invite de commandes*).
  - Allez dans le dossier correspondant à votre projet (ex. : `<Dossier Personnel>\INF201\TP02`).
  - Allez dans le dossier appelé `Debug`.
  - Lancez le fichier exécutable situé dans ce dossier (ici : `TP02`).

3.6 Débogage

3.6.1 Perspective de débogage

- Le débogage est un mode d'exécution particulier, qui permet de rechercher des erreurs d'exécution :
  - déroulement d'un programme pas à pas.
  - visualisation des valeurs des variables et de leur évolution.
  - possibilité de modifier leurs valeurs pendant l'exécution.

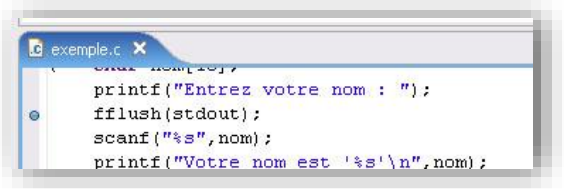


- Pour faire apparaître le bouton correspondant, cliquez sur le bouton *Perspectives* dans la barre d'outils (à droite) et sélectionnez *Debug*.
- Cette action va ouvrir la perspective de débogage.
- Revenez à la perspective C/C++ en cliquant sur le bouton situé dans la barre d'outils (à côté de *Debug*) et ouvrez un programme existant.
- Ouvrez le dialogue du menu *Run* pour modifier la configuration d'exécution du projet que vous voulez débuguer.
- Allez dans l'onglet *Debugger* et désactivez l'option *Stop on startup at*, qui définit par défaut un point d'arrêt au niveau du `main`.

- Cliquez sur *Apply* puis sur *Close*.
- Repassez à la perspective de débogage.

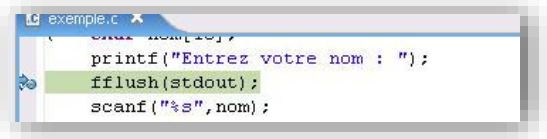
3.6.2 Contrôle de l'exécution

- Vous pouvez créer un point d'arrêt dans votre programme en effectuant un double clic-gauche dans la marge, au niveau de la ligne concernée. Le point d'arrêt est représenté par un point bleu.



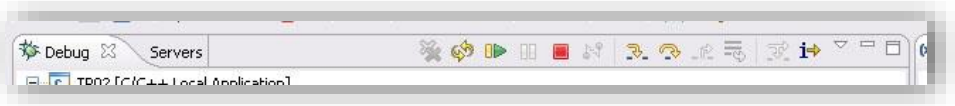
- Pour lancer le débogage, on procède comme pour lancer une exécution : on utilise une configuration d'exécution.
- La différence est qu'au lieu d'utiliser le bouton d'exécution, on utilise celui de débogage, qui est situé juste à côté.

- L'exécution a lieu dans une console système (et non pas une console Eclipse).
- Quand vous lancerez le débogage, le programme sera exécuté jusqu'à votre point d'arrêt, puis sera mis en pause. La ligne courante est surlignée dans la fenêtre affichant le source du programme :



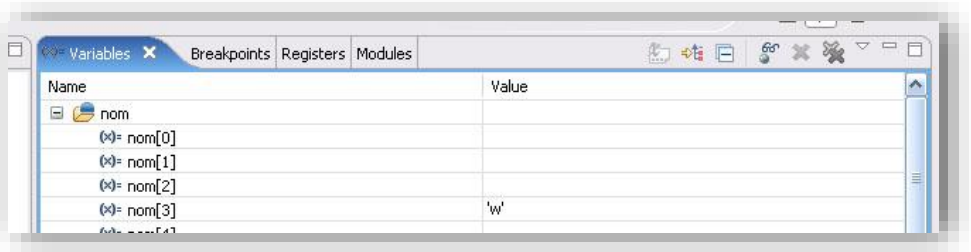
- La fenêtre *Debug* située en haut à gauche est très importante, elle affiche les programmes en cours d'exécution et affiche la pile des fonctions appelées.

- Cette fenêtre dispose également d'une barre d'outils permettant de contrôler l'exécution :
    - *Resume* (triangle vert) : reprend le cours normal de l'exécution, jusqu'à ce qu'un autre point d'arrêt soit rencontré, ou bien jusqu'à la terminaison du programme s'il n'y a pas d'autre point d'arrêt.
    - *Suspend* (deux barres verticales) : permet de mettre manuellement un programme en pause, par exemple lorsqu'il est pris dans une boucle infinie, ce qui permet d'identifier la boucle fautive.
    - *Terminate* (carré rouge) : termine le programme.
    - *Step Into* : exécute la prochaine instruction. Si cette instruction est une fonction, le débogueur entre dans la fonction et permet à l'utilisateur de contrôler son exécution.
    - *Step Over* : exécute la prochaine instruction. S'il s'agit d'une fonction, le débogueur exécute la fonction normalement puis revient au mode d'exécution contrôlé par l'utilisateur.
    - *Step Return* : exécute le reste de la fonction et repasse en mode contrôlé par l'utilisateur une fois la fonction terminée.

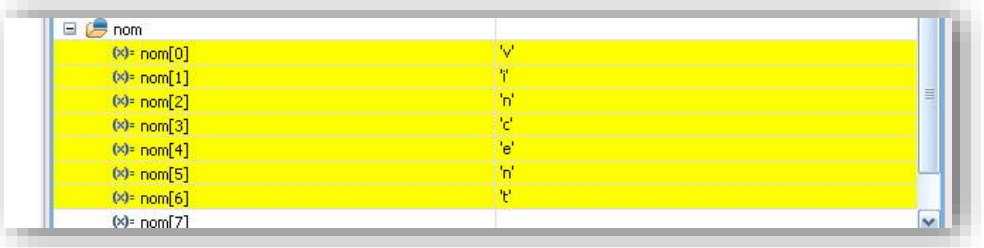


3.6.3 Accès à la mémoire

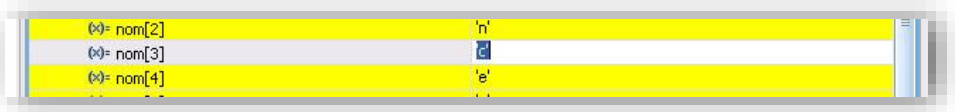
- La fenêtre située en haut à droite permet d'accéder à la mémoire du programme en cours d'exécution. En particulier, l'onglet *Variables* liste toutes les variables existantes et donne accès à leurs valeurs.
    - L'exemple montre une variable `nom`, qui est une chaîne de caractères et dont le contenu est quelconque car elle n'a pas été initialisée.



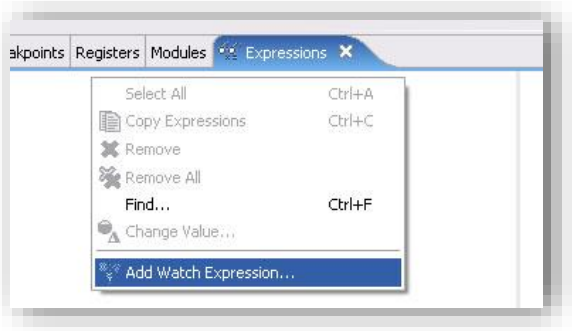
    - Les variables qui ont été modifiées par l'instruction qui vient d'être exécutée (ici un `scanf`) apparaissent en jaune.



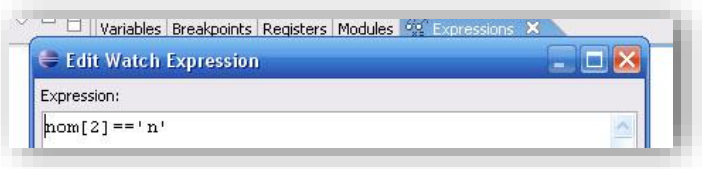
    - Il est possible de modifier manuellement la valeur d'une variable en cliquant dans la colonne *Value*.



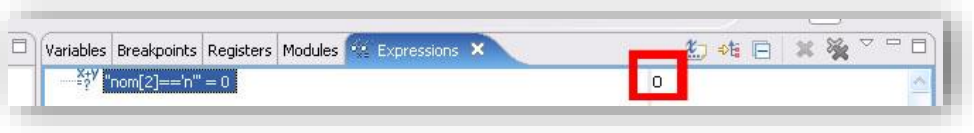
- Une autre possibilité intéressante est la possibilité d'évaluer des expressions pendant l'exécution, en utilisant les valeurs courantes des variables :
    - Si l'onglet Expressions n'apparaît pas dans la fenêtre située en haut à droite :
        - Cliquez sur l'onglet *Variables*.
        - Allez dans *Window>Show View>Other…*.
        - Sélectionnez *Debug>Expressions* puis cliquez sur OK.
        - L'onglet Expressions devrait maintenant apparaître.
    - Sélectionnez l'onglet *Expressions*.
    - Effectuez un clic droit dans le panel et cliquez sur *Add Watch Expression…*.



    - Vous pouvez alors entrer l'expression à évaluer, en validant avec *OK*.



    - Le panel de droite indique le résultat de l'évaluation de l'expression, ou bien un message d'erreur si votre expression ne peut pas être évaluée.



4 Sujets de TP

Les sujets de TP sont rangés de manière suivre la progression du cours. Un TP donné se concentre généralement sur un nouveau point en particulier, mais il est susceptible d'utiliser les concepts manipulés dans les TP précédents.

Le premier tableau suivant liste les TP en précisant les concepts qu'ils utilisent. Les notions vraiment basiques, telles que la syntaxe du C, la structure d'un programme, les entrées-sorties texte, les boucles et les fonctions, ne sont pas explicitement représentées car elles sont utilisées dans quasiment tous les TP (après leur TP d'introduction). Les notions concernées sont dans l'ordre :

- L'utilisation des tableaux,
- Celle des chaînes de caractères,
- La création de bibliothèques spécifiques,
- L'utilisation de la SDL (et donc de fonctions graphiques),
- L'utilisation de pointeurs,
- La définition et l'utilisation de types énumérés,
- Celles de types structurés,
- La lecture et l'écriture dans des fichiers (autres que graphiques),
- L'allocation dynamique de mémoire,
- La définition de fonctions récursives,
- La manipulation de listes chaînées (simplement et doublement),
- L'utilisation de piles de données,
- Celle de files de données,
- L'implémentation d'algorithmes de tris,
- L'étude de la complexité algorithmique.

Le second tableau liste les principaux prérequis pour chaque TP, i.e. les notions qu'il faut avoir déjà abordées pour pouvoir effectuer le TP. Il précise aussi le(s) objectif(s) du TP. Ceux indiqués en gras correspondent aux notions abordées pour la première fois. De nombreux TP ont pour but d'approfondir des concepts déjà utilisés.

Numéro	Nom	Tableaux	Chaînes	Bibliothèque	SDL	Pointeurs	Énumérations	Structures	Fichiers	Allocation	Récursivité	Listes	Piles	Files	Tris	Complexité
01	Entrées-sorties															
02	Types simples															
03	Variables & opérateurs															
04	Structures de contrôle															
05	Crible d'Ératosthène	X														
06	Chaînes de caractères		X													
07	Tableaux multidimensionnels	X														
08	Diagrammes en mode texte															
09	SDL & codage				X											
10	Couleurs d'une image				X											
11	Passage de paramètres															
12	Algorithme de Bresenham				X											
13	Histogramme des couleurs			X	X											
14	Propriétés arithmétiques	X														
15	Algorithmes pour l'arithmétique															
16	Bibliothèque `chaine`	X	X	X		X										
17	Décomposition d'une permutation	X		X		X										
18	Nombres binaires	X		X		X										
19	Algorithme de Johnson	X					X									
20	Manipulation de dates	X		X		X	X	X								
21	Carrés latins	X					X									
22	Représentation d'une promotion	X	X	X			X	X								
23	Partition d'un entier	X				X		X								
24	Rotation d'un carré	X			X			X								
25	Zoom d'une image	X			X	X										
26	Automates finis	X						X								
27	Championnat de football	X				X	X	X								
28	Floutage d'une image	X		X	X	X										
29	Flous avancés	X		X	X	X										
30	Gestion d'un lexique	X	X			X										
31	Allocation dynamique	X	X			X				X						
32	Générateur pseudo-aléatoire	X		X						X						
33	Nombres hexadécimaux		X			X				X						
34	Répertoire téléphonique		X	X		X				X						
35	Fichiers & arguments de programme			X				X	X							
36	Diaporama	X	X		X	X			X	X						
37	Stock d'une librairie	X		X		X		X	X							
38	Automates cellulaires	X		X	X	X			X	X						
39	Fonctions récursives		X								X					
40	Approximations numériques										X					
41	Figures fractales				X						X					
42	Listes chaînées					X						X				
43	Disques & guirlandes			X	X	X		X				X				
44	Listes de caractères								X			X				
45	Tapis de Sierpiński				X			X			X	X				
46	Suite de Syracuse				X	X					X	X				
47	Enveloppe d'un nuage de points				X			X				X				
48	Marches de Graham et Jarvis			X	X	X		X				X				
49	Analyse d'expressions		X										X			
50	Conversion d'expressions		X	X							X		X			
51	Détection de palindromes		X			X				X	X		X			
52	Tours de Hanoï				X	X	X				X					
53	Remplissage de formes				X						X	X	X			X
54	Parcours d'un labyrinthe	X			X		X						X	X		
55	Génération d'un labyrinthe	X			X		X									
56	Tri par dénombrement	X													X	X
57	Tri cocktail	X									X				X	X
58	Représentation des tris	X			X										X	
59	Tris sur listes			X								X			X	
60	Représentation de l'ADN										X	X				X
61	Nombres de grande taille										X	X				X
62	Table de symboles		X	X					X			X				
63	Plus longue sous séquence commune	X									X					X
64	Arbres binaires										X				X	

Liste des points abordés dans chaque TP

N°	Nom	Prérequis principaux	Objectifs
01	Entrées-sorties	Syntaxe du langage C	Écrire une **fonction principale** **Afficher/saisir du texte**
02	Types simples	Écrire une fonction principale Afficher/saisir du texte	Manipuler les **types simples** du C Codage d'un nombre
03	Variables & opérateurs	Notion de fonction Opérateurs arithmétiques	Position des variables en **mémoire** Valeur et type d'une expression
04	Structures de contrôle	Instructions conditionnelles/répétition Faire un tableau de situation	Utiliser les **instructions conditionnelles** Utiliser les **instructions de boucle**
05	Crible d'Érathostène	Syntaxe des tableaux Utiliser des boucles	Manipuler des **tableaux** Approfondir les boucles
06	Chaînes de caractères	Manipuler des tableaux Syntaxe des chaînes de caractères	Manipuler des **chaînes de caractères** Approfondir les tableaux
07	Tableaux multidimensionnels	Manipuler des tableaux Utiliser des boucles	Manipuler des **matrices** Tableaux multidimensionnels en mémoire
08	Diagrammes en mode texte	Syntaxe des fonctions Utiliser des boucles	Définir des **fonctions** Passer des paramètres
09	SDL & codage	Définir des fonctions	Créer/configurer un **projet SDL** **Dessiner** des figures simples
10	Couleurs d'une image	Utiliser la SDL Codage des couleurs	Charger/afficher une **image** Modifier une image
11	Passage de paramètres	Utiliser la bibliothèque mathématique	Approfondir les fonctions Passer un paramètre par adresse
12	Algorithme de Bresenham	Utiliser la SDL Notions de géométrie	Tracer des segments Approfondir la SDL
13	Histogramme des couleurs	Utiliser la SDL Charger une image	Créer une **bibliothèque** Réutiliser des fonctions existantes
14	Propriétés arithmétiques	Manipuler des tableaux	Approfondir les tableaux Approfondir le passage par adresse
15	Algorithmes pour l'arithmétique	Définir des fonctions Notions d'arithmétique	Faire une **preuve de programme**
16	Bibliothèque `chaine`	Syntaxe des chaînes de caractères Syntaxe des pointeurs	Manipuler les **chaînes de caractères** Approfondir la création de bibliothèque
17	Décomposition d'une permutation	Créer une bibliothèque Syntaxe des pointeurs	Manipuler des **pointeurs** Notions sur les permutations
18	Nombres binaires	Créer une bibliothèque Manipuler des pointeurs	Approfondir les tableaux Approfondir les pointeurs
19	Algorithme de Johnson	Syntaxe des énumérations Définir de types personnalisés	Définir des **types personnalisés** Manipuler les **énumérations**
20	Manipulation de dates	Syntaxe des structures/énumérations Définir des types personnalisés	Définir des **structures** Approfondir les énumérations
21	Carrés latins	Définir des énumérations Manipuler des matrices	Approfondir les matrices Approfondir les énumérations
22	Représentation d'une promotion	Un peu tout ce qui a été vu précédemment	Approfondissement général
23	Partition d'un entier	Manipuler des structures Manipuler des pointeurs	Approfondir les structures Approfondir les pointeurs
24	Rotation d'un carré	Manipuler des structures Manipuler des matrices	Approfondir les structures
25	Zoom d'une image	Manipuler des images	Approfondir les matrices
26	Automates finis	Manipuler des structures Manipuler des matrices	Approfondir structures Notion d'automate fini
27	Championnat de football	Manipuler des types personnalisés Manipuler des matrices	Approfondir les pointeurs Approfondir les structures
28	Floutage d'une image	Manipuler des images	Approfondissement général Notion de floutage
29	Flous avancés	Floutage d'images Manipuler des images	Approfondir le floutage
30	Gestion d'un lexique	Manipuler des chaînes de caractères Manipuler des pointeurs	Manipuler des **tableaux de pointeurs** Approfondir les chaînes
31	Allocation dynamique	Manipuler des pointeurs Manipuler des tableaux	**Allocation dynamique** Gestion de la mémoire
32	Générateur pseudo-aléatoire	Allocation dynamique Bibliothèque `histogramme`	Approfondir allocation dynamique Méthodes pseudo-aléatoires

N°	Nom	Prérequis principaux	Objectifs
33	Nombres hexadécimaux	Manipuler des chaînes Allocation dynamique	Approfondir allocation dynamique Approfondir chaînes de caractères
34	Répertoire téléphonique	Allocation dynamique Définir des structures	Approfondir indirections multiples Approfondir tableaux de pointeurs
35	Fichiers & arguments de programme	Syntaxe de la fonction `main`	Lecture/écriture dans un **fichier** Arguments en **ligne de commande**
36	Diaporama	Allocation dynamique Manipuler des fichiers	Approfondir les fichiers Utiliser la **triple indirection**
37	Stock d'une librairie	Allocation dynamique Manipuler des fichiers	Approfondissement général
38	Automates cellulaires	Manipuler des fichiers Allocation dynamique	Approfondir les fichiers Approfondir l'allocation dynamique
39	Fonctions récursives	Écrire des fonctions Notion de récursivité	**Fonctions récursives**
40	Approximations numériques	Fonctions récursives	Approfondir la récursivité Approximation numérique
41	Figures fractales	Fonctions récursives	Approfondissement de la récursivité Notion de fractale
42	Listes chaînées	Notion de liste Écrire une fonction	Manipuler des **listes**
43	Disques & guirlandes	Manipuler des listes	Approfondir les listes Approfondir les types structurés
44	Listes de caractères	Manipuler des fichiers Manipuler des listes	Approfondir les listes Approfondir les fichiers
45	Tapis de Sierpiński	Fonctions récursives Manipuler des listes	Approfondir la récusivité Approfondir les listes
46	Suite de Syracuse	Fonctions récursives Manipuler des listes	Approfondir la récursivité Tracer des **graphiques**
47	Enveloppe d'un nuage de points	Manipuler de listes Manipuler des structures	Manipuler des **listes doubles** Approfondir les types structurés
48	Marches de Graham et Jarvis	Enveloppe d'un nuage de points Manipuler des listes	Approfondir les listes
49	Analyse d'expressions	Manipuler des chaînes de caractères Notion de pile de données	Manipuler les **piles de données**
50	Conversion d'expressions	Manipuler des piles de données Analyse d'expressions	Approfondir les piles de données Approfondir la récursivité
51	Détection de palindromes	Manipuler des piles de données Fonctions récursives	Approfondir les piles de données Approfondir l'allocation dynamique
52	Tours de Hanoï	Fonctions récursives Utiliser des énumérations	Approfondir récursivité
53	Remplissage de formes	Manipuler des piles de données Fonctions récursives	Calculer des **complexités** Approfondir les piles de données
54	Parcours d'un labyrinthe	Manipuler des piles Notion de file de données	Manipuler les **files de données** Approfondir piles de données
55	Génération d'un labyrinthe	Manipuler des énumérations Manipuler un labyrinthe	Approfondissement général
56	Tri par dénombrement	Manipuler des tableaux Notion de tri	Trier un tableau Approfondir le calcul de complexité
57	Tri cocktail	Manipuler des tableaux Tri à bulle	Approfondir le tri de tableau Approfondir le calcul de complexité
58	Représentation des tris	Trier un tableau	Approfondir les tris Approfondir la SDL
59	Tris sur listes	Manipuler des listes Trier un tableau	**Trier une liste** Manipuler des listes
60	Représentation de l'ADN	Manipuler des listes Fonctions récursives	Approfondir les listes Approfondir le calcul de complexité
61	Nombres de grande taille	Manipuler des listes Fonctions récursives	Approfondir la récursivité Approfondir le calcul de complexité
62	Table de symboles	Manipuler des listes Manipuler des chaînes de caractères	Introduction au hachage
63	Plus longue sous séquence commune	Fonctions récursives Calcul de complexité	**Programmation dynamique** Approfondir le calcul de complexité
64	Arbres binaires	Fonctions récursives	**Arbres binaires** **Arbre binaires de recherche**

Liste des prérequis et objectifs de chaque TP

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 01 | entrées-sorties

Présentation

Le but de ce TP est d'apprendre :

- Comment écrire un programme simple, comportant seulement une fonction principale ;
- Comment afficher du texte à l'écran et saisir du texte entré par l'utilisateur.

1 Organisation du code source

Les programmes en langage C sont structurés au moyen de *fonctions*. Cependant, celles-ci seront étudiées plus tard, et on ne les utilisera donc pas dans les premiers TP. Pour cette raison, le code source de ces premiers TP aura une organisation particulière, de la forme suivante :

```
int main()
{  // exercice 1
   ...   // code source de l'exercice 1 ...

   // exercice 2
   ...   // code source de l'exercice 2 ...

   ...   // reste des exercices

   return EXIT_SUCCESS;
}
```

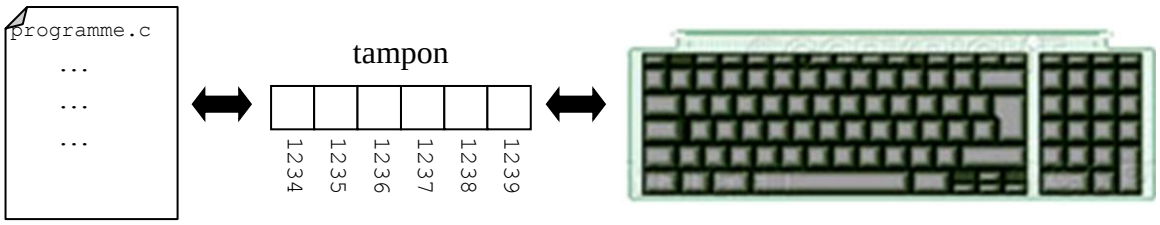
Autrement dit, la fonction `main` contiendra tous les exercices, identifiés par des commentaires appropriés. Quand un exercice sera terminé, le code source correspondant sera lui-même placé en commentaire, afin de ne pas interférer avec l'exercice suivant.

2 Notion d'entrées-sorties

Le terme d'entrées/sorties désigne les mécanismes utilisés pour qu'un programme puisse recevoir/envoyer des données depuis/vers un **flux**. Un flux peut être un fichier, un périphérique (écran, clavier, etc), ou d'autres entités. Dans ce TP, nous nous intéressons seulement à l'entrée **standard**, qui est le clavier, et à la sortie **standard**, qui est l'écran. En langage C, le flux d'entrée standard est appelée `stdin` (pour *st*and*ard in*put) et le flux de sortie standard est appelée `stdout` (pour *st*and*ard out*put).

Il existe de nombreuses méthodes permettant d'effectuer des entrées/sorties en langage C, suivant le type des données à traiter. On distinguera les fonctions qui manipulent les données de manière **non-formatée**, c'est-à-dire caractère par caractère, et celles qui travaillent de manière **formatée**.

Par défaut, l'accès au flux est **bufférisé**, ce qui signifie qu'une zone de mémoire sert de tampon. Par exemple, dans le cas du flux d'entrée standard, les données issues du clavier sont **stockées** dans le tampon, et **attendent** que le programme ait besoin d'elles.



L'intérêt de l'accès bufférisé est que, tant que le programme ne demande pas l'accès au tampon, il est possible de **modifier** les données qui y sont contenues.

Remarque : pour pouvoir utiliser les fonctions d'entrée-sortie présentées dans ce TP, vous devez inclure la bibliothèque `stdio.h` dans votre programme.

3 Entrées/sorties non-formatées

La fonction `int getc(FILE *stream)` renvoie le code ASCII du caractère qui a été lu dans le flux `stream`. Si on veut saisir un caractère au clavier, on écrira :

```
char c;
c = getc(stdin);
```

Si une erreur se produit, la fonction renvoie `-1`. On ne tiendra pas compte de cette possibilité lorsqu'on travaille sur le flux standard `stdin`, mais il est important de connaître cette propriété, qui sera utile plus tard.

La fonction `getchar` fonctionne exactement pareil que `getc`, à la différence qu'on n'a pas à préciser qu'on utilise le flux standard :

```
char c;
c = getchar();
```

Ces deux fonctions `getc` **prennent** un caractère dans le tampon et le **suppriment** du tampon. Si le tampon est **vide**, le programme se **bloque** jusqu'à ce qu'un `char` soit placé dans le tampon, c'est-à-dire : jusqu'à ce qu'une valeur soit saisie au clavier.

La fonction `int putc(int c, FILE *stream)` écrit le caractère dont le code ASCII est `c` dans le flux `stream`. Si on veut afficher un caractère à l'écran, par exemple le caractère `A`, on écrira donc :

```
char c = 'A';
putc(c, stdout);
```

Si une erreur se produit, la fonction renvoie `-1`, sinon elle renvoie le code ASCII du caractère qui a été écrit. Là encore, on ne tiendra pas compte, ici, de la possibilité d'erreur. La fonction `putchar` fonctionne exactement pareil que `putc` sans préciser le flux :

```
char c = 'A';
putchar(c);
```

Exercice 1

En utilisant `getchar` et `putchar`, écrivez un programme qui saisit un caractère et qui l'affiche à l'écran.

Exercice 2

Même question avec deux caractères : d'abord le programme saisit les deux caractères, puis il affiche les deux caractères.

Remarque : pour changer de ligne, vous pouvez afficher le caractère `'\n'`.

4 Sorties formatées

La fonction `putchar` ne permet d'afficher que des caractères. La fonction `printf`, elle, permet d'effectuer une sortie formatée, et d'afficher des valeurs de n'importe quel type simple :

```
printf(format,exp1,...,expn);
```

La fonction affiche les valeurs des expressions `exp1,...,expn`. Le paramètre `format` est une **chaîne de caractère de formatage** indiquant **comment** les expressions doivent être affichées. Cette chaine de caractères peut contenir deux sortes d'informations :

- du texte **normal**, qui sera affiché tel quel
- du texte spécifiant des **formats** d'affichage

Par exemple, pour afficher le texte normal *bienvenue dans le programme*, on fera (dans l'exemple, la ligne foncée représente le résultat à l'écran) :

```
printf("bienvenue dans le programme");
bienvenue dans le programme
```

Un format d'affichage prend la forme d'un **pourcent** `%` suivi d'une expression précisant le format. Par exemple, le format `%d` signifie que l'on veut afficher un entier relatif exprimé en base 10 (entier décimal) :

```
printf("%d",1234);
1234
```

Il est possible d'avoir du texte normal et des formats dans la **même** chaîne de formatage :

```
printf("affichage du nombre : %d",1234);
affichage du nombre : 1234
```

Lors de l'affichage, chaque format présent dans la chaine de formatage est utilisé pour afficher la valeur d'une des expressions passées en paramètre. Le 1er format sert à afficher la 1ère expression, le 2ème format à afficher la 2ème expression, etc. :

```
printf("affichage du nombre 1 : %d. affichage du nombre 2 : %d",12,34);
affichage du nombre 1 : 13. affichage du nombre 2 : 34
```

On manipule des **expressions**, on n'est donc pas limité à des **constantes littérales**, on peut utiliser des **variables** et des **opérateurs**, par exemple :

```
int a=5;
printf("%d plus %d egale %d",10,a,10+a);
10 plus 5 egale 15
```

Remarque : le caractère `%` est un caractère spécial dans une chaîne de formatage, puisqu'il permet de définir un format Si on veut afficher le caractère `%` lui-même, il faut écrire `%%` dans la chaine de formatage, afin que la fonction `printf` sache qu'il ne s'agit pas d'un format :

```
printf("resultat des ventes : %d %%",87);
resultat des ventes : 87 %
```

Les **principaux** codes de formatage sont :

code	résultat affiché
`%d`	nombre entier relatif décimal
`%u`	nombre entier naturel décimal
`%o`	nombre entier naturel octal
`%x` et `%X`	nombre entier naturel hexadécimal
`%p`	nombre entier représentant une adresse (pointeur)
`%c`	caractère
`%f`	nombre réel décimal
`%e` et `%E`	nombre réel décimal en notation scientifique

Remarque : par défaut, un réel est affiché avec *6* chiffres après la virgule.

Il est possible de préciser, entre le `%` et la lettre du format, un **modificateur** de formatage précisant le type de la donnée à afficher :

donnée	codes concernés	option	résultat affiché
entier	d, i, o, x, X, u	aucune	int
		h	(unsigned) short
		l	long
réel	f, e, E	aucune	float
		l	double
		L	long double

En plaçant une valeur entière entre le `%` et la lettre du format, on peut préciser le nombre de chiffres **minimal** à afficher. Pour un réel, le point décimal compte comme un chiffre. Si le nombre à afficher ne contient pas assez de chiffres, les chiffes manquants sont remplacés par des caractères *espace*. Si on préfère remplir le vide avec des caractères *0*, il suffit de faire précéder la valeur minimale d'un zéro.

```
printf("entier:%4d reel:%9f",12,1.23456);
entier:  12 reel: 1.234560
printf("entier:%04d reel:%09f",12,1.23456);
entier:0012 reel:01.234560
```

On peut également fixer la précision, c'est-à-dire le nombre de chiffres **après** la virgule en faisant suivre le nombre précédent d'un point et d'une valeur entière. Si le nombre à afficher contient plus de chiffres, un arrondi est réalisé. Sinon, le nombre est complété avec des zéros :

```
printf("%.8f %.2f",1.23456,1.23456);
1.23456000 1.23
```

Exercice 3

Soient les variables réelles (utilisez des `double`) suivantes : `x1=1.2345`,`x2=123.45`, `x3=0.000012345`, `x4=1e-10` et `x5=-123.4568e15`. Affichez leurs valeurs avec `%f` et avec `%e`.

Exercice 4

Soient les variables réelles (`float`) suivantes : `x1=12.34567`, `x2=1.234567`, `x3=1234567`, `x4=123456.7`, `x5=0.1234567` et `x6=1234.567`. Utilisez `printf` pour obtenir l'affichage suivant :

```
     12.35         1.23
1234567.00    123456.70
      0.12      1234.57
```

Exercice 5

Écrivez un programme qui affiche la valeur 1234,5678 de type `float` avec les formats `%d`, `%f` et `%e`. Qu'observe-t-on pour `%d` ?

5 Entrées formatées

La fonction `scanf` permet de saisir des valeurs de manière formatée. Le principe est le même que pour `printf` : on utilise une chaîne de formatage, et une suite de paramètres.

```
scanf(format,adr1,...,adrn);
```

La différence est qu'ici, les paramètres sont des adresses. Ainsi, pour saisir un entier, on fera :

```
int i;
scanf("%d",&i);
```

L'opérateur `&` permet de préciser qu'on passe en paramètre non pas la variable `i`, mais son adresse. On peut saisir plusieurs valeurs à la fois, l'utilisateur devra les séparer par un retour chariot, un espace ou bien une tabulation :

```
int i,j,k;
scanf("%d%d%d",&i,&j,&k);
```

Cela signifie que par défaut, il n'est **pas possible** de lire le caractère *espace* avec `scanf`.

On utilise avec `scanf` les mêmes codes de formatages (`d`, `u`, `o`, `x`, `X`, `c`, `f`, `e`, `E`) et modificateurs (`h`, `l`, `L`) que pour `printf`. On peut préciser la **longueur maximale** de la donnée lue, en insérant une valeur avant le code de formatage (ou le modificateur). Par exemple, pour saisir un entier de 5 chiffres maximum :

```
int i;
scanf("%5d",&i);
```

Il est également possible de préciser que les valeurs lues doivent être séparées par des caractères particuliers. Par exemple, pour lire *3* valeurs séparées par des points-virgules :

```
int i,j,k;
scanf("%d;%d;%d",&i,&j,&k);
```

Remarque : `scanf` utilise un accès bufférisé au clavier, et ne consomme pas le dernier retour chariot. On a donc le même problème que pour `getchar`.

Exercice 6

Écrivez un programme qui saisit un entier avec `scanf` et affiche le triple de cette valeur.

Exercice 7

Écrivez un programme qui saisit une heure au format suivant : *heures:minutes:secondes*. Le programme doit afficher les valeurs saisies de la manière suivante (en respectant l'alignement) :

```
Entrez l'heure (hh:mm:ss) : 1:2:34
 1 heure(s)
 2 minute(s)
34 seconde(s)
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 02 | types simples

Présentation

Le but de ce TP est de manipuler les types simples du langage C. Pour illustrer le cours, qui portait sur leur représentation en mémoire, nous allons travailler sur la différence existant entre la façon dont un nombre est codé et la façon dont il est saisi/affiché.

Remarque : les justifications et réponses textuelles doivent être indiquées sous forme de commentaires, juste après le programme concerné. Cette remarque est valable pour tous les TP à venir.

1 Caractères

Exercice 1

Écrivez un programme qui saisit un caractère et qui affiche :

- Son code ASCII
- Le caractère suivant dans la table ASCII

exemple :

```
Entrez un caractere : e
Le code ASCII de 'e' est 101
Le caractere suivant dans la table ASCII est 'f'
```

Testez votre programme avec plusieurs caractères. Essayez notamment avec le caractère `'ü'`. Que se passe-t-il ? Pourquoi ? Corrigez l'éventuel problème.

Exercice 2

Écrivez un programme qui saisit un code ASCII et qui affiche le caractère correspondant.

exemple :

```
Entrez le code ASCII : 101
Le caractere correspondant au code ASCII 101 est 'e'
```

2 Entiers

Exercice 3

Écrivez un programme qui affiche les valeurs 12 et 4294967284. Ces deux valeurs doivent être affichées une première fois en tant qu'entiers non-signés, puis une seconde fois en tant qu'entiers signés. Que se passe-t-il, et pourquoi ?

Écrivez un programme qui affiche les constantes 12 et 4294967284. Ces deux valeurs doivent être affichées successivement :

- Au format entier naturel hexadécimal (`%x` ou `%X`) ;
- En tant qu'entiers non-signés (`%u`) ;
- Puis en tant qu'entiers signés (`%d`).

À la suite de votre programme, ajoutez un commentaire dans lequel vous devez :

- Indiquer la suite de bits codant en mémoire les entiers 12 et 4294967284.
- Interpréter les résultats, sachant que $2^{32} = 4294967296$.

Exercice 4

Écrivez un programme qui affiche la valeur 4 en décimal (`%d`), octal (`%o`) et hexadécimal (`%X`). Même question avec la valeur -4. Que se passe-t-il, et pourquoi ?

Exercice 5

Écrivez un programme qui saisit deux valeurs de type `short` et affiche la somme de ces deux valeurs. Testez votre programme avec différentes valeurs, puis calculez $18000 + 19000$ et $56 + (-33333)$. Qu'observez-vous ? Pourquoi ?

3 Réels

Exercice 6

Calculez la représentation de la constante 0,1 de type `float (i.e. 0.1f)` dans la mémoire de l'ordinateur.

Écrivez un programme qui affiche successivement cette constante avec les précisions suivantes :

- Un seul chiffre après la virgule ;
- Dix chiffres après la virgule.

Qu'observez-vous ? Proposez une explication.

Exercice 7

Calculer la valeur des constantes `(1e-9+1e9)-1e9` et `1e-9+(1e9-1e9)` demandez à votre programme d'afficher 10 décimales et expliquez le résultat obtenu.

Proposez une solution pour obtenir un résultat exact.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 03 | variables & opérateurs

Présentation

Le but de ce TP est de comprendre comment les variables sont placées en mémoire, et de manipuler les opérateurs de base du C.

Dans les exemples de ce TP, les `??` représentent en réalité des valeurs numériques. Celles-ci ne sont pas indiquées car le but des exercices est de les deviner avant d'exécuter le programme.

1 Variables & adresses

Exercice 1

Écrivez un programme qui déclare :

- Trois variables *globales* `a`, `b` et `c` de type `short` ;
- Trois variables *locales* à la fonction `main` `d`, `e` et `f` de type `short`.

Dans la fonction `main`, affichez les adresses de ces six variables, en utilisant le format pointeur `%p` dans `printf`, pour afficher une adresse.

exemple : le programme doit afficher des valeurs de la forme suivante (où les `??` correspondent à des valeurs connues seulement à l'exécution) :

```
&a:0x?????? &b:0x?????? &c:0x??????
&d:0x?????? &e:0x?????? &f:0x??????
```

En vous aidant des adresses affichées par votre programme, faites un schéma de l'occupation de la mémoire par ces variables. Qu'observez-vous ?

Exercice 2

Ajoutez une fonction dans votre programme, appelée fonction :

```
void fonction()
{   ...
}
```

Vous devez la compléter en remplaçant les `...` par la déclaration de trois variables `g`, `h`, et `i` de type `short`, et en affichant leurs adresses.

Dans la fonction `main`, après l'affichage de `d`, `e` et `f`, effectuez un appel à la fonction précédente en insérant l'instruction suivante :

```
    fonction();
```

Toujours dans la fonction `main`, après cet appel, créez trois variables locales `g`, `h`, et `i` de type `short`, et affichez leurs adresses.

En utilisant les adresses affichées par le programme, complétez le schéma de l'exercice précédent. Qu'observez-vous ?

2 Opérateurs & transtypage

Exercice 3

Écrivez un programme qui saisit un caractère. Si le caractère est une majuscule, il doit être transformé en minuscule. Sinon, le caractère ne doit pas être modifié. Enfin, le caractère

obtenu doit être affiché. Vous ne devez *pas* utiliser de *valeur entière littérale* (i.e. aucun nombre ne doit apparaître dans le programme).

exemple : si l'utilisateur saisit le caractère `V` :

```
Entrez le caractere a traiter : V
Caractere apres le traitement : v
```

Remarque : on ne considère que les majuscules sans accent (é), cédille (ç), tréma (ö), etc.

Exercice 4

Écrivez un programme qui saisit deux entiers puis calcule et affiche le résultat de la division **entière**.

exemple :

```
Entrez le premier operande : 17
Entrez le second operande : 5
17 = 5*3 + 2
```

Exercice 5

Écrivez un programme qui saisit deux entiers puis calcule et affiche le résultat de la division *réelle*.

exemple :

```
Entrez le premier operande : 10
Entrez le second operande : 4
10/4 = 2.250000
```

Exercice 6

Écrivez un programme qui :

- Demande à l'utilisateur de saisir une valeur réelle ;
- Tronque cette valeur de manière à obtenir au maximum deux chiffres après la virgule ;
- Affiche le résultat de la troncature.

exemple :

```
Entrez le nombre reel : 1.234
Resultat de la troncature : 1.230000
```

Remarque : il ne s'agit pas de simplement limiter le nombre de chiffres **affichés**, mais bien de modifier la **valeur** de la variable. Vous devez utiliser le *transtypage explicite*.

Exercice 7

Analysez le code source suivant, et essayez de prévoir ce qui sera affiché à l'écran. Puis, recopiez-le dans votre programme, exécutez-le et vérifiez vos prédictions.

```
int main()
{  int x=5,y=15;
   float u=2.1,v=5.0;
   printf("x==x : %d\n",x==x);
   printf("x==y : %d\n",x==y);
   printf("x==u : %d\n",x==u);
   printf("x==v : %d\n",x==v);
   printf("x>4 || x<3 : %d\n",x>4 || x<3);
   printf("x>4 && x<3 : %d\n",x>4 && x<3);
   if(x)
      printf("(x) : vrai\n");
   else
      printf("(x) : faux\n");
   printf("x=7 : %d\n",x=7);
   printf("x : %d\n",x);
   printf("x=(7!=8) : %d\n",x=(7!=8));
   printf("x : %d\n",x);
   printf("(float)x : %d\n",(float)x);
}
```

Que pouvez-vous déduire en ce qui concerne la valeur d'une expression d'affectation ?

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 04 | instructions de contrôle

Présentation

Le but de ce TP est de manipuler les structures de contrôle vues en cours, et en particulier les boucles.

1 Utilisation de la bibliothèque mathématique

Vous aurez besoin d'utiliser la fonction mathématique `math.h`, qui contient certaines fonctions nécessaires lors de quelques calculs. Comme pour les autres bibliothèques, il est nécessaire de mentionner explicitement son utilisation dans votre programme grâce à la directive `#include` :

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

Sous Windows, c'est suffisant. Sous Linux, il faut en plus paramétrer votre projet de la façon suivante :

- Dans *Project Explorer* (à gauche), faites un clic-droit sur votre projet et sélectionnez *Properties*.
- Allez dans *C/C++ Build* puis *Settings*.
- Dans l'onglet *Tool Settings*, allez dans *MinGW C Linker* puis *Libraries*.
- À droite, dans *Libraries (-l)*, cliquez sur le bouton permettant d'ajouter une bibliothèque, et entrez simplement `m`, puis validez avec *OK*.
- Cliquez sur *OK* pour fermer la fenêtre.

2 Test simple avec alternative

Exercice 1

Écrivez un programme qui demande à l'utilisateur de saisir trois entiers a, b et c, puis qui affiche le plus grand de ces trois entiers. Respectez exactement la forme de l'exemple donné ci-dessous.

exemple :

```
Entrez les valeurs de a, b et c (sous la forme a:b:c) : 8:12:-32
Le plus grand des entiers est 12.
```

Exercice 2

Écrivez un programme qui demande à l'utilisateur d'entrer trois réels a, b et c puis qui affiche les solutions de l'équation $ax^2 + bx + c = 0$ en fonction des valeurs de a, b et c. Vous devez utiliser la fonction `sqrt()` définie dans la bibliothèque `math.h`, qui calcule la racine carrée d'un réel.

exemples :

- Deux solutions :

```
Entrez les valeurs des coefficients a, b et c du trinome (a doit etre non-nul).
a = 2.1
b = 1.2
c = 0.1
```

```
L'equation  2.10x^2 +  1.20x +  0.10 = 0 admet deux solutions :
   x1 = -0.470142  et  x2 = -0.101287.
```

- Une seule solution :

```
Entrez les valeurs des coefficients a, b et c du trinome (a doit etre non-nul).
a = 4
b = 4
c = 1
L'equation 4.00x^2 + 4.00x + 1.00 = 0 admet exactement une solution :
   x = -1.000000
```

- Pas de solution :

```
Entrez les valeurs des coefficients a, b et c du trinome (a doit etre non-nul).
a = 0.1
b = -1.2
c = 5.2
L'equation  0.10x^2 + -1.20x +  5.20 = 0 n'admet pas de solution.
```

3 Boucles simples

Exercice 3

Écrivez un programme qui :

- Saisit un entier ;
- Renverse l'entier ;
- Affiche le résultat.

exemple :

```
Entrez l'entier a renverser : 1234
Resultat : 4321
```

Exercice 4

Donnez le *tableau de situation* pour la valeur utilisée dans l'exemple précédent.

4 Boucles imbriquées

Exercice 5

Écrivez un programme qui affiche (en respectant l'alignement indiqué dans l'exemple ci-dessous) la somme des p premiers entiers, pour p allant de 2 à $N = 10$.

exemple :

```
1 + 2 = 3
1 + 2 + 3 = 6
...
...
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

Exercice 6

Faites une copie du programme précédent, et modifiez-la de manière à demander à l'utilisateur de saisir la valeur de n :

- Si l'utilisateur saisit une valeur strictement positive, le programme effectue la même tâche qu'à l'exercice précédent, puis redemande une valeur n à l'utilisateur pour recommencer le traitement.
- Si l'utilisateur saisit une valeur négative ou nulle, le programme s'arrête.

exemple :

```
Entrez n ou une valeur<=0 pour terminer : 2
1 + 2 = 3
Entrez n ou une valeur<=0 pour terminer : 4
1 + 2 = 3
1 + 2 + 3 = 6
1 + 2 + 3 + 4 = 10
Entrez n ou une valeur<=0 pour terminer : -1
```

```
Le programme se termine.
```

5 Suite de Fibonacci

On considère la suite de Fibonacci (u_n), $n \in \mathbb{N}$ définie par :

$$\begin{cases} u_0 = 0; u_1 = 1 \\ \forall n \geq 2 : u_n = u_{n-1} + u_{n-2} \end{cases}$$

Exercice 7

En utilisant une boucle, écrivez un programme qui calcule *itérativement* le $n^{ème}$ terme de la suite de Fibonacci. La valeur n doit être demandée à l'utilisateur. À chaque itération, le programme doit afficher u_n.

exemple :

```
Entrez la valeur de n : 16
u0=0
u1=1
u2=1
u3=2
u4=3
u5=5
u6=8
u7=13
u8=21
```

```
u9=34
u10=55
u11=89
u12=144
u13=233
u14=377
u15=610
u16=987
```

Exercice 8

On peut montrer que la suite (u_n), $n \in \mathbb{N}$ vérifie $u_n \underset{+\infty}{\sim} \phi^n$, où ϕ est le nombre d'or. Ainsi, $\lim_{n \to +\infty} \frac{u_n}{u_{n-1}} = \phi$. Faites une copie de votre programme et modifiez-la pour tester cette propriété (utilisez le type `double`). À chaque itération, le programme doit afficher u_n et le rapport $\frac{u_n}{u_{n-1}}$.

Remarque : $\phi = \frac{1+\sqrt{5}}{2} \cong 1{,}61803398874989484820458683436 56$.

exemple :

```
Entrez la valeur de n : 20
u0=0
u1=1
u2=1
  u2/u1=1.000000000000000
u3=2
  u3/u2=2.000000000000000
u4=3
  u4/u3=1.500000000000000
u5=5
  u5/u4=1.666666666666667
u6=8
  u6/u5=1.600000000000000
u7=13
  u7/u6=1.625000000000000
u8=21
  u8/u7=1.615384615384615
u9=34
  u9/u8=1.619047619047619
u10=55
  u10/u9=1.617647058823529
```

```
u11=89
  u11/u10=1.618181818181818
u12=144
  u12/u11=1.617977528089888
u13=233
  u13/u12=1.618055555555556
u14=377
  u14/u13=1.618025751072961
u15=610
  u15/u14=1.618037135278515
u16=987
  u16/u15=1.618032786885246
u17=1597
  u17/u16=1.618034447821682
u18=2584
  u18/u17=1.618033813400125
u19=4181
  u19/u18=1.618034055727554
u20=6765
  u20/u19=1.618033963166706
```

Exercice 9

Faites une copie de votre programme et modifiez-la pour déterminer quel est le rang n à partir duquel ϕ est estimé avec une précision de 10^{-10}. À chaque itération, le programme doit afficher u_n, $\frac{u_n}{u_{n-1}}$ et l'erreur commise.

exemple :

```
u0=0
u1=1
u2=1
u2/u1=1.000000000000000
erreur : -0.618033988740000
u3=2
u3/u2=2.000000000000000
erreur : 0.381966011260000
u4=3
u4/u3=1.500000000000000
erreur : -0.118033988740000
u5=5
u5/u4=1.666666666666667
erreur : 0.048632677926667
u6=8
u6/u5=1.600000000000000
erreur : -0.018033988740000
```

```
u7=13
u7/u6=1.625000000000000
erreur : 0.006966011260000
u8=21
u8/u7=1.615384615384615
erreur : -0.002649373355385
u9=34
u9/u8=1.619047619047619
erreur : 0.001013630307619
u10=55
u10/u9=1.617647058823529
erreur : -0.000386929916470
u11=89
u11/u10=1.618181818181818
erreur : 0.000147829441818
...
```

Remarque : la fonction `fabs`, contenue dans la bibliothèque mathématique `math.h`, permet d'obtenir la valeur absolue d'un réel.

6 Devinette

On veut programmer un jeu simple, dont les règles sont les suivantes :

- D'abord, le programme choisit un nombre entre 1 et 100 au hasard ;
- Puis, il demande à l'utilisateur de deviner cette valeur.
- Si l'utilisateur a bien deviné, le jeu s'arrête.
- Sinon, le programme indique à l'utilisateur si la valeur proposée est trop grande ou trop petite. Puis, l'utilisateur doit proposer une nouvelle valeur, et on recommence ainsi jusqu'à ce que la valeur du programme soit trouvée.

Le nombre de tentatives n'est pas limité : le jeu s'arrête quand l'utilisateur a deviné la valeur choisie par le programme. Le score de l'utilisateur correspond au nombre de tentatives. Bien sûr, le but du jeu est de trouver la valeur en un minimum de tentatives.

Exercice 10

Écrivez un programme qui implémente ce jeu.

exemple :

```
J'ai tiré un nombre au hasard : essayez de le deviner !
  Premiere tentative ? 28
  La valeur 28 est trop grande.
  Tentative 2 ? 5
  La valeur 5 est trop petite.
  Tentative 3 ? 8
  Oui, c'est bien ca (8) ! Bravo, vous avez gagne !
Votre score est : 3
```

Remarque : pour tirer au hasard un entier n entre 1 et 100, utilisez le code source suivant (qui sera expliqué lors d'un TP futur). Vous avez besoin d'inclure la bibliothèque `time.h`.

```
srand(time(NULL));
int n = 1 + rand()%100;
```

À votre avis, quelle stratégie le joueur doit-il adopter pour obtenir le meilleur score (i.e. le plus petit score) ?

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 05 | crible d'Ératosthène

Présentation

Le but de ce TP est de manipuler les notions vues dans les cours et TP précédents, en particulier les tableaux, tests et boucles.

1 Définition

Un nombre entier $p \in \mathbb{N}$ est premier ssi $p > 1$ et p n'est divisible que par 1 et par lui-même. On notera $P = \{2, 3, 5, 7, ...\}$ l'ensemble des nombres premiers. De plus, pour tout $i \in \mathbb{N}$, p_i sera le $i^{ème}$ nombre premier, de sorte que $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, $p_3 = 7$, etc.

2 Approche naïve

Exercice 1

Écrivez un programme qui demande à l'utilisateur d'entrer un nombre entier naturel, et qui teste si ce nombre est premier. Si le nombre n'est pas premier, le programme doit afficher le plus petit diviseur. Le programme devra tester tous les diviseurs possibles en partant de 2.

exemple 1 :

```
Entrez le nombre naturel a tester : 17
17 est un nombre premier.
```

exemple 2 :

```
Entrez le nombre naturel a tester : 12
12 n'est pas un nombre premier : il est divisible par 2.
```

Rappel : l'opérateur permettant d'obtenir le reste (modulo) de la division entière est `%`.

Exercice 2

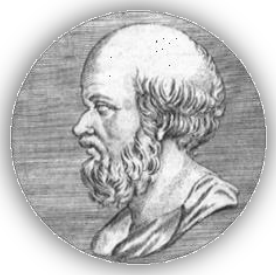
Dans le `main`, faites une copie du code source de votre programme précédent, et modifiez-la de manière à afficher tous les nombres premiers compris entre 1 et la valeur entrée par l'utilisateur (en incluant cette valeur).

```
Entrez un entier naturel (>2) : 9
   1 est un nombre premier.
   2 est un nombre premier.
   3 est un nombre premier.
   5 est un nombre premier.
   7 est un nombre premier.
```

Exercice 3

Faites une copie de votre programme précédent, et modifiez-la de manière à stocker tous les nombres premiers dans un tableau nommé `premier`, avant de les afficher. L'élément *i* du tableau doit contenir le $i^{ème}$ nombre premier : `premier[0]` contient 2, `premier[1]` contient 3, `premier[2]` contient 5, `premier[3]` contient 7, etc.

3 Crible d'Ératosthène



L'inconvénient de l'algorithme naïf est qu'il effectue de nombreuses divisions inutiles. Par exemple, il peut tester la divisibilité par 6 d'un entier qui n'est divisible ni par 3, ni par 2. Ératosthène était un scientifique de l'antiquité, qui a proposé un algorithme appelé Crible d'Ératosthène, qui permet d'éviter ces calculs inutiles.

Exercice 4

L'algorithme du Crible d'Ératosthène pour déterminer les nombres premiers compris entre 2 et N est le suivant :

- On considère un tableau contenant tous les entiers de 2 à N.
- On supprime tous les multiples de 2 (sauf 2).
- Parmi les nombres restants, on supprime tous les multiples de 3 (sauf 3).
- On répète ce traitement jusqu'à ce qu'il n'y ait plus de nombre à traiter.
- Les nombres restants à la fin du traitement sont les nombres premiers recherchés.

Appliquez manuellement cet algorithme pour trouver les nombres premiers compris entre 1 et 30.

Exercice 5

Il est peu pratique d'implémenter directement cet algorithme ici, en raison de la difficulté à supprimer des éléments dans un tableau. On se propose plutôt d'adapter notre programme de façon à exploiter le principe d'Ératosthène :

- Soit le tableau `premier`, utilisé pour stocker les nombres premiers ;
- On considère chaque entier de 2 jusqu'à N ;
- Pour chaque entier n, on teste s'il est premier en le divisant par les nombres premiers inférieurs à n (qui sont contenus dans le tableau `premier`) ;
- Si n est premier, on le rajoute dans le tableau. Sinon, on ne fait rien ;
- On passe à $n + 1$.

Faites une copie du programme précédent et modifiez-la pour y intégrer cette modification.

4 Amélioration

Exercice 6

L'implémentation de cet algorithme nécessite de déclarer un tableau de taille N, ce qui requiert une grande quantité de mémoire (si N est grand). Pourtant, on identifie forcément moins de N nombres premiers, donc une partie de ce tableau est inutilisée. Il est possible d'utiliser un tableau plus petit en utilisant la propriété suivante :

$$P_1 : n \in P \Leftrightarrow \left(\forall p \in P \cap \left[2; \frac{n}{2}\right], p\ ne\ divise\ pas\ n\right)$$

Prouvez la propriété P_1.

Exercice 7

Faites une copie de votre dernier programme, et modifiez-la de manière à intégrer cette amélioration permettant d'utiliser un tableau de taille $N/2$ au lieu de N.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 06 | chaînes de caractères

Présentation

Le but de ce TP est de manipuler les chaînes de caractères : affichage, saisie, modification, comparaison, etc.

1 Rappel

En langage C, une chaîne de caractères est un tableau de valeurs de types `char`, qui se termine par le caractère de fin de chaîne, noté `'\0'`.

Une chaîne de caractère peut être affichée en utilisant la fonction `printf` et le code de formatage `%s` :

```
char chaine[]="abcdef";
printf("la chaine est : %s",chaine);
La chaine est abcdef
```

De la même façon, on peut saisir une chaîne de caractères grâce à la fonction `scanf` et au code de formatage `%s` :

```
char chaine[10];
printf("Entrez la chaine : ");
scanf("%s",chaine);
Entrez la chaine : ghijkl
```

La saisie est terminée par certains caractères spéciaux comme `'\n'`, mais également par le caractère d'espacement `' '`, ce qui constitue un inconvénient si on veut saisir des chaînes contenant des espaces.

La fonction `gets` permet également la saisie de chaînes de caractères :

```
char chaine[10];
printf("Entrez la chaine : ");
gets(chaine);
Entrez la chaine : ghijkl
```

Au contraire de `scanf`, `gets` peut saisir les espaces. Par exemple, si l'utilisateur saisit la chaîne `abcd efg hijk` :

- en utilisant `scanf`, le tableau `chaine` contiendra les valeurs suivantes :

```
{'a', 'b', 'c', 'd', '\0'}
```

- en utilisant `gets`, le tableau `chaine` contiendra les valeurs suivantes :

```
{'a', 'b', 'c', 'd', ' ', 'e', 'f', 'g', ' ', 'h', 'i', 'j', 'k', '\0'}
```

Enfin, la fonction `puts` permet d'afficher une chaîne de caractères :

```
char chaine[]="abcdef";
puts(chaine);
abcdef
```

2 Exercices

Exercice 1

Écrivez un programme qui :

1. Demande à l'utilisateur de saisir une chaîne de caractères.
2. Calcule la longueur de la chaîne, c'est-à-dire le nombre de caractères tapés par l'utilisateur.

exemple :

```
Entrez une chaine de caracteres : abcd
Il y a 4 caracteres dans la chaine "abcd".
```

Remarque : vous devez obtenir exactement l'affichage ci-dessus, notamment les guillemets. Attention à ne pas confondre la longueur de la chaîne et la taille du tableau qui la contient.

Exercice 2

Écrivez un programme demande à l'utilisateur de saisir une chaîne de caractère `chaine1[5]`, puis qui recopie cette chaine dans un autre tableau `chaine2[5]`.

exemple :

```
Entrez une chaine de caracteres : abc
La chaine 2 est : abc
```

La variable `chaine2` pourrait-elle être un tableau contenant moins de 5 caractères ? Plus de 5 caractères ?

Exercice 3

Écrivez un programme qui :

- Demande à l'utilisateur de saisir deux chaînes de caractères, qui sont stockées dans les tableaux `chaine1[5]` et `chaine2[5]`.
- Concatène ces deux chaînes pour obtenir une troisième chaine `chaine3[9]`.
- Affiche `chaine3`.

exemple :

```
Entrez la chaine 1 : abc
Entrez la chaine 2 : defg
La chaine 3 est leur concatenation : abcdefg
```

Pourquoi le tableau `chaine3` n'a-t-il pas une taille de 10 caractères ?

Exercice 4

Écrivez un programme qui saisit puis compare deux chaînes de caractères selon l'ordre lexicographique de la table ASCII.

exemple 1 :

```
Entrez la chaine 1 : def
Entrez la chaine 2 : abc
Resultat : def > abc
```

exemple 2 :

```
Entrez la chaine 1 : abc
Entrez la chaine 2 : abcd
Resultat : abc < abcd
```

exemple 3 :

```
Entrez la chaine 1 : abc
Entrez la chaine 2 : abc
Resultat : abc = abc
```

Rappel : l'ordre lexicographique est une généralisation de l'ordre alphabétique aux caractères qui ne sont pas des lettres (chiffres, ponctuation, etc.).

Exercice 5

Écrivez un programme qui initialise une chaine de caractères `chaine1`, puis qui en fait une copie inversée `chaine2` et l'affiche.

exemple :

```
La chaine 1 est abcd
La chaine 2 est son inversion : dcba
```

Attention : cette fois on ne demande pas à l'utilisateur de saisir la chaîne, elle est directement définie dans le programme.

Exercice 6

Écrivez un programme qui fait la même chose qu'à l'exercice précédent, mais cette fois sans utiliser un deuxième tableau. Autrement dit, vous devez travailler directement dans `chaine1`. Vous avez le droit à une variable `temp` de type `char`.

Pour une chaîne de longueur n, vous devez pour cela appliquer le principe suivant :

- Intervertir le caractère 0 et le caractère $n-1$;
- Intervertir le caractère 1 et le caractère $n-2$
- Intervertir le caractère 2 et le caractère $n-3$
- Etc.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 07 | tableaux multidimensionnels

Présentation

Le but de ce TP est de manipuler des tableaux multidimensionnels d'entiers. On étudie d'abord leur stockage en mémoire, puis on effectue des calculs simples sur des tableaux à 2 dimensions.

1 Occupation mémoire

Exercice 1

Soit un tableau `short m[N][P][Q]` pour $N = 4$, $P = 3$ et $Q = 2$. Dessinez la représentation graphique de son occupation de la mémoire.

Exercice 2

Donnez les formules permettant de calculer les adresses de `m[i]`, `m[i][j]`, et `m[i][j][k]` en fonction de :

- Adresse du tableau : `m` ;
- Type des données qu'il contient : `short` ;
- Index : `i`, `j` et `k`.

Écrivez un programme permettant de vérifier votre calcul : il demande à l'utilisateur de saisir les valeurs `i`, `j` et `k` et affiche chaque adresse de deux façons différentes :

- Adresses directement obtenues en appliquant l'opérateur d'adressage `&` aux 3 expressions mentionnées au début de l'exercice ;
- Adresses calculées avec vos 3 formules.

Bien sûr, si vos formules sont correctes, vous devez obtenir les mêmes adresses avec les deux méthodes.

exemple : (les `??????` représentent des adresses connues seulement à l'exécution)

```
Entrez i,j,k (avec les virgules) : 1,2,0
Adresse du tableau m : 0x??????
Adresse réelle de   m[1] : 0x??????
Adresse calculée de m[1] : 0x??????
Adresse réelle de   m[1][2] : 0x??????
Adresse calculée de m[1][2] : 0x??????
Adresse réelle de   m[1][2][0] : 0x??????
Adresse calculée de m[1][2][0] : 0x??????
```

Remarques :

- `N`, `P` et `Q` doivent être déclarées comme des constantes.
- La fonction `sizeof(t)` permet de déterminer combien d'octets un type `t` occupe en mémoire. Par exemple, `sizeof(char)` renvoie la valeur 1.

Exercice 3

Écrivez un programme qui initialise ce tableau de manière à ce que l'élément situé à l'adresse `m+2x` contienne la valeur `x`. Effectuez une vérification en utilisant vos formules de l'exercice précédent, qui vous permettent de calculer `2x` en fonction de `i`, `j`, et `k`.

exemple :

```
Entrez i,j,k (avec les virgules) : 1,2,0
     Valeur theorique : xxxxx
     Valeur effective  : xxxxx
```

2 Opérations matricielles

Pour des raisons pratiques, on se concentre ici sur des matrices carrées de dimension $N \times N$.

Exercice 4

Écrivez un programme permettant d'afficher une matrice `int m[N][N]` à l'écran, comme ci-dessous.

exemple : affichage de la matrice identité

```
1  0  0  0
0  1  0  0
0  0  1  0
0  0  0  1
```

Remarque : on suppose la matrice contient uniquement des valeurs comprises entre 0 et 99.

Exercice 5

Écrivez un programme qui place dans une matrice `res[N][N]` le résultat de la multiplication d'une matrice `m[N][N]` par un scalaire `int s`. Vous devez initialiser `m` en utilisant la méthode que vous voulez, puis calculer le produit et afficher son résultat `res`. Utilisez (en l'adaptant) le code source de l'exercice précédent pour effectuer l'affichage du résultat.

Exercice 6

Écrivez un programme qui calcule la transposée d'une `m[N][N]` et place le résultat dans la matrice `res[N][N]`. Initialisez et affichez le résultat comme dans l'exercice précédent.

Exercice 7

Écrivez un programme qui calcule la somme de deux matrices `m1[N][N]` et `m2[N][N]` et place le résultat dans la matrice `res[N][N]`. Initialisez et affichez le résultat comme dans les exercices précédents.

Exercice 8

Même chose avec le produit de deux matrices `m1[N][N]` et `m2[N][N]`, dont le résultat est à placer dans une troisième matrice `res[N][N]`.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 08 | diagrammes en mode texte

Présentation

Le but de ce TP est de définir et d'utiliser des fonctions secondaires (par opposition à la fonction principale `main`). De nombreuses fonctions ont déjà été utilisées dans les TP précédents, en particulier pour l'affichage et la saisie de texte. Dans ce TP, nous allons définir nos propres fonctions.

1 Structure des programmes

À partir de ce TP, les programmes seront structurés en utilisant des fonctions. Autrement dit, on ne va plus mettre tout le code source dans la fonction `main` comme auparavant, mais on va plutôt utiliser plusieurs fonctions distinctes. Les noms des fonctions sont donnés dans les exercices.

Attention cependant, car à chaque fois que vous écrivez une nouvelle fonction, vous devez la tester pour vous assurer qu'elle fonctionne. Pour cela, vous devez l'appeler à partir de la fonction `main`, et éventuellement afficher les résultats obtenus si nécessaire.

Un programme aura donc maintenant la forme suivante :

```
int ma_function1(int mon_parametre)
{  ...// traitement de la première fonction
}

float ma_function2(int p1, float p2)
{  ...// traitement d'une autre fonction
}

... // définition d'autres fonctions

int main()
{  // exercice 1
   ...// test de la première fonction

   // exercice 2
   ...// test de la deuxième fonction

   ...// etc.

   return EXIT_SUCCESS;
}
```

2 Carrés

Exercice 1

Écrivez une fonction `void affiche_ligne(int n, char car)` qui affiche `n` fois le caractère `car` suivi du caractère espace `' '`, et finit par un retour à la ligne.

exemples :

- L'appel `affiche_ligne(3,'*')` produira l'affichage suivant :

```
* * *
```

- L'appel `affiche_ligne(5,'+')` produira l'affichage suivant :

```
+ + + + +
```

Exercice 2

En utilisant la fonction `affiche_ligne`, écrivez une fonction `void affiche_carre(int cote, char car)` qui affiche un carré plein à l'aide du caractère `car`, comme dans l'exemple ci-dessous.

exemple : l'appel `affiche_carre(5,'*')` provoque l'affichage suivant :

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

3 Triangles

Exercice 3

En utilisant la fonction `affiche_ligne`, écrivez une fonction `void affiche_triangle(int cote, char car)` qui affiche un triangle plein à l'aide du caractère `car` comme dans l'exemple ci-dessous.

exemple : l'appel `affiche_triangle(5,'*')` provoque l'affichage suivant :

```
*
* *
* * *
* * * *
* * * * *
```

Exercice 4

On voudrait maintenant afficher un triangle, mais en contrôlant son orientation. Écrivez la fonction `affiche_triangle2`, qui a le même en-tête que `affiche_triangle`, avec en plus un nouveau paramètre `direction`. Si ce paramètre vaut 0, l'hypoténuse est tournée vers le haut, comme dans l'exemple précédent. Si le paramètre vaut 1, elle est tournée vers le bas, comme dans l'exemple ci-dessous. Vous devez réutiliser la fonction `affiche_triangle`.

exemple : l'appel `affiche_triangle2(6,'*',1)` provoque l'affichage suivant :

```
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

Exercice 5

En utilisant uniquement la fonction `affiche_triangle2`, écrivez la fonction `void affiche_grand_triangle(int cote, char car)`, qui affiche un triangle du même type que celui donné en exemple ci-dessous.

exemple : l'appel `affiche_grand_triangle(5,'*')` provoque l'affichage suivant :

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

4 Croix

Exercice 6

En utilisant la fonction `affiche_ligne`, écrivez une fonction `void affiche_ligne2(int n1, int n2, char car)` qui affiche `n1` fois 2 caractères espace, puis `n2` fois le caractère `car` suivi du caractère espace `' '`.

exemples : (les caractères espace sont représentés sous forme d'underscores (`'_'`) pour que les exemples soient bien compréhensibles)

- L'appel `affiche_ligne(3,4,'*')` produira l'affichage suivant :

```
____* * * *
```

- L'appel `affiche_ligne(8,2,'+')` produira l'affichage suivant :

```
________________+ +
```

- L'appel `affiche_ligne(0,3,':')` produira l'affichage suivant :

```
: : :
```

Exercice 7

En utilisant la fonction `affiche_ligne2`, écrivez une fonction `void affiche_croix(int cote, char car)` qui dessine une croix comme dans l'exemple ci-dessous. Le paramètre `cote` contrôle la longueur des branches de la croix.

exemple : l'appel `affiche_croix(5,'*')` provoque l'affichage suivant :

```
          *
          *
          *
          *
* * * * * * * * * *
          *
          *
          *
          *
```

Exercice 8

Écrivez une fonction `affiche_croix2(int n, int largeur, char car)` qui prend un paramètre supplémentaire par rapport à la fonction précédente `affiche_croix` : le paramètre `largeur`, qui contrôle l'épaisseur des branches de la croix.

exemple : l'appel `affiche_croix2(6,3,'*')` provoque l'affichage suivant :

```
          * * *
          * * *
          * * *
          * * *
* * * * * * * * * * * *
* * * * * * * * * * * *
* * * * * * * * * * * *
          * * *
          * * *
          * * *
          * * *
```

Remarque : on suppose que le paramètre `largeur` prend une valeur *impaire*.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 09 | introduction à la SDL

Présentation

La SDL (*Simple Direct Media Layer*) est une bibliothèque que nous utiliserons pour programmer des applications graphiques. Il s'agit d'un projet libre (LGPL) et multiplateforme Pour savoir comment installer et utiliser la SDL avec Eclipse et la CDT, veuillez consulter les annexes du cours sur kikencere.

Le but de ce TP est de se familiariser avec cette bibliothèque, son utilisation, et les notions relatives à la représentation des informations graphiques, en particulier les couleurs.

1 Création d'un projet

La SDL est une bibliothèque possédant de nombreuses fonctionnalités, et à cause de cela elle est peut paraitre compliquée. Pour cette raison, vous ne l'utiliserez pas directement : vous passerez uniquement par des fonctions prédéfinies spécialement pour chaque TP. Ces fonctions sont réunies dans une bibliothèque appelée `graphisme`.

L'archive fournie avec ce sujet contient cette bibliothèque `graphisme`, ainsi qu'un programme `main.c`. Notez que la librairie se compose de deux fichiers : le fichier d'en-tête (*header* en anglais, d'où le `.h`) `graphisme.h` et le fichier de définition `graphisme.c`. Le fichier d'en-tête contient :

- Les directives d'inclusion permettant d'utiliser d'autres bibliothèques :
  - `#include <mabibliotheque.h>` : bibliothèques installées.
  - `#include "mabibliotheque.h"` : bibliothèques locales au projet.
- Les définitions de constantes (macros) : `#define FENETRE_LARGEUR 800`
- Les en-têtes (ou déclarations) de fonctions : `void efface_fenetre(void);`

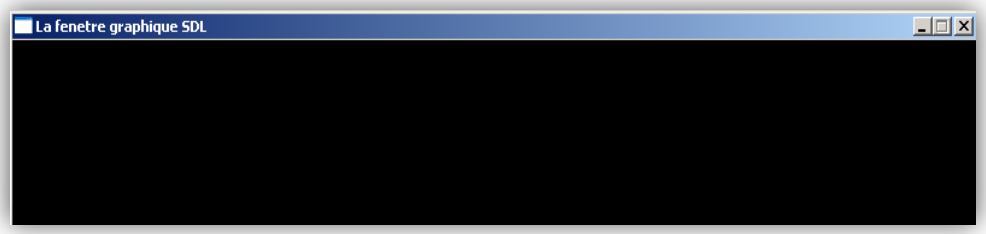
Le fichier de définition contient :

- Une directive d'inclusion pour son fichier `.h` associé.
- L'implémentation de chaque fonction, avec la description de son rôle sous forme de commentaire.

Exercice 1

Créez un nouveau projet, et copiez-y les fichiers contenus dans l'archive fournie avec ce sujet. Pour utiliser la bibliothèque SDL, vous devez d'abord modifier la configuration de votre projet. Pour cela, suivez les instructions de l'annexe concernant l'utilisation de la SDL, disponible sur kikencere.

Compilez et exécutez votre projet : une fenêtre graphique vide (toute noire) devrait s'afficher. Appuyez sur une touche du clavier pour fermer cette fenêtre et terminer le programme.



2 Fenêtre graphique

Exercice 2

Dans la fonction `main`, la fonction `initialise_fenetre` provoque la création et l'ouverture de la fenêtre graphique. La chaîne de caractère qu'elle prend en paramètre est le titre de la fenêtre. Écrivez vos prénom et nom à la place de ce titre.

Les dimensions de votre fenêtre graphique sont définies dans le fichier `graphisme.h` par des constantes `FENETRE_LARGEUR` et `FENETRE_HAUTEUR`. Modifiez-les de manière à obtenir une fenêtre carrée de dimensions 600×600.

Remarque : il peut être nécessaire de cliquer sur *Project* > *Clean…* pour que les modifications soient prises en compte à l'exécution.

3 Représentation en mémoire

La fenêtre graphique est enregistrée en mémoire sous forme d'une matrice appelée *surface*, dont chaque élément représente un pixel de la fenêtre. Un type spécifique appelé `SDL_Surface` est chargé d'implémenter cette matrice.

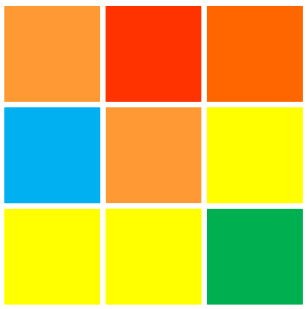
Le nombre de bits alloué au codage d'un pixel (*bits per pixel* ou *bpp*) est défini dans la constante `BPP` de `graphisme.h`. La valeur utilisée dans `graphisme.h` est de 32 bits pour un codage en mode RVB (Rouge-Vert-Bleu), soit 4 octets :

- Un octet est alloué pour représenter l'intensité de chacune des composantes R, V et B, soit 3 octets au total.
- Le 4ème et dernier octet est appelé *canal alpha*. Il permet de gérer la transparence des images. La transparence est utilisée pour créer une image composée en superposant deux images.

La figure ci-dessous illustre ce principe de représentation, pour une fenêtre de 3×3 pixels.

$r = 255$ $v = 153$ $b = 51$ $\alpha = 255$	$r = 255$ $v = 51$ $b = 0$ $\alpha = 255$	$r = 255$ $v = 153$ $b = 51$ $\alpha = 255$
$r = 0$ $v = 176$ $b = 240$ $\alpha = 255$	$r = 255$ $v = 153$ $b = 51$ $\alpha = 255$	$r = 255$ $v = 255$ $b = 0$ $\alpha = 255$
$r = 255$ $v = 255$ $b = 0$ $\alpha = 255$	$r = 255$ $v = 255$ $b = 0$ $\alpha = 255$	$r = 0$ $v = 176$ $b = 80$ $\alpha = 255$

Surface



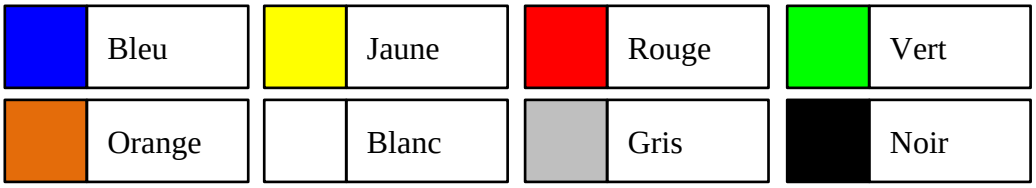
Fenêtre

Dans le cadre d'un projet, on peut donc représenter une couleur de deux façons différentes : une séquence de 4 entiers codés chacun sur 1 octets, ou bien 1 seul entier codé sur 4 octets. La SDL utilise deux types spécifiques pour cela : le type `Uint8` (un octet, comme son nom l'indique) et le type `Uint32` (4 octets). La bibliothèque `graphisme` offre

deux fonctions permettant de passer d'un format à l'autre : `convertis_rvb` (de `Uint8` vers `Uint32`) et `convertis_couleur` (de `Uint32` vers `Uint8`).

Exercice 3

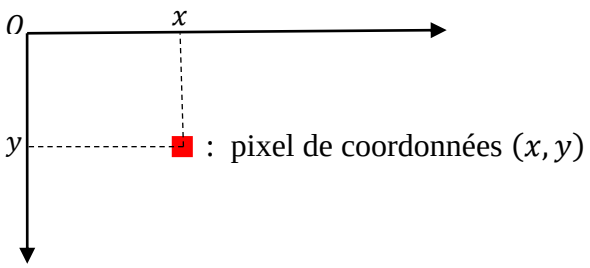
Définissez des constantes (macros) représentant les couleurs suivantes :



Nommez-les selon la forme suivante : `C_COULEUR`. Par exemple : `C_BLEU`, `C_ORANGE`, etc. Pour définir leur valeur, utilisez la fonction `convertis_rvb`. Pour identifier les composantes rouge-vert-bleu à utiliser, vous pouvez soit procéder par essais-erreurs, soit consulter le web (ex : cette page Wikipedia).

4 Dessin dans la fenêtre

Il est important de remarquer que l'origine du repère servant à définir les coordonnées des pixels de la surface n'est *pas* située en bas à gauche, comme c'est la convention en mathématiques. Pour des raisons matérielles, elle est plutôt fixée *en haut* à gauche de l'écran :



Les modifications réalisées sur une surface ne sont pas directement affichées à l'écran. Il existe, comme pour les entrées et sorties standards, un tampon (*buffer*) entre la surface et l'écran. Il faut provoquer le vidage du tampon pour obtenir la mise à jour de l'affichage. Ceci est réalisé grâce à la fonction `rafraichis_fenetre`.

Exercice 4

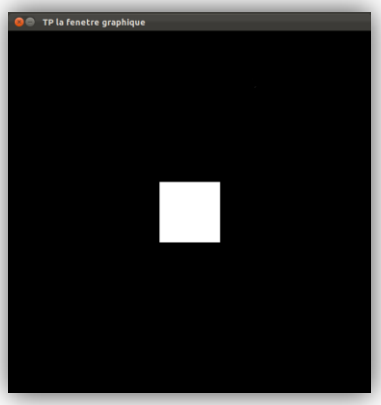
La bibliothèque `graphisme` contient la fonction : `void allume_pixel(int x, int y, Uint32 coul)` qui sert à modifier la couleur d'un pixel de la fenêtre. Ses paramètres sont :

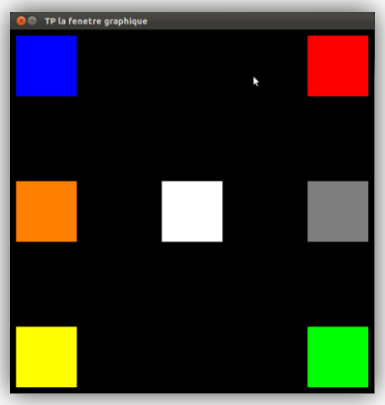
- `x` : l'abscisse du pixel concerné.
- `y` : l'ordonnée du pixel concerné.
- `coul` : la nouvelle couleur du pixel, sous forme d'un entier de type `Uint32`.

Dans le fichier `main.c`, écrivez une fonction `trace_carre(int x, int y, int c, Uint32 coul)` qui dessine à l'écran un carré *plein* dont le sommet supérieur gauche a pour coordonnées `(x,y)` et dont le côté a pour longueur `c`. Le paramètre `coul` correspond à la couleur à utiliser pour dessiner et remplir le carré.

Exercice 5

Dans la fonction `main`, affichez un carré blanc au centre de votre écran. Le résultat obtenu devrait ressembler à la capture d'écran de *gauche*, ci-dessous :





Remarque : pensez à rafraichir la fenêtre graphique !

Exercice 6

Écrivez une fonction `trace_figure()` qui dessine la figure suivante, composée de 7 carrés de couleurs différentes :

- En haut à droite, un carré rouge.
- Au centre, un carré blanc.
- En bas à droite, un carré vert.
- En haut à gauche, un carré bleu.
- En bas à gauche, un carré jaune.
- Au milieu à gauche, un carré orange.
- Au milieu à droite, un carré gris.

Bien entendu, vous devez tester cette fonction en complétant la fonction `main`. Vous devez obtenir quelque chose de similaire à la capture d'écran de *droite*, ci-dessus.

5 Animation de la fenêtre

La fonction `void attends_delai(int delai)` définie dans graphisme prend en paramètre un entier `delai` représentant un temps exprimé en ms. La fonction bloque l'affichage durant ce temps-là. Cette fonction permet d'animer l'image affichée à l'écran : il suffit d'afficher la surface, puis attendre un peu, puis afficher une modification, puis attendre un peu, puis afficher une autre modification, attendre un peu, etc. Sans cette attente, l'affichage serait si rapide qu'un être humain n'aurait pas le temps de voir les différentes étapes de l'animation.

Exercice 7

Définissez une constante (macro) `DELAI` dont la valeur correspond au délai d'attente que vous voulez utiliser dans votre animation. L'intérêt de cette constante est qu'elle vous permettra ensuite de contrôler facilement la vitesse de l'animation. Le délai recommandé ici est d'une seconde (donc 1000 ms).

Ensuite, écrivez une fonction `void anime_figure()` qui effectue l'animation suivante :

- Affiche en haut à droite de l'écran un carré rouge ;
- Déplace le carré au centre de l'écran et il devient blanc ;
- Déplace le carré en bas à droite et il devient vert ;
- Déplace le carré en haut à gauche et il devient bleu ;
- Déplace le carré en bas à gauche et il devient jaune ;
- Déplace le carré au milieu à gauche et il devient orange ;
- Déplace le carré au milieu à droite et il devient gris ;
- Replace le carré en haut à droite et il redevient rouge.

Testez votre fonction `anime_figure` depuis la fonction `main`.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 10 | modification des couleurs

Présentation

Ce TP est basé sur l'utilisation de la SDL, donc vous devez configurer votre projet de manière à pouvoir utiliser cette bibliothèque, comme expliqué dans l'annexe disponible sur kikencere. Une archive contenant la bibliothèque `graphisme` et un album de photos est fournie avec ce sujet. Notez que `graphisme` a été complétée avec de nouvelles fonctions, dont vous aurez besoin dans ce TP. Le but du TP est de modifier les photos en appliquant différents algorithmes de modification des couleurs.

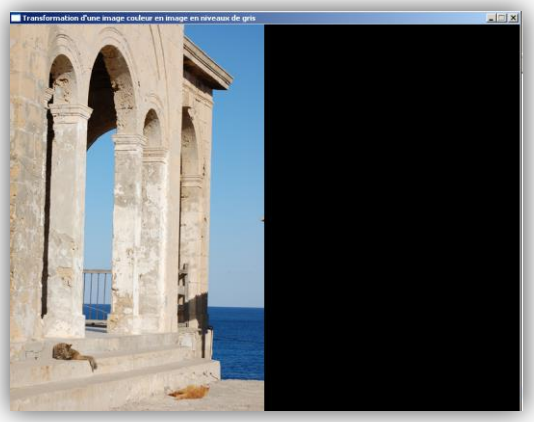
1 Préparation

Exercice 1

Modifiez la fonction `main` afin d'afficher à gauche de l'écran la photo nommée `image1.bmp` contenue dans l'album fourni avec ce sujet. Vous aurez besoin des fonctions suivantes, inclues dans `graphisme` :

- `charge_image` : crée une surface à partir d'un fichier bitmap, c'est-à-dire une image présente en mémoire, mais encore affichée.
- `dessine_surface` : recopie une surface dans la fenêtre graphique.

Vous devez obtenir le résultat suivant :



2 Niveaux de gris

Une image en niveau de gris est composée de pixels dont l'intensité lumineuse varie entre le noir et le blanc. Un pixel d'une image RVB est de couleur grise si toutes ses trois composantes sont égales.

Pour transformer une image couleur en une image en niveaux de gris, on peut utiliser l'algorithme suivant :

- Pour chaque pixel de l'image :
  - On détermine ses quantités de rouge, vert et bleu du pixel.

- On calcule la moyenne de ces trois valeurs, qui correspond au niveau de gris du pixel.
- On calcule de nouvelles composantes RVB, chacune égale à cette valeur.
- On applique ces nouvelles composantes au pixel situé dans la surface résultat.

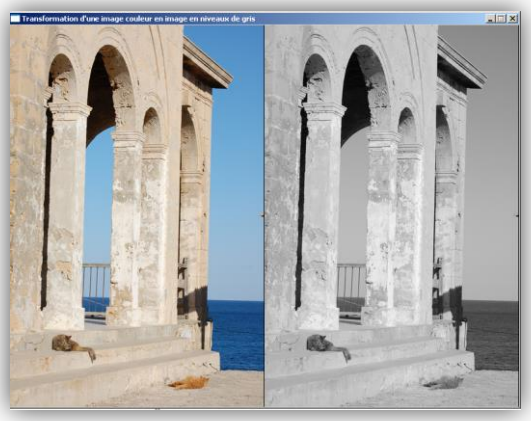
Exercice 2

Écrivez une fonction `SDL_Surface* convertis_gris(SDL_Surface* source)` qui reçoit une image sous la forme de la surface source, et qui crée et renvoie une nouvelle surface contenant la même photo que `source`, mais en niveau de gris. Vous aurez besoin des fonctions suivantes, inclues dans la bibliothèque `graphisme` :

- `initialise_surface` : crée une nouvelle surface vide.
- `composantes_pixel_surface` : renvoie les composantes RVB pour un pixel situé dans une image donnée.
- `allume_pixel_surface` : dessine un pixel dans une surface passée en paramètre.

De plus, il est possible d'accéder aux dimensions d'une surface au moyen de deux champs `w` (*width*) et `h` (*height*) permettant d'obtenir ses largeur et hauteur, exprimées en pixels.

Dans la fonction `main`, affichez à droite de l'écran la version en niveau de gris de la photo `image1.bmp`. Vous devez obtenir le résultat suivant :



Exercice 3

Enregistrez l'image en niveau de gris, sous la forme d'un fichier bitmap appelé `image1.nivgris.bmp`. Vous aurez pour cela besoin de la fonction `sauve_surface` incluse dans `graphisme`.

Remarque : le chemin pris en paramètre par `sauve_surface` est relatif à la *racine* du projet. Pour créer une image appelée `image.bmp` et située dans le dossier `album` de votre projet, vous devez donc utiliser le chemin `"album/image.bmp"`.

Visualisez l'image créée, en utilisant l'outil par défaut de votre système d'exploitation (double-clic sur l'image dans le *Project Explorer* d'Eclipse), afin de vérifier que l'image obtenue est bien correcte.

Remarque : Eclipse ne rafraichit pas automatiquement le *Project Explorer*. Après avoir créé le fichier demandé, vous devez donc sélectionner le dossier `album` et demander son rafraichissement manuellement (clic-droit puis *Refresh*, ou bien touche *F5*), pour voir apparaitre le nouveau fichier.

3 Balance des couleurs

On s'intéresse maintenant à des modifications permettant de changer l'équilibre des couleurs dans une photo. Il ne s'agit donc plus de convertir une photo couleur en une photo en niveaux de gris, mais plutôt de modifier les couleurs de la photo.

Exercice 4

La modification de la balance des couleurs est réalisée en augmentant ou en diminuant chaque composante RVB. Cependant, les composantes étant codées sur un seul octet, il est nécessaire que le résultat de la modification reste compris dans $[0; 255]$. Pour cela, on va utiliser une fonction de seuil. Soit x la valeur originale de la composante et y la modification qu'on veut lui apporter. Alors la fonction de seuil $f : \mathbb{N}^2 \longrightarrow \{0,1, \dots ,255\}$ est définie par :

$$f(x,y) = \begin{cases} 0 \text{ si } x+y < 0 \\ 255 \text{ si } x+y > 255. \\ x+y \text{ sinon} \end{cases}$$

Écrivez la fonction (informatique) `seuil` qui implémente la fonction (mathématique) f. Réfléchissez bien à son en-tête. Depuis la fonction `main`, testez votre fonction `seuil`.

Remarque : si vous utilisez MS Window avec la SDL, la sortie standard sera redirigée vers un fichier `stdout.txt` généré dans le répertoire `debug` de votre projet. C'est donc là qu'il faut regarder pour avoir les résultats de vos tests.

Exercice 5

Écrivez la fonction d'en tête `SDL_Surface* modifie_couleurs(SDL_Surface* source, int delta_r, int delta_v, int delta_b)`, qui reçoit en paramètre la photo originale `source`, et qui renvoie une nouvelle surface, représentant la photo dont les couleurs ont été modifiées. La modification est la suivante :

- Pour chaque pixel de la photo originale :
  - On détermine ses quantités de rouge, vert et bleu du pixel.
  - On calcule de nouvelles composantes en ajoutant respectivement `delta_r`, `delta_v`, et `delta_b` aux composantes rouge, verte et bleue.
  - On applique ces nouvelles composantes au pixel situé dans la surface résultat.

Remarque : vous devez, bien entendu, utiliser la fonction `seuil` pour effectuer le calcul des nouvelles composantes.

Dans la fonction `main`, affichez à droite de l'écran la version modifiée de la photo `image1.bmp`. Par exemple, si on applique `modifie_couleurs(source, 100, 0, 0)`, on va ajouter 100 à la composante rouge sans modifier les deux autres composantes. La photo paraîtra donc plus rouge, comme illustré ci-dessous :

Qu'observez-vous si vous utilisez les valeurs $(100,100,100)$ comme deltas ? Et pour $(-100,-100,-100)$?

Exercice 6

Enregistrez l'image modifiée, sous la forme d'un fichier bitmap appelé `image1.modif.bmp`.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 11 | passage de paramètres

Présentation

Le but de ce TP est de manipuler et de mieux comprendre le fonctionnement des deux méthodes de passage de paramètres vues en cours : le passage par valeur et le passage par adresse.

Remarques :

- Pour répondre aux questions demandant une réponse textuelle (par opposition à un programme), donnez vos explications dans la fonction `main`, sous forme de commentaires.
- Dans les exemples de ce TP, les `??` représentent en réalité des valeurs numériques. Celles-ci ne sont pas indiquées car le but des exercices est de les deviner avant d'exécuter le programme.

1 Expérimentations

Exercice 1

Écrivez une fonction `void test1(int x)` qui reçoit un entier `x` et affiche simplement sa valeur et son adresse. Pour vérifier `test1`, effectuez les opérations suivantes dans la fonction principale `main` :

1. Déclarez et initialisez une variable `x` ;
2. Affichez sa valeur et son adresse avec `printf` ;
3. Appelez la fonction `test1` en lui passant `x` en paramètre ;
4. Affichez encore une fois la valeur et l'adresse de `x`.

exemple :

```
@main : l'adresse de x avant l'appel est 0x??????
@main : la valeur de x avant l'appel est ??
@test1 : la valeur de x est ??
@test1 : l'adresse de x est 0x??????
@main : la valeur de x apres l'appel est ??
@main : l'adresse de x apres l'appel est 0x??????
```

Remarque : pour afficher une adresse avec `printf`, on utilise le format `%p` (au lieu de `%d`).

Avant d'exécuter votre programme, tentez de deviner quelles valeurs vont être affichées à la place des `??` de l'exemple ci-dessus. Cette consigne est valable pour les autres exercices.

Comparez les adresses affichées pour `x` : comment expliquez-vous les différences observées ?

Exercice 2

On garde *exactement* la même fonction `test1`, mais cette fois on procède différemment dans la fonction `main` : au lieu de déclarer une variable `x`, on utilise une variable `n`.

exemple :

```
@main : l'adresse de n avant l'appel est 0x??????
@main : la valeur de n avant l'appel est ??
@test1 : la valeur de x est ??
```

```
@test1 : l'adresse de x est 0x??????
@main : la valeur de n apres l'appel est ??
@main : l'adresse de n apres l'appel est 0x??????
```

Qu'observez-vous ? Comparez avec l'affichage obtenu à l'exercice précédent, et donnez une justification.

Exercice 3

Écrivez une fonction `void test2(int x)` qui reçoit un entier `x`, affiche sa valeur et son adresse, *modifie* `x` de manière à la diviser par 2, puis affiche la nouvelle valeur de `x`. Dans la fonction principale `main`, effectuez les opérations suivantes :

1. Déclarez et initialisez une variable `n` ;
2. Affichez sa valeur et son adresse ;
3. Appelez la fonction `test2` en lui passant `n` en paramètre ;
4. Affichez encore une fois la valeur et l'adresse de `n`.

exemple :

```
@main : la valeur de n avant l'appel est ??
@main : l'adresse de n avant l'appel est 0x??????
@test2 : la valeur de x est ??
@test2 : l'adresse de x est 0x??????
@test2 : la valeur de x apres la division est ??
@main : la valeur de n apres l'appel est ??
@main : l'adresse de n apres l'appel est 0x??????
```

Exercice 4

Écrivez une fonction `int test3(int x)` qui reçoit un entier `x`, affiche sa valeur et son adresse, calcule `x` divisé par 2, affiche le résultat de ce calcul et *renvoie ce résultat par valeur*. Dans la fonction principale `main`, effectuez les opérations suivantes :

1. Déclarez et initialisez une variable `n` ;
2. Affichez sa valeur et son adresse ;
3. Appelez la fonction `test3` en lui passant `n` en paramètre, et utilisez la valeur renvoyée par `test3` pour mettre `n` à jour ;
4. Affichez encore une fois la valeur et l'adresse de `n`.

exemple :

```
@main : la valeur de n avant l'appel est ??
@main : l'adresse de n avant l'appel est 0x??????
@test3 : la valeur de x est ??
@test3 : l'adresse de x est 0x??????
@test3 : le resultat de la division est ??
@main : la valeur de n apres l'appel est ??
@main : l'adresse de n apres l'appel est 0x??????
```

Exercice 5

Écrivez une fonction `void test4(int x, int* resultat)` qui reçoit un paramètre passé par valeur `x` et un paramètre passé par adresse `resultat`. Cette fonction doit effectuer les opérations suivantes :

1. Afficher la valeur et l'adresse de `x` ;
2. Afficher l'adresse correspondant à `resultat` ainsi que la valeur située à cette adresse ;
3. Diviser `x` par 2 et mettre le résultat à l'adresse indiquée par `resultat` ;
4. Afficher de nouveau la valeur et l'adresse de `x` ;
5. Afficher de nouveau l'adresse correspondant à `resultat` et la valeur située à cette adresse.

Remarque : attention, le paramètre `resultat` est lui-même une adresse, il ne s'agit pas d'une variable classique.

Dans la fonction principale `main`, effectuez les opérations suivantes :

1. Déclarez et initialisez une variable `n` ;
2. Déclarez une variable `r` (il n'est pas obligatoire de l'initialiser) ;
3. Affichez les adresses et les valeurs de ces variables ;
4. Appelez la fonction `test4` en lui passant `n` et l'adresse de `r` en paramètres ;
5. Affichez encore une fois les valeurs et adresses de `n` et `r`.

exemple :

```
@main : la valeur de n avant l'appel est ??
@main : l'adresse de n avant l'appel est 0x??????
@main : la valeur de r avant l'appel est ??
@main : l'adresse de r avant l'appel est 0x??????
@test4 : la valeur de x est ??
@test4 : l'adresse de x est 0x??????
@test4 : l'adresse indiquee par le parametre resultat est 0x??????
@test4 : la valeur situee a cette adresse est ??
@test4 : la valeur de x apres la division est ??
@test4 : l'adresse de x apres la division est 0x?
@test4 : l'adresse indiquee par resultat apres la division est 0x??????
@test4 : la valeur situee a cette adresse apres la division est ??
@main : la valeur de n apres l'appel est ??
@main : l'adresse de n apres l'appel est 0x??????
@main : la valeur de r apres l'appel est ??
@main : l'adresse de r apres l'appel est 0x??????
```

Expliquez pourquoi il est inutile d'initialiser la variable `r` dans la fonction `main`.

Comparez l'adresse de `r` dans la fonction `main` et celle indiquée par le paramètre `resultat` dans la fonction `test4` : qu'observez-vous (justifiez) ?

Exercice 6

Écrivez une fonction `void test5(int* x)` qui reçoit un paramètre passé par adresse `x`, et effectue les opérations suivantes :

1. Afficher l'adresse correspondant à `x` ainsi que la valeur située à cette adresse ;
2. Modifier cette valeur en la divisant par 2 ;
3. Afficher de nouveau l'adresse correspondant à `x` et la valeur située à cette adresse.

Dans la fonction principale `main`, effectuez les opérations suivantes :

1. Déclarez et initialisez une variable `n` ;
2. Affichez l'adresse et la valeur de cette variable ;
3. Appelez la fonction `test5` en lui passant l'adresse de `n` en paramètre ;
4. Affichez encore une fois la valeur et l'adresse de `n`.

exemple :

```
@main : la valeur de n avant l'appel est ??
@main : l'adresse de n avant l'appel est 0x??????
@main : la valeur de r avant l'appel est ??
@main : l'adresse de r avant l'appel est 0x??????
@test4 : la valeur de x est ??
@test4 : l'adresse de x est 0x??????
@test4 : l'adresse indiquee par le parametre resultat est 0x??????
@test4 : la valeur situee a cette adresse est ??
@test4 : la valeur de x apres la division est ??
@test4 : l'adresse de x apres la division est 0x?
@test4 : l'adresse indiquee par resultat apres la division est 0x??????
@test4 : la valeur situee a cette adresse apres la division est ??
@main : la valeur de n apres l'appel est ??
@main : l'adresse de n apres l'appel est 0x??????
@main : la valeur de r apres l'appel est ??
@main : l'adresse de r apres l'appel est 0x??????
```

Est-il nécessaire d'initialiser la variable `n` dans la fonction `main` ?

Comparez l'adresse de `n` dans la fonction `main` et celle indiquée par le paramètre `x` dans la fonction `test5` : qu'observez-vous (justifiez) ?

2 Fonctions mathématiques

Exercice 7

Écrivez une fonction `vabs` qui calcule et retourne la valeur absolue d'un réel `x` de type `float`. La fonction doit recevoir `x` par valeur et renvoyer sont résultat par valeur.

N'oubliez pas de tester votre fonction à partir de la fonction `main`. Cette remarque est valide pour tous les exercices.

Exercice 8

Écrivez une fonction `distance`, qui prend en paramètres deux réels de type `float` `x` et `y`, et calcule puis retourne leur distance. Cette fonction doit utiliser la fonction `vabs` précédente. Cette fois, on veut que le résultat de la fonction soit passé par adresse, sous forme d'un 3ème paramètre `res`.

Exercice 9

On veut écrire une fonction `division_entiere` qui calcule et renvoie à la fois le quotient `q` et le reste `r` de la division de deux entiers `x` et `y`. Doit-on utiliser un passage de paramètre par valeur ou par adresse ? Pourquoi ? Écrivez et testez la fonction.

Exercice 10

On veut résoudre une équation du 2ème degré. Pour cela, on veut écrire une fonction `calcule_racines` qui calcule le discriminant du polynôme et l'utilise pour déterminer l'existence de solution(s), puis pour les calculer si c'est possible.

Écrivez une fonction qui reçoit par valeur les trois coefficients `a`, `b` et `c` du polynôme. La fonction doit renvoyer deux sortes de résultats :

- Elle doit renvoyer *par valeur* un code indiquant s'il existe une racine (code 1), deux racines (code 2) ou pas de racine du tout (code 0).
- Si une ou plusieurs racines existent, elle doit la (ou les) renvoyer *par adresse*.

Donc, la fonction a besoin de deux paramètres supplémentaires `r1` et `r2`, qui ne seront pas obligatoirement utilisés (ça dépend du discriminant). Tout l'affichage doit être effectué dans la fonction de l'exercice suivant, et non pas dans `calcule_racine`.

Remarque : La fonction calculant la racine carrée d'un réel `x` en langage C est `sqrt(x)`. Elle est contenue dans la librairie `math.h`, qui devra donc être inclue dans votre programme. Il est également nécessaire de modifier un paramètre de compilation. Dans les propriétés du projet Eclipse, allez dans *C/C++ Build* puis *Settings*, puis *Tool Settings*, puis *C Linker*, puis *Libraries*. À droite, dans Libraries (-l), ajoutez une librairie simplement appelée `m` (m comme mathématiques).

Exercice 11

Écrivez une fonction `void affiche_racines(float a, float b, float c)` qui reçoit les 3 coefficients d'un polynôme du 2ème degré, qui utilise `calcule_racines` pour calculer ses racines et qui affiche le résultat *exactement* comme indiqué ci-dessous.

exemple 1 : pour l'appel `affiche_racines(10, -4, 1)` on obtient l'affichage :

```
Traitement du polynome 10.00x^2 - 4.00x + 1.00 = 0
Il n'existe aucune racine relle pour ce polynome
```

exemple 2 : pour l'appel `affiche_racines(9, 12, 4)` on obtient l'affichage :

```
Traitement du polynome 9.00x^2 + 12.00x + 4.00 = 0
```

```
Il n'existe qu'une seule racine reelle pour ce polynome : r=-0.67
```

exemple 3 : pour l'appel `affiche_racines(3, -5, 2)` on obtient l'affichage :

```
Traitement du polynome 3.00x^2 + -5.00x + 2.00 = 0
Il existe deux racines reelles pour ce polynome : r1=0.67 et r2=1.00
```

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

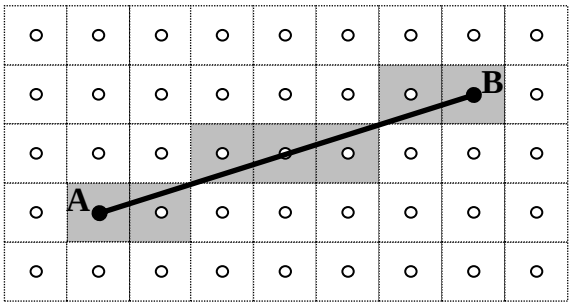
TP 12 | algorithme de Bresenham

Présentation

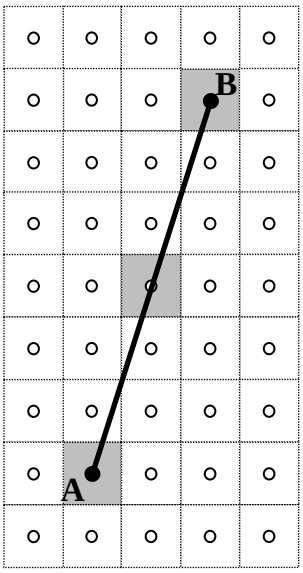
Dans ce TP, nous utiliserons la SDL, donc vous devez configurer votre projet de façon appropriée. Puis, copiez dans votre projet les fichiers de la librairie `graphisme`, contenus dans l'archive fournie avec ce sujet.

1 Rastérisation d'une droite

On veut représenter un segment dans la fenêtre graphique en utilisant la fonction `allume_pixel` de notre bibliothèque graphique. Pour représenter un segment qui joint les centres A et B de deux pixels, on allume, sur chaque colonne, le pixel dont le centre est le plus proche du segment $[AB]$. Dans la figure ci-dessous, les carrés représentent les pixels de l'écran et les ronds les centres de ces pixels. Les pixels allumés sont grisés. Le segment $[AB]$ est donc représenté par les 7 pixels grisés.



Cette méthode n'est valable que pour un segment dont le coefficient directeur est compris entre 0 et 1. En effet, si le coefficient directeur est supérieur à 1, il restera des lignes ne contenant aucun pixel allumé :



On remédiera à ce problème en parcourant les pixels en suivant les *lignes* plutôt que les *colonnes*.

2 Notations

Soient (x_A, y_A) et (x_B, y_B) les coordonnées A et B des extrémités du segment $[AB]$ à dessiner. On note $dx = x_B - x_A$ et $dy = y_B - y_A$.

Soient (x_k, y_k) avec $0 \leq k \leq dx$ les coordonnées des pixels représentant le segment $[AB]$. On a donc $x_0 = x_A$, $y_0 = y_A$ et $x_k = x_B$, $y_k = y_B$.

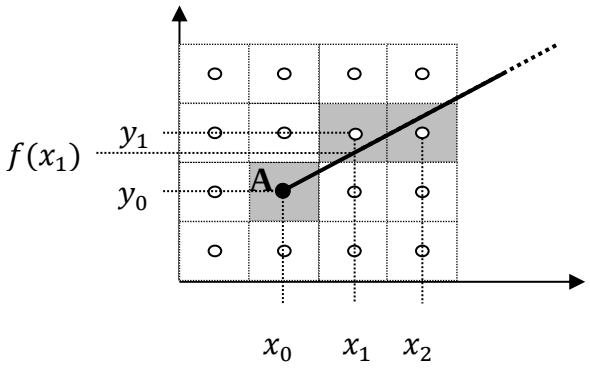
On fait les hypothèses suivantes :

- $x_A < x_B$ et $y_A < y_B$: le point A est situé en bas à gauche du point B (on raisonne dans un repère classique).
- $0 \leq dx \leq dy$: le coefficient directeur de la droite (AB) est compris entre 0 et 1.

L'équation de la droite (AB) peut s'écrire :

$$y - y_0 = \frac{dx}{dy}(x - x_0)$$

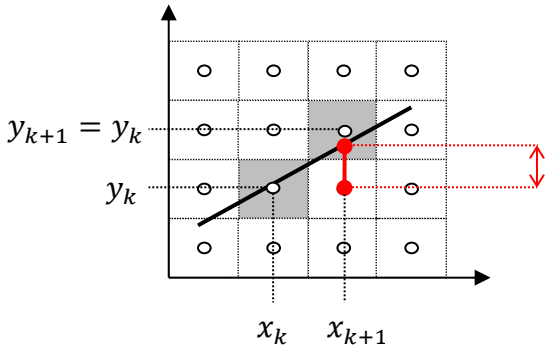
On lui associe l'application affine f définie par $f(x) = \frac{dx}{dy}(x - x_0) + y_0$.



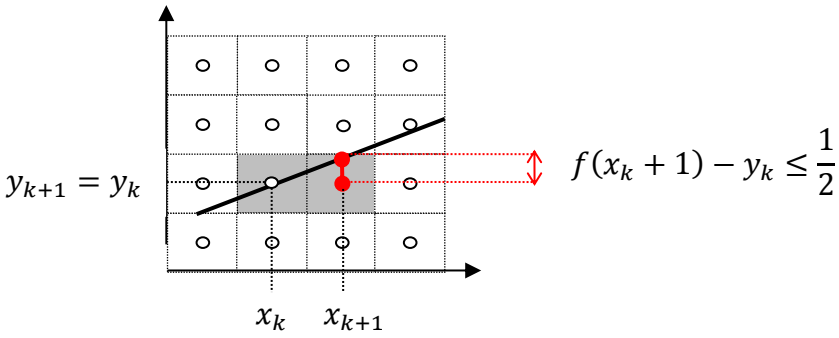
3 Algorithme naïf

On parcourt les pixels suivant les colonnes, donc $x_0 = x_A$ et pour tout k, $x_{k+1} = x_k + 1$. Pour tout k :

- Si $f(x_k + 1) - y_k \geq 0{,}5$:
  - Le pixel de la $(k+1)^{\text{ème}}$ colonne le plus proche de la droite (AB) est le pixel de coordonnées $(x_{k+1}, y_k + 1)$.
  - Donc on choisit : $y_{k+1} = y_k + 1$.



- Sinon :
  - C'est le pixel de coordonnées (x_{k+1}, y_k) qui est le plus proche.
  - Donc on choisit : $y_{k+1} = y_k$.



Exercice 1

Écrivez une fonction `trace_segment_naif(int xa, int ya, int xb, int yb)` qui trace, à l'aide de la méthode décrite ci-dessus, un segment joignant deux pixels A et B. Le pixel A doit être situé à gauche du pixel B et le coefficient directeur de la droite (AB) doit être compris entre 0 et 1.

4 Algorithme de Bresenham

Le principe de l'algorithme de Bresenham a le même résultat que celui exposé au-dessus, mais il n'effectue que des opérations d'addition et de soustraction sur des entiers, améliorant ainsi la vitesse d'exécution.

L'évaluation de l'inégalité $f(x_k + 1) - y_k \geq 0{,}5$ nécessite des calculs sur des nombres réels. Afin de les éviter, l'algorithme de Bresenham travaille plutôt avec l'entier d_k suivant :

$$d_k = 2dx(f(x_k + 1) - y_k) - dx$$

En effet, le signe de d_k permet de déterminer le choix du pixel de coordonnées (x_{k+1}, y_{k+1}) à allumer, d'après l'équivalence suivante (avec $dx > 0$) :

$$(f(x_k + 1) - y_k \geq 0{,}5) \Leftrightarrow (d_k \geq 0)$$

Donc :

- Si $d_k \geq 0$: $y_{k+1} = y_k + 1$
- Si $d_k < 0$: $y_{k+1} = y_k$

Le calcul se fait par récurrence, en utilisant la propriété d'affinité de f :

$$f(x_{k+1} + 1) = f(x_{k+1}) + \frac{dy}{dx}$$

On a les relations :

- $d_0 = 2dx(f(x_0 + 1) - y_0) - dx$
  $= 2dx\big(f(x_0 + 1) - f(x_0)\big) - dx$
  $= 2dy - dx \in \mathbb{N}$
- $d_{k+1} = 2dx(f(x_{k+1} + 1) - y_{k+1}) - dx$
  - Si $d_k < 0$:
    $d_{k+1} = 2dx(f(x_k + 2) - y_k) - dx$
    $= d_k + 2dy$
  - Si $d_k \geq 0$:
    $d_{k+1} = 2dx\big(f(x_k + 2) - (y_k + 1)\big) - dx$
    $= d_k + 2(dy - dx)$

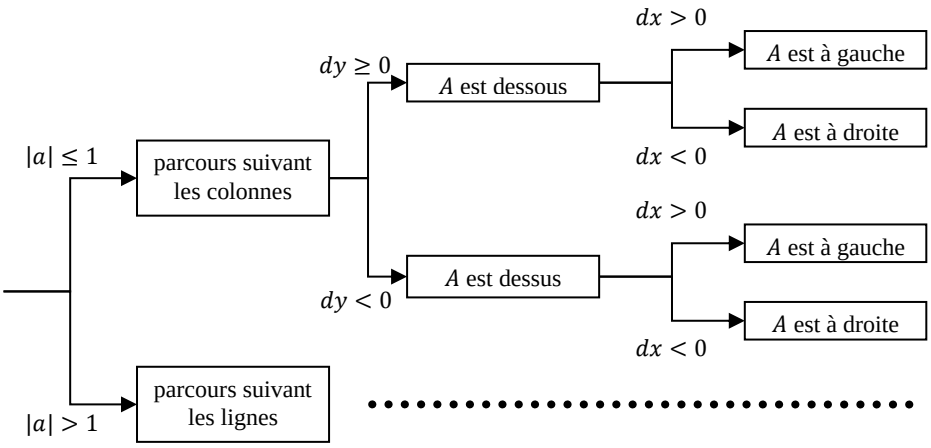
D'après la relation de récurrence ci-dessus, comme les valeurs de dx et dy sont entières, les valeurs de dk s'obtiennent en effectuant uniquement des additions et des soustractions sur des entiers.

Exercice 2

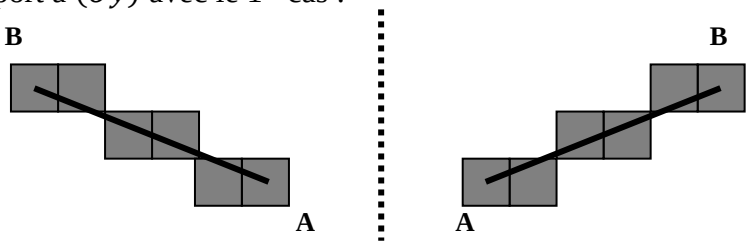
Écrivez une fonction `trace_segment_bresenham1(int xa, int ya, int xb, int yb)` qui trace un segment $[AB]$ en utilisant le principe décrit ci-dessus. On suppose que le point A est situé à gauche de B, et que le coefficient directeur est compris entre 0 et 1.

5 Généralisation

Nous n'avons étudié que le cas où le point A se situe à gauche du point B et où le coefficient directeur est compris entre 0 et 1. Mais il existe au total 8 cas possibles répertoriés dans l'arbre ci dessous :



- Le cas détaillé précédemment est le 1er cas : A est au-dessous et à gauche de B.
- Pour le 2ème cas (A est au-dessous et à droite de B), on peut remarquer une symétrie par rapport à (Oy) avec le 1er cas :



- De même, on peut retrouver les 3ème et 4ème cas par symétrie par rapport à (Ox).
- Enfin, pour effectuer un parcours suivant les lignes (au lieu des colonnes), il suffit d'échanger dans les formule les x_k avec les y_k ; ainsi que les dx avec les dy.

Exercice 3

Copiez-collez votre fonction `trace_segment_bresenham1` pour obtenir une fonction `trace_segment_bresenham2`. Modifiez-la pour qu'elle trace le segment $[AB]$ quelles que soient les positions des points A et B.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 13 | histogramme des couleurs

1 Présentation

Dans ce TP, on se propose de travailler sur la notion d'histogramme. Un histogramme est un graphique permettant de représenter comment se répartissent les valeurs dans un échantillon donné.

Dans un premier temps, on définira des fonctions permettant de définir un histogramme à partir de données quelconques. L'un des objectifs du TP est de constituer une librairie destinée à la gestion d'histogrammes. Dans un second temps, on appliquera ces fonctions à l'analyse d'une image, afin d'en représenter la répartition des couleurs.

Ce sujet est fourni avec les bibliothèques `alea` et `graphisme`. La première doit être complétée pour permettre de générer des nombres aléatoirement. La deuxième ne doit pas être modifiée, et ne sera utilisée que dans la dernière partie du TP. Une troisième bibliothèque `histogramme` devra être créée pour afficher des histogrammes à l'écran.

2 Génération de données de test

La première chose à faire avant de définir les fonctions permettant de créer des histogrammes, et d'obtenir des données pour tester ces fonctions. On s'appuiera pour cela sur la fonction `int genere_entier(int limite)`, disponible dans la bibliothèque `alea`, qui renvoie un entier tiré aléatoirement et compris entre 0 et la valeur `limite`.

Exercice 1

Créez un fichier `main.c` , puis une fonction `main` à l'intérieur de ce fichier. Incluez la librairie `alea`. Dans la fonction `main`, utilisez la fonction `genere_entier` pour obtenir un entier, et affichez-le à l'écran.

Exercice 2

Dans la bibliothèque `alea`, écrivez une fonction d'en tête `void genere_tableau(int tab[], int taille, int limite)` qui initialise aléatoirement le tableau `tab`, en utilisant `taille` entiers générés aléatoirement avec la fonction `genere_entier`.

Exercice 3

Sur le modèle des bibliothèques fournies avec ce sujet, créez et initialisez les deux fichiers nécessaires à la bibliothèque `histogramme` (i.e. `histogramme.h` et `histogramme.c`) Dans cette bibliothèque, créez la constante (macro) `VAL_MAX`, qui doit prendre la valeur 255. Cette constante correspond à la valeur maximale que nous manipulerons dans ce TP.

Toujours dans la bibliothèque `histogramme`, écrivez une fonction `void affiche_tableau(int tab[], int taille)` capable d'afficher le tableau `tab` passé en paramètre, sous forme textuelle. Depuis la fonction `main`, testez vos fonctions en générant aléatoirement un tableau avec `genere_tableau`, puis en l'affichant avec `affiche_tableau`. Vous devez aussi inclure les bibliothèques nécessaires.

exemple : pour le tableau `{4,2,2,10,7,4,3,4,10,7,11,4,0,4,2,5,0,6,8,9,10}` vous devez obtenir *exactement* l'affichage suivant :

```
tab[0]=4
tab[1]=2
tab[2]=2
tab[3]=10
tab[4]=7
tab[5]=4
tab[6]=3
tab[7]=4
tab[8]=10
tab[9]=7
tab[10]=11
tab[11]=4
```

```
tab[12]=0
tab[13]=4
tab[14]=2
tab[15]=5
tab[16]=0
tab[17]=6
tab[18]=8
tab[19]=9
tab[20]=10
...
```

3 Définition de l'histogramme

Exercice 4

Pour afficher l'histogramme, on a d'abord besoin de compter le nombre d'occurrences de chaque valeur contenue dans le tableau : nombre de 0, nombre de 1, nombre de 2, etc. Dans `histogramme`, écrivez une fonction `void decompte_valeurs(int tab[], int taille, int decompte[])`. La fonction reçoit un tableau `tab` contenant `taille` entiers compris entre 0 et `VAL_MAX`. Elle doit compléter le tableau `decompte`, dont la taille est `VAL_MAX+1`, de manière à ce que la valeur `decompte[i]` corresponde au nombre d'occurrences de la valeur `i` dans `tab`.

exemple : pour le tableau précédent, on obtiendrait le décompte `{2,0,3,1,5,1,1,2,1,1,3,1...}`. En d'autres termes : la valeur 0 apparait 2 fois dans `tab`, la valeur 1 n'apparait jamais, la valeur 2 apparait 3 fois, etc.

Testez votre fonction dans la fonction `main` sur le tableau `tab` précédemment créé, et affichez (grâce à `affiche_tableau`) le tableau `decompte` que vous obtenez.

Remarque : on ne connait pas, *a priori*, les valeurs contenues dans le tableau `decompte` reçu par la fonction. Vous devez donc penser à l'initialiser.

Exercice 5

Dans la bibliothèque `histogramme`, écrivez une fonction `void histogramme_horizontal(int decompte[], int taille_d)` qui affiche l'histogramme *horizontal* du tableau de décompte de taille `taille_d` passé en paramètre. Testez votre fonction depuis `main`.

exemple : pour le tableau de décompte précédent, on obtient l'affichage :

```
**

***
*
*****
*
*
**
*
*
***
*
...
```

Exercice 6

Dans la bibliothèque `histogramme`, écrivez une fonction `void histogramme_vertical(int decompte[], int taille_d)` qui affiche cette fois

l'histogramme *vertical* du tableau de décompte passé en paramètre. Testez votre fonction depuis `main`.

exemple : pour le tableau de décompte précédent, on obtient l'affichage :

```
          *
          *
      *   *           *
  *   *   *     *     *
  *   * * * * * * * * *...
```

4 Regroupement des valeurs

Exercice 7

On peut remarquer qu'en raison du nombre de valeurs différentes (de 0 à 255), les histogrammes précédents sont difficilement lisibles. Il est possible d'améliorer cet aspect grâce à la notion de classe. Dans le contexte des histogrammes, une *classe* est un groupement de plusieurs valeurs. Par exemple, au lieu de représenter le nombre de valeurs 0, 1, 2 par 3 lignes (ou colonnes) d'étoiles, on peut toutes les regrouper, et ainsi utiliser une seule ligne (ou colonne) pour représenter les valeurs de la classe $[0; 3]$.

exemple : pour le décompte précédent, si on fait des classes de 4 valeurs, on a :

```
******
*********
******
...
```

Dans `histogramme`, écrivez la fonction `void regroupe_classes(int decompte[], int nbr_classes, int classes[])` qui reçoit en paramètre un tableau de décompte de taille `VAL_MAX+1`, et qui regroupe ses valeurs sur `nbr_classes`. Le tableau `classes` doit être un tableau de taille `nbr_classes`, utilisé pour stocker les valeurs calculées.

Remarque : on suppose que `nbr_classes` est un diviseur de `VAL_MAX+1`.

Exemple : dans le cas de l'exemple précédent, on obtient pour `classes` le tableau `{6,9,6,...}`.

Exercice 8

Puisque nous avons dans `tab` 256 valeurs possibles (allant de 0 à `VAL_MAX`), on se propose de découper notre domaine en 32 classes, chacune contenant donc 8 valeurs : $[0; 7]$, $[8; 15]$, $[16; 23]$, etc.

Dans la fonction `main`, générez avec `genere_tableau` un tableau `tab` de taille 100000. Calculez son décompte avec `decompte_valeurs`, puis utilisez `regroupe_classes` pour regrouper les valeurs obtenues sur 32 classes. Affichez l'histogramme pour ces classes, avec `histogramme_horizontal`.

Exercice 9

La lisibilité a été améliorée, mais dans le cas précédent on voit que s'il y a un grand nombre de valeurs, le nombre important d'étoiles mises sur la même ligne (ou colonne) empêche de lire convenablement l'histogramme. On va donc regrouper les étoiles : au lieu de représenter une seule occurrence, une étoile va maintenant en représenter plusieurs.

Dans `histogramme`, écrivez une fonction `void regroupe_etoiles(int classes[], int nbr_classes, int etoiles[], int echelle)`, qui réalise ce regroupement des étoiles. Le tableau `classes` est celui obtenu avec `regroupe_classes`, donc il s'agit d'un tableau de décompte dont les valeurs ont été regroupées par classe, et sa

longueur est `nbr_classes`. Le tableau `etoiles` est à compléter par votre fonction, de manière à ce que chaque étoile de `etoiles` corresponde à `echelle` étoiles dans `classes`.

exemple : dans le cas de l'exemple précédent, si on applique `regroupe_etoiles` avec une échelle de 3, alors on obtient pour `etoiles` le tableau `{2,3,2,...}`.

5 Couleurs d'une image

On veut maintenant utiliser nos fonctions pour analyser une image. Comme on l'a vu dans les TP précédents, chaque pixel d'une image est caractérisé par ses trois composantes RVB. Nous allons représenter la distribution d'une composante en particulier, sous forme d'histogramme.

Exercice 10

Dans `main.c`, écrivez une fonction `void extrais_composantes(SDL_Surface *surface, int tab_r[], int tab_v[], int tab_b[])` qui reçoit une image sous forme de surface, et qui complète les trois tableaux passés en paramètres. Le nombre d'élément de chaque tableau correspond au nombre de pixels dans l'image. Chaque élément du tableau représente la composante d'un pixel de l'image : rouge pour `tab_r`, vert pour `tab_v` et bleu pour `tab_b`.

Exercice 11

Dans `main.c`, écrivez une fonction `void analyse_surface(SDL_Surface *surface)` qui reçoit une image sous forme de surface, et qui affiche 3 histogrammes verticaux, chacun correspondant à la distribution d'une composante. Pour cela, la fonction doit :

1) Extraire les composantes avec `extrais_composantes` ;
2) Décompter les valeurs avec `decompte_valeurs` ;
3) Regrouper les valeurs en 32 classes avec `regroupe_classes` ;
4) Regrouper les étoiles avec `regroupe_etoiles`, en utilisant une échelle de 100 ;
5) Afficher les trois histogrammes avec `histogramme_vertical`.

Dans la fonction main, chargez une image sous forme de surface, et appliquez `analyse_surface` pour tester votre fonction. La figure ci-dessous donne une idée du résultat obtenu pour la composante rouge, avec l'image `image1.bmp`.

```
L'histogramme pour le rouge est :
------------------------------------------------------------------
|                                                    *           |
|                                                    * *         |
|                                                  * * *         |
|                                                * * * *         |
|                                                * * * *         |
|                                              * * * * * *       |
|                                    *         * * * * * *       |
|      *             * * *   * * * * * *     * * * * * * * *    |
------------------------------------------------------------------
```

L'interprétation de cet histogramme est que la photo comporte beaucoup de rouge : en effet, la distribution montre un pic pour des valeurs élevées de la composante R. Autrement dit, il est fréquent qu'un pixel possède une forte composante de rouge.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 14 | propriétés arithmétiques

Présentation

Le but de ce TP est d'approfondir les tableaux et fonctions, en manipulant des notions issues de l'arithmétique. Nous allons en particulier travailler avec les diviseurs d'un nombre entier, que nous représenterons sous forme de tableau. Pour en simplifier le traitement, nous nous limiterons à des entiers possédant `N` diviseurs (où `N` est une constante).

1 Diviseurs d'un nombre

Exercice 1

Définissez la constante `N` sous forme de macro. Écrivez une fonction `void calcule_diviseurs(int n, int diviseurs[N], int *d)` qui identifie tous les diviseurs de l'entier `n` (y compris `n` lui-même) et les range dans le tableau diviseurs, dans l'*ordre croissant*. La fonction renvoie par adresse le nombre de divisieurs identifiés, grâce au paramètre `d`. On supposera que $n \geq 2$. Testez votre fonction depuis la fonction `main`, en affichant le tableau obtenu.

exemple : pour `n=6`, on obtient le tableau `{1,2,3,6}` et `d=4`.

Exercice 2

Pour tout entier naturel $n \geq 2$, on note $s(n)$ la somme des diviseurs *propres* de n, i.e. la somme de tous ses diviseurs sauf n lui-même.

exemples :

- $s(6) = 1 + 2 + 3 = 6$;
- $s(10) = 1 + 2 + 5 = 8$;
- $s(12) = 1 + 2 + 3 + 4 + 6 = 16$;
- $s(p) = 1$, pour tout nombre premier p.

Écrivez une fonction `int additionne_diviseurs_propres(int n)` qui calcule la somme des diviseurs propres de l'entier `n`.

Exercice 3

On notera $\sigma(n)$ la somme des diviseurs de n, i.e. la somme de tous ses diviseurs (y compris n lui-même).

exemples :

- $\sigma(6) = 1 + 2 + 3 + 6 = 12$;
- $\sigma(10) = 1 + 2 + 5 + 10 = 18$;
- $\sigma(12) = 1 + 2 + 3 + 4 + 6 + 12 = 28$;
- $\sigma(p) = 1 + p$, pour tout nombre premier p.

Écrivez une fonction `int additionne_diviseurs(int n)` qui calcule la somme des diviseurs de l'entier `n`. Vous devez utiliser la fonction `additionne_diviseurs_propres`.

Exercice 4

Écrivez une fonction `void affiche_sommes_diviseurs_propres(int max)` qui affiche :

- Sur une première ligne : la liste des entiers `n` compris entre 2 et `max` ;
- Sur la deuxième ligne : les valeurs de $s(n)$ correspondantes.

On utilisera cette fonction avec un paramètre `max` inférieur ou égal à 59, de sorte que les nombres $s(n)$ aient au plus 2 chiffres. **Attention :** vous ferez en sorte que les nombres n et $s(n)$ soient alignés.

exemple : l'appel `affiche_diviseurs_propres(25)` produit l'affichage suivant :

```
n    : 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
s(n) : 01 01 03 01 06 01 07 04 08 01 16 01 10 09 15 01 21 01 22 11 14 01 36 06 16
```

2 Nombres parfaits, amicaux et sublimes

Exercice 5

On appelle *abondance* d'un nombre $n \geq 2$ la valeur $a(n) = s(n) - n$.

exemples :

- L'abondance de 6 est : $a(6) = s(6) - 6 = 6 - 6 = 0$;
- L'abondance de 10 est : $a(10) = s(10) - 10 = 8 - 10 = -2$;
- L'abondance de 12 est : $a(12) = s(12) - 12 = 16 - 12 = 4$.

Écrivez une fonction `int calcule_abondance(int n)` qui calcule l'abondance du nombre passé en paramètre. Vous devez utiliser `additionne_diviseurs_propres`.

Exercice 6

Un entier $n \geq 2$ est dit *parfait* s'il est égal à la somme de ses diviseurs propres $s(n)$, i.e. : $n = s(n)$. Autrement dit, un entier est parfait si son abondance est nulle, i.e. : $a(n) = 0$.

exemples :

- 6 est parfait, car : $s(6) = 6$;
- 10 n'est *pas* parfait, car : $s(10) \neq 10$;
- 28 est parfait, car : $s(28) = 1 + 2 + 4 + 7 + 14 = 28$.

Écrivez une fonction `int est_parfait(int n)` qui renvoie 1 si l'entier `n` est parfait, et 0 sinon.

Exercice 7

Deux entiers $n, p \geq 2$ sont dits *amicaux* lorsque la somme des diviseurs propres de chacun des deux est égal à l'autre nombre, c'est-à-dire s'ils vérifient *à la fois* $s(n) = p$ *et* $s(p) = n$.

exemple : les nombres 220 et 284 sont amicaux, car on a :

- $s(220) = 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284$
- $s(284) = 1 + 2 + 4 + 71 + 142 = 220$

Écrivez une fonction `int sont_amicaux(int n, int p)` qui retourne 1 si les entiers `n` et `p` sont amicaux, et 0 sinon.

Exercice 8

Un entier $n \geq 2$ est dit *sublime* lorsque le nombre de ses diviseurs et la somme de ses diviseurs sont tous les deux des nombres parfaits.

exemple : 12 est un nombre sublime, car $\sigma(12) = 1 + 2 + 3 + 4 + 6 + 12$, et on a donc à la fois :

- 6 diviseurs, et 6 est un nombre parfait ;
- $\sigma(12) = 28$, et 28 est aussi un nombre parfait.

Écrivez une fonction `int est_sublime(int n)` qui détermine si le nombre `n` est sublime. La fonction renvoie 1 si c'est le cas, et 0 sinon.

Remarque : on n'a identifié à ce jour que deux nombres sublimes.

3 Nombres abondants et déficients

Exercice 9

On appelle nombre *abondant* un nombre $n \geq 2$ dont l'abondance est *strictement positive* : $a(n) > 0$. Au contraire, on appelle nombre *déficient* un nombre dont l'abondance est *strictement négative* : $a(n) < 0$.

exemples :

- 10 est un nombre déficient, car $a(10) = -2$;
- 12 est un nombre abondant, car $a(12) = 14$.

Écrivez les fonctions `int est_abondant(int n)` et `int est_deficient(int n)` qui indiquent si le nombre `n` passé en paramètre est respectivement abondant ou déficient. Chaque fonction doit renvoyer 1 si c'est le cas (nombre abondant ou déficient) et 0 si ce n'est pas le cas.

Exercice 10

Écrivez une fonction `void affiche_nombres(int max, int mode)` qui reçoit en paramètres une valeur maximale `max` et un mode de fonctionnement `mode`. Ce mode a trois valeurs possibles : -1, 0 et $+1$. En fonction du mode, la fonction doit réaliser l'une des trois tâches suivantes :

- Pour -1 : afficher tous les entiers déficients compris entre 2 et `max` ;
- Pour 0 : afficher tous les entiers parfaits compris entre 2 et `max` ;
- Pour $+1$: afficher tous les entiers abondants compris entre 2 et `max`.

exemple : l'appel `affiche_nombres(100,1)` affiche la liste des nombres abondants inférieurs à 100 :

```
 1. 12
 2. 18
 3. 20
 4. 24
 5. 30
 6. 36
 7. 40
 8. 42
 9. 48
10. 54
11. 56
12. 60
13. 66
14. 70
15. 72
16. 78
17. 80
18. 84
19. 88
20. 90
21. 96
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 15 | algorithmes pour l'arithmétique

Présentation

Le but de ce TP est de mettre en œuvre la notion de preuve de programme vue en cours, en l'appliquant à des fonctions simples relatives à l'arithmétique.

1 Division euclidienne

On veut calculer, pour tous entiers positifs a et b avec $b \neq 0$, le quotient q et le reste r de la division Euclidienne de a par b. C'est-à-dire que l'on veut obtenir les entiers r et q tels que : $a = bq + r$, avec $0 \leq b < r$.

Voici la description d'un algorithme qui effectue ce calcul :

- **Initialisation** :
  - $q \leftarrow 0$
  - $r \leftarrow a$
- **Itération** : Tant que $r \geq b$,
  - $q \leftarrow q + 1$
  - $r \leftarrow r - b$

Exercice 1

Appliquez cet algorithme aux entiers $a = 57$ et $b = 11$, décrivez toute les étapes de l'algorithme.

Exercice 2

Écrivez une fonction `division_euclidienne` qui reçoit en paramètres deux entiers a et b, qui calcule le reste r et le quotient q de la division de a par b, et qui renvoie r et q. Cette fonction utilisera l'algorithme décrit ci-dessus.

Exercice 3

Quel est l'invariant de boucle de cet algorithme ? *Justifiez* votre réponse.

2 Plus grand commun diviseur

On note $\text{pgcd}(a, b)$ le *plus grand diviseur commun* de deux entiers a et b. On admet la propriété suivante (qui correspond au théorème de Bachet-Bézout) :

Pour tous entiers positifs a et b, il existe deux entiers u et v tels que :

$$au + bv = \text{pgcd}(a, b)$$

Soient a et b deux entiers, l'algorithme ci-dessous permet de calculer un couple (u, v) qui vérifie l'équation $au + bv = \text{pgcd}(a, b)$:

- **Initialisation :**
  - $(d, u, v) \leftarrow (a, 1, 0)$
  - $(d', u', v') \leftarrow (b, 0, 1)$
- **Itération :** Tant que $d' \neq 0$,

- On note q et r le quotient et le reste de la division euclidienne de d par d'
- $(d', u', v') \leftarrow (r, u - qu', v - qv')$
- $(d, u, v) \leftarrow (d', u', v')$

- **Terminaison :** on a $d = \text{pgcd}(a, b)$

Exercice 4

Appliquez cet algorithme au entiers $a = 57$ et $b = 11$.

Exercice 5

Écrivez une fonction `pgcd` qui reçoit en paramètre deux entiers a et b, qui calcule les entiers u, v et d tels que $d = au + bv = \text{pgcd}(a, b)$, et qui renvoie ces trois valeurs.

Exercice 6

Démontrez que cet algorithme se termine.

Exercice 7

Démontrez que la propriété suivante est un invariant de boucle de l'algorithme :

$$\begin{cases} au + bv = d \\ au' + bv' = d' \\ \text{pgcd}(d, d') = \text{pgcd}(a, b) \end{cases}$$

On pourra noter $({d'}_k, {u'}_k, {v'}_k)$ et (d_k, u_k, v_k) les valeurs respectives de (d', u', v') et (d, u, v) après k itérations.

Exercice 8

Démontrez que la valeur obtenue pour le pgcd à la fin de l'algorithme est correcte.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 16 | bibliothèque `chaines`

Présentation

Dans ce TP, on veut écrire notre propre bibliothèque `chaine`, qui regroupera des fonctions permettant de manipuler des chaînes de caractères. Une bibliothèque est un ensemble de fonctions proches, dans le sens où elles permettent généralement de manipuler les mêmes données. Par exemple, `stdio` se charge des entrées-sorties.

En langage C, une bibliothèque `xxxx` se compose de deux fichiers : le fichier `xxxx.h` contenant les en-têtes des fonctions, les types et les constantes ; et le fichier `xxxx.c` qui contient le corps des fonctions.

Dans les TP où vous devrez définir une bibliothèque, soyez bien attentif à l'emplacement des fonctions, qui vous sera indiqué dans le sujet : soit dans le fichier `main.c`, soit dans la bibliothèque. **Attention :** dans ce dernier cas, cela signifie que les deux fichiers `xxxx.c` et `xxxx.h` doivent être mis à jour. Si l'emplacement n'est pas précisé, alors il faut écrire la fonction dans `main.c`. Dans les deux cas, n'oubliez pas que chaque fonction écrite doit être testée à partir de la fonction `main`.

1 Préparation des fichiers

Exercice 1

Dans votre projet Eclipse, créez les fichiers suivants :

- `main.c` : fichier principal qui contiendra la fonction principale `main`.
- `chaine.h` : fichier d'en-tête (ou *header*) de la bibliothèque.
- `chaine.c` : fichier de corps de la bibliothèque.

Exercice 2

Dans le fichier d'en-tête, copiez-collez le code source suivant, qui permet d'empêcher la bibliothèque d'être chargée plusieurs fois lors de la compilation :

```
#ifndef CHAINE_H_
#define CHAINE_H_

// TODO a completer ici

#endif /* CHAINE_H_ */
```

Dans la suite, ce fichier devra contenir ;

- Les inclusions des bibliothèques utilisées par les fonctions contenue dans notre propre librairie, comme par exemple `#include <stdio.h>` ;
- Les déclarations de constantes (`#define …`), et plus tard de types de données ;
- Les en-têtes des fonctions contenues dans notre bibliothèque.

Le code source correspondant à ces 3 sortes d'éléments sera à rajouter, dans l'ordre indiqué, à la place du commentaire "`// TODO a completer ici`" représenté en rouge dans le code source que vous avez copié-collé.

Exercice 3

Dans le fichier `chaine.c`, rajoutez la ligne suivante :

```
#include "chaine.h"
```

Cette instruction indique que le fichier `chaine.c` est lié au fichier `chaine.h`, et en particulier qu'il utilise les bibliothèques qui y sont incluses, et les constantes et types qui y sont déclarés.

Donc, faites la même chose dans le fichier main.c : cela permettra à votre fonction main d'utiliser les bibliothèques, fonctions, types et constantes utilisées ou définies dans la bibliothèque `chaine`.

Remarque : vous noterez qu'à la différence des utilisations précédentes de `#include`, comme par exemple quand on fait `#include <stdio.h>`, ici le nom de la bibliothèque est entouré de *guillemets* (`"xxx"`) et non pas de *crochets* (`<xxx>`) : cela est dû au fait qu'il s'agit d'une bibliothèque définie *localement* (i.e. dont le code source se trouve dans le projet), et non pas d'une bibliothèque standard localisée au même endroit que le compilateur.

2 Fonctions existantes

Les fichiers sont prêts, et on veut maintenant commencer à rajouter des fonctions à notre bibliothèque. Dans le reste du sujet, pour chaque fonction demandée, vous devez d'abord écrire son en-tête dans `chaine.h`, puis la fonction complète (corps *et* en-tête) dans `chaine.c`. De plus, vous devez tester chaque fonction à partir de la fonction main, comme d'habitude.

Remarque : dans chaine.h, chaque en-têtes doit être suivie d'un *point-virgule* `';'`.

Comme expliqué au début du sujet, une bibliothèque se compose de fonctions traitant toutes du même thème. Ici, il s'agit de fonctions portant sur les chaînes de caractères. Or, nous avons déjà fait un TP sur les chaînes de caractères : nous allons donc commencer par intégrer ce travail précédent à notre bibliothèque. Les fonctions de cette section correspondent à des exercices de ce TP, veuillez consulter sa correction pour les obtenir, et le sujet lui-même si vous voulez plus de détails sur le comportement des fonctions.

Remarque : attention aux en-têtes du présent TP, qui sont légèrement différentes de celles du TP passé.

Exercice 4

Écrivez une fonction `int mesure_chaine(char* chaine)` qui reçoit une chaîne de caractères `chaine` et calcule sa longueur, sans compter le caractère de fin de chaîne `'\0'`.

Exercice 5

Écrivez une fonction `void copie_chaine(char* chaine1, char* chaine2)` qui recopie la chaine de `chaine1` dans `chaine2`. On supposera que `chaine2` correspond à un tableau dont la taille est suffisante pour y recopier `chaine1`.

Exercice 6

Écrivez une fonction `int compare_chaines(char* chaine1, char* chaine2)` qui reçoit deux chaînes et les compare en utilisant l'ordre lexicographique de la table ASCII. La fonction doit renvoyer -1 si la `chaine1` est située avant `chaine2` dans l'ordre lexicographique, 0 si `chaine1` et `chaine2` sont exactement les mêmes, et $+1$ si `chaine1` est situé après `chaine2` dans l'ordre lexicographique.

Exercice 7

Écrivez une fonction `void inverse_chaine(char* chaine)` qui inverse la chaîne de caractère chaine passée en paramètre. Le traitement doit être réalisé *sur place*. Autrement dit : vous devez travailler directement dans `chaine`.

3 Nouvelles fonctions

Exercice 8

Écrivez une fonction `void supprime_majuscules(char* chaine)` qui prend en argument une chaine de caractères `chaine` et qui supprime toutes les majuscules qu'elles contient. Les caractères qui ne sont pas des lettres ne sont pas modifiés. On suppose que la chaine originale ne contient pas d'accents.

exemple : pour `chaine="afegAEfd CDghj!"`, après avoir appelé la fonction, on obtient la chaine `"afegfd ghj!"`.

Remarque : votre fonction ne doit pas utiliser de tableau supplémentaire lors du traitement. Autrement dit, aucun tampon (buffer) n'est autorisé, vous devez travailler directement sur la chaine).

Exercice 9

Écrivez une fonction `void remplace_majuscules(char* chaine)` similaire à `supprime_majuscules`, avec la différence qu'au lieu de supprimer les majuscules, elle les *remplace* par des minuscules.

exemple : pour `chaine="afegAEfd CDghj!"`, après avoir appelé la fonction, on obtient la chaine `"afegaefd cdghj!"`.

Exercice 10

Écrivez une fonction `int compte_espaces(char *chaine)` qui calcule le nombre d'espaces `' '` contenus dans la chaîne de caractères passée en paramètre.

exemple : pour `chaine="un deux trois"`, la fonction doit retourner 2.

Exercice 11

Écrivez une fonction `int compte_mots(char *chaine)` qui calcule le nombre de mots contenus dans la chaîne de caractères `chaine`. On supposera que les mots peuvent être séparés par un ou plusieurs espaces, et qu'il peut y avoir des espaces au début et à la fin de la chaîne.

exemples :

- `compte_mots("un deux trois")` retournera la valeur 3 ;
- `compte_mots("un  deux  trois")` retournera aussi la valeur 3.
- `compte_mots("  un deux  trois")` retournera aussi la valeur 3.
- `compte_mots("  un deux  trois  ")` retournera aussi la valeur 3.

Exercice 12

On dit qu'une chaîne de caractères `chaine1` est un *préfixe* d'une autre chaîne `chaine2` ssi la chaîne `chaine1` correspond exactement au début de la chaîne `chaine2`.

exemples :

- `"bon"` est un préfixe de `"bonjour"`.
- `"bonne"` n'est pas un préfixe de `"bonjour"`.
- `"bonne"` n'est pas un préfixe de `"bon"`.
- `""` (chaîne vide) est préfixe de n'importe qu'elle chaîne.

Écrivez une fonction `int est_prefixe(char* chaine1, char* chaine2)` qui renvoie 1 si `chaine2` est un préfixe de `chaine1`, et 0 sinon.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 17 | décomposition d'une permutation

Présentation

Le but de ce TP est d'abord de manipuler des concepts déjà étudiées : tableaux, pointeurs, passage de paramètres, bibliothèque ; mais aussi de faire un lien avec la notion de permutation étudiée en cours de mathématiques.

1 Notion de permutation

Soit N un entier fixé. Une permutation de l'ensemble $\{0,1,2,\dots,N-1\}$ est une bijection de l'ensemble $\{0,1,2,\dots,N-1\}$ dans lui-même. On peut représenter une permutation σ sur deux lignes sous la forme :

$$\begin{pmatrix} 0 & 1 & 2 & \dots & N-1 \\ \sigma(0) & \sigma(1) & \sigma(2) & \dots & \sigma(N-1) \end{pmatrix}$$

L'ensemble de ces permutations muni de la loi de composition des applications forme un groupe (non commutatif si $N > 2$) à $N!$ éléments.

Dans ce TP, on implémente une permutation σ sous forme d'un tableau `perm[]` où pour tout $i \in \{0,1,2,\dots,N-1\}$, `perm[i]` est l'image $\sigma(i)$ de i par la permutation `perm`.

Exercice 1

Les fonctions écrites pour ce fichier (à l'exception de la fonction main, bien entendu) vont former une bibliothèque appelée `permutation`. Créez d'abord un fichier `permutation.c`, et écrivez les fonctions suivantes dans ce fichier :

- `void saisis_permutation(int perm[N])` : demande à l'utilisateur de saisir au clavier une permutation de longueur `N`.
- `void affiche_permutation(int perm[N])` : affiche la permutation passée en paramètre, en utilisant la notation à 2 lignes expliquée au début du sujet.
- `int verifie_permutation(int perm[N])` : vérifie que le tableau passé en paramètre représente bien une permutation. Utilisez les propriétés mathématiques d'une permutation, expliquées au début du sujet.

Exercice 2

Créez maintenant un fichier d'en-tête (*header file*) `permutation.h`, et copiez-y le code source suivant :

```
#ifndef PERMUTATION_H_
#define PERMUTATION_H_

// a completer ici

#endif /*PERMUTATION_H_*/
```

Ce code source permet d'empêcher la bibliothèque d'être chargée plusieurs fois lors de la compilation.

Complétez ce fichier `permutation.h` en écrivant, à la place du commentaire `"// a completer ici"` :

1. La définition de la constante `N` ;

2. Les en-têtes des 3 fonctions de l'exercice précédent.

Complétez le fichier `permutation.c`, en lui ajoutant la ligne suivante au tout début :

```
#include "permutation.h"
```

Cette ligne permet de relier `permutation.c` à `permutation.h`, et rend possible l'utilisation de la constante `N` dans `permutation.c` (alors qu'elle est elle-même définie dans `permutation.h`).

Exercice 3

Créez un fichier `main.c` dans votre projet, et indiquez que vous y incluez `permutation`, en utilisant `#include` exactement comme dans l'exercice précédent. Ainsi, le fichier `main.c` pourra utiliser les fonctions définies dans la librairie `permutation`.

Dans le fichier `main.c`, testez vos fonctions en écrivant un programme qui :

1. Demande à l'utilisateur de saisir au clavier une permutation.
2. Teste que les valeurs saisies définissent bien une permutation.
   - Si les valeurs entrées définissent une permutation, le programme affiche cette permutation.
   - Sinon, le programme indique à l'utilisateur qu'il s'est trompé, et lui demande de recommencer la saisie.

Le programme ne s'arrête que quand l'utilisateur saisit une permutation valide.

exemple : pour `N=4`

```
Saisissez l'image de 0 : 4
Saisissez l'image de 1 : 3
Saisissez l'image de 2 : 2
Saisissez l'image de 3 : 2
Saisissez l'image de 4 : 0
Les valeurs saisies ne définissent pas une permutation.
Saisissez l'image de 0 : 4
Saisissez l'image de 1 : 3
Saisissez l'image de 2 : 1
Saisissez l'image de 3 : 2
Saisissez l'image de 4 : 0
La permutation saisie est :
0 1 2 3 4
4 3 1 2 0
```

2 Composition et inversion

Exercice 4

Dans la bibliothèque `permutation`, écrivez la fonction `void compose_permutation(int perm1[N], int perm2[N], int resultat[N])` qui prend en argument deux permutations `perm1` et `perm2`, calcule la composition de `perm1` par `perm2`, et place le résultat dans le tableau `resultat`.

exemple : pour `perm1={2,4,1,3,0}` et `perm2={1,4,3,2,0}`, alors la composition obtenue $\sigma_2 \circ \sigma_1$ est `resultat={3,0,4,2,1}`.

Exercice 5

Toujours dans la bibliothèque `permutation`, écrivez la fonction `void inverse_permutation(int perm[N], int resultat[N])`, qui prend en argument une permutation `perm`, qui calcule sa permutation réciproque, et qui la place dans le tableau `resultat`. Utilisez pour cela la propriété suivante :

$$\forall i \in \{0,1,\dots,N-1\}, \sigma^{-1}\big(\sigma(i)\big) = i$$

exemple : pour `perm1={1,4,3,2,0}`, on obtient `resultat={4,0,3,2,1}`.

Exercice 6

Dans `permutation`, écrivez la fonction `int est_identite(int perm[N])` qui renvoie 1 si la permutation passée en paramètre est la permutation identité, et 0 sinon.

exemples :

- Pour `perm={0,1,2,3,4}`, la fonction renvoie 1 ;
- Pour `perm={0,2,1,3,4}`, la fonction renvoie 0.

Exercice 7

En utilisant la bibliothèque `permutation`, complétez la fonction `main` de manière à effectuer les opérations suivantes :

- Calculez la permutation réciproque de la permutation saisie par l'utilisateur ;
- Calculez la permutation obtenue en composant la permutation saisie avec sa réciproque ;
- Vérifiez que la permutation obtenue est bien l'identité.

exemple : (suite de l'exemple de l'exercice 3)

```
La permutation reciproque est :
0 1 2 3 4
4 2 3 1 0
La composee de l'originale par la reciproque est :
0 1 2 3 4
0 1 2 3 4
La composée est bien l'identite.
```

3 Notion de cycle

Une permutation σ est appelée *cycle* de longueur p (ou p-cycle), s'il existe des entiers $i_0, i_1, \dots, i_{p-1}$ distincts dans $\{0,1, \dots, N-1\}$ tels que :

- $\sigma(i_0) = i_1$, $\sigma(i_1) = i_2$,…, $\sigma(i_{p-2}) = i_{p-1}$, et $\sigma(i_{p-1}) = i_0$;
- Et : $\forall i \notin \{i_0, i_1, \dots, i_{p-1}\}, \sigma(i) = i$.

L'ensemble $\{i_0, i_1, \dots, i_{p-1}\}$ s'appelle le *support* du cycle σ. On utilise une notation particulière pour les cycles, le cycle σ est ainsi noté sous la forme : $\sigma = (i_0, i_1, \dots, i_{p-1})$.

Exercice 8

Dans `permutation`, écrivez la fonction `void affiche_cycle(int perm[])` qui prend en argument une permutation cyclique de longueur inconnue et l'affiche en utilisant la notation propre aux cycles. Inutile de tester si la permutation passée en argument est bien un cycle (on suppose que c'est bien le cas).

exemple : pour `cycle={0,4,2,7,3,5,6,1}`, la fonction doit afficher :

```
( 1 4 3 7 )
```

4 Décomposition en cycles

On veut vérifier le théorème suivant :

> *Toute permutation se décompose en produit de cycles à supports disjoints.*

en implémentant un algorithme qui effectue cette décomposition (une preuve de la correction de cet algorithme est une preuve en soi du théorème). On se propose d'utiliser l'algorithme suivant, défini pour une permutation σ de l'ensemble $\{0,1,2 \dots, N-1\}$:

- Si pour tout i, $\sigma(i) = i$, alors σ est l'identité, l'algorithme s'arrête.
- Sinon, il existe i_0 tel que $\sigma(i_0) \neq i_0$ et on affiche le p-cycle $c = (i_0, \sigma(i_0), \dots, \sigma^{p-1}(i_0))$ où p est le plus petit entier k tel que $\sigma^k(i_0) = i_0$.

- On répète ce traitement en remplaçant σ par $c^{-1}\sigma$.
- La permutation σ est égale au produit des cycles affichés par l'algorithme.

Exercice 9

Appliquer *manuellement* cet algorithme aux permutations suivantes : $\begin{pmatrix}0&1&2&3&4&5&6&7&8&9\\7&1&2&4&3&5&9&0&6&8\end{pmatrix}$, $\begin{pmatrix}0&1&2&3&4&5&6&7&8&9\\7&1&2&4&3&5&9&6&0&8\end{pmatrix}$ et $\begin{pmatrix}0&1&2&3&4&5&6&7&8&9\\1&0&2&4&3&5&7&6&9&8\end{pmatrix}$.

Exercice 10

Dans `permutation`, écrivez une fonction `void decompose_permutation(int perm[N])` qui prend en argument une permutation, et qui affiche sa décomposition en cycles en utilisant l'algorithme décrit ci-dessus. Pensez à utiliser les fonctions des exercices précédents.

exemple : pour `perm={2,4,0,7,3,5,6,1}`, la fonction doit afficher :

```
( 0 2 )( 1 4 3 7 )
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 18 | nombres binaires

Présentation

Dans ce TP, on veut effectuer différentes opérations sur des nombres binaires. Les fonctions définies devront être contenues dans une bibliothèque spécifique que vous allez définir, appelée `binaire`. Chaque fonction doit être testée depuis la fonction `main`, sans utiliser aucun `scanf` ou fonction similaire. Autrement dit, l'utilisateur ne doit avoir à saisir de valeurs. En particulier, les tableaux doivent être initialisés lors de leur déclaration.

1 Représentation

On décide de représenter un nombre binaire sous la forme d'un tableau d'entiers `int` de longueur maximale 100. Autrement dit, on manipulera dans ce TP des nombres dont l'expression en base 2 contient au plus 100 chiffres. **Attention**, cependant : pour faciliter le traitement de ces nombres, les chiffres seront placés dans le tableau à l'*inverse* de leur ordre habituel.

exemple : le nombre entier $(10111000)_2$ sera représenté par le tableau `{0,0,0,1,1,1,0,1}`, de longueur 8.

Exercice 1

Créez le fichier `main.c` ainsi que les fichiers de la bibliothèque `binaire`. Dans cette bibliothèque, définissez une constante `N` pour représenter la longueur maximale d'un nombre binaire.

Exercice 2

Dans `binaire`, écrivez une fonction `void affiche_binaire(int nombre[N], int longueur)` qui reçoit en paramètre un tableau représentant un nombre binaire, et un entier représentant le nombre de chiffres dans ce nombre binaire.

exemple : pour le tableau `{0,1,1,1,1,1,1,0,1}` (et donc la longueur 9), qui correspond au nombre binaire $(101111110)_2$, on doit obtenir *exactement* l'affichage suivant :

```
(1 0111 1110)_2
```

Exercice 3

Dans `binaire`, écrivez une fonction `int est_binaire(int nombre[N], int longueur)` qui détermine si le nombre binaire passé en paramètre avec sa longueur est valide. Puisqu'il est exprimé en base 2, ce nombre doit contenir exclusivement les chiffres 0 et 1. La fonction renvoie 1 si le nombre est valide, et 0 sinon.

exemples :

- Nombre `{1,0,1,1,1,0}` : la fonction renvoie 1 (nombre binaire $(011101)_2$).
- Nombre `{1,8,1,1,1,0}` : la fonction renvoie 0 (pas un nombre binaire).

2 Conversion décimal-vers-binaire

Exercice 4

Dans `binaire`, écrivez une fonction `void divise_binaire(int x, int *q, int *r)` qui calcule à la fois le quotient `q` et le reste `r` de la division entière de `x` par 2. Les deux résultats doivent être passés par adresse.

exemple : pour `x=7`, la fonction doit renvoyer `q=3` et `r=1`, puisque $x = 3 \times 2 + 1$.

Exercice 5

Lorsqu'on veut convertir un nombre exprimé en base 10 en un nombre $n \geq 0$ exprimé en base 2, on utilise l'algorithme de la décomposition binaire :

- On effectue d'abord la division entière de n par 2.
  - Le reste r de cette division correspond au *dernier* chiffre du nombre binaire.
  - Le quotient q est utilisé dans le reste du traitement.
- On applique le même principe à q, pour obtenir l'*avant-dernier* chiffre du nombre.
- On recommence ensuite avec le nouveau quotient pour obtenir l'*avant-avant-dernier* chiffre.
- Ainsi de suite, jusqu'à obtenir itérativement le premier chiffre du nombre.

exemple : pour $n = 13$:

$$\begin{aligned} 13 &= 6 \times 2 + \mathbf{1} \\ 6 &= 3 \times 2 + \mathbf{0} \\ 3 &= 1 \times 2 + \mathbf{1} \\ 1 &= 0 \times 2 + \mathbf{1} \end{aligned}$$

On a alors le résultat suivant : $n = (\mathbf{1101})_2$.

Dans `binaire`, écrivez une fonction `void decompose_binaire(int n, int nombre[N], int *longueur)` qui calcule la décomposition binaire du nombre `n` en utilisant l'algorithme décrit ci-dessus. Le résultat de cette décomposition doit être stocké dans le tableau `nombre`, et le nombre de chiffres obtenus doit être renvoyé par adresse au moyen du paramètre `longueur`. Bien sûr, le paramètre `n` est exprimé en base 10.

exemple : pour `n=13`, on obtient le tableau `{1,0,1,1}` et la longueur 4, qui correspondent au nombre binaire $(1101)_2$.

Remarque : vous *devez obligatoirement* utiliser la fonction `divise_binaire` dans `decompose_binaire`.

3 Conversion binaire-vers-décimal

Pour un nombre binaire contenant k chiffres, si on note b_i son $i^{ème}$ chiffre, alors le nombre n correspondant à son expression en base 10 correspond à un polynôme de degré k :

$$n = \sum_{i=0}^{k} b_i 2^i = b_k 2^k + b_{k-1} 2^{k-1} + b_{k-2} 2^{k-2} + \ldots + b_2 2^2 + b_1 2 + b_0$$

exemple : pour le tableau `{1,0,1,1}` (et donc la longueur 4), qui correspond au nombre binaire $(1101)_2$, on a $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13$.

La méthode de Ruffini-Horner permet de calculer les polynômes efficacement. Elle est basée sur la factorisation du polynôme, sous la forme suivante :

$$n = (b_k 2^{k-1} + b_{k-1} 2^{k-2} + b_{k-2} 2^{k-3} + \ldots + b_2 2^1 + b_1) 2 + b_0$$

$$= \Big((b_k 2^{k-2} + b_{k-1}2^{k-3} + b_{k-2}2^{k-4} + \ldots + b_2)2 + b_1\Big)2 + b_0$$
$$= \ldots$$
$$= \Big(\Big(\Big(\big((b_k 2 + b_{k-1})2 + b_{k-2}\big)2 + \ldots\Big)2 + b_2\Big)2 + b_1\Big)2 + b_0$$

Autrement dit, l'algorithme est le suivant :

- On multiplie le chiffre de rang le plus élevé (i.e. b_k) par 2, on additionne le chiffre de rang inférieur (b_{k-1}) ;
- On multiplie le résultat de l'opération précédente par 2, on additionne le chiffre de rang inférieur (b_{k-2}) ;
- On recommence itérativement, jusqu'au chiffre de rang le plus bas (i.e. b_0).

exemple : pour `{1,0,1,1}`, on réalise les étapes suivantes :

$$1 \times 2 + 1 = \mathbf{3}$$
$$\mathbf{3} \times 2 + 0 = \mathbf{6}$$
$$\mathbf{6} \times 2 + 1 = \mathbf{13}$$

Exercice 6

Dans `binaire`, écrivez une fonction `int recompose_binaire(int nombre[N], int longueur)` qui reçoit un nombre binaire et sa longueur, qui le convertit en base 10, puis qui le renvoie par valeur. Vous devez appliquer la méthode de Ruffini-Horner.

4 Opérations

Exercice 7

Dans `binaire`, écrivez une fonction `void calcule_addition(int nombre1[N], int longueur1, int nombre2[N], int longueur2, int nombre3[N], int* longueur3)` qui calcule la somme de `nombre1` et `nombre2`, puis qui place le résultat dans `nombre3`. La longueur du résultat est passée par adresse grâce au paramètre `longueur3`.

exemple : pour les nombres `nombre1={1,0,1}` (c'est-à-dire $(101)_2 = 5$) et `nombre2={1,1}` (c'est-à-dire $(11)_2 = 3$) on obtient les résultats `nombre3={0,0,0,1}` et `longueur3=4`, qui correspondent à la valeur binaire $(1000)_2 = 8$.

Exercice 8

Dans `binaire`, écrivez une fonction `int compare_binaire(int nombre1[N], int longueur1, int nombre2[N] , int longueur2)` qui compare les deux nombres binaires passés en paramètres, et renvoie :

- Un entier positif si le 1er nombre est supérieur au 2nd ;
- Zéro si les deux nombres sont égaux ;
- Un entier négatif si le 1er est inférieur au 2nd.

exemples :

- Pour `{1,1,1}` et `{0,1}` on obtient une valeur positive.
- Pour `{1,1,1}` et `{1,1,1}` on obtient une valeur nulle.
- Pour `{1,0,1}` et `{1,1,1}` on obtient une valeur négative.

5 Exponentiation

Exercice 9

Dans `binaire`, écrivez une fonction `int calcule_puissance1(float x, int n, float *res)` qui calcule *itérativement* x^n. La fonction doit renvoyer cette valeur x^n par

adresse, grâce au paramètre `res`, et elle doit renvoyer par valeur le nombre d'itérations nécessaire au calcule de x^n.

Remarque : quand vous testez votre fonction depuis la fonction `main`, affichez à la fois son résultat et le nombre d'itérations nécessaire au calcul.

exemple : pour `x=2.5` et `n=5`, la fonction renvoie `res=97.65625` et la valeur 5 (i.e. 5 itérations).

```
2.500000^5=97.656250 (5 iterations)
```

Exercice 10

Soit un réel $x \in \mathbb{R}$, et un entier n pour lequel on a la décomposition binaire $n = \sum_{i=0}^{k} b_i 2^i$. Alors x^n peut s'écrire sous la forme suivante :

$$x^n = \prod_{i=0}^{k} \left(x^{2^i}\right)^{b_i}$$

exemple : pour $n = 13$, on a :

$$x^{13} = x^{2^3+2^2+2^0} = x^{2^3} \times x^{2^2} \times x^{2^0} = ((x^2)^2)^2 \times (x^2)^2 \times x$$

Déduisez de cette propriété un algorithme pour calculer x^n à l'aide de la décomposition binaire de n. Implémentez cet algorithme sous la forme d'une fonction `int calcule_puissance2(float x, int n, float *res)`. Comme `calcule_puissance1`, la fonction `calcule_puissance2` doit renvoyer par valeur le nombre d'itérations nécessaires au calcul de x^n.

Remarque : votre fonction ne doit *pas* utiliser `calcule_puissance1`. Testez-la également en affichant à la fois le résultat et le nombre d'itérations nécessaires à son calcul.

exemples :

- Pour `x=2.5` et `n=5`, la fonction renvoie `res=97.65625` et la valeur 3 (i.e. 3 itérations).

```
2.5^5 = 97.65625 (2 iterations)
```

- Pour `x=2.5` et `n=13`, la fonction renvoie `res=149011.609375` et la valeur 4 (i.e. 4 itérations).

```
2.500000^13=149011.609375 (3 iterations)
```

Exercice 11

Pour des valeurs x et n quelconques, combien d'itérations sont nécessaires à `calcule_puissance1` pour calculer x^n ? Et à `calcule_puissance2` ? Laquelle de ces deux fonctions est la plus efficace, et pourquoi ?

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 19 | algorithme de Johnson

Présentation

Le but de ce TP est de travailler les tableaux et les types énumérés. Les en-têtes des fonctions ne sont pas précisées (seulement leur noms) : c'est volontaire, afin de vous faire travailler cet aspect de la programmation. Vous devez déterminer vous-même les types et paramètres les plus appropriés par rapport aux consignes.

1 Présentation

L'algorithme de Johnson (1963) permet de générer efficacement toutes les permutations de l'ensemble $\{0, \dots, n-1\}$.

1.1 Cas $n = 4$

Regroupons les permutations de $\{0,1,2,3\}$ en $3! = 6$ blocs de 4 permutations. On peut observer que le nombre 3 zigzague à travers les blocs :

0 1 2 **3**	**3** 0 2 1	2 0 1 **3**	**3** 2 1 0	1 2 0 **3**	**3** 1 0 2
0 1 **3** 2	0 **3** 2 1	2 0 **3** 1	2 **3** 1 0	1 2 **3** 0	1 **3** 0 2
0 **3** 1 2	0 2 **3** 1	2 **3** 0 1	2 1 **3** 0	1 **3** 2 0	1 0 **3** 2
3 0 1 2	0 2 1 **3**	**3** 2 0 1	2 1 0 **3**	**3** 1 2 0	1 0 3 **3**

On observe également que si on retire le nombre 3, on obtient pour chaque bloc une des $3! = 6$ permutations de $\{0,1,2\}$:

0 1 2	0 2 1	2 0 1	2 1 0	1 2 0	1 0 2

En reprenant le même principe, on peut regrouper ces permutations en $2! = 2$ blocs de 3 permutations, dans lesquels le nombre 2 zigzague, etc.

1.2 Représentation des permutations

Pour générer les permutations en utilisant cette observation, Johnson représente une permutation par des entiers munis d'une *girouette* : on associe à chaque entier une flèche indiquant une direction.

exemple :

$$\overleftarrow{0}, \overleftarrow{2}, \overleftarrow{4}, \overrightarrow{6}, \overleftarrow{5}, \overrightarrow{3}, \overrightarrow{1}$$

Définitions :

- Lorsque la girouette d'un entier *x* est orientée vers un autre entier *y*, on dit que *x regarde y*. Dans l'exemple précédent :
  - 2 regarde 0
  - 4 regarde 2
  - 6 regarde 5
  - 5 regarde 6
  - 3 regarde 1
- Lorsqu'un entier *x* est situé tout à gauche (respectivement tout à droite) et que sa girouette indique la gauche (respectivement la droite), on dit que *x* regarde *dehors*. Dans l'exemple précédent, 0 et 1 regardent dehors.
- On dit qu'un entier est *mobile* s'il ne regarde pas dehors et s'il regarde un entier plus petit que lui. Dans l'exemple, seuls les entiers 2, 4, 6 et 3 sont mobiles.

1.3 Algorithme

L'algorithme de Johnson génère toute les permutations de $\{0, \dots, n-1\}$ de la manière suivante :

- On commence avec la permutation $\overleftarrow{0}, \overleftarrow{1}, \overleftarrow{2}, \overleftarrow{3}, \overleftarrow{4}, \dots, \overleftarrow{n-1}$.
- On répète le traitement suivant :
  - On affiche la permutation courante.
  - On recherche le plus grand entier mobile (PGEM) *m*.
  - Si on trouve un entier mobile :
    - On échange m et l'entier que m regarde, sans changer le sens de leurs girouettes respectives.
    - On inverse le sens des girouettes de tous les entiers $k > m$.
    - On recommence.
- Si on ne trouve pas d'entier mobile, l'algorithme se termine.

2 Implémentation

Exercice 1

En utilisant l'algorithme de Johnson, vérifiez à la main que l'on obtient bien les permutations de $\{0,1,2,3\}$, comme décrit précédemment.

Exercice 2

Définissez un type énuméré `t_girouette`, permettant de décrire l'orientation d'une girouette. Définissez un type structure `t_entier_girouette` permettant de représenter un entier et sa girouette.

Exercice 3

Une permutation sera représentée par un tableau de valeurs de type `t_entier_girouette`. Écrivez une fonction `initialise_permutation` qui effectue l'initialisation (1er point de l'algorithme) de ce tableau, qui sera passé à la fonction en paramètre.

Remarque : notez que la taille n de la permutation est ici *variable* (à la différence du TP sur la décomposition de permutations, où on utilise une constante *N*). Cette taille n doit donc être passée en paramètre.

exemple : pour une taille $n = 6$, la fonction doit renvoyer la permutation $\overleftarrow{0}, \overleftarrow{1}, \overleftarrow{2}, \overleftarrow{3}, \overleftarrow{4}, \overleftarrow{5}, \overleftarrow{6}$.

Exercice 4

Écrivez une fonction `affiche_permutation` qui affiche une permutation passée en paramètre (3ème point de l'algorithme).

exemple : pour la permutation $\overleftarrow{0}, \overleftarrow{2}, \overleftarrow{4}, \overrightarrow{6}, \overleftarrow{5}, \overrightarrow{3}, \overrightarrow{1}$, on affichera :

```
<- <- <- -> <- -> ->
 0  2  4  6  5  3  1
```

Exercice 5

Écrivez une fonction `identifie_pgem` qui calcule la valeur et la position du plus grand entier mobile de la permutation passée en paramètre (4ème point de l'algorithme).

exemple : pour la permutation $\overleftarrow{0}, \overleftarrow{2}, \overleftarrow{4}, \overrightarrow{6}, \overleftarrow{5}, \overrightarrow{3}, \overrightarrow{1}$, la fonction doit renvoyer la position 3, qui correspond à $\overrightarrow{6}$.

Exercice 6

Écrivez une fonction `deplace_entier` qui reçoit en paramètres (au moins) une permutation et la position d'un entier dans cette permutation. On suppose que la position reçue correspond à un entier mobile. La fonction doit échanger l'entier situé à cette position, avec l'entier qu'il regarde (6ème point de l'algorithme).

exemples : pour la permutation $\overleftarrow{0}, \overleftarrow{2}, \overleftarrow{4}, \overrightarrow{6}, \overleftarrow{5}, \overrightarrow{3}, \overrightarrow{1}$

- Si on spécifie la position 1, on obtient $\overleftarrow{2}, \overleftarrow{0}, \overleftarrow{4}, \overrightarrow{6}, \overleftarrow{5}, \overrightarrow{3}, \overrightarrow{1}$.
- Si on spécifie la position 3, on obtient $\overleftarrow{0}, \overleftarrow{2}, \overleftarrow{4}, \overleftarrow{5}, \overrightarrow{6}, \overrightarrow{3}, \overrightarrow{1}$.

Exercice 7

Écrivez une fonction `inverse_girouette` qui inverse, dans une permutation, les girouettes de tous les entiers strictement supérieurs à un entier m (7ème point de l'algorithme). La permutation et m doivent être passés en paramètres.

exemple : pour la permutation $\overleftarrow{0}, \overleftarrow{2}, \overleftarrow{4}, \overrightarrow{6}, \overleftarrow{5}, \overrightarrow{3}, \overrightarrow{1}$ et $m = 4$, on obtient $\overleftarrow{0}, \overleftarrow{2}, \overleftarrow{4}, \overleftarrow{6}, \overrightarrow{5}, \overrightarrow{3}, \overrightarrow{1}$.

Exercice 8

Écrivez une fonction `johnson` qui prend en paramètre un entier n et affiche toutes les permutations de $\{0, \dots, n-1\}$ en utilisant l'algorithme de Johnson.

exemple : pour $n = 4$, on obtient l'affichage suivant (similaire à celui du début du sujet) :

```
 <-  <-  <-  <-
  0   1   2   3
 <-  <-  <-  <-
  0   1   3   2
 <-  <-  <-  <-
  0   3   1   2
 <-  <-  <-  <-
  3   0   1   2
 ->  <-  <-  <-
  3   0   2   1
 <-  ->  <-  <-
  0   3   2   1
 <-  <-  ->  <-
  0   2   3   1
 <-  <-  <-  ->
  0   2   1   3
 <-  <-  <-  <-
```

```
  2   0   1   3
 <-  <-  <-  <-
  2   0   3   1
 <-  <-  <-  <-
  2   3   0   1
 <-  <-  <-  <-
  3   2   0   1
 ->  ->  <-  <-
  3   2   1   0
 ->  ->  <-  <-
  2   3   1   0
 ->  <-  ->  <-
  2   1   3   0
 ->  <-  <-  ->
  2   1   0   3
 <-  ->  <-  <-
  1   2   0   3
```

```
 <-  ->  <-  <-
  1   2   3   0
 <-  <-  ->  <-
  1   3   2   0
 <-  <-  ->  <-
  3   1   2   0
 ->  <-  <-  ->
  3   1   0   2
 <-  ->  <-  ->
  1   3   0   2
 <-  <-  ->  ->
  1   0   3   2
 <-  <-  ->  ->
  1   0   2   3
```

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 20 | manipulation de dates

Présentation

Dans ce TP, on veut créer une bibliothèque `date` permettant de représenter et de manipuler des dates, en définissant nos propres types structure et énuméré. Cette bibliothèque sera ensuite utilisée dans d'autres TP.

1 Représentation des mois

Exercice 1

Créez un projet et les fichiers nécessaires : `date.c`, `date.h` et `main.c`. Initialisez ces fichiers de façon appropriée (cf. les TP précédents portant sur la définition de bibliothèque).

Exercice 2

On veut représenter les mois avec un type énuméré (et non pas avec un entier). Ce type doit contenir les symboles suivants : `jan`, `fev`, `mars`, `avr`, `mai`, `juin`, `juil`, `aout`, `sept`, `oct`, `nov` et `dec`. De plus, on veut que chaque mois soit associé à la valeur numérique appropriée : `jan` à 1, `fev` à 2, etc.

Dans `date.h`, définissez un type énuméré `t_mois` respectant ces contraintes.

Remarque : les types contenus dans une bibliothèque `xxxx` doivent toujours être définis dans le fichier d'en-tête `xxxx.h`.

Exercice 3

Dans `date`, écrivez la fonction `void affiche_mois(t_mois m)` qui affiche le texte complet correspondant au mois passé en paramètre.

exemples :

- `affiche_mois(jan)` doit afficher "`janvier`" ;
- `affiche_mois(fev)` doit afficher "`fevrier`" ;
- etc.

Contrainte : vous n'avez pas le droit d'utiliser `if` dans cette fonction.

2 Représentation des dates

Exercice 4

Dans `date.h`, Définissez un type structure `t_date` permettant de représenter une date décomposée en jour, mois et année, comme par exemple : *20 novembre 2011*. Le jour et l'année sont des entiers naturels, alors que le mois est de type `mois`.

Exercice 5

Dans `date`, écrivez une fonction `affiche_date` qui reçoit une date en paramètre et qui l'affiche *exactement* comme indiqué ci-dessous.

exemple : pour la date précédente, on obtient :

```
20 novembre 2011
```

Exercice 6

Dans `date`, écrivez une fonction `saisis_date` qui permet à l'utilisateur de saisir une date, et qui la renvoie par valeur.

exemple : saisie de la date donnée dans l'exemple précédent

```
Entrez le jour : 20
Entrez le mois :  11
Entrez l'année : 2006
```

Remarque : n'oubliez pas que les valeurs des types énumérés sont en réalité représentées sous la forme d'entiers.

Exercice 7

Dans `date`, écrivez une fonction `int compare_dates(t_date date1, t_date date2)` qui compare les deux dates passées en paramètres et renvoie un entier :

- Négatif si `date1` est *antérieure* à `date2` ;
- Zéro si `date1` et `date2` sont *exactement* les mêmes ;
- Positif si `date1` est *postérieure* à `date2`.

exemples :

- Pour les dates 01/01/2014 et 01/02/2014, la fonction renvoie -1 (ou autre valeur négative) ;
- Pour les dates 01/01/2014 et 01/01/2014, la fonction renvoie 0 ;
- Pour les dates 01/01/2014 et 01/02/2013, la fonction renvoie $+1$ (ou autre valeur positive).

Exercice 8

Dans `date`, écrivez une fonction `copie_date` qui reçoit en paramètre une date (i.e. une valeur de type `t_date`) et crée une nouvelle valeur de type `t_date` qui est la copie de cette date. À vous de décider si la copie doit être passée par adresse ou par valeur.

Remarque : il s'agit bien de créer une *nouvelle* valeur, et non pas de faire pointer deux pointeurs différents sur la *même* valeur en mémoire.

3 Calculs sur les années

Exercice 9

Dans `date`, écrivez une fonction `int est_bissextile(int annee)` qui prend une année en paramètre et renvoie 1 si elle est bissextile, et 0 sinon. D'après Wikipedia, une année est bissextile ssi :

- Soit elle est divisible par 4 mais pas par 100 ;
- Soit elle est divisible par 400.

exemples : 2000 et 2008 sont bissextiles, 1900 et 2011 ne sont pas bissextiles.

Exercice 10

Dans `date`, écrivez une fonction `int indique_jours(int annee, t_mois mois)` qui reçoit en paramètre une année et un mois, et renvoie le nombre de jours composant ce mois lors de cette année-là. Pensez à utiliser la fonction précédente.

Contrainte : vous ne pouvez utiliser qu'un seul `if` dans cette fonction.

Exercice 11

On veut améliorer la fonction `saisis_date`, de manière à s'assurer que les valeurs reçues sont bien valides. Dans `date`, écrivez une fonction `saisis_date_verif` qui effectue les tests suivants lors de la saisie de la date par l'utilisateur :

- L'année doit être un entier compris entre 1900 et 2014 (inclus) ;
- Le mois doit être un entier entre 1 et 12 (inclus) ;
- Le jour doit être un entier entre 1 et le nombre de jours contenus dans ce mois pour cette année-là.

Pour chacune de ces trois données, le programme doit demander à l'utilisateur d'entrer une valeur, en recommençant tant que la valeur n'est pas conforme à ces contraintes.

exemple :

```
Entrez l'annee (1900<=entier<=2011) : 2000
Entrez le mois (1<=entier<=12) : 13
Entrez le mois (1<=entier<=12) : 2
Entrez le jour (1<=entier<=29) : 30
Entrez le jour (1<=entier<=29) : 29
```

4 Calculs divers

Exercice 12

Dans `date`, écrivez une fonction `void ajoute_jour(t_date* date)` qui ajoute un jour à la date passée en paramètre. Attention, votre fonction doit tenir compte du mois et aussi de l'année (qui peut être bissextile).

exemples :

- Pour la date 01/01/2014, la fonction doit renvoyer 02/01/2014.
- Pour la date 31/01/2014, la fonction doit renvoyer 01/02/2014.
- Pour la date 28/02/2014, la fonction doit renvoyer 01/03/2014.
- Pour la date 28/02/2012, la fonction doit renvoyer 29/02/2012.

Exercice 13

Dans `date`, écrivez une fonction `int calcule_duree(t_date date1, t_date date2)` qui calcule le nombre de jours entre les deux dates passées en paramètres. Notez qu'il est possible d'obtenir une valeur négative, si `date1` n'est pas antérieure à `date2`.

Pour cela, vous devez implémenter l'algorithme suivant :

- Identifiez la date la plus petite, et en faire une copie.
- Tant qu'on n'a pas atteint la date la plus grande, on incrémente simultanément :
  - La date la plus petite ;
  - Une variable entière utilisée pour compter les jours.

Vous avez besoin des fonctions `compare_dates`, `copie_date`, et `ajoute_jour`.

exemples :

- Pour les dates 01/01/2014 et 12/02/2014, la fonction doit renvoyer 42.
- Pour les dates 10/01/2014 et 5/01/2014, la fonction doit renvoyer -5.
- Pour les dates 31/12/2013 et 01/01/2014, la fonction doit renvoyer 1.
- Pour les dates 26/02/2014 et 05/03/2015, la fonction doit renvoyer 372.
- Pour les dates 26/02/2012 et 05/03/2013, la fonction doit renvoyer 373.

Remarque : l'algorithme donné ici n'est pas forcément très efficaces du point de vue de la rapidité du calcul, mais il a l'avantage d'être simple à implémenter.

Exercice 14

On veut représenter les recettes (i.e. l'argent gagné) d'une entreprise pour chaque mois d'une année. Pour cela, on utilise un tableau dont chaque élément contient une valeur correspondant à la recette mensuelle : janvier dans la 1ère case, février dans la 2ème, etc.

Dans le fichier `main.c`, écrivez une fonction `void saisis_recettes(int recettes[])` qui reçoit en paramètre un tableau vide suffisamment grand, et qui demande à l'utilisateur de saisir la recette de chaque mois.

Contrainte : aucune variable entière ne doit être utilisée dans cette fonction (à part le tableau `recettes`).

exemple :

```
Entrez la recette du mois de janvier : 1534
Entrez la recette du mois de fevrier : 12351
Entrez la recette du mois de mars : 5621
...
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 21 | carrés latins

Présentation

Le but de ce TP est d'approfondir les tableaux en manipulant des matrices d'entiers particulières, appelée carrés latins. On aura pour cela recours à la notion de type énuméré, déjà abordée dans des TP précédents.

1 Définition

Un *carré latin* d'ordre n est une matrice $n \times n$ d'entiers compris entre 0 et $n-1$, et tel que *chaque ligne* et *chaque colonne* contient tous les entiers de 0 à $n-1$.

exemple : carré latin d'ordre 4 :

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

On notera m_{ij} les éléments $(0 \leq i,j < n)$ d'un carré latin m.

Exercice 1

Cherchez manuellement un carré latin d'ordre 3, et un autre d'ordre 5.

Exercice 2

Écrivez une fonction `void affiche_cl(int m[N][N])` qui affiche un carré latin `m` d'ordre `N`. La constante `N` doit être préalablement déclarée, avec la valeur que vous voulez.

exemple : pour le carré latin de l'exemple précédent, on obtient :

```
0   1   2   3
0   0   3   2
2   3   0   1
3   2   1   0
```

Exercice 3

Pour tout entier n, on peut construire un carré latin d'ordre n par permutation circulaire :

$$\begin{bmatrix} 0 & 1 & 2 & \cdots & n-1 \\ 1 & 2 & 3 & \cdots & 0 \\ 2 & 3 & 4 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n-1 & 0 & 1 & \cdots & n-2 \end{bmatrix}$$

Écrivez une fonction `void genere_cl_permutation(int m[N][N])` qui prend en argument une matrice m d'ordre `N`, et qui initialise cette matrice pour obtenir un carré latin en utilisant le principe décrit ci-dessus.

exemple : pour `N=10`, on obtient la matrice suivante :

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 0
2 3 4 5 6 7 8 9 0 1
3 4 5 6 7 8 9 0 1 2
4 5 6 7 8 9 0 1 2 3
5 6 7 8 9 0 1 2 3 4
6 7 8 9 0 1 2 3 4 5
7 8 9 0 1 2 3 4 5 6
8 9 0 1 2 3 4 5 6 7
9 0 1 2 3 4 5 6 7 8
```

2 Ronds & croix

On veut maintenant tester si une matrice donnée est bien un carré latin. Il est pour cela nécessaire d'écrire plusieurs fonctions effectuant des traitements secondaires. Au cours de ces traitements, on a besoin de distinguer deux types de valeurs, que l'on va représenter par deux symboles : croix (×) et rond (○).

Exercice 4

Définissez un type énuméré `t_symbole` contenant les valeurs `croix` et `rond`.

Exercice 5

Écrivez une fonction `void initialise_ronds_vecteur(t_symbole tab[N])` qui initialise le tableau passé en paramètre avec la valeur `rond`.

Exercice 6

Écrivez une fonction `int compte_ronds_vecteur(t_symbole tab[N])` qui renvoie le nombre d'éléments égaux à `rond` dans le tableau passé en paramètre.

exemple : pour le tableau {**rond**,croix,croix,croix,**rond**,**rond**}, on obtient la valeur 3.

Exercice 7

Écrivez une fonction `int contient_rond_vecteur(t_symbole tab[N])` qui détermine si le tableau passé en paramètre contient au moins un élément égal à `rond`. Si c'est le cas, la fonction renvoie 1, sinon elle renvoie 0.

exemples :

- Pour le tableau {**rond**,croix,croix,croix,**rond**,**rond**}, on obtient la valeur 1.
- Pour le tableau {croix,croix,croix,croix,croix,croix }, on obtient la valeur 0.

Remarque : pensez à utiliser la fonction précédente.

3 Test de propriété

On propose une méthode permettant de vérifier qu'une matrice carrée m est bien un carré latin.

- Pour chaque ligne d'indice i :
  - On initialise un tableau tab avec n ronds.
  - On parcourt la ligne numéro i de la matrice m, et pour chaque coefficient m_{ij} :
    - On place une croix dans la case numéro m_{ij} de tab.

    - Si, après avoir traité toute la ligne, le tableau tab contient toujours au moins un rond, alors la ligne numéro i ne contient pas tous les entiers compris entre 0 et $n-1$, et m n'est pas un carré latin.
    - Sinon (aucun rond), alors on passe à la ligne suivante.
- Si toutes les lignes sont correctes, alors fait la même chose avec les colonnes.
- Si toutes les colonnes sont elles-aussi correctes, alors il s'agit bien d'un carré latin.

exemple : on considère la matrice suivante :

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 1 & 3 & 2 \\ 2 & 3 & 0 & 3 \\ 0 & 2 & 2 & 2 \end{bmatrix}$$

- Pour la ligne 0, on obtient le tableau `{croix,croix,croix,croix}`. Il ne reste pas de rond, car cette ligne contient tous les entiers de 0 à 3.
- Pour la ligne 1, on obtient le tableau `{rond,croix,croix,croix}`. Il reste un rond dans l'emplacement 0 du tableau, car la ligne ne contient pas de valeur 0.
- Pour la ligne 2, on obtient le tableau `{croix,rond,croix,croix}`, car cette fois c'est la valeur 1 qui manque.
- Pour la ligne 3, on obtient le tableau `{croix,rond,croix,rond}`, car à la fois 1 et 3 manquent.

Exercice 8

Écrivez une fonction `int teste_vecteur(int m[N][N], int pos, int ligne)` qui reçoit une matrice carrée `m` et une position `pos`. La position est une valeur comprise entre 0 et $N-1$, et indiquant le numéro d'une ligne ou d'une colonne de `m`. Le dernier paramètre `ligne` indique si on doit considérer la *ligne* numéro `pos` (quand `ligne` vaut 1), ou bien la *colonne* numéro `pos` (quand `ligne` vaut 0). La fonction doit appliquer la méthode décrite ci-dessus sur la ligne/colonne concernée. Elle renvoie 1 si la ligne/colonne est valide, i.e. si elle contient toutes les valeurs entre 0 et $N-1$. Sinon, elle renvoie 0.

exemples : pour la matrice précédente, on obtient les résultats suivants :

- L'appel `teste_vecteur(m,0,1)` renvoie 1 ;
- L'appel `teste_vecteur(m,1,1)` renvoie 0 ;
- L'appel `teste_vecteur(m,2,1)` renvoie 0 ;
- L'appel `teste_vecteur(m,0,0)` renvoie 0 ;

Exercice 9

Écrivez une fonction `int est_carre_latin(int m[N][N])` qui teste si la matrice spécifiée est un carré latin. La fonction doit bien sûr utiliser `teste_vecteur`. Elle renvoie 1 si la matrice est bien un carré latin, et 0 sinon.

4 Orthogonalité

Soient $A = (a_{ij})$et $B = (b_{ij})$ $(0 \leq i,j < n)$ deux carrés latins d'ordre n. On appelle *matrice produit* de A par B la matrice notée $A \times B = ((a_{ij}, b_{ij}))$, dont chaque terme est une paire d'un élément de A et d'un élément de B.

Attention : il ne s'agit pas du produit matriciel classique. De plus, la matrice produit obtenue est carrée et de dimensions $n \times n$.

On dit que deux carrés latins A et B sont *orthogonaux* si la matrice produit $A \times B$ ne contient *pas* deux fois le même élément. La matrice produit obtenue est alors appelée *carré bi-latin* (ou gréco-latin) d'ordre n.

exemples : soient les carrés latins suivants :

$$A = \begin{bmatrix} 0 & 2 & 1 & 3 \\ 3 & 1 & 2 & 0 \\ 2 & 0 & 3 & 1 \\ 1 & 3 & 0 & 2 \end{bmatrix}, B = \begin{bmatrix} 0 & 3 & 2 & 1 \\ 2 & 1 & 0 & 3 \\ 1 & 2 & 3 & 0 \\ 3 & 0 & 1 & 2 \end{bmatrix}, C = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 3 & 0 & 1 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

On a alors les produits suivants :

- $A \times B = \begin{bmatrix} (0,0) & (2,3) & (1,2) & (3,1) \\ (3,2) & (1,1) & (2,0) & (0,3) \\ (2,1) & (0,2) & (3,3) & (1,0) \\ (1,3) & (3,0) & (0,1) & (2,2) \end{bmatrix}$: les couples (a_{ij}, b_{ij}) sont *tous* distincts, donc les carrés latins A et B sont *orthogonaux*, et $A \times B$ est un carré bi-latin.
- $A \times C = \begin{bmatrix} (0,0) & (2,1) & (1,2) & (3,3) \\ (3,3) & (1,0) & (2,1) & (0,2) \\ (2,2) & (0,3) & (3,0) & (1,1) \\ (1,1) & (3,2) & (0,3) & (2,0) \end{bmatrix}$: les carrés latins A et C ne sont *pas* orthogonaux car certains couples tel que $(2,1)$ apparaissent deux fois dans la matrice produit $A \times C$. Donc, cette matrice produit n'est *pas* un carré bi-latin.

Exercice 10

Cherchez manuellement deux carrés latins orthogonaux d'ordre 3.

Exercice 11

Écrivez les fonctions `initialise_ronds_matrice`, `compte_ronds_matrice`, et `contient_rond_matrice` qui utilisent respectivement `initialise_ronds_vecteur`, `compte_ronds_vecteur`, et `contient_rond_vecteur` pour effectuer le même traitement sur des matrices de croix/ronds (au lieu de simples vecteurs).

Exercice 12

On peut généraliser l'algorithme de la section précédente pour tester l'orthogonalité de deux carrés latins :

- On initialise une matrice m avec des ronds.
- Pour chaque valeurs de i et j ($0 \leq i, j < n$) :
  - On place une croix dans la case de m située sur la ligne a_{ij} et sur la colonne b_{ij}.
- Si, après avoir traité toutes les valeurs de i et j, la matrice m contient toujours au moins un rond, alors la matrice produit $A \times B$ ne contient pas tous les couples possibles, A et B ne sont pas orthogonaux et $A \times B$ n'est pas un carré bi-latin.

Écrivez une fonction `int sont_orthogonaux(int a[N][N], int b[N][N])` qui implémente l'algorithme ci-dessus pour tester si deux carrés latins sont orthogonaux.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 22 | représentation d'une promotion

Présentation

Le but de ce TP est de créer une base de données simple, permettant de décrire une promotion d'étudiants. Pour cela, nous allons utiliser des types structure et énuméré. Certains ont déjà été créés lors de TP précédents et sont disponibles dans la bibliothèque `date` fournie avec ce sujet. Les autres sont à créer lors de ce TP, dans une nouvelle bibliothèque `promotion`. La bibliothèque `chaine`, qui permet de manipuler des chaînes de caractères, est aussi fournie avec le sujet et sera utilisée dans la dernière partie du TP.

1 Représentation d'un étudiant

Exercice 1

Créez les fichiers nécessaires à la définition d'une bibliothèque appelée `promotion`. Cette bibliothèque doit être configurée pour elle-même utiliser les bibliothèques `date` et `chaine` fournies avec ce sujet. Créez aussi un fichier principal `main.c` qui permettra de tester les fonctions que vous écrirez. Notez que dans les deux premières parties du TP, vous n'avez pas besoin d'utiliser les fonctions de la bibliothèque `chaine`.

Exercice 2

Dans `promotion`, définissez un type énuméré `t_genre` permettant de représenter le genre d'une personne : soit féminin, soit masculin.

Exercice 3

Dans `promotion`, définissez un type structure `t_etudiant` permettant de représenter un étudiant par son nom, son prénom, sa date de naissance et son genre. Vous devez bien entendu utiliser les types `t_date` et `t_genre` pour les deux derniers champs. Vous définirez une constante `TAILLE_MAX_NOM` pour déterminer le nombre de caractères maximal qu'un nom ou prénom peut contenir.

Exercice 4

Dans `promotion`, écrivez une fonction `saisis_etudiant` permettant à l'utilisateur de saisir un étudiant.

exemples :

```
Entrez le prénom : Emre
Entrez le nom : Emreoglu
Date de naissance :
Entrez le jour : 20
Entrez le mois : 11
Entrez l'année : 1985
Entrez le genre : 1
Entrez le prénom : Zeynep
```

```
Entrez le nom : Öztürk
Date de naissance :
Entrez le jour : 12
Entrez le mois : 4
Entrez l'année : 1988
Entrez le genre : 0
```

Exercice 5

Dans `promotion`, écrivez une fonction `affiche_etudiant` qui affiche un étudiant passé en paramètre.

exemples :

- Pour Emre de l'exemple précédent :

```
Emre Emreoglu, né le 20 novembre 1985
```

- Pour Zeynep de l'exemple précédent :

```
Zeynep Öztürk, née le 12 avril 1988
```

Attention : reproduisez bien l'accord du participe passé du verbe naître !

2 Représentation d'une promotion

Exercice 6

Une promotion correspond à un ensemble d'étudiants. Dans `promotion`, définissez un type structure `t_promo` permettant de représenter un groupe d'étudiants grâce à un tableau `etudiants`. La taille maximale de la promotion devra être fixée grâce à une constante `TAILLE_MAX_PROMO`. La taille effective de la promotion (i.e. combien d'éléments du tableau sont utilisés pour représenter la promotion) devra être représentée sous la forme d'un champ `taille`.

Exercice 7

Dans `promotion`, écrivez une fonction `saisis_promotion` permettant de saisir les étudiants d'une promotion. Après avoir entré les données concernant un étudiant, l'utilisateur devra avoir la possibilité d'arrêter la saisie.

exemple :

```
-- Etudiant n.0 --
Entrez le prénom : Emre
Entrez le nom : Emreoglu
Date de naissance :
Entrez le jour : 20
Entrez le mois :  11
Entrez l'année : 1985
Entrez le genre : 1
Voulez-vous saisir un autre etudiant (O/N) ? O
-- Etudiant n.1 --
Entrez le prénom : Zeynep
Entrez le nom : Öztürk
Date de naissance :
Entrez le jour : 12
Entrez le mois : 4
Entrez l'année : 1988
Entrez le genre : 0
Voulez-vous saisir un autre etudiant (O/N) ? O
-- Etudiant n.2 --
Entrez le prénom : Can
Entrez le nom : Demirkesen
Date de naissance :
Entrez le jour : 8
Entrez le mois :  8
Entrez l'année : 1984
Entrez le genre : 1
Voulez-vous saisir un autre etudiant (O/N) ? N
3 etudiants ont été saisis.
```

Exercice 8

Dans `promotion`, écrivez une fonction `affiche_promotion` qui affiche toute la promotion passée en paramètre.

exemple :

```
0. Emre Emreoglu, né le 20 novembre 1985
1. Zeynep Öztürk, née le 4 avril 1988
2. Can Canoglu, né le 8 août 1984
```

3 Recherche dans la promotion

Pour effectuer des recherches, nous allons utiliser les fonctions de la bibliothèque `chaine`, qui permet de manipuler des chaines de caractères et qui a été écrite lors d'un TP précédent.

Exercice 9

Pour gagner du temps, dans le reste du TP, la promotion ne sera pas initialisée en utilisant la fonction `saisis_promo`, mais plutôt directement dans le programme, en utilisant la syntaxe qui permet d'initialiser une variable de type structure lors de sa déclaration.

Dans `promotion`, écrivez une fonction `t_promo initialise_promotion()` qui renvoie une variable de type `t_promo` contenant les étudiants suivants :

Nom	Prénom	Date de naissance	Genre
Emreoğlu	Emre	20/11/1985	Masculin
Öztürk	Zeynep	08/08/1988	Féminin
Özanoğlu	Özan	10/10/1986	Masculin
Canoğlu	Can	08/08/1984	Masculin
Küçük	Naz	02/02/1985	Féminin
Küçükkarakurt	Dilek	07/07/1985	Féminin
Onay	Sevgi	04/04/1985	Féminin
Koçak	Ozgur	06/05/1985	Masculin
Küçük	Hakan	03/03/1986	Masculin

Remarque : copiez-collez directement le tableau dans Eclipse, puis adaptez pour que le texte obtenu soit conforme à la syntaxe du C.

Exercice 10

dans `promotion`, écrivez une fonction `void cherche_homonymes(t_promo p, char* nom)` qui cherche dans la promotion `p` les étudiants qui ont *exactement* le nom `nom`, et qui les affiche. La fonction ne renvoie rien du tout, elle se contente d'afficher les étudiants trouvés.

exemple : résultat de la recherche de `"Kucuk"`

```
Résultat de la recherche pour la chaîne "Kucuk" :
- Naz Kucuk, née le 2 février 1985
- Hakan Kucuk, né le 3 mars 1986
```

Exercice 11

Dans `promotion`, écrivez une fonction `cherche_prefixe(t_promo p, t_promo* res, char* prefixe)` qui reçoit en paramètres deux promotions `p` et `res`, et une chaîne de caractères `prefixe`.

La fonction doit chercher dans `p` les étudiants dont le nom est préfixé par `prefixe`, mais cette fois *sans les afficher*. La promotion `res` (passée par adresse) est supposée initialisée, mais vide : la fonction doit remplir cette promotion avec les étudiants dont le nom est préfixé par `prefixe`. Le résultat est donc renvoyé par adresse, à travers `res`.

exemple : si on affiche, en utilisant `affiche_promo` depuis la fonction `main`, le résultat de la recherche de `"Kucuk"`, on obtient :

```
0. Naz Kucuk, née le 2 février 1985
1. Dilek Kucukkarakurt, née le 7 juillet 1985
2. Hakan Kucuk, né le 3 mars 1986
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 23 | partition d'un entier

Présentation

Le but de ce TP est d'approfondir la manipulation des structures et des pointeurs sur ces structures.

1 Définition et représentation

On appelle *partition* d'un entier n une suite $u = (u_0, u_1, \ldots, u_k, 0)$ finie et décroissante d'entiers se terminant par 0, et telle que $u_0 + u_1 + \cdots + u_k = n$. La valeur $k + 1$ représente la *longueur* de cette partition (le zéro final n'est pas compté).

exemples : quelques partitions de l'entier 12

- $u = (4,4,3,1,0)$ est une partitions de longueur 4, puisque $12 = 4 + 4 + 3 + 1$.
- $u = (5,4,3,0)$ est une partition de longueur 3, puisque : $12 = 5 + 4 + 3$.
- $u = (12,0)$ est une partition de longueur 1.

Exercice 1

En langage C, nous allons représenter une partition par une structure contenant deux champs :

- L'entier n concerné ;
- La suite u correspondant à sa partition, et représentée sous forme d'un tableau d'entiers terminé par un zéro ;

Définissez une constante `L_MAX` correspondant à la longueur maximale d'une partition. Puis, définissez un type structuré `t_partition` contenant les champs listés ci-dessus, en utilisant `L_MAX`.

exemples : une variable de type `t_partition` pourra être initialisée à la déclaration de manière à décrire les partitions de l'exemple précédent, en procédant de la façon suivante :

```
t_partition p1 = {12, {4,4,3,1,0}};
t_partition p2 = {12, {5,4,3,0}};
t_partition p3 = {12, {12,0}};
```

Exercice 2

Écrivez une fonction `void affiche_partition(t_partition p)` qui affiche la partition `p` passée en paramètre. Vous devez obtenir un affichage de la forme indiquée dans l'exemple ci-dessous.

exemple : pour les partitions de l'exemple précédent, on obtient :

```
12 = 4 + 4 + 3 + 1
12 = 5 + 4 + 3
12 = 12
```

Exercice 3

Écrivez une fonction `void copie_partition(t_partition p, t_partition* copie)` qui copie la partition `p` dans `copie`. Attention, chaque valeur originale de `p` doit être copiée individuellement dans la variable `copie`.

2 Comparaison

On peut définir un ordre appelé *ordre lexicographique* sur les partitions d'un entier n. Pour comparer les deux partitions $u = (u_0, u_1, \dots, u_p, 0)$ et $v = (v_0, v_1, \dots, v_q, 0)$, on applique la règle suivante :

- S'il existe un entier j tel que : $u_0 = v_0$, $u_1 = v_1$, …, $u_{j-1} = v_{j-1}$ et $u_j \neq v_j$, alors :
  - Si $u_j > v_j$, on a $u > v$.
  - Si $u_j < v_j$, on a $u < v$.
- Sinon, c'est-à-dire si $p = q$ et $u_i = v_i$ pour tout i, alors on a $u = v$.

exemples : comparaison de quelques partitions de l'entier $n = 12$:

$(12,0) > (5,4,3,1,0) > (5,4,2,1,0) > (4,4,4,0) > (4,4,3,1,0) > (4,4,2,1,1,0)$

Exercice 4

Écrivez une fonction `int compare_partitions(t_partition p1, t_partition p2)` qui utilise la règle expliquée ci-dessus pour comparer les partitions `p1` et `p2`. La fonction renvoie une valeur positive si $p_1 > p_2$, nulle si $p_1 = p_2$ et négative si $p_1 < p_2$.

exemples :

- Pour $(12,0)$ et $(4,4,4,0)$, la fonction renvoie une valeur positive ;
- Pour $(5,4,2,1,0)$ et $(5,4,2,1,0)$, la fonction renvoie zéro ;
- Pour $(4,4,4,0)$ et $(5,4,2,1,0)$, la fonction renvoie une valeur négative.

Remarque : on supposera que les deux partitions passées en paramètres concernent toutes les deux le même entier n.

3 Partitions triviales

On peut distinguer deux partitions *triviales* pour tout entier $n > 1$:

- La partition triviale *courte*, que l'on notera $\overline{u}$, est la partition de longueur 1 de cet entier, i.e. $\overline{u} = (n, 0)$.
- La partition triviale *longue*, que l'on notera $\underline{u}$, est la partition de longueur n de cet entier, i.e. $\underline{u} = (1, \dots, 1, 0)$.

exemples : pour $n = 12$, les deux partitions triviales sont $\overline{u} = (12,0)$ et $\underline{u} = (1,1,1,1,1,1,1,1,1,1,1,1,0)$.

Remarque : pour toute partition u d'un entier n, on a la propriété $\underline{u} \leq u \leq \overline{u}$.

Exercice 5

Écrivez une fonction `void calcule_partition_courte(int n, t_partition *p)` qui calcule la partition triviale courte de `n` et place le résultat dans le paramètre `p` passé par adresse.

Exercice 6

Même chose avec la fonction `void calcule_partition_longue(int n, t_partition *p)` qui calcule $\underline{u}$.

4 Génération

Exercice 7

Si u est la partition d'un entier n, on peut calculer la partition v qui précède u dans l'ordre lexicographique à l'aide de l'algorithme suivant :

- Tant qu'il existe un entier j tel que $u_j > 1$:

- On pose :
  - $v_0 = u_0, \dots, v_{j-1} = u_{j-1}$
  - $v_j = u_j - 1$
- On calcule l'entier a égal au nombre de 1 dans la partition u, c'est-à-dire l'entier a tel que : $u_{j+1} = u_{j+2} = \cdots = u_{j+a} = 1$.
- On calcule le quotient q et le reste r de la division euclidienne de $a + 1$ par v_j, et on pose :
  - $v_{j+1} = v_{j+2} = \cdots = v_{j+q} = v_j$
  - $v_{j+q+1} = r$ (si $r \neq 0$)
- On ajoute un 0 comme dernier terme de la suite v.

exemple : on considère deux partitions de $n = 12$:

- $u = (5,4,\mathbf{3},0)$
  - Alors : $j = 2$, $v_i = 2$, $a = 0$, $q = 0$ et $r = 1$.
  - Donc : $v = (5,4,\mathbf{2},1,0)$.
- $u = (4,3,\mathbf{2},1,1,1,0)$
  - Alors : $j = 2$, $v_i = 1$, $a = 3$, $q = 4$ et $r = 0$.
  - Donc : $v = (4,3,\mathbf{1},1,1,1,1,0)$.

En appliquant *manuellement* cet algorithme, calculez les partitions de $n = 12$ qui précèdent les partitions :

- (12,0)
- (5,5,1,1,0)
- (1,1,1,1,1,1,1,1,1,1,1,0).

Exercice 8

Écrivez une fonction `void calcule_partition_suivante(t_partition p0, t_partition* p1)` qui calcule la partition `p1` suivant `p0` dans l'ordre lexicographique, en utilisant l'algorithme précédent.

Exercice 9

En utilisant les fonctions précédentes, écrivez une fonction `void affiche_toutes_partitions(int n)` qui calcule et affiche toute les partitions de l'entier `n` dans l'ordre lexicographique.

exemple : pour $n = 9$

```
9 = ( 9 )
9 = ( 8 1 )
9 = ( 7 2 )
9 = ( 7 1 1 )
9 = ( 6 3 )
9 = ( 6 2 1 )
9 = ( 6 1 1 1 )
9 = ( 5 4 )
9 = ( 5 3 1 )
9 = ( 5 2 2 )
9 = ( 5 2 1 1 )
9 = ( 5 1 1 1 1 )
9 = ( 4 4 1 )
9 = ( 4 3 2 )
9 = ( 4 3 1 1 )
```

```
9 = ( 4 2 2 1 )
9 = ( 4 2 1 1 1 )
9 = ( 4 1 1 1 1 1 )
9 = ( 3 3 3 )
9 = ( 3 3 2 1 )
9 = ( 3 3 1 1 1 )
9 = ( 3 2 2 2 )
9 = ( 3 2 2 1 1 )
9 = ( 3 2 1 1 1 1 )
9 = ( 3 1 1 1 1 1 1 )
9 = ( 2 2 2 2 1 )
9 = ( 2 2 2 1 1 1 )
9 = ( 2 2 1 1 1 1 1 )
9 = ( 2 1 1 1 1 1 1 1 )
9 = ( 1 1 1 1 1 1 1 1 1 )
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 24 | rotation d'un carré

Présentation

Le but de ce TP est d'approfondir la manipulation des matrices et des structures. Pour cela, nous allons tracer un carré et lui faire subir des rotations.

Ce TP utilise la SDL, mais ne le configurez pas encore pour la SDL maintenant. Attendez plutôt l'exercice nécessitant la SDL, afin de pouvoir bénéficier avant cela de la sortie texte pour déboguer les premières fonctions.

1 Matrice de rotation

Exercice 1

Écrivez une fonction `affiche_matrice` qui affiche comme dans l'exemple ci-dessous un tableau de réels de type `double`, représentant une matrice 2×2.

exemple : pour la matrice `{{-1,-2},{3,4}}`, on doit obtenir *exactement* l'affichage suivant (utilisez le format approprié dans `printf`) :

```
( -1.000000 -2.000000 )
(  3.000000  4.000000 )
```

Exercice 2

On rappelle que la matrice de rotation d'angle θ s'écrit :

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

Définissez une constante `PI` pour représenter π ($\pi \approx 3{,}1415926$). Puis, écrivez une fonction `initialise_rotation` qui initialise une matrice passée en paramètre, de manière à ce qu'elle représente la rotation d'angle θ, lui aussi passé en paramètre.

exemple : pour l'angle $\theta = \pi/3$, on obtient la matrice suivante :

```
(  0.500000 -0.866025 )
(  0.866025  0.500000 )
```

Remarque : dans cet exercice on a besoin d'utiliser les fonctions *cosinus* et *sinus*. Elles sont implémentées sous la forme des fonctions `cos` et `sin` définies dans la bibliothèque `math.h`. Vous devez, bien entendu, configurer votre projet de manière à utiliser cette bibliothèque. Notez que ces fonctions prennent en paramètre un angle exprimé en *radians*.

2 Rotation d'un vecteur

Exercice 3

Définissez une structure `s_vecteur` représentant un vecteur dans un espace à 2 dimensions, i.e. un vecteur décrit par deux coordonnées réelles x et y.

Exercice 4

Écrivez une fonction `affiche_vecteur` qui affiche un vecteur en respectant exactement la forme décrite dans l'exemple ci-dessous.

exemple : affichage du vecteur `{0,-1}` :

```
(  0.000000 )
( -1.000000 )
```

Exercice 5

Écrivez une fonction `void applique_rotation(double m[2][2], struct s_vecteur v, struct s_vecteur* resultat)` qui prend en paramètre un vecteur `v` et une matrice de rotation `m`, et calcule les coordonnées du vecteur image. Ce résultat est renvoyé via le paramètre `resultat`.

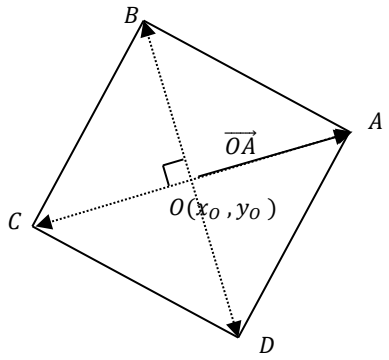
exemple : pour la matrice et le vecteur des exemples précédents, on obtient :

```
(  0.866025 )
( -0.500000 )
```

3 Rotation d'un carré

Exercice 6

On peut calculer les coordonnées des 4 sommets d'un carré $ABCD$ à partir des coordonnées de son centre $O(x_O, y_O)$ et des coordonnées du vecteur $\overrightarrow{OA}(x_{\overrightarrow{OA}} y_{\overrightarrow{OA}})$, qui relie le centre O au sommet A :



On a alors :

- $x_A = x_O + x_{\overrightarrow{OA}}$ et $y_A = y_O + y_{\overrightarrow{OA}}$
- $x_C = x_O - x_{\overrightarrow{OA}}$ et $y_C = y_O - y_{\overrightarrow{OA}}$

Les coordonnées du vecteur $\overrightarrow{OB}$ s'obtiennent par rotation d'angle $\pi/2$ et de centre O du vecteur $\overrightarrow{OA}$, donc :

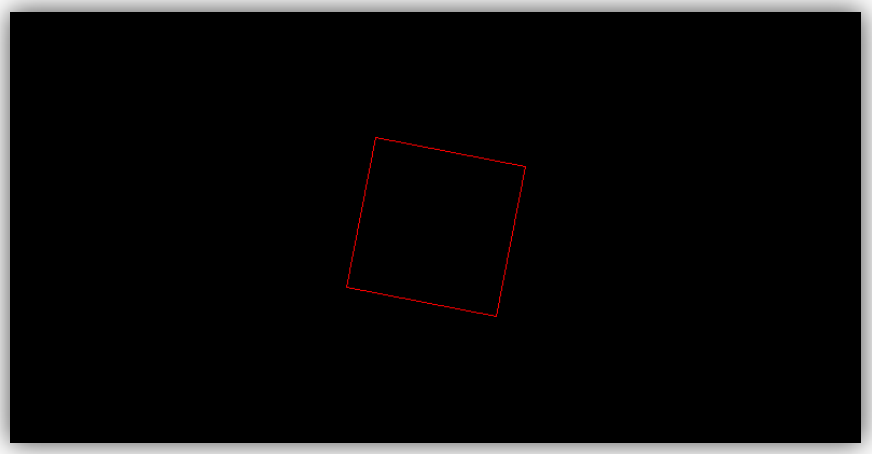
- $x_{\overrightarrow{OB}} = -y_{\overrightarrow{OA}}$ et $y_{\overrightarrow{OB}} = x_{\overrightarrow{OA}}$

On en déduit alors les coordonnées des sommets B et D.

À l'aide de ces remarques, écrivez une fonction `void dessine_carre(int x_O, int y_O, struct s_vecteur v_OA, Uint32 coul)` qui dessine à l'écran le carré $ABCD$ défini par son centre O le vecteur $\overrightarrow{OA}$.

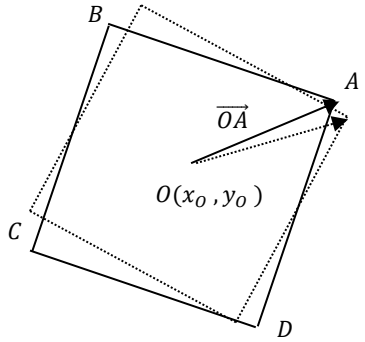
Remarque : pensez à utiliser la fonction `dessine_segment` de la bibliothèque `graphisme`.

exemple : pour le centre $O = (500{,}300)$ et le vecteur $\overrightarrow{OA} = (83, -56)$, on obtient la figure suivante :



Exercice 7

On veut maintenant animer ce carré, en le faisant tourner autour de son centre O. Écrivez une fonction `void tourne_carre(int x_O, int y_O, struct s_vecteur v_OA, Uint32 coul, int k)` qui fait tourner d'un tour complet le carré $ABCD$ en appliquant à chaque étape une rotation d'angle $2\pi/k$ au vecteur $\overrightarrow{OA}$:



L'algorithme est le suivant :

- Tant qu'on n'a pas fait un tour complet,
    1. Tracer le carré en rouge ;
    2. Attendre un certain temps correspondant à la constante `DELAI` définie dans la bibliothèque `graphisme` (et exprimé en ms) ;
    3. Tracer le même carré en noir (pour effacer le carré rouge)
    4. Calculer le nouveau carré en effectuant la rotation d'angle $2\pi/k$.
    5. Recommencer.

Testez votre fonction avec plusieurs valeurs de k. Vous remarquerez que l'animation obtenue n'est pas très fluide : ceci est dû au fait que nous n'utilisons pas de tampon (ou buffer). Nous aborderons ce point dans un futur TP.

Exercice 8

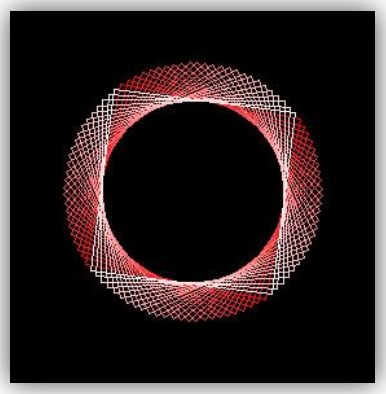
Faites une copie de la fonction précédente, que vous appellerez `tourne_carre2`. Dans cette fonction effectuez les modifications suivantes :

- Supprimez l'étape 3 de l'algorithme précédent ;
- Limitez la rotation totale à un quart-de-tour au lieu d'un tour complet (i.e. $\pi/2$ au lieu de 2π) ;

- Supprimez le paramètre `coul`, et recalculez la couleur à chaque itération. Partez d'une couleur purement rouge (donc $r = 255$, $g = 0$, $b = 0$) et augmentez les composantes verte et bleu de $255/\frac{k}{4}$ à chaque itération (par conséquent, à la dernière itération, la couleur utilisée sera le blanc).

Remarque : pensez à utiliser la fonction `convertis_rvb` de la bibliothèque `graphisme`.

Vous devez obtenir une figure ressemblant à ceci :



Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 25 | zoom d'une image

Présentation

Le but de ce TP est d'implémenter des fonctions permettant de modifier la taille (graphique) d'une image. Ceci nécessitera de traiter des matrices représentant ces images.

1 Préparation

L'archive fournie avec le sujet contient un album d'images déjà utilisées dans certains TP précédents, ainsi que la bibliothèque `graphisme`, elle aussi déjà utilisé. Elle a été complétée de certaines fonctions particulières spécialement pour ce TP, cf. le code source. Vous aurez tout particulièrement besoin des fonctions suivantes :

- `charge_image` : charge un fichier contenant une image, et la renvoie sous la forme de `surface`.
- `sauve_surface` : enregistre une surface sous la forme d'un fichier image.
- `dessine_surface` : affiche une dans la fenêtre principale.
- `initialise_surface` : crée une nouvelle surface (image) vide.
- Des variantes de fonctions utilisées dans les TP précédents, qui travaillent sur une surface passée en paramètre au lieu de travailler sur la fenêtre principale. Leur nom se termine en `_surface`. Exemples : `allume_pixel_surface`, `couleur_pixel_surface`, etc.

Une bonne partie de ce TP consiste à manipuler des images, sous la forme de variables de type `SDL_Surface`. Vous avez besoin de savoir que ce type est une structure, dont les champs `w` et `h` correspondent respectivement à la largeur (*width*) et à la hauteur (*height*) de l'image concernée. Vous pouvez lire ces champs, mais pas les modifier.

Exercice 1

Toutes vos fonctions doivent être écrites dans le fichier `main.c`. Écrivez d'abord une fonction `void trace_surface(SDL_Surface surface)` qui reçoit une surface (donc une image) en paramètre. La fonction doit :

1) Effacer la fenêtre principale ;
2) Afficher l'image `surface` à partir du point $(0,0)$;
3) Attendre que l'utilisateur appuie sur une touche.

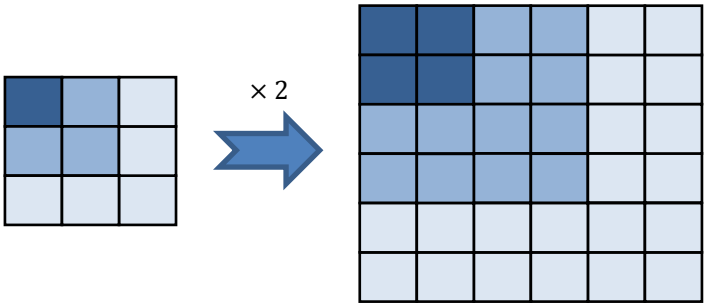
Exercice 2

Complétez la fonction `main` pour afficher l'une des photos fournies avec le sujet. Vous aurez besoin d'utiliser `charge_image` et `trace_surface`.

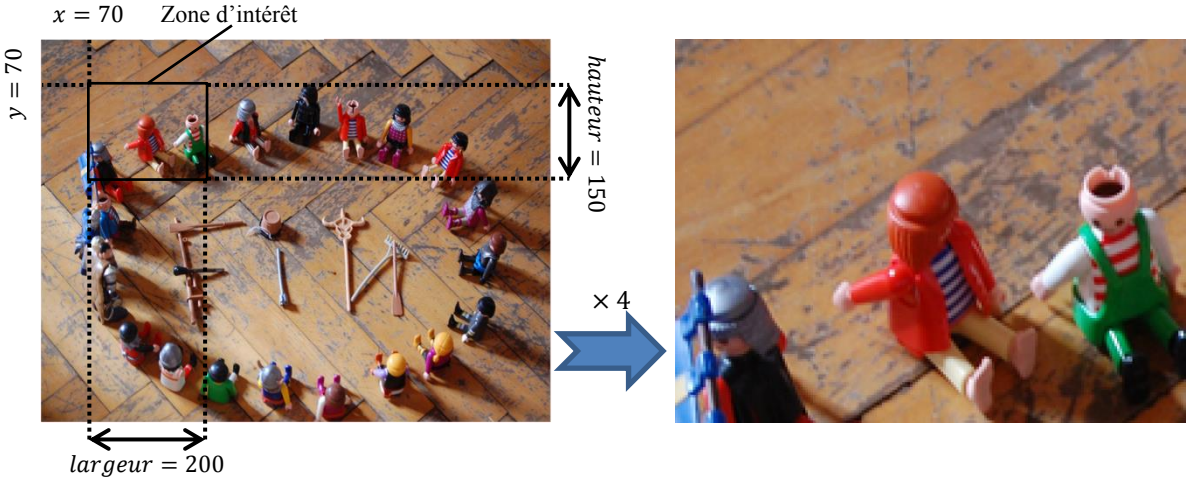
2 Agrandissement

Une méthode élémentaire pour faire un zoom grossissant (*zoom in*) d'un facteur n d'une image (n est un entier), consiste à transformer chaque pixel de l'image initiale en un carré de $n \times n$ pixels où tous les pixels ont exactement la même couleur que le pixel initial.

Une image ne peut pas toujours être entièrement zoomée et affichée à l'écran, car il est possible qu'elle soit trop grande. Pour cette raison, dans ce TP on ne zoome qu'une partie de l'image initiale, appelée *zone d'intérêt*. Celle-ci est définie par l'utilisateur, de manière à ne pas être plus grande que la fenêtre dans laquelle on va ensuite dessiner (sinon, cela provoquera une erreur d'exécution).



Considérons par exemple la figure suivante. L'image de gauche est l'image originale. La zone d'intérêt est définie par son coin supérieur gauche et par ses dimensions. Ici, on a sélectionné le point de coordonnées (150,170), et les dimensions respectives 200 et 150 pixels pour la largeur et la hauteur. De plus, on a agrandit l'image par un facteur 4. En appliquant le principe expliqué ci-dessus, on obtient l'image de droite.



Exercice 3

Écrivez la fonction d'en tête `SDL_Surface* agrandis(SDL_Surface* source, int x, int y, int largeur, int hauteur, int coef)`, où `source` pointe sur la surface à zoomer, `x` et `y` sont les coordonnées du coin supérieur gauche de la zone d'intérêt, `largeur` et `hauteur` sont ses dimensions, et `coef` est le coefficient d'agrandissement.

La fonction doit créer une nouvelle surface destinée à contenir le résultat de l'agrandissement. L'agrandissement est réalisé en traitant individuellement chaque pixel de la zone d'intérêt :

- Lire la couleur du pixel original (dans `source`) ;
- Allumer dans la nouvelle surface autant de pixels de même couleur qu'il est nécessaire.

Dans l'exemple précédent, l'image de droite est obtenue en effectuant l'appel `agrandis(photo,70,70,200,150,4)` sur la surface représentant l'image de gauche.

Exercice 4

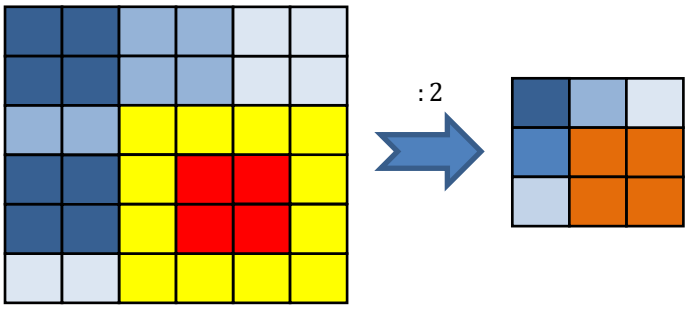
Complétez la fonction `main` de manière à :

4) Afficher un rectangle correspondant à la zone d'intérêt (comme dans l'image de gauche de la figure précédente) ;
5) Attendre que l'utilisateur appuie sur une touche ;
6) Agrandir la photo chargée précédemment ;
7) Afficher le résultat de cet agrandissement à l'écran (toujours en utilisant `trace_surface`) ;
8) Enregistre (grâce à `sauve_surface`) le résultat de l'agrandissement sous la forme d'un fichier bitmap de nom `imagex.zoomin.bmp`, où `x` correspond au numéro du fichier d'image ;
9) Attendre que l'utilisateur appuie sur une touche.

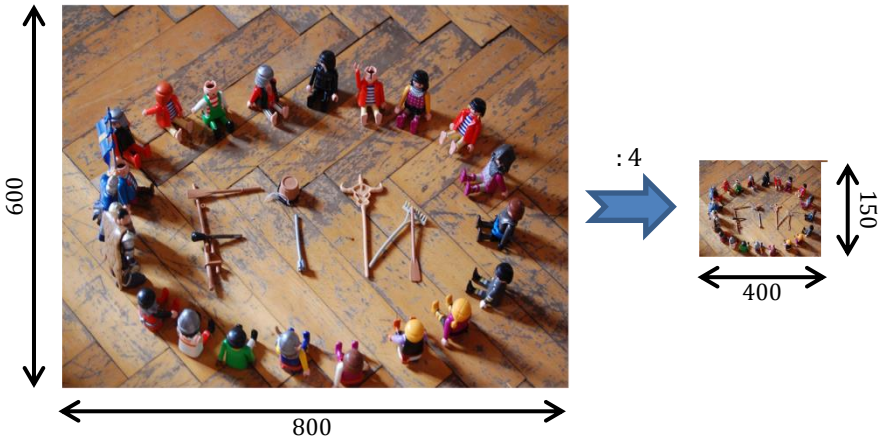
Remarque : attention aux paramètres que vous utilisez. Si l'image obtenue à la suite de l'agrandissement est *trop grande*, elle ne rentrera pas dans la fenêtre, et cela provoquera donc une erreur d'exécution quand vous essaierez de l'afficher.

3 Réduction

On peut appliquer un principe similaire pour *réduire* une image (*zoom out*). La différence est que cette fois, il y a une perte d'information : la résolution de l'image étant moins bonne, certains pixels doivent être supprimés. Une façon naïve de procéder est de substituer à un groupe de pixels, un seul pixel dont la couleur est la moyenne des couleurs originales. Ce principe est illustré dans la figure ci-dessous. Ainsi, les blocs de 3 pixels jaunes et 1 pixel rouge deviennent chacun 1 pixel orange.



Bien sûr, contrairement à l'agrandissement, il n'y a pas de problème avec la dimension de l'image, et la réduction s'applique à l'intégralité de l'image (il n'y a pas de zone d'intérêt). Pour l'exemple précédent, en réduisant par un facteur de 4, on obtient ainsi le résultat suivant :



Remarque : par souci de simplicité, on n'utilisera que des coefficients de réduction qui sont des diviseurs des dimensions de l'image.

Exercice 5

Écrivez la fonction `SDL_Surface* reduis(SDL_Surface* source, int coef)`, où `source` pointe sur la surface à zoomer et `coef` est le coefficient de réduction. La fonction renvoie une nouvelle surface, plus petite, contenant la version réduite de `source`.

Exercice 6

Complétez la fonction `main` pour :

10) Afficher l'image originale à nouveau ;
11) Calculer la version réduite de l'image avec `reduis` ;
12) Enregistrer cette image sous le nom `imagex.zoomout.bmp` ;
13) Attendre que l'utilisateur appuie sur une touche.

Remarque : ces actions viennent compléter les actions déjà programmées dans la fonction `main`.

Exercice 7

Écrivez une fonction `SDL_Surface* reduis_agrandis(SDL_Surface* source, int coef)` qui :

- Réduit la surface `source` d'un facteur `coef` ;
- Applique à la surface obtenue un agrandissement équivalent.

On obtient alors une image de même taille que l'image originale. La fonction doit renvoyer cette image sous la forme d'un pointeur sur une nouvelle surface, comme dans les fonctions précédentes.

Exercice 8

Complétez la fonction `main` pour :

14) Afficher l'image originale à nouveau ;
15) Calculer la version réduite-agrandie de l'image avec `reduis_agrandis` ;
16) Enregistrer cette image sous le nom `imagex.zoomoutin.bmp` ;
17) Attendre que l'utilisateur appuie sur une touche.

Visualiser l'image originale et `imagex.zoomoutin.bmp` : vous devriez visualiser la perte d'information subie lors de la réduction, et mentionnée dans les explications concernant la réduction.

Remarque : comme leur numérotation l'indique, ces actions viennent compléter les actions déjà programmées dans la fonction `main`.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 26 | automates finis

Présentation

Le but de ce TP est d'approfondir l'utilisation des types structurés et des matrices, via l'étude d'automates finis. Tous les types et fonctions devront être écrits dans le fichier `main.c`.

1 Définition

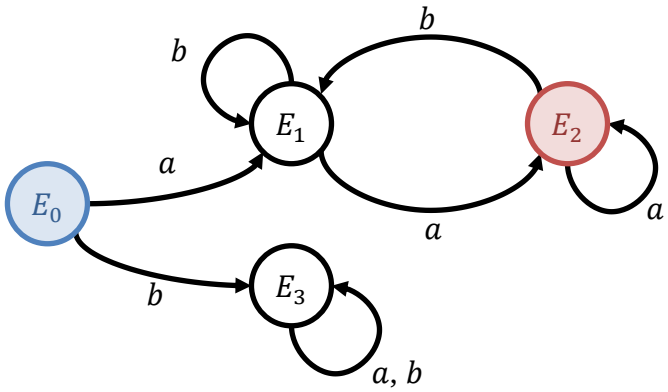
Un automate fini est un modèle mathématique d'ordinateur très simple, capable de reconnaître des *mots* formés de *lettres* contenus dans un *alphabet* prédéfini et respectant une certaine *structure*. On appelle *longueur* d'un mot le nombre de lettres qui le composent. On note ε le mot de longueur 0, appelé *mot vide*.

exemple : avec l'alphabet $\Sigma = \{a, b\}$, on peut produire des mots composés de a et de b, tels que b, $abbbab$, bab, etc.

Les automates les plus simples ont une sortie binaire : tout mot est soit *accepté* par l'automate, soit *refusé*. On peut représenter un automate par un *graphe orienté* :

- Chaque nœud s'appelle un *état* ;
- Chaque lien est un *changement d'état*, associé à une lettre de l'alphabet.

exemple : soit l'automate représenté par le graphe suivant :



Cet automate comporte 4 états E_0, E_1, E_2 et E_3 :

- L'état E_0 représenté en bleu est un état *initial*.
- L'état E_2 représenté en rouge un état *acceptant*
- Les états E_0, E_1 et E_3 sont des états *refusant*.

Les liens de cet automate sont associés aux lettres a ou b, ce qui signifie qu'il est défini pour l'alphabet $\{a, b\}$.

En parcourant le graphe d'un automate depuis l'un de ses états initiaux, on peut décider quels sont les mots qu'il accepte. Ce parcours est réalisé de la façon suivante :

1. On démarre d'un état initial, qui devient l'état courant.

2. On considère chaque lettre constituant le mot, en commençant par la 1ère. À chaque fois, on se déplace dans l'automate en suivant le lien associé à la lettre considérée.
3. Si le parcours se termine sur un état acceptant, alors le mot est accepté. Sinon, il est refusé.

exemple : dans notre graphe d'exemple, on n'a qu'un seul état initial E_0, donc tout parcours démarre obligatoirement de ce nœud. On n'a qu'un seul état acceptant E_2, donc tout mot accepté se termine obligatoirement sur ce nœud.

- Quelques mots acceptés par cet automate, avec le parcours correspondant :
  - $aaba$: E_0, E_1, E_2, E_1, E_2
  - $aaaaa$: $E_0, E_1, E_2, E_2, E_2, E_2$
  - $abbaba$: $E_0, E_1, E_1, E_1, E_2, E_1, E_2$
- Quelques mots refusés par cet automate :
  - $baaa$: E_0, E_3, E_3, E_3, E_3
  - $abbb$: E_0, E_1, E_2, E_2, E_2
  - $abab$: E_0, E_1, E_2, E_3, E_2

On peut démontrer que l'ensemble des mots acceptés par cet automate sont les mots de longueur au moins égale à 2 et qui commencent et se terminent par la lettre a.

On peut représenter un automate sous la forme d'une *matrice de transition*, qui indique les changements possibles pour chaque lettre, à partir de chaque état.

exemple : pour l'automate précédent, on a la matrice de transition suivante :

	a	b
E_0	E_1	E_3
E_1	E_2	E_1
E_2	E_2	E_1
E_3	E_3	E_3

Dans ce TP, on veut implémenter un automate de ce type. Pour simplifier le problème, on fera les hypothèses suivantes :

- L'automate ne contient qu'un seul état initial et un seul état acceptant ;
- L'alphabet est toujours $\Sigma = \{a, b\}$;
- La longueur du mot traité est constante.

2 Implémentation

Exercice 1

Dans `main.c`, définissez les trois constates suivantes, à utiliser dans le reste du TP :

- `LONGUEUR_MAX_MOT` : longueur maximale d'un mot (valeur laissée à votre choix) ;
- `NB_MAX_ETATS` : nombre maximal d'états autorisé pour un automate (utilisez une valeur supérieure ou égale à 5 pour ce TP) ;
- `TAILLE_ALPHABET` : nombre de lettres dans l'alphabet (utilisez 2, puisque notre alphabet est toujours $\{a, b\}$ dans ce TP).

Exercice 2

On représente un automate en mémoire à l'aide d'une structure contenant 4 champs :

- `nb_etats` : nombre d'états de l'automate ;
- `etat_initial` : numéro de l'état initial ;

- `etat_acceptant` : numéro de l'état acceptant ;
- `matrice_transition` : matrice de transition.

Dans `main.c`, définissez un type structuré appelé `t_automate`, qui contient ces 4 champs.

Exercice 3

Écrivez la fonction `void affiche_automate(t_automate a)` qui affiche les champs de l'automate a passé en paramètre.

Pour tester votre fonction, initialisez dans la fonction `main` un automate correspondant à l'exemple précédent. Vous devez effectuer une initialisation à la déclaration, i.e. en utilisant la syntaxe à base d'accolades `{...}` vue en cours.

exemple : si l'automate passé en argument est celui de l'exemple précédent, on obtiendra *exactement* l'affichage ci-dessous :

```
Le nombre d'etats de l'automate est 4
L'etat initial est E0
L'etat acceptant est E2
La matrice de transition est :
      a     b
E0    E1    E3
E1    E2    E1
E2    E2    E1
E3    E3    E3
```

Exercice 4

Écrivez la fonction `void saisis_automate(t_automate* a)` qui demande à l'utilisateur de saisir les différents champs constituant un automate. Les valeurs saisies sont utilisées pour initialiser l'automate passé en paramètre.

exemple :

```
Entrez le nombre d'etats : 4
Entrez l'etat initial : 0
Entrez l'etat acceptant : 2
- Entrez les transitions de l'etat E0 :
  a : 1
  b : 3
- Entrez les transitions de l'etat E1 :
  a : 2
  b : 1
- Entrez les transitions de l'etat E2 :
  a : 2
  b : 1
- Entrez les transitions de l'etat E3 :
  a : 3
  b : 3
```

Exercice 5

Écrivez une fonction `int applique_transition(t_automate a, int etat_courant, char lettre)`, qui reçoit en paramètres l'état courant, i.e. l'état actuellement occupé, et une lettre de l'alphabet Σ. La fonction calcule et retourne l'entier correspondant à l'état obtenu en suivant le lien associé à la lettre à partir de l'état courant, tel que décrit dans la matrice de transition de l'automate.

Remarque : cette fonction s'appelle la *fonction de transition* de l'automate. Son implémentation tient en une seule ligne.

exemples :

- L'appel `applique_transition(a,1,'a')` renvoie la valeur 2.
- L'appel `applique_transition(a,1,'b')` renvoie la valeur 1.
- L'appel `applique_transition(a,2,'b')` renvoie la valeur 1.

- L'appel `applique_transition(a,3,'b')` renvoie la valeur 3.

Exercice 6

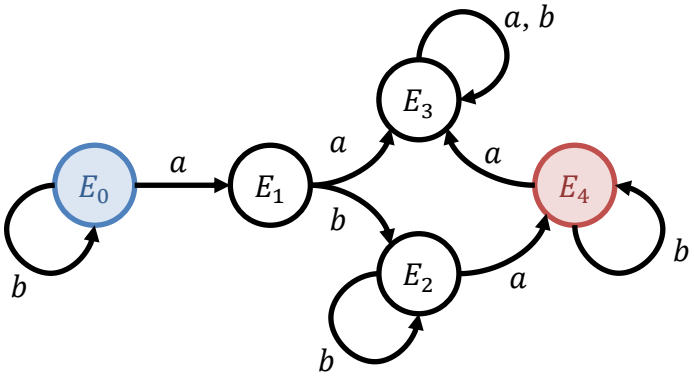
Écrivez une fonction `void teste_mot(t_automate a)` qui réalise les opérations suivantes :

1. Afficher l'automate reçu en paramètre ;
2. Demander à l'utilisateur de saisir une chaine de caractères contenant seulement des lettres de l'alphabet ;
3. Tester si le mot représenté par cette chaine de caractères est accepté ou pas par l'automate.
4. Afficher le résultat de ce test.

Testez votre programme avec l'automate donné en exemple.

Exercice 7

Même chose avec l'automate suivant :



Testez l'automate avec les mots suivants :

- *aabbabb*
- *aabb*
- *abb*
- *abab*
- *ababa*
- *abbabb*
- *babbabb*
- *babba*

Quels sont les mots reconnus par cet automate ?

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 27 | championnat de football

Présentation

Le but de ce TP est d'approfondir des notions déjà vues lors des TP précédents : types structurés, types énumérés, passage par adresse. On veut représenter le championnat de football turc de première division (appelé Süper Lig) en utilisant des types personnalisés.

1 Équipes

On veut tout d'abord représenter les équipes de Süper Lig, dont voici la liste complète :

Code	Nom court	Ville	Nom complet
akh	Akhisar	Akhisar	Akhisar Belediye Gençlik ve Spor Kulübü
ant	Antalyaspor	Antalya	Medical Park Antalyaspor Kulübü
bes	Beşiktaş	İstanbul	Beşiktaş Jimnastik Kulübü
bur	Bursaspor	Bursa	Bursaspor Kulübü Derneği
riz	Rizespor	Rize	Çaykur Rizespor Kulübü
ela	Elazığspor	Elazığ	Sanica Boru Elazığspor Kulübü
esk	Eskişehirspor	Eskişehir	Eskişehirspor Kulübü
fen	Fenerbahçe	İstanbul	Fenerbahçe Spor Kulübü
gal	Galatasaray	İstanbul	Galatasaray Spor Kulübü
gaz	Gaziantepspor	Gaziantep	Gaziantep Spor Kulübü
ank	Gençlerbirliği	Ankara	Gençlerbirliği Spor Kulübü
kar	Karabükspor	Karabük	Kardemir Demir Çelik Karabükspor
kas	Kasımpaşa	İstanbul	Kasımpaşa Spor Kulübü
erc	Erciyesspor	Kayseri	Kayseri Erciyes Spor Kulübü
kay	Kayserispor	Kayseri	Kayserispor
kon	Konyaspor	Konya	Torku Konyaspor Kulübü
siv	Sivasspor	Sivas	Sivasspor Kulübü
tra	Trabzonspor	Trabzon	Trabzonspor A.Ş.

Exercice 1

Définissez un type énuméré `t_code` permettant de représenter les noms des équipes, en utilisant le code unique associé à chacune d'elle dans la table ci-dessus.

Exercice 2

Définissez un type structure `t_equipe` permettant de stocker, pour chaque équipe : son code (`code`), son nom court (`nom`) et sa ville (`ville`). Le code est de type `t_code`, tandis que les autres champs sont de type chaîne de caractère.

Exercice 3

Déclarez de façon globale un tableau `equipes` de type `t_equipe`, permettant de stocker les données décrivant toutes les équipes de Süper Lig. Vous devez initialiser ce tableau lors de sa déclaration, en utilisant les données contenues dans la table ci-dessus.

Exercice 4

Écrivez une fonction `void affiche_nom(t_code code)` qui reçoit le code d'une équipe, et affiche le nom associé à ce code. Bien entendu, il faut utiliser le code en tant qu'index dans le tableau `equipes`.

2 Rencontres

Exercice 5

On veut maintenant représenter le résultat des rencontres entre les différentes équipes. Définissez un type structure `t_rencontre` qui possède les champs suivant : code de l'équipe jouant à domicile (`code_local`), code de l'équipe qui est reçue (`code_visiteur`), nombre de buts marqués par la première (`buts_local`) et par la seconde (`buts_visiteur`).

Exercice 6

Écrivez une fonction `void affiche_rencontre(t_rencontre rencontre)` qui affiche le résultat de la rencontre reçue en paramètre, exactement comme dans l'exemple ci-dessous.

exemple : affichage de la rencontre à l'issue de laquelle Antalya a battu Ganziantep 2-1

```
Antalyaspor 2 - 1 Ganziantepspor
```

Exercice 7

Écrivez une fonction `void initialise_score(t_rencontre* rencontre)` qui initialise aléatoirement le score de la rencontre passée en paramètre. Attention : les codes des équipes sont déjà initialisés, ici on ne s'occupe que du score.

Remarque : pour tirer un entier au hasard entre 0 et 5, utilisez le code source suivant (qui sera expliqué dans un TP ultérieur). Vous avez besoin d'inclure la bibliothèque `time.h`.

- Placez la ligne suivante au tout début de votre fonction `main` :

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

- Utilisez l'instruction suivante à chaque fois que vous avez besoin de tirer une valeur `n` au hasard :

```
int n = rand()%6;
```

3 Championnat

On veut maintenant représenter toutes les rencontres du championnat. On va pour cela utiliser une matrice de type `t_rencontre` et de dimension $N \times N$, où `N` est une constante dont la valeur correspond au nombre d'équipes. L'élément de cette matrice situé sur la ligne i et la colonne j correspond à la rencontre lors de laquelle l'équipe i a reçu l'équipe j (où i et j sont interprétés comme les valeurs numériques des codes de ces équipes).

Remarque : bien entendu, la diagonale de cette matrice ne sera pas utilisée, puisqu'une équipe ne peut pas jouer contre elle-même.

Exercice 8

Écrivez une fonction `void initialise_championnat(t_rencontre championnat[N][N])` qui reçoit une matrice représentant un championnat, et qui initialise aléatoirement toutes les rencontres de ce championnat (en utilisant `initialise_score`).

Exercice 9

Écrivez une fonction `void affiche_championnat(t_rencontre championnat[N][N])` qui reçoit une matrice représentant un championnat (déjà initialisé), et qui affiche tous ces résultats de façon compacte, comme dans l'exemple ci-dessous.

exemple :

```
             Akhisar   Antalyaspor  Besiktas  Bursaspor Rizespor  ...
Akhisar      -         1-0          0-2       0-1       1-2       ...
Antalyaspor  1-3       -            0-4       0-0       1-1       ...
Besiktas     2-1       1-3          -         1-1       0-0       ...
Bursaspor    1-0       2-0          2-1       1-2       4-4       ...
Rizespor     5-0       0-5          5-0       0-5       5-0       ...
...
```

4 Classement

Exercice 10

Pour finir, on veut savoir quelle équipe a gagné le championnat. On se propose de donner 3 points en cas de victoire, 1 point en cas d'égalité, et 0 points en cas de défaite. On utilise un tableau d'entiers de taille `N` pour représenter les points marqués par chaque équipe.

Écrivez une fonction `void raffraichis_points(int points[N], t_rencontre rencontre)` qui met à jour les points contenus dans le tableau `points`, en fonction du résultat de la rencontre `rencontre` passée en paramètre. Vous devez appliquer la règle indiquée ci-dessus.

Exercice 11

En cas d'égalité de points entre plusieurs équipes à la fin du championnat, on avantage celle qui a la meilleure différence de buts. On a donc besoin de savoir le nombre de buts marqués et encaissés par chaque équipe. Là encore, on les représentera avec des tableaux d'entiers de taille `N`.

Écrivez une fonction `void raffraichis_buts(int pour[N], int contre[N], t_rencontre rencontre)` qui met à jour les buts marqués (tableau pour) et encaissés (tableau contre) lors de la rencontre `rencontre` passée en paramètre.

Exercice 12

Écrivez une fonction `void calcule_resultats(int points[N], int pour[N], int contre[N], t_rencontre championnat[N][N])` qui calcule les points, but marqués et buts encaissés pour toutes les rencontres du championnat passé en paramètre. Les résultats de ces calculs sont stockés dans les 3 tableaux passés en paramètres (`points`, `pour`, `contre`).

Remarque : pensez à initialiser les trois tableaux, avant de commencer les calculs.

Exercice 13

Écrivez une fonction `int compare_equipes(t_code eq1, t_code eq2, int points[N], int pour[N], int contre[N])` qui compare deux équipes `eq1` et `eq2` en utilisant leurs résultats passés en paramètres. La fonction doit renvoyer une valeur négative si `eq1` est classée avant `eq2`, la valeur zéro si les deux équipes ont le même classement, et une valeur positive si `eq1` est classée après `eq2`.

Pour déterminer le classement relatif de deux équipes, on utilise les critères précédemment définis. Par ordre décroissant de priorité, ces critères sont :

- Plus grande nombre de *points* marqués ;
- Plus grande *différence* de buts ;
- Plus grand nombre de *buts* marqués.

Exercice 14

On représente le classement final du championnat à l'aide d'un tableau de type `t_code` et de taille `N` : le premier élément contient le code de l'équipe championne, le deuxième contient celui de l'équipe vice-championne, etc.

Écrivez une fonction `void ordonne_equipes(int points[N], int pour[N], int contre[N], t_code classement[N])` qui calcule ce classement final. Votre fonction doit utiliser la fonction `compare_equipes` pour déterminer la position relative de deux équipes.

Utilisez l'algorithme suivant (pas très efficace, mais simple à implémenter) :

- On place des valeurs négatives dans toutes les cases du tableau `classement`.
- Puis, pour chaque équipe *e* :
  - On compare l'équipe à toutes les autres équipes du championnat, en comptant le nombre *k* d'équipes qui sont classées *avant e* : la position de *e* dans `classement` correspond alors à cette valeur *k*.
  - Si `classement[k]` est une valeur négative, alors on place le code de *e* dans cette case du tableau.
  - Sinon, cela veut dire qu'il y a une égalité : une autre équipe est déjà classée en position *k*. On doit alors placer le code *e* dans la prochaine case contenant une valeur négative. Attention : cette case n'est pas forcément la $k+1$, car il est possible que la situation d'égalité concerne plus de 2 équipes.

Exercice 15

Enfin, écrivez une fonction `void affiche_classement(int points[N], int pour[N], int contre[N], t_code classement[N])` qui affiche à l'écran le résultat du championnat, exactement comme indiqué dans l'exemple ci-dessous : classement, points, buts marqués, but encaissés, différence de buts.

exemple : classement purement *aléatoire* !

```
    Equipe          Points  Pour   Contre  Difference
 1.Antalyaspor      60      80     61       19
 2.Rizespor         57      95     76       19
 3.Kayserispor      57      75     65       10
 4.Bursaspor        57      96     87        9
 5.Elazigspor       53      73     60       13
 6.Kasimpasa        52      88     73       15
 7.Akhisar          51      89     86        3
 8.Fenerbahce       50      87     68       19
 9.Gaziantepspor    50      87     80        7
10.Galatasaray      50      86     86        0
11.Besiktas         48      84     90       -6
12.Trabzonspor      47      72     87      -15
13.Erciyesspor      45      87     98      -11
14.Sivasspor        43      69     83      -14
15.Eskisehirspor    42      76     87      -11
16.Genclerbirligi   41      66     85      -19
17.Konyaspor        38      79     87       -8
18.Karabukspor      26      57     87      -30
```

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 28 | floutage d'une image

Présentation

Le but de ce TP est d'appliquer un traitement permettant de rendre une image floue. La manipulation de l'image se fera grâce à la SDL, donc votre projet doit être configuré de manière à pouvoir utiliser cette bibliothèque. Les fonctions que vous écrirez devront constituer une nouvelle bibliothèque appelée `floutage`.

1 Principe

On veut effectuer un traitement sur une image afin de la rendre *floue*. Ce type de traitement peut être utilisé, par exemple, pour atténuer le bruit présent dans une photo. L'approche la plus simple consiste à remplacer, pour chaque pixel, les trois composantes de sa couleur (rouge, vert, bleu) par une somme pondérée des composantes des pixels présents dans son voisinage.

Nous allons représenter cette somme sous la forme d'une matrice carrée C de dimensions $n \times n$, où n est impair, appelée *matrice de convolution*. L'élément central représente le pixel que l'on veut flouter. Considérons par exemple une matrice 3×3, cet élément occupe alors la position c_{11} et on a :

$$C = \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix}$$

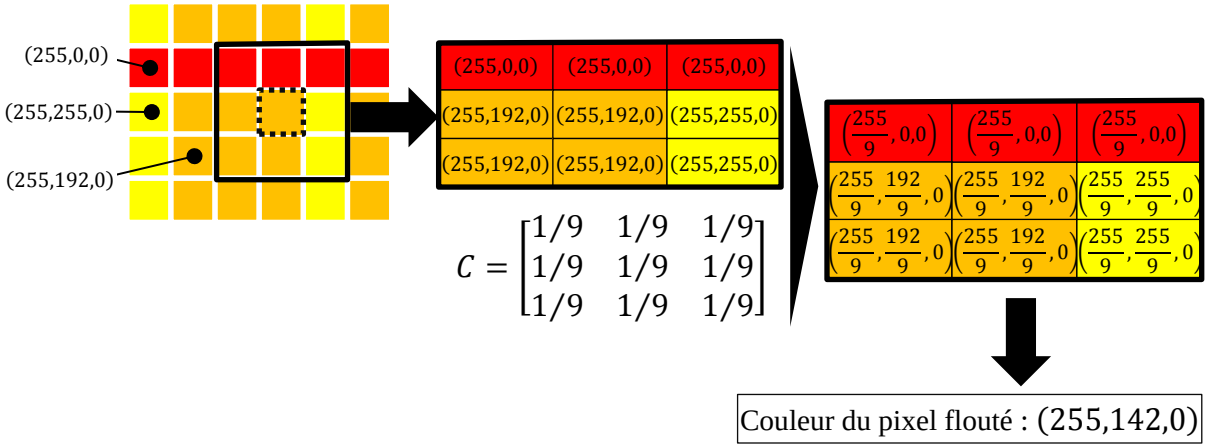
Chaque élément c_{ij} de la matrice correspond à un nombre réel, utilisé comme un poids dans la somme pondérée mentionnée ci-dessus. Ces valeurs doivent être normalisées, de manière à avoir $\sum_{i,j} c_{ij} = 1$.

Soit r_{ij} la composante rouge originale du pixel correspondant à la position (i, j) dans la matrice. Soit r la nouvelle valeur de cette composante après le floutage, *pour le pixel central*. Notez que si les composantes sont codées sur 8 bits, alors toutes ces valeurs sont des entiers compris entre 0 et 255. On obtient r avec le calcul suivant, et le même principe s'applique aux deux autres composantes (vert et bleu) :

$$r = \sum_{i,j} c_{ij} r_{ij}$$

exemple : dans l'exemple ci-dessous, la figure de gauche représente une partie d'une image, chaque carré correspondant à un pixel. On veut traiter le pixel encadré en pointillés. La zone encadrée en traits pleins représente le voisinage du pixel pour $n = 9$.

Dans la figure du milieu, on a les composantes des couleurs de chaque pixel de ce voisinage. Pour la matrice C, on a choisi le cas le plus simple, où $c_{ij} = 1/9$ pour tous les éléments. La figure de droite représente les composantes des couleurs obtenues après avoir appliqué C. La couleur finale du pixel flouté est obtenue en calculant faisant la somme, pour chaque composante, des valeurs pondérées.



2 Implémentation basique

Exercice 1

Créez une bibliothèque `floutage` qui utilise `graphisme.h`. Créez un fichier `main.c` qui utilise `floutage`. Les fonctions de ce TP seront à définir dans la bibliothèque `floutage`.

Exercice 2

Dans notre implémentation, on veut faire varier la taille du voisinage. Pour cette raison, les différentes matrices manipulées (telles que C) ne peuvent pas être passées sous la forme classique d'un paramètre `m[N][N]` (puisqu'on aurait besoin de connaître la taille). Nous allons donc utiliser à la place des pointeurs `*m`, qui pointeront sur des tableaux à deux dimensions. Comme nous l'avons vu en cours, l'accès à l'élément m_{ij} se fera donc grâce à l'expression `*(m+n*i+j)`.

Dans `floutage`, écrivez une fonction `void initialise_matrice_uniforme(float *c, int n)` qui reçoit une matrice `c` de taille $n \times n$ et l'initialise de façon *uniforme*. Autrement dit, chaque élément de la matrice doit avoir la même valeur, et leur somme doit s'élever à 1. Donc, l'initialisation doit utiliser la valeur $1/n^2$.

Remarque : attention aux conversions implicites !

exemple : pour `n=3`, la matrice obtenue doit être

```
0.111111 0.111111 0.111111
0.111111 0.111111 0.111111
0.111111 0.111111 0.111111
```

Exercice 3

Dans `floutage`, écrivez une fonction `void calcule_voisinage(SDL_Surface* image, int x, int y, Uint32* voisinage, int n)` qui calcule le voisinage du pixel de coordonnées (x, y) dans l'image passée en paramètre. Les couleurs des pixels du voisinage sont renvoyés à travers la matrice `voisinage` de taille $n \times n$. Cette matrice sera manipulée comme expliqué ci-dessus.

Testez votre fonction sur le pixel central de l'image contenue dans l'album fourni avec ce sujet.

exemple : pour le pixel central et `n=3`, vous devez obtenir le voisinage suivant (chaque valeur représentant la couleur d'un pixel du voisinage, sous forme de `Uint32`) :

```
14594165 14528372 14068078
14659701 14791801 13739370
13870442 14659958 14265199
```

Rappel : les dimensions d'une variable de type `SDL_Surface` (exprimées en pixels) peuvent être obtenues grâces aux champs `w` (largeur) et `h` (hauteur).

Exercice 4

Dans `floutage`, écrivez une fonction `Uint32 calcule_couleur(SDL_Surface* image, int x, int y, float *c, int n)`. Elle reçoit une image et doit calculer la nouvelle couleur du pixel de coordonnées (x, y) dans le but de flouter l'image. Cette nouvelle couleur sera obtenue en calculant la somme pondérée décrite dans la première section de ce sujet. La couleur obtenue sera ensuite renvoyée par la fonction sous la forme d'une valeur de type `Uint32`. Le paramètre `c` correspond à la matrice C, de taille $n \times n$, à utiliser lors du calcul de la couleur.

Remarque : votre fonction doit utiliser `calcule_voisinage`.

Testez votre fonction en utilisant une matrice 3×3 dont tous les éléments valent $1/9$, comme dans l'exemple de la première section de ce sujet. Initialisez votre matrice de convolution avec la fonction `initialise_matrice_uniforme`. Comme à l'exercice précédent, traitez le pixel central de l'image.

exemple : pour le pixel central, vous devez obtenir la valeur 14330993.

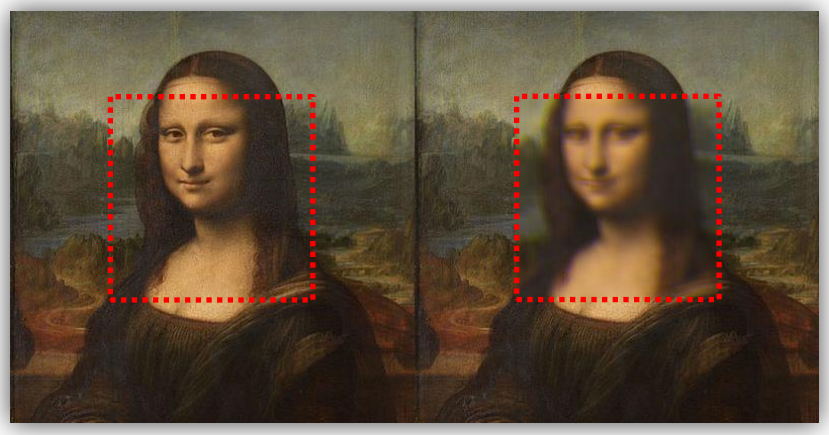
Exercice 5

Dans `floutage`, écrivez une fonction `SDL_Surface* floute_rectangle(SDL_Surface* source, int x, int y, int l, int h, float *c, int n)`. Elle reçoit une image originale `source`, et doit flouter ses pixels situés dans la zone délimitée par le rectangle d'angle supérieur gauche (x, y), de largeur `l` et de hauteur `h` pixels. Les nouveaux pixels doivent être dessinés dans une nouvelle image, dont l'adresse est le résultat de la fonction. La matrice de convolution est représentée par le paramètre `c`, et sa taille est `n`, comme pour la fonction précédente. Votre fonction doit utiliser `calcule_couleur`.

Remarque : avant de commencer à flouter, vous devez faire une copie de l'image `source`. Vous ne devez pas modifier l'image `source`, seulement l'image `cible`.

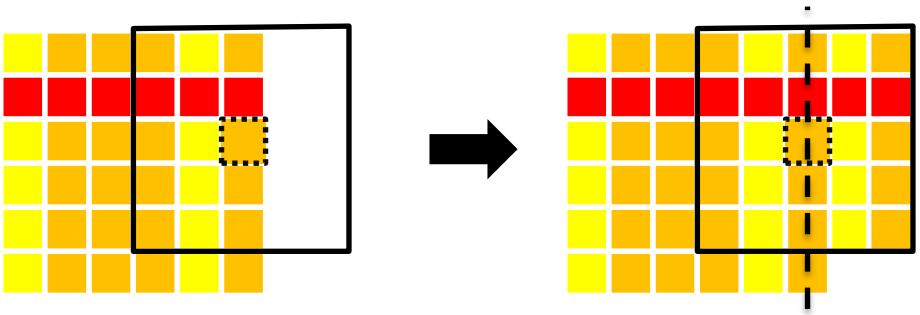
Testez votre fonction sur le rectangle correspondant à `x=50`, `y=50`, `l=50`, et `h=50`. Affichez l'image originale à gauche de l'écran, et l'image floutée à droite. Utilisez un voisinage de taille `n=7`.

exemple : l'image de gauche est l'image originale, celle de droite est le résultat du floutage pour les paramètres indiqués ci-dessus.



3 Gestion des bords

Les valeurs des paramètres utilisés jusqu'ici ont masqué un problème qui se pose quand un pixel se trouve trop près du bord de l'image : son voisinage n'est pas complètement défini. Dans ce cas-là, une méthode simple consiste à dupliquer les parties existantes du voisinage pour le compléter artificiellement. On peut procéder par symétrie axiale, de la manière suivante :



Dans l'exemple ci-dessus, on utilise les pixels de la partie gauche du voisinage pour remplacer les pixels manquants dans la partie droite. Si jamais le pixel concerné se trouve dans un coin de l'image, il suffit de procéder par symétrie centrale.

Exercice 6

Dans `floutage`, écrivez une fonction `void calcule_voisinage2(SDL_Surface* image, int x, int y, Uint32* voisinage, int n)` qui calcule le voisinage du pixel de coordonnées (x, y) dans l'image passée en paramètre. Il s'agit d'une version améliorée de `calcule_voisinage`, qui est capable de traiter de compléter le voisinage du pixel si celui-ci est incomplet.

Exercice 7

Modifiez la fonction `calcule_couleur`, de manière à ce que celle-ci utilise `calcule_voisinage2` au lieu de `calcule_voisinage`. Écrivez une fonction `SDL_Surface* floute_image(SDL_Surface* source, float *c, int n)` qui utilise `floute_rectangle` pour flouter l'intégralité d'une image.

Testez votre fonction sur la même image, avec différentes tailles de voisinage (pensez à utiliser des valeurs impaires !).

exemples : images obtenues pour `n=0`, `3`, `7`, `11`, `19` :

Remarque : vous noterez que le temps de calcul augmente rapidement avec n.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 29 | flous avancés

Présentation

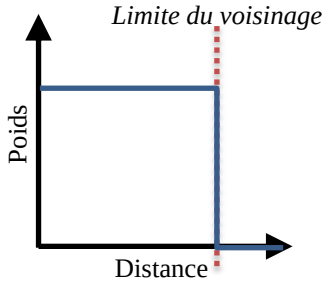
Ce TP est la suite de celui portant sur le floutage d'une image. Il est nécessaire d'avoir d'abord effectué ce TP-là avant de faire ce TP-ci. Certaines des fonctions développées lors du TP précédent sont fournies avec ce sujet, sous la forme de la bibliothèque `floutage.h`, qui sera complétée au cours du présent TP.

1 Flou gaussien

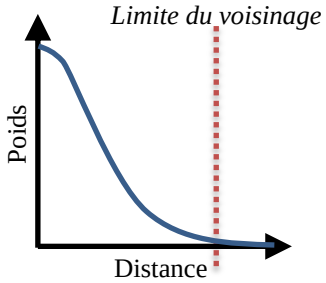
La méthode de floutage étudiée lors du TP précédent reposait sur l'utilisation d'une matrice de convolution uniforme, i.e. contenant des poids tous égaux. Mais on peut aussi utiliser des poids différents. Le *flou gaussien* repose ainsi sur l'idée que le poids des pixels proches du pixel considéré doit être plus élevé que le poids des pixels éloignés. Ce flou tire son nom du fait qu'on se base sur la loi de Gauss pour calculer le poids d'un pixel du voisinage en fonction de sa distance au pixel considéré :

$$c_{ij} = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(i-n/2)^2}{2\sigma^2}} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(j-n/2)^2}{2\sigma^2}} = \frac{1}{\sigma^2 2\pi} e^{-\frac{(i-n/2)^2+(j-n/2)^2}{2\sigma^2}}$$

La variable σ représente la dispersion de la loi de Gauss : plus elle est faible et plus la distance influence le poids.



Poids uniformes



Poids gaussiens

On peut comparer visuellement les poids uniformes et ceux basés sur la loi de Gauss : les graphiques ci-dessus montrent comment le poids évolue en fonction de la distance entre le pixel du voisinage et le pixel considéré.

Exercice 1

Dans `floutage`, écrivez une fonction `void normalise_matrice(float *c, int n)` qui reçoit une matrice de convolution et la normalise, de manière à ce que la somme de ses valeurs soit égale à 1.

exemple : la matrice $\begin{bmatrix}1 & 2 & 1\\2 & 3 & 2\\1 & 2 & 1\end{bmatrix}$ devient $\begin{bmatrix}0{,}07 & 0{,}13 & 0{,}07\\0{,}13 & 0.2 & 0{,}13\\0{,}07 & 0{,}13 & 0{,}07\end{bmatrix}$.

Exercice 2

Écrivez une fonction `void initialise_matrice_gaussienne(float *c, int n, float sigma)` qui initialise une matrice de convolution en utilisant la méthode décrite ci-dessus.

Testez votre fonction en appliquant le flou uniforme et le flou gaussien à l'image, avec la même valeur de `n`, et en affichant simultanément les deux images obtenues.

Remarque : déclarez des constante `PI` et `EXP` pour représenter π (3,14159) et e (2,71828).

exemple : pour `n=7` et `sigma=3`, on obtient les valeurs suivantes :

```
0.011297 0.014915 0.017619 0.018626 0.017619 0.014915 0.011297
0.014915 0.019690 0.023261 0.024590 0.023261 0.019690 0.014915
0.017619 0.023261 0.027480 0.029050 0.027480 0.023261 0.017619
0.018626 0.024590 0.029050 0.030709 0.029050 0.024590 0.018626
0.017619 0.023261 0.027480 0.029050 0.027480 0.023261 0.017619
0.014915 0.019690 0.023261 0.024590 0.023261 0.019690 0.014915
0.011297 0.014915 0.017619 0.018626 0.017619 0.014915 0.011297
```

Exercice 3

Depuis la fonction `main`, expérimentez en appliquant le flou gaussien avec un voisinage assez grand et différentes valeurs de σ.

exemples : $n = 13$ et $\sigma = 0{,}1$; 2 ; 3 ; 4 et 5 :

Appliquez aussi les deux types de flou à l'image, en utilisant les paramètres $n = 13$ et $\sigma = 5$. Notez que le flou obtenu avec la méthode gaussienne (à droite) est plus lissé :

2 Flou cinétique

Le flou cinétique consiste à créer un effet ressemblant à celui obtenu quand un objet en déplacement est pris en photo. Cet effet peut être produit en utilisant une matrice de convolution contenant simplement une ligne. Par exemple, la matrice suivante produit un flou simulant un déplacement diagonal :

$$C = \begin{bmatrix} 1/3 & 0 & 0 \\ 0 & 1/3 & 0 \\ 0 & 0 & 1/3 \end{bmatrix}$$

Exercice 4

Toujours dans la bibliothèque `floutage`, écrivez une fonction `void initialise_matrice_cinetique(float *c, int n, float coef_dir, int multiplicite)` qui initialise une matrice de convolution avec des valeurs permettant d'obtenir un flou cinétique.

Le paramètre `coef_dir` contrôle le coefficient directeur de la droite principale.

exemples : matrices obtenues pour `multiplicite=1` et `coef_dir=0.5`, `1` et `2` :

```
0.0 0.0 0.0 0.0 0.0
0.2 0.0 0.0 0.0 0.0
0.0 0.2 0.2 0.2 0.0
0.0 0.0 0.0 0.0 0.2
0.0 0.0 0.0 0.0 0.0
```

```
0.2 0.0 0.0 0.0 0.0
0.0 0.2 0.0 0.0 0.0
0.0 0.0 0.2 0.0 0.0
0.0 0.0 0.0 0.2 0.0
0.0 0.0 0.0 0.0 0.2
```

```
0.0 0.2 0.0 0.0 0.0
0.0 0.0 0.2 0.0 0.0
0.0 0.0 0.2 0.0 0.0
0.0 0.0 0.2 0.0 0.0
0.0 0.0 0.0 0.2 0.0
```

Remarque : n'oubliez pas qu'avec la SDL, l'axe des ordonnées est inversé par rapport à la convention utilisée en mathématiques. De plus, pensez à distinguer les deux cas suivants :

- Coefficient directeur plutôt vertical (i.e. de valeur absolue supérieure à 1) ;
- Coefficient directeur plutôt horizontal (i.e. de valeur absolue comprise entre 0 et 1).

Le paramètre `multiplicite` indique si on veut dédoubler la droite principale, avec des droites parallèles de poids inférieur. Le fait de dédoubler les droites permet de lisser plus ou moins le floutage. On procède de la façon suivante :

- La droite principale (représentée en bleu dans l'exemple) a un poids de valeur `multiplicite` ;
- Les droites qui sont juste au-dessus et juste au-dessous (en orange) ont poids de valeur `multiplicite-1` ;
- Les droites qui sont au-dessus/dessous de celles-ci (en vert) ont un poids de valeur `multiplicite-2` ;
- Etc.

Une fois que les poids ont été calculés sous forme de valeurs entières comme expliqué ci-dessus, il suffit de normaliser la matrice pour respecter la contrainte qui veut que la somme des poids est 1. On obtient alors des valeurs de la forme de celles données en exemple ci-dessous.

exemples : matrices obtenues pour `coef_dir=0.5` et `multiplicite=1`, `2` et `3` :

```
0.00 0.00 0.00 0.00 0.00
0.20 0.00 0.00 0.00 0.00
0.00 0.20 0.20 0.20 0.00
0.00 0.00 0.00 0.00 0.20
0.00 0.00 0.00 0.00 0.00
```

```
0.05 0.00 0.00 0.00 0.00
0.10 0.05 0.05 0.05 0.00
0.05 0.10 0.10 0.10 0.05
0.00 0.05 0.05 0.05 0.10
0.00 0.00 0.00 0.00 0.05
```

```
0.05 0.02 0.02 0.02 0.00
0.07 0.05 0.05 0.05 0.02
0.05 0.07 0.07 0.07 0.05
0.02 0.05 0.05 0.05 0.07
0.00 0.02 0.02 0.02 0.05
```

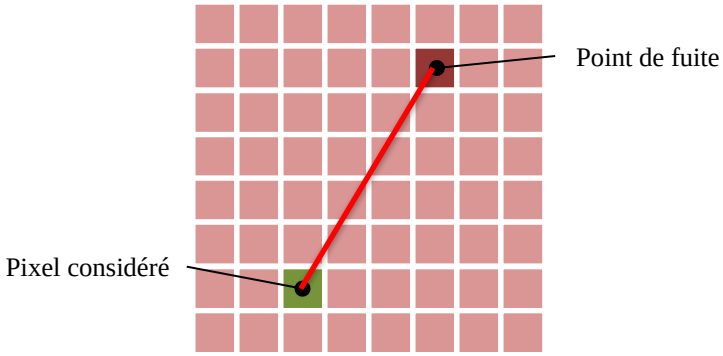
Exercice 5

Dans la fonction `main`, appliquez le flou cinétique en expérimentant avec les valeurs des paramètres, en particulier le coefficient directeur.

exemples : images obtenues avec des coefficient directeurs de `0`, `0.5`, `1`, `2` et `1000` :

3 Flou radial

Le flou radial consiste à effectuer un flou cinétique, mais en utilisant une matrice de convolution différente pour chaque pixel. On va d'abord sélectionner un pixel dans l'image qui servira de *point de fuite*. Puis, le coefficient directeur utilisé pour générer la matrice de convolution d'un pixel donné correspondra à celui de la droite passant par le point de fuite et le pixel concerné. Dans l'exemple ci-dessous, il s'agit de la droite représentée en rouge.



Exercice 6

Dans `floutage`, écrivez une fonction `float calcule_coefficient_directeur(int x1, int y1, int x2, int y2)` qui calcule le coefficient directeur de la droite reliant les points (x_1, y_1) et (x_2, y_2). En dehors du cas général, deux cas particuliers sont à considérer :

- Le cas où la droite est verticale (i.e. $x_1 = x_2$) : on renvoie alors la valeur maximale que le type `float` peut représenter.
- Le cas où les deux points sont confondus (i.e. $x_1 = x_2$ et $y_1 = y_2$) : on renvoie alors la valeur 0.

Remarque : la bibliothèque `float.h` contient la constante `FLT_MAX`, correspondant à la valeur maximale qu'on peut représenter avec un `float`.

exemples :

- Points (10,10) et (50,100) : 2,25 (cas général) ;
- Points (10,10) et (10,100) : `FLT_MAX` (droite verticale) ;
- Points (10,10) et (10,10) : 0 (même point) ;
- Points (10,10) et (100,10) : 0 (droite horizontale).

Exercice 7

Écrivez une fonction `SDL_Surface* floute_image_radial(SDL_Surface* source, int x, int y)` qui applique un flou radial à l'image passée en paramètre, en prenant comme point de fuite le pixel de coordonnées (x, y).

Remarque : lors de la génération des matrices de convolution, fixez les paramètres `multiplicite=1` et `n=11`.

Testez d'abord votre fonction en utilisant comme point de fuite le pixel situé entre les deux yeux de Mon Lisa, de coordonnées $(179,138)$. Expérimentez ensuite avec d'autres points de fuite.

exemples : à gauche, le point de fuite est celui indiqué ci-dessus ; à droite c'est le centre de l'image.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

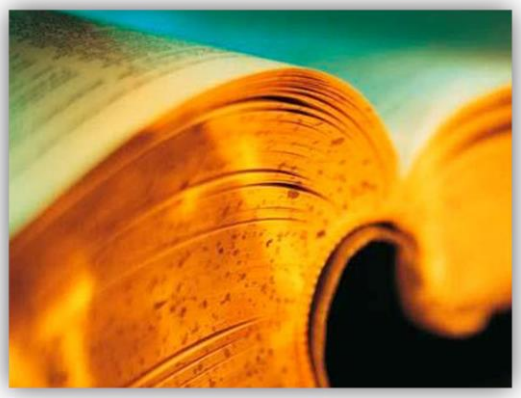
Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 30 | gestion d'un lexique

Présentation

Dans ce TP, on veut implémenter un certain nombre de fonctions permettant d'accéder à un lexique contenant uniquement des mots de 4 lettres. Vous aurez pour cela besoin du contenu de l'archive fournie avec ce sujet. Certains correspondent à la bibliothèque `chaine`, qui permet de manipuler de chaînes de caractères, et qui a été écrite lors d'un TP précédent. D'autres correspondent à une nouvelle bibliothèque `lexique`, que les exercices de ce TP vont permettre de compléter.



1 Définition du lexique

Le fichier `lexique.c` contient une variable *globale* `lexique` désignant un tableau de pointeurs sur des chaînes de caractères. Chacune de ces chaînes est un mot de 4 lettres. Le tableau contient `TAILLE_ LEXIQUE` mots (il y a des répétitions : certains mots sont contenus plusieurs fois). Les mots ne sont pas rangés dans l'ordre lexicographique.

Exercice 1

Donnez le type et la valeur des expressions suivantes :

- `*lexique`
- `lexique+1`
- `*(lexique+2)`
- `*(lexique+1)+1`
- `*(*(lexique+2)+2)`
- `*(*(lexique+3)+5)`

Exercice 2

Écrivez une fonction `affiche_lexique` qui affiche l'intégralité du lexique, en numérotant chaque mot, de la façon suivante :

```
lexique[0]="drap"
lexique[1]="nuee"
lexique[2]="agit"
lexique[3]="mais"
lexique[4]="krak"
...
```

Vous devez respecter exactement cet affichage, y compris les *guillemets* `""`. Cette fonction et toutes les suivantes doivent être placées dans la bibliothèque `lexique` et testées dans le `main`.

2 Recherche dans le lexique

Exercice 3

En utilisant les fonctions de la bibliothèque `chaine`, écrivez une fonction `int cherche_prefixe(char *prefixe)` capable de parcourir le lexique et d'afficher tous les mots qui sont préfixés par `prefixe`. La fonction renvoie un entier correspondant au nombre total de mots préfixés.

exemple : `cherche_prefixe("tr")` doit afficher exactement les lignes suivantes (y compris les guillemets `""` et l'alignement des numéros de ligne) :

```
la chaine "tr" est un prefixe des mots suivants :
  1.lexique[162]="trie"
  2.lexique[164]="tram"
  3.lexique[170]="trac"
  4.lexique[174]="tria"
  5.lexique[178]="trio"
  6.lexique[181]="troc"
  7.lexique[184]="trot"
  8.lexique[187]="trop"
  9.lexique[188]="trou"
 10.lexique[197]="truc"
 11.lexique[366]="tres"
La chaine "tr" est prefixe de 11 mots du lexique
```

Remarque : lors du test de votre fonction `cherche_prefixe`, l'affiche de la *dernière ligne* de l'exemple ci-dessus doit être réalisé dans la fonction `main`.

Exercice 4

Écrivez une fonction `int cherche_premier()` renvoyant la position dans lexique (i.e. l'index) du plus petit mot. Autrement dit : le premier mot du lexique dans l'ordre lexicographique.

exemple : la fonction doit renvoyer l'index 6, qui correspond au mot `lexique[6]="aaai"`.

Exercice 5

Écrivez une fonction `int cherche_premier_partiel(int debut)`, qui cherche elle aussi le mot le plus petit, mais seulement dans la partie du lexique comprise entre `debut` et `TAILLE_LEXIQUE` (et non plus dans le lexique complet).

exemple : l'appel `chercher_premier_partiel(7)` doit renvoyer l'index 930, qui correspond au mot `lexique[930]="abat"`.

3 Tri du lexique

Exercice 6

Écrivez la fonction `void permute_mots(int i, int j)` capable d'échanger deux mots du lexique désignés par leurs positions `i` et `j` dans `lexique`. L'échange doit être réalisé *sans recopier* aucune chaîne de caractères, simplement en manipulant les pointeurs.

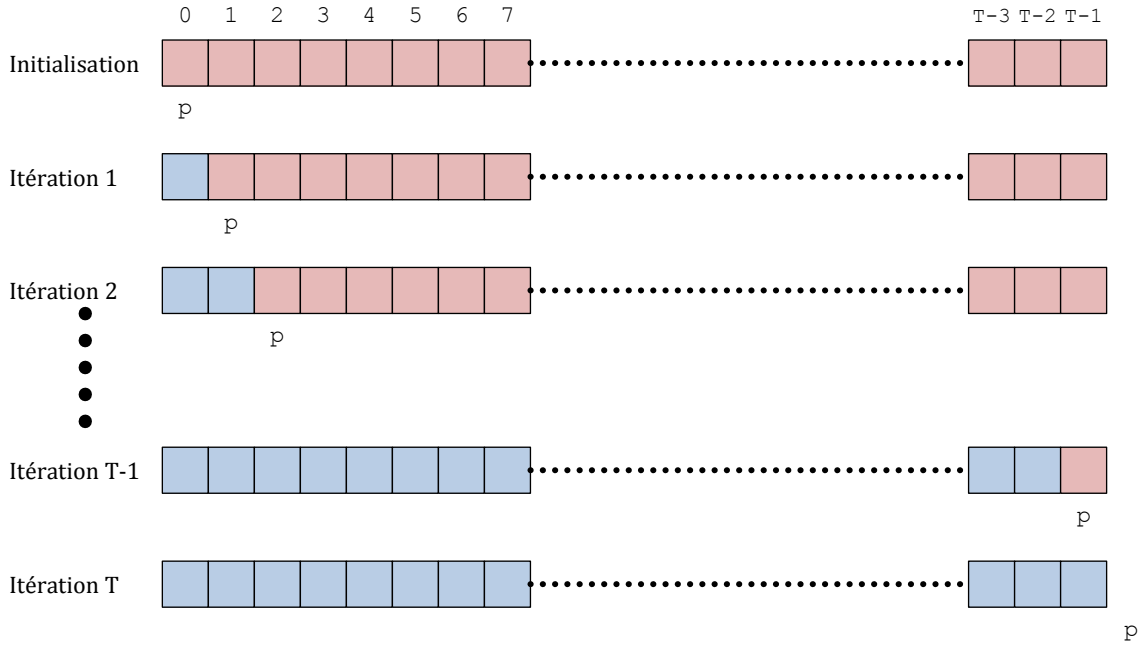
Exercice 7

Écrivez une fonction `trie_lexique` qui trie le lexique par ordre lexicographique en appliquant l'algorithme décrit ci-dessous.

À un moment donné de l'exécution, on considère qu'il y a deux parties dans le lexique :

- La partie située avant une position `p`, qui a déjà été triée ;
- La partie située sur et après cette position `p`, qui contient les mots pas encore triés.

Au début de l'algorithme, cette position `p` est 0, car aucun mot n'a encore été trié. À la fin de l'algorithme, la position `p` est `TAILLE_ LEXIQUE`, car tous les mots ont été triés. Le schéma ci-après illustre cette évolution : la partie bleue est la partie triée, la partie rouge est la partie pas encore triée.



L'algorithme lui-même est le suivant :

- On répète :
  - On détermine la position du plus petit mot `m1` du lexique (d'après l'ordre lexicographique) qui n'a *pas encore été traité* ;
  - On échange sa position avec le premier mot `m2` à ne *pas avoir été déjà traité*. Attention, ici premier signifie : d'indice minimum, donc le mot occupant la position `p` ;
  - On considère que `m1` *a été traité* (i.e. il est à sa position définitive dans le lexique trié) en incrémentant la position `p`.
- Jusqu'à ce qu'on ait traité tous les mots du lexique.

exemple : en affichant le lexique après l'avoir trié, on doit obtenir le résultat suivant :

```
lexique[0]="aaai"
lexique[1]="abat"
lexique[2]="abbe"
lexique[3]="abri"
...
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 31 | allocation dynamique

Présentation

Le but de ce TP est d'étudier comment la mémoire est organisée, et de manipuler les différentes fonctions standard permettant d'allouer/libérer la mémoire de façon dynamique. Une bibliothèque `util` est fournie pour l'un des exercices portant sur les chaînes de caractères (elle est nécessaire seulement si vous travaillez sour Linux, pas Windows).

1 Organisation de la mémoire

On reprend l'exercice du TP sur les variables, qui consistait à créer des variables locales et globales et à afficher leurs adresses afin de déterminer comment la mémoire est organisée pour un programme C.

Le résultat de l'exécution indiquait que les variables globales étaient créées en partant d'une certaine adresse, en incrémentant l'adresse pour chaque nouvelle variable créée, tandis que les variables locales étaient créées en partant d'une adresse inférieure, et en décrémentant l'adresse pour chaque nouvelle variable créée. Pour plus de détails, cf. le corrigé de ce TP.

Exercice 1

Copiez-collez le code source à partir de la correction du TP mentionné ci-dessus. Puis, adaptez-le pour étudier l'allocation dynamique :

- Dans la fonction `main`, déclarez 6 pointeurs sur des valeurs short notés `m`, `n`, `o`, `s`, `t`, et `u`.
- Avant l'appel à `fonction`, allouez dynamiquement une zone mémoire de taille 1 `short` pour `m`, `n` et `o`. Affichez ensuite les adresses de ces zones mémoire.
- Dans la fonction, déclarez 3 pointeurs `short*` : `p`, `q` et `r`, et allouez-leur dynamiquement une zone mémoire de taille 1 `short`. Affichez les adresses de ces zones mémoire.
- Dans la fonction `main`, après l'appel à `fonction`, allouez dynamiquement une zone mémoire de taille 1 `short` pour `s`, `t` et `u`. Affichez les adresses de ces zones mémoire.

Attention : vous devez garder l'affichage des adresses des variables locales et globales existantes.

Indiquez vos résultats et commentez-les. Que pouvez-vous en déduire sur la localisation du *tas* dans la mémoire ?

2 Tableaux d'entiers

Exercice 2

Écrivez une fonction `alloue_tableau1` qui crée un tableau de type `short` et qui l'initialise en affectant la valeur 1 à chaque élément. La taille du tableau (nombre d'éléments) et passée à la fonction sous la forme d'un paramètre `taille`. La zone de mémoire correspondant au tableau doit être allouée dynamiquement. L'adresse de cette zone est

renvoyée par la fonction grâce à l'instruction `return`. La fonction doit renvoyer `NULL` en cas d'erreur lors de l'allocation. Testez votre fonction depuis la fonction `main`, en affichant le contenu du tableau créé.

Remarque : pour l'affichage du tableau, vous pouvez définir une fonction `void affiche_tableau(int *tab, int taille)`, car elle sera réutilisée plus loin.

Exercice 3

Écrivez une fonction `alloue_tableau2` qui réalise la même tâche, mais cette fois l'adresse de la zone doit être passée via un paramètre de la fonction appelé `tab`. Là encore, testez votre fonction dans la fonction `main` en affichant le tableau obtenu.

3 Chaînes de caractères

Remarque : vous devez programmer toutes les manipulations de chaînes de caractères, et donc vous n'avez pas droit aux fonctions de `string.h`.

Exercice 4

Écrivez une fonction `saisis_chaine_tampon` qui saisit une chaine de caractères en utilisant un tampon. L'espace mémoire permettant de stocker la chaîne de caractères doit être alloué dynamiquement, et son adresse renvoyée via un paramètre.

L'algorithme est le suivant :

1) Un tableau de caractères `temp`, est créé de façon *statique*, localement à la fonction. On l'appelle le *tampon* (ou *buffer* en anglais).
2) La chaîne est saisie par l'utilisateur grâce à `scanf` ou `gets`, et est placée dans le tableau `temp`.
3) La longueur de la chaine est calculée.
4) Une zone mémoire de la taille adéquate est allouée dynamiquement.
5) Le contenu du tableau est copié dans cette zone de mémoire.
6) L'adresse de la zone mémoire est renvoyée via le paramètre.

Exercice 5

Écrivez une fonction `saisis_chaine_direct` qui effectue la même opération que `saisit_chaine_tampon`, mais cette fois sans utiliser de tampon (i.e. sans le tableau de caractères `temp` local à la fonction).

Pour cela, vous utiliserez la fonction `getch`, qui permet de saisir un caractère en mode non-bufférisé. La fonction `getch` s'utilise comme `getchar`, mais l'utilisateur n'a pas besoin de taper la touche *entrée* à la fin de la saisie. Son utilisation dépend de votre système d'exploitation :

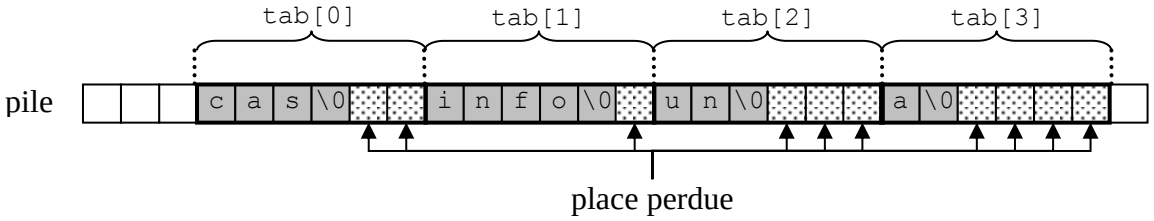
- Sous Windows, vous devez inclure la bibliothèque `conio.h` pour pouvoir utiliser `getch`, et le caractère correspondant à la touche entrée est `'\r'`.
- Sous Linux, vous devez utiliser la fonction `getch` définie dans la bibliothèque `util.h` donné avec le sujet, et le caractère représentant la touche entrée est `'\n'`.

4 Tableaux de pointeurs

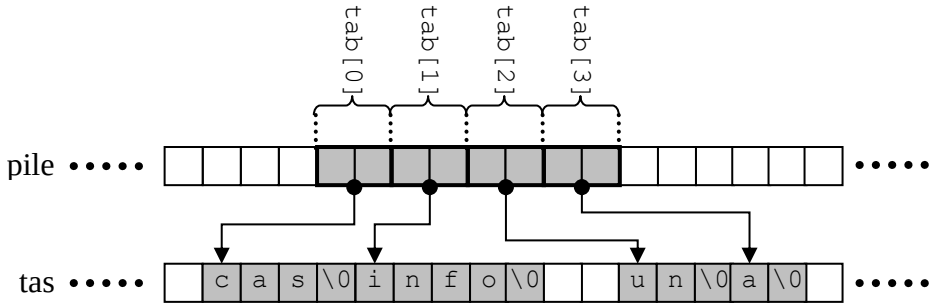
On veut manipuler un ensemble de mots. La méthode la plus simple est de les ranger dans un tableau. Comme un mot correspond à une chaîne de caractères (tableau de type `char`), on obtient finalement, pour représenter notre ensemble, un tableau de tableaux de caractères (tableau de type `char` à deux dimensions).

Le problème est que les mots n'ont pas forcément la même taille. Si on utilise des tableaux statiques pour représenter les mots, on perd de la place en mémoire.

exemple : `char tab[N][M]` pour *N=4* mots de taille *M=6*



Une solution possible consiste à utiliser un tableau de pointeurs. Chaque pointeur pointe vers un tableau de caractères alloué dynamiquement (et donc : sans perte de place).



Exercice 6

Déclarez une variable globale `mots` pouvant contenir `N` pointeurs vers des chaines de caractères. Écrivez ensuite une fonction `void remplis_mots()`, qui :

1) Utilise l'une des deux fonctions précédentes pour saisir la liste de mots et la stocker dans `mots`, comme décrit dans le schéma précédent ;
2) Afficher l'ensemble des mots contenu dans `mots`.

exemple : pour $N = 10$

```
Entrez le mot n.0 : arbre
Entrez le mot n.1 : rododendron
Entrez le mot n.2 : trompe
Entrez le mot n.3 : a
Entrez le mot n.4 : partie
Entrez le mot n.5 : train
Entrez le mot n.6 : balle
Entrez le mot n.7 : chien
Entrez le mot n.8 : anticonstitutionnellement
Entrez le mot n.9 : desoxyribonucleique
Mot n.0 : arbre
Mot n.1 : rododendron
Mot n.2 : trompe
Mot n.3 : a
Mot n.4 : partie
Mot n.5 : train
Mot n.6 : balle
Mot n.7 : chien
Mot n.8 : anticonstitutionnellement
Mot n.9 : desoxyribonucleique
```

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 32 | générateur pseudo-aléatoire

Présentation

Dans ce TP, on s'intéresse aux méthodes permettant à un ordinateur de générer des nombres aléatoires. Nous utiliserons des tableaux et les méthodes d'allocation dynamique vues en cours.

1 Méthode de Lehmer

Dans un grand nombre d'applications, on a besoin de pouvoir générer des nombres de façon aléatoire : calcul statistique, simulation physique, jeux vidéo, cryptographie, etc. Mais, par définition, le fonctionnement d'un ordinateur classique est déterministe : les mêmes causes entraînent toujours le même résultat. Cela signifie qu'il est absolument impossible de générer aléatoirement un nombre avec un ordinateur (sauf si on utilise un matériel spécifique).

Pour résoudre ce problème, différentes méthodes ont été développées pour simuler le hasard, et générer des nombres en essayant de reproduire le mieux possible les propriétés d'un tirage aléatoire. Les algorithmes qui utilisent ces méthodes sont appelés des générateurs *pseudo-aléatoires* de nombres.

Un générateur pseudo-aléatoire permet de produire une séquence de nombres contenus dans un intervalle donné, de manière à ce que chaque nombre soit apparemment indépendant du précédent (on ne peut pas facilement prévoir quel sera le prochain nombre de la séquence). De plus, le générateur est défini de manière à ce que cette séquence reproduise les propriétés d'une distribution de probabilités donnée.

La plupart des générateurs aléatoires simulent une distribution uniforme, c'est-à-dire une distribution pour laquelle toutes les valeurs de l'intervalle possèdent la même probabilité d'apparition. Par exemple, pour un intervalle entier $[0;9]$, chaque valeur $0,\ldots,9$ aura une probabilité $p = 1/10$ d'être tirée au sort.

Un générateur aléatoire est généralement construit à partir d'une suite mathématique (u) possédant certaines propriétés. Dans ce TP, on s'intéresse à la méthode du générateur congruentiel linéaire, développée par Derrick Lehmer en 1948. Elle utilise la suite suivante :

$$u_{i+1} = (au_i + c) \bmod m$$

La tâche consistant à déterminer les valeurs des paramètres a, c et m sort du cadre de ce TP. Nous utiliserons les valeurs $a = 137$, $c = 187$ et $m = 256$, qui fonctionnent relativement bien pour générer des valeurs dans $[0;255]$. Le terme initial u_0 est appelé la *graine* du générateur pseudo-aléatoire. Pour une suite donnée et des paramètres fixés, le fait d'utiliser la même graine produira *toujours* la même séquence de nombres. Cette propriété (parmi d'autres) différencie un générateur *pseudo*-aléatoire d'un générateur aléatoire.

2 Implémentation

Exercice 1

Créez la bibliothèque `alea` (i.e. créez les fichiers nécessaires, en leur donnant des noms appropriés). Dans cette bibliothèque, définissez la fonction `unsigned char genere_nombre_lehmer()` qui calcule le terme suivant dans la suite décrite ci-dessus, et renvoie sa valeur. Vous devez définir toutes les constantes/macros/variables nécessaires, en utilisant les classes de mémorisation appropriées. On supposera que la graine u_0 vaut 0.

Créez un fichier `main.c` qui contiendra la fonction `main`. Dans cette fonction `main`, ajoutez les instructions nécessaires pour tester `genere_nombre_lehmer` : effectuez 10 appels de la fonction et affichez les valeurs obtenues.

exemple : le test doit provoquer exactement l'affichage suivant :

```
graine: 0
nombres: 187 206 249 252 151 138 149 120 243 198
```

Exercice 2

La graine est initialisée de façon *statique*, et par conséquent la fonction `genere_nombre_lehmer` va produire *exactement* la même séquence de nombres à chaque exécution. Pour éviter cela, il faut initialiser la graine *dynamiquement*. Le plus simple est d'utiliser l'heure actuelle : cette valeur sera différente à chaque exécution (car le temps s'écoule), et donc la séquence sera différente à chaque exécution.

La fonction `time_t time(time_t* t)` contenue dans la librairie standard `time.h` prend en paramètre l'adresse d'une variable de type `time_t`, et lui affecte la date et l'heure actuelles exprimées en secondes. Le type `time_t` est aussi défini dans `time.h` et permet de représenter des données temporelles. La fonction renvoie -1 en cas d'erreur, et sinon elle renvoie la date et l'heure actuelles exprimées en secondes (i.e. la même valeur que celle placée dans `t`).

Dans la bibliothèque `alea`, écrivez une fonction `unsigned char init_graine_lehmer()` qui initialise la graine en utilisant la fonction `time`. Votre fonction doit aussi renvoyer la valeur de la graine par valeur. De plus, vous devez tester les éventuels cas d'erreurs (et cette remarque vaut pour toutes vos fonctions en général).

Testez votre fonction `init_graine_lehmer` depuis la fonction `main`, en l'appelant plusieurs fois et en affichant la valeur de la graine à chaque fois. Testez ensuite son effet sur `genere_nombre_lehmer` : exécutez plusieurs fois le test de cette fonction (défini pour l'exercice précédent) et vous devriez obtenir 10 nombres différents à chaque nouvelle exécution.

exemple : pour 3 exécutions, le test doit provoquer un affichage du type :

```
graine: 211
nombres: 166 145 84 175 98 45 208 11 158 73
graine: 226
nombres: 173 80 139 30 201 76 103 218 101 200
graine: 2
nombres: 205 112 171 62 233 108 135 250 133 232
```

Exercice 3

On veut générer `n` nombre entiers et les placer dans un tableau de dimension appropriée. Dans `alea`, écrivez une fonction `unsigned char* genere_tableau_lehmer(int n)` qui alloue dynamiquement la mémoire nécessaire, initialise le tableau aléatoirement et renvoie son adresse. Testez cette fonction dans la fonction `main` et affichez le contenu du tableau obtenu.

exemple : pour $n = 15$, le test doit provoquer un affichage du type :

```
graine: 23
```

```
nombres: [ 10 21 248 115 70 49 244 79 2 205 112 171 62 233 108 ]
```

3 Vérification

Exercice 4

On veut vérifier que les nombres générés suivent bien une distribution uniforme. Pour cela, on peut calculer certaines statistiques, comme la moyenne μ, l'écart-type σ et les valeurs minimales min et maximales max générées. Les valeurs attendues pour cette distribution sont respectivement :

- $\mu = \frac{(0+255)}{2} = 127{,}5$;
- $\sigma = (256^2 - 1)/12 \approx 73{,}9$;
- $min = 0$;
- $max = 255$.

Dans `alea`, écrivez une fonction `calcule_stats` qui calcule et renvoie ces statistiques pour un tableau passé en paramètre. À vous de déterminer les paramètres appropriés, sachant qu'aucun affichage ne doit être réalisé dans cette fonction.

Testez votre fonction depuis la fonction `main`, en affichant ces statistiques pour des tableaux de tailles $n = 10^1, 10^2, 10^3, 10^4$ et 10^5. Que remarquez-vous ?

exemple : le test doit provoquer un affichage du type suivant (attention : vous devez respecter cette mise en forme) :

```
graine: 157
n=    10: min= 56, max=192, moy=131.10, et=53.02
n=   100: min=  2, max=255, moy=117.18, et=77.46
n=  1000: min=  0, max=255, moy=127.08, et=73.64
n= 10000: min=  0, max=255, moy=127.54, et=73.92
n=100000: min=  0, max=255, moy=127.51, et=73.90
```

Remarque : pour calculer l'écart-type, vous aurez besoin des fonctions `pow(x,y)` et `sqrt(x)` contenues dans `math.h`, qui permettent respectivement de calculer x^y et $\sqrt{x}$. Pour utiliser cette bibliothèque sous Linux, il est nécessaire de configurer le linker en allant dans les propriétés du projet, puis *C/C++ Build* > *Settings* > *Tool Settings* > *gcc C linker* > *Libraries*, puis dans l'onglet *Libraries (-l)* ajoutez la librairie `m`.

Exercice 5

Ces statistiques ne permettent toutefois pas de caractériser complètement la distribution des valeurs générées. Pour plus de précision, on peut l'afficher directement sous la forme d'un histogramme.

Dans `alea`, écrivez une fonction `void calcule_repartition` qui reçoit un tableau et calcule la distribution des valeurs qu'il contient. Cette distribution doit prendre la forme d'un tableau `dist` de taille $m = 256$, dont l'élément `dist[i]` contient le nombre d'occurrences de la valeur `i`. Ainsi, `dist[58]` contient le nombre d'apparitions de la valeur 58 dans le tableau généré. Vous devez décider des paramètres nécessaires à votre fonction, sachant qu'elle doit allouer `dist` dynamiquement et qu'elle ne renvoie rien par `return`.

Testez votre fonction depuis la fonction `main` en générant un tableau de taille $n = 10$ puis en affichant le contenu de `dist` et en vérifiant manuellement que les occurrences correspondent.

exemple :

```
nombres: [ 105 236 7 122 4 104 99 182 33 100 ]
dist:
   d(  0)=  0   d(  1)=  0   d(  2)=  0   d(  3)=  0   d(  4)=  1   ........
   d(  8)=  0   d(  9)=  0   d( 10)=  0   d( 11)=  0   d( 12)=  0   ........
......
```

Exercice 6

Dans la fonction `main`, utiliser l'une des deux fonctions `histogramme_xxx` de la bibliothèque `histogramme` pour obtenir une représentation visuelle de la distribution obtenue avec `calcule_repartition`.

Testez votre fonction depuis la fonction `main` sur un tableau de 10000 nombres. La distribution vous parait-elle uniforme ?

exemple :

```
graine: 105
  0 | ****************************************
  1 | ****************************************
  2 | ****************************************
  3 | ****************************************
.......
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 33 | nombres hexadécimaux

Présentation

Le but de ce TP est d'approfondir les chaînes de caractères, les pointeurs et l'allocation dynamique, en représentant et manipulant des nombres hexadécimaux sous forme de chaînes de caractères.

1 Principe

On veut manipuler des entiers naturels exprimés en base 16. Un chiffre de la base 16 correspond soit à un chiffre de la base 10 (0,…,9), soit à l'une des 6 premières lettres de l'alphabet latin (A,…F).

On décide d'utiliser des valeurs de types `char` pour un représenter des chiffres de la base 16, ce qui permet de mélanger chiffres classiques et lettres. Par conséquent, on utilisera des chaînes de caractères pour représenter des nombres de la base 16.

exemples :

- Les chiffres $(1)_{16}$ et $(A)_{16}$ sont respectivement représentés par `'1'` et `'A'`.
- Les nombres $(1234)_{16}$ et $(1FF)_{16}$ sont représentés par `"1234"` et `"1FF"`.

Dans la suite du sujet, le mot *chiffre* désigne un caractère représentant un chiffre de la base 16, et le mot *nombre* désigne une chaîne de caractères représentant un nombre exprimé en base 16.

Remarques :

- Pensez que vous pouvez réutiliser les fonctions que vous écrivez.
- En langage C, il est possible d'additionner les caractères et les entiers.
- Dans la table ASCII, les chiffres sont rangés de `'0'` à `'9'`, les lettres de `'A'` à `'Z'`.
- Dans la table ASCII, `'A'` n'est *pas* placé juste après `'9'`.

2 Opérations simples

Exercice 1

Écrivez une fonction `int longueur_nombre(char *n)` qui calcule la longueur d'un *nombre* `n` (i.e. combien de *chiffres* composent le *nombre*).

Exercice 2

Écrivez une fonction `int verifie_nombre(char *n)` qui vérifie qu'une chaîne de caractères passée en paramètre représente bien un *nombre*. La fonction renvoie la valeur 1 si la chaîne est un *nombre* hexadécimal, ou 0 sinon.

exemples :

- Pour `n="12A4"`, la fonction doit renvoyer 1.
- Pour `n="1 2A4"`, la fonction doit renvoyer 0.
- Pour `n="12G4"`, la fonction doit renvoyer 0.

3 Comparaisons

Exercice 3

Écrivez une fonction `int compare_chiffres(char c1, char c2)` qui reçoit deux *chiffres* et qui effectue leur comparaison. La fonction doit renvoyer une valeur positive si $c_1 > c_2$, nulle si $c_1 = c_2$ ou négative si $c_1 < c_2$.

exemples :

- Pour `c1='A'` et `c2='3'`, la fonction doit renvoyer une valeur positive.
- Pour `c1='6'` et `c2='6'`, la fonction doit renvoyer 0.
- Pour `c1='3'` et `c2='9'`, la fonction doit renvoyer une valeur négative.

Exercice 4

Écrivez une fonction `int compare_nombres(char *n1, char *n2)` qui reçoit deux *nombres* et qui effectue leur comparaison. La fonction doit renvoyer une valeur positive si $n_1 > n_2$, nulle si $n_1 = n_2$ ou négative si $n_1 < n_2$.

exemples :

- Pour `n1="12A"` et `n2="123"`, la fonction doit renvoyer une valeur positive.
- Pour `n1="12A"` et `n2="12A"`, la fonction doit renvoyer 0.
- Pour `n1="12A"` et `n2="12B"`, la fonction doit renvoyer une valeur négative.

4 Conversions

Exercice 5

Écrivez une fonction `char convertit_chiffre(int x)` qui reçoit une valeur $0 \leq x < 16$ exprimée en base 10, et qui renvoie le chiffre correspondant en base 16.

exemples :

- Pour `x=1`, la fonction doit renvoyer `'1'`.
- Pour `x=11`, la fonction doit renvoyer `'B'`.

Exercice 6

Écrivez une fonction `char* convertit_nombre(int x)` qui reçoit une valeur $x > 0$ exprimée en base 10. La fonction doit créer un *nombre* représentant cette valeur en base 16, et renvoyer l'adresse de cette chaîne de caractères. L'espace mémoire occupé par la chaîne doit être alloué dynamiquement.

exemple : pour $x = 255$, la fonction doit renvoyer l'adresse d'une chaîne `"FF"`.

Algorithme :

- Calculer l'espace que va occuper la chaîne
- Faire l'allocation dynamique
- Calculer la représentation du nombre en base 16 : procédez par divisions successives par 16.

5 Modifications

Exercice 7

Écrivez une fonction `void incremente_nombre(char **n)` qui permet d'ajouter la valeur 1 à un *nombre* `n`. La fonction doit éventuellement agrandir la zone de mémoire allouée à la chaîne de caractères.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 34 | répertoire téléphonique

Présentation

Le but de ce TP est d'approfondir les mécanismes d'allocation dynamique de la mémoire, ainsi que les indirections multiples (pointeurs de pointeurs). Pour le dernier exercice, vous aurez besoin de certaines fonctions de manipulation des chaînes de caractères, contenues dans la bibliothèque `chaine` fournie avec ce sujet.

1 Numéros de téléphone

On veut représenter et manipuler un ensemble de numéros de téléphone. Pour cela, on va utiliser des tableaux d'entiers. Le problème est qu'on ne sait pas à l'avance combien de numéros de téléphone on va devoir stocker. On ne sait pas non plus la longueur des numéros : les numéros de personnes habitant à Istanbul sont de la forme `XXX XX XX`, les numéros de téléphones portables ou pour le reste de la Turquie sont de la forme `0 XXX XXX XX XX`, les numéros pour la France sont de la forme `00 33 X XX XX XX XX`, etc.

Il serait possible d'utiliser des tableaux très grands pour stocker tous ces numéros, mais cela pose deux problèmes :

- On ne peut pas être certain que ces tableaux seront assez grands
- S'ils sont trop grands, on gaspille de l'espace mémoire.

Pour éviter cela, nous allons utiliser des pointeurs et les fonctions d'allocation dynamique de la mémoire (`malloc`, `calloc`, etc.).

Exercice 1

On veut représenter un numéro téléphonique sous la forme d'un tableau d'entiers de taille indéterminée. Ce tableau ne contient que des chiffres : pas de ponctuation ou d'espace `" "`. La fin de ce tableau est marquée par une valeur spéciale : -1.

Créez une nouvelle bibliothèque `repertoire`. Dans `repertoire.h`, définissez une constante `FIN` qui prend la valeur entière -1. Cette constante devra être utilisée dans vos fonctions lorsque vous testerez la fin d'un numéro.

exemple : les numéros `123 456 789` et `12 34 56 789` sont tous les deux représentés par le tableau `{1,2,3,4,5,6,7,8,9,FIN}`.

Remarque : on utilise un système assez similaire à celui des chaînes de caractères. La différence est que dans celles-ci, on utilise le caractère `'\0'` pour marquer la fin de chaîne. Ici, on ne peut pas utiliser ce caractère, car sa valeur numérique est 0. Or, 0 est une valeur tout à fait acceptable dans un numéro de téléphone. Il nous faut une valeur qui ne peut pas apparaitre normalement dans un numéro : par exemple, une valeur négative comme -1.

Exercice 2

Dans `repertoire`, écrivez une fonction `void affiche_numero(int *)` qui reçoit un tableau d'entiers représentant un numéro, et qui l'affiche à l'écran, en tenant compte du fait que la fin du numéro est marquée par la valeur `FIN`.

exemple : pour le numéro de l'exemple précédent, on obtient simplement :

```
123456789
```

Exercice 3

Dans `repertoire`, écrivez une fonction `void saisis_numero(int** numero)` qui permet à l'utilisateur de saisir un numéro de téléphone. Cette saisie doit être effectuée caractère par caractère, grâce à la fonction `getch` déjà utilisée lors du TP sur l'allocation dynamique.

La fonction doit filtrer la chaîne, de manière à ne garder que les chiffres. Ces chiffres sont stockés dans un tableau d'entiers qui doit faire *exactement* la taille nécessaire. Il doit être réservé dynamiquement avec `malloc`, et l'adresse de l'espace mémoire réservé doit être renvoyée via le paramètre `numero`.

exemple :

```
Entrez le numéro de téléphone : 0 589 123 45 67
```

Notez que le texte `Entrez le numéro de téléphone` est affiché depuis la fonction `main`. On obtient le tableau suivant en mémoire :

?	?	?	0	5	8	9	1	2	3	4	5	6	7	FIN	?	?	?

Remarque : Un tableau d'entiers est de type `int*`. L'adresse d'un tableau d'entier est donc de type `int**`, d'où le type du paramètre `numero`.

2 Contacts

Un numéro de téléphone seul n'est pas très utile : encore faut-il savoir à qui il appartient. On veut donc non seulement stocker les numéros, mais aussi les noms de leurs propriétaires. Pour cela, nous allons utiliser un type structure.

Exercice 4

Dans `repertoire`, définissez un type structuré `t_contact` permettant de représenter à la fois un nom `nom` (i.e. une chaîne de caractères) et un numéro de téléphone `numero` (i.e. un tableau d'entiers terminé par `FIN`). Notez bien qu'on ne connait pas *a priori* les tailles de ces tableaux.

Exercice 5

Dans `repertoire`, écrivez les fonctions `affiche_contact(t_contact contact)` et `saisis_contact(t_contact* contact)`. La première permet d'afficher un contact à l'écran, alors que la seconde permet de saisir un contact et de le renvoyer par adresse. Vous devez bien entendu utiliser les fonctions existantes.

exemples :

- Saisie d'un contact :

```
Entrez le nom du contact : Can
Entre le numero de Can : 0538 458 12 34
```

- Affichage de ce même contact :

```
Can : 05384581234
```

Remarque : pour le nom, pensez à vous inspirer de la fonction `saisis_chaine_direct` du TP sur l'allocation dynamique.

3 Répertoire

Nous allons utiliser un tableau de contacts pour représenter le répertoire. Il est également nécessaire de savoir combien de contacts notre répertoire contient.

Exercice 6

Ces deux informations peuvent être représentées ensemble, sous la forme d'un type structure. Dans `repertoire`, définissez le type `t_repertoire`, qui contient un tableau de contacts appelé `contacts` et un entier `taille` indiquant le nombre de contacts dans `contacts`. Encore une fois, on ne connait pas *a priori* la taille maximale du répertoire.

Exercice 7

Dans `repertoire`, écrivez la fonction `void initialise_repertoire (t_repertoire* repertoire)` qui initialise les deux champs du répertoire passé en paramètre, de manière à obtenir un répertoire vide.

Exercice 8

Dans `repertoire`, écrivez la fonction `ajoute_contact(t_repertoire* repertoire, t_contact contact)`, qui permet d'ajouter un nouveau contact `contact` dans un répertoire existant `repertoire`.

Remarque : lorsque vous testez votre fonction, n'oubliez pas d'initialiser votre répertoire avant d'y ajouter un contact.

Exercice 9

Dans `repertoire`, écrivez la fonction `affiche_repertoire(t_repertoire repertoire)` qui affiche tout le contenu d'un répertoire à l'écran.

exemples :

- Pour un répertoire vide :

```
Taille du repertoire : 0 contact.
```

- Pour un répertoire contenant plusieurs contacts :

```
Taille du repertoire : 3 contacts.
  1.Can : 05384581234
  2.Zeynep : 4443322
  3.Cem : 0587123123456
```

Exercice 10

Dans `repertoire`, écrivez la fonction `t_contact* cherche_nom(t_repertoire repertoire, char* nom)` qui cherche si le répertoire contient un contact qui s'appelle `nom`. Si ce contact existe, la fonction renvoie son adresse. Sinon, elle renvoie `NULL`.

exemples : on reprend le répertoire de l'exemple précédent

- Une recherche du nom `"Cemm"` doit renvoyer `NULL`.
- Une recherche du nom `"Zeynep"` doit renvoyer l'adresse du 2ème contact.

Remarque : pour cet exercice, vous devez utiliser la bibliothèque `chaine`.

Exercice 11

Dans `repertoire`, écrivez la fonction `supprime_contact(t_repertoire* repertoire, t_contact* contact)` qui supprime du répertoire le contact spécifié. On suppose que ce contact est bien présent dans le répertoire, et que l'adresse passée en paramètre n'est pas `NULL`.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 35 | fichiers et arguments d'un programme

Présentation

Le but de ce TP est de traiter deux nouveaux points :

- Le passage d'arguments à un programme via la ligne de commande :
- L'accès aux fichiers en lecture et en écriture.

Nous allons pour cela nous appuyer sur deux bibliothèques écrites lors de TP précédents : `date`, qui est spécialisée dans la représentation et la manipulation des dates, et `promotion`, qui permet de représenter un groupe d'étudiants.

Remarques : pensez bien à traiter les erreurs potentielles des fonctions d'accès aux fichiers.

1 Arguments d'un programme

Lorsqu'on exécute un programme depuis la ligne de commande du système d'exploitation, il est possible de lui passer des arguments. Par exemple, dans la commande suivante, le programme est `cd` tandis que l'expression `/home` est son argument :

```
cd /home
```

En langage C, ces arguments passés au programme sont récupérés dans la fonction `main` de la façon suivante :

```
int main(int argc, char *argv[])
{  ...
}
```

L'entier `argc` indique le nombre d'arguments qui ont été passés par la ligne de commande, et le tableau `argv` contient ces différents arguments, sous la forme de chaînes de caractères.

Remarque : le nom du programme exécuté est compté dans les arguments, donc `argv[0]` correspond toujours à ce nom (ici : `TP01.exe`).

Exercice 1

Créez votre projet Eclipse, puis créez le programme principal `main.c`. Écrivez une fonction `void affiche_arguments(int argc, char *argv[])` qui affiche simplement le nombre de arguments total reçus par le programme, le nom du programme et la valeur de tous les autres arguments reçus. Cette fonction doit être appelée depuis la fonction `main`.

Pour tester votre programme, vous devez l'exécuter depuis un terminal système, et non pas depuis Eclipse. Ouvrez d'abord un terminal système, puis déplacez-vous jusqu'au dossier contenant votre projet. Allez ensuite dans le dossier `Debug` et entrez le nom du fichier exécutable suivi par les arguments que vous voulez lui donner.

exemple : supposons que :

- Le projet s'appelle `TP` ;
- Il se situe dans le dossier `/home/student/eclipse/workspace` ;
- On veut lui passer les arguments `aaa`, `111`, `bcd` et `123456`.

Alors la séquence de commandes à entrer dans le terminal système sera :

```
cd /home/student/eclipse/workspace/TP
cd Debug
TP aaa 111 bcd 123456
```

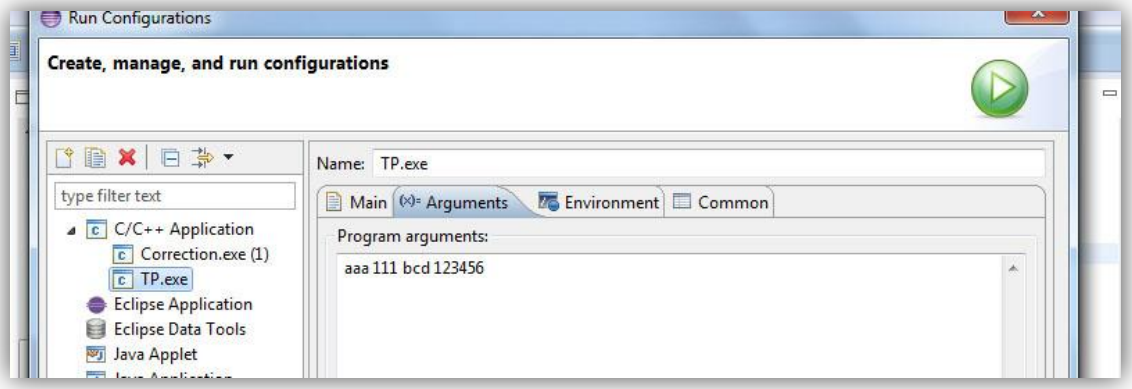
Et vous devez obtenir l'affichage suivant :

```
Nombre d'arguments recus : 5
Nom du programme : /home/student/eclipse/workspace/TP/TP
Argument 1 : aaa
Argument 2 : 111
Argument 3 : bcd
Argument 4 : 123456
```

Exercice 2

Il est également possible de passer des arguments à un programme quand on l'exécute depuis Eclipse, bien que cela soit moins pratique.

- Allez dans le menu *Run* > *Run configurations…* : une nouvelle fenêtre apparait.
- Dans la liste des projets situés dans la partie gauche, cliquez sur celui que vous voulez exécuter.
- Dans la partie droite, cliquez sur l'onglet *Arguments* : vous pouvez alors entrer vos arguments dans la zone *Program arguments*. Ils seront transmis à votre programme à chaque fois que vous l'exécuterez.



Exécutez votre programme à partir d'Eclipse, en utilisant les mêmes arguments que dans l'exercice précédent. Vous devez obtenir exactement le même affichage qu'avant.

2 Accès non-formaté à un fichier

Exercice 3

Dans le programme principal `main.c`, écrivez une fonction `ecris_chaine` qui demande à l'utilisateur de saisir une chaîne de caractères et qui l'écrit en *mode caractère* dans un fichier `donnees.txt` situé dans le projet. En cas d'erreur, fonction doit renvoyer -1 ; en cas de succès, elle doit renvoyer le nombre de caractères écrits. Attention, vous ne devez *pas* écrire le caractère de fin de chaîne `'\0'` dans le fichier.

Testez votre fonction depuis la fonction `main`, et vérifiez le fichier obtenu en l'ouvrant avec l'éditeur de texte d'Eclipse.

exemple : pour la chaîne `"abcdef"`, si tout se passe bien, la fonction renvoie 6 et le fichier `donnees.txt` contient exactement et uniquement les caractères `abcdef`.

Remarques :

- Les chemins relatifs sont exprimés par rapport à la racine du projet.
- Votre programme va créer un fichier dans le projet, mais celui-ci n'apparaitra pas automatiquement. Pour le voir, il faut rafraichir le projet : dans *Projet Explorer* (à gauche), faites un clic-droit sur votre projet, puis *Refresh*.

Exercice 4

Écrivez une fonction `lis_chaine` qui lit le contenu du fichier texte précédent, donc toujours en *mode caractère*, et qui affiche la chaîne de caractères obtenue. En cas d'erreur, La fonction doit renvoyer -1 ; en cas de succès, elle doit renvoyer le nombre de caractères lus. Attention : le fichier texte ne contient pas le `'\0'` final.

exemple : pour le fichier créé à l'exemple précédent, la fonction doit renvoyer la valeur 6 et afficher :

```
La chaine lue est : "abcdef"
```

Exercice 5

Écrivez une fonction `int copie_fichier(char *source, char *cible)` qui crée une copie du fichier `source` appelée `cible`. Pour effectuer la copie, vous devez lire des *lignes de texte* dans le fichier source et les écrire dans le fichier cible. En cas de succès, la fonction renvoie le nombre de lignes copiées, en cas d'erreur elle renvoie -1. Attention, ici on ne travaille plus avec de simples caractères comme dans les exercices précédents, mais avec des *chaînes* de caractères.

Depuis la fonction `main`, testez votre fonction sur le fichier `donnees.txt` en créant une copie que vous appellerez `copie.txt`, elle aussi située dans le projet Eclipse. Avant d'exécuter votre programme, rajoutez plusieurs lignes de texte dans `donnees.txt` à l'aide de l'éditeur d'Eclipse.

exemple : si le fichier donnees.txt contient le texte ci-dessous, alors elle doit renvoyer la valeur 4.

```
Ceci est la premiere ligne.
Et ceci est la deuxieme ligne.
Et encore une autre, la troisieme ligne.
Enfin, on finira avec une 4eme ligne.
```

Exercice 6

On veut généraliser le programme en utilisant le passage d'arguments étudié dans la première partie de ce TP :

- Le premier argument du programme doit correspondre au fichier source ;
- Le second argument doit correspondre au fichier cible.

Dans la fonction `main`, effectuez les tests nécessaires sur les arguments reçus par le programme, et appelez la fonction `copie_fichier` avec les paramètres appropriés.

exemple : la commande suivante permet de créer une copie de `donnees.txt` appelée `copie2.txt` :

```
TP donnees.txt copie2.txt
```

Remarque : cette commande peut aussi bien être exécutée dans le terminal système, ou bien configurée dans Eclipse comme expliqué précédemment.

3 Accès formaté à un fichier

Exercice 7

Dans la bibliothèque `date`, écrivez la fonction `int sauve_date(t_date d, FILE *fp)` utilisant l'accès formaté pour écrire une date `d` dans un fichier *déjà ouvert en écriture*, désigné par le pointeur `fp`. En cas d'erreur la fonction renvoie -1, et en cas de succès 0.

Contrainte : vous ne pouvez utiliser qu'une seule fois la fonction d'écriture en accès formaté. Utilisez `"/"` pour séparer les différentes composantes d'une date.

Exercice 8

Dans la bibliothèque `date`, écrivez une fonction `int charge_date(t_date *d, FILE *fp)` qui réalise l'opération inverse de `sauve_date`, c'est-à-dire qui lit une date dans un fichier *déjà ouvert en lecture*, désigné par le pointeur `fp`. La fonction renvoie -1 en cas d'erreur, et 0 en cas de succès. La date lue est passée par adresse.

Contrainte : vous ne pouvez utiliser qu'une seule fois la fonction de lecture en accès formaté.

Exercice 9

Modifiez le fichier `date.txt` en ajoutant 10 zéros après l'année de la première date. Chargez et affichez cette date. Que remarquez-vous ? Comment l'expliquez-vous ?

Exercice 10

Même chose, mais cette fois en remplaçant les `"/"` de la première date par des `":"`.

Exercice 11

Dans la bibliothèque `promotion`, écrivez les fonctions `sauve_etudiant` et `charge_etudiant`, permettant respectivement d'enregistrer et de lire un étudiant dans un fichier.

Remarque : utilisez une tabulation `'\t'` pour séparer les différents champs décrivant chaque étudiant.

Exercice 12

Toujours dans la bibliothèque `promotion`, écrivez les fonctions `sauve_promotion` et `charge_promotion`, qui effectuent les mêmes actions pour une `promotion`. En cas d'erreur, ces deux fonctions doivent renvoyer -1 ; en cas de succès, elles doivent renvoyer le nombre d'étudiants écrits ou lus.

exemple : pour la promotion générée par `initialise_promotion`, la fonction doit renvoyer la valeur 9.

Remarque : représentez chaque étudiant sur une ligne distincte.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 36 | diaporama

Présentation

Dans ce TP, on veut afficher une séquence de photos en utilisant la SDL. La liste des photos à afficher est contenue dans un fichier texte. Le but de ce TP est d'approfondir le passage de paramètres par adresse, les pointeurs, l'allocation dynamique, et la manipulation de chaînes de caractères.

La bibliothèque `chaine` contient quelques fonctions de manipulation des chaînes de caractères, utiles dans la deuxième partie du TP. La bibliothèque `graphisme` est fournie pour utiliser la SDL, cependant elle n'est nécessaire que dans la dernière partie du TP. Pour vous faciliter l'écriture des fonctions, il est conseillé de ne configurer le projet pour la SDL que lorsque vous arriverez à cette dernière partie.

1 Noms des images

Les noms des fichiers contenant les images à afficher sont listés dans un fichier texte, lui-même contenu dans le dossier `album` fourni avec le sujet. On veut représenter cette liste de noms de fichiers sous la forme d'un tableau contenant des pointeurs vers des chaînes de caractères. Chaque chaîne correspondra à un nom de fichier.

Cependant, on ne connait pas à l'avance la taille de ce tableau, car on ne sait pas combien de noms d'image la liste contient. Le tableau doit donc être alloué dynamiquement. La même remarque est valable pour les chaînes, qui devront elles aussi être allouées dynamiquement.

Exercice 1

Écrivez une fonction `int lis_nom(FILE *fp, char** nom)` qui reçoit en paramètre un fichier déjà ouvert en lecture `fp`, et l'adresse d'un pointeur sur des caractères (i.e. l'adresse d'une chaîne de caractères) `nom`. Cette fonction doit lire *un seul* nom dans `fp`, et le placer dans `nom`. De plus, elle doit renvoyer la valeur 0 si la fin du fichier a été atteinte lors de cette lecture, ou 1 sinon.

Voici l'algorithme à utiliser :

- Allouer un tableau de 1 caractère (pour contenir le `'\0'` final).
- Lire le contenu du fichier 1 caractère à la fois. Pour chaque caractère lu, on doit effectuer les opérations suivantes :
  - D'abord on ré-alloue l'espace mémoire de `nom`, de manière à l'agrandir d'un caractère ;
  - Ensuite, on copie le caractère lu, à l'emplacement approprié dans `nom`.
- On s'arrête si l'une des deux conditions suivantes est atteinte :
  - On est arrivé à la fin du fichier : dans ce cas-là, on renvoie la valeur 0.
  - On est arrivé à la fin de la chaîne (i.e. le caractère lu est `'\n'`) : on renvoie la valeur 1.
  - Dans les deux cas, il faut aussi compléter nom avec le dernier caractère `'\0'`.

Testez votre fonction à partir de la fonction `main`, en procédant de la manière suivante :

- Ouvrez le fichier `image.txt`, contenu dans le dossier `album` fourni avec le sujet ;
- Appelez `lis_nom` en récupérant son résultat (l'entier renvoyé),
- Affichez l'entier renvoyé et la chaîne obtenue,
- Recommencez l'appel 4 fois (pour lire tous les noms contenus dans `images.txt`).

Vous devez obtenir *exactement* l'affichage suivant :

```
res=1 - nom=image2.bmp
res=1 - nom=image9.bmp
res=1 - nom=image6.bmp
res=0 - nom=
```

Exercice 2

Écrivez une fonction `int charge_liste(char fichier[], char*** images)` qui reçoit en paramètre `fichier` (le nom du fichier texte contenant la liste d'images), ouvre ce fichier texte en lecture, charge son contenu, et le place dans le tableau `images` de manière à obtenir un tableau de pointeurs vers des chaînes de caractères, chacune correspondant au nom d'une image. Le tableau image doit être (ré-)alloué dynamiquement, de manière à faire exactement la taille nécessaire. La fonction doit renvoyer un entier correspondant au nombre de noms d'images lus. Vous devez, bien entendu, utiliser la fonction `lis_nom`.

L'algorithme à appliquer est le suivant :

- Ouvrez (en lecture) le fichier passé en paramètre ;
- Lisez chaque nom d'image contenu dans ce fichier grâce à `lis_nom`. Pour chaque nom lu, il faut effectuer les opérations suivantes :
  - Ré-allouer l'espace mémoire de `images`, afin de l'agrandir d'une adresse ;
  - Placer dans `images`, à l'emplacement approprié, l'adresse de la chaîne de caractère lue par `lis_nom`.
- On s'arrête quand on a atteint la fin du fichier (i.e. quand `lis_nom` renvoie 0).
- On ferme le fichier.

Dans la fonction `main`, testez votre fonction de la façon suivante :

- Appliquez `charge_liste` au fichier `images.txt` contenu dans le dossier `album` du projet, afin d'initialiser le tableau `images`.
- Affichez le contenu du tableau `images`, de manière à obtenir *exactement* l'affichage ci-dessous (position et nom de chaque image dans le fichier).

```
0. image2.bmp
1. image9.bmp
2. image6.bmp
```

2 Chemins des images

Les noms des images contenus dans le fichier texte `images.txt` ne comportent pas le dossier contenant ces images (i.e. le dossier `album`). Il est nécessaire de compléter leur chemin avec ce dossier, afin de pouvoir charger les images ensuite.

Exercice 3

Écrivez une fonction `void ajoute_dossier(char* images[], int n, char dossier[])` qui reçoit un tableau `images` contenant des pointeurs vers des noms d'images,

la taille `n` de ce tableau (i.e. le nombre de pointeurs qu'il contient) et le nom `dossier` du dossier contenant les images.

La fonction doit modifier chaque nom contenu dans `images`, de manière à le préfixer avec le nom du dossier. Par exemple, pour une image `"image1.bmp"` et un dossier `"album"`, on veut ajouter le préfixe `"album/"`, de manière à obtenir le chemin `"album/image1.bmp"`.

Remarque : comme d'habitude, pour Windows il faut remplacer `'/'` par `'\\'`.

L'algorithme à appliquer est le suivant :

- On calcule la longueur (en caractères) du préfixe.
- On traite séparément chaque `nom` contenu dans `images`.
- On augmente (par ré-allocation) l'espace mémoire assigné à chaque `nom`.
- On déplace les caractères présents vers la fin de la chaîne.
- On copie le préfixe au début de la chaîne.

Remarque : utilisez la bibliothèque chaine pour calculer les longueurs.

Dans la fonction `main`, testez votre fonction de la façon suivante :

- Appelez `charge_liste` comme précédemment.
- Appelez `ajoute_dossier`, en utilisant le dossier `"album"`.
- Affichez le contenu du tableau `images`, comme précédemment. Là encore, vous devez obtenir *exactement* l'affichage ci-dessous.

```
0. album\image2.bmp
1. album\image9.bmp
2. album\image6.bmp
```

3 Affichage des images

Dans cette partie, nous allons profiter du tableau images, qui contient les chemins de toutes les images que l'on veut afficher. Nous allons dessiner ces images à l'écran en utilisant la SDL. La transition entre les images se fera automatiquement, avec un certain délai de quelques secondes pour avoir le temps de voir les images.

Vous devez d'abord configurer votre projet pour qu'il utilise la SDL. Pensez également, dans la fonction `main`, à initialiser la fenêtre graphique avec `initialise_fenetre`.

Exercice 4

Écrivez une fonction `void diaporama_image(char *chemin, int delai)` qui reçoit le chemin complet d'une image et un délai exprimé en ms. La fonction doit afficher l'image au centre de la fenêtre graphique, puis attendre un certain délai avant de se terminer.

Les différentes étapes de cette fonction sont :

- Charger l'image ;
- Calculer les coordonnées d'affichage ;
- Effacer l'écran ;
- Afficher l'image ;
- Attendre le délai spécifié.

Rappels :

- Le type `SDL_Surface` possède des champs `w` et `h` permettant d'accéder respectivement à la largeur et à la hauteur d'une image.
- La bibliothèque `graphisme` contient deux constantes `FENETRE_LARGEUR` et `FENETRE_HAUTEUR`, représentant les dimensions de la fenêtre graphique.

Testez votre fonction dans la fonction `main`, en spécifiant manuellement un chemin complet valide et un délai de 4 secondes.

Exercice 5

Écrivez une fonction `void diaporama_tout(char fichier[], char dossier[], int delai)`, qui affiche successivement toutes les images du diaporama, en les enchaînant automatiquement, et en respectant le délai spécifié. Le paramètre `fichier` est le nom du fichier texte contenant la liste des images. Ce fichier texte, ainsi que les images elles-mêmes, sont contenus à l'emplacement spécifié, `dossier`.

L'algorithme à appliquer est le suivant :

- On construit une chaîne de caractères en concaténant `dossier` et `fichier` (séparés par un `'/'`)
- On charge la liste d'images contenue dans ce fichier ;
- On ajoute `dossier` aux noms de cette liste pour obtenir les chemins complets des images ;
- Pour chaque chemin obtenu, on charge et affiche l'image.

Bien sûr, il faut utiliser les fonctions des exercices précédents.

Testez votre fonction dans la fonction `main`, en utilisant le dossier `"album"`, le fichier `"images.txt"` et un délai de 1 seconde.

Remarque : vous pouvez manuellement éditer le fichier texte `images.txt` pour y rajouter d'autres images, et rallonger ainsi la durée du diaporama.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 37 | stock d'une librairie

Présentation

Le but de ce TP est d'approfondir certaines notions déjà vues en cours et appliquées lors d'autres TP, en particulier : allocation dynamique, types structurés, et accès aux fichiers. L'archive fournie avec ce sujet contient la bibliothèque `date`, écrite et utilisée lors de TP précédents, et qui est nécessaire pour certaines questions.

1 Structure de données

On veut réaliser un programme permettant de représenter l'ensemble des livres d'occasion (i.e. des livres qui ne sont pas neufs) vendus par une librairie. Chaque livre est caractérisé par un code composé de 5 caractères, et par la date à laquelle il a été publié.

La librairie peut posséder plusieurs exemplaires d'un même livre. Chaque exemplaire peut avoir un prix différent (suivant qu'il est en bon état ou en mauvais état).

Exercice 1

Dans votre projet, créez une bibliothèque `livre`, et définissez un type `t_livre` permettant de représenter un livre par :

- Son code ;
- Sa date de publication ;
- Le nombre d'exemplaires que la librairie possède ;
- Les prix du livre (chaque exemplaire peut avoir un prix différent).

Remarque : on ne connait pas à l'avance le nombre d'exemplaires d'un livre, et on ne veut pas fixer de limite a priori : il faut donc utiliser un tableau alloué dynamiquement.

2 Affichage & saisie

Exercice 2

Dans la bibliothèque `livre`, écrivez une fonction `void affiche_livre(t_livre livre)` qui affiche les caractéristiques du livre passé en paramètre.

exemple : livre de code `AGZ12` publié le 12/10/2004 et présent en 3 exemplaires de prix 12,5, 20 et 12,3 € :

```
AGZ12 - 12 octobre 2004 - 3 exemplaires : 12.50 20.00 12.30
```

Exercice 3

Dans la bibliothèque `livre`, écrivez une fonction `void saisis_livre(t_livre *livre)` qui permet à l'utilisateur de saisir les informations concernant un livre.

exemple : pour le livre de l'exemple précédent :

```
Entrez le code : AGZ12
Entrez la date de publication :
Entrez le jour : 12
Entrez le mois : 10
Entrez l'annee : 2004
Entrez le nombre d'exemplaires : 3
```

```
Entrez le prix de l'exemplaire 1 : 12.5
Entrez le prix de l'exemplaire 2 : 20
Entrez le prix de l'exemplaire 3 : 12.3
```

3 Enregistrement & chargement

Exercice 4

Dans la bibliothèque `livre`, écrivez une fonction `void sauve_livre(t_livre livre, FILE *f)` qui enregistre le livre passé en paramètre dans le fichier désigné par `f` (on suppose que le fichier a déjà été ouvert en écriture).

Remarque : dans le fichier créé, séparez les champs en utilisant des tabulations `'\t'`.

Exercice 5

Dans la bibliothèque `livre`, écrivez une fonction `void charge_livre(t_livre* livre, FILE *f)` qui charge un livre dans le fichier désigné par `f` (on suppose que le fichier a déjà été ouvert en lecture), et qui passe ce livre par adresse.

4 Stock de la librairie

On utilise un tableau de type `t_livre` pour représenter tous les livres stockés dans la librairie. On suppose que la librairie peut stocker seulement un nombre fini `N` d'exemplaires : ce tableau peut donc être alloué de façon statique, puisqu'on sait qu'il contiendra au maximum `N` livres.

Exercice 6

Dans la bibliothèque `livre`, écrivez une fonction `void saisis_stock(t_livre stock[N], int* n)` qui permet de saisir tous les livres constituant le stock d'une librairie. Le tableau reçu est supposé initialement vide. La fonction doit utiliser `saisis_livre` pour le remplir, et permettre à l'utilisateur de s'arrêter quand il le souhaite, comme dans l'exemple ci-dessous. De plus, elle doit vérifier que l'utilisateur ne saisit pas plus de `N` livres. Le paramètre `n` sert à renvoyer par adresse le nombre de livres saisis.

exemple :

```
Livre 1 :
Entrez le code : AGZ12
Entrez la date de publication :
Entrez le jour : 12
Entrez le mois : 10
Entrez l'annee : 2004
Entrez le nombre d'exemplaires : 3
Entrez le prix de l'exemplaire 1 : 12.5
Entrez le prix de l'exemplaire 2 : 20
Entrez le prix de l'exemplaire 3 : 12.3
Voulez-vous saisir un autre livre (O/N) ? O
Livre 2 :
Entrez le code : ZZX38
Entrez la date de publication :
Entrez le jour : 5
Entrez le mois : 4
Entrez l'annee : 2014
Entrez le nombre d'exemplaires : 5
Entrez le prix de l'exemplaire 1 : 99.99
Entrez le prix de l'exemplaire 2 : 100
Entrez le prix de l'exemplaire 3 : 122
Entrez le prix de l'exemplaire 4 : 92
Entrez le prix de l'exemplaire 5 : 100.50
Voulez-vous saisir un autre livre (O/N) ? N
Termine : 2 livres ont été saisis
```

Exercice 7

Dans la bibliothèque `livre`, écrivez une fonction `void affiche_stock(t_livre stock[N], int n)` qui affiche les `n` livres contenus dans le tableau stock passé en paramètre, à la manière de l'exemple ci-dessous.

exemple : pour les livres saisis dans l'exemple précédent

```
1.AGZ12 - 12 octobre 2004 - 3 exemplaires : 12.50 20.00 12.30
2.ZZX38 - 5 avril 2014 - 5 exemplaires : 99.99 100.00 122.00 92.00 100.50
```

Exercice 8

Dans la bibliothèque `livre`, écrivez une fonction `void sauve_stock(t_livre stock[N], int n, char* fichier)` qui reçoit en paramètre un tableau représentant le stock de la librairie, et une chaîne de caractères représentant le nom d'un fichier. La fonction doit ouvrir ce fichier en écriture, et y enregistrer les `n` livres présents dans le stock. Bien sûr, la fonction doit utiliser `sauve_livre`.

Testez votre fonction en saisissant quelques livres avec `saisis_stock`, puis en enregistrant avec `sauve_stock` le stock obtenu, et enfin en visualisant directement dans Eclipse le contenu du fichier texte ainsi produit.

Exercice 9

Dans la bibliothèque `livre`, écrivez une fonction `void charge_stock(t_livre stock[N], int* n, char* fichier)` qui reçoit en paramètre un tableau représentant le stock de la librairie, qui est pour l'instant vide, et une chaîne de caractères représentant le nom d'un fichier. La fonction doit ouvrir ce fichier en lecture, et en charger tous les livres contenus, pour les placer le tableau `stock`. Bien entendu, la fonction doit utiliser `charge_livre`. Le paramètre `n` sert à renvoyer par adresse le nombre de livres lus dans le fichier.

Testez votre fonction en éditant, directement dans Eclipse, le fichier texte produit à l'exercice précédent, de façon à le compléter avec quelques livres supplémentaires ; puis en chargeant ce fichier avec `charge_stock` et enfin en l'affichant avec `affiche_stock`.

5 Opérations sur le stock

Exercice 10

Dans la bibliothèque `livre`, écrivez une fonction `int compte_exemplaires(t_livre stock[N], int n)` qui compte le nombre d'exemplaires en stock.

Exercice 11

Dans la bibliothèque `livre`, écrivez une fonction `float estime_stock(t_livre stock[N], int n)` qui reçoit en paramètre un stock (tableau de livres `stock` et nombre de livres `n`), et qui calcule et renvoie la valeur marchande du stock, i.e. la somme des prix de tous les exemplaires stockés.

exemple : pour le stock des exemples précédents, la fonction doit renvoyer $12{,}5 + 20 + 12{,}3 + 99{,}99 + 100 + 122 + 92 + 100{,}5 = 559{,}29$ €.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

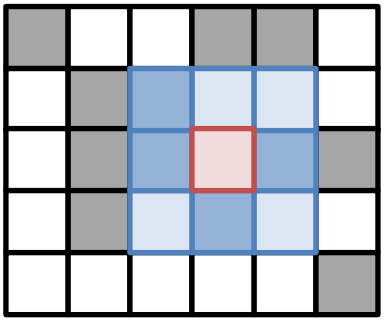
TP 38 | automates cellulaires

Présentation

Dans ce TP, nous allons utiliser nos connaissances sur les matrices, les fichiers et l'allocation dynamique pour implémenter des *automates cellulaires* simples. On s'intéressera en particulier au *jeu de la vie* de Conway.

1 Définition

Un automate cellulaire est un ensemble de *cellules* disposées de façon géométrique, par exemple sous forme de grille 2D comme ci-dessous. Chaque cellule est caractérisée par son *état*, qui est une valeur discrète. Dans le cas le plus simple, l'état est binaire : la cellule est soit morte, soit vivante.



L'état d'une cellule à un instant donné t dépend des états de ses voisines à l'instant précédent $t-1$. Le *voisinage* d'une cellule peut être défini de différentes façons. Ainsi, dans l'exemple ci-dessus, on considère que les 8 cellules colorées en bleu sont les voisines de la cellule colorée en rouge. La fonction utilisée pour mettre à jour l'état d'une cellule en fonction des états de ses voisines peut prendre de nombreuses formes.

L'étude des automates cellulaires fait partie de l'informatique théorique, et permet de modéliser et de comprendre les propriétés de systèmes dynamiques complexes. La forme d'automate cellulaire la plus connue est le Jeu de la vie, défini par John Conway dans les années 1970. Les états sont binaires, et les cellules sont réparties comme dans la figure ci-dessus, avec le même voisinage que celui donné en exemple. La règle de mise à jour des états est la suivante :

- Si une cellule est *morte* :
  - Si elle possède exactement 3 voisines vivantes, elle renait (elle devient vivante).
  - Sinon, elle reste morte.
- Si une cellule est *vivante* :
  - Si elle possède exactement 2 ou 3 voisines vivantes, elle reste vivante.
  - Sinon, elle meurt.

Dans la figure ci-dessus, les cases sombres représentent des cellules vivantes. La cellule rouge est morte, et elle possède 4 cellules vivantes dans son voisinage, donc elle va rester dans cet état (morte).

2 Représentation

On va représenter un automate cellulaire par une matrice de $N \times M$ entiers, la valeur 0 représentant une cellule morte et la valeur 1 une cellule vivante.

Exercice 1

Les fonctions concernant les automates cellulaires vont être placées dans une bibliothèque spéciale appelée `automate`, qui utilisera la bibliothèque `graphisme` pour donner une représentation graphique. Créez les fichiers nécessaires, et déclarez les constantes `N` et `M`.

Remarque : pour des raisons esthétiques, utilisez des valeurs multiples de la résolution de la fenêtre graphique (cf. les constantes dans la bibliothèque `graphisme`). Par exemple, 60 et 80.

Exercice 2

Dans `automate`, écrivez une fonction `void vide_automate(int automate[N][M])` qui reçoit une matrice représentant un automate, et qui place toutes ses cellules à l'état mort (i.e. valeur 0).

Exercice 3

Dans `automate`, écrivez une fonction `void copie_automate(int automate1[N][M], int automate2[N][M])` qui recopie les états des cellules du 1er automate dans celles du 2nd.

Exercice 4

Dans `automate`, écrivez une fonction `void dessine_automate(int automate[N][M])` qui dessine l'automate dans la fenêtre graphique, en utilisant au maximum l'espace disponible. Les cellules vivantes doivent être représentées en blanc, les mortes en noir, et un quadrillage gris doit être tracé par-dessus, comme dans les captures qui suivent.

3 Simulation

Exercice 5

Dans `automate`, écrivez une fonction `int compte_voisines_vivantes(int automate[N][M], int i, int j)` qui compte le nombre de cellules *vivantes* dans le voisinage de la cellule située à la position (i, j) dans la matrice formant l'automate.

Remarque : si le voisinage est incomplet parce que la cellule se trouve en bordure de matrice, on considère les cellules manquantes comme mortes.

Exercice 6

Dans `automate`, écrivez une fonction `void itere_automate(int automate[N][M])` qui met à jour toutes les cellules de l'automate. La fonction doit renvoyer 1 si au moins une cellule a changé d'état, et 0 si aucune n'a changé d'état.

Remarque : à la fin du traitement, la matrice `automate` doit contenir les états *mis à jour*. Cependant, le calcul de mise à jour nécessite d'utiliser la matrice *originale*. Il est donc nécessaire de la copier grâce à la fonction `copie_automate`.

Exercice 7

Dans `automate`, écrivez une fonction `void simule_automate(int automate[N][M], int iterations)` qui effectue plusieurs itérations de mise à jour de l'automate. Si la valeur du paramètre `iterations` est 0, alors la fonction doit effectuer des

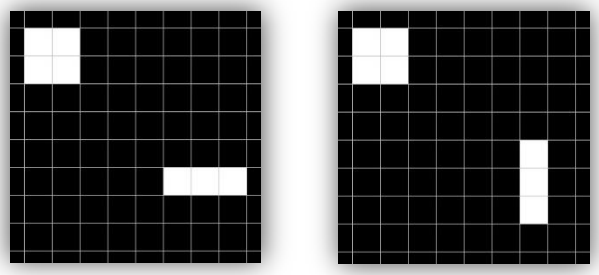
mises à jour tant qu'au moins une cellule change d'état. Si la valeur de `iterations` est un entier non-nul, alors la fonction doit effectuer le nombre indiqué d'itérations.

À chaque itération effectuée, la fonction doit dessiner l'automate obtenu, et attendre un certain temps grâce à `attends_delai`.

Exercice 8

Pour tester vos fonctions, initialisez un automate à zéro avec `vide_automate`, puis construisez les deux figures suivantes :

- Un *bloc* de 2 × 2 cellules vivantes ;
- Un *vecteur* de 1 × 3 cellules vivantes.



La première figure (bloc) ne doit pas évoluer au cours des itérations, alors que la seconde (vecteur) va alterner entre les formes horizontales et verticales du vecteur, comme représenté ci-dessus.

4 Structures cellulaires

Une part importante du travail sur les automates cellulaires consiste à identifier des *structures cellulaires*, i.e. des groupes de cellules possédant des propriétés particulières. Le bloc et le vecteur donnés en exemple dans l'exercice précédent sont des structures cellulaires : le premier est une *structure stable* alors que le second est un *oscillateur*.

Exercice 9

Le dossier `structures` fourni avec ce sujet contient plusieurs structures différentes, représentées sous formes de fichiers textes. La forme de ces fichiers textes est la suivante :

- Les nombres de lignes et de colonnes dans la structure, séparés par une tabulation ;
- Une matrice représentant la structure, dont les valeurs sont séparées par des tabulations.

exemple : la structure vecteur de l'exemple précédent est représentée de la façon suivante (les underscores `'_'` représentent des tabulations)

```
5__5
0__0__0__0__0__
0__0__0__0__0__
0__1__1__1__0__
0__0__0__0__0__
0__0__0__0__0__
```

Après avoir étudié attentivement ce format de fichier, écrivez dans `automate` une fonction `int charge_structure(char* fichier, int** structure, int* lignes, int* colonnes)` qui charge la structure cellulaire contenue dans le fichier dont le nom `fichier` est passé en paramètre.

Cette fonction doit allouer dynamiquement la place nécessaire à la matrice représentant la structure cellulaire, puis initialiser cette matrice avec les valeurs contenues dans le fichier. Elle doit renvoyer −1 en cas d'erreur, et 0 en cas de succès. Notez bien que les dimensions de la matrice, *ainsi que la matrice elle-même*, sont passées par adresse.

Exercice 10

Dans `automate`, écrivez une fonction `void ajoute_structure(int automate[N][M], int* structure, int lignes, int colonnes, int i, int j)` qui place la structure cellulaire spécifiée à la position spécifiée dans l'automate passé en paramètre. Autrement dit, la fonction va recopier le contenu de la matrice représentant la structure cellulaire à l'endroit approprié dans la matrice représentant l'automate cellulaire. L'élément $(0,0)$ de `structure` doit être placé à la position (i,j) de `automate`, et ainsi de suite.

Exercice 11

On peut maintenant jouer avec les différents types de structures existant :

- **Statique** : structure qui est stable dans le temps, i.e. si on ne la perturbe pas, elle n'évolue pas du tout.
- **Oscillateur** : structures qui évolue dans le temps, mais qui est caractérisée par un cycle, i.e. elle revient toujours à sa forme initiale, après un nombre fini d'étapes.
- **Vaisseau** : structure qui ressemble à un oscillateur, à la différence qu'elle se déplace.
- **Puffeur** : structure qui ressemble à un vaisseau, à la différence qu'elle laisse de nouvelles structures derrière elle au cours de son déplacement.
- **Canon** : structure qui ressemble à un oscillateur, à la différence qu'elle produit des vaisseaux.
- **Mathusalem** : structure relativement simple, mais instable et qui prend un grand nombre d'itérations avant de se stabiliser.

Les vaisseaux, en particulier sont caractérisés par leur vitesse, exprimée en nombre de cellules par itération. Quelle sont les vitesses des vaisseaux proposés sous forme de fichiers texte ? Pouvez-vous en proposer un plus rapide ?

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 39 | fonctions récursives

Présentation

Le but de ce TP est d'écrire des fonctions récursives simples. On les appliquera à différents types de données : chaînes de caractères et entiers. Les exercices de ce TP sont indépendants : vous ne devez pas réutiliser les fonctions précédentes (sauf si le contraire est explicitement indiqué).

1 Chaînes de caractères

Exercice 1

Écrivez une fonction `est_contenu` qui reçoit en paramètres une chaîne de caractères `chaine` et un caractère `c`. La fonction doit récursivement chercher `c` dans `chaine`. Si `c` est contenu dans `chaine`, la fonction doit renvoyer 1, sinon, elle doit renvoyer 0.

exemples :

- Pour `chaine="edbac"` et `c='b'` la fonction doit renvoyer 1.
- Pour `chaine="edbac"` et `c='f'` la fonction doit renvoyer 0.

Exercice 2

Écrivez une fonction `compte_occurrences` qui reçoit en paramètres une chaîne de caractères `chaine` et un caractère `c`. La fonction doit calculer récursivement le nombre d'occurrences de `c` dans `chaine` (i.e. le nombre de fois que `c` apparaît).

Remarque : il s'agit d'une *variable* `c`, et non pas de la *valeur littérale* `'c'`.

exemples :

- Pour `chaine="aedabac"` et `c='a'` la fonction doit renvoyer 3.
- Pour `chaine="aedabac"` et `c='b'` la fonction doit renvoyer 1.
- Pour `chaine="aedabac"` et `c='f'` la fonction doit renvoyer 0.

Exercice 3

Écrivez une fonction `calcule_longueur` qui reçoit en paramètre une chaîne de caractères `chaine`, calcule récursivement sa longueur (sans compter le `'\0'`) et la renvoie par valeur.

Exercice 4

Écrivez une fonction `calcule_position` qui reçoit en paramètres une chaîne de caractères `chaine` et un caractère `c`. La fonction doit récursivement chercher `c` dans `chaine`. Si `c` est contenu dans `chaine`, la fonction doit renvoyer par valeur la position de sa première occurrence. Sinon, elle doit renvoyer -1.

exemples :

- pour `chaine="edbac"` et `c='b'` la fonction doit renvoyer 2.
- pour `chaine="edbac"` et `c='f'` la fonction doit renvoyer -1.

2 Entiers

Exercice 5

On considère la *suite de Fibonacci* définie par (pour $n \in \mathbb{N}$) :

$$u_n = \begin{cases} u_0 = 0,\ u_1 = 1 \\ \forall n \geq 2, u_n = u_{n-1} + u_{n-2} \end{cases}$$

Écrivez une fonction `calcule_fibonacci` qui reçoit `n` en paramètre, calcule récursivement le terme u_n, puis le renvoie par valeur.

Donnez l'arbre des appels pour $n = 4$. En observant l'arbre obtenu, pensez-vous qu'il serait possible d'écrire une fonction plus rapide ?

Exercice 6

On veut calculer récursivement le *plus grand commun diviseur* (PGCD) de deux nombres (x, y). On sait que :

- Si l'un des deux nombres est nul, alors le PGCD est l'autre nombre.
- Sinon :
  - si $x \geq y$ alors le PGCD de (x, y) est égal à celui de $(x - y, y)$.
  - si $x < y$ alors le PGCD de (x, y) est égal à celui de $(x, y - x)$.

Écrivez une fonction `calcule_pgcd` recevant deux paramètres `x` et `y`, calculant récursivement leur PGCD, et le renvoyant par valeur.

Exercice 7

Le calcul de x^y peut se faire à l'aide de la relation de récurrence suivante :

$$x^y = \begin{cases} 1 \text{ si } y = 0 \\ (x^2)^{y/2} \text{ si } y \neq 0 \text{ et } y \text{ est pair} \\ x \times (x^2)^{y/2} \text{ si } y \neq 0 \text{ et } y \text{ est impair} \end{cases}$$

Écrivez une fonction `long calcule_puissance(long x, long y)` qui applique cette relation pour calculer x^y.

Donnez l'arbre des appels pour $x = 2$ et $y = 15$. Cette méthode est-elle plus ou moins efficace que celle vue en cours ?

3 Chaînes de caractères (suite)

Exercice 8

Un nombre romain peut se composer des chiffres suivants (représentés par des lettres **majuscules**) :

- M = 1000
- D = 500
- C = 100
- L = 50
- X = 10
- V = 5
- I = 1

De plus, un nombre romain est interprété grâce aux règles suivantes :

- Si le nombre ne contient qu'un seul chiffre, alors la valeur du nombre est égale à celle du chiffre.
- Si le nombre contient au moins deux chiffres, on considère les deux chiffres les plus à gauche :
  - Si le 1er chiffre est supérieur ou égal au 2ème, alors la valeur du nombre est égale à la **somme** du 1er chiffre et de la valeur du nombre composé des chiffres restants.
  - Si le 1er chiffre est inférieur au 2ème, alors la valeur du nombre est égale à la **différence** entre la valeur du nombre composé des chiffres restants et le 1er chiffre.

exemples :

- IIII = IV = 4
- VIIII = IX = 9
- MIM = MCMXCIX = 1999…

Écrivez d'abord une fonction `int evalue_chiffre_romain(char chiffre)`, qui n'est *pas* récursive, et qui renvoie la valeur numérique associée au chiffre romain passé en paramètre sous la forme d'un simple caractère.

Ensuite, écrivez une fonction `int evalue_nombre_romain(char *nombre)` qui reçoit en paramètre un nombre romain représenté par une chaîne de caractères, et qui renvoie par valeur le résultat de l'évaluation récursive de ce nombre. Vous devez bien sûr utiliser `evalue_chiffre_romain`.

Remarque : on suppose que la syntaxe du nombre est correcte, inutile de la vérifier.

Exercice 9

On dit qu'une chaîne de caractères `chaine1` est un *préfixe* d'une autre chaîne `chaine2` ssi la chaîne `chaine1` correspond exactement au début de la chaîne `chaine2`.

exemples :

- `"bon"` est un préfixe de `"bonjour"`.
- `"bonne"` n'est pas un préfixe de `"bonjour"`.
- `"bonne"` n'est pas un préfixe de `"bon"`.
- `""` (chaîne vide) est préfixe de n'importe qu'elle chaîne.

Écrivez une fonction récursive `int est_prefixe(char* chaine1, char* chaine2)` qui renvoie 1 si `chaine2` est un préfixe de `chaine1`, et 0 sinon.

Donnez l'arbre des appels pour les appels suivants :

- `est_prefixe("bon", "bonjour")`
- `est_prefixe("bonne", "bonjour")`

Exercice 10

On dit que la chaîne `chaine1` *contient* une autre chaine `chaine2` ssi la chaîne `chaine2` est un préfixe d'une sous-chaîne de `chaine1`. De façon équivalente, on dit aussi que `chaine2` est contenue dans `chaine1`.

exemples :

- `"njo"` est contenue dans `"bonjour"`.
- `"onje"` n'est pas contenue dans `"bonjour"`.
- `"bonjour"` n'est pas contenue dans `"onj"`.
- Aucune chaîne non vide n'est contenue dans la chaîne vide `""`.

Écrivez une fonction récursive `int est_contenue(char* chaine1, char* chaine2)` qui détermine si la chaîne de caractères `chaine1` est contenue dans la chaine `chaine2`. Elle renvoie 1 si c'est le cas, et 0 sinon. Vous devez utiliser la fonction `est_prefixe`.

Donnez ensuite l'arbre des appels pour les appels suivant :

- `est_contenue("njo","bonjour")`
- `est_contenue("onje","bonjour")`

Exercice 11

Écrivez une fonction récursive `void ftnirp(char *chaine)` qui affiche une chaîne de caractères à l'envers, sans la modifier.

exemple : `ftnirp("abcdef")` doit afficher `fedcba`.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 40 | approximations numériques

Présentation

Le but de ce TP est d'utiliser les notions vues jusqu'à présent en cours et TP, en les appliquant à deux problèmes d'approximation numérique. On s'intéressera d'abord à la fonction cosinus, puis à la constante π.

1 Développement en série entière

Pout tout réel θ, on peut calculer la valeur de $\cos(\theta)$ en utilisant le développement en série entière de la fonction cosinus :

$$\cos(\theta) = \sum_{n=0}^{+\infty} (-1)^n \frac{\theta^{2n}}{(2n)!}$$

Pour obtenir une valeur approchée de $\cos(\theta)$, on peut ne calculer que les k premiers termes de ce développement :

$$\cos(\theta) \approx \sum_{n=0}^{k-1} (-1)^n \frac{\theta^{2n}}{(2n)!}$$

Pour que l'approximation soit bonne, il faut choisir une valeur de k qui dépend de θ. En pratique, cette formule devient incalculable pour un ordinateur, même pour des valeurs de θ assez petites (par exemple $= 3\pi$). Par contre pour des valeurs telles que $|\theta| < 1$, elle peut être utilisée.

Exercice 1

Écrivez une fonction *récursive* `int calcule_factorielle(int n)` qui calcule et renvoie $n!$.

exemple : $6! = 720$

Exercice 2

Écrivez une fonction *récursive* `double calcule_puissance(double x, int n)` qui calcule x^n en utilisant le principe de l'exponentiation rapide suivant :

$$x^0 = 1$$
$$x^n = \begin{cases} x(x^k)^2 \text{ si } n = 2k+1 \\ (x^k)^2 \quad \text{si } n = 2k \end{cases}$$

exemple : $2^6 = 64$

Exercice 3

Écrivez une fonction `double calcule_cosinus(double theta, double delta)` qui utilise le principe d'approximation par série entière décrit en début de section, pour calculer le cosinus de `theta`. Votre fonction utilisera les fonctions des deux exercices précédents.

Au lieu d'utiliser une valeur k fixée à l'avance, on spécifiera une valeur δ correspondant à un changement minimal dans le principe itératif suivant :

- On calcule le 1er terme du développement.
- On calcule le 2ème terme du développement.
- On les additionne pour obtenir une approximation du cosinus.
- Tant que la valeur absolue du dernier terme calculé est supérieure à δ :
  - On calcule le terme suivant, qu'on ajoute à l'approximation du cosinus.

On s'arrête donc quand un terme ne modifie plus l'approximation de façon significative.

Pour des raisons de débogage, votre fonction doit afficher l'évolution de l'approximation et les termes calculés, comme indiqué ci-dessous. De plus, vous devez afficher 10 chiffres après la virgule, afin de pouvoir observer l'évolution de la précision jusqu'au bout du traitement.

exemple : affichage obtenu pour l'approximation de $\pi/6$, avec `delta=e-8` :

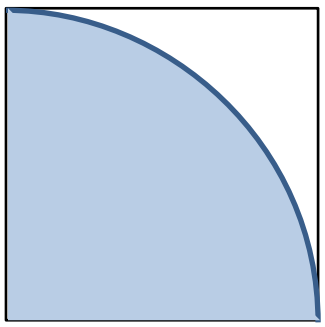
```
i=0   1.0000000000
i=1   1.0000000000 - 0.1370697535 = 0.8629302465
i=2   0.8629302465 + 0.0031313529 = 0.8660615994
i=3   0.8660615994 - 0.0000286143 = 0.8660329852
i=4   0.8660329852 + 0.0000001401 = 0.8660331252
i=5   0.8660331252 - 0.0000000004 = 0.8660331248
cos(0.523583)~0.866033 vs. cos(0.523583)=0.866033
```

Remarque : dans la dernière ligne de l'affichage, on compare la valeur estimée et celle calculée avec la bibliothèque mathématique `math.h`.

2 Méthode de Monte-Carlo

L'expression Monte-Carlo désigne une famille de méthodes reposant sur le hasard pour estimer certaines valeurs. Dans notre cas, nous voulons estimer la valeur de π.

Considérons la figure suivante, qui représente le 1er quadrant du disque-unité :



On a les aires suivantes :

- Aire du quart-de-disque : $A_D = \frac{1}{4}\pi r^2 = \frac{\pi}{4}$
- Aire du carré : $A_C = r^2 = 1$.

Supposons qu'on tire au sort un point dans le carré considéré, de façon *uniforme* (i.e. chaque coordonnée a la même chance d'être tirée que n'importe quelle autre). Alors la probabilité que ce point soit situé dans le disque est égale à la proportion de l'aire du carré occupée par le quart-de-disque, i.e. $p = A_D/A_C = \frac{\pi}{4}/1 = \frac{\pi}{4}$. Par conséquent, on a $\pi = 4p$.

Donc, si on réussit à estimer p, on aura aussi estimé π. Une méthode simple consiste à :

- Tirer un grand nombre n_C de points dans le carré ;
- Compter le nombre n_D de ces points qui sont tombés dans le quart-de-disque.
- Alors $p \simeq n_D/n_C$, i.e. la proportion de points tirés tombés dans le quart-de-disque.

Exercice 4

Écrivez une fonction `double tire_reel()` qui tire au sort une valeur réelle comprise entre 0 et 1 (inclus).

Rappel : pour tirer une valeur au hasard en langage C, vous devez utiliser les fonctions `srand` et `rand`. La première sert à initialiser la graine du générateur pseudo-aléatoire, en général en utilisant la fonction `time` de la bibliothèque `time.h`. La seconde sert à tirer une valeur entière comprise entre 0 et `RAND_MAX` (incluses). Pour plus de détails, voyez les TP concernés.

Exercice 5

Écrivez une fonction `void calcule_pi(int nc)` qui calcule π en utilisant la méthode décrite ci-dessus. Le paramètre `nc` correspond au nombre de points à tirer dans le carré.

Testez la précision de cette méthode à partir de la fonction `main`, en utilisant des valeurs de `nc` de plus en plus grandes, comme dans l'exemple ci-dessous.

exemple : estimations obtenues pour quelques valeurs croissantes de `nc` :

```
          5 points : pi~2.400000
         10 points : pi~3.600000
        100 points : pi~2.920000
       1000 points : pi~3.160000
      10000 points : pi~3.152400
     100000 points : pi~3.147040
    1000000 points : pi~3.142104
   10000000 points : pi~3.141124
  100000000 points : pi~3.141719
 1000000000 points : pi~3.141502
```

Remarque : bien sûr, à cause de l'aspect aléatoire de cette approche, chaque exécution est susceptible de produire des résultats légèrement différents.

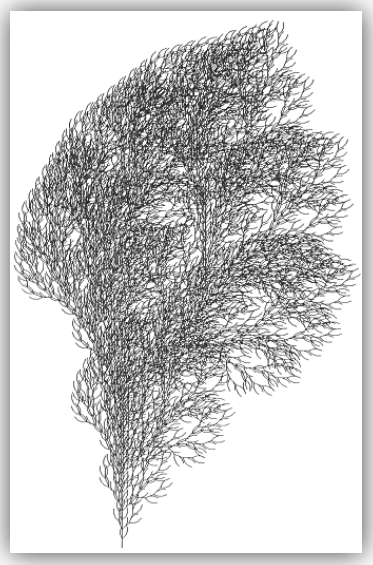
Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 41 | figures fractales



1 Présentation

Une figure fractale est construite en répétant un motif à l'infini : si on agrandit la figure, on retrouve le même motif. Ce type de construction se retrouve fréquemment dans la nature : cristaux, végétaux, etc. La programmation récursive est tout particulièrement adaptée à la reproduction de figures fractales.

Pour dessiner ces figures, vous utiliserez la SDL. Dans Eclipse, créez un nouveau projet, et configurez-le pour la SDL[1]. Puis, copiez-y les fichiers contenus dans l'archive qui accompagne ce sujet. Ceux-ci définissent la bibliothèque graphisme, déjà utilisée précédemment.

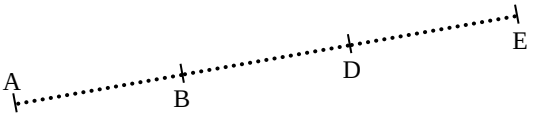
Vous devez également configurer votre projet pour l'utilisation de la librairie `math.h`. Vous aurez besoin de ses fonctions suivantes :

- `double sqrt(double x)` : calcule la racine carrée d'un nombre `x`.
- `double sin(double a)` : calcule le sinus de l'angle `a` exprimé en *radians*.
- `double cos(double a)` : calcule le cosinus de l'angle `a` exprimé en *radians*.

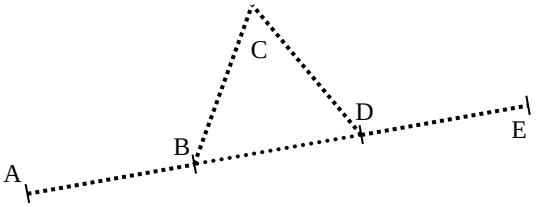
2 Flocon de von Koch

Un segment de *von Koch* est une figure fractale obtenue en partant d'un segment $[AE]$:

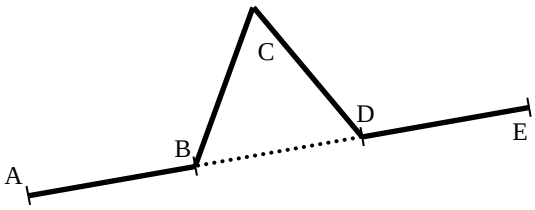
On partage ce segment $[AE]$ en trois parties $[AB]$, $[BD]$ et $[DE]$ de longueurs égales :



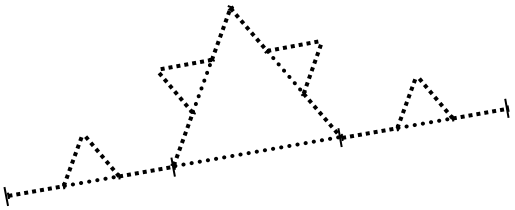
On place le point C de manière à ce que (B, C, D) forme un triangle équilatéral :



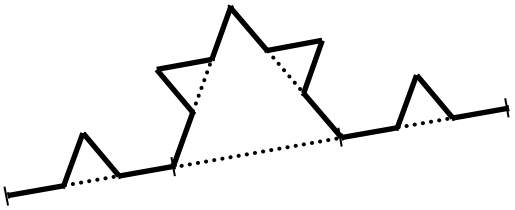
Dans le cas le plus simple du segment de von Koch, on trace les segments $[AB]$, $[BC]$, $[CD]$ et $[DE]$ afin d'obtenir la figure suivante :



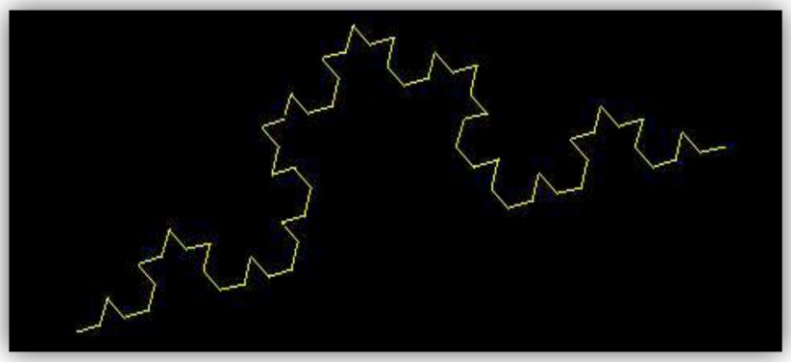
Cependant, pour obtenir l'effet fractal, il faut répéter récursivement cette construction. Pour cela, au lieu de tracer les segments comme ci-dessus, on applique une autre fois la même transformation, mais cette fois-ci aux segments $[AB]$, $[BC]$, $[CD]$ et $[DE]$. On obtient le résultat suivant :



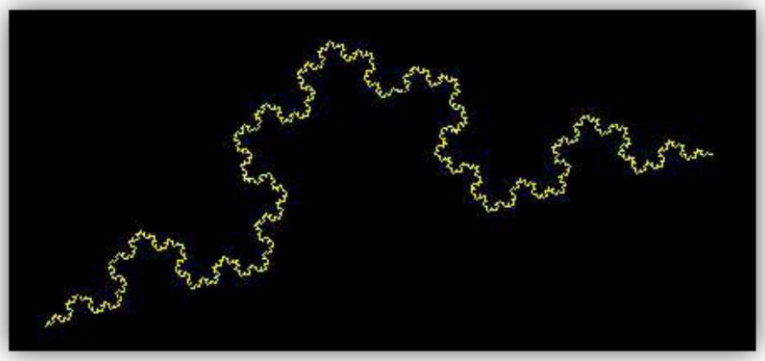
Et après avoir tracé les segments appropriés, on a :



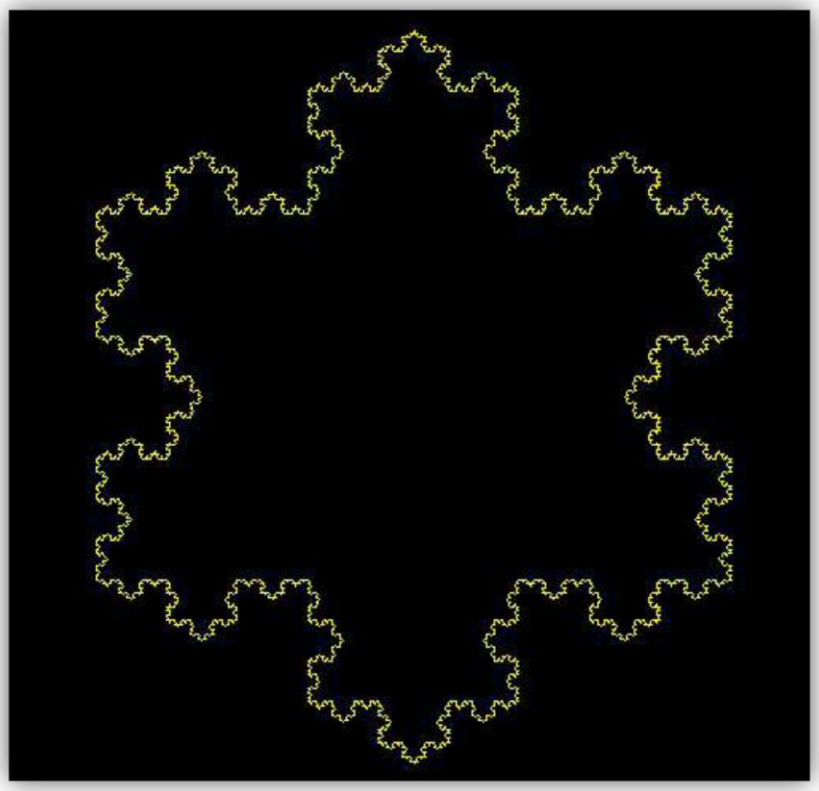
On répète récursivement le procédé pour obtenir une figure fractale. Si on réalise $n = 4$ itérations, on obtient la figure suivante :



Et si on fait $n = 6$ itérations, on a :

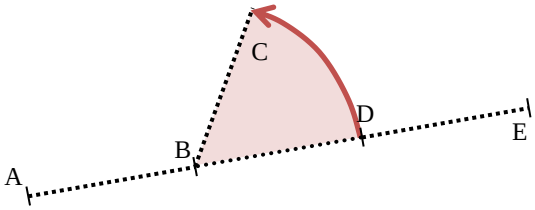


Si on applique ce traitement non pas à un seul segment, mais aux trois côtés d'un triangle équilatéral, on obtient un *flocon de von Koch* complet :



Exercice 1

Calculez les coordonnées des points B, C et D en fonction des coordonnées des points A et E. Vous remarquerez que C est l'image de D par la rotation d'axe B et d'angle $\pi/3$



Rappel : les coordonnées de l'image (x', y') d'un point (x, y) par une rotation d'angle α et de centre l'origine $(0,0)$ sont :

$$x' = x\cos(\alpha) - y\sin(\alpha)$$
$$y' = x\sin(\alpha) + y\cos(\alpha)$$

Créez un fichier `main.c` et écrivez une fonction `void calcule_segment_vonkoch(int xa, int ya, int* xb, int* yb, int* xc, int* yc, int* xd, int* yd, int xe, int ye)` qui reçoit en paramètres les coordonnées des points A et E de la figure précédente, et calcule les coordonnées des points B, C et D (attention aux passages par adresse !).

Exercice 2

Vérifiez vos calculs en écrivant une fonction `void teste_segment_vonkoch(int xa, int ya, int xe, int ye)` qui reçoit en paramètres les coordonnées des points A et E de la figure simple précédente, et trace les segments $[AB]$, $[BC]$, $[CD]$, et $[DE]$.

Exercice 3

En utilisant `calcule_segment_vonkoch`, écrivez une fonction récursive `void trace_segment_vonkoch(int x1, int y1, int x2, int y2, int n)` qui dessine la

figure obtenue en appliquant `n` fois la méthode de von Koch au segment de coordonnées (`x1`,`y1`) et (`x2`,`y2`).

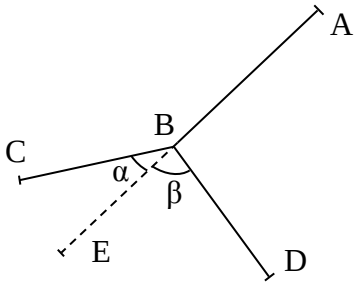
Exercice 4

En utilisant la fonction `segment_von_koch`, écrivez une fonction `void trace_flocon_vonkoch(int n)` qui dessine au centre de l'écran un flocon de von Koch en appliquant `n` fois la méthode de von Koch à chacun des côtés d'un triangle équilatéral.

Pour que la figure soit tracée pas à pas, vous devez utiliser les fonctions `attends_delai` et `rafraichis_fenetre` de manière à introduire un délai entre le tracé de chaque segment.

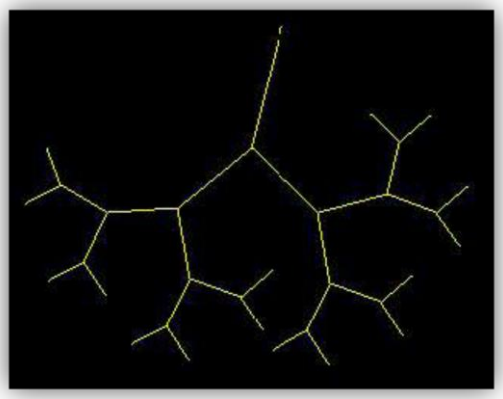
3 Fractale arborescente

En utilisant le même principe, il est possible de créer des figures fractales à partir de nombreuses figures géométriques différentes. Prenons un arbre binaire (i.e. à deux branches) :

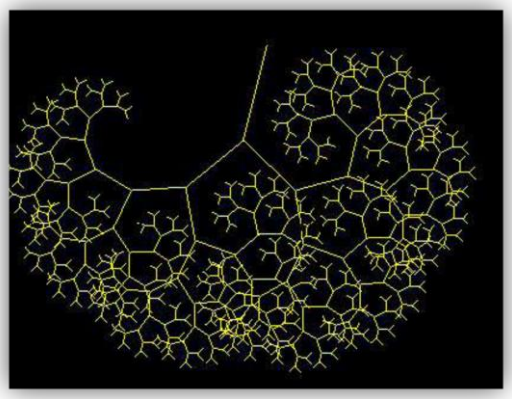
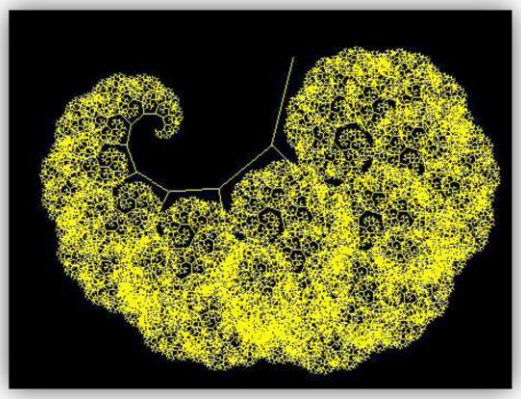


On part d'un segment $[AE]$ que l'on partage en deux en plaçant un point B tel que $BE = k \times AE$ avec $0 < k \leq 1$. Puis, on obtient les points C et D par rotations de centre B et d'angles respectifs α et β (ces deux angles peuvent être différents).

On répète ensuite récursivement le traitement sur $[BC]$ et $[BD]$, ainsi que sur leurs sous-segments, pour obtenir la figure fractale. En réitérant le traitement 5 fois, on obtient l'arbre suivant :



Et en le réitérant 10 et 15 fois :

Exercice 5

Écrivez une fonction `void calcule_segment_fractarbre(int xa, int ya, int* xb, int* yb, int* xc, int* yc, int* xd, int* yd, int xe, int ye)` qui calcule les coordonnées des points B, C et D à partir de celles de A et E.

Exercice 6

Vérifiez vos calculs en écrivant une fonction `void teste_segment_fractarbre(int xa, int ya, int xe, int ye)` qui reçoit en paramètres les coordonnées des points A et E de la figure simple précédente, et trace les segments $[AB]$, $[BC]$, et $[BD]$.

Exercice 7

Écrivez une fonction récursive `void trace_arbre(int x_1, int y_1, int x_2, int y_2, int n)` qui dessine la courbe obtenue en appliquant *n* fois la méthode précédente au segment de coordonnées (`x_1`,`y_1`) et (`x_2`,`y_2`). Vous utiliserez les paramètres suivants : $k = 0{,}3$; $\alpha = \pi/5$ et $\beta = -\pi/3$.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 42 | listes chaînées

Présentation

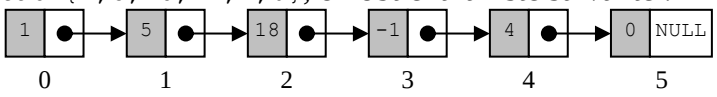
Le but de ce TP est d'utiliser les listes chaînées vues en cours. Les deux bibliothèques `liste_s` (listes simples) et `liste_d` (listes doubles) correspondant aux structures de données et fonctions étudiées en cours sont fournies avec ce sujet. Vous devez écrire vos propres fonctions dans le fichier `main.c`.

1 Opérations sur une liste simple

Exercice 1

Écrivez une fonction `int remplis_liste(int tab[], int taille, liste_s *l)` qui reçoit l'adresse d'une liste vide déjà initialisée et la remplit avec les valeurs contenues dans le tableau passé en paramètre. La fonction renvoie -1 si une erreur se produit, et 0 sinon.

exemple : pour `tab={1,5,18,-1,4,0}`, on obtient la liste suivante :



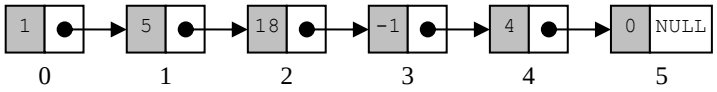
Exercice 2

Écrivez une fonction `int calcule_longueur(liste_s l)` qui renvoie le nombre d'éléments présents dans la liste `l` passée en paramètre.

Exercice 3

Écrivez une fonction `int recherche_valeur(liste_s l, int v)` qui recherche une valeur `v` dans une liste simplement chaînée `l` (qui n'est pas ordonnée). La fonction doit renvoyer la position de la première occurrence de la valeur dans la liste, ou -1 si la liste ne contient pas d'éléments de valeur `v`.

exemple :



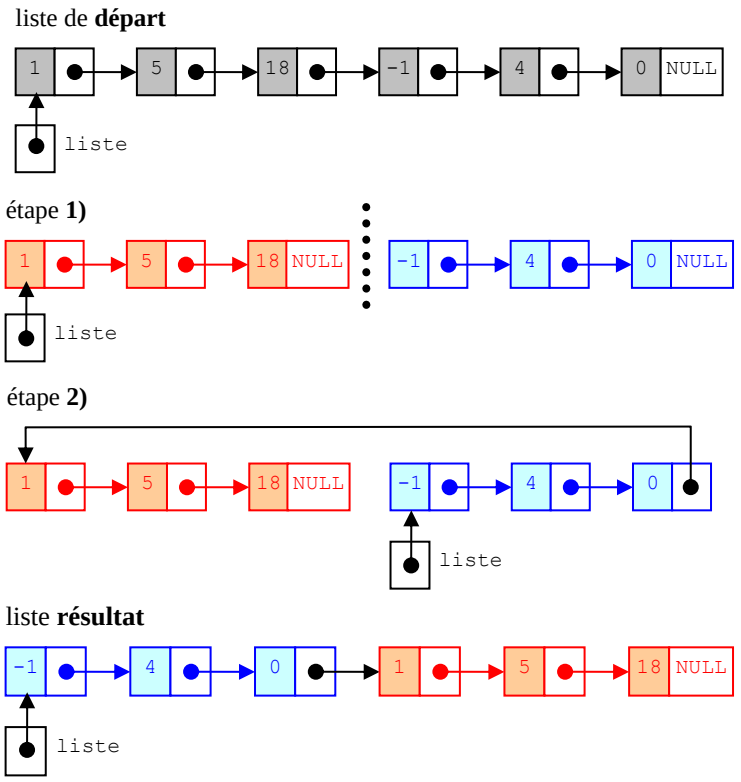
Si on recherche la valeur -1 dans la liste ci-dessus, la fonction doit renvoyer 3. Si on recherche la valeur 99, elle doit renvoyer -1.

Exercice 4

Écrivez une fonction `void melange_liste(liste_s *l)` qui prend un pointeur sur une liste simplement chaînée en paramètre, et dont le rôle est de :

1. couper la liste en deux, à la moitié (à un élément près si la taille est impaire)
2. intervertir les deux moitiés.

exemple :



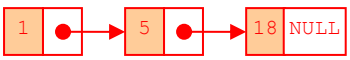
2 Opérations sur plusieurs listes simples

Exercice 5

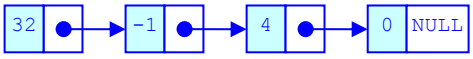
Écrivez une fonction `void fusionne_listes(liste_s *l1, liste_s l2)` qui prend en paramètres un pointeur sur une liste simplement chaînée `l1`, et une liste simplement chaînée `l2`. La fonction doit rajouter la liste `l2` à la fin de la liste `l1`.

exemple :

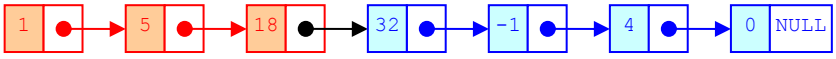
liste **1** avant la fusion



liste **2**



liste **1** après la fusion



Exercice 6

On dit qu'une liste `l2` préfixe une liste `l1` si `l2` apparaît entièrement au début de `l1`.

exemples :

l1	**l2**	**préfixe ?**
{**1, 2, 4,** 8, 0, 3}	{**1, 2, 4**}	oui
{**1,** 2, 4, 8, 0, 3}	{**1**}	oui
{**1, 2, 4, 8**}	{**1, 2, 4, 8**}	oui
{}	{}	oui
{1, 2, 4, 8, 0, 3}	{}	oui
{1, 2, 4, 8, 0, 3}	{5, 6}	non
{1, 2, **4, 8, 0,** 3}	{**4, 8, 0**}	non
{**1, 2, 4, 8, 0, 3**}	{**1, 2, 4, 8, 0, 3,** 7}	non

Écrivez une fonction récursive `int est_prefixe(liste_s l1, liste_s l2)` qui prend deux listes `l1` et `l2` en paramètres, et renvoie un entier indiquant si `l2` est préfixe de `l1`. L'entier doit valoir 1 pour oui et 0 pour non.

3 Opérations sur une liste double

Exercice 7

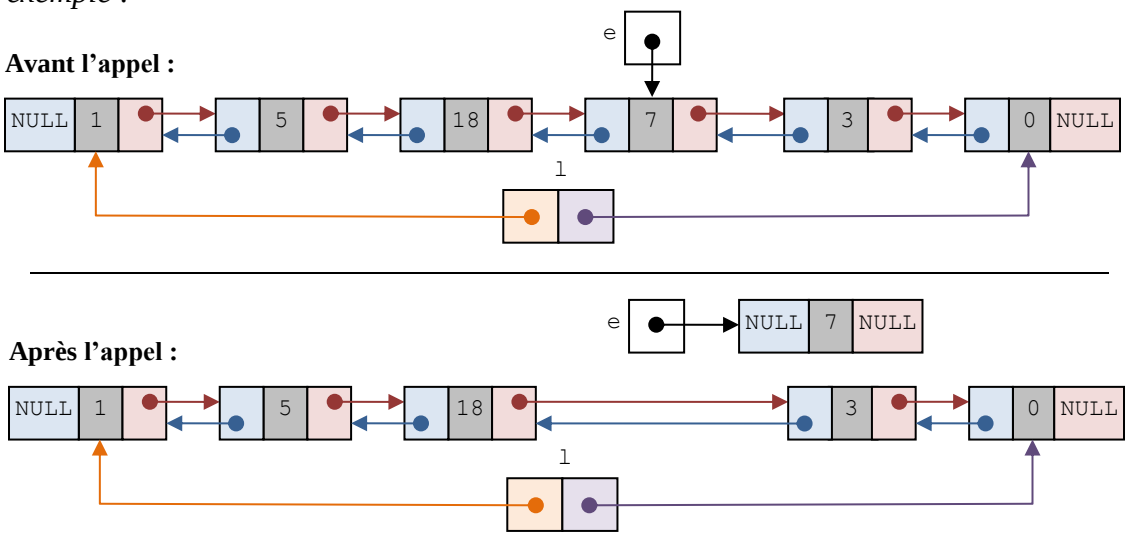
Écrivez une fonction `int echange_elements(liste_d *l, int p, int q)` qui intervertit les éléments d'une liste `l` doublement chaînée situés aux positions `p` et `q`. La fonction renvoie 0 en cas de succès ou -1 en cas d'erreur.

exemple : si on applique la fonction à la liste `1,25,34,59,2,6` et aux éléments situés aux positions `2` et `4`, on obtient la liste `1,25,2,59,34,6`.

Exercice 8

Complétez la fonction `int detache_element(liste_d *l, element_d *e)` qui enlève l'élément `e` de la liste `l`. Attention, l'élément ne doit pas être supprimé : il s'agit seulement de modifier les pointeurs des éléments de `l` de manière à ce que `e` n'en fasse plus partie.

exemple :



Si la liste `l` ne contient pas l'élément `e`, la fonction doit renvoyer -1, et sinon elle doit renvoyer 0. Il est alors recommandé de distinguer les 4 cas suivants :

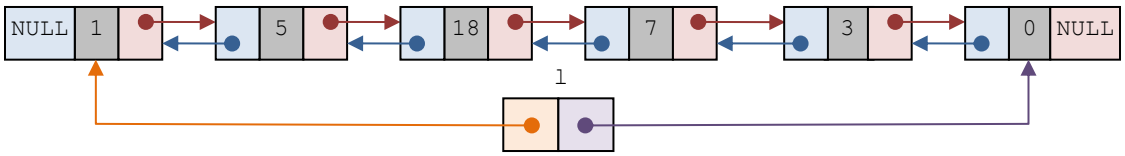
- La liste contient un seul élément (qui est bien sûr `e`) ;
- La liste contient plusieurs éléments et `e` est au début ;
- La liste contient plusieurs éléments et `e` est à la fin ;
- La liste contient plusieurs éléments et `e` n'est ni au début, ni à la fin.

Exercice 9

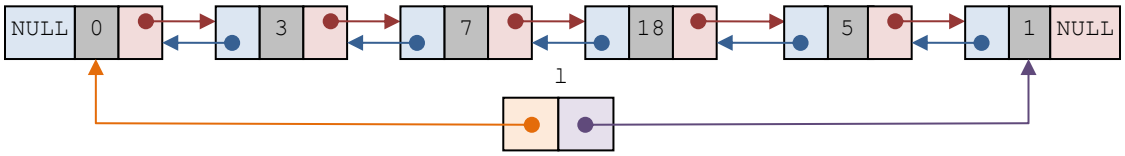
Écrivez une fonction `void inverse_liste(liste_d *l)` qui inverse l'ordre des éléments d'une liste `l`. Votre fonction devra utiliser la fonction `detache_element_d`.

exemple :

Avant l'appel :



Après l'appel :



Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 43 | disques & guirlandes

Présentation

Le but de ce TP de manipuler des listes en traçant différentes figures à base de disques. Pour cela, on utilisera la SDL, et la bibliothèque `graphisme` qui est fournie avec ce sujet. Cependant, ne configurez pas encore le projet pour utiliser la SDL, car les premiers exercices du TP nécessitent d'afficher du texte dans la console.

La bibliothèque `liste_s` est aussi fournie avec le sujet, pour la manipulation de listes simplement chaînées. Les fonctions et types écrits au cours du TP seront à placer dans l'une des deux nouvelles bibliothèques que vous devrez créer : `disque` et `guirlande`.

1 Points

Exercice 1

Créez une nouvelle bibliothèque appelée `disque`. Dans cette bibliothèque, définissez un type structuré `t_point`, permettant de représenter un point du repère en utilisant ses deux coordonnées `x` et `y`, toutes les deux exprimées en pixels.

Exercice 2

Dans `disque`, écrivez une fonction `void affiche_point(t_point p)` qui affiche le point `p` sous forme de texte, dans la console.

exemple : le point de coordonnées (12,13) sera affiché de la façon suivante :

```
(12,13)
```

2 Disques

Exercice 3

Dans `disque`, définissez un type structuré `t_disque` permettant de représenter un disque grâce à son `centre` (un point) et son `rayon` exprimé en pixels. Vous devez bien entendu utiliser le type `t_point`.

Exercice 4

Dans `disque`, écrivez une fonction `void affiche_disque(t_disque d)` qui affiche le disque `d` sous forme de texte, dans la console.

exemple : le disque de centre (12,13) et de rayon 9 sera affiché de la façon suivante :

```
(c : (12,13), r : 9)
```

Exercice 5

Dans `disque`, écrivez les fonctions `void modifie_centre(t_disque* d, int d_x, int d_y)` et `void modifie_rayon(t_disque* d, int d_r)` qui permettent respectivement de modifier le centre et le rayon d'un disque existant.

exemples :

- Si on appelle `modifie_centre` sur le disque de l'exemple précédent, avec $d_x = 10$ et $d_y = 50$, l'affichage du résultat donne :

```
(c : (22,63), r : 9)
```

- Si on appelle `modifie_rayon` sur le disque de l'exemple précédent, avec $d_r = 100$, l'affichage du résultat donne :

```
(c : (12,13), r : 109)
```

Exercice 6

Dans `disque`, écrivez la fonction `t_disque genere_disque()` qui génère un disque en sélectionnant aléatoirement un centre et un rayon.

Rappel : la génération d'un nombre aléatoire en C passe par les fonctions `srand(time(NULL))` (pour l'initialisation de la série pseudo-aléatoire) et `rand()` (pour tirer un nombre entier compris entre 0 et `RAND_MAX`), contenues toutes les deux dans `stdlib`. La fonction `time` est, elle, contenue dans `time.h`.

Remarque : vous avez besoin des constantes `FENETRE_LARGEUR` et `FENETRE_HAUTEUR` définies dans `graphisme`, donc il est nécessaire d'inclure cette bibliothèque dans `disque`.

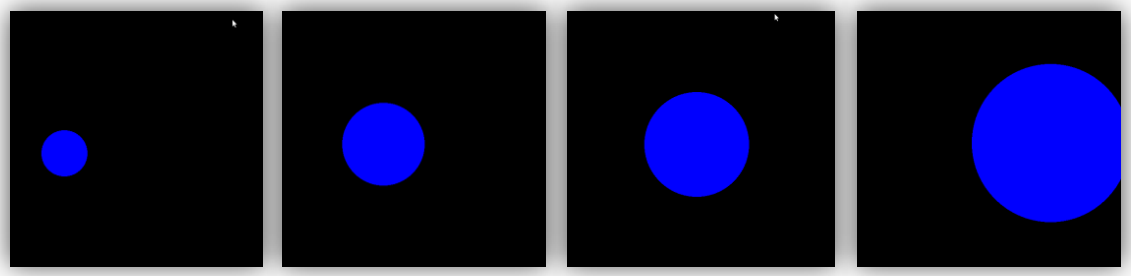
Exercice 7

Configurez votre projet pour utiliser la SDL. Notez que la bibliothèque `graphisme` contient une fonction `dessine_disque`, permettant d'afficher un disque en précisant la position de son centre et la couleur à utiliser.

À l'aide de cette fonction, écrivez dans `disque` une fonction `void anime_disque()` qui affiche un disque traversant l'écran de la gauche vers la droite, en augmentant progressivement son diamètre.

Remarque : vous devez utiliser la fonction `attends_delai(temps)` pour ralentir l'exécution, où `temps` est une durée exprimée en ms. Pensez aussi à utiliser les fonctions des exercices précédents. Pensez aussi à utiliser `rafraichis_fenetre` quand c'est nécessaire.

exemple :



3 Guirlandes

Exercice 8

On veut maintenant représenter une *guirlande* en utilisant une liste de disques. Modifiez d'abord la bibliothèque `liste_s` fournie avec ce sujet, afin qu'elle manipule des disques au lieu d'entiers :

- Changez son nom en `liste_s_disque` ;
- Modifiez la structure `element_s` afin qu'elle contienne un champ `t_disque d` à la place du champ `int valeur` ;
- Adaptez les fonctions de cette bibliothèque de façon appropriée, aussi bien leur en-tête dans le fichier `.h` que leur corps dans le fichier `.c`.

Créez ensuite une nouvelle bibliothèque `guirlande`, qui utilise `disque`.

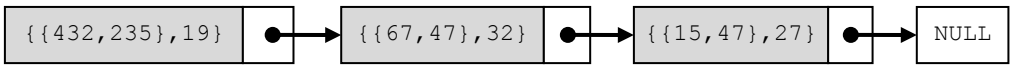
Exercice 9

Dans `guirlande`, écrivez une fonction `int genere_guirlande(liste_s* l, int n)`, qui génère aléatoirement une liste de disques de longueur `n`, grâce à la fonction `genere_disque`. La fonction doit renvoyer -1 en cas d'erreur et 0 en cas de succès.

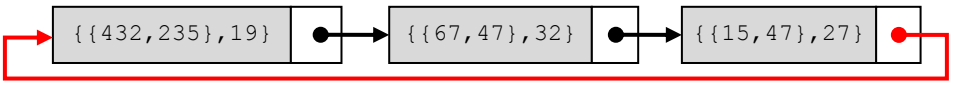
De plus, la fonction doit modifier la liste obtenue de manière à la *refermer* : le dernier élément de la liste doit pointer sur le premier élément (au lieu de pointer sur `NULL`).

exemple :

- Une liste obtenue pour `n=3`, telle qu'elle est avant d'être refermée :



- La même liste, une fois qu'elle a été refermée :

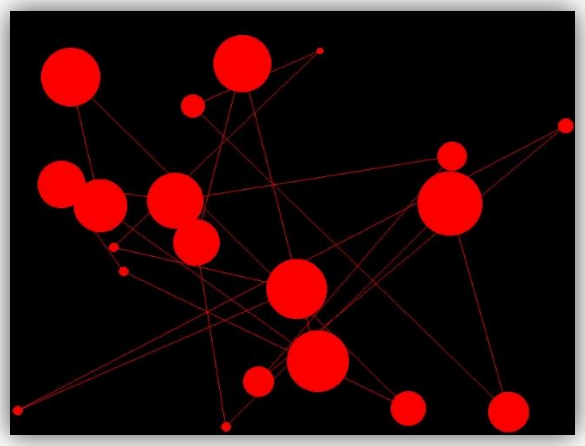


Exercice 10

Dans `guirlande`, écrivez une fonction `void dessine_guirlande(liste_s l, Uint32 coul)`, qui dessine chaque disque contenu dans la liste, en reliant les centres de chaque paire de disques consécutifs par un segment. Les disques et les segments doivent être dessinés dans la couleur passée en paramètre.

Remarque : vous devez placer le premier élément de la guirlande dans une variable spécifique, afin de pouvoir déterminer quand vous avez fait le tour de toute la liste.

exemple : vous devez obtenir un affichage de la forme suivante, ici pour 20 disques :



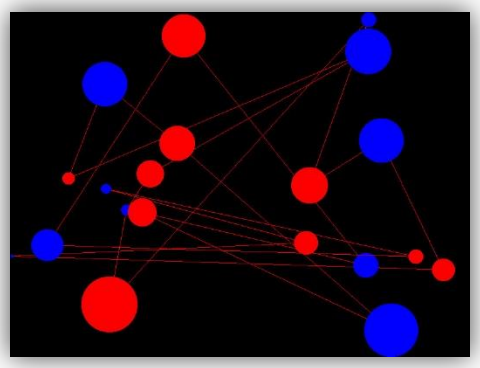
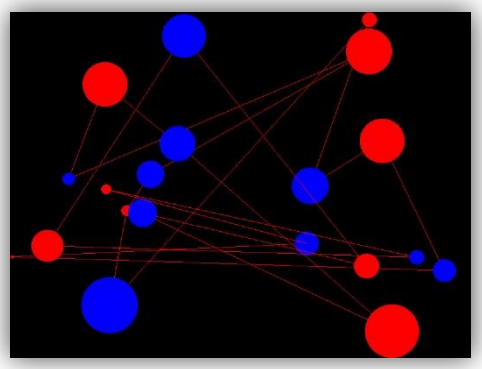
Exercice 11

Dans `guirlande`, écrivez une fonction `void clignote_guirlande(liste_s l, Uint32 coul1, Uint32 coul2, int k)` qui fait clignoter la guirlande `l`. Un disque sur deux doit être dessiné avec la couleur `coul1`, et l'autre moitié avec `coul2`. Puis, la guirlande doit être redessinée en inversant les couleurs, pour obtenir le clignotement. Les segments doivent toujours rester de couleur `coul1`.

Le paramètre `k` indique le nombre de clignotements à effectuer. Là encore, utilisez `attends_delai` pour contrôler la vitesse de l'animation. N'utilisez pas un délai trop petit, pour éviter les crises d'épilepsie !

Remarque : on supposera que le nombre de disque dans la guirlande est *pair*.

exemple : guirlande de 20 disques, avec un clignotement alternant le rouge et le bleu :

Exercice 12

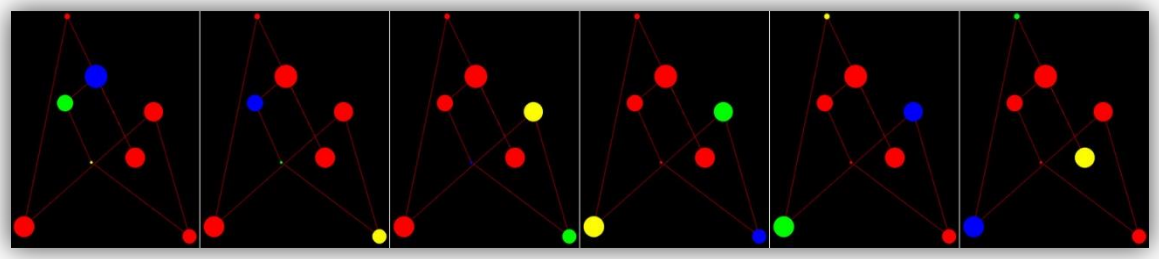
Dans `guirlande`, écrivez une fonction `void anime_guirlande(liste_s l, Uint32 coul, Uint32 couls[], int c, int k)`, qui reçoit une guirlande `l`, une couleur principale `coul` et un tableau de couleurs secondaires `couls`. Le paramètre `c` représente la taille du tableau `couls` (i.e. le nombre de couleurs secondaires).

La fonction doit d'abord dessiner la guirlande `l` en utilisant la couleur `coul`, qui est la couleur principale. Puis, elle doit effectuer le traitement itératif suivant :

- À la première itération, les disques sont coloriés de la façon suivante :
  - Disque numéro 0 en `couls[0]` ;
  - Disque numéro 1 en `couls[1]` ;
  - ...
  - Disque numéro $c - 1$ en `couls[c-1]` ;
  - Disque numéro c et tous les disques suivants en `coul`.
- À la seconde itération :
  - Disque numéro 0 : reprend la couleur principale `coul` ;
  - Disque numéro 1 : `couls[0]` ;
  - Disque numéro 2 : `couls[1]` ;
  - ...
  - Disque numéro c : `couls[c-1]` ;
  - Le reste des disques (après le numéro $c + 1$) est colorié avec `coul`.
- On recommence jusqu'à avoir réalisé `k` itérations.

Remarque : les segments entre les disques doivent toujours être dessinés en utilisant la couleur `coul`.

exemple : 6 premières itérations pour une guirlande de 8 disques rouges, avec les couleurs secondaires bleu, vert et jaune (dans cet ordre) :



Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 44 | listes de caractères

Présentation

On veut créer un programme qui ouvre un premier fichier texte, lit son contenu, passe les minuscules en majuscules, et enregistre le résultat dans un second fichier texte. Le texte sera représenté en mémoire sous la forme d'une liste de caractères, et pour cette raison la bibliothèque `liste_s` est fournie avec le sujet.

1 Préparation

Exercice 1

Adaptez la bibliothèque `liste_s` fournie avec le sujet, de manière à ce qu'elle permettre de traiter des listes de caractères et non pas des listes d'entiers.

Exercice 2

Écrivez une fonction `void convertis_liste(liste_s l)` prenant une liste en entrée, et échangeant chaque lettre minuscule par la lettre majuscule correspondante.

On peut remarquer que la fonction modifie la liste, et pourtant le paramètre est passé directement (`liste`) et non pas à travers un pointeur (`liste*`). Comment expliquez-vous cette observation ?

Remarque : pensez à utiliser la fonction `initialise_liste_s` pour tester votre fonction.

2 Lecture

Exercice 3

Écrivez une fonction `liste construis_liste(FILE *fp)` reçoit en paramètre un fichier texte `fp` déjà ouvert en lecture. La fonction doit lire le contenu de ce fichier en *mode caractère* et construire une liste en utilisant la fonction `insere_element`. Vous devez l'implémentez sans utiliser aucune chaîne de caractères.

Considérez-vous cette approche comme efficace, notamment en termes de parcours de la liste construite ? Justifiez.

Remarque : pour tester votre fonction, vous devez créer un fichier texte.txt à placer dans la racine de votre projet, dans lequel vous copiez-collez n'importe quel texte.

Exercice 4

Proposez une autre version plus efficace de `construis_liste`, que vous appellerez `construis_liste2`. Vous ne devez toujours pas utiliser de chaîne de caractères, mais cette fois vous ne devez pas utiliser `insere_element` non plus.

Expliquez en quoi cette version est plus efficace que la précédente.

3 Écriture

Exercice 5

Écrivez une fonction `void ecris_liste(liste_s l, FILE *fp)` qui reçoit en paramètre une liste de caractères `l` et un fichier `fp` déjà ouvert en écriture. La fonction doit écrire les caractères dans le fichier, en utilisant le *mode caractère*.

Exercice 6

Écrivez une fonction `void copie_fichier(char *source, char *cible)`, qui crée le fichier `cible`, qui est une copie du fichier `source`, en précédant de la façon suivante :

- Lire le fichier `source` et créer une liste de caractères représentant son contenu ;
- Convertir ces caractères en majuscules ;
- Enregistrer la liste obtenue dans le fichier `cible`.

Vous devez utiliser le plus possible les fonctions déjà écrites. Penser à refermer les fichiers, une fois que vous n'avez plus besoin d'y lire ou d'y écrire.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 45 | tapis de Sierpiński

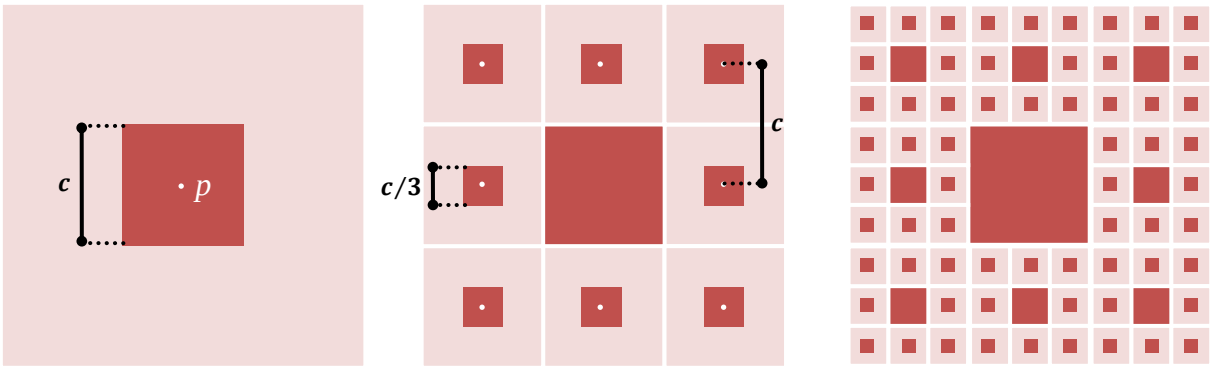
Présentation

Dans ce TP, vous allez écrire un programme qui trace des tapis de Sierpiński de façons récursive et itérative. Pour cela, vous utiliserez différentes bibliothèques connues : `graphisme`, qui permet de dessiner avec la SDL, et `liste_s` qui permet de manipuler des listes simplement chaînées. Vous devez donc configurer votre projet pour utiliser la SDL, puis y copier les fichiers contenus dans l'archive fournie avec ce sujet. Celle-ci comporte également un dossier `exemples` contenant des exécutables montrant la construction de la figure de façons récursive et itérative.

Vous remarquerez que la bibliothèque `liste_s` a été modifiée pour manipuler des valeurs de type `point` (et non pas des `int`). Ces points sont des structures définies par une abscisse (`x`), une ordonnée (`y`) et une couleur (`coul`). La fonction `affiche_point` permet d'afficher sous forme de texte les coordonnées du point `p` passé en paramètre. Cette fonction ne sera pas utilisée directement pour tracer la figure, mais elle vous sera utile pour tester vos fonctions.

1 Définition

Le tapis de Sierpiński est une figure définie par le mathématicien polonais Wacław Sierpiński en 1916.



étape 1 *étape 2* *étape 3*

Cette figure peut se construire à partir de différentes formes géométriques. Pour simplifier le traitement, nous utiliserons des carrés. La construction se décompose en plusieurs étapes. A la première étape, un carré de côté c et de centre p est tracé. A la deuxième étape, 8 carrés de côté $c/3$ sont tracés autour du carré initial. A la troisième étape, 8 carrés de côté $c/9$ sont tracés autour de chacun des 8 carrés de la deuxième étape. Le traitement peut continuer ainsi indéfiniment.

2 Tracé de carrés

Exercice 1

Écrivez une fonction `trace_carre(point p, int cote)` qui dessine un carré dont `p` est le *centre* et `cote` est la longueur du côté (en pixels). Vous devez utiliser uniquement la

fonction `allume_pixel` pour dessiner. De plus, les contours et l'intérieur du carré doivent être de la même couleur que `p`.

Exercice 2

En utilisant `dessine_carre`, écrivez une fonction `dessine_carres(liste_s l, int cote)` qui, pour chaque point `p` contenu dans la liste `l`, trace un carré de centre `p` et de côté `cote`.

3 Version récursive

La version récursive consiste à :

- Dessiner le carré de centre p et de coté c ;
- Calculer les centres des 8 carrés qui l'entourent ;
- Recommencer le traitement pour chacun d'eux, avec un coté $c/3$.

En théorie, on peut continuer le traitement à l'infini. Mais le carré le plus petit que l'on peut tracer sur l'écran d'un ordinateur est un pixel. Ceci constituera donc notre *cas d'arrêt*.

Exercice 3

Écrivez une fonction `int calcule_liste_centres(point p, int cote, liste_s *l)`, qui reçoit en paramètres un point `p` et une liste vide `l`. Elle doit calculer les centres des 8 carrés situés autour du carré de centre `p` et de côté `cote`, puis les placer ces centres dans la liste. De plus, en cas d'erreur, la fonction renvoie -1, et sinon elle renvoie 0.

Exercice 4

Écrivez une fonction `int trace_sierpinski_rec(point p, int cote)` qui implémente récursivement la construction d'un tapis de Sierpiński. Le paramètre `p` correspond au centre du premier carré de la figure, et `cote` est le côté de ce carré. Vous aurez besoin des fonctions `calcule_liste_centres` et `trace_carre`. En cas d'erreur, la fonction renvoie -1, sinon elle renvoie 0.

4 Version itérative

L'algorithme pour la version itérative repose sur l'utilisation d'une liste contenant tous les points restant à traiter. Pour chacun de ces points :

- On trace le carré correspondant ;
- On calcule les 8 points qui l'entourent ;
- Ces points sont placés dans la liste qui servira pour l'itération suivante.

Lors de la première itération il n'y a qu'un seul point dans la liste, puis il y en a 8 lors de la deuxième itération, 64 lors de la troisième, etc. Ici aussi, la condition d'arrêt porte sur la dimension du côté des carrés.

Exercice 5

Écrivez une fonction `int calcule_liste_tous_centres(liste_s *l1, liste_s *l2, int cote)`. La liste `l1` est une liste de points, et la liste `l2` est une liste vide. Les points de `l1` sont tous des centres de carrés de même côté `cote`. À la fin du traitement, la liste `l2` doit contenir les centres de tous les carrés entourant ceux de `l1`. Pour chaque point de `l1`, la fonction doit donc calculer les 8 points qui l'entourent, et les placer dans `l2`. Pour cela, vous devez utiliser `calcule_liste_centres`. En cas d'erreur, la fonction renvoie -1, sinon elle renvoie 0.

Exercice 6

Écrivez une fonction `int trace_sierpinski_it(point p, int cote)` qui implémente itérativement la construction d'un tapis de Sierpiński. Le paramètre `p` correspond au centre du premier carré de la figure, et `cote` est la dimension de ce carré. Vous aurez

besoin des fonctions `trace_carres` et `calcule_liste_tous_centres`. En cas d'erreur, la fonction renvoie -1, sinon elle renvoie 0.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 46 | suite de Syracuse

Présentation

Le but de ce TP est d'effectuer différents calculs sur la suite de Syracuse, en manipulant des listes doublement chaînées. La bibliothèque `liste_d` est donc fournie avec ce sujet.

La dernière partie consiste à donner une représentation graphique de certains résultats obtenus, et pour cela nous utiliserons la SDL et la bibliothèque `graphisme`. Cependant, ne configurez pas encore votre projet pour l'utilisation de la SDL, afin de pouvoir bénéficier de la sortie texte de votre programme. Lorsque la SDL sera nécessaire, cela sera indiqué dans l'énoncé de l'exercice concerné, et vous pourrez alors configurer votre projet de façon appropriée.

1 Définition

On définit la *suite de Syracuse* de terme initial $u_0 = k$ ($k \in \mathbb{N}^*$), de la manière suivante :

$$\forall i \geq 0, u_{i+1} = \begin{cases} u_i/2 \text{ si } u_i \text{ est pair} \\ 3u_i + 1 \text{ si } u_i \text{ est impair} \end{cases}$$

D'après la *conjecture de Syracuse*, quelle que soit la valeur initiale u_0, il existe un rang j tel que $u_j = 1$. À partir de ce rang j, la suite devient cyclique de cycle 1, 4, 2. Cette valeur j est appelée *temps d'arrêt* de la suite (pour le terme initial considéré).

exemple : pour $n = 5$, les termes de la suite sont : $u_0 = 5$, $u_1 = 16$, $u_2 = 8$, $u_3 = 4$, $u_4 = 2$, $u_5 = 1$, $u_6 = 4$, $u_7 = 2$, $u_8 = 1$, etc. Donc pour ce terme initial, on a $j = 5$.

Remarque : une conjecture est une propriété qui n'a pas été démontrée, mais pour laquelle on n'a jamais trouvé de contre-exemple, et que l'on suppose donc (temporairement) vraie.

Exercice 1

Dans ce TP, nous aurons tout particulièrement besoin d'insérer des éléments en fin de liste. Écrivez une fonction `insere_element_fin_d` qui insère un élément `e` à la fin d'une liste *doublement* chaînée `l`, *sans* utiliser la fonction `insere_element_d`.

2 Implémentation itérative

Exercice 2

Écrivez une fonction *itérative* `int calcule_syracuse_it1(int u0, liste_d *l)` qui prend en paramètre le terme initial `u0` et une liste vide `l`, et qui remplit cette liste avec les termes de la suite de Syracuse allant de u_0 à u_j. La position d'un terme dans la liste doit correspondre à sa position dans la suite. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

exemple : pour `u0=5`, la fonction renvoie la liste `5,16,8,4,2,1`.

Exercice 3

Faites une copie de la fonction précédente, renommez-la `calcule_syracuse_it2`, et effectuez les modifications nécessaires pour qu'elle renvoie (en plus de la liste et du code d'erreur) :

- Le plus grand terme m rencontré au cours du traitement ;
- Le temps d'arrêt, i.e. le rang j du premier terme égal à 1.

exemple : pour `u0=5`, en plus de la liste, la fonction renverra `m=16` et `j=5`.

3 Implémentation récursive

Exercice 4

Écrivez une fonction `int calcule_syracuse_rec1(liste_d *l)` qui effectue le même traitement que `calcule_syracuse_it1`, mais récursivement. Contrairement à `calcule_syracuse_it1`, on supposera que la liste `l` n'est pas vide. Donc le dernier terme de la liste correspond au dernier terme de la suite ayant été calculé *pour l'instant*. Par exemple, si la liste contient les valeurs 5, 16, 8, alors on sait que le prochain terme à calculer est celui qui suit 8.

Écrivez aussi la fonction `int calcule_syracuse_rec1_init(int u0, liste_d *l)` qui initialise la liste vide `l` avec le terme initial `u0`, puis utilise `calcule_syracuse_rec1` afin de lancer le traitement récursif.

Exercice 5

Modifiez les deux fonctions de l'exercice précédent, de manière à obtenir les fonctions `calcule_syracuse_rec2` et `calcule_syracuse_rec2_init`, capables de renvoyer m et j.

4 Représentation graphique

Exercice 6

On s'intéresse maintenant à la distribution du temps d'arrêt j par rapport aux termes de départ. Écrivez une fonction `int vide_liste_d(liste_d *l)` qui reçoit une liste `l` (supposée contenir des éléments) et supprime tous ses éléments en utilisant `supprime_element_d`. La fonction renvoie -1 en cas d'erreur, et 0 en cas de succès.

Exercice 7

Écrivez une fonction `int calcule_temps(liste_d *l, int k)` qui utilise l'une des fonctions `calcule_syracuse` pour calculer j pour toutes les valeurs de u_0 comprises entre 2 et k. Autrement dit, on calcule j pour $u_0 = 2$, puis pour $u_0 = 3$, puis $u_0 = 4$, etc., jusqu'à $u_0 = k$.

Chacune de ces valeurs doit être stockée dans la liste l passée en paramètre, qui est initialement vide. Après l'exécution de la fonction, le tout premier élément de la liste correspond donc au temps d'arrêt obtenu pour $u_0 = 2$, l'élément suivant au temps d'arrêt obtenu pour $u_0 = 3$, et ainsi de suite.

exemple : pour `k=100`, on obtient la liste suivante :

```
1 7 2 5 8 16 3 19 6 14 9 9 17 17 4 12 20 20 7 7 15 15 10 23 10 111 18 18 18 106
5 26 13 13 21 21 21 34 8 109 8 29 16 16 16 104 11 24 24 24 11 11 112 112 19 32
19 32 19 19 107 107 6 27 27 27 14 14 14 102 22 115 22 14 22 22 35 35 9 22 110
110 9 9 30 30 17 30 17 92 17 17 105 105 12 118 25 25 25
```

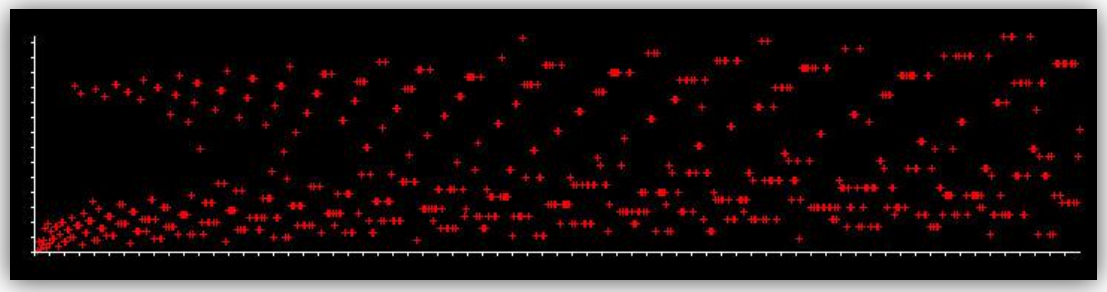
Remarque : pensez à utiliser `vide_liste_d` pour supprimer la liste calculée par `calcule_syracuse` (et non pas la liste résultat de `calcule_temps`) à chaque itération, sinon vous allez progressivement remplir la mémoire, et provoquer une erreur d'exécution (que l'on appelle une *fuite de mémoire*).

Exercice 8

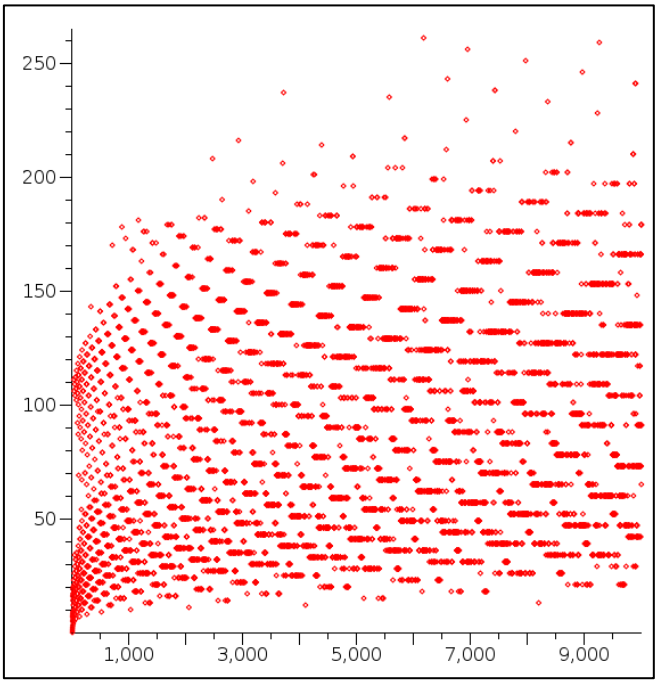
Finalement, configurez votre projet pour utiliser la SDL et la bibliothèque `graphisme`. Tracez un graphique représentant en abscisse les termes initiaux u_0 et en ordonnée les temps d'arrêt correspondant.

Pour simplifier le travail, on utilisera une échelle de $1:1$, i.e. un pixel dans le graphique représente une différence de 1 entre les u_0 ou j. Le graphique doit être placé au centre de la fenêtre graphique, à la fois horizontalement et verticalement. Des traits doivent être tracés sur les axes toutes les 10 unités (donc tous les 10 pixels, puisque l'échelle est $1:1$).

exemple : pour les 700 premiers temps d'arrêt, on obtient le graphique suivant :



Pour référence, le même graphique donné pour les 10000 premiers temps d'arrêt (tiré de la page Wikipedia).



Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 47 | enveloppe d'un nuage de points

Présentation

Ce TP porte sur le calcul d'enveloppes géométriques, et leur tracé grâce à la SDL. Vous devez donc configurer votre projet de manière à pouvoir utiliser la SDL. Vous avez également besoin de la bibliothèque `math`, qui peut nécessiter une configuration spéciale si vous travaillez sous Linux.

1 Préparation

Le sujet est fourni avec une archive contenant les bibliothèques `graphisme` et `liste_dbl` déjà utilisées précédemment.

À noter que `graphisme` contient deux fonctions particulièrement utiles pour ce TP :

- `void genere_point(int *x, int *y)` : permet de générer aléatoirement les coordonnées (x, y) d'un point. Ces coordonnées sont, bien entendu, renvoyées par adresse.
- `void dessine_point(int x, int y, Uint32 coul)` : permet de dessiner un point de coordonnées (x, y), sous la forme d'une croix.

La bibliothèque `liste_d` contient également une fonction nécessaire à ce TP :

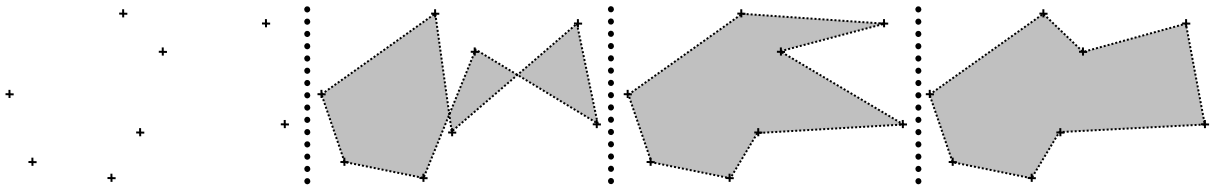
- `void trie_liste_decrois_d(liste *l)` : permet de trier une liste dans l'ordre décroissant.

Dans la bibliothèque `math`, vous aurez notamment besoin de la fonction suivante :

- `double acos(double a)` : permet de calculer un angle (exprimé en radians) à partir de son cosinus.

2 Définition

On dit qu'un polygone est **croisé** s'il possède au moins deux côtés sécants. L'**enveloppe** quelconque d'un nuage de points est un polygone non-croisé dont les sommets sont les points du nuage.



un nuage de points | *un polygone croisé construit sur le nuage* | *une enveloppe du nuage* | *une autre enveloppe du même nuage*

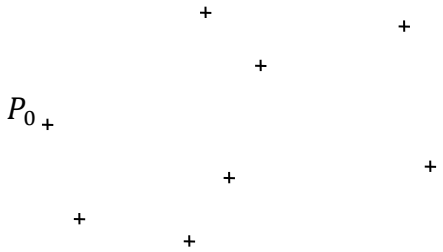
L'objectif de ce TP est de tracer l'une des enveloppes possibles pour un nuage de points généré aléatoirement.

3 Calcul de l'enveloppe

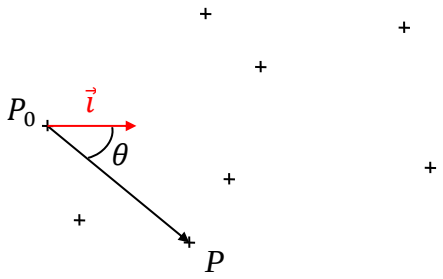
Déterminer une enveloppe consiste à trier les points afin d'obtenir un chemin fermé, non-sécant, et joignant chacun des points du nuage.

Algorithme :

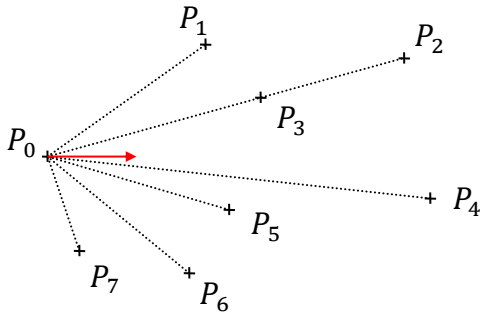
1. Soit P_0 le point le plus à gauche de l'écran (si plusieurs points sont sur la même colonne, P_0 est le point le plus haut).



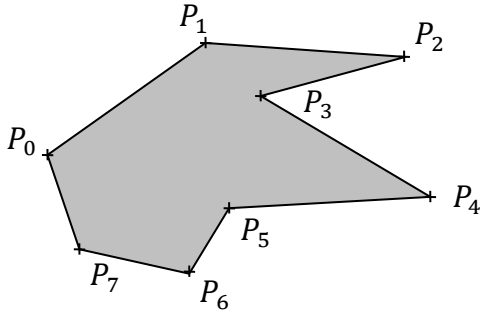
2. Pour chaque point P du nuage de points, on calcule l'angle $\theta = \left(\widehat{\overrightarrow{P_0P}, \vec{\imath}}\right)$, où $\vec{\imath}$ désigne un vecteur directeur de l'axe horizontal. On a la propriété $\theta \in \left]-\frac{\pi}{2}, \frac{\pi}{2}\right[$.



3. On trie les points (sauf P_0) par θ décroissants. Si deux points on le même θ, on considère que le plus haut est le plus grand. On obtient donc une liste ordonnée des points : $(P_0, P_1, \dots, P_n)$.



4. On trace les côtés du polygone $(P_0, P_1, \dots, P_n)$.



4 Implémentation

Exercice 1

On veut représenter la liste de points en utilisant une liste chaînée. Un élément de cette liste sera caractérisé par :

1. Des coordonnées *x* et *y* (exprimées en pixels)
2. Un angle θ (exprimé en radians)

On ne va donc plus manipuler une liste d'entiers basiques, comme c'était le cas précédemment (par exemple dans le TP sur les listes chaînées).

Vous devez adapter la bibliothèque `liste_d` aux nouvelles données :

- Modifiez la structure de données `element_d`.
- Modifiez les en-têtes des fonctions `cree_element_d` et `affiche_liste_d` dans `liste_d.h`
- Modifiez les corps des fonctions dans `liste_d.c`.

Exercice 2

Dans `main.c`, écrivez une fonction `int genere_nuage(int n, liste_d *l)` qui crée un nuage de points sous la forme d'une liste de n points générés aléatoirement (en utilisant `genere_point`). Les θ seront initialisés à 0. La fonction renverra 0 en cas de succès et -1 en cas d'erreur.

Exercice 3

Écrivez une fonction `trace_nuage(liste_d l , Uint32 coul)` qui dessine les points contenus dans la liste passée en paramètre, en utilisant la couleur `coul`. Pensez à vous servir de `dessine_point`.

Exercice 4

Écrivez une fonction `element_d* extrais_P0(liste_d *l)` qui :

1. Calcule quel est le point de la liste qui correspond à P_0 ;
2. Enlève ce point de la liste, mais *sans supprimer* l'élément correspondant ;
3. Et renvoie un pointeur sur l'`element_d` correspondant à P_0.

Testez votre fonction en affichant d'une autre couleur le point P_0.

Attention :

- N'oubliez pas que la liste est doublement chaînée.
- N'*utilisez pas* la fonction `supprime_element_d`, car elle désallouera P_0.

Exercice 5

Écrivez une fonction `float calcule_angle(int v_x1, int v_y1, int v_x2, int v_y2)` qui calcule l'angle $\left(\widehat{\overrightarrow{v_1}, \overrightarrow{v_2}}\right)$ exprimé en radians, $\overrightarrow{v_1}$ et $\overrightarrow{v_2}$ ayant pour coordonnées respectives (x_1, y_1) et (x_2, y_2).

Rappels :

- Calcul du cosinus : $cos\left(\widehat{\overrightarrow{v_1}, \overrightarrow{v_2}}\right) = \frac{\overrightarrow{v_1} \cdot \overrightarrow{v_2}}{|\overrightarrow{v_1}||\overrightarrow{v_2}|}$ où $|\vec{u}|$ représente la norme de $\vec{u}$.
- Calcul du produit scalaire (dans une base orthonormale) : $\overrightarrow{v_1} \cdot \overrightarrow{v_2} = x_1 x_2 + y_1 y_2$.

Attention : l'angle obtenu n'est **pas orienté** (au sens trigonométrique).

Exercice 6

Écrivez une fonction `void calcule_theta(liste_d *l, element_d p0)` qui calcule (grâce à `calcule_angle`) le θ de chaque point P, en fonction de P_0. N'oubliez pas d'orienter l'angle renvoyé par `calcule_angle`, suivant la position de P relativement à P_0 (P est-il au-dessus ou au-dessous de P_0 ?).

Exercice 7

Dans `liste_d.c`, complétez la fonction `int compare_elements_d(element_d e1, element_d e2)` dont le rôle est de comparer deux éléments `e1` et `e2`, et de renvoyer un entier positif si $e_1 > e_2$, négatif si $e_1 < e_2$, ou nul sinon.

Remarque : la relation d'ordre entre les éléments a été décrite dans le principe de calcul de l'enveloppe.

Exercice 8

Dans `main.c`, écrivez une fonction `calcule_enveloppe(liste_d *l)` qui ordonne la liste `l` passée en paramètre, de manière à obtenir son enveloppe. Vous utiliserez la fonction `trie_liste_decrois_d` de la bibliothèque `liste_d`.

Remarque : pensez à rajouter dans la liste le point P_0, qui avait été précédemment retiré.

Exercice 9

Dans `main.c`, écrivez une fonction `trace_polygone(liste_d l, Uint32 coul)` qui dessine le polygone (fermé) constitué des points contenus dans une liste `l` quelconque passée en paramètre. Utilisez les fonctions `attends_delai` et `rafraichis_fenetre` de manière à ce que le polygone soit dessiné progressivement (un côté à la fois).

Exercice 10

Dans la fonction `main`, après avoir testé chaque fonction séparément, ajoutez un programme qui :

1. Initialise le nuage de points ;
2. Affiche ce nuage ;
3. Calcule son enveloppe ;
4. Affiche cette enveloppe sous la forme d'un polygone fermé.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 48 | marches de Graham & Jarvis

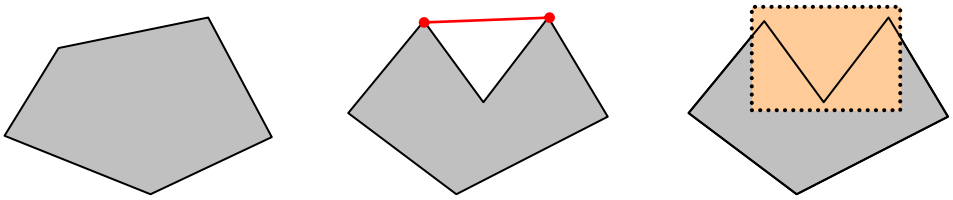
Présentation

Ce TP est la suite directe du TP sur l'*enveloppe d'un nuage de points*. Il est basé sur l'utilisation de fonctions développées au cours de ce TP. Par conséquent, comme lui il nécessite que votre projet soir configuré de manière à pouvoir utiliser la SDL et la bibliothèque `math`.

Le sujet est fourni avec une archive contenant les bibliothèques `graphisme` et `liste_d` déjà utilisées précédemment, ainsi que la bibliothèque `enveloppe`, qui contient les fonctions du TP sur l'enveloppe d'un nuage de points.

1 Définitions

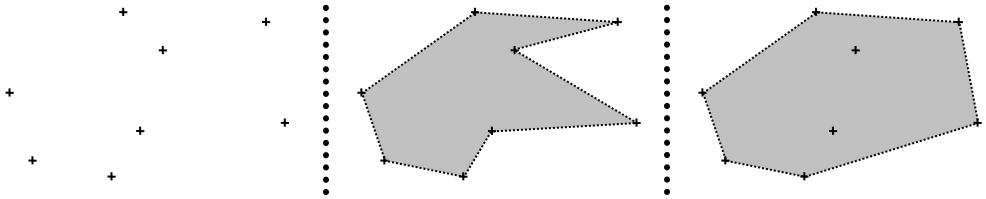
Un polygone est dit *convexe* si toutes ses diagonales se trouvent à l'intérieur du polygone. S'il existe au moins une diagonale qui n'est pas entièrement incluse dans le polygone, on dit que le polygone possède une *concavité*. Dans ce cas là, le polygone est *non-convexe*.



polygone convexe *polygone non-convexe* *concavité*

Lors du TP sur l'enveloppe d'un nuage de points, nous avons calculé l'*enveloppe quelconque* d'un nuage de points. Cette enveloppe correspond à un polygone, qui est en général non-convexe.

L'*enveloppe convexe* d'un nuage de points est le plus petit (pour la relation d'inclusion) polygone convexe contenant tous les points du nuage.



un nuage de points *une enveloppe non-convexe du nuage* *l'enveloppe convexe du nuage*

L'objectif de ce TP est de calculer l'*enveloppe convexe* d'un nuage de points à partir d'une *enveloppe quelconque*.

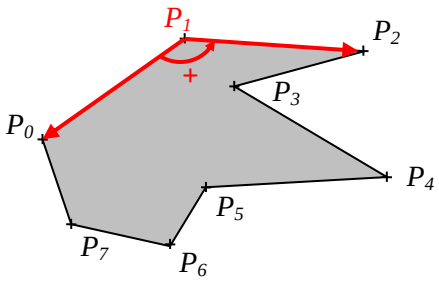
2 Marche de Graham

2.1 Principe

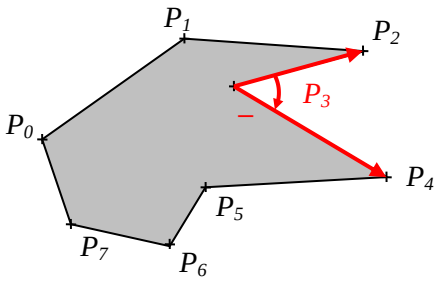
Cet algorithme permet d'obtenir l'enveloppe convexe en supprimant toutes les concavités présentes dans une enveloppe quelconque. Nous prendrons comme point de départ une liste de points ordonnés de manière à constituer une enveloppe quelconque (cf. le TP sur l'*enveloppe d'un nuage de points*).

Pour un point P_i, on peut remarquer la propriété suivante :

- Si P_i correspond à une concavité, alors l'angle $\left(\overrightarrow{P_iP_{i-1}}, \overrightarrow{P_iP_{i+1}}\right)$ appartient à $]-\pi; 0[$.
- Sinon, l'angle $\left(\overrightarrow{P_iP_{i-1}}, \overrightarrow{P_iP_{i+1}}\right)$ appartient à $]0; \pi[$.



point normal



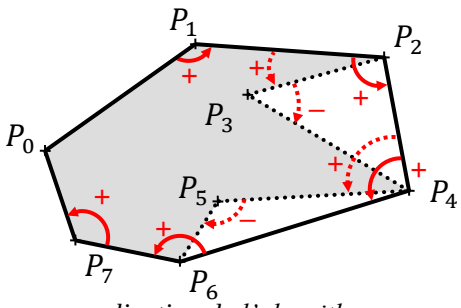
concavité

En calculant le déterminant des deux vecteurs, on peut donc facilement détecter les concavités :

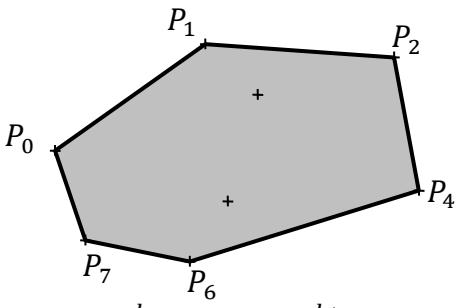
- $\left[det\left(\overrightarrow{P_iP_{i-1}}, \overrightarrow{P_iP_{i+1}}\right) \leq 0\right] \Leftrightarrow \left[\left(\overrightarrow{P_iP_{i-1}}, \overrightarrow{P_iP_{i+1}}\right) \in]-\pi; 0[\right]$: le point P_i est à l'intérieur de l'enveloppe convexe.
- $\left[det\left(\overrightarrow{P_iP_{i-1}}, \overrightarrow{P_iP_{i+1}}\right) > 0\right] \Leftrightarrow \left[\left(\overrightarrow{P_iP_{i-1}}, \overrightarrow{P_iP_{i+1}}\right) \in]0; \pi[\right]$: le point P_i est un sommet de l'enveloppe convexe.

Algorithme :

- Soit $(P_0, P_1, \ldots, P_n)$ la liste ordonnée des points, formant une enveloppe quelconque.
- Pour chaque P_i $(0 < i \leq n)$:
  - Si le déterminant est négatif ou nul :
    - P_i est supprimé de la liste.
    - Si le point précédant P_i est P_0, on passe à P_{i+1}.
    - Sinon on teste à nouveau ce point précédent.
  - Sinon : P_i est conservé, on passe à P_{i+1} (le point suivant).
- À la fin du traitement, les points restants dans la liste sont les sommets de l'enveloppe convexe.



application de l'algorithme



enveloppe convexe obtenue

Remarque : pour P_n, on considérera le couple de vecteurs $\left(\overrightarrow{P_nP_{n-1}}, \overrightarrow{P_nP_0}\right)$.

2.2 Implémentation

Exercice 1

Écrivez une fonction `float calcule_determinant(int v_x1, int v_y1, int v_x2, int v_y2)` qui calcule le déterminant du couple de vecteurs $(\overrightarrow{v_1}, \overrightarrow{v_2})$, $\overrightarrow{v_1}$ et $\overrightarrow{v_2}$ ayant pour coordonnées respectives (x_1, y_1) et (x_2, y_2)

Exercice 2

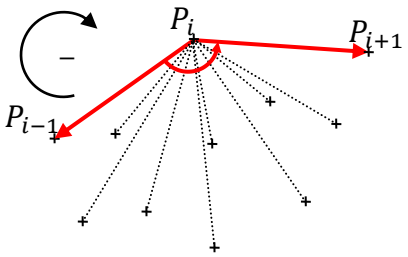
Écrivez une fonction `void marche_graham(liste_d *l)` qui calcule l'enveloppe convexe du nuage de points représenté par la liste `l`. Cette liste est supposée ordonnée de manière à correspondre à une enveloppe quelconque. Vous supprimerez les éléments de `l` en appliquant l'algorithme de Graham.

3 Marche de Jarvis

3.1 Principe

À la différence de la marche de Graham, la marche de Jarvis ne nécessite pas une liste ordonnée des points. Elle repose sur la propriété suivante :

Considérons deux points P_{i-1} et P_i appartenant à l'enveloppe convexe, tels que si on parcourt cette enveloppe dans le sens négatif, P_i est situé juste après P_{i-1}. Alors P_{i+1} (le point suivant P_i dans l'enveloppe) est le point du nuage tel que l'angle $\left(\overrightarrow{P_iP_{i-1}}, \overrightarrow{P_iP_{i+1}}\right)$ est maximal.

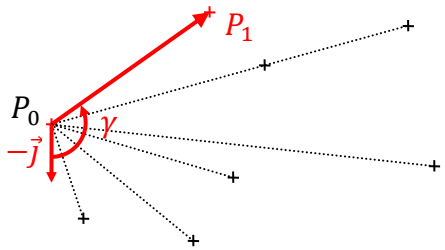


En d'autres termes : quand on connaît un côté de l'enveloppe convexe, il est possible de déterminer le point suivant.

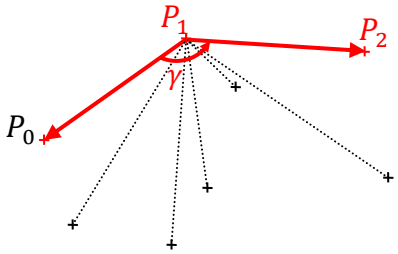
Algorithme :

- *Initialisation :*
  - Soit P_0 le point le plus à gauche et en haut de l'écran (comme précédemment).
  - On note P_1 le point suivant de l'enveloppe convexe. Pour le calculer, on aurait besoin de connaître le côté précédent, mais ce n'est pas le cas. On

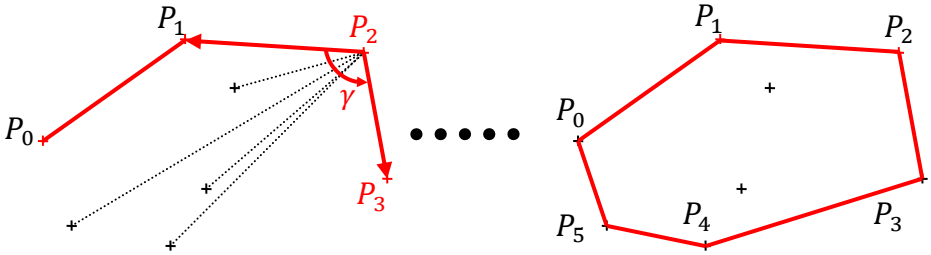
utilise à la place le vecteur $-\vec{j}$ (aucun point ne peut être à gauche de P_0). P_1 est alors le point tel que l'angle $\gamma = \left(-\vec{j}, \widehat{\overrightarrow{P_0P_1}}\right)$ est maximal.



- *Cas général :* P_2 (le point suivant P_1 dans l'enveloppe convexe) est le point tel que l'angle $\gamma = \left(\widehat{\overrightarrow{P_1P_0}, \overrightarrow{P_1P_2}}\right)$ est maximal. Si plusieurs points ont un angle γ maximal (i.e. ces points sont alignés avec P_1), on retient celui qui est le plus éloigné de P_1.



- En appliquant itérativement ce traitement jusqu'à retomber sur P_0, on construit l'enveloppe convexe.



3.2 Implémentation

Exercice 3

Écrivez une fonction `void calcule_gamma(liste_d *l, element_d p1, element_d p2)` qui reçoit une liste de points et deux points de référence P_1 et P_2. La fonction doit mettre à jour le champ `angle` de chaque élément P de la liste, de manière à ce qu'il contienne la valeur $\gamma = \left(\widehat{\overrightarrow{P_2P_1}, \overrightarrow{P_2P}}\right)$.

Exercice 4

Écrivez une fonction `element_d* extrais_suivant(liste_d *l, element_d p2)` qui reçoit une liste d'éléments, et détermine quel est le point (parmi ceux de la liste) suivant dans l'enveloppe, i.e. celui qui a le γ maximal (les γ ayant déjà été calculés grâce à `calcule_gamma`). Il faut également extraire ce point de la liste (sans supprimer l'élément correspondant, bien entendu), et renvoyer son adresse.

Remarques :

- Inspirez-vous de la fonction `extrait_P0` écrite précédemment.
- Le paramètre `p2` est utile en cas d'égalité.

Exercice 5

Écrivez une fonction `element_d* clone_element(element_d e)` qui construit un clone de l'élément `e` passé en paramètre. Un clone est un nouvel élément contenant les mêmes valeurs `x`, `y`, `angle`, `precedent` et `suivant` que `e`. Le clone est renvoyé par adresse.

Exercice 6

Écrivez une fonction `liste_d marche_jarvis(liste_d *l)` qui calcule l'enveloppe convexe du nuage de points représenté par la liste `l`. Dans cette fonction, vous manipulerez deux listes :

- La liste `l2` des points qui appartiennent à l'enveloppe convexe, que vous initialiserez avec P_0 et P_1.
- La liste `l` des autres points, qui n'appartiennent pas (encore) à l'enveloppe convexe.

La liste `l2` sera construite itérativement en appliquant la marche de Jarvis.

Remarque : le point P_0 doit être à la fois dans `l2` (puisqu'il fait partie de l'enveloppe convexe) et dans `l` (puisqu'il est utilisé dans la condition d'arrêt de l'algorithme). Il est impossible que le même élément appartienne à deux listes à la fois, vous devrez donc cloner l'élément représentant P_0.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 49 | analyse d'expressions

Présentation

Le but de ce TP est de manipuler les piles de données vues en cours. Pour cela, on s'intéressera à l'analyse d'expressions représentant des calculs arithmétiques simples.

1 Notation polonaise inversée

La notation la plus répandue pour écrire une expression arithmétique est la notation *infixée* : un opérateur binaire est placé entre ses deux opérandes : $1 + 2$. Cette notation nécessite parfois l'utilisation de parenthèses pour exprimer certains calculs, comme par exemple : $(1 + 2) \times (3 + 4)$.

Il existe également des notations dites *préfixées* et *postfixées*. Avec une notation préfixée, l'opérateur précède ses opérandes $(+1\ 2)$ alors qu'il leur succède si on utilise une notation postfixée $(1\ 2\ +)$.

La *notation polonaise* est une notation préfixée proposée par le mathématicien polonais Jan Łukasiewicz en 1920. Dans cette notation, chaque opérateur possède une arité bien définie : il n'existe pas d'opérateurs homonymes. Par exemple, l'opérateur soustraction (binaire) n'est pas représenté par le même signe que l'opérateur négation (unaire). La notation polonaise offre la particularité de ne pas nécessiter de parenthèses : l'expression infixée $(1 + 2) \times (3 + 4)$ devient : $\times + 1\ 2 + 3\ 4$.

La notation polonaise inversée (NPI) est une version postfixée de la notation polonaise. L'avantage sur la notation polonaise est que l'algorithme qui permet d'évaluer des expressions NPI est plus simple. C'est pour cette raison que les premières calculatrices électroniques (dans les années 60) utilisaient cette notation. Avec la NPI, l'expression précédente prend la forme suivante : $1\ 2 + 3\ 4 + \times$.

2 Méthode d'évaluation

Pour évaluer une expression écrite en NPI, on utilise une pile de données et on parcourt l'expression en partant de la gauche et en traitant chaque *signe*. Ici, le mot *signe* désigne un élément quelconque de l'expression, c'est-à-dire un opérande ou un opérateur.

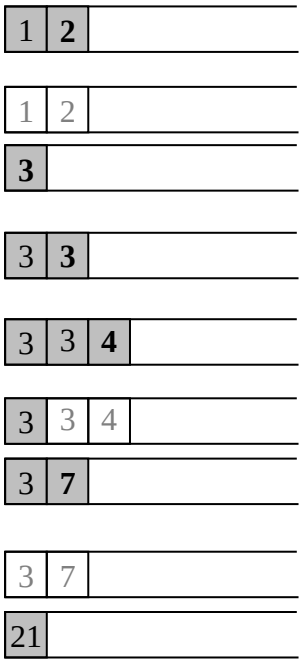
- Tant qu'il reste des signes dans l'expression :
  - Si le signe est un *opérande* :
    - Il est empilé.
  - Si le signe est un *opérateur* :
    - On dépile un nombre d'opérandes correspondant à son arité ;
    - On applique l'opérateur à ces opérandes ;
    - Le résultat est empilé.
- S'il ne reste plus de signe dans l'expression, alors le *sommet* de la pile correspond au *résultat* de l'évaluation.

exemple : évaluation de l'expression $1\ 2 + 3\ 4 + \times$

- **1** $2 + 3\ 4 + \times$:

1	

    - On empile 1.
- 1 **2** + 3 4 +× :
    - On empile 2.
- 1 2 **+** 3 4 +× :
    - On dépile deux opérandes : 2 et 1.
    - On calcule 1 + 2.
    - On empile le résultat 3.
- 1 2 + **3** 4 +× :
    - On empile 3.
- 1 2 + 3 **4** +× :
    - On empile 4.
- 1 2 + 3 4 **+**× :
    - On dépile deux opérandes : 4 et 3.
    - On calcule 3 + 4.
    - On empile le résultat 7.
- 1 2 + 3 4 +**×** :
    - On dépile deux opérandes : 7 et 3.
    - On calcule 3 × 7.
    - On empile le résultat 21.
- 1 2 + 3 4 +× :
    - Il n'y a plus de signe dans l'expression.
    - Le résultat est le sommet : 21.



3 Implémentation de la méthode

Nous manipulerons exclusivement des opérandes entiers. L'expression à évaluer sera représentée par une chaine de caractères pouvant contenir les caractères suivants :

- Les opérateurs :
    - `'+'` (addition).
    - `'-'` (soustraction).
    - `'*'` (multiplication).
    - `'/'` (division entière).
    - `'%'` (reste de la division entière).
- Les chiffres de `'0'` à `'9'`.
- Le caractère espace `' '`.

Le caractère espace est utilisé seulement pour séparer deux opérandes qui ne sont pas déjà séparés par un opérateur. Le caractère espace ne doit apparaître que dans ce cas bien précis. En particulier, les opérandes et les opérateurs ne sont pas séparés par un espace.

exemples : l'expression 1 2 + 3 4 +× :

- est représentée par la chaîne `"1 2+3 4+*"`.
- n'est *pas* représentée par la chaîne `"1 2 + 3 4 + *"`.

L'archive fournie avec ce sujet contient une bibliothèque `pile_liste` correspondant à l'implémentation du type abstrait *pile* au moyen d'une liste. La bibliothèque contient également les fonctions de manipulation des piles vue en cours. L'archive contient aussi la bibliothèque `liste_s` (liste simplement chaînée), qui est utilisée par `pile_liste`.

Exercice 1

Écrivez une fonction `int est_operateur(char c)` qui reçoit un caractère `c`. La fonction doit renvoyer 1 si le caractère passé en paramètre est un opérateur, ou 0 sinon.

exemples :

- Pour `c='1'`, la fonction doit renvoyer 0.
- Pour `c=' '`, la fonction doit renvoyer 0.
- Pour `c='*'`, la fonction doit renvoyer 1.

Exercice 2

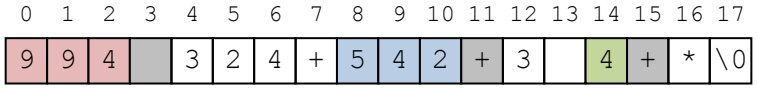
Écrivez une fonction similaire `int est_chiffre(char c)` qui reçoit un caractère et détermine si le caractère est un chiffre.

Exercice 3

Écrivez une fonction `int calcule_entier(char *exp, int *pos)` qui reçoit l'expression `exp` et le numéro `pos` d'un de ses caractères. La fonction doit calculer la valeur du nombre entier dont les chiffres sont compris entre le caractère numéro `pos` (inclus) et le prochain caractère qui ne soit pas un chiffre. Elle doit modifier `pos` de manière à ce qu'il indique ce caractère de séparation. Enfin, elle doit renvoyer la valeur de l'entier.

Remarque : on suppose que le caractère initialement indiqué est un chiffre.

exemples : la fonction reçoit l'expression suivante :



- Pour **`*pos=0`** : le résultat est `994` et `*pos` a la valeur `3` à la fin du traitement.
- Pour **`*pos=8`** : le résultat est `542` et `*pos` prend la valeur `11`.
- Pour **`*pos=14`** : le résultat est `4` et `*pos` prend la valeur `15`.

Exercice 4

Écrivez une fonction `int applique_operateur(char opt, int opd1, int opd2)` qui renvoie le résultat du calcul consistant à appliquer l'opérateur représenté par le caractère `opt` aux deux opérandes `opd1` et `opd2` passés en paramètres.

Remarque : on suppose que le caractère `opt` représente bien un opérateur.

exemple : `applique_operateur('+',2,3)` doit renvoyer la valeur 5.

Exercice 5

En utilisant les fonctions précédentes et les fonctions de manipulation de piles, écrivez une fonction `int evalue_NPI(char *expression, int *resultat)` qui évalue une expression écrite en NPI. Vous appliquerez l'algorithme décrit plus haut.

Le résultat de l'évaluation sera renvoyé par adresse via le paramètre `resultat`. En cas d'erreur, la fonction doit renvoyer -1 par valeur. En cas de succès, elle renvoie 0. Une erreur se produit quand l'expression passée en paramètre ne respecte pas les règles décrites précédemment.

4 Gestion des erreurs

Exercice 6

Écrivez une fonction `void interface_NPI()` qui :

- Demande à l'utilisateur de saisir une expression
- Évalue l'expression
  - En cas de succès : la fonction affiche le résultat.
  - En cas d'erreur : la fonction affiche un message d'erreur.

exemples :

- Une expression correcte :

```
Entrez l'expression NPI a evaluer : 1 2+3 4+*
Le resultat est : 21.
```

- Une expression incorrecte :

```
Entrez l'expression NPI a evaluer : 1 2 +
Erreur lors de l'evaluation.
```

Exercice 7

Le message d'erreur de la fonction précédente est imprécis, ce qui rend difficile la correction de l'éventuelle erreur détectée dans une expression. Faites une copie de `evalue_NPI` que vous appellerez `evalue_NPI_2`, et modifiez-la pour qu'elle renvoie différents codes d'erreur :

- La chaîne représentant l'expression est vide : -6
- La chaîne contient un caractère interdit : -5
- La chaîne contient un espace mal placé : -4
- Il manque un opérateur dans la chaine : -3
- Il manque un opérande dans la chaine : -2
- Une autre erreur survient (par exemple : erreur d'accès mémoire dans une fonction de la pile) : -1
- Il n'y a pas d'erreur : 0

Ces différentes valeurs doivent être définies sous formes de constantes.

Exercice 8

Faites une copie de `interface_NPI` que vous appellerez `interface_NPI_2` et modifiez-la de manière à lui faire afficher un message différent pour chaque code d'erreur possible.

exemple :

```
Entrez l'expression NPI a evaluer : 1 2+3 4+
Erreur : un operateur est manquant.
```

Exercice 9

Faites des copies des fonctions `evalue_NPI_2` et `interface_NPI_2` que vous appellerez respectivement `evalue_NPI_3` et `interface_NPI_3`. Modifiez-les de manière à faire apparaître la position du caractère de la chaîne qui a provoqué l'erreur détectée.

exemple :

```
Entrez l'expression NPI a evaluer : 1 2 +
Erreur : un espace est mal place (pos.3).
```

Exercice 10

Écrivez une fonction `void souligne_erreur(char *expression, int position)` qui affiche l'expression en la soulignant jusqu'à la position passée en paramètre.

exemple : pour l'expression et la position de l'erreur de l'exemple précédent

```
   1 2 +
   ^^^^
```

Exercice 11

Faites une copie de la fonction `interface_NPI_3` que vous appellerez `interface_NPI_4`, et modifiez-la de manière à lui faire afficher l'expression soulignée en cas d'erreur.

exemple :

```
Entrez l'expression NPI a evaluer : 1 2+3 4+
Erreur : un operateur est manquant (pos.8).
   1 2+3 4+
   ^^^^^^^^
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 50 | conversion d'expressions

Présentation

Ce sujet fait suite au TP sur le traitement d'expressions mathématiques écrites en notation polonaise inversée (NPI). Le but est d'approfondir l'utilisation des piles de données ainsi que de la récursivité.

La bibliothèque `npi` fournie avec ce sujet contient l'implémentation des principales fonctions du TP précédent, à utiliser dans ce TP. Les fonctions que vous écrirez au cours de ce TP doivent constituer une nouvelle bibliothèque `infixe`, qui utilise elle-même `npi`. Comme d'habitude, chaque fonction doit être testée dans la fonction `main` du fichier principal `main.c`.

1 Expression infixées

Nous avons vu dans ce TP que la NPI offre l'avantage de faciliter l'évaluation des expressions. Par contre, la NPI ne parait pas très pratique pour une personne habituée à la notation infixée.

Pour bénéficier des avantages propres à chaque notation, nous allons définir des fonctions permettant de convertir une expression infixée en NPI. Afin de simplifier la conversion, nous allons faire des hypothèses quant à la forme des expressions *infixées* à traiter :

- Elles contiennent seulement des caractères représentant :
  - Des chiffres.
  - Des parenthèses.
  - Les opérateurs entiers binaires `'+'`, `'-'`, `'*'`, `'/'` et `'%'`.
- Elles sont *complètement* parenthésées :
  - Tout triplet $op_1 \blacksquare op_2$ constitué d'un opérateur ■ et de ses opérandes op_1 et op_2 est entouré d'une paire de parenthèses : $(op_1 \blacksquare op_2)$.
  - Les parenthèses ne doivent apparaître qu'à cette seule occasion.

exemples :

- Expressions complètement parenthésées :
  - `(1+3)`
  - `(2-((1+3)+5))`
- Expressions *pas* complètement parenthésées :
  - `1+3+5`
  - `(1+3)+5`
  - `1+3`
  - `(1)`
  - `2-1+3`

2 Parenthésage

La première étape consiste à tester si l'expression infixée à traiter est bien complètement parenthésée. On peut utiliser pour cela le principe récursif suivant :

Une expression *exp* est complètement parenthésée si et seulement si $exp = (op_1 \blacksquare op_2)$ avec :

- L'opérande *op1* est soit une expression complètement parenthésée, soit un entier.
- L'opérateur ■ est l'un des opérateurs autorisés.
- L'opérande *op2* est soit une expression complètement parenthésée, soit un entier.

Exercice 1

Dans `infixe`, écrivez une fonction `void passe_entier(char *exp, int *pos)` qui prend en paramètres une expression `exp` et le numéro `pos` d'un caractère de cette expression. Le caractère `exp[*pos]` est le premier chiffre d'un nombre de l'expression. La fonction doit modifier `*pos` de manière à ce que `exp[*pos]` soit le premier caractère suivant ce nombre.

exemple : pour `exp="((12+3456)*78)"` et `*pos=5`

avant d'appliquer la fonction : `*pos=5`

(	(	1	2	+	3	4	5	6	)	*	7	8	)

après avoir appliqué la fonction : `*pos=9`

(	(	1	2	+	3	4	5	6	)	*	7	8	)

Exercice 2

En utilisant le principe récursif proposé ci-dessus, écrivez dans `infixe` une fonction `int est_correcte_rec(char *exp, int *pos)` qui teste si l'expression infixée `exp` est correcte. Le pointeur `pos` représente le numéro du prochain caractère à traiter, par conséquent il devra pointer sur une variable contenant la valeur 0 lors de l'appel initial. La fonction doit vérifier à la fois que l'expression est correctement parenthésée, et qu'elle ne contient que des caractères autorisés. Elle renvoie 1 si l'expression est correcte, et 0 sinon.

exemples :

- Pour l'expression `(1+3)`, la fonction renvoie 1 et donne à `pos` la valeur 5.
- Pour l'expression `1+3`, la fonction renvoie 0 (et la valeur de `pos` n'a plus d'importance)

Exercice 3

On peut remarquer que le principe récursif présenté ne traite pas le cas `"(0+(1+2))+3"` car la vérification de l'expression est arrêtée après la dernière parenthèse fermante. Écrivez dans `infixe` une fonction `int est_correcte(char *exp)` qui utilise `est_correcte_rec` pour vérifier l'expression, puis qui traite ce cas particulier. La fonction renvoie 1 si l'expression est correcte, et 0 sinon.

exemples :

- Pour l'expression `(0+(1+2))+3`, la fonction renvoie 0 ;
- Pour l'expression `((0+(1+2))+3)`, la fonction renvoie 1.

3 Conversion & évaluation

Exercice 4

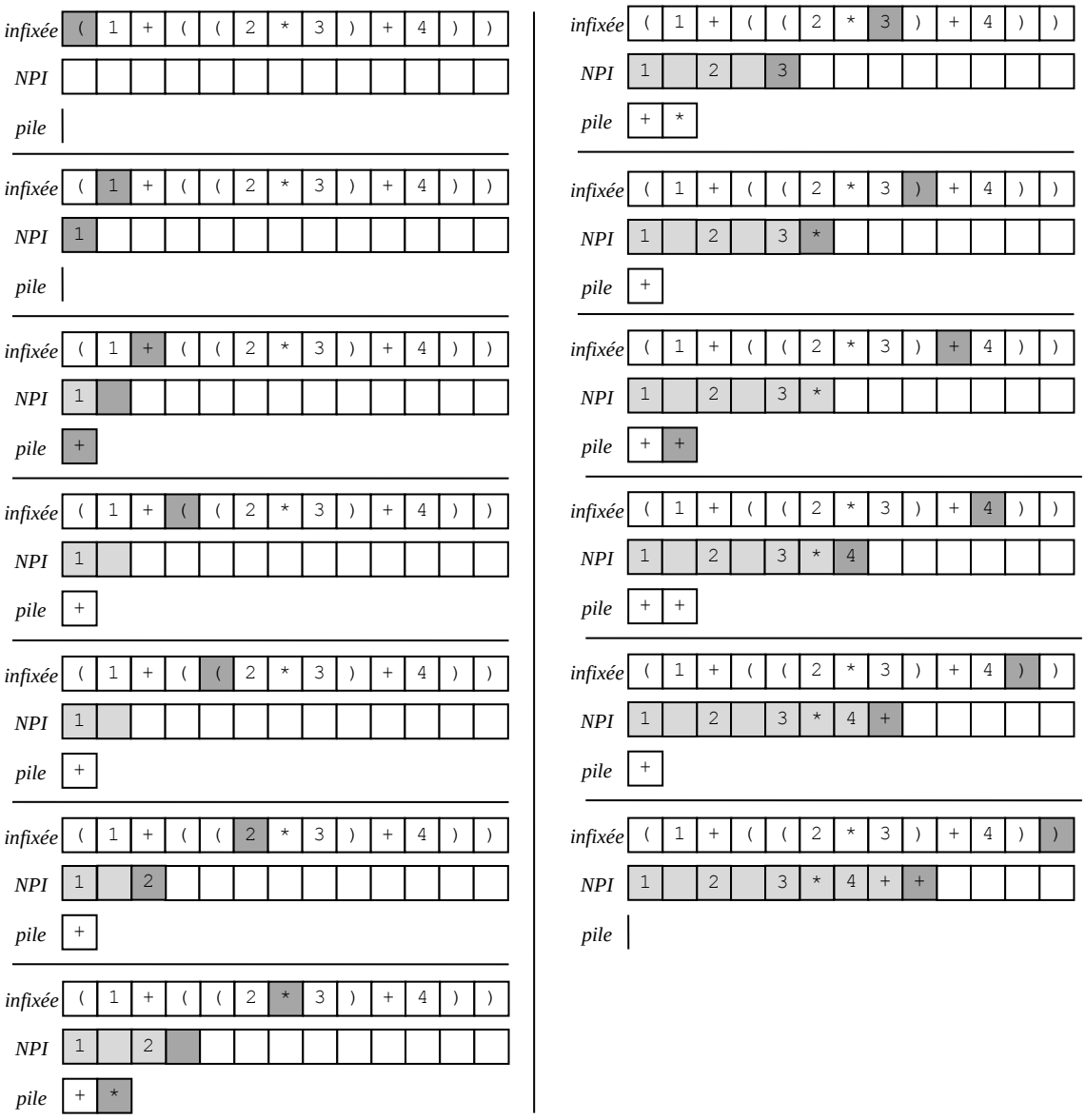
Pour convertir une expression infixée en NPI, on utilise une pile de données et on parcourt l'expression en partant de la gauche.

- Tant qu'il reste des caractères dans l'expression :
  - Si le caractère est un chiffre : il est copié à la fin de la chaîne résultat.
  - Si le caractère est un opérateur :
    - Il est empilé.
    - Si le dernier caractère copié dans la chaîne résultat est un chiffre : un caractère espace `' '` est placé à la fin de la chaîne résultat.

- Si le caractère est une parenthèse ouvrante `'('` : on passe simplement au caractère suivant.
- Si le caractère est une parenthèse fermante `')'` : l'opérateur situé au sommet de la pile est dépilé et copié à la fin de la chaîne résultat.

Remarque : l'utilisation des caractères espace `' '` dans l'expression obtenue doit respecter les contraintes définies dans le TP sur les NPI.

exemple : conversion de `"(1+((2*3)+4))"`



Dans `infixe`, écrivez une fonction `int convertis_infixe_vers_npi(char *exp1, char *exp2)` qui convertit une expression infixée `exp1` en une expression NPI `exp2`. La fonction renvoie 0 en cas de succès et -1 en cas d'erreur.

exemple : l'expression `"(1+((2*3)+4))"` devient `"1 2 3*4++"`.

Exercice 5

Dans `infixe`, écrivez une fonction `int evalue_infixe(char *expression, int *resultat, int *position)` qui reçoit une expression infixée et l'évalue. Le résultat est

renvoyé par adresse via le paramètre `resultat`. La fonction doit réaliser les opérations suivantes :

- Tester si l'expression infixée reçue est complètement parenthésée ;
- La convertir en une expression NPI ;
- Évaluer cette nouvelle expression.

exemple : pour l'expression "`(1+((2*3)+4))`", la fonction renvoie 11.

Le paramètre `position` sert à renvoyer la position de l'erreur, le cas échéant. De plus, la fonction doit renvoyer 0 en cas de succès et un *code spécifique* en cas d'erreur :

- -7 si l'expression originale n'est pas complètement parenthésée ;
- -8 si un problème se produit lors de la conversion
- Autrement : le code d'erreur renvoyé lors de l'évaluation de l'expression NPI.

Remarque : définissez des constantes pour représenter les codes d'erreur.

Exercice 6

Dans `infixe`, copiez la fonction `void interface_npi()` de la bibliothèque `npi`, et renommez-la en `void interface_infixe()`. Modifiez cette fonction de manière à ce qu'elle évalue des expressions infixées, et affiche les messages d'erreur correspondants.

exemples :

- Une expression correcte :

```
Entrez l'expression infixe a evaluer : (1+((2*3)+4))
Le resultat est : 11.
```

- Une expression qui n'est pas complètement parenthésée :

```
Entrez l'expression infixe a evaluer : (1+((2*3)+4)
Erreur : l'expression n'est pas completement parenthesee.
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 51 | détection de palindromes

Présentation

Le but de ce TP est d'approfondir des notions déjà étudiées en cours et lors des TP précédents, en particulier : chaînes de caractères, pointeurs, allocation dynamique, fonctions récursives, piles de données.

1 Notion de palindrome

Un *palindrome* est un mot symétrique, c'est-à-dire que l'on peut le lire dans les deux sens (en partant du début ou de la fin). Exemples de mots constituants des palindromes : rotor, kayak, radar, selles, été… Ce TP porte sur l'implémentation de différents algorithmes permettant de déterminer si une chaîne de caractères donnée est un palindrome.

2 Par renversement

Exercice 1

Écrivez une fonction `inverse_chaine(char *chaine)` qui reçoit en paramètre un pointeur sur une chaîne de caractères `chaine` et construit une nouvelle chaîne correspondant à l'inverse de `chaine`. L'espace mémoire occupé par cette nouvelle variable doit être *minimal*. La fonction doit renvoyer un pointeur vers la nouvelle chaîne, ou la valeur `NULL` en cas d'erreur.

exemple : appliquée à une chaîne `"abcd"`, la fonction doit renvoyer un pointeur sur une chaîne `"dcba"`.

Exercice 2

Écrivez une fonction récursive `compare_chaines(char *chaine1, char *chaine2)` qui compare caractère par caractère les deux chaînes passées en paramètres. La fonction renvoie 1 si les deux chaînes sont identiques et 0 sinon.

exemples :

- L'appel `compare_chaines("abcd","abce")` renvoie 0 ;
- L'appel `compare_chaines("abcd","abcd")` renvoie 1.

Exercice 3

En utilisant les fonctions `inverse_chaine` et `compare_chaines`, écrivez une fonction `int est_palindrome1(char *chaine)` qui détermine si une chaîne de caractères est un palindrome : elle doit renvoyer 1 si la chaîne est un palindrome, et 0 sinon.

exemples :

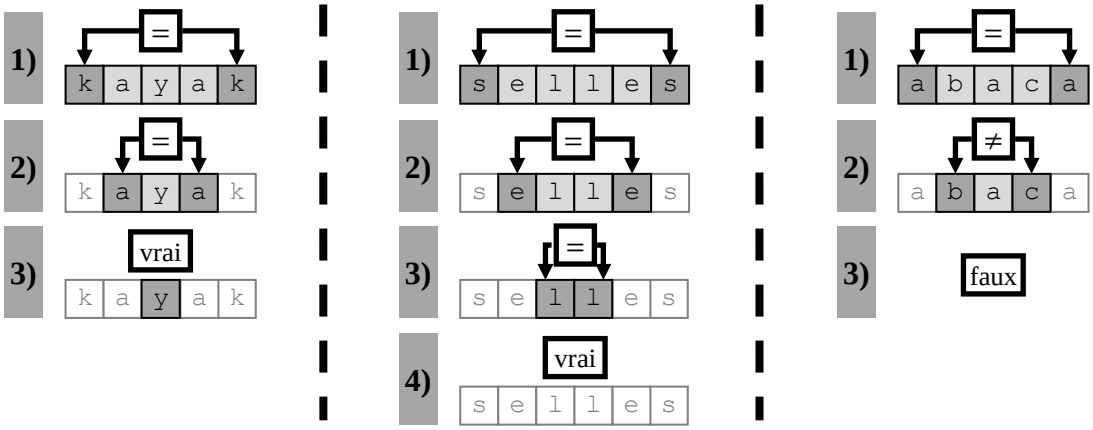
- L'appel `est_palindrome1("abcd","eizyuepzep")` renvoie 0 ;
- L'appel `est_palindrome1("abcd","dcba")` renvoie 1.

3 Par double parcours

Un autre algorithme pour déterminer si une chaîne de caractères est un palindrome consiste à comparer les caractères en partant des deux extrémités (le début et la fin) et en se déplaçant vers le centre :

- On continue à se déplacer vers le centre tant que les caractères sont égaux.
- On s'arrête quand on arrive au centre (c'est un palindrome) ou quand les deux caractères comparés ne sont pas égaux (ce n'est pas un palindrome).

exemples : on considère successivement les chaînes `"kayak"`, `"selles"` et `"abaca"` :



Exercice 4

Écrivez une fonction itérative `int est_palindrome2_it(char *chaine)` qui prend une chaîne de caractères en paramètre et implémente l'algorithme du double parcours pour déterminer s'il s'agit d'un palindrome. La fonction renvoie 1 si le mot est un palindrome et 0 sinon.

Exercice 5

Exprimez l'algorithme précédent sous forme récursive, en précisant les éventuels cas d'arrêt, cas d'erreur et cas général (ou cas généraux).

Écrivez ensuite la fonction récursive `int est_palindrome2_rec_sec(char *debut, char *fin)` qui implémente cet algorithme. Les paramètres `debut` et `fin` pointent respectivement sur le premier et le dernier caractères de la chaîne à traiter.

Exercice 6

Enfin, écrivez la fonction `int est_palindrome2_rec(char *chaine)`, qui initialise les paramètres `debut` et `fin`, puis effectue le premier appel de la fonction `est_palindrome2_rec_sec`.

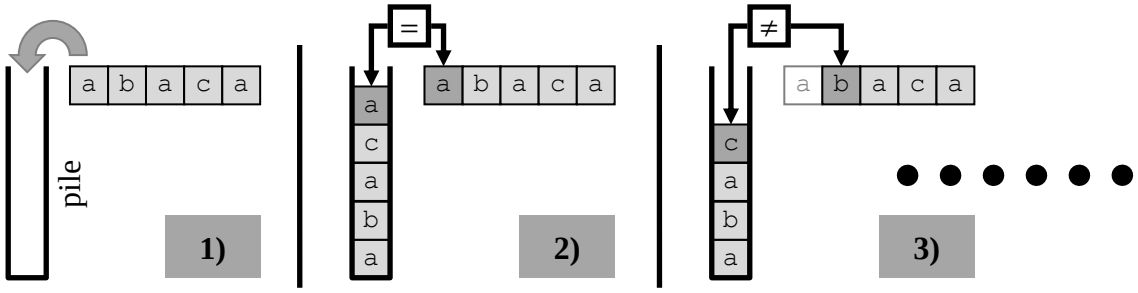
4 Par empilement

Le dernier algorithme utilise une pile de caractères pour effectuer l'analyse. Pour ces exercices, vous disposez de la bibliothèque `pile_liste` définie en cours et utilisée en TP. Elle a été adaptée pour pouvoir traiter des piles de valeurs de type `char`.

La première étape de l'algorithme consiste à empiler le contenu de la chaîne dans la pile, caractère par caractère. La deuxième étape consiste à comparer la chaîne avec le contenu de la pile, caractère par caractère. On considère le premier caractère de la chaîne et le sommet de la pile :

- S'ils sont différents, alors la chaîne n'est pas un palindrome.
- Sinon, on dépile et on recommence avec le caractère suivant dans la chaîne.

- Si la pile est vide, c'est qu'il s'agit d'un palindrome.



Exercice 7

Écrivez une fonction `int initialise_pile(char *chaine, pile *p)` qui reçoit un pointeur sur une pile vide et y recopie les caractères composant la chaîne reçue en paramètre. La fonction renvoie la valeur −1 si une erreur se produit, et 0 sinon.

Exercice 8

En utilisant la fonction `initialise_pile` et l'algorithme présenté ci-dessus, écrivez une fonction `int est_palindrome3(char *chaine)` qui détermine si une chaîne de caractères est un palindrome. La fonction renvoie les mêmes valeurs que les autres fonctions `est_palindromex`.

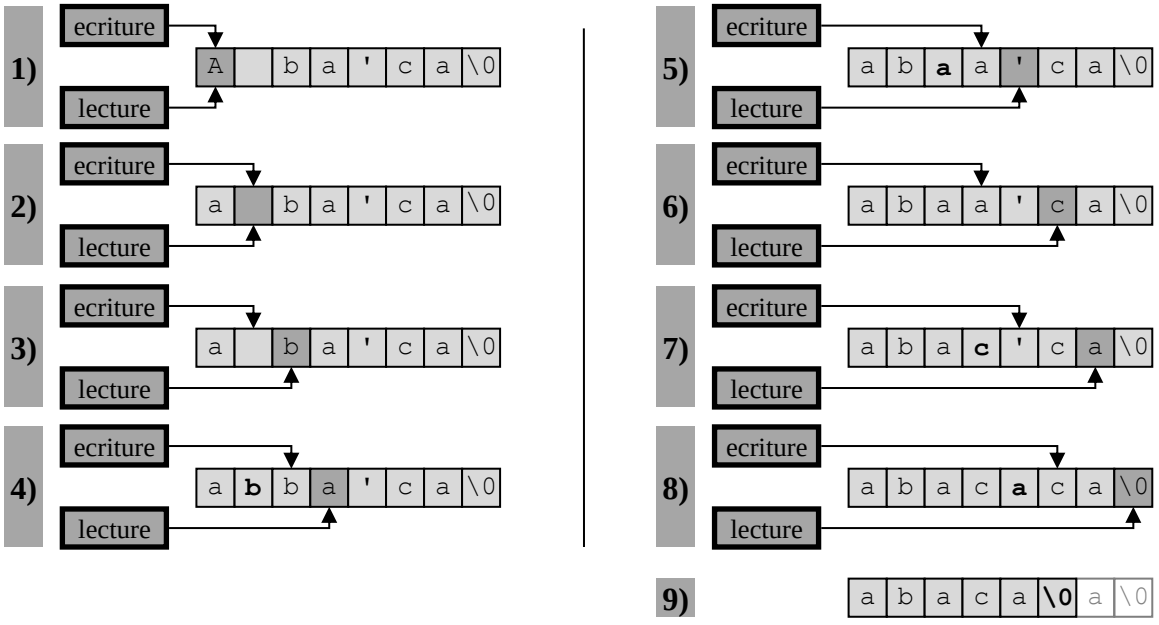
5 Généralisation

On peut généraliser la notion de palindrome à une phrase. Dans ce cas-là, on ignore généralement la ponctuation, les accents et les espaces, par exemple : "Élu par cette crapule !", "Engage le jeu, que je le gagne.", "La mère Gide digère mal."…

Exercice 9

On veut utiliser les fonctions précédentes pour traiter ces palindromes. Pour cela, il suffit de nettoyer les chaines de caractères, de manière à ne garder que les lettres. De plus, les lettres majuscules doivent être transformées en minuscules. On veut effectuer ce traitement en modifiant directement la chaine concernée (i.e. sans passer par un autre tableau).

exemple : sur la phrase `"A ba'ca"` (phrase qui n'a aucune signification)



Écrivez une fonction récursive `void nettoie_chaine(char *lecture, char *ecriture)` qui effectue le nettoyage décrit ci-dessus. Le pointeur `lecture` indique quel est le prochain caractère à traiter, le pointeur `ecriture` indique à quel endroit le prochain caractère devra être recopié.

Exercice 10

Écrivez une fonction `int est_palindrome_phrase(char *phrase)` qui teste si la phrase passée en paramètre est un palindrome, en utilisant les fonctions des exercices précédents.

exemple : pour la phrase "Engage le jeu, que je le gagne !", la fonction doit renvoyer la valeur 1.

Remarque : par souci de simplification, on supposera que la phrase ne contient pas de caractères accentués.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

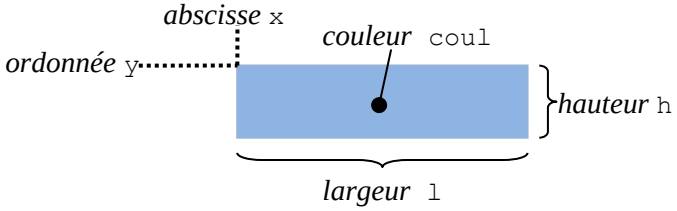
Damien Berthet & Vincent Labatut

TP 52 | tours de Hanoï

Présentation

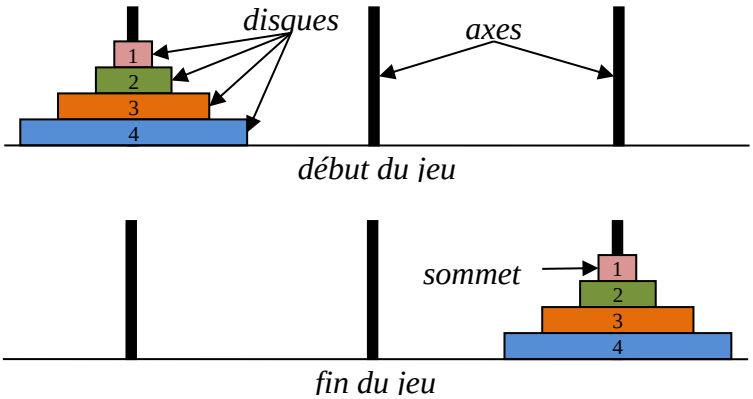
Ce TP repose sur la SDL, donc votre projet doit être configuré de manière à pouvoir l'utiliser. L'archive fournie avec ce sujet contient la bibliothèque `pile_tab` écrite en cours, qui implémente le type abstrait pile pour les entiers.

L'archive inclut également la bibliothèque `graphisme`, qui permet de dessiner grâce à la SDL. En plus des fonctions habituelles, vous y trouverez une fonction `remplis_rectangle(int x, int y, int l, int h, Uint32 coul)` qui permet de dessiner un rectangle plein :



1 Description

Le problème des tours de Hanoï est un jeu de réflexion publié par le mathématicien Édouard Lucas en 1883. L'univers du problème se compose de 3 axes verticaux et d'un ensemble de n disques de diamètres différents, et percés en leur centre. Au début du problème, tous les disques sont empilés sur l'axe de gauche. Le but du jeu est de déplacer la pile sur l'axe de droite en un minimum d'actions.



Le déplacement des disques est soumis aux règles suivantes :

- On ne peut déplacer qu'*un seul* disque à la fois.
- On ne peut prendre un disque sur une pile que s'il se trouve au *sommet* de cette pile.
- On ne peut poser un disque sur une pile que si le disque qui est au sommet de cette pile est d'un *diamètre supérieur*.

De plus, on fait l'hypothèse qu'au début du jeu, les disques sont disposés de manière à respecter ces règles.

2 Implémentation

Nous allons utiliser la structure de données des piles pour représenter les empilements de disques manipulés dans le problème des tours de Hanoï. Chaque disque sera représenté par un entier correspondant à son numéro. Un disque sera déplacé en effectuant des actions empiler/dépiler.

Différentes constantes sont définies dans le fichier `main.c` afin de faciliter la manipulation des disques :

- `N` : nombre de disques du problème.
- `AXE_LARGEUR`, `AXE_HAUTEUR` : largeur et hauteur des axes exprimées en pixels.
- `DISQUE_LARGEUR_MIN` : largeur du plus petit disque exprimée en pixels.
- `DISQUE_LARGEUR_COEF` : valeur utilisée pour calculer la largeur d'un disque.
- `DISQUE_HAUTEUR` : hauteur des disques exprimée en pixels.

Exercice 1

Dans le fichier `main.c`, créez un type énuméré `position` qui permettra de représenter la position d'un axe. Ce type contient 3 symboles : `gauche` < `centre` < `droite`.

Exercice 2

Créez un type structure `axe` qui nous servira à représenter un des trois axes du problème. Chaque axe est caractérisé par :

- Sa position ;
- Ses disques (représentés par une pile d'entiers).

Exercice 3

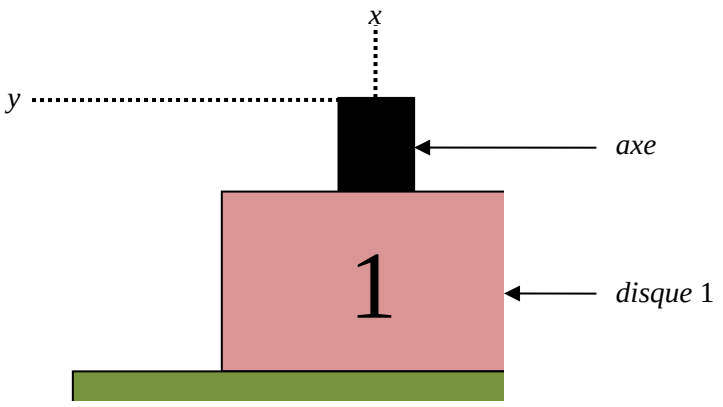
Écrivez une fonction `int initialise_probleme(axe *a_g, axe *a_c, axe *a_d)`, qui initialise les axes gauche `a_g`, central `a_c` et droit `a_d`. La fonction renvoie 0 en cas de succès ou -1 en cas d'erreur.

Exercice 4

Écrivez une fonction `Uint32 quelle_couleur(int disque)` qui prend en paramètre le numéro d'un disque `disque` (<`N`), et renvoie une constante correspondant à une couleur SDL. Chaque disque doit avoir une couleur différente.

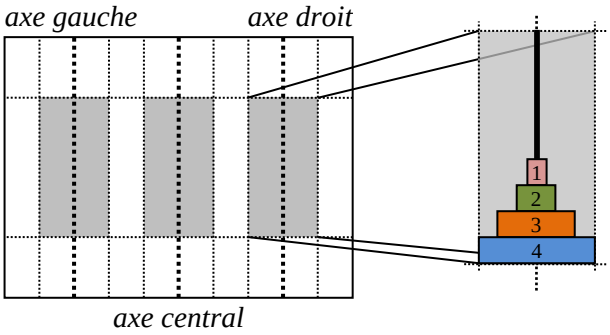
Exercice 5

En utilisant seulement les fonctions du type abstrait pour accéder à la pile, écrivez une fonction `trace_axe(axe a, int x, int y)`, qui dessine un axe et ses disques. La coordonnée `x` correspond au centre de l'axe, et `y` correspond au côté supérieur de l'axe. La largeur du disque numéro i est égale à `DISQUE_LARGEUR_MIN` $+i \times$ `DISQUE_LARGEUR_COEF`.



Exercice 6

Écrivez une fonction `trace_probleme(axe a, axe b, axe c)`, qui dessine les axes et les disques. Attention, `a` n'est pas forcément l'axe gauche, ni `b` et `c` les axes central et droit. Vous devez utiliser le champ indiquant la `position` de chaque axe, pour savoir où le placer.

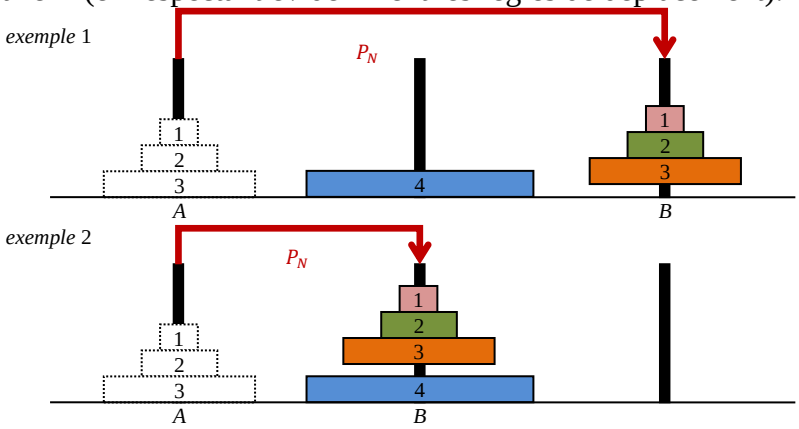


Remarque : les trois axes doivent être bien répartis à gauche, au milieu et à droite de l'écran.

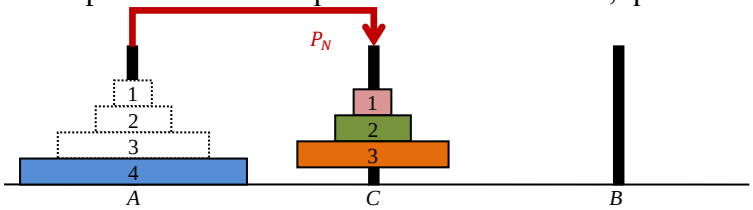
3 Résolution

Il est possible de montrer que le problème des tours de Hanoï admet une solution quel que soit le nombre N de disques utilisés.

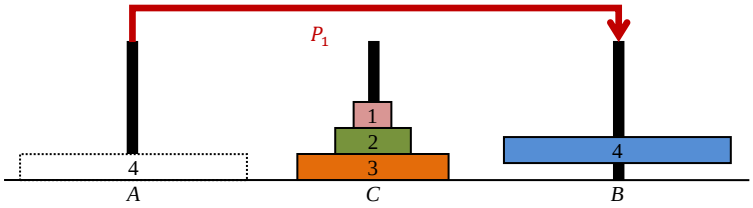
- **Propriété P_N :** il est possible de déplacer une pile de N disques d'un axe A vers un axe B (en respectant évidemment les règles de déplacement).



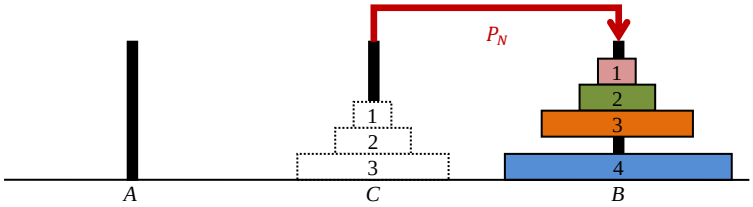
- **Cas de base** : $N = 1$: évident.
- **Cas de récurrence** :
  - On suppose que P_N est vraie.
  - On doit montrer P_{N+1}, i.e. montrer qu'il est possible de déplacer une pile de $N + 1$ disques d'un axe A vers un axe B.
  - Comme P_N est vraie, il est possible de déplacer les N disques supérieurs de notre pile de $N + 1$ disques vers l'axe inutilisé, que nous appellerons C.



  - Comme P_1 est vraie, il est possible de déplacer le disque de diamètre $N + 1$ vers l'axe B.



- Comme P_N est vraie, il est possible de déplacer les N disques de l'axe C vers l'axe B.



- On a montré P_{N+1}.

- **Conclusion** : P_N est vraie pour tout $N \geq 1$.

Exercice 7

Écrivez une fonction `int deplace_disque(axe *a, axe *b)` qui prend le disque situé au sommet de l'axe `a` et qui le place au sommet de l'axe `b`. Si ce déplacement est impossible (pile pleine, pile vide, diamètres des disques incompatibles…), la fonction doit renvoyer -1. Sinon, elle renvoie 0.

Remarque : `a` et `b` sont des axes quelconques.

Exercice 8

Écrivez une fonction `rafraichis_probleme(axe a, axe b, axe c)` qui utilise `trace_probleme` et les fonctions de la SDL pour redessiner le problème à l'écran. La fonction doit :

- Réinitialiser l'écran avec `efface_ecran` ;
- Dessiner l'état courant du problème avec `trace_probleme` ;
- Rafraichir l'écran avec `rafraichis_ecran` ;
- Attendre pendant quelques millisecondes avec `attend_delai`.

Exercice 9

Écrivez une fonction récursive `resous_hanoi(int n, axe *a, axe *b, axe *c)` qui utilise le principe récursif expliqué ci-dessus pour résoudre le problème des tours de Hanoï. On suppose que les axes passés en paramètres ont été préalablement initialisés. Le but est de déplacer les n disques supérieurs de l'axe `a` vers l'axe `b`. Donc, initialement on aura $n = N$.

Votre fonction devra afficher pas-à-pas le déroulement de la résolution, grâce à la fonction `rafraichis_probleme`.

Remarque : pour avoir un aperçu de ce qui vous est demandé, exécutez l'un des fichiers `exempleWin.exe` (pour Windows) ou `exempleLnx` (pour Linux) placés dans l'archive fournie avec ce sujet.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 53 | remplissage de formes

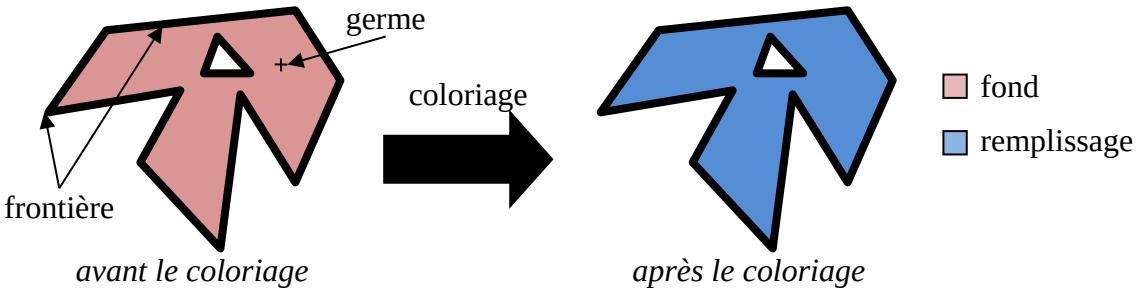
1 Présentation

Dans ce TP, nous nous intéresserons à des algorithmes qui permettent de remplir des surfaces fermées avec une couleur de fond. On utilisera les notions vues précédemment, en particulier les piles, la récursivité, et le calcul de complexité.

2 Préparation

On utilisera la terminologie suivante :

- *frontière* : ensemble des pixels délimitant la forme à colorier.
- *couleur de fond* : couleur initiale de la forme
- *couleur de remplissage* : couleur utilisée pour colorier la forme.
- *germe* : pixel de coordonnées (x_G, y_G) situé dans la forme (pas sur la frontière) et choisi au hasard.



Vous trouverez avec ce sujet une archive contenant la bibliothèque `graphiques`, dans laquelle vous trouverez les fonctions déjà utilisées dans les TP précédents, mais aussi la fonction `Uint32 couleurPixel(int x, int y)`, qui retourne le code couleur du pixel de coordonnées `(x ,y)`.

L'archive contient également une bibliothèque `figures`, qui vous permettra de tester votre implémentation des algorithmes de remplissage. Cette bibliothèque contient les fonctions suivantes :

- `void rectangle(int x, int y, int largeur, int hauteur, Uint32 coul)` : dessine un rectangle dont le coin supérieur gauche a pour coordonnées `(x ,y)` et dont les dimensions sont `largeur` et `hauteur`.
- `void flocon_von_koch(int n, Uint32 coul)` : dessine un flocon de von Koch (cf. le TP sur les *figures fractales*).
- `void trous(Uint32 coul)` : dessine une figure comportant des trous.

Remarque : le flocon et la figure à trous étant centrés sur l'écran, vous pouvez choisir comme germe le pixel central de coordonnées $(400,300)$.

Enfin, l'archive contient la bibliothèque `pile_liste` permettant de manipuler des piles d'entiers, qui sera nécessaire pour l'algorithme du balayage horizontal.

Le programme `main` contient une variable `COULEUR_FRONTIERE` représentant la couleur de la frontière, que vous devrez donc utiliser pour dessiner les figures à remplir.

3 Remplissage par frontière

À partir du germe, on applique l'algorithme récursif suivant :

- **Cas d'arrêt :** le pixel courant est sur la frontière, ou bien il est déjà de la couleur de remplissage.
- **Cas général :** le pixel n'est pas sur la frontière et n'est pas de la couleur de remplissage :
  - Il doit être colorié avec la couleur de remplissage.
  - Le même traitement est effectué sur les pixels situés en haut, en bas, à droite et à gauche du pixel courant.

Exercice 1

Écrivez une fonction récursive `void remplissage_frontiere(int xg, int yg, Uint32 coul)` qui implémente cet algorithme.

Exercice 2

On suppose que les complexités spatiales et temporelles de `couleur_pixel` et `affiche_pixel` sont toutes les deux en $O(1)$. Calculez les complexités spatiale et temporelle de votre fonction.

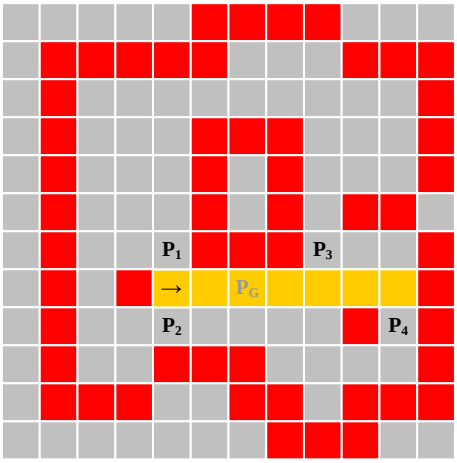
4 Balayage horizontal

Cet algorithme nécessite l'utilisation d'une pile de données dont chaque élément contient les deux coordonnées d'un pixel.

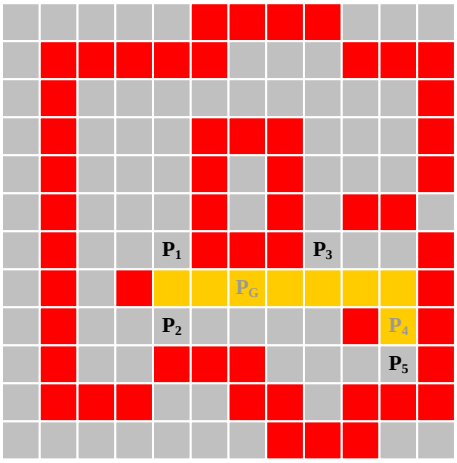
- **Initialisation :**
  - La pile est initialisée avec le germe.
- **Traitement :** tant que la pile n'est pas vide
  - Le sommet de la pile devient le pixel courant, la pile est dépilée.
  - Le segment horizontal contenant le pixel courant et délimité par les frontières de la forme est colorié avec la couleur de remplissage.
  - Un pixel de chacun des segments situés au-dessus ou au-dessous du segment courant sont empilés.

exemple :

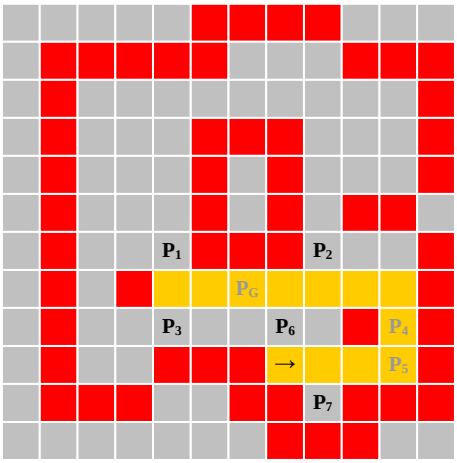
- Le pixel P_G est le germe, le segment contenant P_G est colorié.
- P_G est dépilé et les pixels P_1, P_2, P_3 et P_4 sont empilés.



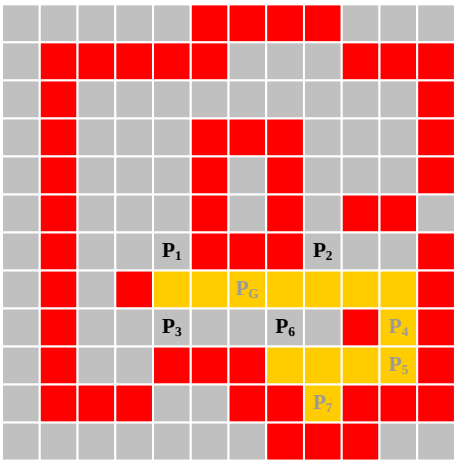
- P_4 est dépilé**,** le segment contenant P_4 est colorié.
- P_5 est empilé.



- P_5 est dépilé, le segment contenant P_5 est colorié.
- P_6 et P_7 sont empilés.



- P_7 est colorié puis dépilé.
- etc.
- L'algorithme se termine lorsque la pile est vide
- On remarque que le segment contenant P_3 sera colorié lors du dépilement de P_6.



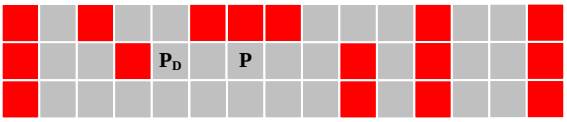
Exercice 3

Renommez les bibliothèques `pile_liste` et `liste_s` en `pile_liste_coord` et `liste_s_coord`. Modifiez-les ensuite de manière à pouvoir utiliser des piles dont chaque élément contient deux entiers `x` et `y` (et non plus un seul entier `valeur`).

Exercice 4

Écrivez une fonction `int debut_segment(int x, int y)` qui reçoit les coordonnées `(x,y)` d'un point P, et qui détermine (et renvoie par valeur) l'abscisse du point P_D situé le plus à gauche du même segment horizontal.

exemple :



Exercice 5

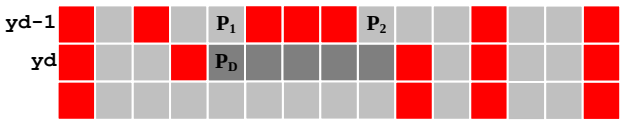
Écrivez une fonction `void colorie_segment(int xd, int yd, Uint32 coul)` qui effectue la deuxième étape du traitement : le coloriage d'un segment horizontal délimité à gauche par le point de coordonnées `(xd,yd)` et à droite par la frontière.

Exercice 6

Écrivez une fonction `void empile_suivants(pile *p, int xd, int yd, int position, Uint32 coul)` qui effectue le troisième point du traitement : la recherche des segments suivants. En partant de la colonne `xd`, et en allant jusqu'à la frontière, la fonction doit empiler dans `p` les pixels représentant les segments rencontrés sur la ligne `(yd+position)`. De cette manière, la même fonction peut être utilisée pour déterminer les segments suivants situés au-dessus du segment courant, et ceux situés au-dessous.

exemples :

- `position = -1`



- `position = +1`

Exercice 7

Écrivez la fonction `void balayage_horizontal(int xg, int yg, Uint32 coul)` qui implémente l'algorithme de remplissage par balayage horizontal décrit plus haut en utilisant les fonctions précédentes.

Exercice 8

On suppose que les complexités spatiales et temporelles des opérations sur les piles (`empile`, `depile`, `sommet`, `cree_pile`) sont toutes en $O(1)$. Calculez les complexités spatiale et temporelle de votre fonction, puis comparez-les à celles obtenues pour le remplissage par frontière.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 54 | parcours d'un labyrinthe

Présentation

Dans ce TP, nous allons résoudre le problème consistant à trouver un chemin dans un labyrinthe. Ceci nous permettra d'illustrer la différence entre l'utilisation de piles et de files de données.

1 Introduction

Le Labyrinthe éponyme est un bâtiment construit par l'architecte Dédale pour enfermer le Minotaure. Nous nous contenterons d'un labyrinthe plus modeste, prenant la forme d'une figure géométrique complexe, construite de manière à ce qu'il soit difficile de la traverser. Le but de ce TP est d'implémenter des algorithmes permettant de trouver un chemin allant de l'entrée à la sortie d'un labyrinthe.

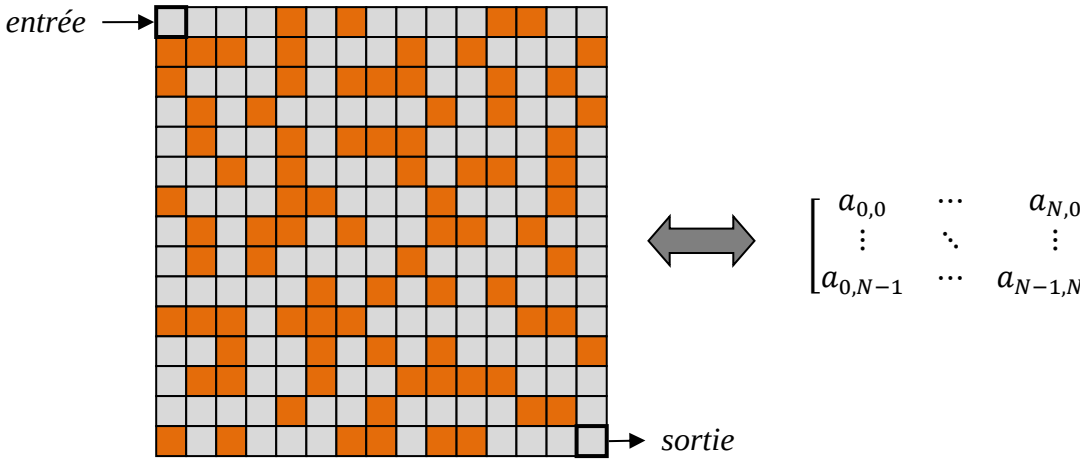
L'archive fournie avec le sujet contient notamment la bibliothèque `labyrinthe`, qui définit la dimension du labyrinthe grâce à la constante `DIM_LABY`, et les fonctions suivantes :

- `void initialise_labyrinthe_1(valeur_case laby[DIM_LABY][DIM_LABY])` : initialise le tableau spécifié avec un labyrinthe.
- `void initialise_labyrinthe_2(valeur_case laby[DIM_LABY][DIM_LABY])` : initialise le tableau avec un autre labyrinthe.
- `void dessine_labyrinthe(valeur_case laby[DIM_LABY][DIM_LABY])` : dessine le labyrinthe correspondant au tableau spécifié.

Remarque : il est normal que des erreurs apparaissent à la première compilation, car l'un des types requis est manquant : il fait l'objet du premier exercice.

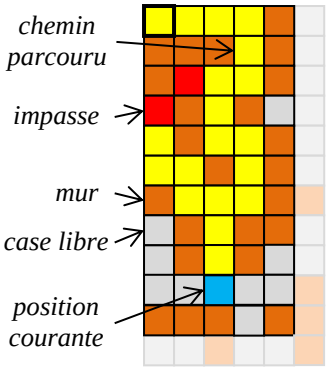
2 Représentation

Le labyrinthe sera représenté en mémoire par une matrice carrée (i.e. : un tableau à deux dimensions), chacun des éléments du tableau correspondant à une case du labyrinthe. La case située en haut à gauche correspond à l'*entrée* du labyrinthe, et celle située en bas à droite est la sortie, comme représenté dans le schéma.



La valeur contenue dans une case du tableau dépend de la nature de la case correspondante dans le labyrinthe. On a les possibilités suivantes :

- *Case libre* : case que l'on peut parcourir et qui n'a pas encore été testée (représentée en gris sur le schéma suivant).
- *Case courante* : position courante (représentée en bleu).
- *Case contenant un mur* : case que l'on ne peut pas parcourir (représentée en marron).
- *Case appartenant à un chemin* : case qui a déjà été parcourue (représentée en jaune).
- *Case échec* : case qui a déjà été parcourue et qui constitue une impasse (représentée en rouge).



Exercice 1

Dans la librairie `labyrinthe`, définissez un type énuméré `valeur_case` permettant de représenter les 5 valeurs possibles d'une case du labyrinthe : `libre`, `courante`, `mur`, `chemin` et `echec`.

3 Résolution du problème

La méthode la plus simple pour rechercher le chemin est d'utiliser une file ou une pile pour tester tous les chemins possibles jusqu'à tomber sur une solution.

Algorithme :

- Initialement, la file (resp. pile) est vide.
- On initialise la position courante avec l'entrée du labyrinthe.
- On répète le traitement suivant :
  - On enfile (resp. empile) toutes les cases voisines de la position courante.
  - On détermine la nouvelle position courante :
    - On récupère la case qui est en tête (resp. au sommet) de la file (resp. pile).
    - On défile (resp. dépile) la file (resp. pile).
    - Si la case est accessible alors elle devient la nouvelle position courante.
    - Sinon on recommence avec la case suivante.
    - Si la file (resp. pile) est vide, alors la recherche est terminée : il n'y a pas de solution.
- Jusqu'à ce que la position courante soit la sortie du labyrinthe (succès) ou que la file (resp. file) soit vide (échec).

Remarques :

- Les cases voisines d'une case donnée sont les quatre cases situées au-dessus, au-dessous, à gauche et à droite.
- On dit qu'une case est *accessible* si elle se situe à l'intérieur du labyrinthe et si sa valeur dans le tableau est `libre`.
- On dit qu'une case est une *impasse* si elle est dans le labyrinthe et si toutes ses voisines sont inaccessibles.

Exercice 2

Écrivez une fonction `int est_dans_labyrinthe(int x, int y)` qui renvoie 1 si la case de coordonnées (x, y) est dans le labyrinthe, et 0 sinon.

Exercice 3

Écrivez une fonction `int est_accessible(valeur_case laby[DIM_LABY][DIM_LABY], int x, int y)` qui renvoie 1 si la case de coordonnées (x, y) est accessible, et 0 sinon.

Exercice 4

Écrivez une fonction `int est_impasse(valeur_case laby[DIM_LABY][DIM_LABY], int x, int y)` qui renvoie 1 si la case de coordonnées (x, y) est une impasse, et 0 sinon.

Exercice 5

La recherche de solution utilisant la file est appelée *recherche en largeur*. Écrivez une fonction `void recherche_largeur(valeur_case laby[DIM_LABY][DIM_LABY])` qui implémente l'algorithme précédent en utilisant une file. Vous utiliserez pour cela la bibliothèque `file_liste` incluse dans le projet. Elle a été modifiée de manière à pouvoir manipuler des éléments contenant deux valeurs entières `x` et `y`.

Vous devez modifier les valeurs du tableau représentant le labyrinthe, de manière à indiquer les cases parcourues, les impasses et la position courante. Pour visualiser le déroulement de l'algorithme, vous utiliserez la fonction `dessine_labyrinthe` à chaque itération.

Exercice 6

La recherche de solution utilisant la pile est appelée *recherche en profondeur*. En utilisant la bibliothèque `pile_liste` et en respectant les mêmes consignes que pour `recherche_largeur`, écrivez une fonction `void recherche_profondeur(valeur_case laby[DIM_LABY][DIM_LABY])` qui implémente l'algorithme de recherche en profondeur.

Exercice 7

Testez les deux types de recherche sur le premier labyrinthe. Quelles différences remarque-t-on entre la recherche en profondeur et la recherche en largeur ? Testez-les sur le second labyrinthe. Qu'en déduisez-vous ?

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 55 | génération d'un labyrinthe

Présentation

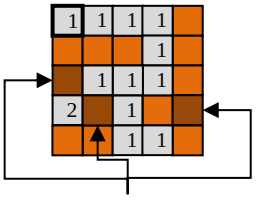
Ce TP est complémentaire de celui sur la recherche de chemin dans un labyrinthe. Il s'agit ici de générer des labyrinthes, de façon partiellement aléatoire. L'archive fournie avec le sujet contient certaines fonctions écrites pour le TP précédent, en particulier `dessine_labyrinthe` et `recherche_profondeur`. De plus, le type énuméré `valeur_case`, utilisé pour représenter la nature des cases du labyrinthe, est déjà créé.

1 Méthode par fusion

Il existe différentes méthodes pour générer aléatoirement un labyrinthe. Nous allons utiliser la méthode par fusion, qui fonctionne en manipulant des *zones* :

- Une zone est un ensemble de cases accessibles.
- Une zone est caractérisée par un numéro unique dans le labyrinthe.
- Une zone est connexe (il est possible d'atteindre n'importe quelle case de la zone à partir de n'importe quelle autre case de la zone).
- Une zone est close (elle contient toutes les cases qu'on peut atteindre à partir de chaque case de la zone).

labyrinthe contenant deux zones



murs utiles

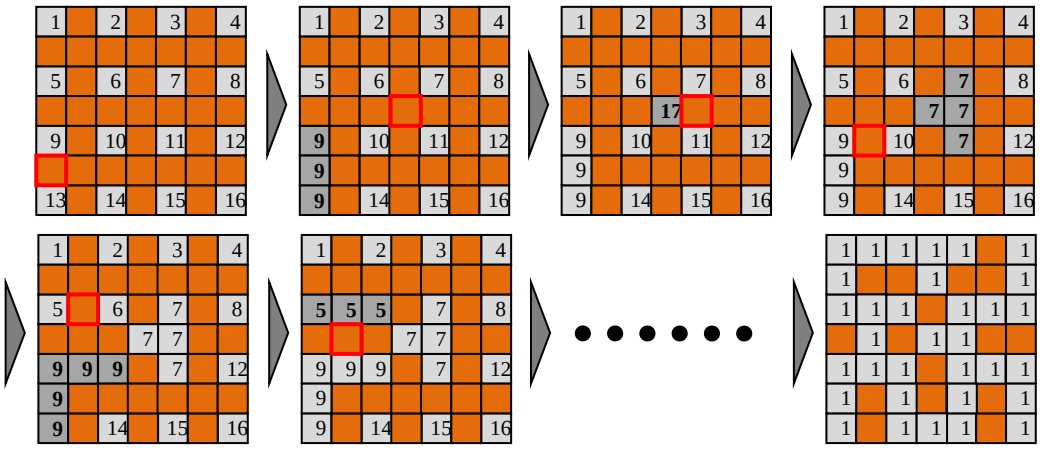
De plus, on dira qu'une case contenant un mur est *utile* si :

- Soit toutes les cases voisines sont aussi des murs.
- Soit la suppression du mur permet de fusionner au moins deux zones différentes.

Algorithme :

- Initialisation :
  - On part d'un labyrinthe dont les murs sont disposés de manière à former une grille.
  - Chaque case libre constitue une zone. Les zones sont numérotées en partant de 1 pour la zone correspondant à la case d'entrée.
- Fusion :
  - On sélectionne aléatoirement une case contenant un mur utile.
  - Le mur est supprimé, et les zones qu'il séparait sont fusionnées.
  - La zone obtenue porte le plus petit numéro des zones fusionnées.
  - Si le mur utile était entouré de murs, une nouvelle zone est créée (avec un nouveau numéro).
- On répète la fusion jusqu'à ce qu'il n'y ait plus qu'une seule zone.

exemple :



Contrairement au TP précédent, nous allons dans un premier temps représenter le labyrinthe sous la forme d'une matrice d'entiers, et non pas de valeurs de type `valeur_case`. On appellera cette structure un pseudo-labyrinthe. On utilisera la valeur 0 pour représenter les murs, et toute autre valeur positive correspondra au numéro d'une zone (donc à une case libre). Cela signifie donc que les zones sont numérotées à partir de 1.

La fonction `dessine_pseudo_labyrinthe` contenue dans la bibliothèque `labyrinthe` est fournie spécifiquement pour afficher ce type de matrice entière : chaque zone est représentée d'une couleur différente.

Exercice 1

Écrivez une fonction `int initialise_grille(int pseudo[DIM_LABY][DIM_LABY])` qui dispose les murs dans le pseudo-labyrinthe de manière à former une grille, et qui numérote les cases libres. La fonction doit renvoyer le nombre de zones créées.

Exercice 2

Écrivez une fonction `int est_utile(int pseudo[DIM_LABY][DIM_LABY], int x, int y)` qui renvoie 1 si le mur de coordonnées (x, y) est utile, et 0 sinon.

Exercice 3

Écrivez une fonction `void remplace_valeur(int pseudo[DIM_LABY][DIM_LABY], int ancienne, int nouvelle)` qui remplace dans le tableau `pseudo` toute occurrence de la valeur `ancienne` par une occurrence de la valeur `nouvelle`.

Exercice 4

Écrivez une fonction `void supprime_mur(int pseudo[DIM_LABY][DIM_LABY], int x, int y, int *n)` qui supprime le mur situé aux coordonnées (x, y). La fonction doit également déterminer les zones à fusionner puis effectuer la fusion (grâce à la fonction `remplace_valeur`), en n'oubliant pas de traiter la case de coordonnées (x, y). Le paramètre `n` sera éventuellement utilisé pour numéroter une nouvelle zone.

Exercice 5

Écrivez une fonction `int teste_zone_unique(int pseudo[DIM_LABY][DIM_LABY])` qui renvoie 1 s'il n'y a qu'une seule zone dans le pseudo-labyrinthe, et 0 sinon.

Exercice 6

La bibliothèque `labyrinthe` contient une fonction `void tire_mur(int pseudo[DIM_LABY][DIM_LABY], int *x, int *y)` qui tire au sort un des murs du pseudo-labyrinthe et qui renvoie ses coordonnées par les paramètres `x` et `y`.

Utilisez cette fonction ainsi que les fonctions précédentes pour écrire une fonction `void genere_pseudo_labyrinthe(int pseudo[DIM_LABY][DIM_LABY])` qui génère aléatoirement un pseudo-labyrinthe en utilisant l'algorithme décrit plus haut. Vous devez utiliser `dessine_pseudo_labyrinthe` pour donner le détail pas-à-pas de cette construction.

Exercice 7

On veut appliquer à notre labyrinthe un algorithme de recherche de chemin écrit lors du TP précédent. Mais pour cela, on a besoin d'une matrice contenant des valeurs de type `valeur_case` et non pas des entiers.

Écrivez une fonction `convertis_labyrinthe(int pseudo[DIM_LABY][DIM_LABY], valeur_case laby[DIM_LABY][DIM_LABY])` qui reçoit un pseudo-labyrinthe `pseudo` déjà créé avec `genere_pseudo_labyrinthe`, ainsi qu'un labyrinthe vide `laby`, et qui initialise `laby` grâce aux valeurs contenues dans `pseudo`.

Exercice 8

Dans la fonction `main`, créez un pseudo-labyrinthe avec `genere_pseudo_labyrinthe`, puis convertissez-le grâce à `convertis_labyrinthe` pour obtenir un labyrinthe, et enfin appliquez `recherche_profondeur` pour en trouver la solution.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 56 | tri par dénombrement

Présentation

Dans ce TP, on veut implémenter le tri par dénombrement, qui n'a pas été étudié en classe. Nous allons pour cela développer une bibliothèque `tri_tab`, qui sera dédiée au tri de tableaux d'entiers.

1 Initialisation

Exercice 1

Créez les fichiers d'une nouvelle bibliothèque `tri_tab`. Définissez-y une constante `N` correspondant à la taille des tableaux que l'on voudra trier.

Exercice 2

On veut pouvoir initialiser des tableaux non-triés, de façon aléatoire. Pour cela, nous allons nous servir des fonctions `srand` et `rand`, qui ont déjà été utilisées plusieurs fois dans des TP précédents. Pour rappel :

- `void srand(unsigned int seed)` : initialise le générateur pseudo-aléatoire en prenant le paramètre `seed` comme graine. Il ne faut l'utiliser qu'une seule fois, elle est nécessaire à la création de la série de nombres.
- `int rand(void)` : renvoie l'entier suivant dans la série pseudo-aléatoire. La série doit d'abord avoir été initialisée avec `srand`. L'entier obtenu est compris entre 0 et la constante `RAND_MAX` (inclus).

Ce générateur pseudo-aléatoire approxime une distribution uniforme, c'est-à-dire que la probabilité qu'un entier $x \in [0, RAND\_MAX]$ apparaisse quand on utilise `rand` est de $1/(RAND\_MAX + 1)$.

Il faut remarquer que toute la série de nombres dépend complètement et uniquement de la graine utilisée pour initialiser le générateur. Par conséquent, si on utilise plusieurs fois la même graine, on obtiendra exactement les mêmes valeurs, et dans le même ordre. Pour éviter cette répétition, il est nécessaire d'utiliser une graine différente à chaque initialisation du générateur.

La fonction `time(NULL)` renvoie une valeur de type `long` correspondant au nombre de secondes écoulées depuis le *1er janvier 1970 00:00:00*. Ce nombre varie en permanence, et il constitue donc une bonne graine.

Dans `tri_tab`, écrivez une fonction `int genere_entier(int inf, int sup)` qui renvoie un entier généré pseudo-aléatoirement et appartenant à $[inf, sup]$. La première fois qu'elle est appelée (et uniquement cette fois-là) votre fonction doit initialiser le générateur grâce à `srand`.

Remarque : pensez aux différentes classes de mémorisation d'une variable.

Exercice 3

Dans `tri_tab`, écrivez une fonction `void affiche tableau(int tab[], int taille)` permettant d'afficher un tableau contenant `taille` éléments.

exemple : pour `tab={1,5,6,9,77,1,5,6}`, on doit obtenir :

```
{1 5 6 9 77 1 5 6}
```

Exercice 4

Dans `tri_tab`, écrivez une fonction `void init_tableau(int tab[N], int n)` qui initialise pseudo-aléatoirement un tableau de taille N avec des entiers appartenant à $[0, n]$.

exemple : un appel avec `n=10` pourrait générer quelque chose comme `{1,4,0,5,7,3,1,6}`.

Exercice 5

Pour tester certains tris, il est préférable que chaque valeur dans le tableau soit unique. Dans `tri_tab`, écrivez une fonction `void init_tableau_unique(int tab[N], int n)` qui initialise pseudo-aléatoirement un tableau de taille N avec des entiers appartenant à $[0, n]$, de manière à a ce qu'aucun *doublon* n'apparaisse (i.e. la même valeur ne doit pas apparaître plusieurs fois dans le tableau.

Remarque : la fonction ne peut marcher correctement que si on a $n \geq N - 1$.

exemples :

- Le tableau de l'exemple précédent ne peut pas être généré par cette fonction, car il contient deux fois la valeur 1.
- Par contre, le tableau `{1,4,0,5,7,3,2,6}` pourrait être généré par cette fonction.

2 Algorithme de tri

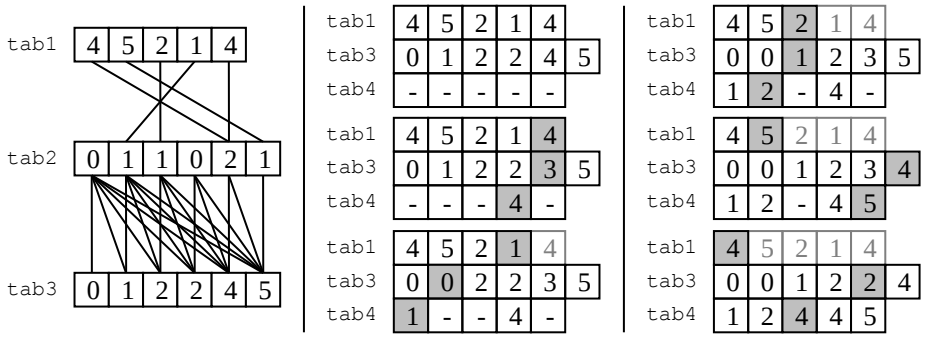
On veut trier un tableau `tab1` de N entiers appartenant à $[0, M[$, une même valeur pouvant apparaître plusieurs fois dans le même tableau. Nous allons pour cela utiliser le tri par dénombrement (également appelé *tri comptage* ou *tri casier*). Il s'agit d'une méthode de tri différente de celles vues en cours, car ici on ne compare pas directement les éléments de la séquence entre eux.

L'algorithme comprend 3 étapes :

- **1ère étape :** dénombrement
  - On dénombre (i.e. on compte) combien de fois chaque valeur v de `tab1` apparaît dans `tab1`.
  - Ces dénombrements sont stockés dans un tableau `tab2` de taille M.
  - Pour tout $v \in [0, M[$, la valeur contenue dans `tab2[v]` représente le nombre d'occurrences de v dans `tab1`.
- **2ème étape :** distribution
  - On calcule la distribution cumulative de `tab1` grâce à `tab2`.
  - On stocke le résultat dans un tableau `tab3` de taille M.
  - Pour tout $v \in [0, M[$, la valeur contenue dans `tab3[v]` représente le nombre de valeurs dans `tab1` qui sont inférieures ou égales à v.
- **3ème étape :** construction
  - On construit le tableau trié `tab4` à partir du tableau initial `tab1` et de sa distribution `tab3`.
  - En partant de la fin de `tab1`, on considère chaque valeur v.
  - La valeur de distribution `tab3[v]` correspond au nombre de valeurs inférieures ou égales à v. Donc cette valeur de distribution peut être utilisée pour déterminer la position de v dans le tableau trié `tab4`.

- On décrémente la valeur `tab3[v]`, puis on insère x dans `tab4[tab3[v]]`.

exemple : $N = 5$ et $M = 6$



Exercice 6

Dans `tri_tab`, définissez une constante `M` correspondant à la borne supérieure des valeurs placées dans les tableaux à trier. Puis, toujours dans `tri_tab`, écrivez une fonction `void calcule_denombrement(int tab1[N], int tab2[M])` qui dénombre les valeurs du tableau `tab1` et place le résultat de ce dénombrement dans le tableau `tab2`.

exemple : pour `N=10`, `M=5`, `tab1={0,1,3,1,0,1,1,4,0,0}`, on obtient `tab2={4,4,0,1,1}`.

Remarque : attention lorsque vous utilisez init_tableau lors du test de votre fonction, car celle-ci génère des entiers dans $[inf, sup]$, et non pas dans $[inf, sup[$.

Exercice 7

Dans `tri_tab`, écrivez une fonction `void calcule_distribution(int tab2[M], int tab3[M])` qui calcule le tableau de distribution `tab3` grâce au tableau de dénombrement `tab2`.

exemple : pour `N=10`, `M=5`, `tab2={4,4,0,1,1}` on obtient `tab3={4,8,8,9,10}`.

Exercice 8

Dans `tri_tab`, écrivez une fonction `void calcule_construction(int tab1[N], int tab3[M], int tab4[N])` qui remplit le tableau trié `tab4` grâce au tableau initial `tab1` et au tableau de distribution `tab3`.

exemple : pour `N=10`, `M=5`, `tab1={0,1,3,1,0,1,1,4,0,0}`, `tab3={4,8,8,9,10}`, on obtient `tab4={0,0,0,0,1,1,1,1,3,4}`.

Exercice 9

En utilisant les fonctions précédentes, écrivez dans `tri_tab` une fonction `void tri_denombrement(int tab1[N], int tab4[N])` qui trie le tableau `tab1` et stocke le résultat de ce tri dans le tableau `tab4`.

exemple : pour `N=10`, `M=5`, `tab1={0,1,3,1,0,1,1,4,0,0}`, on obtient `tab4={0,0,0,0,1,1,1,1,3,4}`.

Exercice 10

Calculez la complexité spatiale de `tri_denombrement`. Ce tri est-il en place ? Est-il stable ?

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 57 | tri Cocktail

Présentation

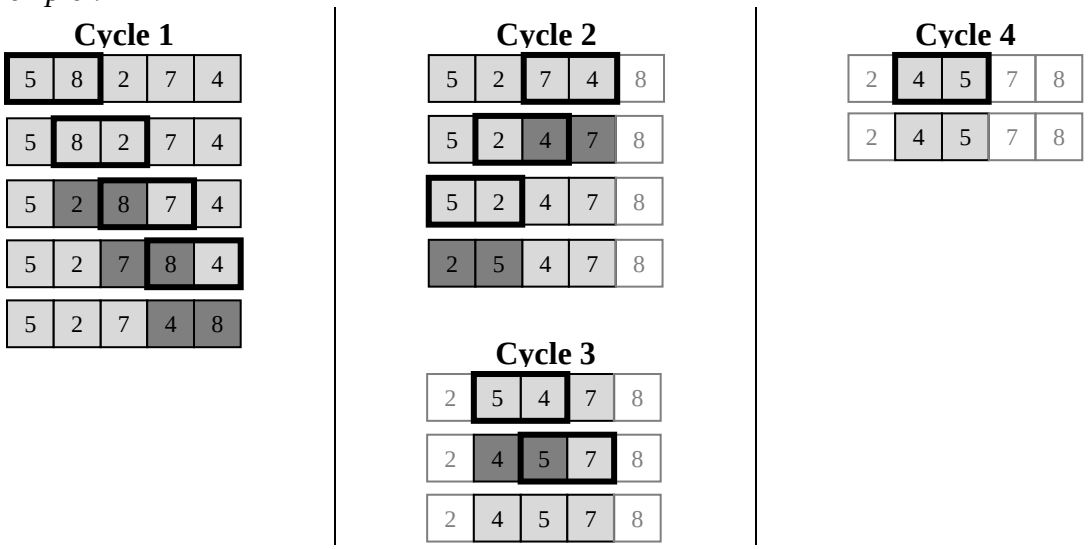
Le but de ce TP est d'approfondir la notion de complexité algorithmique via un tri qui n'a pas été vu en cours : le tri Cocktail. Nous l'appliquerons à des tableaux d'entiers, comme en cours et dans les TP précédents. Pour vous aider, la bibliothèque `tri_tab` est fournie avec ce sujet.

1 Version itérative

Le tri cocktail est une variante du tri à bulle, utilisant une bulle bidirectionnelle. Dans le tri à bulle, à chaque cycle, la bulle traverse le tableau toujours dans le même sens, en général du début vers la fin.

Dans le tri cocktail, la bulle effectue des allez-retours entre le début et la fin du tableau. De plus, on ne considère pas le tableau entier, mais seulement un sous-tableau, qui est réduit d'un élément à chaque cycle. En effet, on considère qu'à la fin d'un cycle, le dernier élément traité a atteint sa place définitive.

exemple :



Légende :

- les éléments encadrés sont les éléments comparés.
- les éléments foncés sont les éléments qui ont été échangés.
- les éléments blancs sont les éléments qui ne font pas partie du sous-tableau courant.

Exercice 1

Écrivez une fonction `void tri_cocktail_it(int tab[N])` chargée d'appliquer le tri cocktail décrit ci-dessus à un tableau d'entiers `tab` de taille `N`.

Exercice 2

Quelles sont les propriétés de ce tri (caractère en place et stabilité) ?

Exercice 3

Calculez les complexités asymptotiques spatiale et temporelle de votre fonction, dans le pire des cas, en détaillant vos calculs comme en cours.

2 Comparaison avec le tri à bulle

Exercice 4

Comparez les complexités temporelles du tri à bulle et du tri cocktail : que pouvez-vous en déduire ?

Exercice 5

On veut maintenant comparer les deux algorithmes, mais pas de façon asymptotique. Pour cela, on va s'intéresser aux nombres de comparaisons et d'échanges (entre deux entiers du tableau) effectués par chaque algorithme. Ainsi, dans l'exemple précédent, le tri cocktail effectue 10 comparaisons et 6 échanges.

Soit le tableau `tab={7,4,2,8,3,1}`. En utilisant la même présentation que dans l'exemple du sujet, appliquez à ce tableau les deux algorithmes de tri. Déterminez le nombre de comparaisons et d'échanges effectués par chaque algorithme.

Remarque : vous utiliserez la version du tri à bulle présentée en cours.

Lequel des deux algorithmes est le plus efficace pour trier `tab` ? Cela est-il cohérent avec la conclusion que vous aviez tirée de la comparaison de leurs complexités temporelles asymptotiques ?

3 Version récursive

Exercice 6

Écrivez une fonction récursive `void tri_cocktail_rec(int tab[N], int deb, int fin, int direction)` implémentant le tri cocktail. Chaque appel de cette fonction doit correspondre à un cycle du tri.

Le paramètre `tab` correspond au tableau à trier, `deb` et `fin` sont respectivement les index des éléments marquant le début et la fin du sous-tableau courant, et `direction` indique le sens de parcours du tableau : 0 (du début vers la fin) ou 1 (de la fin vers le début).

Exercice 7

Calculez la complexité temporelle asymptotique dans le pire des cas de la fonction `tri_cocktail_rec`.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 58 | représentation des tris

Présentation

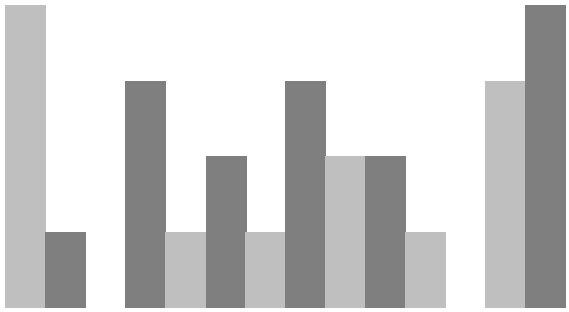
Dans ce TP, l'objectif est d'implémenter une représentation graphique de certains tris vus en cours. Pour cette raison, ce sujet est fourni avec deux bibliothèques : `graphisme`, qui permet d'utiliser la SDL et a déjà été utilisée de nombreuses fois, et `tri_tab` qui contient des algorithmes de tri implémentés pour traiter des tableaux d'entiers. Les fonctions à écrire pour ce TP seront toutes à placer dans la bibliothèque `tri_tab` (à l'exception de la fonction principale `main`, bien entendu).

1 Représentation des données

On s'inspire des vidéos de ce type, dans lesquelles chaque entier du tableau à trier est représenté par un rectangle dont :

- La hauteur correspond à la de l'entier ;
- La position horizontale (abscisse) correspond à l'index de l'entier dans le tableau.

exemple : pour un tableau `{4,1,0,3,1,2,1,3,2,2,1,0,3,4}`, on veut obtenir quelque chose du type :



Pour rendre la représentation plus lisible, on alternera une teinte foncée et une teinte claire de la même couleur, comme ci-dessus avec le gris (le 1er rectangle est gris clair, le 2ème gris foncé, etc.). De plus, on utilisera trois couleurs différentes :

- Gris pour représenter les éléments qui n'ont pas encore été traités ;
- Rouge pour représenter les éléments qui sont en train d'être déplacés ou échangés ;
- Bleu pour représenter les éléments qui ont atteint leur position finale (i.e. ils sont au bon endroit dans le tableau).

À cela, on peut rajouter la couleur noir, qui permettra d'effacer un rectangle (en dessinant un rectangle noir par-dessus).

Exercice 1

Dans `tri_tab`, écrivez une fonction `void dessine_valeur(int index, int valeur, int mode)` qui dessine le rectangle correspondant à la valeur spécifiée, sachant qu'elle se trouve à la poisition index. Le paramètre index est une valeur numérique indiquant la couleur à utiliser :

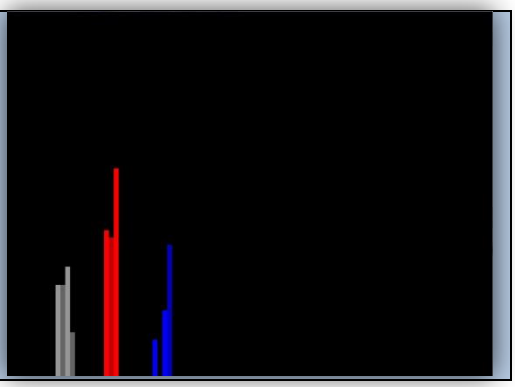
- 0 : noir ;
- 1 : gris (valeur pas encore traitée) ;

- 2 : rouge (valeur en cours de traitement) ;
- 3 : bleu (valeur occupant sa position finale).

Vous devez utiliser les constantes de `tri_tab` : `N` (nombre d'éléments le tableau) pour déterminer la position horizontale du rectangle et `M` (valeur maximale d'un élément) pour déterminer sa hauteur, de manière à occuper au maximum l'espace disponible dans la fenêtre. Pensez à utiliser les constantes `FENETRE_LARGEUR` et `FENETRE_HAUTEUR` de `graphisme`.

exemple : pour les instructions suivantes, avec `N=100` et `M=100`, on obtient *exactement* l'affichage ci-dessous :

```
dessine_valeur(10,25,1);
dessine_valeur(11,25,1);
dessine_valeur(12,30,1);
dessine_valeur(13,12,1);
dessine_valeur(20,40,2);
dessine_valeur(21,38,2);
dessine_valeur(22,57,2);
dessine_valeur(30,10,3);
dessine_valeur(31,0,3);
dessine_valeur(32,18,3);
dessine_valeur(33,36,3);
raffraichis_fenetre();
```



Remarque : pour des raisons de rapidité, la fonction `raffraichis_fenetre` ne doit *pas* être appelée dans votre fonction `dessine_valeur`.

Exercice 2

Selon le même principe, écrivez dans `tri_tab` une fonction `void dessine_tableau(int tab[N], int mode)` qui effectue l'affichage d'un tableau entier.

Remarque : cette fois, la fonction `raffraichis_fenetre` doit être appelée directement depuis votre fonction `dessine_tableau`.

Exercice 3

Toujours d'après le même principe, écrivez dans `tri_tab` une fonction `void dessine_sous_tableau(int tab[N], int debut, int fin, int mode)` qui effectue l'affichage de la partie du tableau comprise entre les positions `debut` et `fin` (incluses).

Remarque : votre fonction `dessine_sous_tableau` doit aussi appeler `raffraichis_fenetre`.

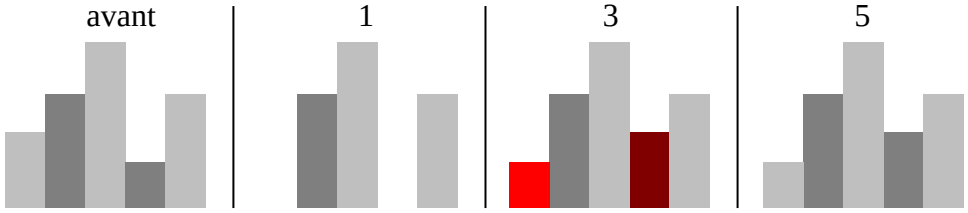
Exercice 4

L'opération d'interversion est commune à plusieurs algorithmes de tri. Elle consiste à échanger les positions de deux valeurs du tableau. Dans `tri_tab`, écrivez une fonction `void echange_valeurs(int tab[N], int i, int j)` qui réalise cet échange de valeurs, ainsi que l'affichage graphique approprié.

L'algorithme est le suivant :

1. Effacer les rectangles représentant les deux valeurs actuelles, en dessinant des rectangles noirs par-dessus ;
2. Mettre à jour le tableau (i.e. échanger les valeurs numériques) ;
3. Dessiner les nouveaux rectangles en rouge ;
4. Rafraîchir et attendre un certain délai (qui peut être fixé par une constante `DELAI`) ;
5. Dessiner les mêmes rectangles en gris, *sans rafraichir* cette fois.

Certaines de ces étapes sont illustrées par le schéma suivant, dans lequel on veut intervertir les entiers situés aux positions 0 et 3 :



Exercice 5

Une opération plus simple qui apparait dans certains algorithmes (généralement ceux qui ne sont pas des tris *sur place*) consiste simplement à écraser une valeur existante. Dans `tri_tab`, écrivez une fonction `void ecrase_valeur(int tab[N], int i, int valeur)` qui remplace la valeur de `tab[i]` par `valeur`, et qui modifie l'affichage de façon appropriée :

1. Effacer le rectangle représentant `tab[i]` ;
2. Mettre à jour le tableau `tab` ;
3. Dessiner le nouveau rectangle en rouge ;
4. Rafraîchir et attendre un certain délai ;
5. Dessiner le même rectangle en gris, *sans rafraichir*.

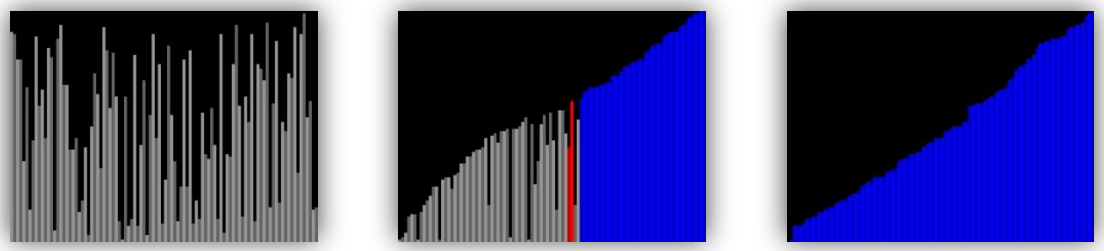
2 Algorithmes de tri

Exercice 6

Dans `tri_tab`, modifiez la fonction du tri à bulles, `tri_bulles`, de manière à :

- Afficher le tableau entier en gris avant de traitement ;
- Réaliser le tri en utilisant `echange_valeurs` ;
- Ajouter un appel à `dessine_valeur` dans la boucle principale, pour afficher en bleu les valeurs qui ont atteint leur position finale ;
- Afficher le tableau entier en bleu à la fin du traitement.

Les captures d'écran ci-dessous représentent l'affichage obtenu au début, au milieu et à la fin du traitement :



Remarque : pour faciliter votre travail, il est recommandé de :

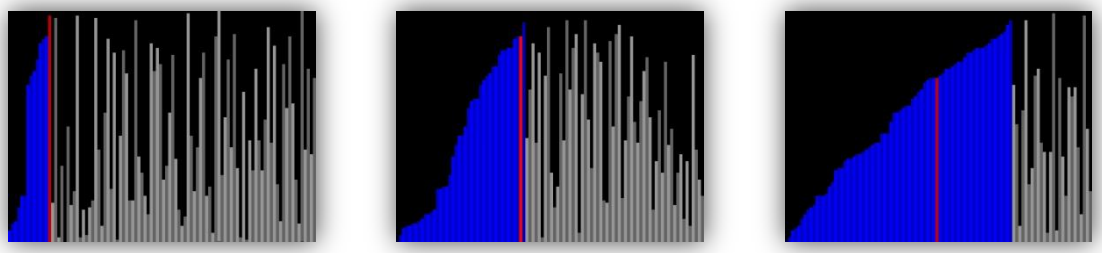
- Commencer par afficher l'évolution du traitement sans se soucier des couleurs ;
- Puis introduire la couleur rouge ;
- Et enfin la couleur bleue.

Exercice 7

Même chose pour le tri par sélection.

Exercice 8

Pour le tri par insertion, il n'y a pas d'interversion, donc vous ne devez pas utiliser `echange_valeurs`, mais plutôt `ecrase_valeur`. On représentera en bleu la partie du tableau qui est déjà triée.

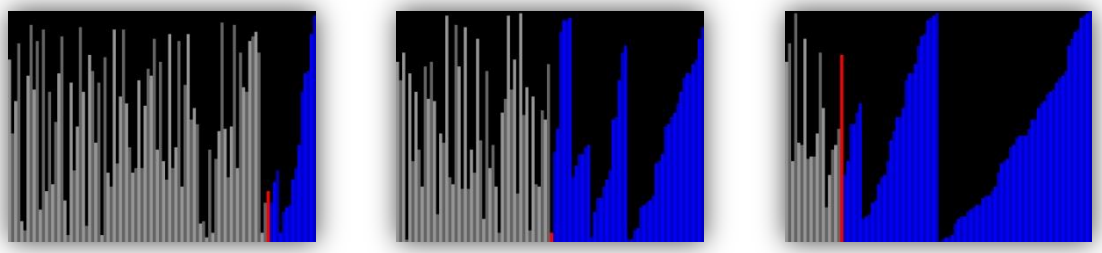


Exercice 9

Il n'y a pas non plus d'interversion dans le tri fusion, donc vous devez aussi utiliser `ecrase_valeur` plutôt qu'`echange_valeurs` .

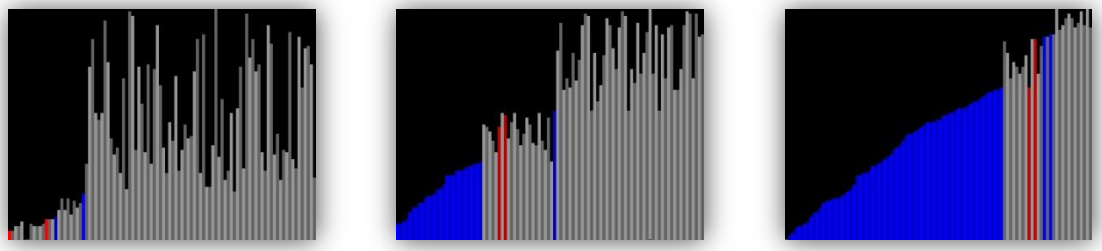
Vous remarquerez que l'implémentation de la bibliothèque `tri_tab` n'est pas tout à fait la même que celle vue en cours : elle a été modifiée pour permettre l'affichage que l'on veut réaliser aujourd'hui. En particulier, tous les tableaux manipulés sont de taille `N`, et les sous-tableaux sont spécifiés en utilisant des paramètres supplémentaires `debut` et `fin`, qui indiquent… le début et la fin du sous-tableau.

On utilisera la couleur bleue pour représenter la fin d'une étape de fusion, et ainsi indiquer l'emplacement final relativement à un sous-tableau, comme ci-dessous :



Exercice 10

Pareil pour le tri rapide, et cette fois vous devez utiliser `echange_valeur`.



Remarque : le tri par dénombrement ne se prête pas vraiment à ce type de représentation graphique, aussi il ne sera pas traité.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 59 | tris sur listes

Présentation

Dans ce TP, nous allons adapter certains des algorithmes de tri vus en cours, qui étaient destinés aux tableaux, afin de les appliquer à des listes. Pour cette raison, la bibliothèque `liste_d` est fournie avec ce sujet.

Remarque : pensez à la fonction `initialise_liste_d` pour initialiser rapidement une liste.

1 Tri par sélection

Exercice 1

Écrivez une fonction `element* identifie_extremum(liste_d l, int mode)` qui parcourt une liste et retourne un pointeur sur l'élément contenant sa valeur la plus extrême. Le mode indique de quel extremum il s'agit : 0 pour le minimum et 1 pour le maximum. Si la valeur extrême apparait plusieurs fois dans la liste, la fonction doit renvoyer le premier élément contenant cette valeur. Si la liste est vide, la fonction doit renvoyer `NULL`.

exemples : pour la liste (5,18,22,35) :

- Le mode 0 renvoie le premier élément (min) ;
- Le mode 1 renvoie le dernier élément (max).

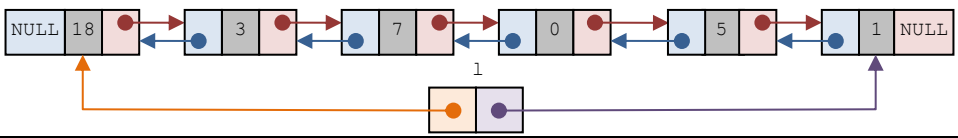
Exercice 2

Écrivez une fonction `void tri_selection_rec(liste_d *l)` qui effectue *récursivement* le tri de la liste `l` dans l'ordre croissant en utilisant l'algorithme du tri par sélection. La version récursive de cet algorithme est :

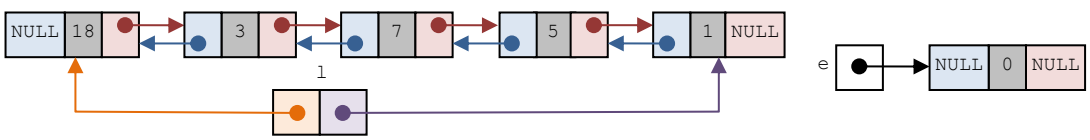
- Si la liste contient un seul élément (ou aucun) : elle est déjà triée.
- Sinon :
  - Identifier et extraire le minimum dans la liste de longueur n ;
  - Trier la sous-liste obtenue, dont la longueur est $n-1$;
  - Insérer le minimum au début de cette sous-liste triée, pour obtenir la liste complète triée.

exemple :

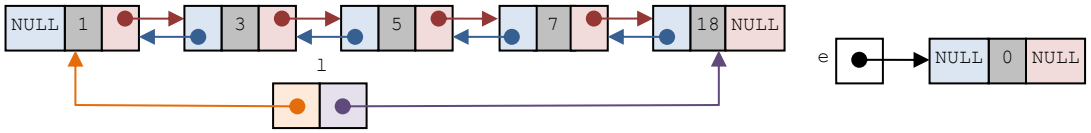
Liste initiale



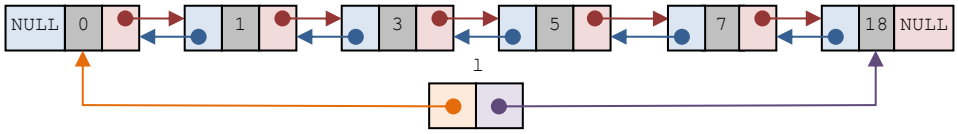
On identifie et on extraie le minimum de la liste



On trie récursivement la sous-liste



On insère le minimum au début de la sous-liste triée



Remarque : notez que la bibliothèque `liste_d` contient une fonction `int detache_element(liste_d *l, element_d *e)` (écrite lors d'un TP précédent) qui permet de retirer l'élément `e` de la liste `l`, sans supprimer l'élément `e`. Elle renvoie -1 en cas d'erreur.

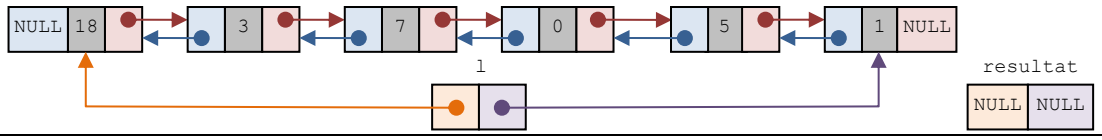
Exercice 3

Écrivez une fonction `void tri_selection_it(liste_d *l)` qui effectue le même travail que la fonction précédente, mais cette fois ci de manière *itérative*. La version itérative de cet algorithme est :

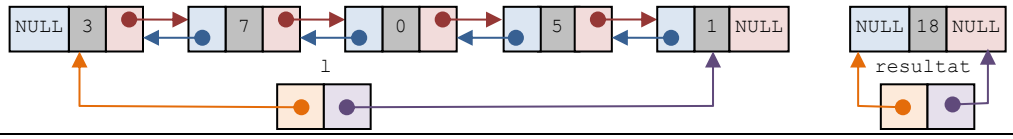
- Créer une liste vide résultat ;
- Tant que la liste `l` n'est pas vide :
  - Identifier et extraire son maximum ;
  - Insérer le maximum au début de la liste résultat ;

exemple :

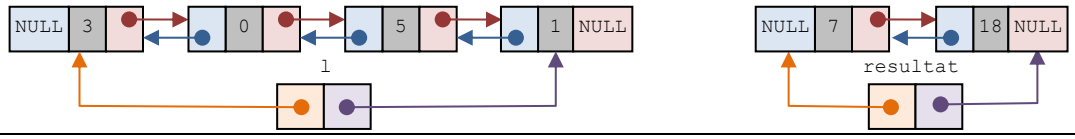
Liste initiale



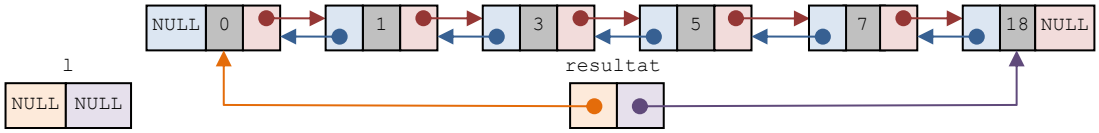
On identifie et on extraie le maximum de la liste, puis on l'insère au début de la liste résultat



On recommence



On continue, jusqu'à ce que la liste soit vide: la liste résultat obtenue est dans l'ordre croissant



2 Tri par insertion

Exercice 4

Écrivez une fonction `int insere_element_trie(liste_d *l, element_d *e)` qui insère un élément `e` au bon endroit dans une liste `l` triée dans l'ordre croissant. Vous ne

devez utiliser aucune fonction de la bibliothèque `liste_d`. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

exemple : si la fonction doit insérer 22 dans (5,18,35), alors on obtient (5,18,22,35).

Exercice 5

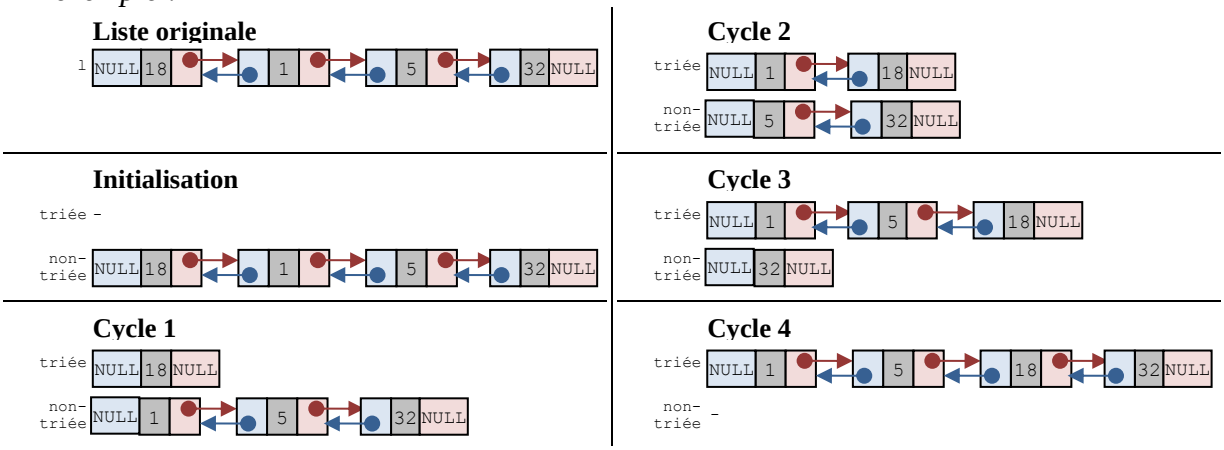
Écrivez une fonction itérative `void trie_insertion(liste_d *l)` qui applique l'algorithme du tri par insertion à une liste d'entiers `l`. Vous utiliserez la fonction `insere_element_trie` pour réaliser les insertions.

Pour rappel, le principe du tri par insertion est de diviser la liste initiale à trier en deux sous-listes : une sous-liste *triée* et une sous-liste *non-triée*.

- Initialisation :
  - La sous-liste triée est initialement vide ;
  - La sous-liste à trier contient tous les autres éléments.
- Traitement : pour chaque élément de la sous-liste non-triée :
  1. On enlève cet élément de la sous-liste non-triée ;
  2. On l'insère au bon endroit dans la sous-liste triée.

À la fin du traitement, la sous-liste non-triée est vide et la sous-liste triée contient tous les éléments de la liste initiale, rangés dans le bon ordre.

exemple :

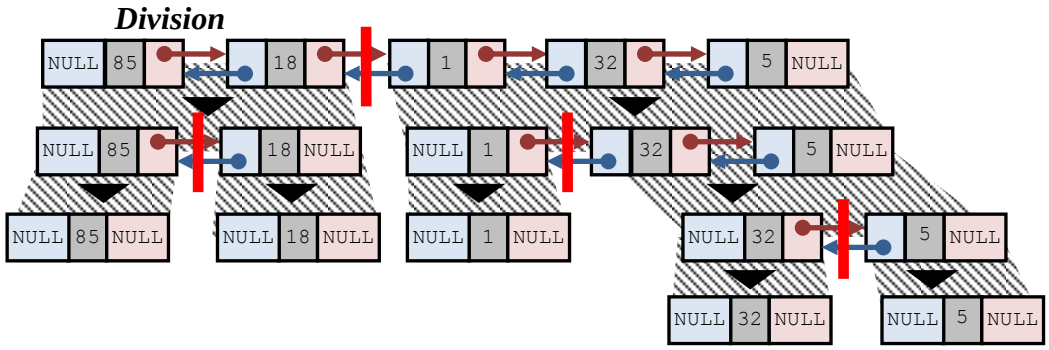


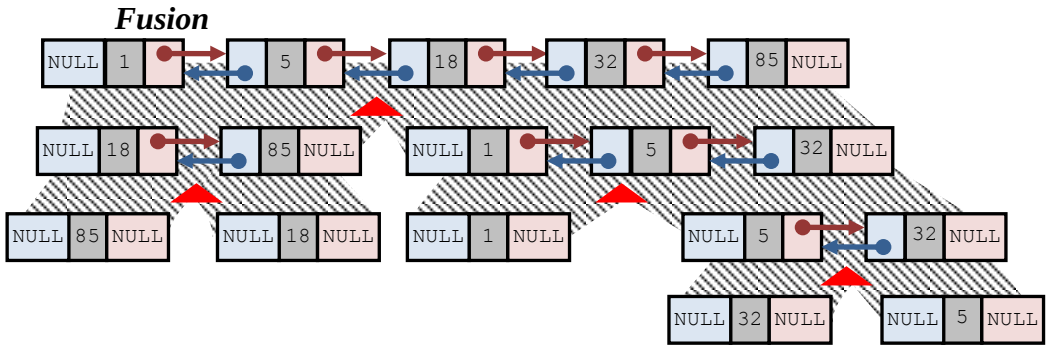
3 Tri fusion

Le principe de l'algorithme du tri fusion est récursif. Il est décrit par les cas suivants :

- Cas d'arrêt : la liste est vide ou ne contient qu'un seul élément (elle est déjà triée) ;
- Cas général : il se compose de trois phases :
  - *Division* : la liste est coupée en deux parties de même longueur (à un élément près) ;
  - *Tri* : chaque moitié de liste est triée séparément et récursivement ;
  - *Fusion* : les deux moitiés triées sont fusionnées pour obtenir une version triée de la liste initiale

exemple :





Exercice 6

Écrivez une fonction `element_d* renvoie_milieu(liste_d l)` prenant une liste `l` en paramètre et renvoyant un pointeur vers son élément central. Pour une liste de n éléments, on considère que l'élément central est le $(n/2+1)^{\text{ème}}$. On suppose que la liste reçue contient au moins deux éléments.

exemple : si la liste `l` est $(8;4;6;2)$ ou $(8;4;6;2;7)$, la fonction renvoie un pointeur sur l'élément contenant 6.

Exercice 7

Écrivez une fonction `void calcule_division(liste_d l, liste_d *lg, liste_d *ld)` prenant en paramètres une liste `l` contenant au moins deux éléments, et deux listes `lg` et `ld` initialisées mais vides. Votre fonction doit utiliser `renvoie_milieu` pour couper `l` en deux sous-listes `lg` et `ld` de même longueur, à un élément près.

exemple : si la liste `l` est $(8;4;6;2;7)$, alors `lg` et `ld` seront respectivement $(8;4)$ et $(6;2;7)$.

Exercice 8

Écrivez une fonction `void calcule_fusion(liste_d *l, liste_d lg, liste_d ld)` recevant en paramètre une liste `l` quelconque, et deux sous-listes déjà triées dans l'ordre croissant `lg` et `ld`. Votre fonction doit initialiser `l` avec l'union de `lg` et `ld`, tout en préservant l'ordre croissant. On suppose que `lg` et `ld` sont toutes les deux non-vides.

exemple : si `lg` est $(4;8)$ et `ld` est $(2;6;7)$ alors `l` est $(2;4;6;7;8)$.

Exercice 9

Écrivez une fonction *récursive* `void tri_fusion(liste_d *l)` qui prend en paramètre une liste `l` et la trie dans l'ordre croissant en appliquant l'algorithme du tri fusion. Vous utiliserez pour cela les fonctions `division` et `fusion` précédemment écrites.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 60 | représentation de l'ADN

Présentation

Le but de ce TP est de manipuler des listes représentant des brins d'ADN, de façon itérative et récursive. La fin du TP est consacrée au calcul de complexité algorithmique.

1 Codage de l'information

L'ADN (acide désoxyribonucléique) est une molécule utilisant une représentation de l'information spécifique (génotype) pour représenter les caractéristiques des êtres vivants et de certains virus (phénotypes). Cette information génétique est codée grâce à un alphabet de quatre bases azotées : adénine (A), cytosine (C), guanine (G) et thymine (T). On peut considérer un brin d'ADN comme une séquence formée avec ces quatre bases.

Exercice 1

On veut représenter un brin d'ADN par une liste simplement chaînée dont chaque élément représente une base azotée. Une base azotée sera représentée par un caractère (A, C, G ou T). Modifiez la structure et/ou les en-têtes de la bibliothèque `liste_s` fournie en annexe, de manière à l'adapter au cas de l'ADN.

Exercice 2

Écrivez une fonction *récursive* `int est_adn(liste_s brin)` qui teste si la liste `brin` passée en paramètre correspond bien à un brin d'ADN (i.e. est une séquence de bases azotées). La fonction renvoie 1 si la liste représente bien un brin d'ADN, et 0 sinon.

exemples :

- Pour la séquence A-T-G-C-G-C : la fonction doit renvoyer 1.
- Pour la séquence A-T-G-C-**D**-G : la fonction doit renvoyer 0.

Exercice 3

Écrivez une fonction `int genere_adn(liste_s *brin, int n)` qui crée un brin d'ADN en combinant aléatoirement les quatre bases de manière à obtenir une séquence de longueur `n`. Lors de l'appel de la fonction, la liste `brin` est vide. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

Vous devez utiliser les fonctions `srand(unsigned int)` et `rand()` déjà vues en TP, de manière à ce que le brin généré soit différent à chaque appel de la fonction.

2 Structure en double hélice

L'ADN est composé de deux brins se faisant face, et formant une double hélice (cf. la figure en début de sujet). Chaque base azotée d'un des brins fait face à sa base

complémentaire située sur l'autre brin. L'adénine et la thymine sont complémentaires, tandis que la cytosine et la guanine sont complémentaires.

exemple : les brins A-T-G-C-T-T-T-A-G et T-A-C-G-A-A-A-T-C sont complémentaires.

Exercice 4

Écrivez une fonction récursive `int calcule_complementaire(liste_s original, liste *complementaire)` qui prend en paramètre une liste `original` représentant un brin d'ADN, et une liste vide (mais correctement initialisée) `complementaire`. La fonction doit remplir `complementaire` de manière à obtenir une séquence complémentaire à `original`. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

exemple : pour la séquence A-T-G-C-A-A, la fonction doit renvoyer la séquence T-A-C-G-T-T.

Exercice 5

Écrivez une fonction *récursive* `int sont_egales(liste_s brin1, liste_s brin2)` qui compare deux brins d'ADN représentés par les listes `brin1` et `brin2`. La fonction renvoie 1 si les deux brins sont constitués de la même séquence de bases azotées, et 0 sinon.

exemples :

- Pour A-T-C et A-T-C : la fonction renvoie 1.
- Pour A-T-C et A-T-G : la fonction renvoie 0.
- Pour A-T-C et A-T-C-G : la fonction renvoie 0.

Exercice 6

En utilisant les deux fonctions précédentes, écrivez une fonction `int sont_complementaires(liste brin1, liste brin2)` qui teste si les deux brins passés en paramètres sont complémentaires. La fonction renvoie 1 si les deux brins sont complémentaires, 0 s'ils ne sont pas complémentaires, et -1 en cas d'erreur.

exemples :

- Pour A-T-C et T-A-G : la fonction renvoie 1.
- Pour A-T-C et A-A-G : la fonction renvoie 0.
- Pour A-T-C et T-A-G-C : la fonction renvoie 0.

Exercice 7

Calculez la complexité temporelle asymptotique de la fonction `sont_complementaires` dans le pire des cas (pensez à utiliser le formulaire donné en cours).

Remarque : on considère que les fonctions de la bibliothèque `liste_s` ont les complexités temporelles asymptotiques suivantes :

- Créer un élément ou insérer/supprimer/accéder au premier élément de la liste : $O(1)$.
- Afficher la liste ou insérer/supprimer/accéder à un élément quelconque de la liste : $O(N)$.

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 61 | nombres de grande taille

Présentation

Le but de ce TP est de manipuler les listes chaînées et le traitement récursif, à travers la représentation et le calcul de nombres de grande taille. Pour cette raison, la bibliothèque `liste_s` est fournie avec ce sujet.

Pour simplifier les calculs de complexité, on considèrera que toutes les fonctions contenues dans cette bibliothèque ont une complexité asymptotique temporelle dans le pire des cas en $O(1)$.

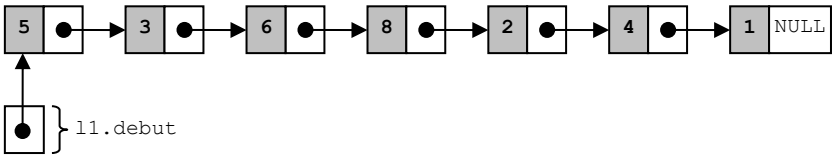
Notez que la bibliothèque `liste_s` contient une fonction `initialise_liste_s` permettant d'initialiser rapidement une liste à partir d'un tableau d'entier. L'ordre des valeurs dans le tableau est reproduit dans la liste.

1 Représentation

Les types simples du langage C ne permettent pas de manipuler des entiers de grandes tailles, c'est-à-dire des entiers dont l'écriture comporte beaucoup de chiffres. Une solution consiste à représenter un tel nombre sous la forme d'une liste doublement chainée, dans laquelle chaque élément correspond à un chiffre.

De plus, pour des raisons pratiques, on préfèrera placer les chiffres dans l'ordre inverse de l'ordre habituel (i.e. en partant du chiffre de poids le plus faible, au lieu de celui de poids le plus fort). En effet, les nombres entiers sont manipulés la plupart du temps en partant des unités. C'est notamment le cas pour les opérateurs arithmétiques de type addition, multiplication, soustraction, etc. Le fait de renverser le nombre permet de placer les unités en premier dans la liste.

exemple : l'entier 1428635 est représenté par la liste `l1` suivante :



Exercice 1

Écrivez une fonction *récursive* `int verifie_gn(liste_s l)` qui teste que chaque élément de la liste `l` est bien un chiffre (et non pas un nombre). La fonction renvoie 1 si la liste est bien un grand nombre, et 0 sinon.

exemple : pour la liste `l1` de l'exemple précédent, le résultat de `verifie_nombre(l1)` sera 1. Si le premier élément de `nbre` était 51 au lieu de 5, alors la fonction renverrait 0.

Exercice 2

Écrivez une fonction *récursive* `int compte_chiffres(liste_s l)` qui prend en paramètre une liste `l` représentant un grand nombre, et compte le nombre de chiffres qui le constituent.

exemple : pour la liste `l1` de l'exemple précédent, `compte_chiffres(l1.debut)` doit retourner 7.

Quel est le type de récursivité que vous avez utilisé ? Calculez la complexité asymptotique temporelle dans le pire des cas.

2 Affichage et conversion

Exercice 3

Écrivez une fonction `void affiche_gn_it(liste_s l)` qui affiche le grand nombre représenté par `l`. Pour des raisons de lisibilité, un caractère espace `' '` doit être affiché tous les 3 chiffres. Cette fois, vous avez le droit d'utiliser `compte_chiffres`.

exemple : l'affichage de la liste `l1` précédente donne le résultat suivant :

```
1 428 635
```

Exercice 4

Écrivez une version récursive de la même fonction, appelée cette fois `void affiche_gn_rec(liste_s l, int cpt)`. Pour cette fonction, vous n'avez plus le droit d'utiliser `compte_chiffres`. Le paramètre `cpt` sert à compter le nombre de chiffres traités ; il doit valoir 1 lors du premier appel.

exemple : l'appel `affiche_gn_rec(l1,1)` donne le même affichage que ci-dessus.

Exercice 5

Écrivez une fonction `int convertis_chaine(char* chaine, liste_s* l)` qui reçoit un grand nombre représenté sous la forme d'une chaîne de caractères `chaine`, et qui la convertir en une liste chaînée `l`. Bien sûr, chaque caractère doit être convertit en un chiffre. La fonction retourne -1 en cas d'erreur et 0 en cas de succès.

exemple : pour `"1428635"`, on obtient la liste `l1` de l'exemple précédent.

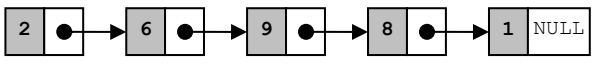
Remarque : on suppose que la chaîne ne contient que des chiffres.

3 Opérations

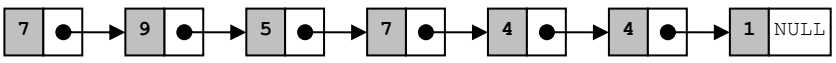
Exercice 6

Écrivez une fonction `int additionne_gn(liste_s l1, liste_s l2, liste_s *res)` qui additionne les entiers représentés par les listes `l1` et `l2`, et place le résultat dans la liste `res`, qui est passée par adresse. La fonction doit renvoyer -1 en cas d'erreur et 0 en cas de succès.

exemple : pour la listes `l1` des exemples précédents et la liste `l2` représentant la valeur 18962 :



alors on obtient le résultat `res` suivant, représentant la valeur 1447597 :

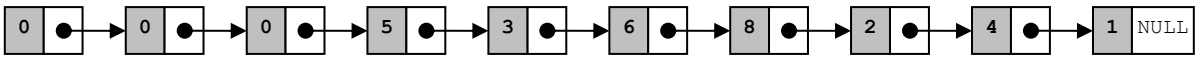


Remarque : pensez à compter les retenues lors de l'addition de deux chiffres.

Exercice 7

Écrivez une fonction *récursive* `int multiplie_gn_p10(liste_s *l, int p)` qui multiplie par 10^p le grand nombre représenté par la liste `l` passée en paramètre. Cette fonction doit renvoyer -1 en cas d'erreur et 0 en cas de succès.

exemple : si la fonction est appliquée à la liste `l1` (donnée en exemple précédemment) et au paramètre $p = 3$, on obtient comme résultat la liste suivante, qui représente le grand nombre 1428635000 :

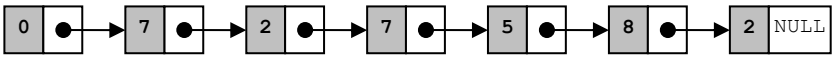


Quel est la taille des données pour cette fonction ? Calculez la complexité asymptotique temporelle dans le pire des cas.

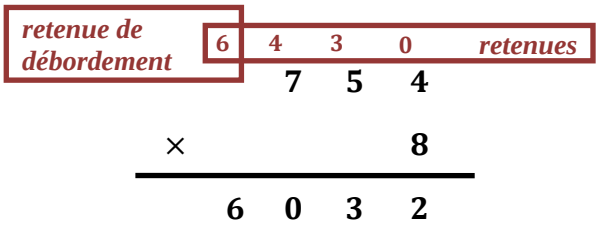
Exercice 8

Écrivez la fonction *récursive* `int multiplie_gn_chiffre(liste_s l, int c, liste_s *res)` qui reçoit en paramètres une liste `l` représentant un grand nombre, une liste vide `res`, et un chiffre `c` (i.e. $0 \leq c \leq 9$). La fonction doit calculer le produit des valeurs représentées par `l` et `c`, et placer le résultat dans `res`. Cette fois, la liste `l` ne doit pas être modifiée.

exemple : si on multiplie la liste précédente `l1` par 2, alors on obtient la liste suivante, qui représente la valeur 2857270 :

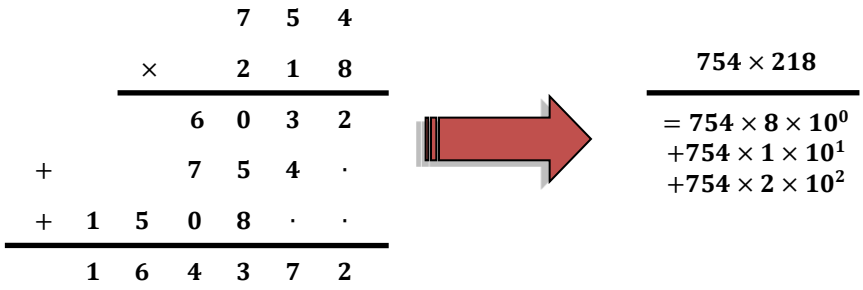


Remarque : là-encore, pensez à tenir compte des retenues :



Exercice 9

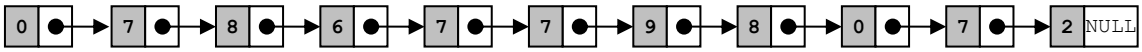
On remarque que le produit de deux nombres à plusieurs chiffres peut se décomposer en une somme de produits de la façon suivante :



En utilisant ce principe, écrivez la fonction `int multiplie_gn(liste_s l1, liste_s l2, liste_s *res)` qui reçoit en paramètres deux listes `l1` et `l2` correspondant à des grands nombres, et une liste vide `res`. La fonction doit calculer le produit des valeurs représentées par `l1` et `l2`, et placer le résultat dans `res`. La fonction ne doit pas modifier `l1` ni `l2`.

Remarque : vous pouvez utiliser les fonctions écrites lors des exercices précédents. Pensez également à utiliser la fonction `vide_liste_s` de la bibliothèque `liste_s`, qui permet de supprimer tous les éléments contenus dans une liste.

exemple : le produit des listes `l1` par `l2`, utilisées dans les exemples précédents, donne la liste suivante, correspondant à la valeur 27089776870 :



4 Tri

Exercice 10

Écrivez la fonction *récursive* `int compare_gn(liste l1, liste l2)` qui prend en paramètres deux listes `l1` et `l2` représentant des grands nombres, et renvoie un entier dépendant de leur comparaison :

- Une valeur négative quelconque si $l_1 < l_2$;
- La valeur zéro si $l_1 = l_2$;
- Une valeur positive quelconque si $l_1 > l_2$.

exemple : pour les listes `l1` et `l2` des exemples précédents, on obtient une valeur strictement positive.

Exercice 11

Soit la fonction `tri` suivante, qui trie un tableau d'entiers de taille `N` :

```
void tri(int tab[N])
{  int i,j,temp;
   for(i=1;i<N;i++)
   {  temp = tab[i];
      j = i-1;
      while(j>=0 && tab[j]>=temp)
      {  tab[j+1] = tab[j];
         j--;
      }
      tab[j+1] = temp;
   }
}
```

Quel est l'algorithme de tri (étudié en cours) implémenté par cette fonction ? Donnez, sans la calculer (ça a déjà été fait en cours), sa complexité asymptotique temporelle dans le pire des cas.

On veut appliquer le même traitement à un tableau contenant des grands nombres. Adaptez la fonction précédente de manière à obtenir la fonction `void trie_gn(liste_s tab[N])`, qui trie un tableau de `N` grands nombres.

exemple : pour $N = 5$ et les valeurs 1428635, 3649492, 9651105, 1121111 et 2233222, l'affichage du tableau trié est :

```
1. 1 121 111
2. 1 428 635
3. 2 233 222
4. 3 649 492
5. 9 651 105
```

Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 62 | table de symboles

Présentation

Le but de ce TP est de créer un programme capable de lire un fichier texte et de calculer la *fréquence lexicale* de chaque mot qu'il contient. Pour cela, vous définirez et utiliserez une structure de données spécifique appelée *table de symboles*, en vous basant sur une liste chaînée. La bibliothèque `liste_s` est fournie avec ce sujet à cet effet. Vous utiliserez également la bibliothèque `chaine`, qui permet de manipuler des chaînes de caractères.

1 Structure de données

La fréquence lexicale d'un mot correspond à son nombre d'occurrences, i.e. au nombre de fois qu'il apparait dans le texte. Par exemple, le mot "vous" apparait 2 fois dans la section d'avertissement.

De façon générale, une table de symboles permet d'associer une *valeur* (ou un ensemble de valeurs) à un objet unique appelé *clé* (ou symbole). Deux éléments de la table ne peuvent pas avoir la même clé, par contre ils peuvent avoir des valeurs identiques.

Vous allez implémenter une version simplifiée d'une table de symboles en utilisant une liste simplement chaînée. Dans notre cas, chaque clé correspond à un mot (ex. : "vous"), et la valeur associée est la fréquence lexicale de ce mot (ex. : 2).

Exercice 1

Renommez la bibliothèque `liste_s` en `liste_s_chaine` et modifiez-la de manière à l'adapter au problème traité ici. Vous effectuerez les modifications que vous jugerez nécessaires sur les structures de données et les fonctions existantes, de manière à ce qu'une liste soit formée d'éléments contenant chacun :

- Une clé prenant la forme d'une chaîne de caractères `cle` ;
- Une valeur entière `valeur`.

Créez un fichier `main.c` contenant une fonction `main` dans laquelle vous testerez ces modifications en créant, initialisant et affichant une liste similaire à l'exemple ci-dessous.

exemple : affichage d'une liste contenant trois clés `"cle1"`, `"cle2"` et `"cle3"` de valeurs respectives 2, 0 et 5

```
{ cle1:2 cle2:0 cle3:5 }
```

Exercice 2

Créez une bibliothèque `table_symb`. Définissez-y un type `tabsymb` pour représenter une table de symboles à partir d'une liste. Sur le modèle de ce qui a été fait en cours et en TP avec les *piles* et *files* de données, ajoutez à cette bibliothèque une fonction `cree_tabsymb` permettant de créer et d'initialiser une nouvelle table de symboles vide.

Exercice 3

Dans la bibliothèque `table_symb`, écrivez une fonction `int ajoute_symb(tabsymb *t, char *cle)` qui prend en paramètre une table de symboles `t` et lui rajoute un élément dont la clé est `cle`.

- Si aucun élément contenant cette clé n'existe dans la liste, un nouvel élément est inséré avec une valeur de 1. **Attention :** il est essentiel d'initialiser cet élément avec une copie de la chaîne de caractères `cle` (obtenue avec `copie_chaine_dyn`, de la bibliothèque `chaine`), et non pas avec `cle` elle-même.
- S'il existe déjà un élément avec cette clé, la valeur associée est alors simplement incrémentée.

La fonction renvoie -1 en cas d'erreur et 0 en cas de succès. Notez que l'ordre des éléments dans la liste n'a aucune importance.

Dans la fonction `main`, testez `cree_tabsymb` et `ajoute_symb` : créez une table de symboles, ajoutez-y les éléments décrits dans l'exemple suivant, et affichez la table après chaque insertion.

exemple : affichage obtenu après l'ajout successif des clés `vous`, `truc`, `vous` et `mien`. Notez que l'affichage est réalisé dans la fonction `main`, et non pas dans `ajoute_symb`.

```
Ajout de la clé "vous"
{ vous:1 }
Ajout de la clé "mien"
{ mien:1 vous:1 }
```

```
Ajout de la clé "vous"
{ truc:1 vous:2 }
Ajout de la clé "truc"
{ truc:1 mien:1 vous:2 }
```

Exercice 4

Dans la bibliothèque `table_symb`, écrivez une fonction `int verifie_symb(tabsymb t, char *symb)` qui prend en paramètre une table de symboles `t` et renvoie la valeur qui y est associée au symbole dont la clé est `symb`. Si aucun élément contenant cette clé n'existe dans la liste, la valeur renvoyée est 0 (i.e. aucune occurrence).

Dans la fonction `main`, testez votre fonction en l'appliquant à la table de l'exercice précédent et aux clés `vous` et `notre`.

exemple : (affichage réalisé dans la fonction `main`)

```
Recherche de la clé "vous" : 2 occurrence(s)
Recherche de la clé "notre" : 0 occurrence(s)
```

2 Analyse du texte

Exercice 5

Dans la bibliothèque `table_symb`, écrivez une fonction `int analyse_texte(char *nom_fichier, tabsymb *t)` qui ouvre en lecture le fichier portant le nom `nom_fichier`, lit sont contenu et réalise son analyse. Cette analyse consiste à ajouter chaque mot dans la table de symboles `t`. Comme d'habitude, la fonction renvoie -1 en cas d'erreur et 0 en cas de succès. L'accès au fichier doit obligatoirement se faire en mode *formaté*.

Testez votre fonction en l'appliquant depuis la fonction `main` au fichier texte `texte.txt` fourni avec le sujet. Il s'agit d'une partie de l'article de Wikipédia consacré au langage C. Notez que ce fichier a été prétraité pour ne contenir que des lettres minuscules afin de vous simplifier le travail. Affichez la table de symboles obtenue.

exemple : application au texte fourni avec le sujet et affichage de la table obtenue :

```
{ javascript:1 citer:1 peut:1 enrichi:1 extension:1 origine:1 celle:1...
```

3 Tri des symboles

Le texte analysé contient de nombreux mots, il est difficile d'identifier lesquels sont les plus importants. On se propose d'ordonner la liste en fonction de leurs occurrences, i.e. en considérant le champ `valeur` des éléments constituant la liste.

Exercice 6

Dans `liste_s_chaine`, écrivez une fonction `int compare_elements(element_s *e1, element_s *e2)`, qui permet de comparer les deux éléments passés en paramètres. La fonction renvoie un entier :

- *Négatif* si la valeur contenue dans `e1` est inférieure à celle de `e2`.
- *Positif* si la valeur de `e1` est supérieure à celle de `e2`.

Si les deux valeurs sont *égales*, on compare les *clés* contenues dans les éléments. La fonction renvoie alors un entier :

- *Négatif* si la clé contenue dans `e1` est placée avant celle de `e2` dans l'ordre lexicographique ;
- *Positif* si la clé de `e1` est après celle de `e2`.
- *Nul* si les deux clés sont exactement les mêmes.

Remarque : ce dernier cas est traité pour être exhaustif, mais dans le cadre de ce TP, la fonction ne peut pas renvoyer 0 puisqu'une clé ne peut pas apparaitre plusieurs fois dans la table de symboles.

exemples :

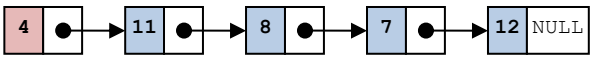
- Pour `e1={"vous",1}` et `e2={"moi",4}`, on obtient une valeur *négative* car `1<4`.
- Pour `e1={"vous",1}` et `e2={"moi",1}`, on obtient une valeur *positive* car `1=1` et `"vous">"moi"`.

Exercice 7

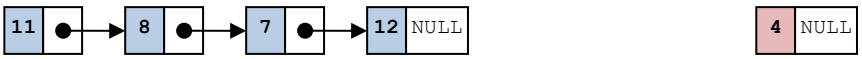
Dans `liste_s_chaine`, écrivez une fonction `element_s* etete_liste(liste_s *l)` qui étête la liste passée en paramètre, c'est-à-dire qui en extrait le premier élément (i.e. l'élément de tête). Attention, cet élément ne doit pas être désalloué : seulement sorti de la liste.

exemple : pour simplifier la figure, on ne représente que les clés des éléments :

- Supposons qu'on applique la fonction à la liste suivante :



- Alors on obtient la liste et l'élément suivants :



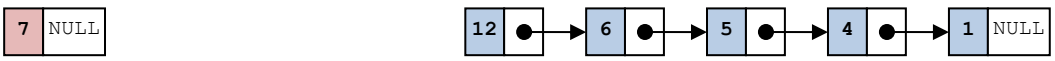
Remarque : on suppose que la liste n'est *pas* vide.

Exercice 8

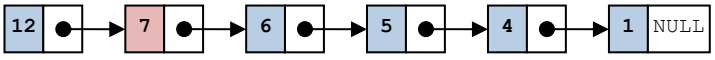
Dans `liste_s_chaine`, écrivez une fonction `void insere_element_dec(liste_s *l, element_s *e)` qui reçoit une liste `l`, triée dans l'ordre décroissant, et un élément `e` ; et qui insère `e` à la position appropriée dans `l`, de manière à ce que la liste reste triée après cette opération.

exemple :

- Considérons qu'on insère l'élément suivant dans la liste suivante :



- Alors on obtient la liste suivante :



Exercice 9

Dans `liste_s_chaine`, écrivez une fonction `void trie_liste_dec(liste_s *l)` qui trie la liste passée en paramètre dans l'ordre *croissant*, en utilisant le principe du *tri par insertion* vu en cours pour des tableaux.

Rappel : le principe du tri par insertion est de séparer la séquence à trier en deux partie : une partie triée, qui contient initialement seulement le premier élément, et une partie non-triée, qui contient initialement tout le reste de la liste. On considère ensuite chaque élément de la partie non-triée, et on l'insère au bon endroit dans la partie triée.

Dans notre cas, on manipule des listes, et on peut donc utiliser cette structure de données pour représenter chaque partie comme une liste séparée. Attention, ici la liste triée doit l'être dans l'ordre *décroissant*, et non pas l'ordre croissant habituel du cours.

Algorithme :

- Initialisation :
  - Créez une nouvelle liste `l2` contenant toute la liste originale `l` sauf le premier élément.
  - Dans la liste originale `l`, ne gardez que le premier élément.
  - La nouvelle liste correspond à la partie non-triée, la liste originale à la partie triée.
- Itération : pour chaque élément de la nouvelle liste `l2` :
  - On extrait cet élément de `l2` avec `etete_liste` ;
  - On l'insère au bon endroit dans `l` avec `insere_element`.

exemple : pour la liste du premier exercice, l'affichage de la liste triée donne :

```
{ cle3:5 cle1:2 cle2:0 }
```

4 Filtrage des symboles

Utilisez la fonction `trie_liste_dec` pour trier la table de symboles obtenue à partir du texte. En affichant cette table triée, on peut identifier les mots les plus fréquents du texte. On s'aperçoit alors que la plupart d'entre eux sont génériques, et n'ont pas un sens permettant de caractériser le texte. Par exemple, les premiers mots obtenus sont :

```
{ de:36 des:17 la:14 le:13 c:13 les:12 langage:12 d:12 et:10 en:10 un:9 est:9...
```

Les mots du type *à*, *le*, *je*, etc., sont appelés des *mots vides*. On veut maintenant supprimer ces mots de notre table de symboles, afin de ne garder que les mots intéressants.

Exercice 10

Le fichier `motsvides.txt` fournis avec ce sujet contient une liste des principaux mots vides de la langue française. Écrivez une fonction `int filtre_mots_vides(char *nom_fichier, tabsymb *t)` qui réalise les opérations suivantes :

- On ouvre le fichier `nom_fichier` en lecture, qui correspond à une séquence de mots (i.e. dans le cas présent : le fichier `motsvides.txt`).
- On lit itérativement chaque mot contenu dans le fichier. Notez que chaque mot se trouve sur une ligne distincte. Pour chaque mot :
  - On cherche le mot parmi les clés de la table de symboles `t`.
  - Si on le trouve, l'élément correspondant doit être supprimé de la table.
- Le traitement se termine quand on a traité tous les mots contenus dans le fichier.
- On ferme le fichier.

La fonction renvoie -1 en cas d'erreur, et 0 en cas de succès. Testez votre fonction sur la table précédente, à partir de la fonction *main*, et affichez la liste obtenue.

exemple : le début de la liste devrait correspondre à :

```
{ langage:12 est:9 programmation:6 niveau:5 langages:5 types:4 systeme:4...
```

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 63 | plus longue sous-séquence commune

Présentation

Le but de ce TP est d'approfondir les notions de complexité et de récursivité, via la détection de plus longue séquence communes. C'est aussi l'occasion d'introduire le concept de programmation dynamique.

1 Notion de PLSC

Soit $X = (x_0, \dots, x_{p-1})$ une *séquence* de p éléments x_i. On appelle *sous-séquence* de X toute séquence obtenue en supprimant zéro ou plusieurs éléments de X.

exemples : pour $X = (2,7,1,8,2,8,5,1)$:

- $(7,8,8)$ et $(\ )$ sont des sous-séquences de X ;
- $(8,2,1)$ n'en est pas une.

Soit $Y = (y_0, \dots, y_{q-1})$ une autre séquence. On appelle *sous-séquence commune* à X et Y toute séquence qui est à la fois une sous-séquence de X et une sous-séquence de Y.

exemples : pour le X précédent et $Y = (2,4,7,3,9,8,6,1)$

- $(\ )$ est, par définition, toujours une sous-séquence commune à X et Y ;
- $(2,8)$ et $(7,8,2)$ sont deux sous-séquences communes à X et Y ;
- Ce n'est pas le cas de $(4,7,8)$ ou $(2,7,6)$.

Une *plus longue sous-séquence commune* (PLSC) à X et Y est une sous-séquence commune à X et Y de longueur maximale. On la note $PLSC(X,Y)$.

Remarque : X et Y peuvent avoir plusieurs PLSC différentes, mais toutes ces PLSC ont la même longueur.

exemple : pour les X et Y des exemples précédents, $(2,7,8,1)$ est une $PLSC(X,Y)$.

2 Longueur d'une PLSC

Soient $X = (x_0, \dots, x_{p-1})$ et $Y = (y_0, \dots, y_{q-1})$ deux séquences. On utilise les notations suivantes :

- Sous-séquence de X : $X_i = (x_0, \dots, x_{i-1})$ pour $1 \leq i \leq p$.
- Sous-séquence de Y : $Y_j = (y_0, \dots, y_{j-1})$ pour $1 \leq j \leq q$.
- Longueur de $PLSC(X_i, Y_j)$: $l_{i,j}$.

De plus, on pose : $X_0 = (\ )$ et $Y_0 = (\ )$.

Exercice 1

Montrez que pour $1 \leq i \leq p$ et $1 \leq j \leq q$:

$$l_{i,j} = \begin{cases} 1 + l_{i-1,j-1} \text{ si } x_{i-1} = y_{j-1} \\ \max(l_{i-1,j}, l_{i,j-1}) \text{ sinon} \end{cases}$$

3 Approche naïve

Exercice 2

Nous allons manipuler des séquences d'entiers représentées par des tableaux. En utilisant le principe récursif de l'exercice précédent, écrivez la fonction `int plsc1(int tab1[], int taille1, int tab2[], int taille2)`. Cette fonction renvoie la taille de la plus longue sous-séquence commune aux tableaux `tab1` et `tab2`, de tailles respectives `taille1` et `taille2`.

Exercice 3

Donnez l'arbre d'appels pour $X = (1,4,5)$ et $Y = (6,2)$.

Exercice 4

Calculez la complexité temporelle de la fonction `plsc1`, en fonction de p et q.

4 Programmation dynamique

Exercice 5

À partir de la version naïve de la fonction, écrivez une fonction plus efficace `int plsc2(int tab1[], int taille1, int tab2[], int taille2, int l[P+1][Q+1])`, qui calcule la matrice L de taille $(p+1) \times (q+1)$ telle que :

$$L = \begin{bmatrix} l_{0,0} & \cdots & l_{0,q} \\ \vdots & \ddots & \vdots \\ l_{p,0} & \cdots & l_{p,q} \end{bmatrix}$$

La fonction doit être optimisée de manière à ne *pas* effectuer plusieurs fois le même calcul.

Exercice 6

Calculez la complexité temporelle de la fonction `plsc2`, en fonction de p et q.

5 Construction d'une PLSC

La matrice L obtenue grâce à `plsc2` peut être utilisée pour construire une PLSC. Par exemple, soient $X = (1,5,3,4,5)$ et $Y = (3,4,2,5)$.

$L_{i,j}$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	1
3	0	1	1	1	1
4	0	1	2	2	2
5	0	1	2	2	3

- Voici la matrice L obtenue.
- X est représenté verticalement et Y horizontalement.

$L_{i,j}$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	1
3	0	1	1	1	1
4	0	1	2	2	2
5	0	1	2	2	3

- On peut observer que $L_{5,4} = 3$ et que $x_4 = y_3 = 5$.
- Intuitivement, on en déduit qu'il existe une PLSC Z telle que $PLSC(X_5, Y_4) = (z_0, z_1, 5)$.
- D'où : $PLSC(X_4, Y_3) = (z_0, z_1)$.

$L_{i,j}$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	1
3	0	1	1	1	1
4	0	1	2	2	2
5	0	1	2	2	3

- On peut observer que $x_3 \neq y_2$ et $L_{4,3} = 2$.
- Or $L_{4,2} = 2$ et $L_{3,3} = 1$ i.e. $L_{4,2} = L_{4,3}$.
- On en déduit que $PLSC(X_4, Y_3) = PLSC(X_4, Y_2) = (z_0, z_1)$.

$L_{i,j}$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	1
3	0	1	1	1	1
4	0	1	2	2	2
5	0	1	2	2	3

- On peut observer que $x_3 = y_1 = 4$ et $L_{4,2} = 2$.
- Intuitivement, on en déduit qu'il existe une PLSC telle que $PLSC(X_4, Y_2) = (z_0, 4)$.
- D'où : $PLSC(X_3, Y_1) = (z_0)$.

$L_{i,j}$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	1
3	0	1	1	1	1
4	0	1	2	2	2
5	0	1	2	2	3

- On peut observer que $x_2 = y_0 = 3$ et $L_{3,1} = 1$.
- Intuitivement, on en déduit qu'il existe une PLSC telle que $PLSC(X_3, Y_1) = (3)$.
- D'où : $PLSC(X_2, Y_0) = (\quad)$.
- Le traitement s'achève donc ici.

$L_{i,j}$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	1
3	0	1	1	1	1
4	0	1	2	2	2
5	0	1	2	2	3

- On obtient $z_2 = 5$, $z_1 = 4$ et $z_0 = 3$.
- Donc finalement $PLSC(X, Y) = (3,4,5)$.

Exercice 7

Formalisez (sans le démontrer) un principe récursif permettant de construire une PLSC à partir de la matrice L.

Exercice 8

Grâce au principe obtenu à la question précédente, écrivez une fonction `int calcule_plsc(int tab1[P], int tab2[Q], int tab3[])` qui calcule une PLSC de `tab1` et `tab2` et la stocke dans `tab3` (on suppose que la taille de `tab3` est suffisante). La valeur renvoyée par la fonction correspond à la longueur des PLSC.

Pour aboutir à ces résultats, la fonction doit réaliser les opérations suivantes :

1. Initialiser L avec la valeur -1 ;
2. Appeler `plsc2` pour calculer L ;
3. Appliquer l'algorithme décrit ci-dessus.

Faculté d'ingénierie et de technologie – Génie informatique

Algorithmique et programmation

Damien Berthet & Vincent Labatut

TP 64 | arbres binaires

Présentation

Le but de ce TP est de manipuler des arbres binaires, d'abord simples via des opérations relativement simples, puis des arbres binaires de recherche, via une variante du tri par insertion déjà étudié pour les tableaux et listes.

Remarque : pour manipuler les arbres, vous ne devez utiliser que les fonctions appartenant aux types abstraits définis en cours.

Vous aurez aussi besoin de récupérer la fonction `genere_entier` contenue dans la bibliothèque `tri_tab` utilisées lors de TP précédents. Copiez-collez-la dans votre propre fichier `main.c`.

1 Arbres binaires

Exercice 1

Écrivez une fonction *récursive* `arbre genere_arbre_complet(int h)` qui génère aléatoirement un arbre binaire *complet* de hauteur `h`, contenant des valeurs comprises entre 0 et 99 (incluses). Pour rappel, dans un arbre complet, tous les nœuds sauf les feuilles possèdent le même nombre de fils (ici : `2`, puisqu'il s'agit d'un arbre binaire).

Remarque : utilisez `genere_entier` de `tri_tab`.

exemple : un arbre généré pour `h=4`, affiché avec `affiche_arbre` :

```
76
╠═98
│ ╠═15
│ │ ╠═99
│ │ ╚═4
│ ╚═13
│   ╠═8
│   ╚═6
╚═62
  ╠═99
  │ ╠═65
  │ ╚═97
  ╚═49
    ╠═31
    ╚═72
```

Exercice 2

Écrivez une fonction *récursive* `arbre genere_arbre_degenere(int h)` qui génère aléatoirement un arbre binaire *dégénéré* de hauteur `h`, contenant des valeurs comprises entre 0 et 99 (incluses). Pour rappel, un arbre dégénéré est un arbre linéaire.

exemple : un arbre généré pour `h=4`, affiché avec `affiche_arbre` :

```
52
╚═32
  ╚═89
    ╚═57
```

Exercice 3

Écrivez une fonction *récursive* `arbre genere_arbre_aleatoire(int n)` qui génère aléatoirement un arbre binaire dont à la fois les valeurs et la *structure* sont aléatoires. L'arbre doit contenir `n` nœuds.

Pour déterminer la structure, on utilise la méthode suivante. On tire au sort une valeur dans $\{0; 1; 2\}$:

- Si c'est 0, on crée une seule branche gauche, contenant $n - 1$ nœuds.
- Si c'est 1, on fait pareil mais pour la branche droite.
- Si c'est 2, on crée deux branches contenant chacune la moitié des $n - 1$ nœuds restants.

exemple : un arbre généré pour `n=18`, affiché avec `affiche_arbre` :

```
14
└─47
  ├─71
  │ └─40
  │   └─83
  │     ├─18
  │     │ └─89
  │     └─51
  │       └─12
  │         └─62
  └─45
    └─78
      └─5
        ├─17
        │ └─13
        └─80
          ├─95
          └─15
```

Exercice 4

Écrivez une fonction `int calcule_hauteur(arbre a)` qui calcule récursivement la hauteur d'un arbre `a`.

exemple : pour l'arbre généré dans l'exercice précédent, on obtient la valeur 7.

Exercice 5

Écrivez une fonction `int compte_noeuds(arbre a)` qui compte récursivement le nombre de nœuds dans un arbre `a`.

exemple : pour l'arbre généré dans l'exercice précédent, on obtient la valeur 18.

2 Arbres binaires de recherche

Il est relativement simple d'implémenter un algorithme de tri en utilisant un arbre binaire *de recherche*, puisque par définition, cette structure de données stocke les valeurs de façon ordonnée.

Le tri se fait en deux étapes :

- On crée un arbre vide et on y insère tous les éléments à trier.
- On construit la séquence triée en effectuant un parcours infixe de l'arbre.

Exercice 6

Écrivez une fonction `void tableau_vers_arbre(int tab[N], arbre *a)` qui recopie le contenu d'un tableau `tab` non-trié de taille `N` dans un arbre de recherche `a`, qui doit être initialement vide.

Exercice 7

Écrivez une fonction *récursive* `void arbre_vers_tableau(arbre a, int tab[N], int *index)` qui recopie dans le tableau `tab` les valeurs contenues dans l'arbre binaire de recherche `a`, en effectuant un parcours infixe. Le paramètre `index` est un pointeur sur le numéro de la prochaine case du tableau qu'il faudra remplir (`index` doit donc pointer initialement sur le premier élément du tableau).

Exercice 8

Écrivez une fonction `void tri_arbre(int tab[N])` qui utilise `tableau_vers_arbre` et `arbre_vers_tableau` pour trier le tableau `tab`.

Remarque : dans le principe, ce tri est similaire au tri par insertion, à la différence qu'on insère dans un arbre au lieu d'insérer directement dans une structure séquentielle comme un tableau ou une liste. Par conséquent, en raison des restrictions pesant sur ces arbres, les valeurs à trier doivent être uniques (pas de répétition).