Université Claude Bernard Lyon 1

LIFAPSD – Algorithmique, Programmation et Structures de données

Nicolas Pronost

LIRIS

Chapitre 5
Type de données abstraits et programmation séparée

Les grands problèmes du génie logiciel

- Maîtriser la complexité
- Réutiliser du code existant
- Capitaliser les efforts
- Détecter les erreurs le plus tôt possible

Maîtriser la complexité

- Diviser pour régner = découper l'application en modules
  - aussi indépendants que possible
  - faciles à comprendre et à programmer individuellement
  - testables individuellement puis tous ensemble
- La difficulté est de trouver le bon découpage
  - trop découpé: difficile de s'y retrouver
  - pas assez découpé: dur à maintenir, développer et tester

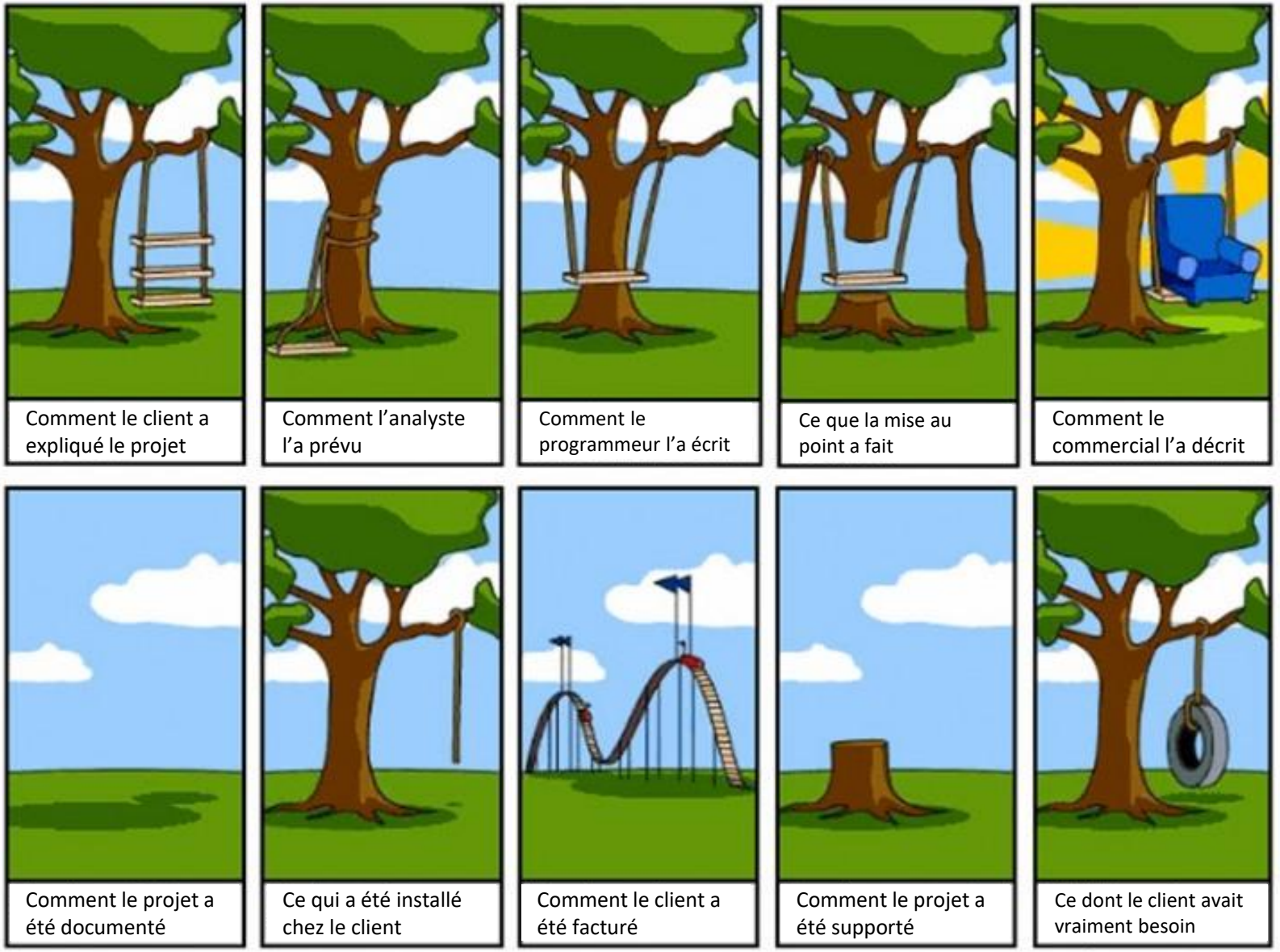
Réutiliser du code existant

- Résoudre une fois pour toutes une famille de problèmes
  - ex. somme, produit de nombres complexes, produit vectoriel
  - ex. dessin de formes graphiques en 2D ou en 3D, com. réseau
- Bibliothèque d'usage général
  - le calcul avec des nombres complexes est un problème général, non lié à une application particulière
  - de même pour le dessin 2D (ex. jeu vidéo, simulation scientifique, GUI)
  - il y a beaucoup de bibliothèques pour beaucoup d'usages, surtout en C/C++
- Design pattern
  - conception classique (ex. ensemble de classes) pour répondre à un problème classique
  - ex. *Listener*, *Singleton*, *Factory*, *Server-Client*, ...
- Chercher, connaître et se mettre à jour sur l'existant

Capitaliser les efforts

- Quel est l'intérêt d'une fonction s'il faut lire toutes ses instructions pour comprendre son effet?
- Comment développer dans un projet de plusieurs millions de lignes de codes s'il faut tout mémoriser?
- Notion d'abstraction
  - créer des fonctions offrant un service clair, compréhensible et utilisable sans avoir à lire le code
  - permet de traiter des problèmes de haut niveau, en s'affranchissant des problèmes de niveau inférieur déjà résolus
- Isolation entre modules
  - si l'interface d'un module est juste, la correction du code d'une fonction n'a pas d'influence sur le code qui l'utilise

Détecter les erreurs le plus tôt possible



Détecter les erreurs le plus tôt possible

- Plus une erreur est détectée tard, plus elle coûte cher
- Il faut se doter d'une méthode de développement par étapes
- La fin de chaque étape doit être actée et évaluée
- Importance de l'étape de spécification de chaque module
  - erreur (bug) = défaut de réalisation des spécifications
- « Ecrivez ce que vous faîtes, faîtes ce qui est écrit, prouvez que vous le faîtes »
- Tester, tester et encore tester
  - Tests unitaires: chaque module individuellement
  - Tests d'intégration: les modules entre eux
- *Plus sur la conception logicielle en LIFAP4*

Les types de données abstraits (TDA)

- Nés de ces préoccupations de génie logiciel
- Définition: un TDA est un ensemble de données organisé de sorte que les spécifications des objets et des opérations sur ces objets soient séparés de la représentation interne des objets et de la mise en œuvre des opérations
- Un TDA est donc
  - un type de données
  - doté d'un ensemble d'opérations
  - dont les détails d'implémentation restent cachés (abstraction)
- Les TDA sont les briques de base d'une conception modulaire

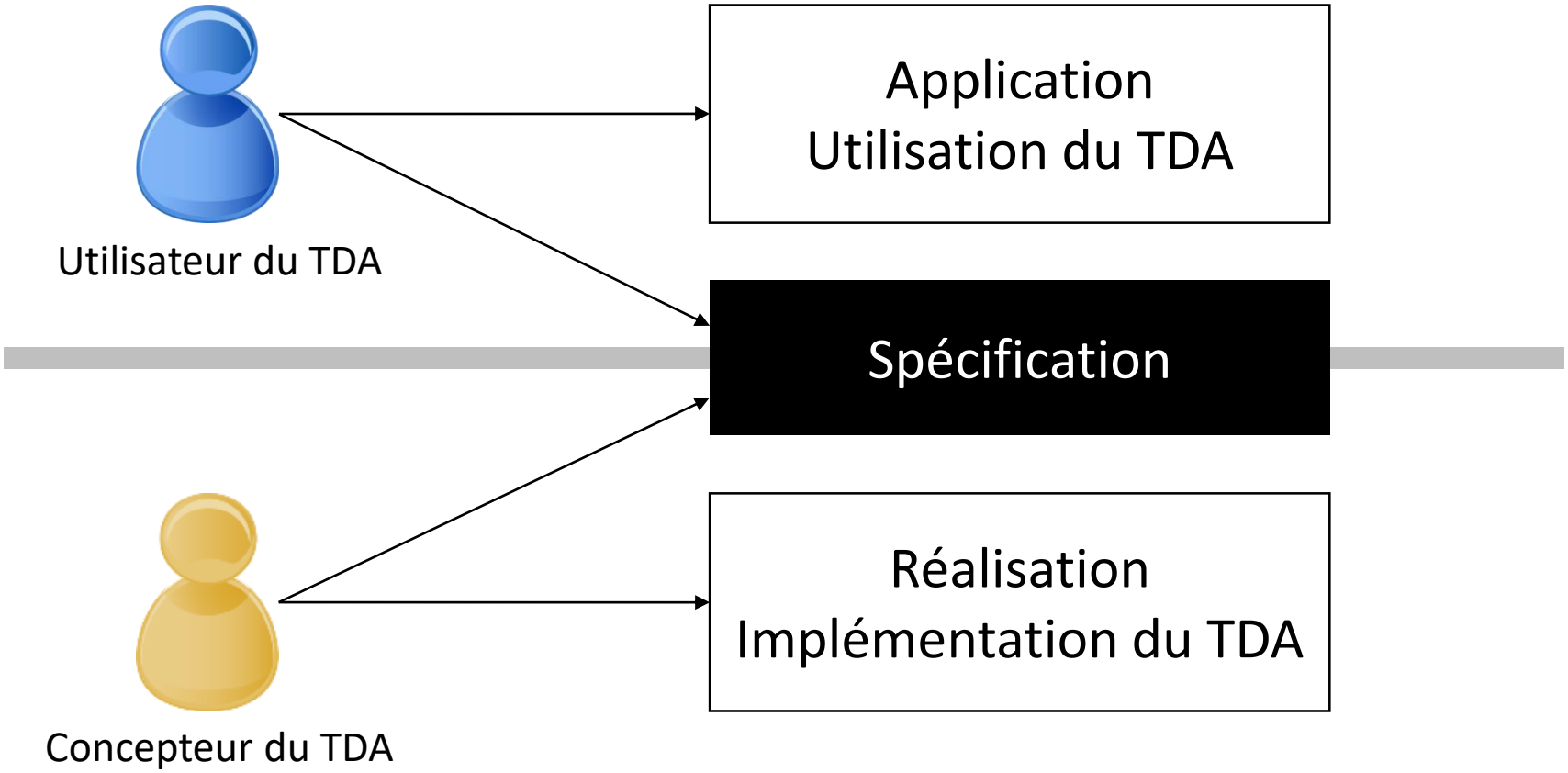
Les types de données abstraits (TDA)

- Exemple: le type **int** en C++
  - fourni avec les opérations +, -, *, /, %
  - il n'est pas nécessaire de connaître la représentation interne des **int** ni les algorithmes des opérations pour pouvoir les utiliser
  - il faut cependant connaître les pré- et post-conditions de ces opérations (ex. pour éviter les dépassements de capacité)
- On peut construire des TDA plus complexes à partir des types primitifs
  - créer un type (ex. une structure ou une classe) dont la représentation interne est cachée
  - offrir les opérations de haut niveau nécessaires en spécifiant bien les pré- et post-conditions, mais en cachant le code

Exemple de TDA

- Objectif: concevoir une bibliothèque pour les opérations sur les nombres complexes
- Vision interne (concepteur)
  - Comment représenter un nombre complexe?
    - forme cartésienne $a + bi$ ou exponentielle $\rho e^{i\theta}$?
  - Comment le TDA sera-t-il utilisé? Quoi cacher, quoi exposer?
    - module, conjugué, conversion de forme...?
  - Comment implémenter efficacement les opérations?
    - une forme est-elle plus efficace qu'une autre pour multiplier deux nombres complexes?
- Vision externe (utilisateur)
  - Quels sont les services offerts par ce TDA?
  - Ce TDA répond-il à mes besoins?
  - Ce TDA est-il suffisamment performant?

Les deux points de vue d'un TDA



TDA et module

- Un TDA est décrit dans le cadre d'un module
- Un module regroupe des définitions
  - de constantes
  - de variables globales
  - de types
  - de procédures et de fonctions qui permettent de manipuler ces types
- Cet ensemble de définitions forme un tout cohérent
- On sépare interface (spécification, déclarations, entêtes) et implémentation (définitions, code)
- On sépare les modules et le programme principal

Module: norme algorithmique

Module nom_module {*rôle du module*}

- **Importer:**
  - Déclaration des modules extérieurs utilisés dans l'interface du module
- **Exporter:**
  - Déclaration des types, procédures, fonctions, constantes, variables globales offerts par le module
- **Implémentation:**
  - Déclaration de modules extérieurs utilisés dans l'implémentation du module
  - Définition des types, procédures, fonctions, constantes, variables globales offerts par le module
  - Définition éventuelle de types, procédures, fonctions, constantes, variables globales internes au module (utiles pour l'implémentation du module mais non exportés)
- **Initialisation:**
  - Actions à exécuter au début du programme pour garantir une utilisation correcte du module

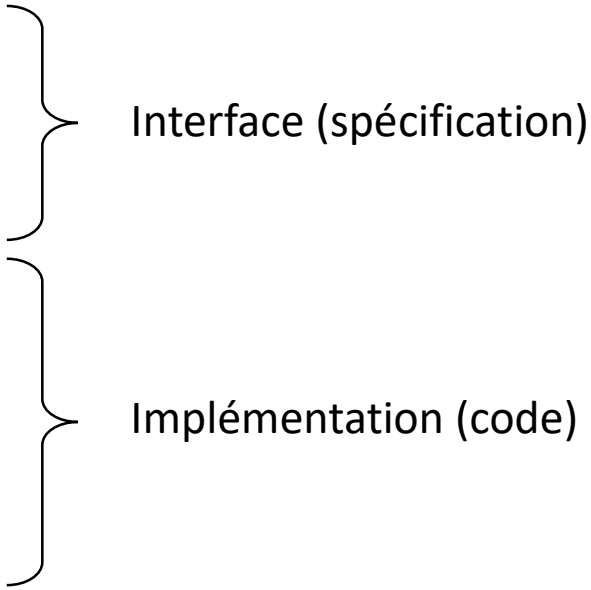
FinModule nom_module

Module: norme algorithmique

Module nom_module {*rôle du module*}

- **Importer:**
  - Déclaration ...
- **Exporter:**
  - Déclaration ...
- **Implémentation:**
  - Déclaration ...
  - Définition ...
  - Définition ...
- **Initialisation:**
  - Actions ...

FinModule nom_module



Exemple pour un tableau dynamique

Module TableauDynamique {un tableau de taille variable}

- **Importer:**
  - Module Element
- **Exporter:**
  - Type TableauDynamique
    - Constructeur TableauDynamique()
      - Postconditions : réservation d'un tableau de taille 1 Element sur le tas
    - Destructeur ~TableauDynamique()
      - Postconditions : libération de la mémoire utilisée sur le tas
    - Procédure ajoutElement (e: Element)
      - Postcondition : une copie de e est insérée à la fin du tableau, extension de l'espace mémoire alloué au tableau si nécessaire
      - Paramètre en mode donnée : e
    - Etc. : suppression élément, affichage, tri, écriture sur fichier, ...

Exemple pour un tableau dynamique

- **Implémentation:**
  - Type TableauDynamique = Classe
    adressePremierElt : pointeur sur Element
    capacite : entier
    tailleUtilisee : entier

    - Constructeur TableauDynamique()
      Début
        adressePremierElt ← **réserve** tableau [1..1] de Element
        capacite ← 1
        tailleUtilisee ← 0
      Fin

    - Destructeur ~TableauDynamique()
      Début
        libère adressePremierElt
        capacite ← 0
        tailleUtilisee ← 0
      Fin

    - Etc.

    Fin classe

FinModule TableauDynamique

Exemple de programme utilisateur

```
Importer:
   module TableauDynamique
   module Element

Variables:
  monTab : TableauDynamique
  monElt : Element
  finSaisie : booléen

Début
  finSaisie ← faux
  Répéter
     monElt.saisieClavier()
     monTab.ajoutElement(monElt)
     afficher("Voulez-vous saisir un élément supplémentaire?")
     lire(finSaisie)
  jusqu'à ce que (finSaisie = vrai)

  monTab.trier()
  monTab.afficher()
  monTab.ecrireSurFichier("mes_elements_tries.txt")

Fin
```

Séparation interface et implémentation

- Imaginons que l'on choisisse de représenter le tableau dynamique différemment
  - ex. la taille du tableau allouée est toujours exactement égale au nombre d'éléments du tableau (plus besoin du membre capacité)
- Grâce à une conception modulaire et compartimentée, on ne change que le module TableauDynamique et que la partie implémentation de ce module
  - les entêtes des fonctionnalités sont toujours les mêmes
  - certaines fonctions peuvent changer (comportement interne au module différent) mais elles ne sont pas exportées donc pas visibles dans l'interface
  - la partie Exporter est inchangée, donc les utilisateurs du module n'ont pas besoin de changer leurs codes
- On peut aussi faire 2 modules avec les mêmes interfaces mais des implémentations différentes
  - Sans doute un certain nombre de doublons dans le code...
  - Approche orienté objet: héritage de classes (*cf. LIFAPOO*)

Mise en œuvre en C++

- Répartition du module sur 2 fichiers
  - Fichier .h (fichier d'entêtes) = fichier de promesses des fonctionnalités
    - il contient l'équivalent des parties Importer et Exporter
    - il contient aussi malheureusement les définitions des types (ex. les champs des structures et les membres des classes)
  - Fichier .cpp (fichier source) = mise en œuvre des fonctionnalités
    - il contient l'équivalent des parties Implémentation et Initialisation
    - sauf les définitions de type (dans le fichier .h)
  - L'utilisateur du module écrit les instructions appelant les fonctionnalités dans un fichier (en général le fichier source d'un autre module ou le programme principal)
    - Pour cela il n'a besoin de regarder que le fichier d'entêtes du module utilisé

TableauDynamique.h

```
#ifndef TABDYN_H
#define TABDYN_H

#include <string>
#include "Element.h"

class TableauDynamique {
   public:
      Element * adressePremierElt;
      int capacite;
      int tailleUtilisee;

      TableauDynamique ();
      ~TableauDynamique ();

      void ajoutElement (Element e);
      void trier ();
      void afficher () const;
      void ecrireSurFichier(std::string nom_fichier) const;

    private:
      void extension();
};
#endif
```

TableauDynamique.h

```
#ifndef TABDYN_H
#define TABDYN_H

#include <string>
#include "Element.h"

class TableauDynamique {
   public:
      Element * adressePremierElt;
      int capacite;
      int tailleUtilisee;

      TableauDynamique ();
      ~TableauDynamique ();

      void ajoutElement (Element e);
      void trier ();
      void afficher () const;
      void ecrireSurFichier(std::string nom_fichier) const;

    private:
      void extension();
};
#endif
```

Pour éviter les multiples inclusions

TableauDynamique.h

```
#ifndef TABDYN_H
#define TABDYN_H

#include <string>
#include "Element.h"

class TableauDynamique {
   public:
      Element * adressePremierElt;
      int capacite;
      int tailleUtilisee;

      TableauDynamique ();
      ~TableauDynamique ();

      void ajoutElement (Element e);
      void trier ();
      void afficher () const;
      void ecrireSurFichier(std::string nom_fichier) const;

    private:
      void extension();
};
#endif
```

On importe les modules dont on a besoin: Element pour la donnée membre adressePremierElt et le paramètre de ajoutElement, et string pour le paramètre de ecrireSurFichier

TableauDynamique.h

```
#ifndef TABDYN_H
#define TABDYN_H

#include <string>
#include "Element.h"

class TableauDynamique {
   public:
      Element * adressePremierElt;
      int capacite;
      int tailleUtilisee;

      TableauDynamique ( ) ;
      ~TableauDynamique ( ) ;

      void ajoutElement (Element e) ;
      void trier ( ) ;
      void afficher () const;
      void ecrireSurFichier(std::string nom_fichier) const;

    private:
      void extension( ) ;
};
#endif
```

Les fonctions membres sont maintenant uniquement déclarées (point virgule, pas d'accolade { }). Elles seront implémentées dans le fichier source (.cpp)

TableauDynamique.h

```
#ifndef TABDYN_H
#define TABDYN_H

#include <string>
#include "Element.h"

class TableauDynamique {
   public:
      Element * adressePremierElt;
      int capacite;
      int tailleUtilisee;

      TableauDynamique ();
      ~TableauDynamique ();

      void ajoutElement (Element e);
      void trier ();
      void afficher () const;
      void ecrireSurFichier(std::string nom_fichier) const;

    private:
      void extension();
};
#endif
```

Vous pouvez rassembler les déclarations de données membres de même type et même spécificateur, par exemple: `int capacite, tailleUtilisee;`

TableauDynamique.h

```
#ifndef TABDYN_H
#define TABDYN_H

#include <string>
#include "Element.h"

class TableauDynamique {
   public:
      Element * adressePremierElt;
      int capacite;
      int tailleUtilisee;

      TableauDynamique ();
      ~TableauDynamique ();

      void ajoutElement (Element e);
      void trier ();
      void afficher () const;
      void ecrireSurFichier(std::string nom_fichier) const;

   private:
      void extension();
};
#endif
```

Notez que l'on ne peut pas « cacher » une fonctionnalité interne avec des classes. Tous les membres (publics et privés) doivent apparaitre dans le fichier d'entêtes.

TableauDynamique.cpp

```
#include "TableauDynamique.h"

TableauDynamique::TableauDynamique() {
   adressePremierElt = new Element [1];
   capacite = 1;
   tailleUtilisee = 0;
}

TableauDynamique::~TableauDynamique () {
   delete [] adressePremierElt;
   capacite = 0;
   tailleUtilisee = 0;
}

void TableauDynamique::ajoutElement (Element e) { ... }

void TableauDynamique::trier () { ... }

void TableauDynamique::afficher () const { ... }

void TableauDynamique::ecrireSurFichier(std::string nom_fichier) const { ... }

void TableauDynamique::extension() { ... }
```

TableauDynamique.cpp

```
#include "TableauDynamique.h"

TableauDynamique::TableauDynamique() {
   adressePremierElt = new Element [1];
   capacite = 1;
   tailleUtilisee = 0;
}

TableauDynamique::~TableauDynamique () {
   delete [] adressePremierElt;
   capacite = 0;
   tailleUtilisee = 0;
}

void TableauDynamique::ajoutElement (Element e) { ... }

void TableauDynamique::trier () { ... }

void TableauDynamique::afficher () const { ... }

void TableauDynamique::ecrireSurFichier(std::string nom_fichier) const { ... }

void TableauDynamique::extension() { ... }
```

On inclut le header pour avoir la définition de la classe (entre autres). Si on a besoin de d'autres modules, on les inclut ici (ex. iostream ou math). Notez l'utilisation des guillemets lorsque le module est spécifié relativement au fichier.

TableauDynamique.cpp

```
#include "TableauDynamique.h"

TableauDynamique::TableauDynamique() {
    adressePremierElt = new Element [1];
    capacite = 1;
    tailleUtilisee = 0;
}

TableauDynamique::~TableauDynamique () {
    delete [] adressePremierElt;
    capacite = 0;
    tailleUtilisee = 0;
}

void TableauDynamique::ajoutElement (Element e) { ... }

void TableauDynamique::trier () { ... }

void TableauDynamique::afficher () const { ... }

void TableauDynamique::ecrireSurFichier(std::string nom_fichier) const { ... }

void TableauDynamique::extension() { ... }
```

On utilise la notation nom_classe::nom_membre pour indiquer que l'on implémente un membre de la classe

TableauDynamique.cpp

```
#include "TableauDynamique.h"

TableauDynamique::TableauDynamique() {
   adressePremierElt = new Element [1];
   capacite = 1;
   tailleUtilisee = 0;
}

TableauDynamique::~TableauDynamique () {
   delete [] adressePremierElt;
   capacite = 0;
   tailleUtilisee = 0;
}

void TableauDynamique::ajoutElement (Element e) { ... }

void TableauDynamique::trier () { ... }

void TableauDynamique::afficher () const { ... }

void TableauDynamique::ecrireSurFichier(std::string nom_fichier) const { ... }

void TableauDynamique::extension() { ... }
```

Comme prévu, on peut accéder directement aux membres de la classe (instance sur laquelle la fonction est appelée), quelque soit leurs spécificateurs d'accès.

main.cpp

```
#include <iostream>
#include "TableauDynamique.h"
#include "Element.h"

int main() {
   TableauDynamique monTab;
   Element monElt;
   int finSaisie = 0;

   do {
      monElt.saisieClavier();
      monTab.ajoutElement(monElt);
      std::cout << "Voulez-vous saisir un élément supplémentaire (1:oui, 0:non)?";
      std::cin >> finSaisie;
   } while (finSaisie == 1);

   monTab.trier();
   monTab.afficher();
   monTab.ecrireSurFichier("mes_elements_tries.txt");

   return 0;
}
```

main.cpp

```
#include <iostream>
#include "TableauDynamique.h"
#include "Element.h"

int main() {
   TableauDynamique monTab;
   Element monElt;
   int finSaisie = 0;

   do {
      monElt.saisieClavier();
      monTab.ajoutElement(monElt);
      std::cout << "Voulez-vous saisir un élément supplémentaire (1:oui, 0:non)?";
      std::cin >> finSaisie;
   } while (finSaisie == 1);

   monTab.trier();
   monTab.afficher();
   monTab.ecrireSurFichier("mes_elements_tries.txt");

   return 0;
}
```

On importe les modules nécessaires. iostream pour cin et cout, Element et TableauDynamique pour les classes. Notez qu'importer Element est optionnel puisque TableauDynamique l'importe déjà lui-même (utilité des #ifndef)

main.cpp

```
#include <iostream>
#include "TableauDynamique.h"
#include "Element.h"

int main() {
   TableauDynamique monTab;
   Element monElt;
   int finSaisie = 0;

   do {
      monElt.saisieClavier();
      monTab.ajoutElement(monElt);
      std::cout << "Voulez-vous saisir un élément supplémentaire (1:oui, 0:non)?";
      std::cin >> finSaisie;
   } while (finSaisie == 1);

   monTab.trier();
   monTab.afficher();
   monTab.ecrireSurFichier("mes_elements_tries.txt");

   return 0;
}
```

Notez que si on n'indique pas **using namespace** std; en début de fichier on doit ajouter std:: devant les fonctions cin et cout. Attention ne pas confondre les doubles deux points d'un namespace et ceux de l'implémentation d'une fonction membre dans un fichier source.

main.cpp

```
#include <iostream>
#include "TableauDynamique.h"
#include "Element.h"

int main() {
   TableauDynamique monTab;
   Element monElt;
   int finSaisie = 0;

   do {
      monElt.saisieClavier();
      monTab.ajoutElement(monElt);
      std::cout << "Voulez-vous saisir un élément supplémentaire (1:oui, 0:non)?";
      std::cin >> finSaisie;
   } while (finSaisie == 1);

   monTab.trier();
   monTab.afficher();
   monTab.ecrireSurFichier("mes_elements_tries.txt");

   return 0;
}
```

Notez également que c'est le fichier d'entête (.h) qui est importé, pas le source (.cpp). Un programme externe au module TableauDynamique n'a pas besoin de savoir comment il est implémenté, uniquement quelles sont ses fonctionnalités.

main.cpp

```
#include <iostream>
#include "TableauDynamique.h"
#include "Element.h"

int main() {
   TableauDynamique monTab;
   Element monElt;
   int finSaisie = 0;

   do {
      monElt.saisieClavier();
      monTab.ajoutElement(monElt);
      std::cout << "Voulez-vous saisir un élément supplémentaire (1:oui, 0:non)?";
      std::cin >> finSaisie;
   } while (finSaisie == 1);

   monTab.trier();
   monTab.afficher();
   monTab.ecrireSurFichier("mes_elements_tries.txt");

   return 0;
}
```

Appels aux constructeurs par défaut des deux classes (réservations de mémoire sur la pile).

main.cpp

```
#include <iostream>
#include "TableauDynamique.h"
#include "Element.h"

int main() {
   TableauDynamique monTab;
   Element monElt;
   int finSaisie = 0;

   do {
      monElt.saisieClavier();
      monTab.ajoutElement(monElt);
      std::cout << "Voulez-vous saisir un élément supplémentaire (1:oui, 0:non)?";
      std::cin >> finSaisie;
   } while (finSaisie == 1);

   monTab.trier();
   monTab.afficher();
   monTab.ecrireSurFichier("mes_elements_tries.txt");

   return 0;
}
```

Accès à des membres publics, ici appel à des fonctions membres, on pourrait aussi faire `monTab.tailleUtilisee;` (puisque donnée membre publique)

Remarque sur les déclarations

- Notez que lors de la déclaration d'une fonction/procédure (dans le .h), on n'a pas besoin de donner les noms des paramètres, juste les types suffisent

mais c'est moins lisible/intuitif pour l'utilisateur de la classe, donc à éviter

```
Date(int,int,int);
// quels sont ces entiers? j,m,a? a,m,j? j,j,j?
// sans commentaire impossible de le savoir
```

Compilation de plusieurs fichiers

- Nous avons donc maintenant plusieurs fichiers sources à compiler (1 par module) pour que le programme fonctionne correctement
- Donc toute fonction appelée dans une autre fonction (ex. le main) doit avoir été déclarée ou définie avant (idem pour les procédures)
- Déclaration = juste l'entête de la fonction suivie d'un point virgule (dans le .h)
- Définition = entête + code entre accolades (dans le .cpp)

Compilation de plusieurs fichiers

- Jusqu'à présent en TP: un seul fichier, et les fonctions, procédures et classes étaient définies avant le main
  - On aurait pu aussi déclarer les fonctions avant le main, puis les définir après le main
- Ici, le main appelle des fonctions et procédures, et utilise des types définis dans d'autres fichiers (TableauDynamique et Element)

Comment faire ?

- Importer les entêtes des modules nécessaires
  - Dans main.cpp :
  - Dans TableauDynamique.h :

```
#include "TableauDynamique.h"
```

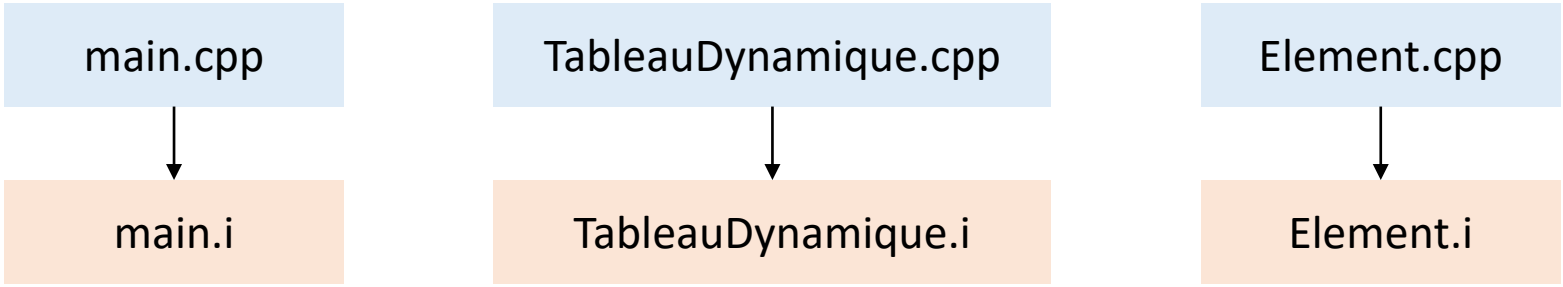
```
#include "Element.h"
```

  - Le (pré)compilateur va remplacer cette ligne par le contenu du fichier .h, donc les déclarations des types/fonctions/procédures offerts par le module apparaîtront en début de fichier
  - On évite les inclusions multiples grâce au

```
#ifndef TABDYN_H
#define TABDYN_H
#endif
```

Compilation de plusieurs fichiers

- Exemple avec nos 3 unités de compilations (3 fichiers contenant des définitions de types/fonctions/procédures)

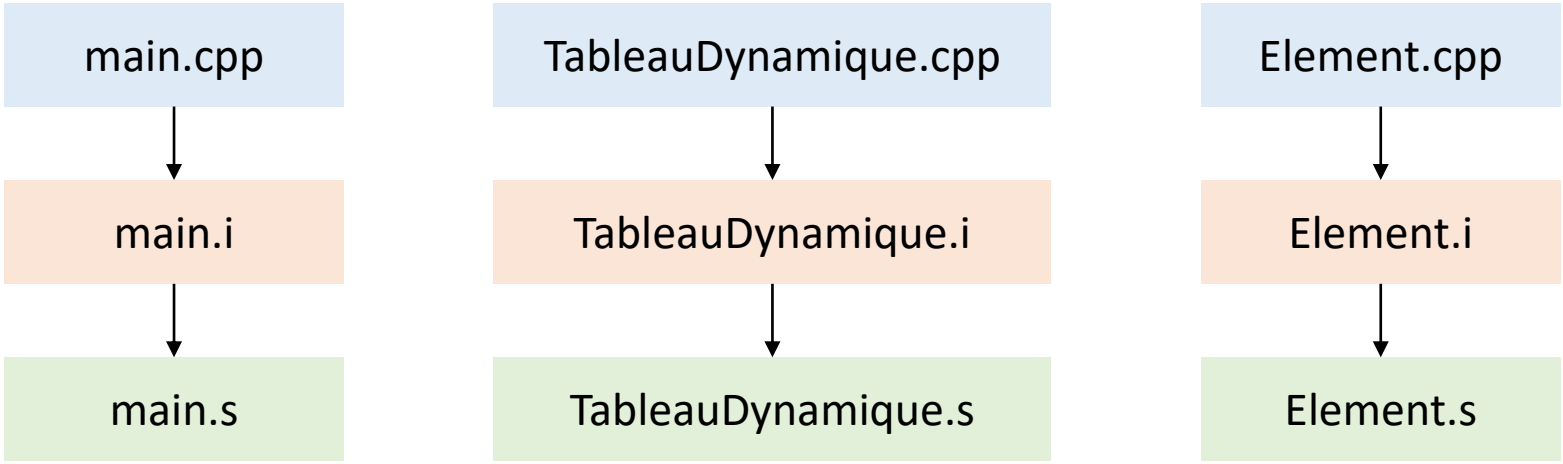


1ère étape: le préprocesseur traite les directives (ex. **#include** et **#define**)

- Fichiers sources complétés (encore lisible avec un éditeur de texte)

Compilation de plusieurs fichiers

- Exemple avec nos 3 unités de compilations (3 fichiers contenant des définitions de types/fonctions/procédures)

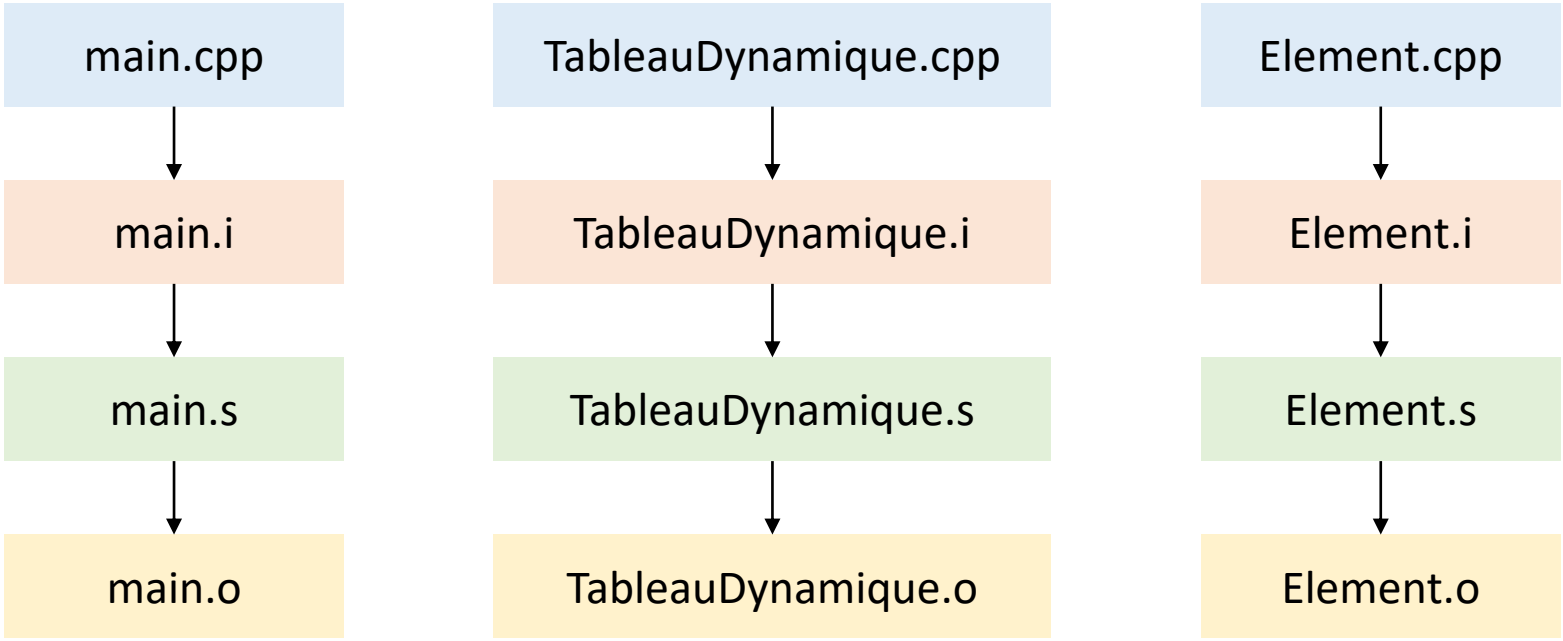


2ème étape: le compilateur traduit les fichiers en langage d'assemblage

➢ Fichiers en langage d'assemblage

Compilation de plusieurs fichiers

- Exemple avec nos 3 unités de compilations (3 fichiers contenant des définitions de types/fonctions/procédures)

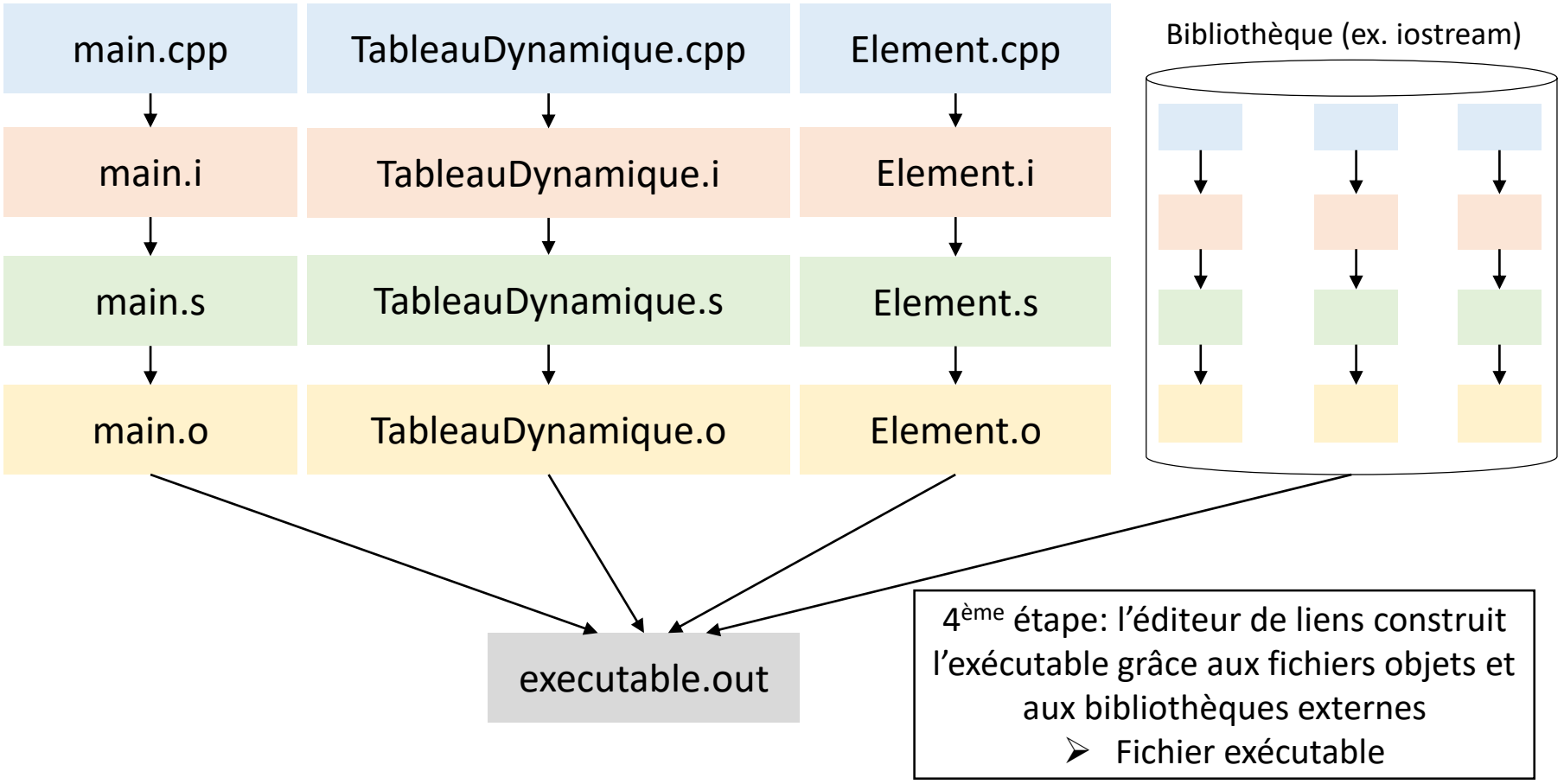


3ème étape: le compilateur traduit les fichiers assemblage en langage machine

➢ Fichiers objets en langage machine

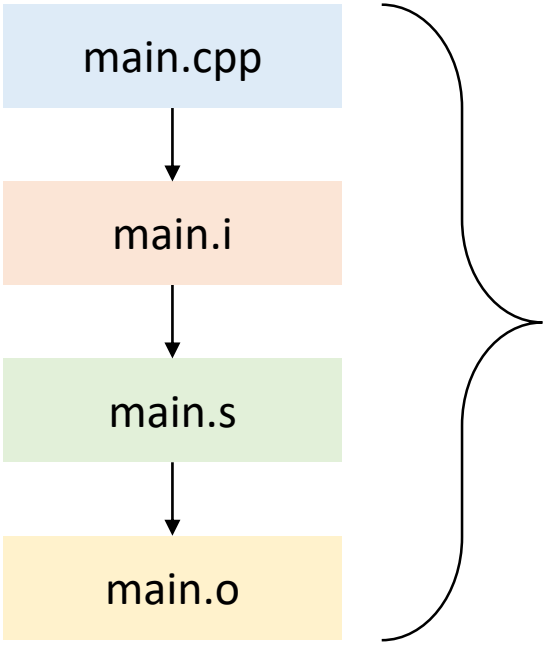
Compilation de plusieurs fichiers

- Exemple avec nos 3 unités de compilations (3 fichiers contenant des définitions de types/fonctions/procédures)



Compilation de plusieurs fichiers

- Les étapes 1, 2 et 3 peuvent être faites en une seule commande du compilateur g++
  - et les fichiers .i et .s sont automatiquement supprimés



Une seule commande:

```
$> g++ –Wall –c main.cpp
```

-c indique de ne produire que le .o *(qui peut être spécifié par l'option –o, sinon même nom que le cpp)*

Compilation de plusieurs fichiers

- Faire de même pour tous les modules :

```
$> g++ -Wall -c TableauDynamique.cpp
$> g++ -Wall -c Element.cpp
```

  - Créé les fichiers TableauDynamique.o et Element.o

- Puis faire l'édition des liens :

```
$> g++ main.o TableauDynamique.o Element.o -o executable.out
```

  - Créé le fichier exécutable executable.out qui peut se lancer par :

```
$> ./executable.out
```

Compilation de plusieurs fichiers

- Si un fichier d'entête est modifié, il faut recompiler le module et tous ceux qui l'utilisent, et refaire l'édition des liens
- Si un fichier source est modifié, il faut recompiler le module et refaire l'édition des liens

Ca fait beaucoup de commandes g++ pour de petites modifications...

Ca demande de connaître quel module utilise quels autres pour ne pas recompiler des fichiers pour rien...

- On voudrait recompiler automatiquement et uniquement les modules nécessaires: Makefile et commande make

Commande make

- make est une commande Unix qui permet d'automatiser la compilation
- Particulièrement utile quand un programme est découpé en modules et donc réparti dans plusieurs fichiers
- Principe:
  - On écrit une fois pour toutes un ficher (appelé Makefile, pas d'extension) qui indique comment construire l'exécutable
    - on spécifie les dépendances entre les fichiers
  - Quand on modifie un fichier (.h ou .cpp), on tape seulement la commande make et seuls les modules nécessaires sont recompilés, et l'exécutable reconstruit
  - La commande make analyse les fichiers utilisés pour la compilation
    - Si un fichier .cpp est plus récent que son fichier .o, on le recompile
    - Si un fichier .o est plus récent que l'exécutable, on reconstruit l'exécutable

Syntaxe du fichier Makefile

- Le fichier Makefile contient un ensemble de règles suivant le format:

```
cible: listes des dépendances <EOL>
<tabulation>commande
```

- Une cible peut être le fichier exécutable, les fichiers objets, mais aussi utilisée pour d'autres actions (suppression de fichiers intermédiaires, génération de documentation, soumission à un serveur, tests, gestionnaire mémoire etc.)

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

La commande make commence par essayer de fabriquer la première cible du fichier Makefile, ici « all » qui sera à jour quand la dépendance executable.out sera à jour

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

La commande make essaye de mettre à jour la nouvelle cible executable.out, qui sera à jour lorsque les dépendances main.o, TableauDynamique.o et Element.o seront à jour

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

La commande make essaye de mettre à jour la cible main.o, qui sera à jour lorsque les dépendances main.cpp, TableauDynamique.h et Element.h seront à jour. Ces dépendances ne sont pas des cibles dans le Makefile, ce sont donc des fichiers. make regarde si au moins un de ces fichiers est plus récent que main.o

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

Si oui, la commande correspondant à la cible main.o est exécutée (compilation de main.cpp).

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

Notez la présence de l'option –g dans la commande g++. Il s'agit de l'option indiquant que l'on compile en mode debug. Juste ne pas mettre de –g pour le mode release.

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

La cible main.o est maintenant à jour, on revient à la dépendance précédente (TableauDynamique.o dans les dépendances de executable.out)

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

La commande make essaye de mettre à jour la cible TableauDynamique.o, qui sera à jour lorsque les dépendances TableauDynamique.h/.cpp et Element.h seront à jour. Ces dépendances ne sont pas des cibles dans le Makefile, ce sont donc des fichiers. make regarde si au moins un de ces fichiers est plus récent que TableauDynamique.o

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

Si oui, la commande correspondant à la cible TableauDynamique.o est exécutée. La cible TableauDynamique.o est maintenant à jour, on revient à la dépendance précédente (Element.o dans les dépendances de executable.out)

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

La commande make essaye de mettre à jour la cible Element.o, qui sera à jour lorsque les dépendances Element.h et Element.cpp seront à jour. Ces dépendances ne sont pas des cibles dans le Makefile, ce sont donc des fichiers. make regarde si au moins un de ces fichiers est plus récent que Element.o

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

Si oui, la commande correspondant à la cible Element.o est exécutée. La cible Element.o est maintenant à jour, on revient à la dépendance précédente.

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

executable.out a toutes ces dépendances mises à jour, donc la commande correspondant à la cible executable.out est exécutée (édition des liens). La cible executable.out est maintenant à jour, on revient à la dépendance précédente (all).

Exemple de fichier Makefile

```
all: executable.out

executable.out: main.o TableauDynamique.o Element.o
	g++ -g main.o TableauDynamique.o Element.o -o executable.out

main.o: main.cpp TableauDynamique.h Element.h
	g++ -g -Wall -c main.cpp

TableauDynamique.o: TableauDynamique.h TableauDynamique.cpp Element.h
	g++ -g -Wall -c TableauDynamique.cpp

Element.o: Element.h Element.cpp
	g++ -g -Wall -c Element.cpp

clean:
	rm *.o

veryclean: clean
	rm *.out
```

all a toutes ces dépendances mises à jour (il y avait seulement executable.out), la commande (vide) de la cible all est exécutée. C'était la première cible, la commande make est finie (et tout est bien à jour!).

Commande make

- Dans l'exemple précédent, pour nettoyer les fichiers il faut taper la commande:

```
$> make veryclean
```

  - Pour directement mettre à jour la cible veryclean

- Les commandes suivantes sont équivalentes

```
$> make
```

```
$> make all
```

```
$> make executable.out
```

- A retenir
  - Les dépendances d'un exécutable sont les .o du projet
  - Les dépendances d'un .o sont les fichiers .h et .cpp du module et les .h utilisés par le module