ENSEIRB 2006-2007 - Première année
PG101 - Programmation en C

TD 6 : Compilation séparée (commentaires)

1 Le processus de compilation

Exercice 1: Pré-compilation

Lors de l'exécution de la commande `gcc -E prog1.c -o prog1.i`, un programme ressemblant beaucoup à `prog1.c` est écrit dans le fichier `prog1.i`. Voici les différences que nous pouvons noter :

- les directives `#include <stdio.h>` et `#define N 10` ont disparu
- une suite de déclarations sont apparues en tête de la fonction principale `main`
- les occurrences de `N` dans la fonction `main` ont toutes été remplacées par la valeur `10`

La page `man` de `gcc` nous renseigne sur le rôle de l'option `-E` :

```
-E  Stop after the preprocessing stage; do not run the
    compiler proper.  The output is in the form of
    preprocessed source code, which is sent to the standard
    output.
```

Ainsi l'option `-E` consiste en l'exécution du préprocesseur `cpp` qui interprète toutes les *directives* débutant par `#` telles : `#include`, `#define`, `#ifdef`, etc.

Nous pouvons en déduire que `#include <stdio.h>` signifie que le préprocesseur doit inclure le fichier `stdio.h` (situé dans le répertoire `/usr/include` habituellement) en lieu et place de la directive. Ce sont les déclarations que nous avons vu apparaître. Quant à la directive `#define N 10`, elle définit le nom symbolique comme remplaçant la valeur `10`. Le précompilateur remplace donc toutes les occurrences de `N` par `10`

Exercice 2:

1. Nous observons à l'intérieur du fichier `prog1.s` la transcription du programme `prog1.c` en langage assembleur. En effet, d'après page `man` de `gcc`, voici le rôle tenu par l'option `-S` :

   ```
   -S  Stop after the stage of compilation proper; do not
       assemble.  The output is in the form of an assembler
       code file for each non-assembler input file specified.
   ```

   `gcc` produit donc dans ce cas le code assembleur correspondant à un fichier C préprocessé

2. On observe des différences entres les deux fichiers pour la simpel et bonne raison qu'ils ont été compilés sur des architectures différentes, donc vers des langages d'assemblages différents. Votre fichier a été compilé sur une SPARC/Solaris, alors que le fichier montré dans l'énoncé a été compilé sur un PowerPC/MacOSX.

   La phase de compilation est donc celle qui transforme un code C indépendant de la plateforme en un code assembleur dépendant de celle-ci.

Exercice 3:

Le fichier `prog1.o` contient le *code objet* correspondant au programme `prog1.c`. Le code objet est obtenu par assemblage du code assembleur. C'est essentiellement la traduction du code assembleur qui est encore lisible par un être humain en un langage lisible par l'ordinateur.

D'après la page `man` de `gcc`, le rôle de l'option `-c` est le suivant :

```
-c  Compile or assemble the source files, but do not link.
    The linking stage simply is not done.  The ultimate
    output is in the form of an object file for each source
    file.
```

Ceci confirme bien le fait que `gcc` va jusqu'à produire le code objet lorsque l'option `-c` lui est passée

Exercice 4:

La commande `gcc -o prog1 prog1.o` termine le processus de compilation en effectuant *l'édition des liens*. Cette étape consiste principalement à lier les appels des fonctions au code desdîtes fonctions.

Le fichier `prog1` contient alors un programme *exécutable* qui correspond au code C contenu dans `prog1.c` et au code de toutes les fonctions utilisées par celui-ci.

La commande `gcc -o prog1 prog1.o` indique que les fonctions contenues dans `prog1.o` vont être utilisées pour l'édition des liens. Il est par ailleurs implicite que les fonctions de la bibliothèque C standard (`printf`, `scanf`, etc) sont elles aussi ajoutées.

Il est parfois nécessaire d'ajouter d'autres bibliothèques. Nous avons vu qu'au niveau du code C, cela se fait au moyen de la directive `#include <math.h>` (pour le cas de la bibliothèque mathématique). Ceci permet au compilateur de vérifier que les fonctions de cette bibliothèque (`sqrt`, `sin`, etc) sont correctement utilisées. Cependant, à l'exécution, le code de ces fonctions est nécessaire. La bibliothèque n'est pas liée implicitement aux programmes, il faut pour cela spécifier explicitement l'option `-lm` (pour la bibliothèque mathématique, `-lXYZ` pour la bibliothèque `XYZ`). C'est ce que nous avions fait dans la feuille TD2 pour l'exercice de calcul de la somme de vecteurs par la commande : `gcc prog.c -Wall -lm -o prog`.

Remarque : l'édition des liens n'est pas directement effectuée par `gcc`, mais par le programme `ld` appelé par `gcc`. C'est pourquoi, les messages d'erreur d'édition des liens font apparaître le programme `ld`

Exercice 5:

D'après les pages `man` de `gcc`, nous voyons que les options `-E`, `-S` et `-c` réalisent de plus en plus d'opérations : `-S` réalise `-E` ainsi que la compilation et `-c` réalise `-S` ainsi que l'assemblage.

En pratique, pour obtenir le code objet pour un programme `prog.c`, nous pouvons donc nous contenter de la commande :

```
gcc -Wall -c prog.c
```

Remarque : l'option `-o prog.o` n'est pas nécessaire, c'est le fichier objet créé par défaut.

Ensuite, pour obtenir un programme exécutable `prog`, il suffit d'exécuter la commande d'édition des liens :

```
gcc -o prog prog.o
```

Remarque : comme nous l'avons fait jusqu'ici, nous pouvons toujours procéder en une seule commande :

```
gcc -Wall -o prog prog.c
```

qui en l'absence de toute option n'arrête par le processus de compilation et va jusqu'à l'édition des liens.

2 Code modulaire

Exercice 6:

1. Ces deux fichiers vont reprendre la partie du code dédiée au type `struct point`. Le fichier de déclaration (`point.h`) sert au compilateur à déterminer si l'utilisation du type `struct point` est correcte. Pour cela, il doit disposer d'une déclaration du type et de la déclaration de toutes les fonctions utilisables sur ce type :

   ```
   struct point {
     float x, y;
   };

   struct point point__creer(float, float);
   struct point point__copier(struct point);
   int point__egal(struct point, struct point);
   void point__afficher(struct point);
   ```

   Le fichier d'implantation (`point.c`) sert à fournir le code des fonctions déclarées dans le fichier de déclaration :

   ```
   #include <stdio.h>
   #include "point.h"

   struct point point__creer(float x, float y) {
     struct point p;

     p.x = x;
   ```

```
  p.y = y;

  return p;
}

struct point point__copier(struct point p) {
  return p;
}

int point__egal(struct point p1, struct point p2) {
return ((p1.x == p2.x) && (p1.y == p2.y));
}

void point__afficher(struct point p) {
  printf("(%f, %f)\n", p.x, p.y);
}
```

Notons que le fichier `point.c` inclue le fichier `point.h` puisque lors de sa compilation, `gcc` devra connaître le type `struct point` qui y est défini. On remarque aussi que la directive d'inclusion utilise des double-quote `"point.h"` plutôt que `<point.h>` car la bibliothèque `point` est définie dans le répertoire courant (et non pas dans le répertoire de la bibliothèque C standard).

Le fichier d'implantation peut contenir des définitions de fonctions dont la déclarations n'est pas fournie dans le fichier de déclaration. Ceci permet de définir des fonctions internes à la bibliothèque auxquelles on ne veut pas fournir d'accès à l'utilisateur.

Remarque : d'un point de vue méthodologique, le fichier `point.h` vous sert à donner à l'utilisateur de la bibliothèque `point` les informations dont il a besoin pour l'utiliser. Idéalement, il ne doit même pas avoir accès au fichier d'implantation `point.c`

La commande de compilation pour la bibliothèque est :

```
gcc -Wall -c point.c
```

2. Modifier le programme revient donc à retirer le code qui a été déporté dans la bibliothèque. Il faut en outre inclure le fichier de déclaration `point.h` pour que le compilateur dispose des déclarations du type `struct point` et des différentes fonctions. Ceci nous donne le programme :

```
#include "point.h"

int main() {
  struct point p1, p2;

  p1 = point__creer(3.2, 4.7);
  point__afficher(p1);
  p2 = point__copier(p1);
  point__afficher(p2);

  return 0;
}
```

Notons qu'il n'est plus utile d'inclure `stdio.h` puisque la fonction `main` ne fait appel à aucune fonction d'entrée-sortie

La commande de compilation du programme est donc :

```
gcc -Wall -c prog3.c
```

L'édition des liens consiste à produire un programme exécutable. Pour cela, nous avons besoin du code de la fonction principale (contenu dans `prog3.o` après la compilation), mais aussi du code des fonctions appelées par ce programme (contenu dans `point.o` après la compilation). La commande qui lie ces codes et produit un fichier exécutable est :

```
gcc -o prog3 prog3.o point.o
```

3. À l'exécution, les deux programmes se comportent évidemment de façon identique
4. Le paradigme de la programmation modulaire est que l'utilisateur d'une bibliothèque doit passer par les fonctions offertes par celle-ci pour agir sur les variables dont le type est défini par la bibliothèque. Utiliser l'opérateur d'affectation contrevient à ceci. L'interêt de ceci est qu'une modification du type est prise en charge par les fonctions et donc (quasi)transparente pour l'utilisateur, ce que ne peut pas garantir l'opérateur d'affectation (voir l'exercice «Maintenant et approche modulaire»)

Remarque : la notion de «bibliothèque» correspond en C à un couple de fichiers de déclaration/d'implantation (ou même éventuellement plusieurs fichiers de déclaration et d'implantation)

Exercice 7:

Voici le programme :

```
#include "point.h"

int main() {
  struct point p;

  p = point__creer(2.9876, -0.21341);
  point__afficher(p);

  return 0;
}
```

Nous compilons `prog4.c` par la commande :

```
gcc -Wall -c prog4.c
```

Notons qu'il n'est pas nécessaire de compiler à nouveau `point.c` car cela a déjà été fait lors de l'exercice précédent et aucune modification n'a été apportée au code.

L'édition des liens se fait alors par la commande :

```
gcc -o prog4 prog4.o point.o
```

qui génère un fichier exécutable `prog4`.

Remarque : nous voyons un autre bénéfice de la programmation modulaire dans la réutilisation du code sans recopier celui-ci. Ceci facilite la maintenance du code : il y a un unique fichier à modifier lorsqu'une erreur est découverte ou qu'une fonctionnalité doit être ajoutée

Exercice 8:

1. Nous devons modifier :
   - la définition du type `struct point` pour y ajouter le champ `c` de type `enum couleur` et ajouter la définition de ce type à la bibliothèque
   - le code des 3 fonctions définies dans la bibliothèque pour qu'elles prennent en compte le champ `c`

   Ces modifications doivent préserver l'interface de la bibliothèque, c'est à dire les prototypes des fonctions définis dans `point.h` autant que possible puisque tous les programmes qui utilisent la bibliothèque ont été créés sur la foi de celles-ci. Toute modification dans ces prototypes engendre potentiellement des modifications dans les programmes utilisant la bibliothèque
2. Afin d'être conservatifs, nous devons préserver les interfaces des fonctions. Or, dans le cas du *constructeur* `point__creer`, ceci n'est pas possible car nous devons passer une information de couleur pour initialiser correctement la variable de type `struct point`.

   Or, si nous modifions le prototype de cette fonction, nous devons aussi modifier les programmes l'utilisant. Dans notre cas, ceci n'est pas grave puisqu'il n'y a que deux minuscules programmes, mais il n'en serait pas de même avec une bibliothèque distribuée à grande échelle et utilisée par des centaines ou des milliers de programmes, ou par des programmes ayant des milliers de lignes.

   Nous pouvons cependant atteindre notre but. Nous allons préserver la fonction `point__creer` conservant ainsi l'interface existante. La convention est que cette fonction initialise le champs `c` à une valeur fixée, par exemple `NOIR`. Puis nous ajoutons un deuxième constructeur qui permet, lui, d'initialiser la couleur à une valeur choisie par l'utilisateur. Ceci nous donne l'interface de bibliothèque suivante :

```
enum couleur {
  ROUGE, BLEU, VERT, BLANC, NOIR, JAUNE
};

struct point {
  float x, y;
  enum couleur c;
};

struct point point__creer(float, float);
struct point point__creer_couleur(float, float, enum couleur);
struct point point__copier(struct point);
int point__egal(struct point, struct point);
void point__afficher(struct point);
```

On y remarque le nouveau type `enum couleur` pour le représentation des couleurs, et le nouveau constructeur `point__creer_couleur`.
Le code est modifié pour tenir compte du nouveau champs `c` et pour fournir l'implantation de `point__creer_couleur` :

```
#include <stdio.h>
#include "point2.h"

struct point point__creer(float x, float y) {
  struct point p;

  p.x = x;
  p.y = y;
  p.c = NOIR;

  return p;
}

struct point point__creer_couleur(float x, float y, enum couleur c) {
  struct point p;

  p = point__creer(x, y);
  p.c = c;

  return p;
}

struct point point__copier(struct point p) {
  return p;
}

int point__egal(struct point p1, struct point p2) {
return ((p1.x == p2.x) && (p1.y == p2.y) && (p1.c == p2.c));
}

void point__afficher(struct point p) {
  printf("(%f, %f) ", p.x, p.y);
  switch(p.c) {
  case ROUGE:
    printf("ROUGE");
    break;
  case BLEU:
    printf("BLEU");
    break;
  case VERT:
    printf("VERT");
    break;
```

```
    case BLANC:
      printf("BLANC");
      break;
    case NOIR:
      printf("NOIR");
      break;
    case JAUNE:
      printf("JAUNE");
      break;
    }
    printf("\n");
}
```

Le nouveau constructeur utilise l'ancien pour les initialisation des coordonnées : ceci évite de réécrire le code inutilement.

Remarque : absolument aucune modification n'est nécessaire dans le code de nos deux programmes !

3. Nous avons modifié notre bibliothèque, nous devons donc la recompiler :

   ```
   gcc -Wall -c point.c
   ```

   Bien que les interfaces utilisées par nos programmes n'aient pas changé, il est nécessaire de les recompiler car le type `struct point` a été modifié :

   ```
   gcc -Wall -c prog3.c
   gcc -Wall -c prog4.c
   ```

   Enfin, on termine par l'édition des liens :

   ```
   gcc -o prog3 prog3.o point.o
   gcc -o prog4 prog4.o point.o
   ```

   Remarque : sans la modification du type, il aurait même été inutile de recompiler nos programmes : l'édition des liens avec la nouvelle version de `point.o` aurait suffit.

   Ceci explique aussi pourquoi on passe bien souvent des pointeurs sur les structures en paramètres aux fonctions (et non pas les structures), car quel que soit le changement interne à la structure, le paramètre de la fonction ne change pas (il s'agit toujours d'un pointeur sur cette structure), n'imposant alors pas de recompilation. Il faut bien entendu voir cela dans le cadre d'un très gros programme, installé chez un client, où on n'a pas la liberté d'agir simplement, soit parce qu'on agit à distance, soit parce qu'une recompilation nécessite de rendre inopérante l'application pendant trop longtemps. Certaines applications que vous utilisez aujourd'hui (le navigateur `firefox`, la suite bureautique `openoffice.org`, etc) nécessitent plusieurs heures de compilation sur nos machines de bureau. C'est souvent bien pire lorsqu'il s'agit d'applications professionnelles.

Exercice 9:

1. D'après le paragraphe ci-dessus, nous avons besoin : d'ajouter un point à la ligne, d'en retirer un, et de modifier un point. Avant l'ajout de tout point, la ligne n'en

contient aucun. Il nous faut donc une fonction nous permettant d'obtenir une ligne vide à partir de laquelle nous pourrons construire nos lignes polygônales. Enfin, après avoir lu l'énoncé, nous savons que nous devons afficher les lignes.

La fonction d'ajout, de modification et de retrait d'un point doit prendre les coordonnées du point en paramètre. Nous avons deux choix : soit nous représentons ces coordonnées par une valeur de type `struct point`, soit nous utilisons deux valeurs de type `float`. Nous allons préférer la première solution pour la raison suivante. Supposons que nous décidions de travailler dans un univers à 3 dimensions, avec la seconde solution, il faut modifier les prototypes des fonctions de la bibliothèque `ligne`, ce qui demande beaucoup de travail et qui est dénué de sens : en 3D comme en 2D, une ligne polygônale est une suite de points. Avec la première solution, toutes les modifications sont internes à la bibliothèque `point` (comme toutes les solutions, celle que nous retenons présente aussi des défauts que nous ignorons ici : à vous d'en trouver).

Nous obtenons donc les prototypes suivants :

```
void ligne__vide(ligne);
void ligne__ajouter_point(ligne, point);
void ligne__retire_point(ligne, point);
void ligne__modifier_point(ligne, point, point);
void ligne__afficher(ligne);
```

2. Prenons l'exemple d'une ligne de 5 points dans laquelle nous devons retirer le 3ème point. Notre ligne est donc le tableau suivant :

```
|-----------|-----------|-----------|-----------|-----------|
| (1.0,2.3) | (4.5,3.2) | (1.0,0.5) | (0.7,0.7) | (9.8,1.1) |
|-----------|-----------|-----------|-----------|-----------|
```

Sachant que le tableau est une suite de cases contigües, il n'est pas possible de supprimer la 3ème case. Il faut donc recopier les points 4 et 5 dans les cases 3 et 4 respectivement, pour obtenir in fine :

```
|-----------|-----------|-----------|-----------|-----------|
| (1.0,2.3) | (4.5,3.2) | (0.7,0.7) | (9.8,1.1) | (9.8,1.1) |
|-----------|-----------|-----------|-----------|-----------|
```

La dernière case n'étant plus utilisée. Accessoirement, il manque à ce type le nombre de points qui composent la ligne.

Hormis le manque d'une information, nous avons de plus un algorithme de suppression inefficace puisqu'il nécessite dans le pire des cas (suppression du 1er élément) le décalage de toutes les cases du tableau.

3. Le problème que nous avons soulevé provient du fait que les cases d'un tableau sont nécessairement contigües. Comment peut-on s'affranchir de cette contrainte ? Dans un tableau, la contigüité sert à définir la position de la case suivante : pour une case donnée du tableau, la case suivante (s'il y a lieu) est celle qui la suit immédiatement en mémoire. Pour briser cette constrainte, il nous suffit donc d'ajouter une information dans chaque case du tableau qui indique l'adresse mémoire de la case suivante. C'est la structure bien connue de *liste chaînée*.

Sur notre exemple, nous voyons que cela résout notre problème de suppression. À partir de la configuration initiale :

```
   ffbff10     ffbff11     ffbff12     ffbff13     ffbff14
|-----------|-----------|-----------|-----------|-----------|
| (1.0,2.3) | (4.5,3.2) | (0.7,0.7) | (9.8,1.1) | (9.8,1.1) |
| ffbff11   | ffbff12   | ffbff13   | ffbff14   | ?         |
|-----------|-----------|-----------|-----------|-----------|
```

Pour supprimer la troisième case, il suffit de dire que la suivante de la deuxième est maintenant la quatrième en modifiant l'information de pointage :

```
tete: ffbff10
```

```
   ffbff10     ffbff11     ffbff12     ffbff13     ffbff14
|-----------|-----------|-----------|-----------|-----------|
| (1.0,2.3) | (4.5,3.2) | (0.7,0.7) | (9.8,1.1) | (9.8,1.1) |
| ffbff11   | ffbff13   | ffbff13   | ffbff14   | ?         |
|-----------|-----------|-----------|-----------|-----------|
```

Notons que nous devons de plus conserver l'adresse de la première case occupée, autrement dit la *tête* de la liste (pensez à la suppression du premier point pour vous convaincre de l'utilité de ceci).

Il nous reste deux problèmes à résoudre :

- puisque notre liste chaînée est stockée dans un tableau, nous avons deux moyens d'en identifier une case, soit par son indice dans le tableau (0, 1, 2, etc), soit par son adresse (`ffbff10`, `ffbff11`, etc). Les deux choix sont totalement valides ici.
  Nous allons choisir les adresses pour simplifier l'écriture des fonctions sur le type `ligne` (nous reviendrons sur ce point plus tard). Notons que ce n'est pas un choix systématique, et le fait de prendre des pointeurs peut aussi poser de graves problèmes auxquels nous ne serons pas confrontés ici (ce point sera abordé au S2).
  Dernier aspect à traiter sur ce problème : quelle est l'adresse de la case suivant la dernière case ? Il n'existe évidemment pas de telle case, et nous prendrons ici très naturellement l'adresse `NULL`
- sur l'exemple ci-dessus, une fois la suppression effectuée, la troisième case devient libre, c'est à dire qu'elle ne stocke plus d'information valide. Si nous voulons ajouter un nouveau point à la liste, il est intéressant de réutiliser cette case. Plus généralement, initialement, la ligne est vide de tout point, donc toutes les cases sont libres. Il faut donc distinguer les cases qui sont libres de celles qui ne le sont pas.
  Nous avons essentiellement deux choix : soit une troisième information dans chaque case du tableau qui indique si celle-ci est libre ou non :
  ```
  tete: ffbff10
  ```

  ```
     ffbff10     ffbff11     ffbff12     ffbff13     ffbff14
  |-----------|-----------|-----------|-----------|-----------|
  | (1.0,2.3) | (4.5,3.2) | (0.7,0.7) | (9.8,1.1) | (9.8,1.1) |
  | ffbff11   | ffbff13   | ffbff13   | ffbff14   | NULL      |
  | occupée   | occupée   | libre     | occupée   | occupée   |
  |-----------|-----------|-----------|-----------|-----------|
  ```

  soit une seconde liste chaînée qui lie entre elles les cases libres :
  ```
  tete: ffbff10
  libres: ffbff12
  ```

```
   ffbff10     ffbff11     ffbff12     ffbff13     ffbff14
|-----------|-----------|-----------|-----------|-----------|
| (1.0,2.3) | (4.5,3.2) | (0.7,0.7) | (9.8,1.1) | (9.8,1.1) |
| ffbff11   | ffbff13   | NULL      | ffbff14   | NULL      |
|-----------|-----------|-----------|-----------|-----------|
```

Nous allons opter pour la seconde possibilité. En effet, dans le premier cas, la recherche d'une case vide pour ajouter un nouveau point peut nécessiter le parcours de tout la tableau, alors que dans le second cas, il suffit de regarder la tête de la liste des cases libres.

Notons que la liste des cases libres est une liste chaînée : il n'y a pas de case suivant sa dernière case, donc dans l'exemple ci-dessus, la case d'adresse `ffbff12` pointe sur la case suivante d'adresse `NULL`.

Une liste chaînée est une séquence de *cellules* (ou cases). Voici les informations qui doivent être stockées pour chaque cellule :
- le point stocké dans la cellule
- l'adresse de la cellule suivante

Nous en déduisons donc la structure de données suivante :

```
struct cellule {
  struct point p;
  struct cellule *suite;
};
```

Remarquons que l'ajout d'un point dans la ligne se fait en fin de ligne. Cette opération nécessite donc potentiellement de parcourir toute la liste des points. Afin de rendre l'opération plus efficace, nous pouvons conserver l'adresse de la dernière cellule occupée dans la liste des points. Pour représenter la ligne, nous avons donc :
- un tableau de cellules
- l'adresse de la première cellule contenant un point de la ligne
- l'adresse de la dernière cellule contenant un point de la ligne
- l'adresse de la première cellule vide

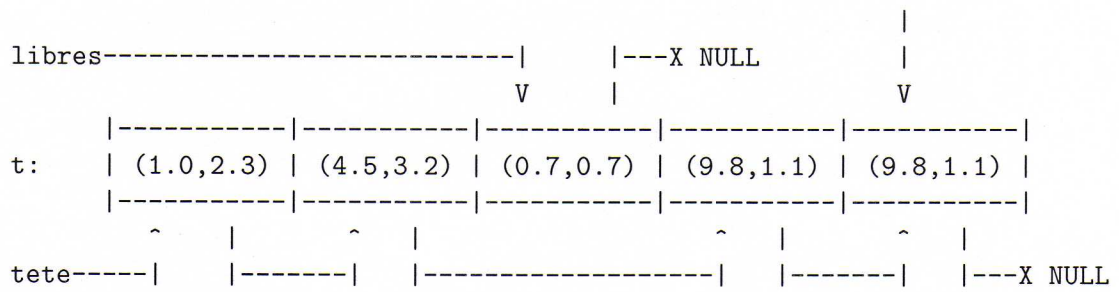
Nous en déduisons la structure de données suivante :

```
#define MAX_PT 30

struct s_ligne {
  struct cellule t[MAX_PT];
  struct cellule *tete;
  struct cellule *queue;
  struct cellule *libres;
};

typedef struct s_ligne ligne;
```

Le type `ligne` est un alias du type `struct s_ligne`.

Voici une illustration de cette structure de données sur l'exemple précédent issu du retrait du 3ème élément :

```
queue-------------------------------------------------|
```



Le fichier interface correspondant est donc :

```
#include "point.h"

/* Nombre max de points dans une ligne */
#define MAX_PT 30

/* Cellule de la liste chainee de points */
struct cellule {
  struct point p;
  struct cellule *suite;
};

/* Representation d'une ligne */
struct s_ligne {
  struct cellule t[MAX_PT];
  struct cellule *tete;
  struct cellule *queue;
  struct cellule *libres;
};

/* Alias pour simplifier le nommage */
typedef struct s_ligne ligne;

void ligne__vide(ligne *);
void ligne__ajouter_point(ligne *, struct point);
void ligne__retirer_point(ligne *, struct point);
void ligne__modifier_point(ligne *, struct point, struct point);
void ligne__afficher(ligne *);
```

Nous avons modifié les prototypes des fonctions pour passer des pointeurs : `ligne *` plutôt que des lignes. Nous vous laissons réfléchir aux raisons qui nous ont motivés.

4. Commençons par l'implantation de la fonction `ligne__vide`. À l'issue de l'exécution de cette fonction, la structure `ligne` doit représenter une ligne vide de tous points :
   - la liste des points étant vides, `tete` et `queue` pointent sur `NULL`
   - toutes les cellules de `t` sont libres, donc `libres` pointe sur la premiere cellule, et elles sont toutes chaînées entre elles.
     Chaîner les cellules entre elles revient à faire pointer `c->suite` (pour `c` de type `struct cellule *` pointant sur une case de `t`) sur `c+1`, la case suivante de `t` (par

l'arithmétique de pointeurs). Ceci à l'exception de la dernière cellule pour laquelle `c->suite` pointe sur `NULL`

L'ajout d'un point en fin de ligne par la fonction `ligne__ajouter_point` consiste tout d'abord à trouver une cellule vide. On prend la première de la liste `libres` s'il y en a une. Nous devons donc retirer la cellule choisie `c_libre` de la liste des cellules libres :

```
libres      -------    --------
  |------->|      |--->|       |...
             -------    --------
```

doit devenir :

```
  |-----------------|
libres      -------  | --------
         |->|      |  ->|       |...
  |------|  -------    --------
  |
c_libre
```

Ensuite, nous devons recopier le point `p` à ajouter dans le champ `p` de `c_libre`. Pour finir, nous devons ajouter `c_libre` à la fin de la ligne :

- `c_libre` n'a pas de cellule suivante puisque `p` est le dernier point de la ligne

```
             -------
c_libre----->|      |---X NULL
             -------
```

- si la ligne est vide de tous points, alors `c_libre` devient le premier point de la ligne

```
tete---X NULL
```

```
             -------
c_libre----->|      |---X NULL
             -------
```

devient :

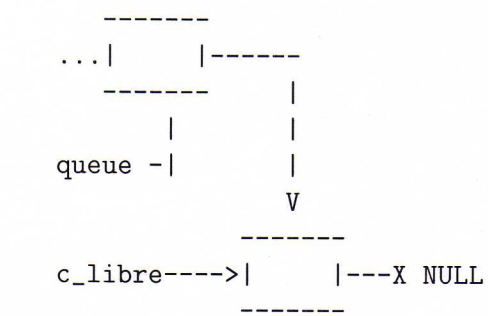
```
tete------------|
                V
             -------
c_libre----->|      |---X NULL
             -------
```

- si la ligne n'est pas vide, alors `c_ligne` devient le suivant de l'ancien dernier point de la ligne

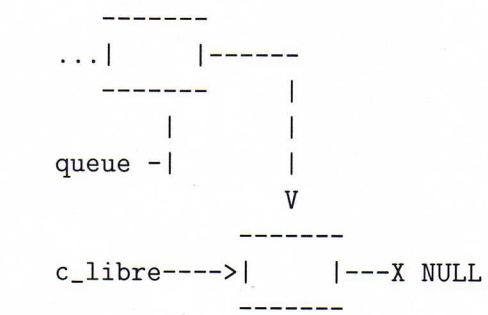
```
   -------
...|      |---X NULL
   -------
       |
queue -|
```

```
             -------
c_libre----->|      |---X NULL
             -------
```

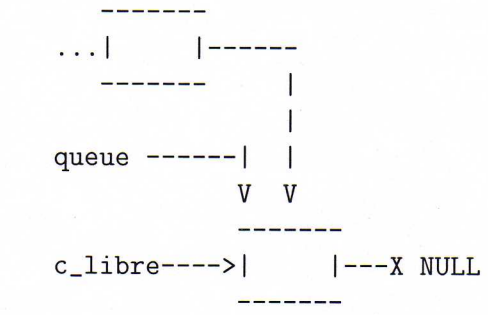
devient :



- `c_ligne` devient la dernière cellule de la liste, `p` étant le dernier point de la ligne

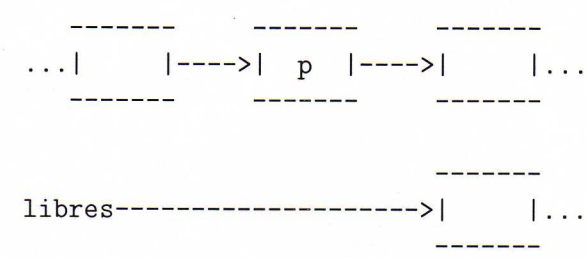


devient :

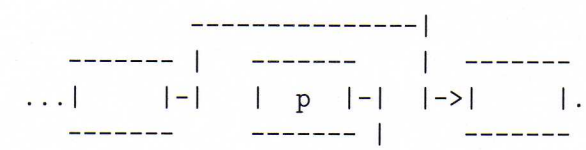


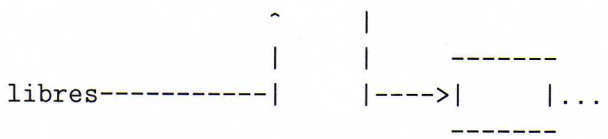
(notons que dans ce cas, `queue` pointait éventuellement sur `NULL` si la ligne était vide, mais ça n'a pas d'influence sur le code)

Nous considérons maintenant le retrait d'un point `p` par la fonction `ligne__retirer_point`. Imaginons que nous sachions dans quelle cellule se trouve `p` :



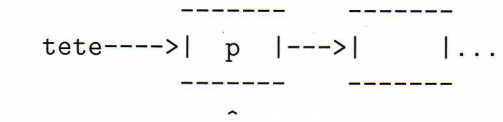
Le retrait consiste alors à modifier le chaînage pour éliminer la cellule de la ligne, et l'ajouter en tête (par simplicité) à la liste des cellules libres :



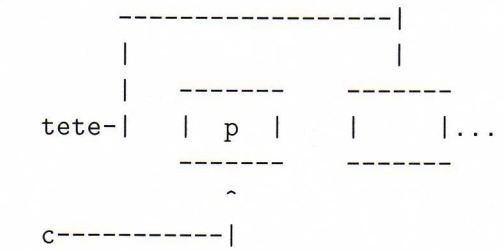


Notons que pour modifier le chaînage nous devons connaître l'adresse de la cellule qui précède celle contenant `p`. L'opération de retrait consiste donc en :

- la recherche de la première cellule qui contient `p` (s'il y en a une). Ceci se fait en parcourant la liste qui est vue comme une file. La file considérée ici est la suite des adresses des cellules de la ligne terminée par `NULL`. La butée n'est pas une adresse valide, elle n'est donc pas traitée comme les autres, nous avons donc un schéma de programme de style `while` que nous complétons en répondant aux questions suivantes :
  - `<obtenir le 1er élément>` il nous faut désigner le premier élément de la file, donc l'adresse de a première cellule de la liste. Pour cela, nous introduisons une variable `c` de type `struct cellule *`. Le premier élément est donné par la valeur de `c`
  - `<obtenir l'élément suivant>` consiste à considérer l'élément suivant dans la liste : `c = c->suite`
  - `<élément != butée>` la butée est atteinte ici soit lorsqu'on arrive à la fin de la liste, lorsque `c` est égale à `NULL`, soit lorsque `c` contient un point égal à `p`. Ce test s'écrit donc : `(c != NULL) && ( ! point_egal(c->p, p))`
  - `<traiter l'élément courant>` il s'agit de conserver le pointeur sur la cellule courante (qui sera la cellule précédent celle contenant `p` cas échéant). Pour cela, nous introduisons une variable `c_prec` de type `struct cellule *`, et le traitement s'écrit : `c_prec = c`
  - `<initialisations>` pour que `c` désigne le premier élément de la file, il faut l'initialiser à `l->tete`. Enfin, avant tout traitement, la cellule précédent `c` à l'adresse `NULL` (il n'y en a pas)
  - `<présentation du résultat>` il s'agit simplement de la valeur de `c` qui vaut `NULL` si `p` n'apparaît pas dans la ligne et l'adresse de la cellule le contenant sinon
- si une cellule contenant `p` a été trouvée (`c` est non `NULL`), alors, il se présente deux cas. Le premier a été illustré ci-dessus : il s'agit du cas où la cellule contenant `p` n'est pas la première de la ligne. Nous illustrons ici le second cas (`c_prec` vaut `NULL`) :

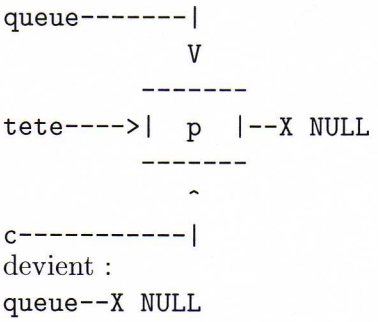


devient :

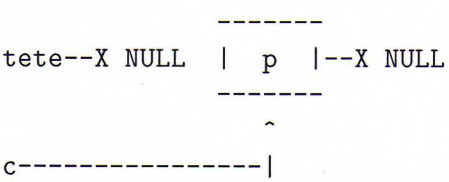


La tête de liste pointe sur la suite de la liste (cela est valide même lorsque la suite de liste est `NULL`), et `c` est ajoutée à la liste des cellules libres comme illustré ci-dessus.

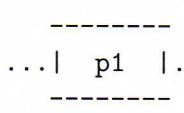
Dans le cas où la cellule retirée est la seule de la ligne, il faut par ailleurs mettre à jour le pointeur sur la dernière cellule `queue` qui devient alors `NULL` puisqu'il n'y a plus aucune cellule :



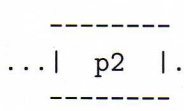
devient :



Pour rappel, le traitement donné pour `tete` ci-dessus est encore valide dans ce cas.
L'implantation de la fonction `ligne__modifier_point` consiste à transformer :



en :



Nous remarquons qu'il n'y a aucune manipulation de chaînage à effectuer, il s'agit simplement de changer une valeur. Le cœur du traitement est la recherche de la première cellule contenant le point `p1` (s'il y en a une). Il s'agit d'un traitement de file identique au cas du retrait d'un point, sauf qu'il n'est pas nécessaire de conserver un pointeur sur la cellule précédente puisqu'on ne fait aucune modification de chaînage. À l'issue de la recherche, `c` contient soit `NULL`, soit l'adresse de la cellule contenant `p1`, auquel cas, `c->p = point__copie(p2)` réalise le traitement souhaitée.

Enfin, l'affichage par la fonction `ligne__afficher` consiste là encore en un traitement de file identique aux précédents, hormis le traitement de l'élément qui doit ici être affiché. Ceci se réalise simplement par : `point__affiche(c->p)` en supposant que `c` pointe sur la cellule courante.

Nous obtenons donc le code suivant :

```
#include <stdio.h>
#include "point.h"
#include "ligne.h"

void ligne__vide(ligne *l) {
struct cellule *c;
```

```
/* Pas de points dans la ligne */
l->tete = NULL;
l->queue = NULL;
/* La première cellule libre est la première case du tableau */
l->libres = l->t;
/* Chainage des cellules libres */
c = l->t;
while (c < l->t+MAX_PT-1) {
c->suite = c+1;
c++;
}
/* Chainage de la derniere cellule libre */
c->suite = NULL;
}


void ligne__ajouter_point(ligne *l, struct point p) {
struct cellule *c_libre;

/* Recuperation d'une cellule libre */
if (l->libres != NULL) {
c_libre = l->libres;
l->libres = l->libres->suite;

/* Copie du point dans la cellule libre */
c_libre->p = point__copier(p);

/* Ajout de la cellule libre en fin de ligne */
c_libre->suite = NULL;
if (l->queue != NULL)
l->queue->suite = c_libre;
l->queue = c_libre;
if (l->tete == NULL)
l->tete = c_libre;
}
}


void ligne__retirer_point(ligne *l, struct point p) {
struct cellule *c_prec, *c;

/* Recherche de la cellule contenant p */
c_prec = NULL;
c = l->tete;
while ((c != NULL) && (! point__egal(c->p, p))) {
```

```
c_prec = c;
c++;
}

/* Teste si p appartient a la ligne */
if (c != NULL) {
/* Retire la cellule de la ligne */
if (c_prec == NULL) {
/* Si p est en premiere position dans la ligne */
l->tete = c->suite;
if (l->queue == c)
l->queue = NULL;
}
else {
/* Si p n'est pas en premiere position dans la ligne */
c_prec->suite = c->suite;
}
/* Ajoute la cellule a la liste des cellules libres */
c->suite = l->libres;
l->libres = c;
}
}



void ligne__modifier_point(ligne *l, struct point p1, struct point p2) {
struct cellule *c;

/* Recherche de la cellule contenant p */
c = l->tete;
while ((c != NULL) && (! point__egal(c->p, p1)))
c++;

/* Si p1 est dans la ligne, on le remplace par p2 */
if (c != NULL)
c->p = point__copier(p2);
}



void ligne__afficher(ligne *l) {
struct cellule *c;

/* Parcours de la ligne et affichage de chaque point */
c = l->tete;
while (c != NULL) {
point__afficher(c->p);
```

```
c++;
}
}
```

5. Voici le programme principal :

```
#include <stdio.h>
#include "ligne.h"

int main() {
ligne l;
int i;
struct point p;

ligne__vide(&l);
for (i=0; i<5; i++) {
p = point__creer(i*2.0, i*3.0);
ligne__ajouter_point(&l, p);
}
ligne__afficher(&l);

p = point__creer(4.0, 6.0);
ligne__retirer_point(&l, p);
printf("Retrait du point: ");
point__afficher(p);
ligne__afficher(&l);


return 0;
}
```

6. Voici les commandes de compilation et d'édition des liens :

```
gcc -Wall -c point.c
gcc -Wall -c ligne.c
gcc -Wall -c prog7.c
gcc -o prog7 prog7.o ligne.o point.o
```

On constate alors l'erreur suivante en compilant `ligne.c` :

```
cc -Wall  -c  ligne.c
In file included from ligne.h:1,
                 from ligne.c:3:
point.h:1: error: redefinition of 'struct point'
```

Le compilateur nous informe que le type `struct point` est défini deux fois. Ceci est dû au fait que `ligne.h` inclut `point.h` et que `ligne.c` inclut à la fois `ligne.h` et `point.h`. Donc au niveau de `ligne.c`, la bibliothèque `point.h` est incluse 2 fois, entraînant la double définition du type `struct point`.

Une première solution consiste à retirer certaines inclusions en comptant sur le fait que le fichier retirer à un endroit est inclus à un autre endroit. Par exemple, nous

pourrions retirer l'inclusion de `point.h` dans `ligne.c` en sachant qu'il est inclus dans `ligne.h` lui-même inclus dans `ligne.c`. Cependant, ce serait une très mauvaise idée. D'une part, ceci ne serait pas maintenable : il n'y a aucune trace de quelle bibliothèque est incluse par quelle bibliothèque. D'autre part, sur des programmes incluant plusieurs bibliothèque cette recherche serait complètement impossible a réaliser (c'est déjà le cas pour `stdio.h`). Enfin, ce n'est pas toujours possible.
La bonne solution consiste à utiliser les directives de compilation `#ifndef` et `#define`. Nous modifions tous nos fichiers d'interface pour qu'ils aient la structure suivante :

```
#ifndef POINT_H
#define POINT_H

<declarations>

#endif
```

Lors de la première inclusion du fichier, la variable `POINT_H` n'est pas définie, donc le test `#ifndef` (if not defined) passe. La directive `#define POINT_H` définit maintenant cette variable puis les déclarations sont produites. Pour les inclusions suivantes, la variable `POINT_H` est maintenant définie, et les déclarations ne sont plus produites. Ainsi, chaque déclaration n'est produite qu'une seule fois. Il faudra maintenant prendre l'habitude de procéder ainsi.
Voici les nouveaux fichiers d'interface `point.h` :

```
#ifndef POINT_H
#define POINT_H

struct point {
  float x, y;
};

struct point point__creer(float, float);
struct point point__copier(struct point);
int point__egal(struct point, struct point);
void point__afficher(struct point);

#endif
```

et `ligne.h` :

```
#ifndef LIGNE_H
#define LIGNE_H

#include "point_def.h"

/* Nombre max de points dans une ligne */
#define MAX_PT 30

/* Cellule de la liste chainee de points */
struct cellule {
```

```
  struct point p;
  struct cellule *suite;
};

/* Representation d'une ligne */
struct s_ligne {
  struct cellule t[MAX_PT];
  struct cellule *tete;
struct cellule *queue;
  struct cellule *libres;
};

/* Alias pour simplifier le nommage */
typedef struct s_ligne ligne;

void ligne__vide(ligne *);
void ligne__ajouter_point(ligne *, struct point);
void ligne__retirer_point(ligne *, struct point);
void ligne__modifier_point(ligne *, struct point, struct point);
void ligne__afficher(ligne *);

#endif
```

Avec ce mécanisme, nous continuons à inclure les bibliothèque que nous utilisons, sans nous soucier de leur inclusion dans une autre des bibliothèques utilisées par notre programme.

7. Après compilation, nous obtenons l'affichage :

```
(0.000000, 0.000000)
(2.000000, 3.000000)
(4.000000, 6.000000)
(6.000000, 9.000000)
(8.000000, 12.000000)
Retrait du point: (4.000000, 6.000000)
(0.000000, 0.000000)
(2.000000, 3.000000)
(6.000000, 9.000000)
(8.000000, 12.000000)
```

qui nous semble concluant.

Remarque : maintenant que vous avez développé deux bibliothèques, réfléchissez aux tests que vous devez effectuer pour garantir un niveau de fiabilité suffisant de votre code