



# LIF5 - Algorithmique et programmation procédurale

Carole Knibbe

Samir Akkouché



# Plan du cours

1. TDA et programmation séparée
  - a. Les grands problèmes du Génie Logiciel
  - b. Les TDA et les modules
2. Tableaux dynamiques
  - a. Introduction
  - b. Description du module Tab\_Dynamique
  - c. Mise en œuvre et complexité des opérations
3. Listes chaînées
4. Arbres binaires
5. Piles et Files

# Chapitre 6:

## Types de données abstraits et programmation séparée

*"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."*

Martin Golding



# Les grands problèmes du génie logiciel

- 30% des projets informatiques sont annulés avant la mise en production (Aberdeen)
  - United Airlines annule son projet de système de réservation de places
  - Avis Europe abandonne le déploiement de l'ERP Peoplesoft : 45M€
- 50% des projets informatiques ne répondent pas au cahier des charges business (Gartner)
  - Système de réservation de places Socrate de la SNCF
- 50% des projets informatiques dépassent le budget prévu (Gartner)
  - Système d'information de la Bibliothèque Nationale de France, 1999 : 22 mois de retard, budget initial dépassé de 40%
- Aux USA, on estime à 60 milliards de dollars par an les pertes dues aux bugs logiciels

# 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

Montre	Combien de lignes de code, à votre avis ?
Téléphone mobile	
Automobile	
Central téléphonique	
Noyau Linux	
Système de combat du porte-avions Charles de Gaulle	
Portail Yahoo	
Windows NT	
Linux	
Direction générale de la comptabilité publique (Bercy)	

# Maîtriser la complexité

Montre	~ 2 000 instructions
Téléphone mobile	~ 150 000 instructions
Automobile	~ 1 million d'instructions
Central téléphonique	~ 1 million d'instructions
Noyau Linux	3,7 millions d'instructions
Système de combat du porte-avions Charles de Gaulle	~ 8 millions d'instructions
Portail Yahoo	~ 11 millions d'instructions
Windows NT	16,5 millions d'instructions
Linux	100 millions d'instructions
Direction générale de la comptabilité publique (Bercy)	160 millions d'instructions

# Maîtriser la complexité

- Diviser pour régner = découper l'application en modules :
  - aussi indépendants que possibles
  - faciles à comprendre
  - testables individuellement (puis ensemble bien sûr)
- (Difficulté = trouver le bon découpage !)





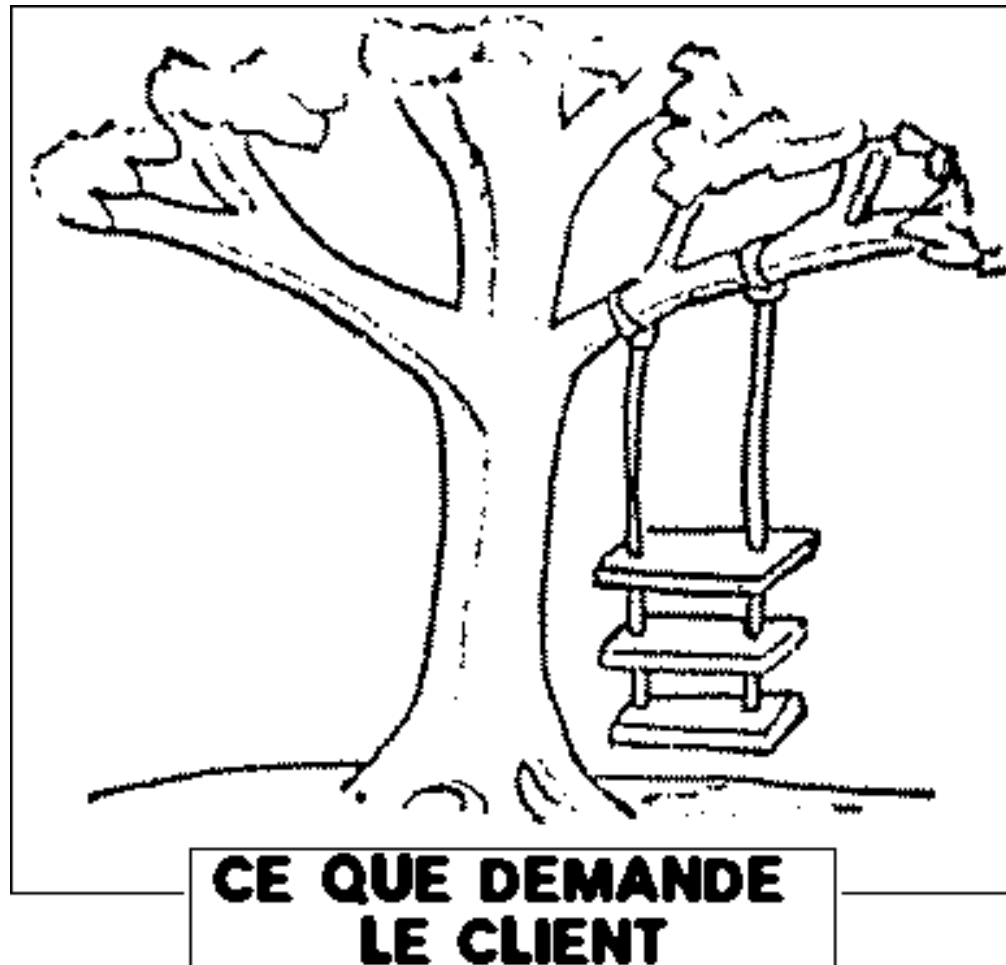
# Réutiliser du code existant

- Résoudre une fois pour toutes une famille de problèmes
  - somme, produit de nombres complexes
  - dessin de formes graphiques en 2D ou en 3D
- Bibliothèque d'usage général
  - le calcul avec des nombres complexes est un problème général, non lié à une application particulière
  - idem pour le dessin en 2D ou 3D : jeux vidéo, simulations scientifiques, etc

# 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 ?
- 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

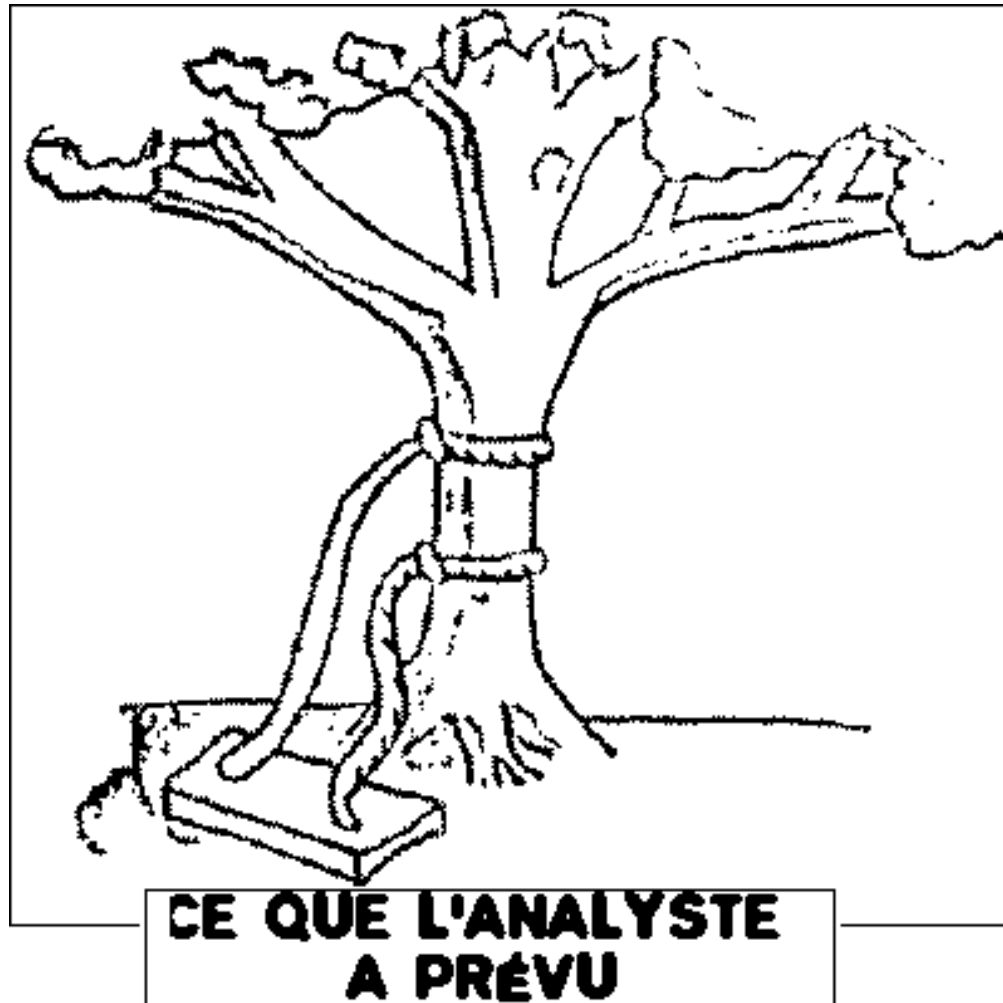
# Détecter les erreurs le plus tôt possible



# Détecter les erreurs le plus tôt possible



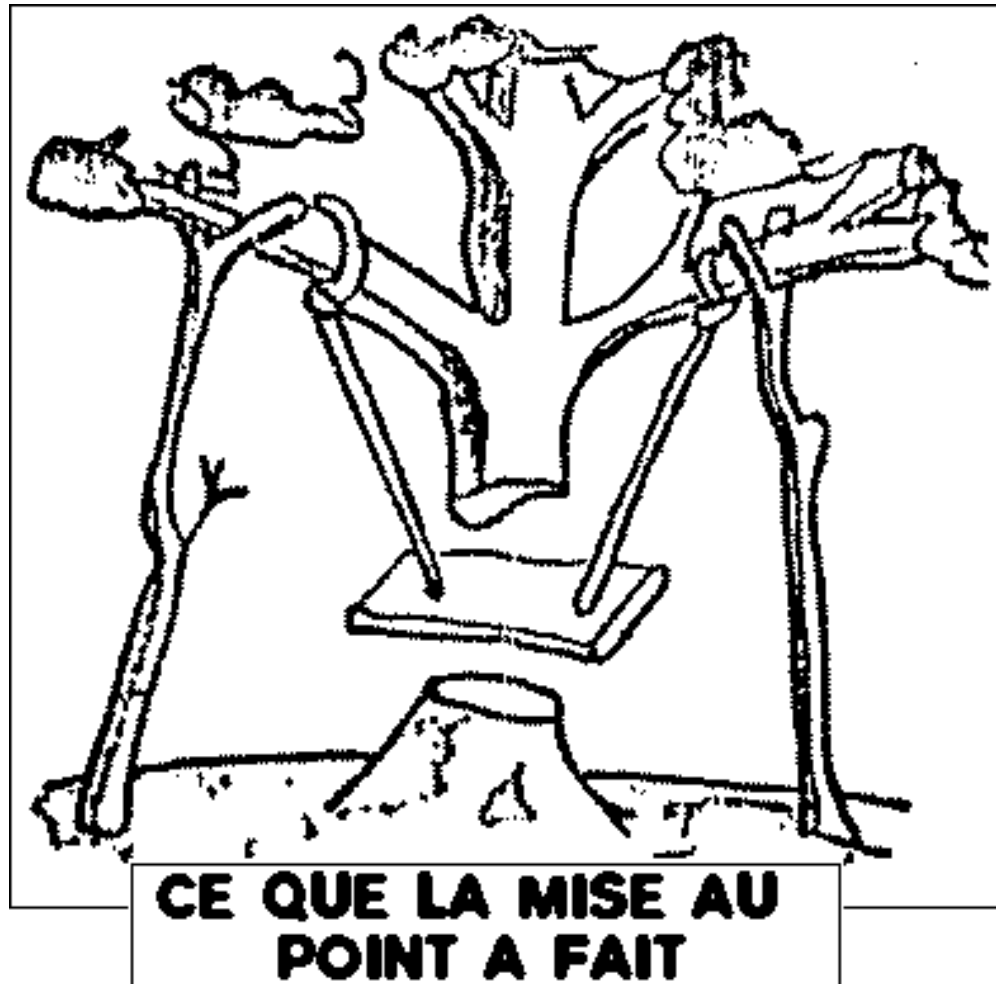
# Détecter les erreurs le plus tôt possible



# Détecter les erreurs le plus tôt possible



# 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 faites, faites ce qui est écrit, prouvez que vous le faites »
- Tester, tester, tester
  - tests unitaires = chaque module individuellement
  - tests d'intégration



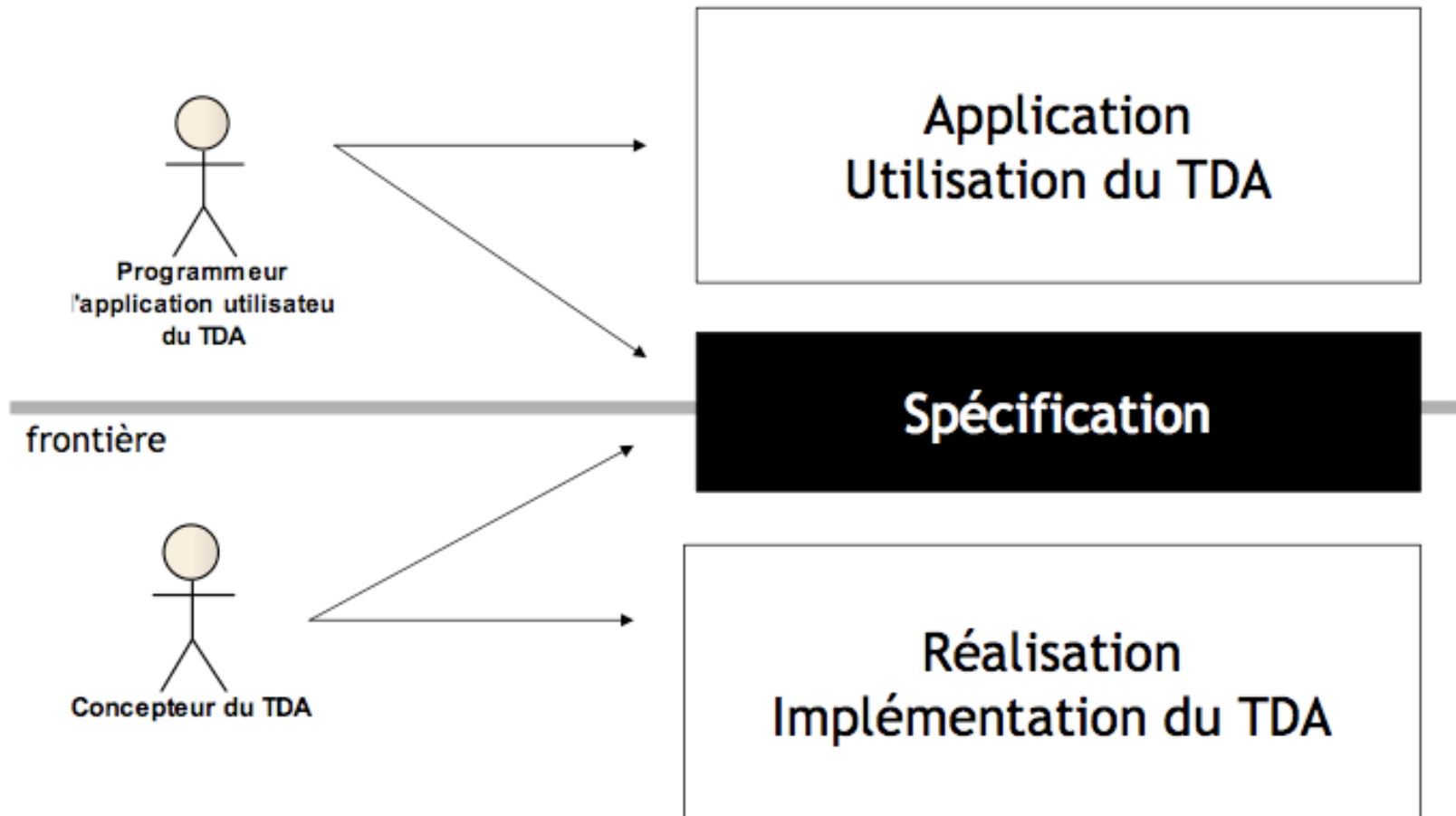
# Les types de données abstraits (TDA)

- 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 séparées de la représentation interne des objets et de la mise en oeuvre 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 préconditions de ces opérations, pour éviter les dépassements de capacités par exemple
- On peut construire des TDA plus complexes à partir des types de base
  - créer un type (ex. avec une struct en C) 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

# TDA : Deux points de vue différents



# TDA et modules

- 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)

# Module : norme algorithmique

Module nom\_module {*rôle du module*}

- Importer :
  - **Déclaration** des modules extérieurs utilisés dans l'interface de nom\_module
- Exporter :
  - **Déclaration** des types, procédures, fonctions, constantes, variables gloables offerts par nom\_module
- Implantation :
  - **Déclaration** des modules extérieurs utilisés dans l'implantation de nom\_module
  - **Définition** des types, procédures, fonctions, constantes, variables globales offertes par nom\_module
  - **Définition** éventuelle de types, procédures, fonctions, constantes, variables gloables internes au module (utiles pour l'implantation de nom\_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 :

...

- Exporter :

...

- Implantation :

• ...

• ...

• ...

- Initialisation :

• ....

FinModule

Interface (spécification)

En C: Fichier .h

Implantation (code)

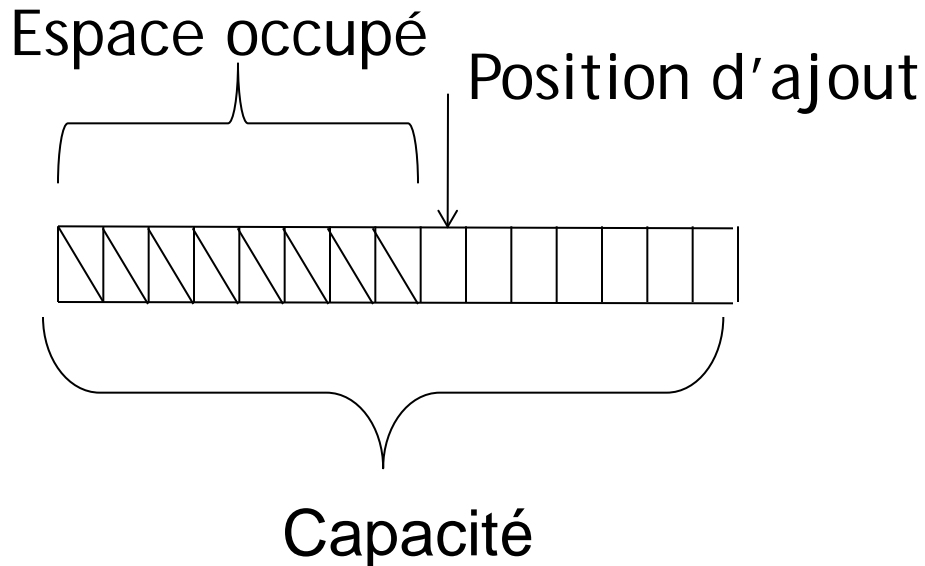
En C : Fichier .c

# Tableaux Dynamiques



# Caractéristiques d'un tableau

- Les cellules sont contigues en mémoire
- Capacité fixée à l'avance
- Accès direct
- Deux parties distinctes : Partie pleine et partie avec des éléments non significatifs (pour le problème traité)





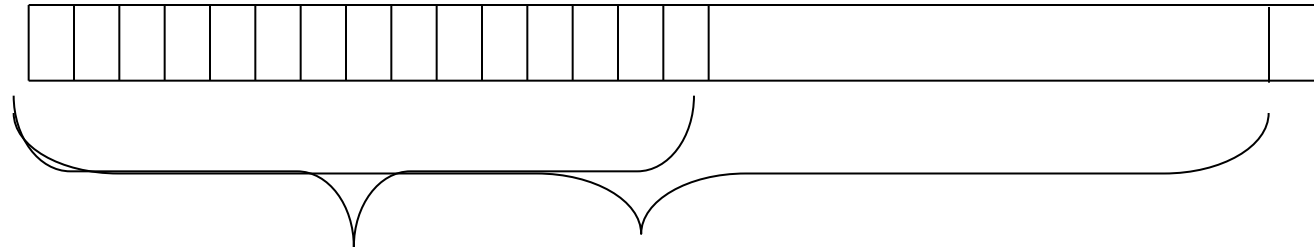


# Tableau Dynamique

Type à accès direct dont on ne **S'OCCUPE PAS** de la taille.

- Il faut pouvoir augmenter la taille à loisir(et la réduire).
- Transparent pour l'utilisateur

**Type** TableauDynamique = structure  
    adressePremierElt : pointeur sur Element  
    capacite : entier  
    tailleUtilisee : entier  
Fin structure TableauDynamique



Capacité

## Module TableauDynamique

- **Importer :**

Module Element

- **Exporter :**

**Type** TableauDynamique

**Procédure** initialiserTabDyn(T: TableauDynamique )

Préconditions : T n'a pas déjà été initialisé

Postconditions : réservation d'un tableau de taille 1 Element dans le tas, initialisation de la capacité et de la taille utilisée de T

Paramètres en mode donnée-résultat : T

**Procédure** testamentTabDyn(T : TableauDynamique)

Préconditions : T initialisé

Postconditions : libération de la mémoire utilisée par T dans le tas

Paramètre en mode donnée-résultat : T

**Procédure** ajoutElementTabDyn (T: TableauDynamique, e: Element)

Préconditions : T initialisé

Postcondition : une copie de e est insérée à la fin de T, extension de l'espace mémoire alloué au tableau si nécessaire

Paramètre en mode donnée : e

Paramètre en mode donnée-résultat : T

**Etc :** suppression élément, affichage, tri, écriture sur fichier...

## - Implantation :

**Type** TableauDynamique = structure  
    adressePremierElt : pointeur sur Element  
    capacite : entier  
    tailleUtilisee : entier  
Fin structure TableauDynamique

**Procédure** initialiser(T: TableauDynamique)  
Début  
    T.adressePremierElt ← réserve tableau [1..1] de Elements  
    T.capacite ← 1  
    T.tailleUtilisee ← 0  
Fin initialiser

**Procédure** testament(T: TableauDynamique)  
Début  
    libère T.adressePremierElt  
    T.capacite ← 0  
    T.tailleUtilisee ← 0  
Fin testament

**Etc.**

# Module TableauDynamique : un exemple de programme utilisateur

```
Importer
  module TableauDynamique
  module Element

Début
  monTab : TableauDynamique
  monElt : Element
  finSaisie : booléen

  finSaisie <- faux
  initialiserTabDyn(monTab)
  Répéter
    saisieClavierElement(monElt)
    ajoutElementTabDyn(monTab, monElt)
    afficher(« Voulez-vous saisir un élément supplémentaire ? »)
    lire(finSaisie)
  Jusqu'à ce que (finSaisie = vrai)

  trierTabDyn(monTab)
  afficherTabDyn(monTab)
  ecrireSurFichier(monTab, « mes_elements_tries.txt »)

  testamentTabDyn(monTab)
Fin
```

# Intérêt de séparer interface et implantation

- On peut aussi choisir de représenter les complexes en coordonnées polaires (module, argument)
- On ne change que la partie « implantation »
- La partie « exporter » (interface) est inchangée, donc les utilisateurs du module n'ont pas besoin de changer leurs codes
- On peut faire 2 modules avec les mêmes interfaces mais des implantations différentes
  - Extension mémoire par doublements successifs
  - ... ou bien par ajout d'une quantité constante de cases
  - l'utilisateur choisit le module qui lui convient le mieux en fonction de ses besoins

# Mise en œuvre en C

Répartition du module sur 2 fichiers :

- Fichier .h (fichier d'entêtes) = fichier de promesses !
  - contient l'équivalent des parties « importer » et « exporter »
  - et contient aussi **malheureusement** les définitions de type
- Fichier .c (fichier source) = mise en oeuvre :
  - contient l'équivalent des parties « implantations » et « initialisation »
  - ... sauf les définitions de type

L'utilisateur du module écrit son « main » dans un troisième fichier

Il n'a besoin de regarder que le .h pour pouvoir utiliser le module

# TableauDynamique.h

```
#ifndef __TABDYN
#define __TABDYN

#include "Element.h"

struct sTableauDynamique {
    Element * adressePremierElt;
    int capacite;
    int tailleUtilisee;
};
typedef struct sTableauDynamique TableauDynamique;

void initialiserTabDyn(TableauDynamique * T);
/* Précondition : T n'a pas déjà été initialisé */
/* Postcondition : réservation mémoire d'un tableau de 1 Element */

void testamentTabDyn(TableauDynamique * T);
/* Précondition : T a déjà été initialisé */
/* Postcondition : libération de la mémoire occupée par le tableau */

Etc.

#endif
```

# TableauDynamique.c

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

void initialiserTabDyn(TableauDynamique * T) {
    T->adressePremierElt = malloc(1*sizeof(Element));
    T->capacite = 1;
    T->tailleUtilisee = 0;
}

void testamentTabDyn(TableauDynamique * T) {
    free(T->adressePremierElt);
    T->capacite = 0;
    T->tailleUtilisee = 0;
}

void ajoutElementTabDyn(TableauDynamique * T, Element e) {    }

Etc.
```



# main.c

```
#include <stdio.h>                /* Chevrons : Fichiers système */
#include "TableauDynamique.h"      /* Guillemets : Fichiers du répertoire courant */
#include "Element.h"

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

    initialiserTabDyn(&monTab);
    do {
        saisieClavierElement(&monElt);
        ajoutElementTabDyn(&monTab, monElt);
        printf("Voulez-vous saisir un élément supplémentaire (1:oui, 0:non)?");
        scanf("%d", &finSaisie);
    } while (finSaisie == 1);

    trierTabDyn(&monTab);
    afficherTabDyn(&monTab);
    ecrireSurFichier(monTab, « mes_elements_tries.txt »);

    testamentTabDyn(&monTab);
    return 0;
}
```

# Comment compiler un programme réparti sur plusieurs fichiers ?

- Toute fonction appelée dans une autre fonction (par 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
- Définition = entête + code entre accolades

# Comment compiler un programme réparti sur plusieurs fichiers ?

- Jusqu'à présent en TP : un seul fichier, et les fonctions et procédures appelées étaient définies avant le main
- On aurait pu aussi déclarer les fonctions avant le main, puis les définir après le main si on l'avait souhaité
- Mais ici, le main appelle des fonctions et des procédures définies dans un autre fichier : `TableauDynamique.c` → comment faire ?
- Solution : **`#include "TableauDynamique.h"`** au début de `main.c`
  - le compilateur va remplacer cette ligne par le contenu du fichier `.h`, donc les déclarations des fonctions offertes par le module apparaîtront au-dessus du main
  - On évite les inclusions multiples grâce au **`#define __TABDYN`** et au **`#ifndef __TABDYN`**

# Comment compiler un programme réparti sur plusieurs fichiers ?

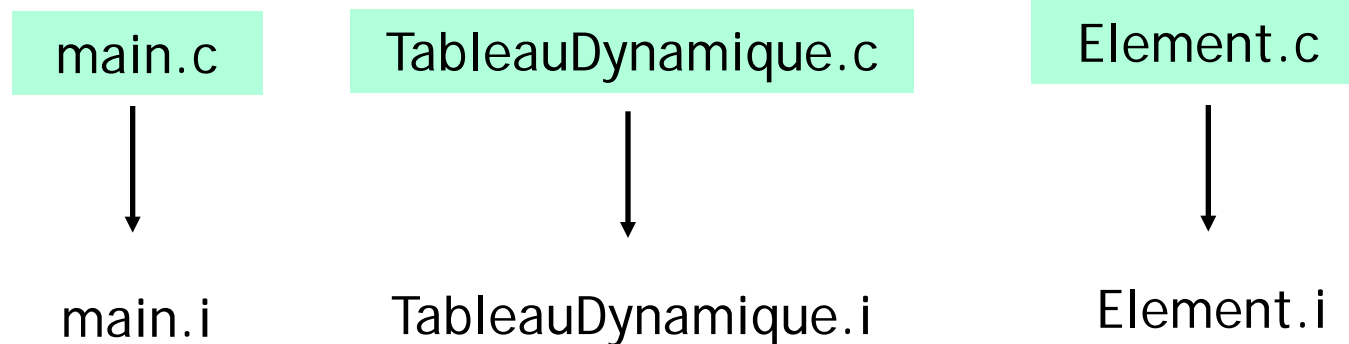
main.c

TableauDynamique.c

Element.c

ici, on a 3 **unités de compilation** =  
3 fichiers contenant des définitions  
de fonctions ou procédures

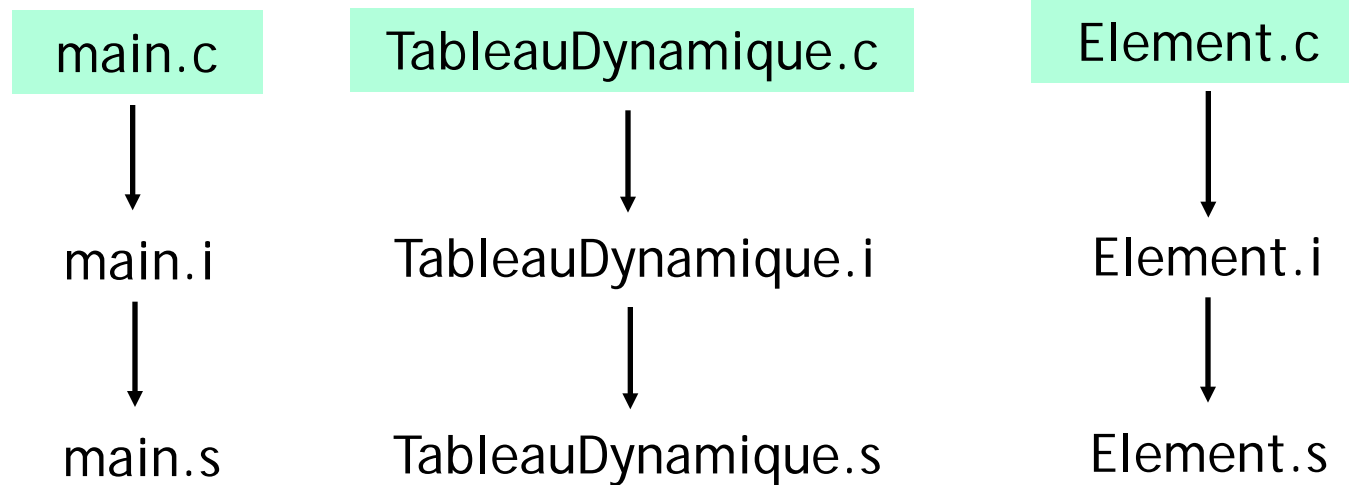
# Comment compiler un programme réparti sur plusieurs fichiers ?



1e étape : le préprocesseur fait les `#include`, `#define`, etc.

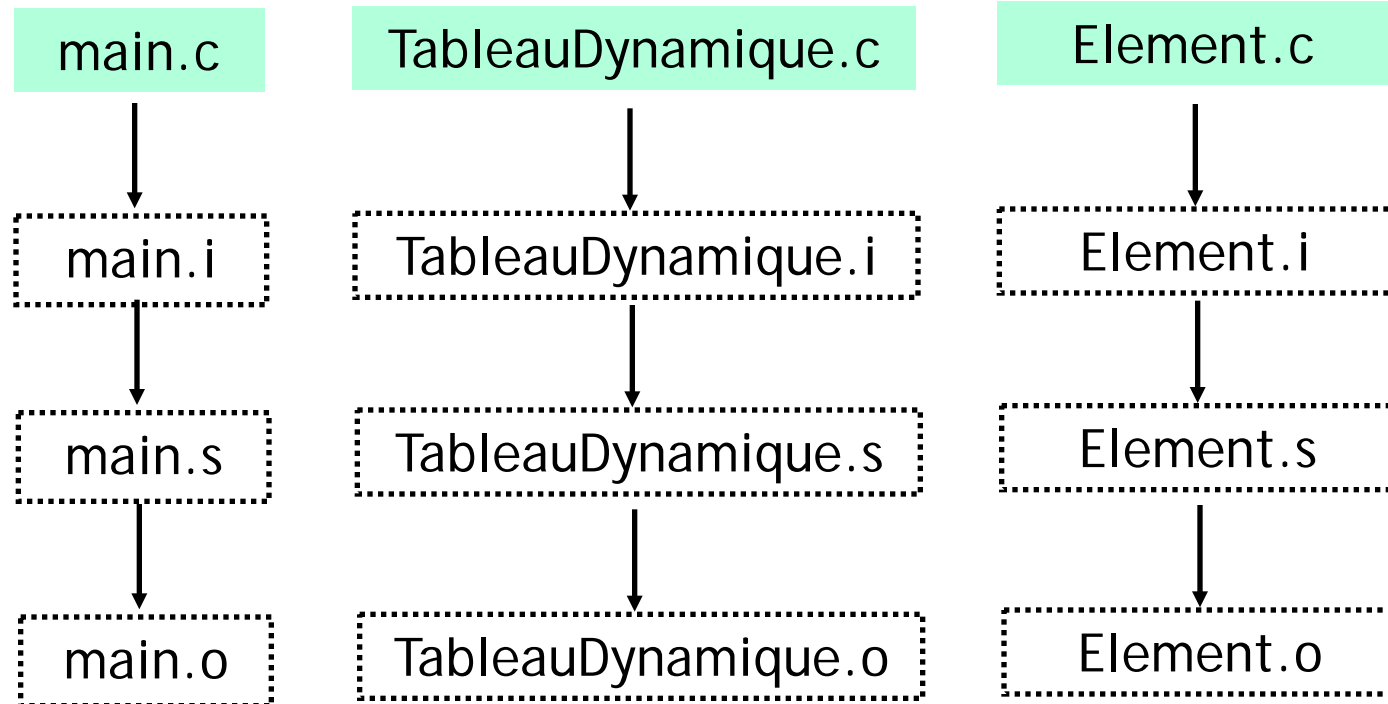
➔ Fichiers sources complétés, lisibles avec un éditeur de texte

# Comment compiler un programme réparti sur plusieurs fichiers ?



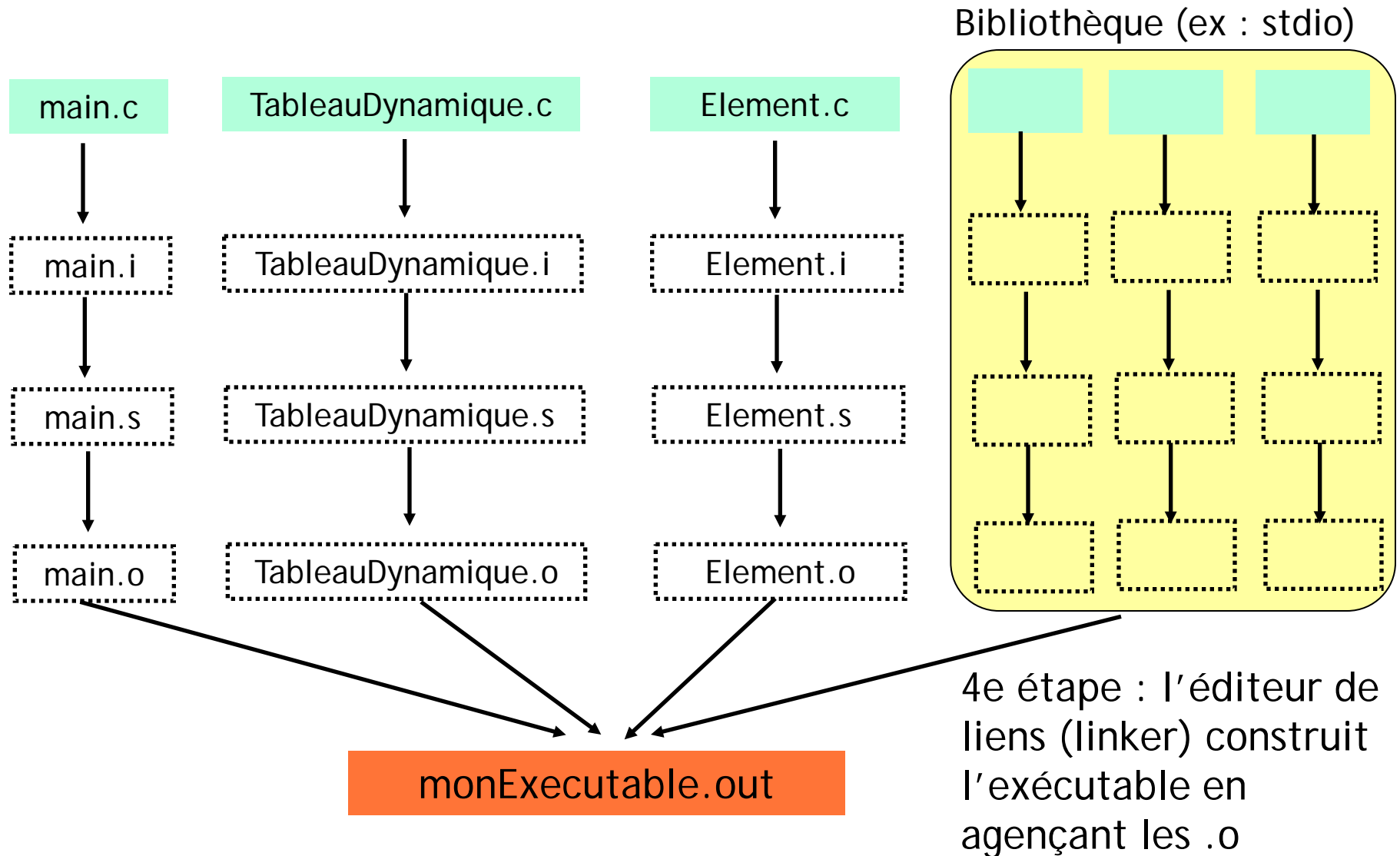
2e étape : le compilateur traduit les fichiers en langage d'assemblage

# Comment compiler un programme réparti sur plusieurs fichiers ?



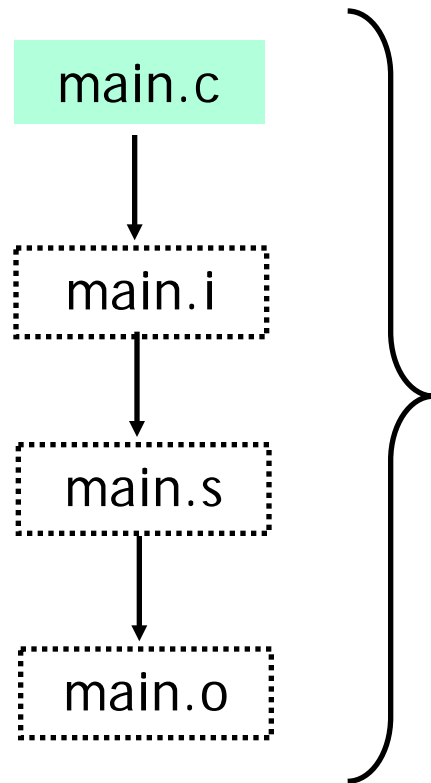
3e étape : le compilateur traduit du langage d'assemblage vers le langage machine

# Comment compiler un programme réparti sur plusieurs fichiers ?





# Comment compiler un programme réparti sur plusieurs fichiers ?

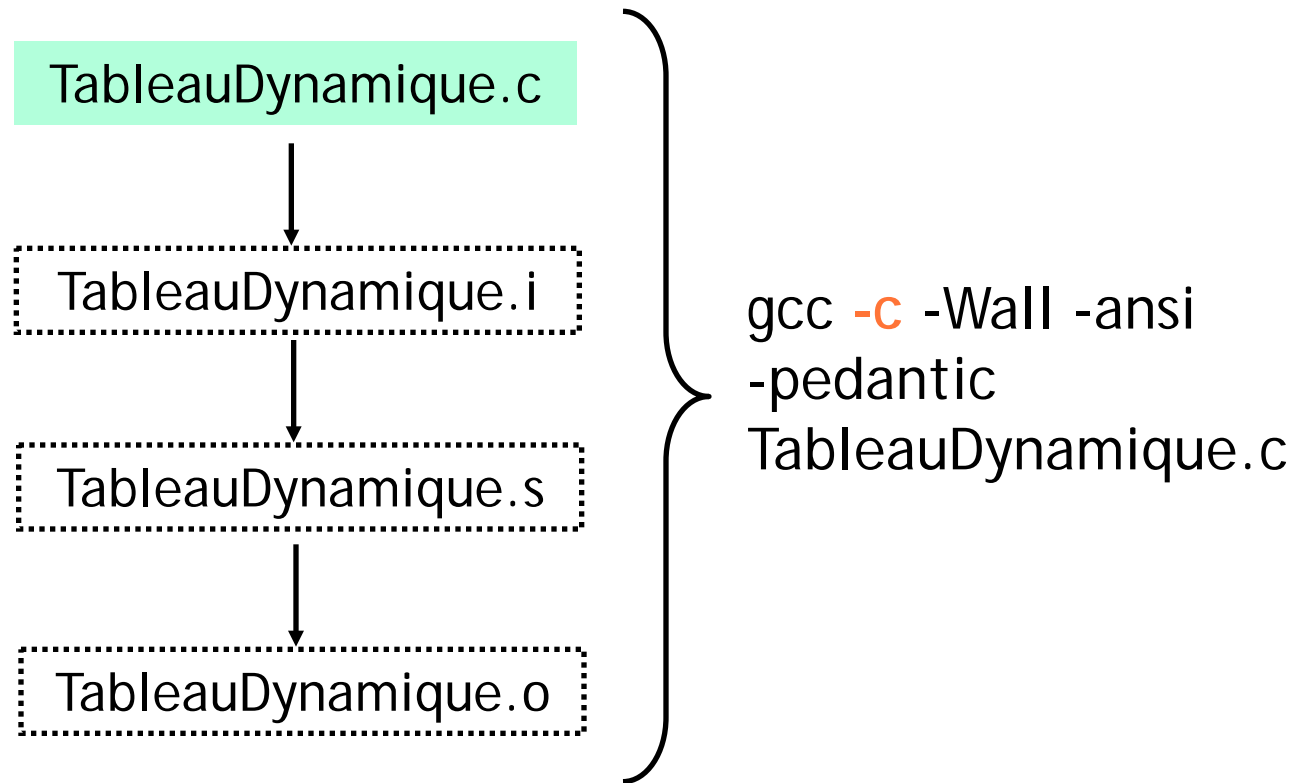


Une commande :

gcc -c -Wall -ansi -pedantic main.c

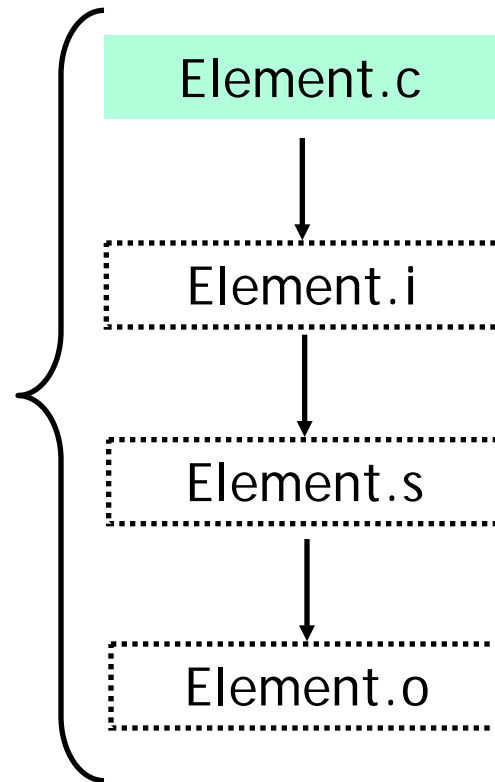
ne pas tenter de faire  
l'exécutable, s'arrêter au .o

# Comment compiler un programme réparti sur plusieurs fichiers ?



# Comment compiler un programme réparti sur plusieurs fichiers ?

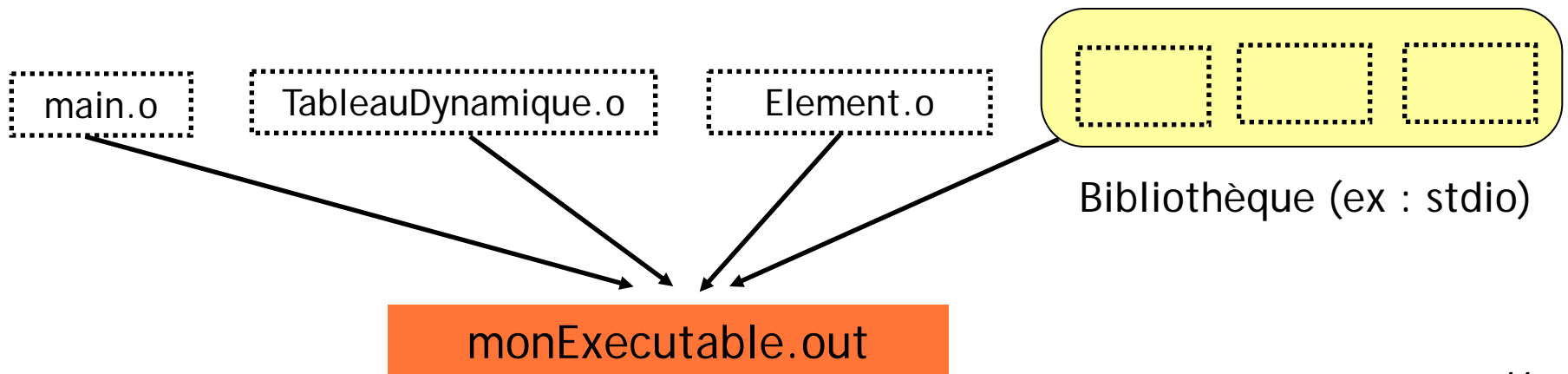
```
gcc -c -Wall -ansi  
-pedantic Element.c
```



# Comment compiler un programme réparti sur plusieurs fichiers ?

Edition des liens :

```
gcc main.o TableauDynamique.o Element.o -o monExecutable.out
```



# Makefile : une seule commande au lieu de 3

- « Make » est un outil Unix qui permet d'automatiser la compilation
- Particulièrement utile quand un programme est découpé en modules et donc réparti dans plusieurs fichiers
- Idée :
  - on écrit une fois pour toutes un fichier (appelé Makefile) qui indique comment construire l'exécutable
  - quand on modifie un fichier source, on tape seulement « make » : les modules modifiés (et eux seuls) sont recompilés et l'exécutable est reconstruit



# Makefile : exemple

```
monExecutable.out: main.o TableauDynamique.o Element.o
    gcc main.o TableauDynamique.o Element.o -o monExecutable.out
```

```
main.o: main.c TableauDynamique.h Element.h
    gcc -c -Wall -ansi -pedantic main.c
```

```
TableauDynamique.o: TableauDynamique.c TableauDynamique.h Element.h
    gcc -c -Wall -ansi -pedantic TableauDynamique.c
```

```
Element.o: Element.c Element.h
    gcc -c -Wall -ansi -pedantic Element.c
```



# Makefile : exemple

- Syntaxe générale : série de paires de lignes de la forme

**cible: liste des dépendances directes de cette cible**

**[TABULATION] commande pour fabriquer la cible à partir des dépendances**

- Si les fichiers dont la cible dépend sont plus récents que la cible, alors make va détecter qu'il faut refaire ces fichiers, puis refaire la cible en exécutant la commande.
- Quand on tape make, c'est par défaut la première cible qui est refaite (si nécessaire).
- Une cible pour l'exécutable : les dépendances sont les fichiers .o
- Une cible par .o : les dépendances sont le fichier .c et les .h qui sont inclus dans ce .c

# Makefile : variantes

```
all: monExecutable.out
```

```
main.o: main.c TableauDynamique.h Element.h  
    gcc -c -Wall -ansi -pedantic main.c
```

```
TableauDynamique.o: TableauDynamique.c TableauDynamique.h Element.h  
    gcc -c -Wall -ansi -pedantic TableauDynamique.c
```

```
Element.o: Element.c Element.h  
    gcc -c -Wall -ansi -pedantic Element.c
```

```
monExecutable.out: main.o TableauDynamique.o Element.o  
    gcc main.o TableauDynamique.o Element.o -o monExecutable.out
```

```
clean:  
    rm -r *.o monExecutable.out
```

make clean

make all





# Makefile : variantes

CC = gcc

option= -Wall -ansi -pedantic

all: monExecutable.out

main.o: main.c TableauDynamique.h Element.h  
\$(CC) \$(option) -c main.c

TableauDynamique.o: TableauDynamique.c TableauDynamique.h Element.h  
\$(CC) \$(option) -c TableauDynamique.c

Element.o: Element.c Element.h  
\$(CC) \$(option) -c Element.c

monExecutable.out: main.o TableauDynamique.o Element.o  
\$(CC) main.o TableauDynamique.o Element.o -o monExecutable.out

clean:  
rm -r \*.o monExecutable.out



# Makefile : variantes

```
CC = gcc
option = -Wall -ansi -pedantic
objets = main.o TableauDynamique.o Element.o

all: monExecutable.out

main.o: main.c TableauDynamique.h Element.h
    $(CC) $(option) -c main.c

TableauDynamique.o: TableauDynamique.c TableauDynamique.h Element.h
    $(CC) $(option) -c TableauDynamique.c

Element.o: Element.c Element.h
    $(CC) $(option) -c Element.c

monExecutable.out: $(objets)
    $(CC) $(objets) -o monExecutable.out

clean:
    rm -r *.o monExecutable.out
```

# Tableau Dynamique : Suite

## Module TableauDynamique

### -----Suite-----

**fonction** `tailleutilisee(T : TableauDynamique) : entier`

Précondition : T bien initialisé,

Résultat : retourne le nombre effectif d'éléments dans le tableau

Paramètre en mode donnée : t

**Procédure** `ajoutElementTabDyn (T: TableauDynamique, e: Element)`

Préconditions : T initialisé

Postcondition : une copie de e est insérée à la fin de T, extension de l'espace mémoire alloué au tableau si nécessaire

Paramètre en mode donnée : e

Paramètre en mode donnée-résultat : T

**procedure** `insereElement( t : TableauDynamique, e: Element, i: entier)`

Précondition : t est bien initialisé et  $i < \text{tailleutilisee}$

Postcondition : e est inséré à la position i dans t

Paramètres en mode donnée : e, i

Paramètre en mode donnée-résultat : t



# Tableau Dynamique : Suite

```
#ifndef __TABDYN
#define __TABDYN

#include "Element.h"

struct sTableauDynamique {
    Element * adressePremierElt;
    int capacite;
    int tailleUtilisee;
};
typedef struct sTableauDynamique TableauDynamique;

void initialiserTabDyn(TableauDynamique * );

void testamentTabDyn(TableauDynamique * );

int tailleutilisee(TableauDynamique * );

void ajouteElement(TableauDynamique *, Element );

void insereElement(TableauDynamique *, Element , int);

#endif2
```

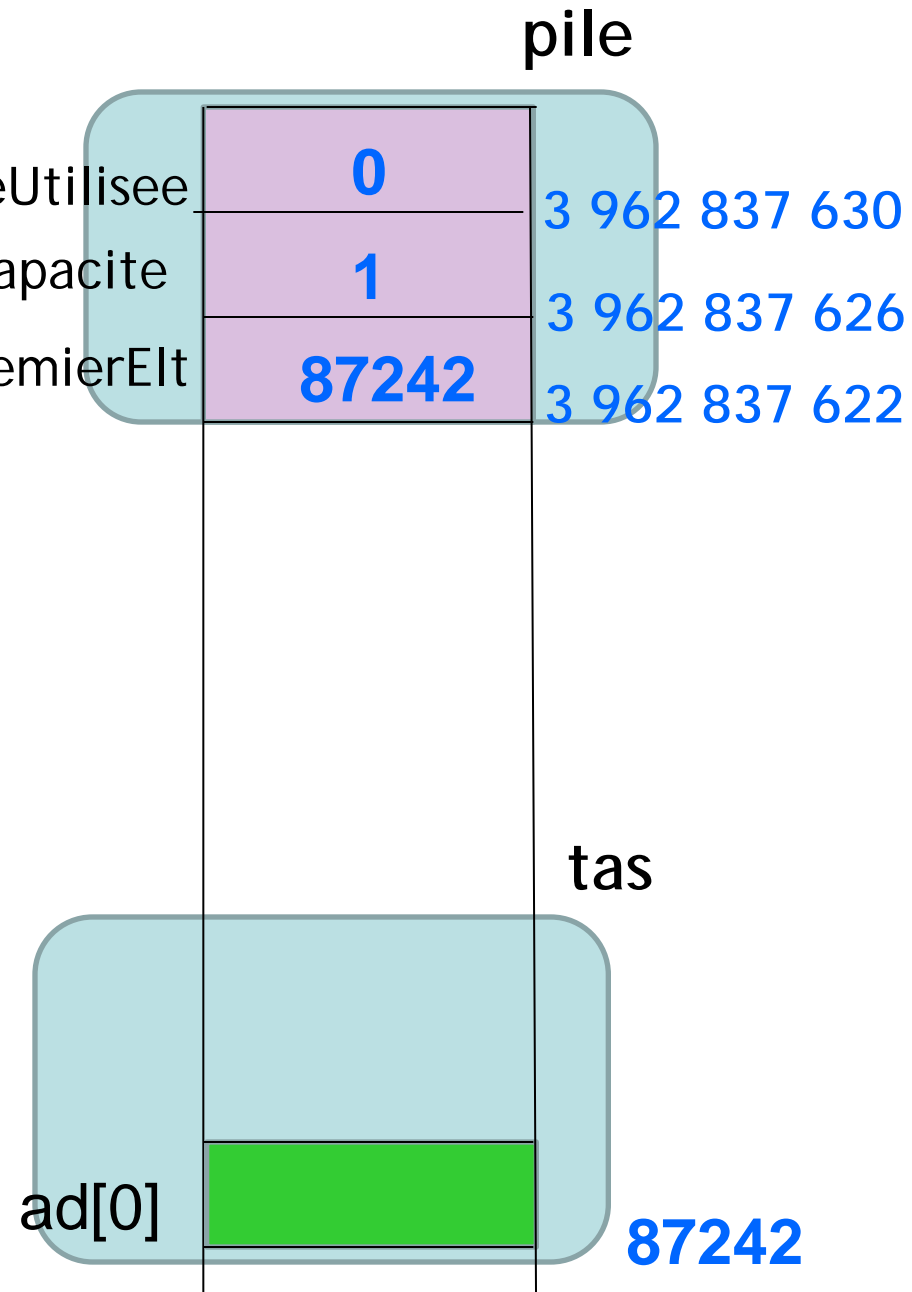


# initialisation

```
void initialiser(TableauDynamique * t)
{
    (*t).capacite = 1;
    (*t).ad = malloc(sizeof(Element));
    (*t).tailleUtilisee = 0;
}

/* Attention ici ad = adressePremierElt */
```

```
int main()
{
    TableauDynamique a;
    initialiser(&a);
}
```





# Ajout d'éléments

```
void ajouteElement(TableauDynamique *t, Element e)
```

```
{int i; Element *temp;
```

```
If((*t).tailleUtilisee == (*t).capacite)
```

```
{
```

```
temp = (*t).adressePremierElt;
```

```
(*t).ad = malloc(2*(*t).capacite*sizeof(Element));
```

```
for( i=0;i< (*t).capacite);i++){(*t).ad[i]=temp[i];}
```

```
(*t).capacite = 2*(*t).capacite;
```

```
free(temp);
```

```
}
```

```
(*t).ad[(*t).tailleUtilisee]=e;
```

```
(*t).tailleUtilisee++;
```

```
}
```

tailleUtilisee

capacite

adressePremierElt

temp

**pile**

4

6

87254

87242

3 962 837 630

3 962 837 626

3 962 837 622

3 962 837 618

**tas**

ad[5]

ad[4]

ad[3]

ad[2]

ad[1]

ad[0]

e

34

15

21

34

15

21

87274

87270

87266

87262

87258

87254

87250

87246

87242

# Etude des coûts

	Coût : tableau statique		Tableau dynamique
Ajout	Temps constant	$O(1)$	?
Insertion	Temps linéaire	$O(n)$	?
Suppression	Temps linéaire	$O(n)$	$O(n)$ ou ?
modification	Temps constant	$O(1)$	$O(1)$
recherche	$O(?)$		$O(?)$

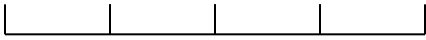



# Coût de l'insertion dans un tableau dynamique

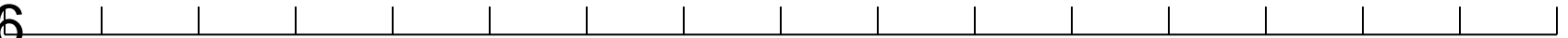
- Stratégies d'insertion
  - Doubler la taille du tableau
  - Augmenter le tableau d'une taille constante

$2^0=1$  

$2^1=2$  

$2^2=4$  

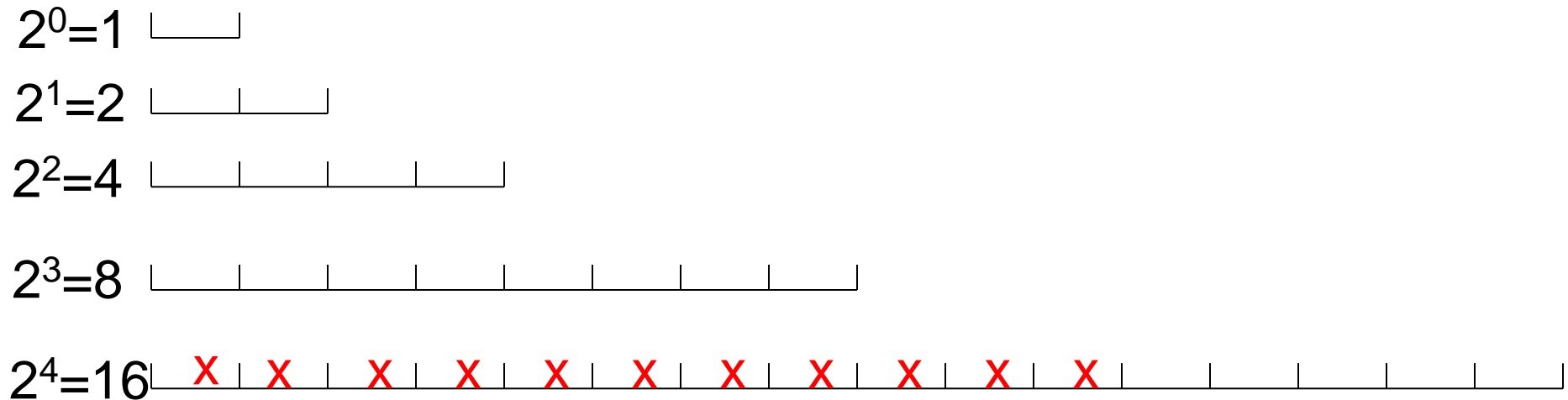
$2^3=8$  

$2^4=16$  

# Coût de l'insertion dans un tableau dynamique

$$2^{p-1} < n < 2^p < 2n$$

$$\text{Coût total} = n + \sum_{k=0}^{p-1} 2^k < n + 2^p < 3n$$



# Coût de l'insertion dans un tableau dynamique

$$\text{Coût amorti} = \frac{\text{Coût total}}{n} \leq \frac{3n}{n} = 3$$