COURS C++

Programmation d'un type abstrait de données en C+

et compilation séparée

I.U.T. informatique – Semestre 2

Christine JULIEN

La programmation d'un type abstrait de données en C+ est basée sur l'incarnation d'un type abstrait de données. Elle s'appuie sur la spécification fonctionnelle du TAD et sur sa représentation physique (cf. Cours structures de données – chapitre 1).

Elle impose l'édition de deux fichiers distincts :

- *le premier est appelé **fichier de spécification** et a pour extension **.h**. Il contient la définition du type associé à la représentation physique d'une variable du TAD et la signature des différentes opérations (spécification des sous-programmes).*
- *Le deuxième est appelé **fichier d'implémentation** et a pour extension **.C**. Il contient le corps de chacune des opérations du TAD (définition des sous-programmes).*

*La conception d'une application multi-fichier nécessite l'utilisation d'un mécanisme de **compilation séparée** des différents modules afin de produire un programme exécutable. L'utilitaire **make** sous Unix permet de maintenir des applications multi-fichiers.*

TABLE DES MATIERES

Chapitre 1 : PROGRAMMATION D'UN TAD

1.1. Spécification fonctionnelle d'un TAD

La spécification fonctionnelle d'un TAD précise l'ensemble des opérations autorisées sur les représentants du type (syntaxe) ainsi que les propriétés de ces opérations (sémantique) (cf. Cours Structures de données - chapitre 1 - § 2).

1.1.1. Spécification fonctionnelle du TAD Couleur

Une couleur peut être codée en machine de plusieurs façons selon la palette des couleurs désirées.

Il existe plusieurs modèles de codage de la couleur. Les principaux sont le modèle RVB (Rouge, Vert, Bleu) utilisé pour les écrans cathodiques et le modèle CMJ (Cian, Magenta, Jaune) pour les imprimantes.

Nous avons choisi de spécifier par le TAD *Couleur* le type couleur dans le modèle RVB. Ce modèle définit 256 nuances pour chaque couleur de base rouge, verte et bleue. La couleur finale est obtenue par combinaison de ces 3 couleurs (soit plus de 16 millions de couleurs possibles : 256^3).

Par exemple, en supposant que R, V et B désignent respectivement les nuances de rouge, vert et bleu :

- le noir est codé par R=0, V=0, B=0
- le blanc est codé par R=255, V=255, B=255
- le jaune est codé par R=255, V=255, B=0
- le bleu profond est codé par R=0, V=0, B=128

1.1.1.1. Opérations du TAD Couleur

opérations

uneCouleur : Entier x Entier x Entier → Couleur
rouge : → Couleur
vert : → Couleur
bleu : → Couleur
nuanceRouge : Couleur → Entier
nuanceVert : Couleur → Entier
nuanceBleu : Couleur → Entier
valeurRVB : Couleur → Entier
modifierRouge : Couleur x Entier → Couleur
modifierVert : Couleur x Entier → Couleur
modifierBleu : Couleur x Entier → Couleur

préconditions

Pour r, v, b de type Entier

uneCouleur (r, v, b) **est défini ssi** 0≤r≤255 et 0≤ v≤255 et 0≤ r≤ 255

modifierRouge (c, r) **est défini ssi** 0≤r≤255

modifierVert (c, v) **est défini ssi** 0≤v≤255

modifierBleu (c, b) **est défini ssi** 0≤b≤255

1.1.1.2. Propriétés du TAD Couleur

Pour r, v, b, r1, v1, b1, r2, v2, b2 de type Entier

(P1) rouge = uneCouleur (255, 0, 0)
(P2) vert = uneCouleur (0, 255, 0)
(P3) bleu = uneCouleur (0, 0, 255)
informations caractérisant les constantes du type

(P2) nuanceRouge (uneCouleur (r, v, b)) = r
(P3) nuanceVert (uneCouleur (r, v, b)) = v
(P4) nuanceBleu (uneCouleur (r, v, b)) = b
(P5) valeurRVB (uneCouleur (r, v, b)) = $r*256^2 + v*256^1 + b*256^0$
informations caractérisant une couleur

(P6) modifierRouge (uneCouleur (r1, v, b), r2) = uneCouleur (r2, v, b)
(P7) modifierVert (uneCouleur (r, v1, b), v2) = uneCouleur (r, v2, b)
(P8) modifierBleu (uneCouleur (r, v, b1), b2) = uneCouleur (r, v, b2)

1.1.2. Spécification fonctionnelle du TAD Point

Nous reprenons ici la spécification du TAD Point vu en Cours de structures de données (chapitre 2).

1.1.2.1. Opérations du TAD Point

opérations

pointOrigine : → Point
unPoint : Réel x Réel x Couleur x Réel → Point
abscisse : Point → Réel
ordonnée : Point → Réel
couleur : Point → Couleur
taille : Point → Réel
modifierCouleur : Point x Couleur → Point
modifierTaille : Point x Réel → Point
translater : Point x Réel x Réel → Point
mettreAEchelle : Point x Réel → Point

1.1.2.2. Propriétés du TAD Point

Pour c, c1, c2 de type Couleur ; x, y, h, t, t1, t2, tx, ty de type Réel

(P1) pointOrigine = unPoint (0.0, 0.0, uneCouleur (0, 0, 0), 1.0)

(P2) abscisse (unPoint (x, y, c, t)) = x
(P3) ordonnée (unPoint (x, y, c, t)) = y
(P4) couleur (unPoint (x, y, c, t)) = c
(P5) taille (unPoint (x, y, c, t)) = t

(P6) modifierCouleur (unPoint (x, y, c1, t), c2) = unPoint (x, y, c2, t)

(P7) modifierTaille (unPoint (x, y, c, t1), t2) = unPoint (x, y, c, t2)

(P8) translater (unPoint (x, y, c, t), tx, ty) = unPoint (x+tx, y+ty, c, t)

(P9) mettreAEchelle (unPoint (x, y, c, t), h) = unPoint (x*h, y*h, c, t)

1.2. Spécification d'un TAD en C+

La spécification d'un TAD en C+ est éditée dans un fichier d'extension .h. Elle fournit la représentation du type associé à la représentation physique du TAD et la signature des fonctions C+ associées aux opérations du TAD.

Pour écrire le fichier de spécification, il faut donc s'appuyer sur la représentation physique et sur les opérations du TAD.

La représentation du type associée à la représentation physique est toujours introduite par la définition d'un enregistrement (*struct*) même si elle n'est constituée que d'un seul champ. Une telle structure garantit ainsi la possibilité d'utiliser l'affectation entre deux variables de même type.

Remarque :

Le préprocesseur C++ fournit des instructions qui contrôlent les portions de texte à compiler. C'est ce qu'on appelle la ***compilation conditionnelle***. Des décisions peuvent être prises en fonction de la définition d'un symbole.

L'instruction de contrôle a la syntaxe suivante où *symbole* désigne un identificateur de symbole :

```
#ifndef symbole
#define symbole
        portion de programme
#endif
```

Cette commande a pour effet d'inclure dans la compilation la portion de programme comprise entre *#ifndef* et *#endif* si l'identificateur n'a pas été défini en tant que symbole.

Il est intéressant d'insérer ce symbole dans les fichiers de spécification qui ont l'extension *.h*. En effet, cette directive assure l'unicité du code contenu dans un fichier et par conséquent évite des erreurs à la compilation (*twice defined*).

Par convention, pour un fichier .h décrivant la spécification d'un TAD *T*, ce symbole aura pour identificateur *_T_H*.

1.2.1. Spécification du TAD Couleur en C+ : fichier *couleur.h*

Nous avons choisi de représenter le type *Couleur* par un enregistrement à trois champs r, v et b désignant respectivement la nuance rouge, verte et bleue de la couleur :

```
#ifndef _COULEUR_H
#define _COULEUR_H

#include "/usr/local/public/BIBLIOC++/entreeSortie.h"

// définition du type couleur
struct Couleur
{
    int r;          // nuance de rouge
    int v;          // nuance de vert
    int b;          // nuance de bleu
};

// signatures des opérations sur une couleur

// crée une couleur à partir d'une nuance de rouge r, d'une nuance de vert v
// et d'une nuance de bleu b
// nécessite 0<=r<=255
// nécessite 0<=v<=255
```

```
// nécessite 0<=b<=255
Couleur uneCouleur(const int r, const int v, const int b) throw (Chaine);

// constante désignant la couleur rouge
Couleur rouge();

// constante désignant la couleur verte
Couleur vert();

// constante désignant la couleur bleue
Couleur bleu();

// fournit la nuance de rouge d'une couleur c
int nuanceRouge(const Couleur &c);

// fournit la nuance de vert d'une couleur c
int nuanceVert(const Couleur &c);

// fournit la nuance de bleu d'une couleur c
int nuanceBleu(const Couleur &c);

// fournit la valeur RVB d'une couleur c
long valeurRVB(const Couleur &c);

// modifie une couleur c à partir d'une nouvelle nuance de rouge r
// nécessite 0<=r<=255
void modifierRouge(Couleur &c, const int r) throw (Chaine);

// modifie une couleur c à partir d'une nouvelle nuance de vert v
// nécessite 0<=v<=255
void modifierVert(Couleur &c, const int v) throw (Chaine);

// modifie une couleur c à partir d'une nouvelle nuance de bleu b
// nécessite 0<=b<=255
void modifierBleu(Couleur & c, const int b) throw (Chaine);

#endif
```

Remarque :

Les préconditions non vérifiée pour les opérations partielles se traduisent par une exception dans le code.

1.2.2. Spécification du TAD Point en C++ : fichier *point.h*

Nous avons choisi de représenter le type *Point* par un enregistrement à quatre champs x, y, c et t désignant respectivement l'abscisse, l'ordonnée, la couleur et la taille du point :

```
#ifndef _POINT_H
#define _POINT_H

#include "couleur.h"

// définition du type Point
struct Point
{
    float x;        // abscisse
    float y;        // ordonnee
    Couleur c;      // couleur
    float t;        // taille
};

// signatures des opérations sur un point

// constante désignant le point origine
Point pointOrigine();
```

```
// construit un point à partir d'une abscisse x, d'une ordonnée y, d'une couleur c
// et d'une taille t
Point unPoint(const float x, const float y, const Couleur &c, const float t);

// fournit l'abscisse d'un point p
float abscisse(const Point &p);

// fournit l'ordonnée d'un point
float ordonnee(const Point &p);

// fournit la couleur d'un point p
Couleur couleur(const Point &p ;

// fournit la taille d'un point p
float taille(const Point &p);

// modifie la couleur d'un point p à partir d'une nouvelle couleur c
void modifierCouleur(Point &p, const Couleur &c);

// modifie la taille d'un point à partir d'une nouvelle taille t
void modifierTaille(Point &p, const float t);

// translate un point p à partir de la valeur de translation sur l'axe de
// l'abscisse tx et de la valeur de translation sur l'axe de l'ordonnée ty
void translater(Point &p, const float tx, const float ty);

// modifie l'échelle d'un point p à partir de sa nouvelle échelle h
void mettreAEchelle(Point &p, const float h);

#endif
```

Remarque :

Le compilateur doit connaître la représentation interne d'une couleur. Pour cette raison, il est nécessaire d'inclure dans *point.h* le fichier *couleur.h*.

1.3. Implémentation d'un TAD en C+

L'implémentation d'un TAD en C+ est éditée dans un fichier d'extension .C. Elle fournit le corps des différentes fonctions C+ déclarées dans le fichier de spécification.

Le fichier d'implémentation d'un TAD doit nécessairement contenir la directive d'inclusion du fichier de spécification associé.

L'implémentation s'appuie sur les propriétés de la spécification fonctionnelle du TAD.

1.3.1. Implémentation du TAD Couleur en C+ : fichier *couleur.C*

```
#include "couleur.h"
Couleur uneCouleur(const int r, const int v, const int b) throw (Chaine)
{
    Couleur c;
    if (r < 0 || r > 255)
    {
        throw uneChaine("ERREUR NUANCE DE ROUGE");
    }
    if (v < 0 || v > 255)
    {
        throw uneChaine("ERREUR NUANCE DE VERT");
    }
    if (b < 0 || b > 255)
    {
        throw uneChaine("ERREUR NUANCE DE BLEU");
    }
    c.r = r;
    c.v = v;
    c.b = b;
    return (c);
```

```
}
Couleur rouge()
{
    Couleur c;
    c.r = 255;
    c.v = 0;
    c.b = 0;
    return (c);
}
Couleur vert()
{
    Couleur c;
    c.r = 0;
    c.v = 255;
    c.b = 0;
    return (c);
}
Couleur bleu()
{
    Couleur c;
    c.r = 0;
    c.v = 0;
    c.b = 255;
    return (c);
}
int nuanceRouge(const Couleur &c)
{
    return (c.r);
}
int nuanceVert(const Couleur &c)
{
    return (c.v);
}
int nuanceBleu(const Couleur &c)
{
    return (c.b);
}
long valeurRVB(const Couleur &c)
{
    return (long (c.r * 256 * 256) + long (c.v * 256) + long (c.b));
}
void modifierRouge(Couleur &c, const int r) throw (Chaine)
{
    if (r < 0 || r > 255)
    {
        throw uneChaine("ERREUR NUANCE DE ROUGE");
    }
    c.r = r;
}
void modifierVert(Couleur &c, const int v) throw (Chaine)
{
    if (v < 0 || v > 255)
    {
        throw uneChaine("ERREUR NUANCE DE VERT");
    }
    c.v = v;
}
void modifierBleu(Couleur &c, const int b) throw (Chaine)
{
    if (b < 0 || b > 255)
    {
        throw uneChaine("ERREUR NUANCE DE BLEU");
    }
    c.b = b;
}
```

1.3.2. Implémentation du TAD Point en C+ : fichier *point.C*

```
#include "point.h"

Point pointOrigine()
{
    Point p;
    p.x = 0.0;
    p.y = 0.0;
    p.t = 1.0;
    p.c = uneCouleur(0, 0, 0);
    return (p);
}

Point unPoint(const float x, const float y, const Couleur &c, const float t)
{
    Point p;
    p.x = x;
    p.y = y;
    p.c = c;
    p.t = t;
    return (p);
}

float abscisse(const Point &p)
{
    return (p.x);
}

float ordonnee(const Point &p)
{
    return (p.y);
}

Couleur couleur(const Point &p)
{
    return (p.c);
}

float taille(const Point &p)
{
    return (p.t);
}

void modifierCouleur(Point &p, const Couleur &c)
{
    p.c = c;
}

void modifierTaille(Point & p, const float t)
{
    p.t = t;
}

void translater(Point &p, float tx, const float ty)
{
    p.x = p.x + tx;
    p.y = p.y + ty;
}

void mettreAEchelle(Point & p, const float h)
{
    p.x = p.x * h;
    p.y = p.y * h;
}
```

1.4. Utilisation d'un TAD en C+

Pour une application donnée, le programmeur peut faire référence à un ou plusieurs types abstraits : il suffit d'inclure dans le source les spécifications (fichiers .h) des types abstraits utilisés.

Les seules opérations autorisées sur un type abstrait de données sont celles fournies dans le fichier de spécification. Le programmeur n'a pas besoin de connaître la représentation physique du type.

Exemple : programme *milieuSegment.C*

```
//-------------------------------------------------------------------------//
// nom du programme : milieuSegment.C                                      //
// rôle : lit au clavier les coordonnées de deux points définissant        //
//        un segment puis calcule et affiche le point milieu de ce segment //
//-------------------------------------------------------------------------//

#include "point.h"
#include "/usr/local/public/BIBLIOC++/entreeSortie.h"
#include "/usr/local/public/BIBLIOC++/chaine.h"

// définition de l'enregistrement Segment
struct Segment
{
    Point org;      // origine du segment
    Point ext;      // extrémité du segment
};

// lit un point noir par saisie de ses coordonnées
void lirePoint(Point & p);

// retourne le point milieu d'un segment s
Point pointMilieu(const Segment &s);

// affiche les caractéristiques d'un point p
void ecrirePoint(const Point p);


// programme principal
int main()
{
    Segment s;      // segment dont on veut calculer le milieu

    // lire le point origine du segment
    lirePoint(s.org);
    // lire le point extrémité du segment
    lirePoint(s.ext);
    // calculer et afficher les caractéristiques du point milieu
    // du segment
    ecrireNL("caracteristiques du point milieu du segment : ";
    // calculer et afficher les caractéristiques du point milieu
    // du segment
    ecrirePoint(pointMilieu(s));
}

// définition des sous-programmes
void lirePoint(Point &p)
{
    float x;        // valeur de l'abscisse du point
    float y;        // valeur de l'ordonnée du point

    ecrire(uneChaine("entrer l'abscisse du point : "));
    lire(x);
    ecrire(uneChaine ("entrer l'ordonnee du point : "));
    lire(y);
    p = unPoint(x, y, uneCouleur(0, 0, 0), 1);
}
```

```
Point pointMilieu(const Segment &s)
{
    Point p;
    p = pointOrigine();
    translater
        ( p,
        (abscisse(s.org) + abscisse(s.ext)) / 2,
        (ordonnee(s.org) + ordonnee(s.ext)) / 2);
    return (p);

    // une autre version aurait consisté à utiliser le constructeur :
    // return
    //    (unPoint (abscisse (s.org) + abscisse (s.ext)) / 2,
    //    ordonnee (s.org) + ordonnee (s.ext)) / 2,
    //    uneCouleur (0, 0, 0),
    //    1) ;
}

void ecrirePoint(const Point &p)
{
    ecrire(uneChaine("abscisse : "));
    ecrireNL(abscisse (p));
    ecrire(uneChaine("ordonnee : "));
    ecrireNL(ordonnee(p));
    ecrire(uneChaine("couleur RVB : "));
    ecrireNL(valeurRVB(couleur (p)));
    ecrire(uneChaine("taille : "));
    ecrireNL(taille(p) ;
}
```

Remarque : Le fait de représenter un type par un enregistrement confère au type l'opération d'affectation. Par contre les opérations de comparaison (== et !=) sont impossibles (cf. cours C++ 1ère partie sur les enregistrements).

Chapitre 2 : COMPILATION SEPAREE EN C+

2.1. Concepts de base de la compilation séparée

Une application utilisant des TAD est conçue comme une collection de modules. Chaque module mémorisé dans un fichier constitue une *unité de programme* (contrairement au cas où le programme est mémorisé dans un fichier unique).

Chaque unité de programme constitue une *unité de compilation* qui sera soumise au compilateur C++ pour générer un fichier objet.

En règle générale, les avantages de la compilation séparée sont multiples :

- chaque unité de compilation peut être éditée et compilée séparément, d'où un gain de temps dans la phase de développement de l'application (c'est le cas des TAD),
- dans le cadre d'un projet, les diverses unités de compilation peuvent être développées par des personnes différentes (par exemple *entreeSortie* pour faire des entrées/sorties),
- les différentes unités de compilations peuvent être codées dans des langages différents, sous réserve que les modules objets générés soient compatibles (par exemple appels dans un programme C++ de sous-programmes écrits en assembleur),
- certaines unités de compilation peuvent correspondre à des bibliothèques de sous-programmes fournis par le compilateur (par exemple *math* pour manipuler des fonctions mathématiques provenant de la bibliothèque standard C++).

Le programme objet exécutable est obtenu en assemblant les différents modules objets issus de la compilation séparée des différentes unités de compilation. L'utilitaire réalisant cette fonction est appelé *éditeur de liens*.

Ce mécanisme de compilation séparée n'est pas propre au langage C++. Il peut s'adapter à une application écrite dans n'importe quel langage de programmation (COBOL par exemple).

2.1.1. Processus de compilation séparée en C++

Pour chaque unité de compilation correspondant à un fichier *fic.C* de l'application, on produit le module objet correspondant *fic.o*, par la commande :

```
g++ -c fic.C
```

Remarque :

A la commande de compilation peuvent être associées des options.

Par exemple, l'option *-g* doit être utilisée pour pouvoir utiliser le débogueur *ddd*.

A partir des différents modules objets résultant d'un compilation, on produit le module objet exécutable *f.exe* par la commande d'édition de liens :

```
g++ -o f.exe fic1.o fic2.o … ficn.o
```

Remarque :

A la commande d'édition de liens peuvent être associées des options. Par exemple, l'option *–l<bibliothèque>* doit être utilisée lorsque l'application utilise des sous-programmes externes provenant de bibliothèques autres que la bibliothèque standard C++ ou bien l'option *-g* pour pouvoir utiliser le débogueur *ddd*.

2.1.2. Mise en œuvre au travers de l'application milieuSegment.exe

Cette application fait intervenir plusieurs fichiers d'extension .C :

- *couleur.C* (implémentation du TAD Couleur).
- *point.C* (implémentation du TAD Point).
- *milieuSegment.C* (contient le programme principal *main()*). Ce fichier utilise "*entreeSortie.h*" et "*chaine.h*" faisant partie de la bibliothèque *libTPC+*.

Avant de produire l'exécutable *milieuSegment.exe*, il est nécessaire de compiler séparément les fichiers d'extension .C. L'ordre de compilation des fichiers n'a aucune importance.

```
g++ -c couleur.C –g                    -- génération du fichier couleur.o
g++ -c point.C -g                      -- génération du fichier point.o
g++ -c milieuSegment.C -g              -- génération du fichier milieuSegment.o
```

Le fichier exécutable est produit ensuite par la commande d'édition de liens :

```
-- génération du fichier exécutable milieuSegment.exe
g++ -o milieuSegment.exe couleur.o point.o milieuSegment.o –lTPC+ -g
```

Remarque :

Si l'édition de liens produit des erreurs de symboles, il est fort probable que l'utilisateur a oublié d'inclure tous les fichiers objets de l'application dans la commande.

L'exécution de l'application s'obtient par :

```
./milieuSegment.exe
```

Remarque :

Lorsqu'on modifie un fichier partagé, c'est le cas des fichiers d'extension *.h*, on doit mettre à jour toutes les unités de compilation où ce fichier est référencé par une commande *#include*. Ainsi, si l'utilisateur modifie le fichier *point.h*, une nouvelle compilation des fichiers *point.C* et *milieuSegment.C* s'avère nécessaire avant de faire l'édition de liens.

2.2. L'utilitaire make

2.2.1. Fonctionnalités

Lorsqu'on développe une application multi-fichier, une simple modification peut conduire à la recompilation de nombreux fichiers. Une compilation manuelle impose de connaître les dépendances entre fichiers, ce qui est d'autant plus fastidieux.

L'utilitaire ***make*** sous Unix permet de créer et de maintenir des applications multi-fichiers. Il permet notamment d'automatiser le processus de compilation séparée.

À partir de relations de dépendance entre fichiers, de commandes de génération de fichiers et des dates de dernière modification des fichiers, cet utilitaire réactualise automatiquement les fichiers qui ont besoin de l'être, ou les crée s'ils n'existent pas. Tout ceci afin de produire une version à jour du programme exécutable.

L'utilisation de la commande ***make*** suppose l'existence d'un fichier dont le nom par défaut est ***Makefile*** (ou ***makefile***). Toutefois le nom de ce fichier peut être choisi par l'utilisateur (cf. § 2.2.3.).

2.2.2. Le fichier *Makefile*

Ce fichier contient la description des dépendances entre fichiers intervenant dans l'application à maintenir et les actions à entreprendre pour remettre à jour l'application dans le cas où des fichiers sont modifiés (compilation, édition de liens).

2.2.2.1. Les enregistrements du fichier *Makefile*

De manière générale, ce fichier est décrit par une suite d'enregistrements de la forme :

```
<fichier cible> : <liste des fichiers dont dépend le fichier cible>
<tab>       <commande à exécuter pour réactualiser le fichier>
```

où :

<*fichier cible*> est le nom du fichier à générer,

<*liste des fichiers*> est la liste des fichiers prérequis qui seront utilisés pour générer le fichier cible ; chaque nom de fichier doit être précédé du symbole espace,

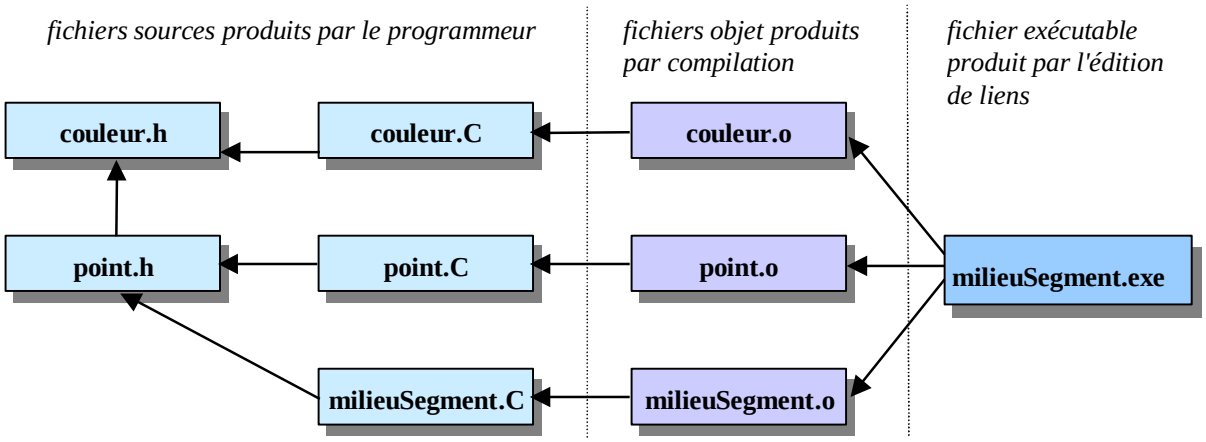
<*commande*> est la commande mise en œuvre pour générer le fichier cible.

Remarques :

- La deuxième ligne de chaque enregistrement, si elle existe, doit obligatoirement être précédée d'un caractère de tabulation,
- Le premier enregistrement doit correspondre à la commande d'édition de liens.

2.2.2.2. Dépendances entre fichiers

Les dépendances entre fichiers peuvent être facilement schématisées par une hiérarchie entre les différents fichiers intervenant dans l'application comme le montre la figure suivante :



Par exemple :

- le fichier objet *milieuSegment.o* dépend des fichiers *milieuSegment.C*, *point.h* et *couleur.h*, ce qui se traduit par :

```
milieuSegment.o : milieuSegment.C point.h couleur.h
```

- le fichier exécutable dépend des fichiers objet *milieuSegment.o*, *point.o* et *couleur.o*, ca qui se traduit par :

```
milieuSegment.exe : milieuSegment.o point.o couleur.o
```

On obtient ainsi le fichier *Makefile* :

```
# edition de liens
milieuSegment.exe : milieuSegment.o point.o couleur.o
    g++ -o milieuSegment.exe milieuSegment.o point.o couleur.o –lTPC+ -g

# compilation du module couleur
couleur.o : couleur.h couleur.C
    g++ -c couleur.C –g

# compilation du module point
point.o : point.h couleur.h point.C
    g++ -c point.C -g

# compilation du module milieuSegment
milieuSegment.o : couleur.h point.h milieuSegment.C
    g++ -c milieuSegment.C -g
```

2.2.2.3. Utilisation de variables

Le programmeur peut définir des variables de type chaîne de caractères et les utiliser dans la définition des relations de dépendance entre fichiers et dans les commandes de génération de fichiers cibles.

La définition d'une variable est la suivante :

```
<variable>=<chaîne de caractères>
```

où :

<*variable*> est le nom symbolique de la variable,

<*chaîne de caractères*> est une suite de caractères éventuellement vide.

Pour désigner le contenu d'une variable, il faut placer le symbole entre parenthèses précédé du caractère $:

```
$(variable)
```

Par exemple :

```
# définition de la variable contenant la liste des fichiers objet
BIN=milieuSegment.o point.o couleur.o

# Edition de liens
milieuSegment.exe : $(BIN)
    g++ -o milieuSegment.exe $(BIN) –lTPC+ -g

# compilation du module couleur
couleur.o : couleur.h couleur.C
    g++ -c couleur.C –g

# compilation du module point
point.o : point.h couleur.h point.C
    g++ -c point.C -g

# compilation du module milieuSegment
milieuSegment.o : couleur.h point.h milieuSegment.C
    g++ -c milieuSegment.C -g
```

2.2.2.4. Règles de génération par défaut

Certaines générations de fichiers se font toujours de la même façon. C'est notamment le cas de la compilation des programmes sources en modules objet. Dans ce cas, il n'est pas nécessaire de fournir les commandes de génération, car l'utilitaire *make* dispose d'une règle par défaut qui lui indique comment passer d'un fichier *.C* à un fichier *.o*.

En outre**,** l'utilitaire *make* utilise des variables prédéfinies dans les règles de génération par défaut. Certaines de ces variables servent à désigner les compilateurs et les options de compilation. Elles peuvent être modifiées par le programmeur dans le fichier *Makefile*.

```
CXX : contient le nom du compilateur C++
CXXFLAGS : contient la liste des options utilisées à la compilation
```

Par exemple :

```
# définition de la variable contenant la liste des fichiers objet
BIN=milieuSegment.o point.o couleur.o

# définition de la variable contenant le nom du compilateur C++
CXX=g++

# définition de la variable contenant les options de compilation
CXXFLAGS=-g

# edition de liens
milieuSegment.exe : $(BIN)
    $(CXX) -o milieuSegment.exe $(BIN) -lTPC+ -g

# compilation du module couleur
couleur.o : couleur.h couleur.C

# compilation du module point
point.o : point.h couleur.h point.C

# compilation du module milieuSegment
milieuSegment.o : couleur.h point.h milieuSegment.C
```

2.2.3. Utilisation du Makefile

L'édition d'un fichier *Makefile* est faite par l'intermédiaire d'un éditeur de texte.

La syntaxe d'appel de l'utilitaire make est :

```
make <options>
```

où :

<options> est un champ optionnel. Parmi les principales options, nous trouvons :

-f <fichier> : désigne un fichier de directives autre que *makefile* ou *Makefile*

-p : affiche la liste des variables prédéfinies et les règles de génération par défaut.