Cours de Programmation Impérative des astuces pour bien coder

Julien David

A101 - david@lipn.univ-paris13.fr

La lisibilité du code

Un code lisible

Les techniques

- le nom des variables et des fonctions,
- l'indentation,
- les commentaires,
- le découpage en fonctions.

Le nom des fonctions et des variables

Que fait cette fonction ?

```
1 unsigned int f(unsigned int n){
2   unsigned int a=1, b=1;
3   unsigned int c;
4   for(c=0;c<n;c++){
5     a=a+b;
6     b=b-a;
7   }
8   return a;
9 }
```

Le nom des fonctions et des variables

```
1 unsigned int fibonacci(unsigned int n){
2   unsigned int current=1, prec=1;
3   unsigned int i;
4   for(i=0;i<n;i++){
5     current=current+prec;
6     prec=current-prec;
7   }
8   return current;
9 }
```

N'est-ce pas plus simple à lire ?

L'indentation

L'indentation

Vous trouvez ça lisible ?

```
1       unsigned int fibonacci(unsigned int n){
2  unsigned int current=1, prec=1;
3  unsigned int i;
4  for(i=0;i<n;i++)
5  {current=current+prec;
6     prec=current-prec;}
7  return current;}
```

L'indentation

C'est pourtant simple

Quand on commence :

- une boucle : `for`, `while`, `do while`, ...
- une condition : `if`, `else if`, `else`, ...
- la définition d'une fonction

Bref **à chaque fois qu'on écrit le symbole** {

- on va à la ligne,
- on se **décale à droite**.

A l'inverse, **à chaque fois qu'on écrit le symbole** }

- on va à la ligne,
- on se **décale à gauche**.

L'indentation

```
1 unsigned int fibonacci(unsigned int n){
2   unsigned int current=1, prec=1;
3   unsigned int i;
4   for(i=0;i<n;i++){
5     current=current+prec;
6     prec=current-prec;
7   }
8   return current;
9 }
```

Les commentaires

Les commentaires

Pour faire quoi ?

Lorsque l'on travaille sur un gros projet, il arrive souvent :

- que l'on travaille à plusieurs sur un même programme,
- que l'on doive rependre son code plusieurs mois plus tard.

On veut donc pouvoir :

- communiquer son code aux autres,
- relire son code plus facilement après une longue période.

Les commentaires

Que raconter ?

Les commentaires au début d'une fonction doivent décrire :

- les paramètres d'entrées,
- le résultat,
- un descriptif de ce que fait la fonction,
- **pour les balèzes** : la complexité en temps et en espace.

Les commentaires

La syntaxe

Il existe plusieurs normes. L'important est de rester **cohérent**.
Je donne ici, à titre d'exemple, la norme du logiciel **Doxygen** :

```
1 /**
2 Description de la fonction.
3 @param premier parametre de la fonction
4 @param deuxieme parametre de la fonction
5 @return ce que la fonction renvoie
6 */
```

Les commentaires

```
1  /**
2     Cette fonction calcule la n-ieme valeur de la fonction de fibonacci.
3     Rappel: fibonacci(n)=fibonacci(n-1)+fibonacci(n-2)
4     et fibonacci(0)=1 et fibonacci(1)=1
5     Complexite en temps: O(n)
6     Complexite en espace: O(1)
7     @param n est un entier positif.
8     @return la n-ieme valeur de la fonction de fibonacci.
9  */
10 unsigned int fibonacci(unsigned int n){
11   unsigned int current=1, prec=1;
12   unsigned int i;
13   for(i=0;i<n;i++){
14     current=current+prec;
15     prec=current-prec;
16   }
17   return current;
18 }
```

Les commentaires

Les commentaires c'est comme le café, la bouffe grasse, . . .

C'est bien mais il ne faut pas en abuser.

Les commentaires

```c
/**
   Cette fonction calcule la n-ieme valeur de la fonction de fibonacci.
   Rappel: fibonacci(n)=fibonacci(n-1)+fibonacci(n-2)
   et fibonacci(0)=1 et fibonacci(1)=1
   Complexite en temps: O(n)
   Complexite en espace: O(1)
   @param n est un entier positif.
   @return la n-ieme valeur de la fonction de fibonacci.
*/
unsigned int fibonacci(unsigned int n){ /* ca c'est la fonction fibonacci */
  unsigned int current=1, prec=1; /* La j'ai declare des variables */
  unsigned int i; /* Et puis la une autre, je trouvais ca mieux de le faire sur deux
  for(i=0;i<n;i++){ /* ca c'est une boucle for au cas ou ca ne serait pas clair */
    current=current+prec; /* la je calcule fibonacci(i)=fibonacci(i-1)+fibonacci(i-2
    prec=current-prec; /* la je me decale d'une valeur */
  }
  return current; /* je renvoie le resultat */
}
```

À vous de trouver un juste milieu

Le découpage en fonctions

Découpage en fonctions

En quoi est-ce mieux ?

- permet d'éviter de faire des copier/coller de code.
- améliore la lisibilité.
- un bout de code transformé en fonction est facilement réutilisable.

Découpage en fonctions

Reprenons le tri par sélection.

```
1 void tri_selection(int * tab,int n){
2   int i;
3   int pos_min;
4   for(i=0;i<n;i++){
5     pos_min=recherche_pos_min(&(tab[i]),n-i);
6     swap(&tab[i],&tab[pos_min]);
7   }
8 }
```

On lit directement que l'algorithme fait

- la recherche du plus petit élément d'une partie du tableau,
- l'echange de valeurs de deux cases.

En un examen rapide, on a un aperçu du principe de l'algorithme.

Découpage en fonctions

```
1  int recherche_position_min(int *tab, int n){
2    int i;
3    int pos_min=0;
4    for(i=1;i<n;i++){
5      if(tab[i]<tab[pos_min])
6        pos_min=i;
7    }
8    return pos_min;
9  }
10
11 void swap(int * x, int * y){
12   int tmp=*x;
13   *x=*y;
14   *y=tmp;
15 }
16
17 void tri_selection(int * tab,int n){
18   int i;
19   int pos_min;
20   for(i=0;i<n;i++){
21     pos_min=recherche_pos_min(&(tab[i]),n-i);
22     swap(&tab[i],&tab[pos_min]);
23   }
24 }
```

Les deux autres fonctions sont réutilisables dans d'autres contextes !

Découpage en fonctions

Essayons maintenant de comprendre le même algorithme, sans le découpage en fonctions.

```
1  void tri_selection(int * tab,int n){
2    int i,j,tmp;
3    int pos_min;
4    for(i=0;i<n;i++){
5      pos_min=tab[i];
6      for(j=i+1;j<n;j++){
7        if(tab[j]<tab[pos_min])
8          pos_min=j;
9      }
10     tmp=tab[i];
11     tab[i]=tab[pos_min];
12     tab[pos_min]=tmp;
13   }
14 }
```

Compilation séparée

Séparer le code

La séparation

- Définition d'une structure `truc`
  → `truc.h`
- Déclaration des fonctions qui manipulent la structure `truc`
  → `truc.h`
- Définition des fonctions qui manipulent la structure `truc`
  → `truc.c`
- Le programme, la fonction main
  → `main.c`

Exemple

On veut définir la notion de pixel

Un pixel se compose :

- Des coordonnées entières (un point).
- Une couleur.

Les structures

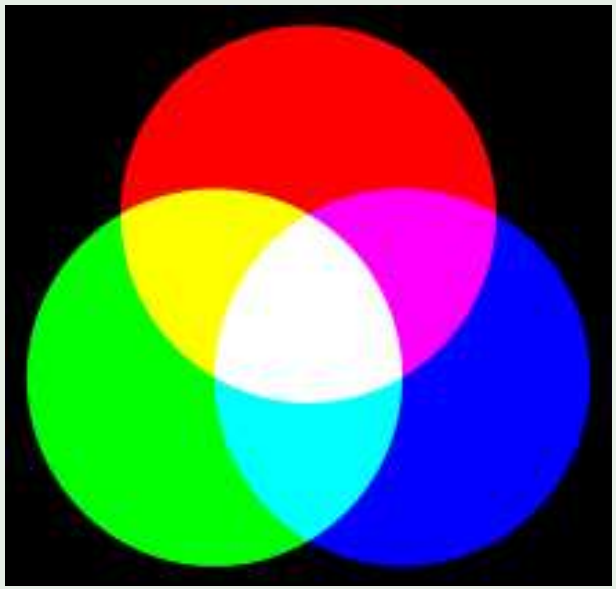
On aura donc besoin de définir :

- Une structure `point_t`.
- Une structure `couleur_t`.
- Une structure `pixel_t` qui utilise les deux premières.

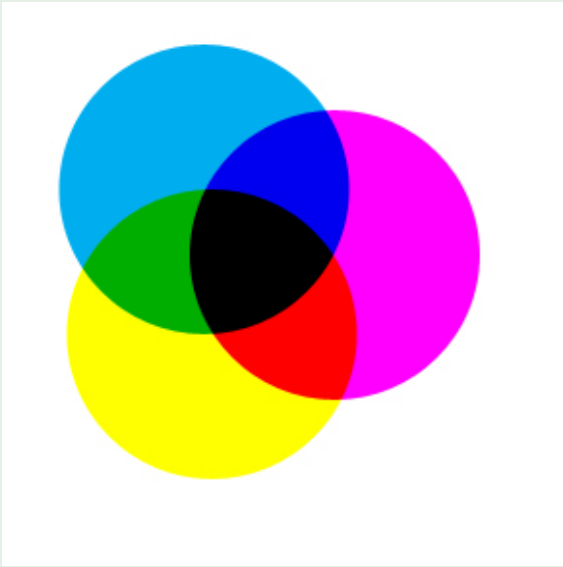
Une couleur ?

Il existe plusieurs façon de caractériser une couleur.

Modèle RVB



Modèle CMJN



Supposons que l'on choisisse le RVB .

Les structures

```c
/* Fichier point.h */
struct point_s{
  int x;
  int y;
};
typedef struct point_s point_t;
```

```c
/* Fichier couleur.h */
struct couleur_s{
  unsigned short rouge;
  unsigned short vert;
  unsigned short bleu;
};
typedef struct couleur_s couleur_t;
```

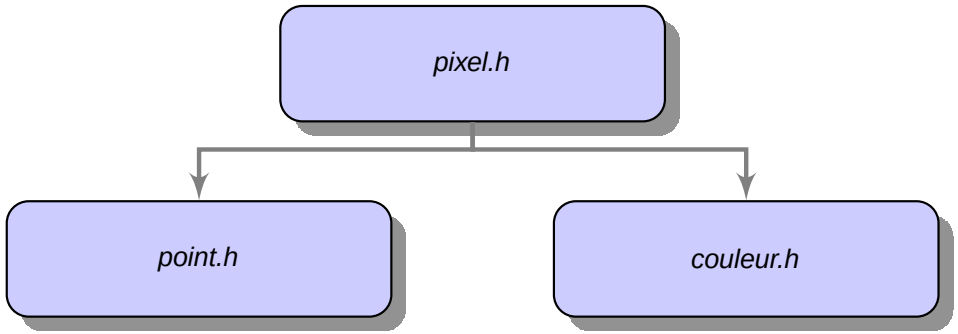
```c
/* Fichier pixel.h */
#include "point.h"
#include "couleur.h"

struct pixel_s{
  point_t coord;
  couleur_t col;
};
typedef struct pixel_s pixel_t;
```

Diagramme de dépendance

Dépendances

- La notion de `pixel_t` dépends de celle de `point_t` et de `couleur_t`,
- Ces dépendances sont identifiables grâce aux `include`,
- On représente ces dépendances à l'aide d'un diagramme.



Les fonctions

Manipuler des structures

Lorsqu'on utilise une variable structurée, le moins on a besoin de connaître comment cette structure est définie pour la manipuler, le mieux on se porte.

Pour cette raison

On va créer un ensemble de fonctions permettant de manipuler cette structure.

Déclarations des fonctions

```
1  /* Fichier point.h */
2  /* Definition de la structure */
3  struct point_s{
4    unsigned int x;
5    unsigned int y;
6  };
7  typedef struct point_s point_t;
8
9  /* Declaration des fonctions */
10
11 point_t creer_point(unsigned int x,unsigned int y);
12
13 int cmp_point(point_t p1,point_t p2);
14
15 void afficher_point(point_t p1);
16
17 void swap_point(point_t * p1,point_t * p2);
```

Déclarations des fonctions

```c
/* Fichier couleur.h */
/* Definition de la structure */
struct couleur_s{
  unsigned short rouge;
  unsigned short vert;
  unsigned short bleu;
};
typedef struct couleur_s couleur_t;

/* Declaration des fonctions */

couleur_t creer_couleur(unsigned short r,unsigned short v,unsigned short b);

int cmp_couleur(couleur_t p1,couleur_t p2);

void afficher_couleur(couleur_t p1);

void swap_couleur(couleur_t * p1,couleur_t * p2);
```

Déclarations des fonctions

```
1  /* Fichier pixel.h */
2  #include "point.h"
3  #include "couleur.h"
4
5  /* Definition de la structure */
6  struct pixel_s{
7    point_t coord;
8    couleur_t col;
9  };
10 typedef struct pixel_s pixel_t;
11
12 /* Declaration des fonctions */
13
14 pixel_t creer_pixel(point_t pos,couleur_t c);
15
16 int cmp_pixel(pixel_t p1,pixel_t p2);
17
18 void afficher_pixel(pixel_t p1);
19
20 void swap_pixel(pixel_t * p1,pixel_t * p2);
```

Définitions des fonctions

Les fichiers `.c`

- Le fichier `truc.c` contient un `#include ‘‘truc.h’’`
- si nécessaire, `truc.c` peut contenir d’autres `#include`
- chaque fonction déclarée dans `truc.h` doit être définie dans `truc.c`

Par manque de place,

sur les slides qui suivent :

- il manque les commentaires,
- on ne définie que certaines fonctions.

Définitions des fonctions

```
1  /* Fichier point.c */
2  #include "point.h"
3  #include<stdio.h>
4
5  point_t creer_point(unsigned int x,unsigned int y){
6    point_t res;
7    res.x=x;
8    res.y=y;
9    return res;
10 }
11
12 void afficher_point(point_t p1){
13   printf("X=%u Y=%u\n",p1.x,p1.y);
14 }
```

- On utilise la fonction `printf` donc on inclut `stdio.h`

Définitions des fonctions

```c
/* Fichier pixel.c */
#include "pixel.h"

/* Definition des fonctions */

void afficher_pixel(pixel_t p1){
  afficher_point(p1.coord);
  afficher_couleur(p1.col);
}
```

- On peut faire appel à des fonctions définies dans d'autres fichiers.
- Ici, la déclaration de ces fonctions est accessible par `pixel.h`.

Définitions des fonctions

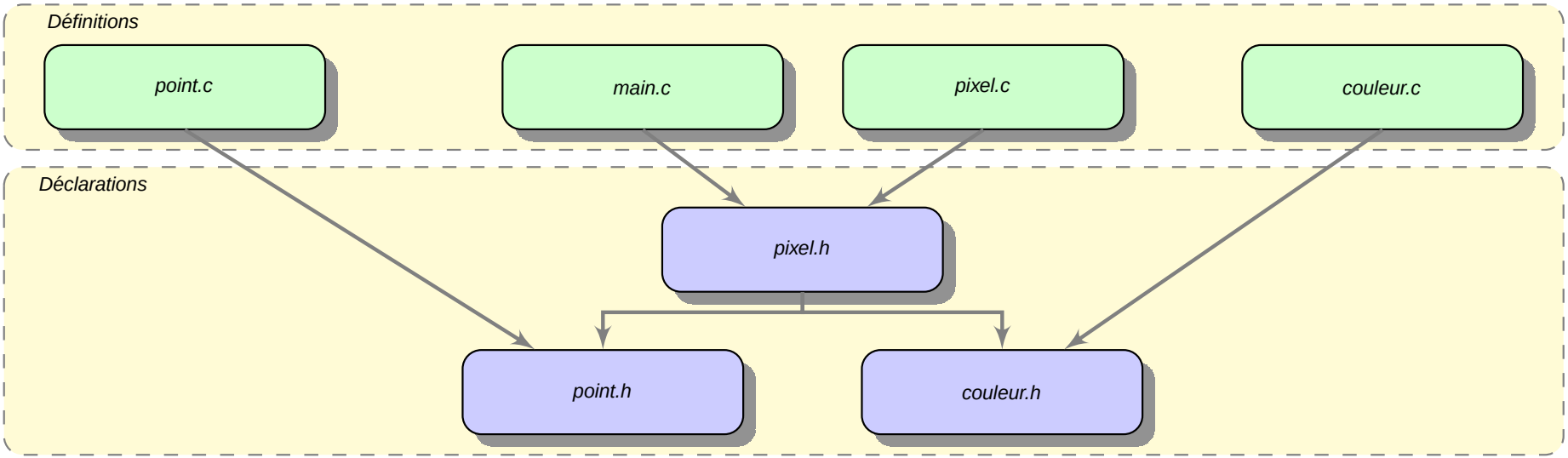
```c
/* Fichier main.c */
#include<stdio.h>
#include<stdlib.h>
#include"pixel.h"



int main(){
  pixel_t p1=creer_pixel(creer_point(1,3),creer_couleur(65535,0,0));
  afficher_pixel(p1);
  return EXIT_SUCCESS;
}
```

- On place toujours la fonction main dans un nouveau fichier.
- De cette façon on peut écrire plusieurs programmes en écrivant plusieurs main dans plusieurs fichiers.

Diagramme de dépendance



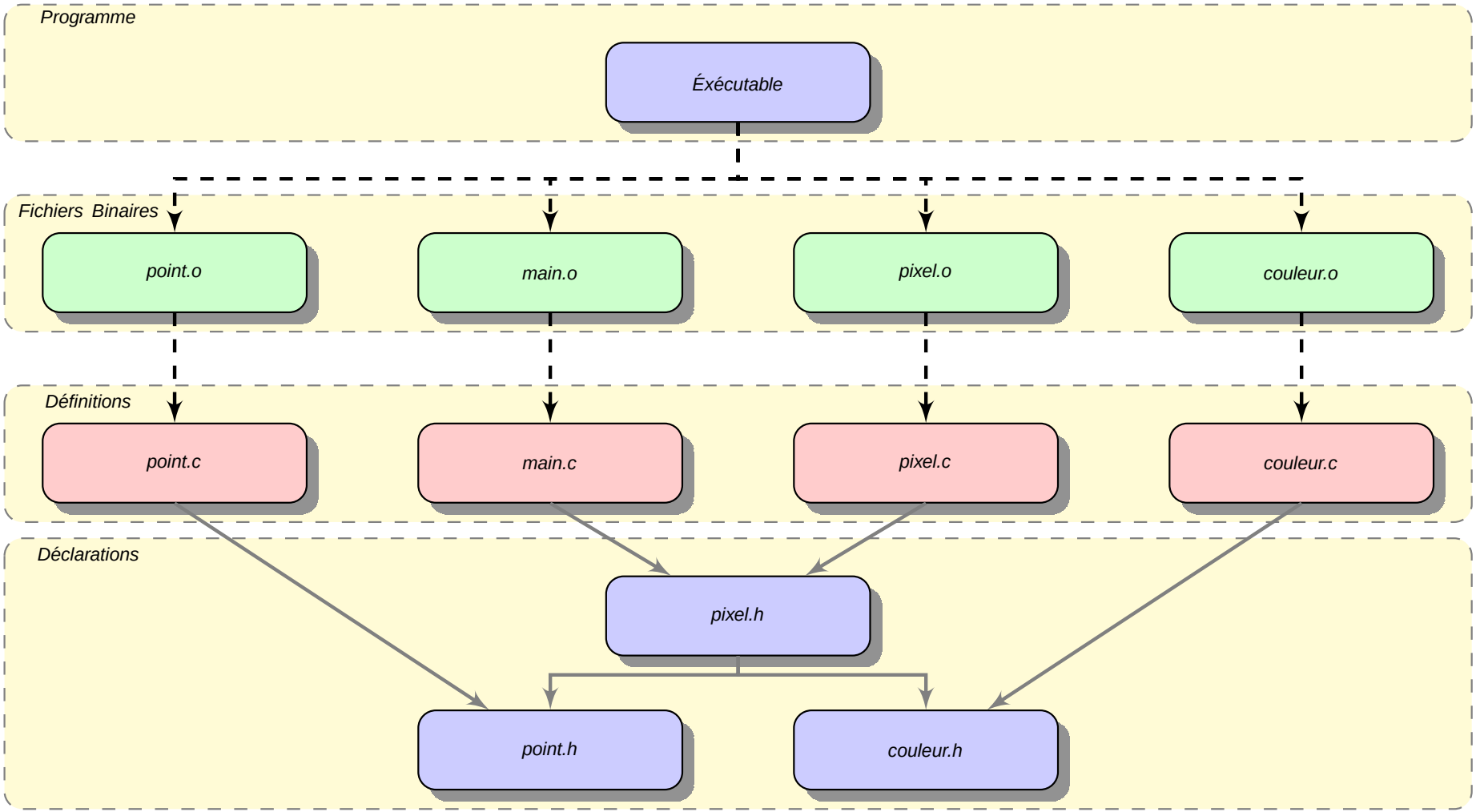
La compilation

Compilation séparée

Pour compiler le programme,

- on compile chaque fichier `.c` séparement
  Exemple : `gcc -c point.c`
- pour chaque fichier `.c` on obtient un fichier `.o`,
- toutes les fonctions définies dans le `.c` sont transformées en langage machine.

Diagramme de dépendance



Conflit sur les inclusions

Conflit sur les inclusions

- Certains fichiers `.h` sont inclus plusieurs fois.
- Au moment où l'on tente de compiler l'exécutable, il y aura des conflits d'inclusions.
- On va donc s'assurer que `.h` seront inclus au plus une fois.

Version finale des .h

```c
/* Fichier point.h */
#ifndef _POINT_H
#define _POINT_H
/* Definition de la structure */
struct point_s{
  unsigned int x;
  unsigned int y;
};
typedef struct point_s point_t;

/* Declaration des fonctions */

point_t creer_point(unsigned int x,unsigned int y);

int cmp_point(point_t p1,point_t p2);

void afficher_point(point_t p1);

void swap_point(point_t * p1,point_t * p2);

#endif
```

Le Makefile

La commande `make`

La commande make permet de construire des programmes en composant différents fichiers sources.
Cette commande se configure grâce à un fichier : le `Makefile`.

Le `Makefile`

Contient un ensemble d'informations permettant de construire l'exécutable.

- à partir de quels fichiers `.o` l'exécutable est-il construit ?
- comment convertir les fichiers `.c` en fichiers `.o` ? (on spécifie les options de compilation)

Si on a déjà construit le diagramme de dépendance :

- pour chaque flêche en pointillés dans le diagramme, on construit une **règle** dans le Makefile.

Le Makefile : syntaxe

Syntaxe d'une règle

`nom_regle: cible`$_1$ `cible`$_2$ `...` `cible`$_k$
`< TAB > commande à exécuter`

- les cibles sont des prérequis qui doivent être vérifés avec l'exécution de la commande qui se trouve à la ligne suivante,
- il est impératif de commencer la deuxième ligne par une tabulation.

Exemple

Pour compiler le fichier `point.c` et obtenir le fichier `point.o`

```
point.o: point.c
< TAB > gcc -c -Wall -ansi -pedantic -O3 point.c
```

Le Makefile : exemple naïf

```
1  all: point.o couleur.o pixel.o main.o
2    gcc -Wall -ansi -pedantic point.o couleur.o pixel.o main.o -o programme
3
4  point.o: point.c
5    gcc -c -Wall -ansi -pedantic point.c
6
7  couleur.o:couleur.c
8    gcc -c -Wall -ansi -pedantic couleur.c
9
10 pixel.o:pixel.c
11   gcc -c -Wall -ansi -pedantic pixel.c
12
13 main.o:main.c
14   gcc -c -Wall -ansi -pedantic main.c
```

Pour compiler, il suffit de taper `make` dans le repertoire où se trouve le `Makefile`

Améliorer son Makefile

Les variables

- il est possible de déclarer des variables dans le Makefile.

Le Makefile : exemple 2

```
1  # Indiquer le compilateur
2  CC=gcc
3
4  # Options du compilateur
5  CFLAGS  = -g -Wall -ansi -pedantic -O3
6
7  # Declarer vos propres variables
8  FICHIERS=point.o couleur.o pixel.o main.o
9  PROG=programme
10
11 all: $(FICHIERS)
12   gcc $(CFLAGS) $(FICHIERS) -o $(PROG)
13
14 point.o: point.c
15   gcc -c $(CFLAGS) point.c
16
17 couleur.o:couleur.c
18   gcc -c $(CFLAGS) couleur.c
19
20 pixel.o:pixel.c
21   gcc -c $(CFLAGS) pixel.c
22
23 main.o:main.c
24   gcc -c $(CFLAGS) main.c
```

Améliorer son Makefile

Makefile avancés

- Dans ce cours, on a vu les bases du Makefile.
- Ils peuvent en réalité être beaucoup plus complexes et bien mieux décrits.
- En TD/TP, on appliquera les notions d'aujourd'hui au programme des TD4 et TD5.