




Devoir maison n°1 - À faire pour le 15 Novembre

Modéliser un distributeur automatique de produits

Notions abordées

- manipulation de structures, de tableaux, statiques et dynamique
- découpage d'un gros code en plusieurs fichiers
- compilation séparée
- utilisation d'un Makefile

 Ce projet peut être réalisé seul, en binôme ou en trinôme. Les différentes parties bonus peuvent être développées par des personnes différentes du groupe, mais les questions de base doivent être maîtrisées par tous les membres du groupe.

Selon votre aisance en programmation, vous pourrez passer les questions notées  , qui servent essentiellement à la compréhension ou au contraire passer les questions, notées * ou * * selon leur difficulté, qui servent à peaufiner le programme mais ne sont pas essentielles. Bien sûr il est tout à fait possible de s'attaquer aux questions * ou * * même si on a eu besoin de faire les questions  . Dans tous les cas, **il faut assurer les questions notées  (comme une case à cocher) dont les réponses sont essentielles pour obtenir un programme qui tourne à la fin.**

Introduction

Objectifs

Le but de ce projet est de vous faire découvrir et/ou travailler les notions citées plus haut, mais l'objectif concret de ce projet est la modélisation d'un distributeur de produits. Cela inclut la gestion du stock de produits, la gestion de la caisse, y compris le rendu de monnaie, l'interface avec l'utilisateur, c'est-à-dire des affichages, des saisies, mais aussi la traduction de la numérotation des produits en vitrine. Pour avoir un aperçu du programme auquel vous devez aboutir, vous pouvez exécuter le programme disponible sur cahier de prépa, nommé `exemple_distributeur`.

Pour rappel, il faut donner les droits d'exécution à ce fichier, par exemple avec la commande `chmod 755 exemple_distributeur`.

Votre programme pourra être plus simple en ce qui concerne l'affichage, ou au contraire plus évolué si vous voulez développer plus que ce qui est proposé ici, mais on devra pouvoir acheter un produit à partir de son numéro en vitrine et d'un lot de pièces de monnaie, la monnaie devra être rendue et le stock de ce produit mis à jour.

Architecture du code

Le programme final utilise un grand nombre de fonctions, qu'il faut toutes tester sur plusieurs entrées, entrées qu'il faut préalablement construire puisqu'il s'agit surtout de structures (et pas d'objets de type de base comme des entiers ou des booléens qu'on peut écrire directement dans l'appel). Si on codait tout ça dans un seul et même fichier, on aurait un fichier de plusieurs centaines de lignes, dans lequel il ne serait pas facile de se repérer.

On va donc au contraire découper le code dans plusieurs fichiers.

Séparation par nature de code

En général, on sépare ce qui relève de la déclaration (déclaration de structure, déclaration de fonction), de ce qui relève du code des fonctions ou encore du test des fonctions.

- *Fichiers de déclarations (aussi appelés fichiers d'interface)*

Les déclarations sont placées dans un fichier `xxx.h` (h comme *header i.e.* en-tête en anglais). On rappelle que la déclaration d'une fonction donne son type de sortie, son nom et les types de ses arguments. C'est comme la définition de fonction sans le corps entre accolades, avec **un point-virgule** à la place.

- *Fichiers de définition de fonctions (aussi appelés fichiers d'implémentation)*

Les définitions des fonctions sont placées dans un fichier `xxx.c` qui ne contient pas de fonction `main`. Il ne s'agit donc pas d'un programme, on ne peut pas l'exécuter. Par convention (à prendre comme une obligation), le fichier `xxx.c` contient la définition de toutes les fonctions déclarées dans le `xxx.h`. On inclut `xxx.h` dans `xxx.c` via `#include "xxx.h"` (avec des guillemets et non des chevrons comme pour la librairie standard), notamment afin que les structures déclarées dans `xxx.h` soient connues (par le compilateur) et puissent être utilisées. On peut aussi inclure si besoin un autre fichier `xyz.h`. On pourra alors utiliser dans `xxx.c` les fonctions et les types déclarés dans `xyz.h`, sans qu'il y ait besoin de définir les fonctions qui y sont déclarées, les définitions de ces fonctions se trouvant dans `xyz.c`.

- *Fichiers de tests*

Afin de tester les fonctions déclarées dans `xxx.h` et définies dans `xxx.c`, on crée un programme nommé `test_xxx.c`. Après les inclusions nécessaires (souvent `stdio.h`, `assert.h` et surtout `xxx.h`), ce fichier est uniquement constitué d'une fonction `main` dans laquelle on teste chacune des fonctions.

On rappelle que les fonctions avec valeur de retour doivent être testées dans des `assert`. De plus, comme on l'a vu en TP, le test d'égalité entre la valeur retournée et la valeur attendue peut nécessiter une fonction dans le cas où l'opérateur `==` se révèle insuffisant. On peut penser au cas des `float`, où des problèmes de précisions faussent le test avec `==`, ou aux chaînes de caractères pour lesquelles `==` teste seulement l'égalité des adresses.

À l'inverse les fonctions produisant un affichage sont seulement lancées. Cependant, afin d'aider à savoir si le test a réussi ou échoué, il est recommandé lorsque cela est possible, d'indiquer en commentaire le résultat attendu.

Séparation thématique du code

De plus on sépare aussi le code par thématique : on rassemble les fonctions qui ont le même genre d'effet ou qui s'appuient sur un même objet. Par exemple, `stdio.h` rassemble des fonctions pour gérer les entrées/sorties tandis que `stdbool.h` rassemble des fonctions opérant sur les booléens.

Dans ce projet, on rassemblera les fonctions qui agissent sur une même structure, que ce soit des fonctions d'initialisation de la structure, de mise-à-jour de la structure, d'extraction d'information ou d'affichage.

Organisation pour ce projet

Dans ce projet, vous aurez a minima 5 structures :

- produits
- numerotation
- disposition
- caisse
- distributeur

Afin de gérer les fonctions qui ne sont pas directement liées à l'une de ces structures, on considère deux autres "thèmes" :

- **coherence** pour les fonctions qui testent la cohérence entre des instances de deux structures. Par exemple, une fonction qui teste si une numérotation donnée est cohérente avec une disposition donnée, sera déclarée dans `coherence.h` car elle n'a pas plus de raisons d'être déclarée dans `numerotation.h` que dans `disposition.h`, ni le contraire.
- **auxiliaire** pour les fonctions qui n'ont a priori rien à voir avec ces structures. Par exemple si vous avez besoin – sait-on jamais ? – d'une fonction qui convertit un nombre d'heures en nombre de minutes, vous la déclarerez dans `auxiliaire.h`.

Sachant tout cela, vous ne serez pas étonné de trouver dans l'archive `projet_distributeur_vide` disponible sur cahier de prépa les fichiers suivants :

- 7 fichiers de déclarations nommés `produits.h`, `numerotation.h`, `disposition.h`, `caisse.h`, `distributeur.h`, `coherence.h` et `auxiliaire.h`.
- 7 fichiers de définition de fonctions nommés `produits.c`, `numerotation.c`, `disposition.c`, `caisse.c`, `distributeur.c`, `coherence.c` et `auxiliaire.c`.
- 7 fichiers de tests de fonctions nommés `test_produits.c`, `test_numerotation.c`, `test_disposition.c`, `test_caisse.c`, `test_distributeur.c`, `test_coherence.c` et `test_auxiliaire.c`.

En revanche vous serez peut-être étonné de trouver un fichier nommé **Makefile**. Ce fichier est là pour vous simplifier la compilation, grâce à lui vous disposez de commandes raccourcies pour compiler vos fichiers comme il se doit. Mais expliquons déjà comment compiler tous ces fichiers.

Compilation séparée

On a déjà évoqué en cours que la compilation se fait en deux étapes. Si vous êtes familier avec les commandes qui permettent de compiler en deux temps, avec des fichiers compilés intermédiaires (appelés fichiers objets), et familier de l'utilisation d'un **Makefile** vous pouvez passer cet exercice. Si vous êtes seulement à l'aise avec la compilation séparée, mais pas avec le **Makefile**, faites les questions 1 et 2 puis passez à la question 11. Si cette compilation en deux étapes ne vous dit rien du tout, suivez cet exercice pas à pas.

Les conclusions de cet exercice sont inscrites à sa suite, dans tous les cas vérifiez que vous maîtrisez tout ça avant de vous lancer dans le vif du sujet.

Exercice 0 Petit exemple de compilation séparée

Dans cet exercice on applique la discipline de séparation du code décrite ci-dessus sur un tout petit exemple. Ce n'est pas très pertinent sur un si petit code, mais le but est de vous faire comprendre comment compiler ces fichiers, et de voir quelles erreurs de compilation peuvent apparaître.

Avant de commencer, téléchargez sur cahier de prépa l'archive `exo0_vider.zip`. Après extraction, vous devez avoir quatre fichiers : `a.h`, `a.c`, `test_a.c` et `Makefile`.

Question 1

Dans le fichier `a.c`, donnez la définition de la fonction nommée `affiche_a` qui prend en entrée un entier positif `n` et affiche `n` fois le caractère `'a'`. Si des instructions se trouvent déjà dans le corps de cette fonction, comme `printf("Définissez la fonction affiche_a\n");` ou `assert(false);`, supprimez-les quand vous définissez la fonction. Dé-commentez la déclaration de cette fonction dans le fichier `a.h`. Attendez pour compiler.

Question 2

Complétez la fonction `main` du fichier `test_a.c` pour qu'elle teste la fonction `affiche_a`. À nouveau supprimez les instructions devenues inutiles qui se trouvaient dans le corps de cette fonction. Attendez pour compiler.

Question 3

Essayez de compiler le fichier `a.c` comme on compilait jusqu'ici, c'est-à-dire par `gcc a.c -o a`.

Vous obtenez une erreur du compilateur `undefined reference to `main``. Elle signifie que ce fichier ne peut être transformé en code exécutable puisqu'il n'y a pas de fonction `main`. En effet, ce code ne dit pas ce qui devrait se passer à l'exécution. Les fichiers de définition de fonction doivent être compilés différemment.

Question 4

Tapez `gcc -c a.c -o a.o`.

Si votre code est correct, rien ne s'affiche mais vous avez un fichier `a.o` dans votre dossier.

La commande `gcc -c` a généré un fichier appelé fichier objet, ici ici `a.o`, qui contient le code compilé des définitions de fonctions, mais la compilation s'est arrêtée avant de produire un fichier exécutable. En particulier cette compilation admet que les fonctions déclarées sont définies quelque part, et attend la prochaine étape de compilation pour accéder à leurs définitions.

Question 5

Pour constater ce dernier point, dé-commentez dans le fichier `a.h` la déclaration de la fonction `affiche_point`. Ajoutez un appel à cette fonction dans votre définition de `affiche_a` dans `a.c`. Compilez à nouveau avec `gcc -c a.c -o a.o`. Normalement aucune erreur n'apparaît.

Question 6

Commentez la commande `#include "a.h"` dans le fichier `a.c` et compilez à nouveau ce fichier avec `gcc -c a.c -o a.o`

Cette fois une erreur (enfin un warning) apparaît, et indique notamment :
`implicit declaration of function 'affiche_point'`.

Cela signifie que la déclaration de la fonction `affiche_point` est manquante.

Question 7

Au choix, supprimez l'appel à `affiche_point` et supprimez sa déclaration ou bien donnez une définition de `affiche_point` dans `a.c`. Dans les deux cas, dé-commentez la commande `#include "a.h"` dans le fichier `a.c`. Compilez à nouveau ce fichier avec `gcc -c a.c -o a.o`. vous ne devez plus avoir d'erreur ni de warning, et un fichier objet nommé `a.o` a du être créé.

Question 8

Puisque le fichier `test_a.c` code bien un programme (avec une fonction `main`), compilez-le avec la commande utilisée jusqu'ici : `gcc test_a.c -o test_a`.

Si vous obtenez l'erreur `implicit declaration of function 'affiche_a'`, c'est que vous avez oublié d'inclure `a.h`. Ajoutez `#include "a.h"` et recompilez.

Là vous obtenez une erreur `undefined reference to 'affiche_a'`. C'est normal, la définition de cette fonction se trouve dans un autre fichier, `a.c`. Pour résoudre le problème, on pourrait ajouter `#include "a.c"` dans `test_a.c`. Cela aurait pour effet de copier le code du fichier `a.c` au début du fichier `test_a.c` avant de le compiler. Ainsi la définition de `affiche_a` serait présente, et la compilation produirait bien un exécutable `test_a`.

Cependant **on ne fera jamais ça**, car cette solution fait recompiler les fonctions définies dans `a.c` à chaque fois qu'on compile un fichier avec `#include "a.c"`. À la place on va utiliser le code déjà compilé des définitions de `a.c`, c'est-à-dire le fichier objet `a.o`.

Question 9

Tapez `gcc a.o test_a.c -o test_a`.

Vous obtenez un exécutable nommé `test_a`, lancez-le pour vérifier que tout fonctionne normalement.

Question 10

Modifiez la définition de la fonction `affiche_a` pour qu'elle affiche le même nombre de 'a', mais séparés par des espaces. Refaites les étapes de compilation nécessaires pour tester à nouveau votre code.

Vous avez du faire `gcc -c a.c -o a.o` puis `gcc a.o test_a.c -o test_a`. À chaque modification d'un fichier d'implémentation, il faut a minima recompiler le fichier objet correspondant, et en général recompiler le programme pour tester cette fonction. De plus, lorsqu'on aura un code plus important, les programmes de test pourront nécessiter de rassembler plusieurs fichiers objets (*i.e.* des `.o`), et l'oubli de l'un d'eux entraînera une erreur de type `undefined reference to`. C'est donc assez fastidieux. C'est pourquoi l'utilisation d'un `Makefile` peut se révéler profitable.

Un `Makefile` permet de définir des commandes qui lancent des commandes de compilation, qu'on peut voir comme des raccourcis. Par exemple,

```
a.o : a.c
    gcc -c a.c -o a.o
```

définit le raccourci `make a.o` qui lancera la commande `gcc -c a.c -o a.o`.

Question 11

Supprimez vos fichiers compilés : `a.o` et `test_a`. Tapez `make a.o` dans le terminal *Vous pouvez utiliser l'autocomplétion grâce à la touche tabulation.*

Vous voyez apparaître `gcc -c a.c -o a.o`, c'est la commande qui a été lancée. Vous pouvez aussi constater que le fichier `a.o` a bien été compilé.

Question 12

Supprimez le fichier que vous venez de compiler : `a.o`. Tapez `make test_a` dans le terminal.

Vous voyez apparaître `gcc -c a.c -o a.o`, puis `gcc test_a.c a.o -o test_a`. Ce sont les deux commandes qui ont été lancées. Vous pouvez aussi constater que les fichiers `a.o` et `test_a` ont bien été compilés. En regardant dans le fichier `Makefile` comment est définie le raccourci, vous trouvez la définition suivante :

```
test_a : a.o test_a.c
    gcc test_a.c a.o -o test_a
```

Cela peut vous surprendre, car le raccourci `make test_a` ne semble pas appeler `gcc -c a.c -o a.o`. Mais on a omis de considérer l'information placée après le `:`, elle indique quels fichiers doivent être compilés et à jour pour lancer la commande indiquée à la ligne suivante. Ici, on a précisé qu'il faut que le fichier `a.o` soit compilé et à jour pour lancer `gcc test_a.c a.o -o test_a`, on parle de dépendance entre `test_a` et `a.o`. Comme le fichier `a.o` n'existait pas, `make` a cherché à le compiler, et pour ça il a suivi ce qu'on avait donné plus haut comme instructions (en définissant le raccourci `make a.o`), à savoir `gcc -c a.c -o a.o`.

Question 13

Supprimez le fichier `test_a` seulement, et relancez `make test_a`.

Cette fois le fichier `a.o` était déjà présent et à jour, le raccourci `make test_a` a donc seulement déclenché la compilation `gcc test_a.c a.o -o test_a`.

On peut se demander à quoi sert de préciser `a.c` et `a.h` comme dépendances de `a.o`. Cela permet de préciser que le fichier `a.o` doit être recompilé si le fichier `a.c` ou le fichier `a.h` a été modifié depuis la dernière compilation de `a.o`.

Question 14

Supprimez à nouveau le fichier `test_a`, puis modifiez le fichier `a.c` et enregistrez ces modifications. Relancez `make test_a`. *Vous pouvez aussi faire la même chose en modifiant `a.h`.*

Cette fois le fichier `a.o` était déjà présent, mais il n'était pas à jour, `make` a alors cherché à le recompiler, et pour ça il a suivi ce que l'on a donné plus haut, c'est pourquoi la commande de compilation `gcc -c a.c -o a.o` a été effectuée avant `gcc test_a.c a.o -o test_a`.

Si les dépendances entre fichiers compilés sont mal définies dans le `Makefile`, il peut arriver qu'on modifie un fichier mais qu'on n'observe pas les modifications à l'exécution du programme, car il manque une re-compilation quelque part. Si ce genre de chose arrive, ou justement pour l'éviter, on peut supprimer tous les fichiers compilés, et tout recompiler à chaque fois. Pour cela, on définit un raccourci `make clean` dans le `Makefile`, qui supprime tous les fichiers d'extension `.o` (grâce à la commande `rm -f *.o`) et qui supprime un par un les fichiers exécutables (ici seulement `test_a` grâce à `rm -f test_a`).

Question 15

Supprimez tous les fichiers compilés grâce au raccourci `make clean`. Constatez que ça a marché, relancez des compilations... bref familiarisez vous avec les commandes `make`.

À retenir

- La compilation se fait en deux étapes.
- La première nécessite que tout soit bien déclaré, et si ce n'est pas le cas on obtient des erreurs de type `implicit declaration of`
- La deuxième nécessite que les fonctions appelées soient bien définies, et doit avoir accès à ces définitions, si ce n'est pas le cas on obtient des erreurs de type `undefined reference to`
- La première étape produit un fichier compilé intermédiaire, qu'on nomme avec l'extension `.o` par convention.
- La deuxième étape produit un fichier exécutable, qu'on nomme sans extension par convention. Pour cela, le point d'entrée est la fonction `main`. Le code qu'on donne à compiler doit de ce fait contenir une et une seule fonction `main`.
- le `Makefile` permet de définir des raccourcis qui lanceront les commandes de compilation dont on a besoin.

En pratique (sans Makefile)

- on déclare les fonctions dans un fichier `xxx.h`
- on définit toutes les fonctions déclarées dans `xxx.h` dans un fichier `xxx.c` qui inclut `xxx.h` grâce à `#include "xxx.h"`
- on compile ces définitions via la commande `gcc -c xxx.c -o xxx.o`.
- on teste toutes ces fonctions dans la fonction `main` d'un fichier `test_xxx.c` qui inclut lui aussi `xxx.h` grâce à `#include "xxx.h"`
- on compile ce programme de test avec les définitions de fonctions déjà compilées via la commande `gcc xxx.o test_xxx.c -o xxx`
- on lance le programme de test via `./xxx`

Au boulot !

Exercice 1 Structure produits

Afin de gérer un ensemble fini de produits, on utilise la structure `produits` définie comme suit (dans le fichier `produits.h`).

```
5 struct produits {
6     int nb_p;
7     //le nombre de produits
8     int* t_stock;
9     //pointeur vers un tableau de nb_p entiers indiquant le stock de chaque
    ↪ produit
10    int* t_prix;
11    //pointeurs vers un tableau de nb_p entiers indiquant le prix de chaque
    ↪ produit
12};
```

Le champ `nb_p` donne le nombre de produits différents, au sens où l'on appelle produit un type de produit (par exemple on parle de bouteille d'eau comme d'*un* produit même si on a *quatre* bouteilles d'eau identiques). Implicitement on met alors en bijection l'ensemble des produits à modéliser et $[0..nb_p - 1[$. Ainsi les cases d'indice $i \in [0..nb_p - 1[$ des tableaux `t_stock` et `t_prix` donnent des informations à propos du même produit, celui d'indice i . Plus précisément `t_stock[i]` donne combien d'exemplaire de ce produit sont disponible. Ce peut être 0, mais ce ne peut être une valeur négative. `t_prix[i]` indique quant à lui le prix en centimes de ce produit (idem cette valeur ne peut être strictement négative).

On a choisi dans ce projet de traiter toutes les sommes d'argent comme des nombres de centimes afin d'éviter tous les problèmes d'arrondi liés au codage des `float`.

Question 1

Dans le fichier `auxiliaire.c` codez la fonction `affiche_montant` qui affiche une somme en euros à partir d'un nombre de centimes. Par exemple,

- `affiche_montant(50)` devra afficher 0€50,
- `affiche_montant(500)` devra afficher 5€00
- `affiche_montant(5)` devra afficher 0€05

Supprimez les instructions devenues inutiles dans le corps de cette fonction (comme `assert(false)`) et dé-commentez la déclaration de cette fonction dans `auxiliaire.h`, (et ne la modifiez pas).

Compilez cette définition (avec `gcc -c auxiliaire.c -o auxiliaire.o` ou avec `make auxiliaire.o`). Testez ensuite cette fonction dans `test_auxiliaire.c` (à compiler avec `make test_auxiliaire` ou `gcc auxiliaire.o test_auxiliaire.c -o test_auxiliaire` puis à exécuter).

Question 2

Dans le fichier `produits.c` codez la fonction `nb_produits` qui affiche le nombre de *produits* en stock dans `prd` de type `struct produits`. Attention, dans cette phrase "produit" désigne en fait une occurrence d'un produit. Par exemple, on renverra 6 si on a modélisé un ensemble de 4 bouteilles d'eau identiques et de 2 paquets de chips identiques (auquel cas `nb_p` vaut 2 car on a deux types de produits). Comme précédemment, vous supprimerez les instructions devenues inutiles, et dé-commenterez la déclaration de cette fonction dans `produits.h`.

Compilez cette définition (avec `gcc -c produits.c -o produits.o` ou avec `make produits.o`).

Testez ensuite cette fonction dans le fichier `test_produits.c` (à compiler avec `make test_produits`

ou `gcc produits.o test_produits.c -o test_produits` puis à exécuter).

Question 3

Dans le fichier `produits.c` codez la fonction `valeur_globale` qui affiche la valeur globale du stock dans `prd` de type `struct produits`, c'est à dire la somme cumulée de tous les produits en stock. À nouveau compilez et testez cette fonction.

Question 4

Dans le fichier `produits.c` codez la fonction `affiche_info_globale` qui affiche le nombre de produit et la valeur globale du stock dans `prd` de type `struct produits`. On attend un affichage du genre suivant.

Il y a 21 produits totalisant 38€25.

Question 5

Dans le fichier `produits.c` codez la fonction `affiche_info_detailed` qui affiche un tableau résumant le prix et le stock de chaque type de produit dans `prd` de type `struct produits`. On attend un affichage du genre suivant.

```
Liste des produits
=====
prix | quantité
-----
0€55 | 0
1€20 | 1
2€00 | 2
0€85 | 3
1€00 | 0
1€00 | 0
1€50 | 5
2€00 | 3
3€00 | 5
1€00 | 2
-----
```

Dans la suite on ne précise plus qu'il faut, pour chaque fonction, supprimer les instructions devenues inutiles, dé-commenter la déclaration dans le fichier d'interface, compiler, puis créer un programme de test, le compiler et l'exécuter pour vérifier. Cependant, comme pour l'exercice précédent, les déclarations de fonction sont déjà données, en commentaire, dans les fichiers `.h`, et les commandes de compilation utiles font l'objet d'un raccourci `make` défini dans le `Makefile`.

Exercice 2 Structure numerotation

On suppose que les produits du distributeurs sont numérotés en vitrine. Chaque numéro a au plus 2 chiffres (en base 10, comme usuellement), et n'est attribué qu'à un produit au plus (*i.e.* à 1 ou 0). En revanche les numéros possibles (*i.e.* les nombres entre 0 et 99), ne sont pas tous attribués à un produit. Afin de gérer l'ensemble des numéros attribués à l'un des produit, c'est-à-dire des numéros présents en vitrine, on utilise un tableau de booléens. On associe ce tableau à sa longueur dans la structure `numerotation` définie comme suit (dans le fichier `numerotation.h`).

```
5 | struct numerotation{
6 |     int nb_num;      //le nombre de numéros possibles
7 |     bool* t_num;     //indique pour chaque numéro s'il est présent
8 | };
```

Pour i entre 0 et `nb_num-1`, `t_num[i]` vaut `true` si le numéro est présent, et `false` sinon.

Question 1

Dans le fichier `numerotation.c` codez la fonction `affiche_numrt` qui affiche les numéros présents dans la vitrine selon `numrt` de type `struct produits`. On attend un affichage du genre suivant.

```
numéros présents : 11 12 13 21
```

Question 2 *

Dans le fichier `numerotation.c` codez la fonction `affiche_numrt_diz` qui affiche les numéros présents dans la vitrine selon `numrt` de type `struct produits`, en allant à la ligne dès qu'on change de dizaine, et en évitant les lignes vides pour les dizaines non représentées. On attend un affichage du genre suivant.

```
numéros présents :
11 12 13
21 23 25
30 32
```

Question 3

Comme vous avez pu le constater en testant vos fonctions, créer à la main une numérotation est un peu fastidieux. Codez donc la fonction `create_numrt_from_tab` dans le fichier `numerotation.c`. Cette fonction prend en argument un pointeur vers un tableau d'entiers `tab`, la longueur de ce tableau `lg_tab` et un entier `nb_num > 1`, et retourne une numérotation de taille `nb_num` dans laquelle les numéros présents sont exactement les entiers figurant dans `tab`. On suppose donc que les entiers de `tab` sont tous compris entre 0 et `nb_num-1`. *Pensez à réserver un espace mémoire persistant pour stocker le tableau de booléens constituant le deuxième champ de la structure retournée.*

Question 4

Dans le fichier `numerotation.c` codez la fonction `ind_from_num` qui donne, pour un numéro `num` et une numérotation `numrt`, l'indice du produit ayant le numéro `num` si celui-ci est présent en vitrine, et -1 sinon. Par exemple, en notant `numrt2` la numérotation affichée à la question 2,

- `ind_from_num(11, numrt2)` doit retourner 0, car c'est le plus petit numéro présent
- `ind_from_num(12, numrt2)` doit retourner 1, car c'est le deuxième plus petit numéro présent
- `ind_from_num(13, numrt2)` doit retourner 2, car c'est le troisième plus petit numéro présent
- `ind_from_num(14, numrt2)` doit retourner -1, car ce numéro n'est pas présent

Question 5

Dans le fichier `numerotation.c` codez la fonction `prochain_num` qui donne, pour un numéro `num_0` et une numérotation `numrt`, le plus petit numéro supérieur ou égal à `num_0` présent selon `numrt`. Par exemple, en notant `numrt2` la numérotation affichée à la question 2,

- `prochain_num(0, numrt2)` doit retourner 11 qui est le plus petit numéro présent supérieur ou égal à 0
- `prochain_num(1, numrt2)` doit retourner 11 qui est le plus petit numéro présent supérieur ou égal à 1
- `prochain_num(11, numrt2)` doit retourner 11 qui est le plus petit numéro présent supérieur ou égal à 11
- `prochain_num(11+1, numrt2)` doit retourner 12, car c'est le plus petit numéro présent supérieur ou égal à 12, *i.e.* strictement supérieur à 11
- `prochain_num(13+1, numrt2)` doit retourner 21, car c'est le plus petit numéro présent supérieur ou égal à 14, *i.e.* strictement supérieur à 13
- `prochain_num(32+1, numrt2)` doit retourner -1, car il n'y a aucun numéro présent supérieur ou égal à 33, *i.e.* strictement supérieur à 32

Question 6

Dans le fichier `numerotation.c` codez la fonction `num_from_ind` qui donne, pour un indice positif `ind` et une numérotation `numrt`, le numéro du $(ind+1)$ ème du produit, c'est-à-dire le $(ind+1)$ ème plus petit numéro présent selon `numrt`, et -1 si `ind` est trop grand. Par exemple, en notant `numrt2` la numérotation affichée à la question 2,

- `num_from_ind(0, numrt2)` doit retourner 11, car c'est le plus petit numéro présent
- `ind_from_num(1, numrt2)` doit retourner 12, car c'est le deuxième plus petit numéro présent
- `ind_from_num(2, numrt2)` doit retourner 13, car c'est le troisième plus petit numéro présent
- `ind_from_num(7, numrt2)` doit retourner 32, car c'est le huitième plus petit numéro présent
- `ind_from_num(8, numrt2)` doit retourner -1, car il n'y pas de neuvième plus petit numéro présent puisqu'il n'y a que 8 numéros présents.

La fonction `prochain_num` peut s'avérer utile pour coder cette fonction.

Question 7

Dans le fichier `coherence.c` codez la fonction `coherence_numrt_prdts` qui teste si une numérotation `numrt` et un stock de produits `prd` sont cohérents, c'est-à-dire s'il y a autant de numéros présents dans `numrt` que de produits dans `prd`. *Pour compiler le fichier `test_coherence.c` dans lequel vous testez la la fonction `coherence_numrt_prdts` utilisez `make test_coherence_1` à ce stade.*

Exercice 3 Structure disposition

Afin de produire un affichage un peu réaliste des produits du distributeur, on s'intéresse à leur disposition. On suppose que les produits sont organisés sur plusieurs rangées (horizontales), et que le nombre de produits (de type de produits) pouvant apparaître sur chaque rangée est fixé, on parle alors de case. On considère la structure `disposition` pour indiquer de combien de rangées dispose le distributeur, et combien chacune d'elles offre de case.

```
10 struct disposition{
11     int nb_rg;
12     //le nombre de rangées
13     int* t_rg;
14     //le nombre de produits possibles sur chaque rangée
15 };
```

On ne se préoccupe pas ici de la profondeur de chaque case, en fait on suppose que cette profondeur est infinie, c'est-à-dire que peu importe le stock d'un produit, on le placera dans une et une seule case du distributeur. En particulier, les produits qui ne sont plus en stock occupent aussi une place dans la mesure où l'on remplira cette case avec ce produit lors du réapprovisionnement.

Question 1

Dans le fichier `disposition.c` codez la fonction `affiche_dispo_simple` qui à partir d'une disposition `dispo` affiche pour chaque rangée prévue par `dispo` le nombre de cases prévues par `dispo`. On attend un affichage du genre suivant (notez qu'on numérote 1 la première rangée).

```
-----
rangée 1 : 3 produits
rangée 2 : 3 produits
rangée 3 : 2 produits
```

Les numéros présents selon la numérotation du distributeur seront attribués à chaque case dans l'ordre croissant, de gauche à droite pour chaque rangée, depuis la rangée du haut à la rangée du bas.

Question 2

Dans le fichier `disposition.c` codez la fonction `affiche_dispo_simple_num` qui à partir d'une disposition `dispo` et d'une numérotation `numrt` affiche pour chaque rangée prévue par `dispo` les numéros des cases sur cette rangée selon `numrt`. On attend un affichage du genre suivant (notez qu'on numérote toujours 1 la première rangée).

```
-----
rangée 1 : 11 12 13
rangée 2 : 21 23 25
rangée 3 : 30 32
```

Pour être efficace votre fonction devrait éviter l'appel à la fonction `num_from_ind`, mais plutôt utiliser la fonction `prochain_num`.

On dira qu'une numérotation `numrt` et une disposition `dispo` sont cohérentes, s'il y a autant de numéros présents dans `numrt` que de cases dans `dispo`, et si de plus les numéros présents sur une rangée i ont i comme chiffre des dizaines. Là encore on suppose que les ranges sont numérotées à partir de 1, et de haut en bas. Si la première condition est satisfaite, alors la deuxième se vérifie

en un coup d'œil sur l'affichage produit par `affiche_dispo_simple_num`. Néanmoins on voudrait pouvoir tester cette cohérence de manière automatique.

Question 3

Dans le fichier `coherence.c` codez la fonction `coherence_numrt_dispo` qui teste si une numérotation `numrt` et une disposition `dispo` sont cohérentes. *Pour compiler le fichier `test_coherence.c` dans lequel vous ajoutez des tests pour la fonction `coherence_numrt_dispo` utilisez maintenant la commande `make test_coherence`.*

La suite de cet exercice est dédiée à la réalisation d'un affichage graphique (et pas textuel) qui s'adapte à la disposition. il s'agit uniquement de fonctions bonus qu'il est conseillé de passer dans un premier temps, pour y revenir quand vous aurez un distributeur fonctionnel.

On cherche à produire un affichage de même largeur pour chaque rangée, y compris si celles-ci ont des nombres des cases différents (on suppose qu'aucune rangée de la disposition n'a 0 case).

Les cases d'une même rangée doivent se partager la largeur de l'affichage en part égales. On exprime cette largeur en nombre de case élémentaires, `nb_elem` dans le code, où une case élémentaire est de taille 2 caractères. Ainsi sur une largeur de 2 cases élémentaires, on peut représenter une case par `[--]`. On évitera de représenter une case sur une seule case élémentaire, car on aurait un affichage comme `[]` qui ne permet pas d'afficher le numéro.

Pour la disposition que la fonction `affiche_dispo_simple` affiche comme à droite ci-dessous, on souhaite obtenir un affichage comme à gauche.

<pre> ----- rangée 1 : 3 produits rangée 2 : 3 produits rangée 3 : 2 produits rangée 4 : 6 produits rangée 5 : 4 produits </pre>	<pre> ----- [-----] [-----] [-----] [-----] [-----] [-----] [-----] [-----] [--] [--] [--] [--] [--] [--] [----] [----] [----] [----] ----- </pre>
--	--

Le nombre de cases élémentaires utilisé ici était 12.
 Sur la rangée 1, il y a 3 cases, chacune utilise 4 cases élém. soit 8 caractères : 2 crochets et 6 tirets.
 Sur la rangée 2, il y a 3 cases, chacune utilise 4 cases élém. soit 8 caractères : 2 crochets et 6 tirets.
 Sur la rangée 3, il y a 2 cases, chacune utilise 6 cases élém. soit 12 caractères : 2 crochets et 10 tirets.
 Sur la rangée 4, il y a 6 cases, chacune utilise 2 cases élém. soit 4 caractères, : 2 crochets et 2 tirets.
 Sur la rangée 5, il y a 4 cases, chacune utilise 3 cases élém. soit 6 caractères : 2 crochet et 4 tirets.

Pour une disposition de 5 rangées, ayant respectivement 3, 3, 2, 4 et 5 cases, on voudrait obtenir l'affichage ci-dessous (imprimé en police réduite afin de tenir sur une largeur de page).

```

-----
[-----] [-----] [-----]
[-----] [-----] [-----]
[-----] [-----]
[-----] [-----] [-----] [-----]
[-----] [-----] [-----] [-----]
-----

```

La plus grande difficulté est donc de calculer ce nombre `nb_elem`, qui permet d'occuper toute la largeur avec des cases de tailles identiques sur chaque rangée, et qu'aucune des cases soit réduite à une case élémentaire.

Question 4 **

Codez dans le fichier `dispo.c` la fonction `nbc_elem_dispo` qui calcule le nombre de cases élémentaires à utiliser en largeur pour obtenir un affichage de la disposition prise en argument similaire à ce qui est décrit plus haut.

Question 5 *

Codez dans le fichier `dispo.c` la fonction `affiche_trait` qui affiche un trait de longueur un nombre de cases élémentaires passé en argument (soit le double de tirets du bas, non espacés). Cette fonction servira à tracer un trait au dessus et un trait au dessous de l’affichage des cases par rangées, comme c’est fait dans les exemple ci-dessus.

Question 6 *

Codez dans le fichier `dispo.c` la fonction `affiche_case_vider` qui affiche une case vide de largeur un nombre de cases élémentaires $n > 1$ passé en argument.

Question 7 *

Codez dans le fichier `dispo.c` la fonction `affiche_ligne_vider` qui prend en argument un nombre `nbc_elem` de cases élémentaires et un nombre `nb_cases` de cases (cases de la rangée), et qui affiche une ligne constituée de `nb_cases` cases vides de même largeur de sorte que la largeur totale de la ligne soit de `nbc_elem` cases élémentaires. *On vous laisse préciser les hypothèses pour que ce soit possible.*

Question 8 *

Codez dans le fichier `dispo.c` la fonction `affiche_dispo_vider` qui prend en argument une disposition et qui affiche pour chacune de ses rangées des cases vides de même largeur, et ce de manière à ce que toutes les lignes aient la même largeur. Autrement dit la fonction qui réalise l’affichage qu’on vise, sans numéro.

Question 9 **

Codez dans le fichier `dispo.c` la fonction `affiche_case_num` qui prend en argument un nombre de cases élémentaires $n > 1$ et un numéro `num` entre 0 et 99, et qui affiche une case de largeur `n` cases élémentaire en tout, avec au milieu une écriture à deux chiffres de `num`.

Question 10 **

Codez dans le fichier `dispo.c` la fonction `affiche_ligne_num` qui prend en argument un nombre `nbc_elem` de cases élémentaires et un nombre `nb_cases` de cases (cases de la rangée), ainsi qu’une numérotation `numrt` et un premier numéro `num_0` présent selon `numrt`, **qui affiche** une ligne constituée de `nb_cases` cases avec numéro de même largeur de sorte que la largeur totale de la ligne soit de `nbc_elem` cases élémentaires, et de sorte que les numéros affichés soient des numéros successifs selon `numrt`, avec `num_0` comme premier numéro, **et qui retourne** le numéro suivant le dernier affiché dans `numrt`. *On vous laisse préciser les hypothèses pour que ce soit possible.*

Question 11 **

Codez dans le fichier `dispo.c` la fonction `affiche_dispo_num` qui prend en argument une disposition et une numérotation, et qui affiche pour chacune des rangées des cases de même largeur, avec le numéro donné par la numérotation, et ce de manière à ce que toutes les lignes aient la même largeur. Autrement dit la fonction qui réalise l’affichage qu’on vise, avec les numéros cette fois

Exercice 4 Structure caisse

On suppose fixé le système monétaire, et même fixées les capacités mécaniques du distributeur : il ne peut gérer que des pièces de 2€, 1€, 50ct, 20ct, 10ct et 5ct. C'est pourquoi on déclare constants le nombre de pièces `nb_pc` et le tableau des valeurs en centimes de ces `nb_pc` pièces dans le fichier `caisse.c` :

```
7 | const int nb_pc = 6;  
8 | const int val_pc[6] = {200, 100, 50, 20, 10, 5};
```

Ceci étant fixé, pour décrire une caisse, ou plus largement un multi-ensemble de pièces, il suffit d'indiquer pour chaque `i` entre 1 et `nb_pc-1` combien de pièces de valeur `val_pc[i]` il y a. C'est pourquoi on définit la structure suivante dans `caisse.h`

```
10 | struct caisse{  
11 |     int* t_pc;  
12 |     //le nombre de pièces disponibles pour chaque pièces  
13 | };
```

Une caisse bien initialisée est donc un pointeur vers un tableau de 6 entiers puisque `nb_pc = 6`.

Question 1

Codez dans le fichier `caisse.c` la fonction `solde` qui renvoie la somme des valeurs en centimes de toutes les pièces contenues dans la caisse qu'elle prend en argument.

Question 2

Codez dans le fichier `caisse.c` la fonction `affiche_caisse` qui affiche pour chaque valeur de pièce possible, le nombre de pièces de cette valeur contenues dans la caisse qu'elle prend en argument. On attend un affichage du type suivant.

```
pièces de 2€00 : 10  
pièces de 1€00 : 5  
pièces de 0€50 : 2  
pièces de 0€20 : 0  
pièces de 0€10 : 2  
pièces de 0€05 : 6
```

Question 3

Codez dans le fichier `caisse.c` la fonction `ajoute_caisse` qui prend en argument deux caisses `c1` et `c2`, et qui modifie les valeurs de `c1` en ajoutant, pour chaque valeur de pièce, le nombre de pièces de cette valeur dans `c2`. Autrement dit cette fonction ajoute les pièces de `c2` à celles de `c1`.

Question 4

Codez dans le fichier `caisse.c` la fonction `prend_monnaie` qui crée une caisse initialement vide, qui la remplit de pièces selon ce que saisit l'utilisateur, puis retourne cette caisse. Pour l'interface avec l'utilisateur, on attend des affichages du genre suivant :

"Combien de pièces de 2€00 voulez-vous insérer dans la machine ?".

Pensez que la mémoire réservée pour la caisse que vous créez doit être persistante.

Question 5

Codez dans le fichier `caisse.c` la fonction `caisse_suffit` qui prend en argument une caisse `c` et un montant `m` en centimes et qui renvoie s'il est possible de constituer la somme `m` avec les pièces disponibles dans `c`.

Question 6

Codez dans le fichier `caisse.c` la fonction `rend_monnaie` qui prend en argument une caisse `c` et un montant `m` en centimes qu'il est possible de former avec les pièces de `c`, et qui affiche comment. Plus précisément, à chaque pièce utilisée pour former le montant `m` vous afficherez cette pièce, ce qui mime un peu un rendu de monnaie où les pièces tomberaient une à une. On attend donc un affichage du genre suivant.

```
une pièce de 2€00
une pièce de 2€00
une pièce de 1€00
une pièce de 0€10
une pièce de 0€10
une pièce de 0€05
```

Exercice 5 Structure distributeur

Un distributeur est constitué d'un stock de produits à vendre et d'une caisse pour gérer les paiements en pièces. De plus, les produits sont présentés en vitrine selon une organisation pouvant être modélisée par un objet de type `disposition`, et avec des numéros pouvant être modélisés par un objet de type `numerotation`. C'est pourquoi on définit finalement la structure `distributeur` comme suit dans le fichier `distributeur.h`.

```
15 | struct distributeur{
16 |     struct produits prd;
17 |     struct numerotation numrt;
18 |     struct disposition dispo;
19 |     struct caisse c;
20 | };
```

Question 1

Codez dans le fichier `distributeur.c` la fonction `selection_produit` qui prend en argument un distributeur, qui affiche les numéros présents en vitrine (rangée par rangée selon la disposition si vous l'avez codé), puis demande à l'utilisateur de saisir un numéro, jusqu'à ce qu'un numéro de produit disponible soit saisi, auquel cas on retourne l'indice du produit ayant ce numéro en vitrine.

Si le numéro saisi n'est pas présent en vitrine, avant de relancer la saisie on affiche le message suivant.
Numéro non valide. Réessayez.

Si le numéro saisi correspond à un produit qui n'est plus en stock, avant de relancer la saisie on affiche le message suivant.

Ce produit n'est plus en stock, désolé.

Si vous souhaitez autre chose, tapez le numéro correspondant.

Vous pouvez relancer le programme `exemple_distributeur` pour un exemple d'interface attendue.

Question 2

Codez dans le fichier `distributeur.c` la fonction `transaction` qui demande à l'utilisateur de choisir un produit en stock, puis lui demande de rentrer des pièces dans la machine, qui réalise la transaction si c'est possible, ou l'interrompt sinon, qui demande à l'utilisateur s'il souhaite effectuer une nouvelle transaction et qui retourne si c'est le cas ou non.

Pour que la transaction puisse avoir lieu il faut non seulement que l'utilisateur ait rentré assez de pièces pour payer le produit, mais aussi que la caisse du distributeur permette de lui rendre la monnaie. *Si l'utilisateur met des pièces inutiles dans le distributeur, on se réserve le droit de refuser la transaction même si après avoir encaissé ses pièces on aurait pu lui rendre la monnaie. Un cas extrême de cette situation est le cas d'une caisse vide, et d'un utilisateur qui met deux pièces de 2€ pour payer un produit à 1€ ... on pourra lui dire qu'on ne peut pas lui rendre la monnaie même si ce n'est pas tout à fait exact, l'idée étant qu'il peut réessayer en mettant seulement une pièce de 1€.* Si la transaction a lieu, mettez à jour le stock du produit, et rendez la monnaie avant de demander à l'utilisateur s'il veut commencer une nouvelle transaction. À l'inverse si la transaction n'a pas lieu, rendez à l'utilisateur exactement les pièces qu'il a introduites, une à une, avant de lui demander s'il veut commencer une nouvelle transaction.

Exercice 6 Idées bonus

On liste ci-dessous des idées en vrac d'améliorations possibles de ce projet. Avant de vous lancer dans la mise en œuvre de l'une de ces pistes, sauvegardez une version qui fonctionne de votre code. Avec un `make clean` puis une compression de tout le dossier, et le tour est joué.

- * ajouter un nom à chaque produit, et l'afficher après sélection du numéro de produit, pour demander confirmation
- * ajouter un affichage du stock disponible pour chaque produit
- * ajouter la possibilité d'annuler à tout moment de la transaction, en tapant -2 par exemple
- * améliorer le fonction `prend_monnaie` pour la rendre plus *user friendly*, c'est-à-dire plus pratique pour l'utilisateur
- * améliorer la procédure de transaction pour ne plus rejeter de transaction qui serait réalisable si on encaissait les pièces avant de décider si on peut rendre la monnaie.
- * * créer une fonction qui après chaque transition vérifie l'état du distributeur et produit un message d'alerte si des produits ne sont plus en stock ou s'il manque un type de pièce et que les pièces plus petites disponibles ne permettent pas de former la valeur de la pièce manquante (par exemple, on n'affiche pas d'alerte s'il n'y a plus de pièces de 50ct mais qu'il reste deux pièces de 20ct et un pièce de 10ct au moins).
- * * améliorer la structure de caisse pour tenir compte des capacités de stockages limitées d'une caisse réelle. On pourra alors déclencher une alerte aussi quand la caisse est pleine.
 - * * prendre en compte la profondeur du distributeur, et donc le fait qu'un même produit peut être réparti dans plusieurs cases si son stock est trop important pour la profondeur d'une case. On devra alors tester la cohérence d'un stock avec les profondeurs. De plus il faudra éviter des alertes inutiles, ne pas indiquer qu'il manque un produit s'il est encore disponible dans une autre case.