UNIVERSITÉ PARIS DESCARTES

T04 – Définition d'un type abstrait de données
T05 – Exemple du TAD de Pile
T06 – TAD de Pile, définition abstraite
T08 – Notion de contrat en programmation
T09 – Contrôle des pré et postconditions
T10 – Donnée concrète
T11 à T12– Compilation séparée
T13 à T16– Exemple, composant de date
T17 – Primitives de Pile
T18 – Donnée concrète de Pile
T19 à T23– Corps du composant (Pile.cpp)
T24 – Graphe de dépendance
Codage d'un composant de Pile
T25 à T27 – Entête
T28 à T30 – Corps
T31 à T33 – Test

SDA - Structures de Données et Algorithmes

Equipe pédagogique
Marie-José Caraty, Denis Poitrenaud, Julien Rossit,
Camille Kurtz, Jacques Alès-Bianchetti, Eloi Keita

Cours n° 4

Type abstrait de données (TAD)
La pile - définition abstraite et implémentation

Bibliographie
B. Stroustrup, Le langage C++, Campus Press.
C. Carrez, Structures de données en Java, C++ et Ada, Dunod.

Sommaire

1. TYPE ABSTRAIT DE DONNEES

Introduction à l'abstraction de données

Définition d'une structure de données
Composition de données unies par une même sémantique décrivant un ensemble de données (**type**)

Concept d'abstraction de données [Hoare, années 70]
Extension de la définition d'un **type** de données aux **opérations** sur ces données **indépendamment** de **l'implémentation** de la structure de données (i.e., choix de la représentation des données en mémoire)

Types abstraits algébriques [théorie du milieu des années 70]
Permet une description des types de données en terme d'abstraction et un **support de preuve** de la validité de leurs opérations

Concept objet
Données et méthodes (fonctions) opérant sur la/les donnée(s) sont **réunies** dans une même **entité logicielle** (classe) avec le principe d'**encapsulation des données** (accessibles et/ou modifiables par des méthodes)

1. TYPE ABSTRAIT DE DONNEES

Définition d'un Type Abstrait de Données (TAD)

Définition/signature d'un type abstrait de données
- une déclaration des TAD définis et utilisés ainsi que des constantes utilisées
- une description fonctionnelle des opérations
- une description axiomatique de la sémantique des opérations
  i) préconditions sur les opérations et ii) axiomatique (support de preuve)

Indépendante de l'implémentation[1] du TAD
- de la représentation des données en mémoire
  choix de la structure de données dite « donnée concrète »
- de l'implémentation des opérations associées

Implémentation du TAD
Choix d'une donnée concrète dans un langage donné (C++, Java, …)
Critères de la qualité de l'implémentation
- efficacité de l'accès aux données
- simplicité

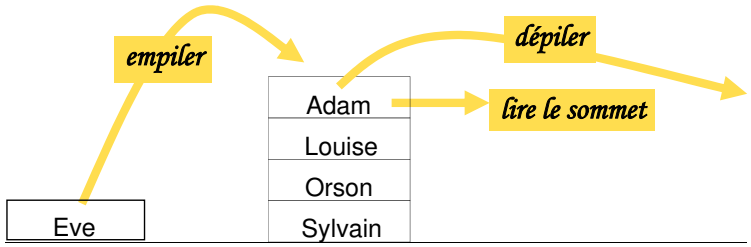
*Note : [1]**Implémentation d'une entité** (donnée, fonction, …)*
Façon dont est programmée l'entité dans un langage donné

Exemple du TAD de pile – Introduction

Définition d'une séquence S = <e_1 e_2 … e_n > avec card (S) = n
Opération successeur définie (précondition : l'élément n'est pas le dernier élément)
Spécification du type d'accès (séquentiel ou direct)
et de l'exploitation d'une relation d'ordre entre les éléments

Une pile est une séquence d'éléments accessibles par une seule extrémité appelée le sommet

Accès séquentiel à un élément, sans exploitation de relation d'ordre entre éléments
Modèle LIFO (Last In, First Out) : dernier entré, premier sorti



Une pile de quatre noms

Remarque : Dans l'axiomatique, les contraintes informatiques ne sont pas prises en compte

Exemple du TAD de pile – Définition abstraite

Type Abstrait de Données : Pile
Pile **utilise** E (l'ensemble des éléments de pile) et Booléen
pileVide ∈ Pile

Description fonctionnelle des opérations

push pop top

empiler :	Pile × E → Pile	Ajout d'un élément au sommet de pile
dépiler :	Pile → Pile	Suppression du sommet de pile
sommet :	Pile → E	Lecture du sommet de pile
estVide :	Pile → Booléen	Indicateur de pile vide

Description axiomatique de la pile
∀p ∈ Pile, ∀ e ∈ E

1) estVide (pileVide) = vrai
2) estVide (empiler (p, e)) = faux
3) sommet (empiler (p, e)) = e
4) dépiler (empiler (p, e)) = p
5) (⌉∃) p ∈ Pile / p = depiler (pileVide)
6) (⌉∃) e ∈ E / e = sommet (pileVide)

Préconditions des opérations
∀p ∈ Pile, ∀ e ∈ E
dépiler (p) : ⌉estVide (p)
sommet (p) : ⌉estvide (p)

Exemple du TAD de pile – Manipulation de la pile

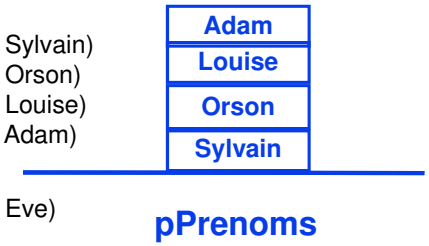
Déclarations d'utilisation de la pile pPrénoms

pPrenoms ∈ Pile avec E : l'ensemble des prénoms

Suite d'actions

1. pPrenoms ← pileVide
2. pPrenoms ← empiler (pPrenoms , Sylvain)
3. pPrenoms ← empiler (pPrenoms , Orson)
4. pPrenoms ← empiler (pPrenoms , Louise)
5. pPrenoms ← empiler (pPrenoms , Adam)
6. pPrenoms ← dépiler (pPrenoms)
7. pPrenoms ← empiler (pPrenoms , Eve)
8. pPrenoms ← dépiler (pPrenoms)
9. pPrenoms ← dépiler (pPrenoms)

e ∈ E, e ← sommet (pPrenoms)



Question : Quelle est la valeur de l'élément e ?
Dessiner l'évolution de la pile pPrenoms après chaque action

Notion de contrat en programmation

Contrat entre (1) le **concepteur/développeur** de la fonction
et (2) le **programmeur/utilisateur** de la fonction
- défini dans la documentation de la fonction (`@pre` et `@post`)
- contrôlé dans le code

Précondition *Rejoint les axiomes de précondition des TAD*
Domaine de validité des données d'entrée de la fonction (paramètres d'entrée)

Postcondition
Propriétés du traitement attendu (relation entre l'entrée et la sortie)

Nature du contrat
Si les données (fournies en entrée à la fonction par le code utilisateur) vérifient les préconditions
alors les postconditions (décrivant les propriétés du traitement attendu) sont vérifiées

Pour SDA , les postconditions ne sont pas requises
- Les préconditions (seules) devront être données dans la documentation
- Les préconditions devront être vérifiées dans le code par assertion

Postulat : si les données en entrée vérifient les préconditions, le résultat de la fonction est valide

Contrôle par assertions – Préconditions et postconditions

```
#include <cassert>     // en C++
#include <assert.h>    // en C

assert(<expression_booleenne>);
```

assert évalue son **argument**
Si le résultat est **false** (C++) ou **nul** (C)

- **affichage** (pour le débogage) au minimum
  - du nom du fichier source,
  - du numéro de la ligne de l'assertion
- **appel** de l'instruction **abort()** (sortie de programme immédiate)

Utilité des assertions pour vérifier les préconditions (et postconditions)
d'une fonction exprimées par des expressions booléennes
ou des fonctions indicatrices (retournant un booléen)

Remarque :
Un appel à **abort** est rarement acceptable dans du code de production
Le mécanisme d'exception en C++ (cf. cours CDL de 2ème année) permet d'éviter
l'abandon d'exécution (tout particulièrement indiqué pour les assertions de précondition)

Nombre rationnel : quotient de deux entiers relatifs

Choix d'une donnée concrète – Le type Rationnel (1/2)

```
struct Rationnel {
  …
  …
};
```

```
/* structure composée du
 * numérateur et du dénominateur */
struct Rationnel {
   int num;   // numérateur
   int den;   // dénominateur
};
```

```
/* structure composée du tableau
 * des 2 valeurs tab[0] et tab[1]
 * resp. numérateur et
 * dénominateur) */
struct Rationnel {
   int tab[2];
};
```

Exemple de deux données concrètes
structures de stockage possibles
d'un nombre rationnel en mémoire

```
Rationnel initialiser(int n, int d);
Rationnel additionner(const Rationnel& r1, const Rationnel& r2);
Rationnel multiplier(const Rationnel& r1, const Rationnel& r2);
//...
void afficher(const Rationnel& r);
```

Objectif :
Choisir la donnée concrète permettant
(1) la meilleure efficacité des opérations associées
(2) la meilleure lisibilité

Compilation séparée – Structuration des sources

La **compilation séparée** est **indispensable** au développement de logiciels professionnels (centaines de milliers/millions de lignes)

- Permet d'organiser le logiciel en plusieurs fichiers sources
  - **compilables** (temps acceptable de compilation)
  - **maîtrisables** (temps acceptable de correction des erreurs)
- Rapidité de compilation du logiciel
- Développement du logiciel en équipe (projet)

Tous les paradigmes de programmation « moderne »
– modulaire – orientée objet – orientée composant
tendent vers le concept d'abstraction des données (comme les TAD)
Regroupement logique des opérations/fonctions qui travaillent
sur une même structure de données
dans une même **unité logicielle compilable** (module, classe, composant)

En C++ et en C, l'unité logicielle **UL** est organisée en deux fichiers

- **interface** – un fichier **UL.h** (en C++ et en C) [entête (*header*), spécification]
- **implémentation** – fichier **UL.cpp** en C++ ou **UL.c** en C [source, corps]

Réutilisation des fonctions de l'UL dans un fichier compilable (e.g. main.cpp/main.c) :

```
#include "UL.h"
```


Compilation séparée – Structuration d'une unité logicielle

Structuration d'une unité logicielle (UL)

- **Un fichier de spécification** (**UL.h**)
  l'interface de UL (services offerts à un programmeur « utilisateur de l'UL ») contenant
  - déclarations de types (et de constantes)
  - déclarations de fonctions (services offerts à un programmeur « utilisateur »)
  - des directives de précompilation pour éviter les déclarations multiples (cause d'erreurs à la compilation)

```
#ifndef _UL_
#define _UL_

  // Interface de l'unité logicielle UL

#endif
```

Remarque : **Seule** l'interface **UL.h** doit être **connue d'un programmeur pour réutiliser les services** de l'unité logicielle **UL**
d'où la nécessité de sa documentation

- **Un fichier corps** (UL.cpp en C++, UL.c en C) contenant :
  - Définition (corps) de chacune des fonctions de UL.h

Compilation séparée – Spécification du type Date : Date.h (1/3)

A compléter : – initialisation d'une date (jour, mois, année a), – test de validité de date, – incrémentation de date (de n jours), ...

La documentation de réutilisation du composant est générée à partir de ce fichier

```
#ifndef _DATE_
#define _DATE_

/**
 * @file Date.h
 * @brief Entête du composant de date
 */
```

sem04-cours-Cpp1

Donnée concrète (structure de sockage d'une date)

```
/**
 * @brief Structure de données de type Date
 */
struct Date {
   unsigned short jour, mois, annee;
};
```

Interface du composant de date

```
/**
 * @brief Saisie d'une date
 * @return la date saisie
 */
 Date saisir();

/**
 * @brief Affichage d'une date
 * @param[in] d : la date à afficher
 */
void afficher(const Date& d);

#endif
```


Compilation séparée – Corps du type Date : Date.cpp (2/3)

Documentation de Date.cpp adaptée à celle de Date.h

```
/**
 * @file Date.cpp
 * @brief Corps du composant de date
 */
```

sem04-cours-Cpp1

```
#include <iostream>
using namespace std;

#include "Date.h"
```

Fichier où sont déclarés le type Date et les prototypes des fonctions (sur les dates)

définition des opérations de l'interface

```
/**
 * @brief Saisie d'une date
 * @return la date saisie
 */
Date saisir() {
   Date d;
   cout << "Date (jour? mois? annee?) ? ";
   cin >> d.jour >> d.mois >> d.annee;
   return d;
}
```

définition de l'opération (saisir)

```
/**
 * @brief Affiche une date
 * @param[in] d : la date à afficher
 */
void afficher(const Date& d) {
   cout << d.jour << '/' << d.mois << '/' << d.annee << "  ";
}
```


Compilation séparée – Utilisation du type Date : testDate.cpp (3/3)

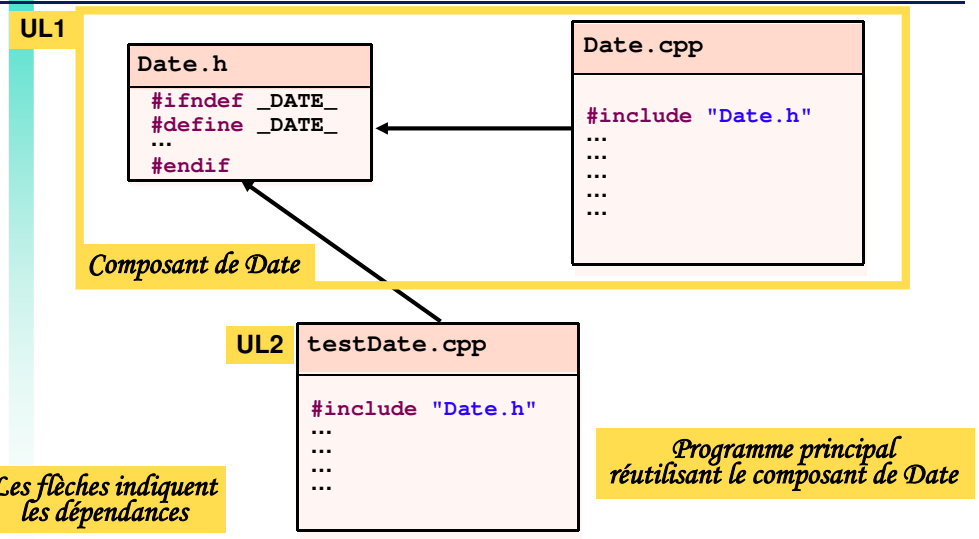
```
/**
 * @file testDate.cpp
 * @brief Test du composant de date
 */

#include <iostream>
using namespace std;

#include "Date.h"
```

Inclusion requise

```
int main() {
   Date d;

   cout << "Test du composant de date" << endl;

   cout << "Saisir des dates jusqu'à la saisie d'une année nulle"
        << endl;
   do {
      d = saisir();
      cout << "Date saisie : ";
      afficher(d); cout << endl;
   } while (d.annee!=0);

   cout << "Fin de saisie détectée." << endl;

   return 0;
}
```

```
Test du composant de date
Saisir des dates jusqu'à la saisie d'une année nulle
Date (jour? mois? annee?) ? 4 7 1783
Date saisie : 4/7/1783
Date (jour? mois? annee?) ? 14 7 1789
Date saisie : 14/7/1789
Date (jour? mois? annee?) ? 1 1 0
Date saisie : 1/1/0
Fin de saisie détectée.
```


Les différentes unités logicielles (UL)



cf. Annexe du programme documenté

Spécification du composant (de) Pile – Pile.h

Donnée concrète de Pile

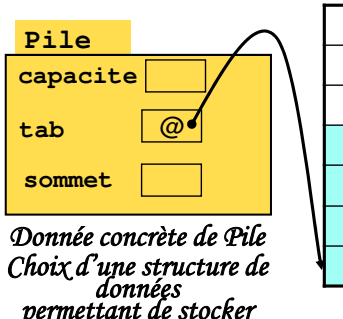
Opérations du composant Pile

```
#ifndef _PILE_
#define _PILE_

/**
 * @file Pile.h
 * @brief Composant de pile à capacité fixée
 */

#include "Item.h" // Généralisation de la pile à tout item

struct Pile {
   unsigned int capacite;  // capacité de la pile (c>0)
   Item* tab; // tableau des éléments de pile en mémoire dynamique
   int sommet; // indice de sommet de pile dans tab
};

void initialiser (Pile& p, unsigned int c);
void detruire(Pile& p);
bool estPleine(const Pile& p);
bool estVide(const Pile& p);
Item sommet(const Pile& p);
void empiler(Pile& p, const Item& it);
void depiler(Pile& p);

#endif
```

Opérations liées à l'implémentation du TAD Pile

Opérations du TAD Pile

Donnée concrète de Pile – Mode de représentation



Donnée concrète de Pile
Choix d'une structure de données permettant de stocker une pile à capacité paramétrée

Sémantique de la représentation
- Les éléments d'une pile sont empilés de l'indice 0 à sommet
- Convention de transcodage :
  si sommet = -1,
  alors la pile est vide
- Invariant de Pile
  -1 ≤ sommet ≤ (capacite -1)

Invariant de composant
Propriété toujours vraie avant ou après appel de toute fonction du composant

Objectif dans le codage de chacune des opérations
- gérer les attributs (`capacite`, `tab`, `sommet`) de la donnée concrète de Pile suivant la fonctionnalité de l'opération
- en cohérence avec les préconditions décrites en commentaire
- contrôler les préconditions par assertion

Corps du composant (de) Pile – Pile.cpp (1/5)

Simulation de la constante PileVide du TAD

```
/**
 * @brief Initialiser une pile vide
 * la pile est allouée en mémoire dynamique
 * @see detruire, pour une désallocation en fin d'utilisation
 * @param[out] p : la pile à initialiser
 * @param[in] c : capacité de la pile (nb maximum d'items stockés)
 * @pre c>0
 */
```

Algorithme
// Initialiser les attributs (capacité, sommet et tab) de p
// au valeurs d'une pile vide de capacité c
p.capacité ← c
allouer p.tab en mémoire dynamique : tableau de c éléments de type Item
p.sommet ← -1

```
void initialiser(Pile& p, unsigned int c) {
   assert(c>0);
   p.capacite = c;
   p.tab = new Item[c];
   p.sommet = -1;
}
```

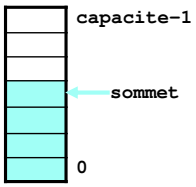

Corps du composant (de) Pile – Pile.cpp (2/5)

```
/**
 * @brief Désallouer une pile
 * @see initialiser, la pile a déjà été initialisée
 * @param[in,out] p : la pile à désallouer
 */
```

Algorithme Désallouer p.tab en mémoire dynamique

```
void detruire(Pile& p) {
   delete [] p.tab;
   p.tab = NULL;
}
```

```
/**
 * @brief Lire l'item au sommet de pile
 * @param[in] p : la pile
 * @return la valeur de l'item au sommet de pile
 * @pre la pile n'est pas vide
 */
```



Algorithme Retourner l'élément au sommet de la pile

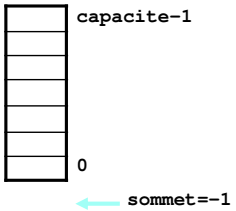
```
Item sommet(const Pile& p) {
   assert(!estVide(p));
   return p.tab[p.sommet];
}
```

Corps du composant (de) Pile – Pile.cpp (3/5)

```
/**
 * @brief Test de pile vide
 * @param[in] p : la pile testée
 * @return true si p est vide, false sinon
 */
```

Algorithme p est vide si le sommet vaut -1

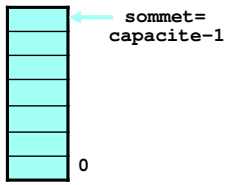
```
bool estVide(const Pile& p) {
    return (p.sommet == -1);
}
```




```
/**
 * @brief Test de pile pleine
 * @param[in] p : la pile testée
 * @return true si p est pleine, false sinon
 */
```

Algorithme p est pleine si le sommet vaut (capacite-1)

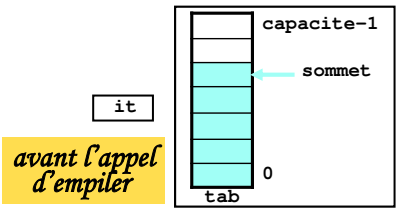
```
bool estPleine(const Pile& p) {
    return (p.sommet == (p.capacite-1));
}
```

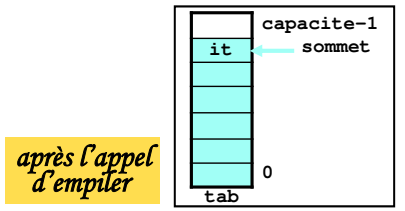


Corps du composant (de) Pile – Pile.cpp (4/5)

```
/**
 * @brief Empiler un item sur une pile
 * @param[in,out] p : la pile
 * @param[in] it : l'item à empiler
 * @pre p n'est pas pleine
 */
```

Algorithme incrémenter le sommet // mise à jour du sommet (indice de mémorisation de l'item it)
mémoriser it au sommet de la pile



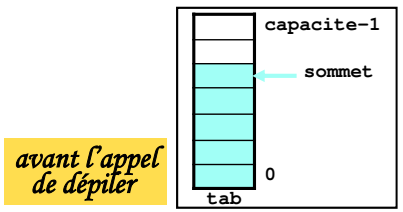


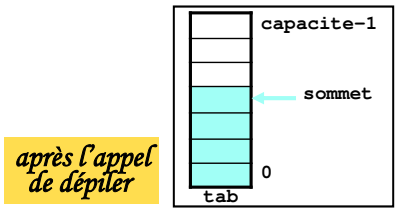

```
void empiler(Pile& p, const Item& it) {
    assert(!estPleine(p));
    p.sommet++;
    p.tab[p.sommet] = it;
}
```


Corps du composant (de) Pile – Pile.cpp (5/5)

```
/**
 * @brief Dépiler l'item au sommet de pile
 * @param[in,out] p : la pile
 * @pre p n'est pas vide
 */
```

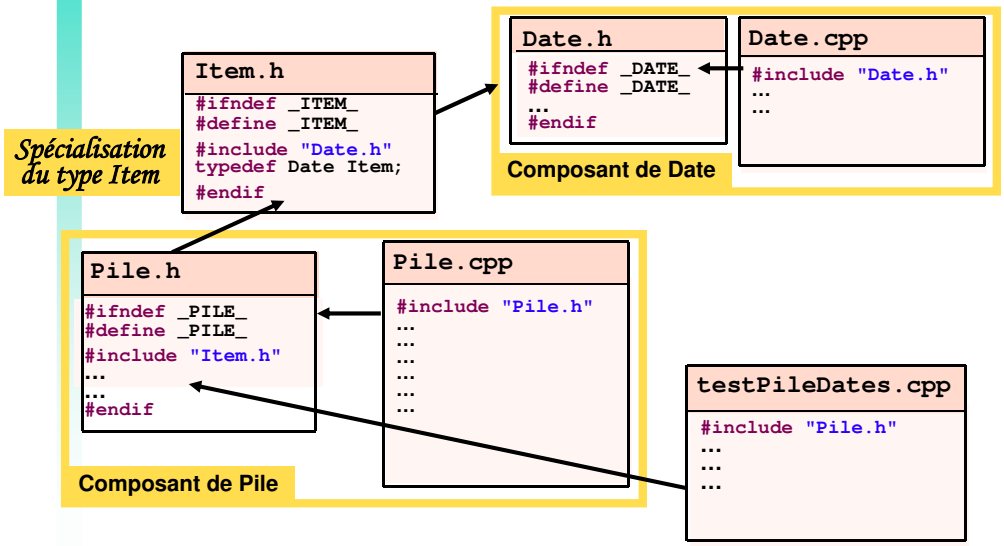
Algorithme décrémenter le sommet // mise à jour du sommet






```
void depiler(Pile& p) {
    assert(!estVide(p));
    p.sommet--;
}
```


Test d'une pile de dates - Graphe de dépendance



Annexe du programme documenté

Spécification du composant (de) Pile – Pile.h (1/3)

sem04-cours-Cpp2

```
#ifndef _PILE_
#define _PILE_

/**
 * @file Pile.h
 *
 * @brief Composant de pile à capacité fixée
 */

#include "Item.h"

struct Pile {
   unsigned int capacite;  // capacité de la pile (c>0)
   Item* tab;   // tableau des éléments de pile en mémoire dynamique
   int sommet;  // indice de sommet de pile dans tab
};

 /**
 * @brief Initialiser une pile vide
 * la pile est allouée en mémoire dynamique
 * @see detruire, pour une désallocation en fin d'utilisation
 * @param[out] p : la pile à initialiser
 * @param[in] c : capacité de la pile (nb maximum d'items stockés)
 * @pre C>0
 */
void initialiser(Pile& p, unsigned int c);

/**
 * @brief Désallouer une pile
 * @see initialiser, la pile a déjà été initialisée
 * @param[in,out] p : la pile à désallouer
 */
void detruire(Pile& p);
```


Annexe du programme documenté

Spécification du composant (de) Pile – Pile.h (2/3)

```
/**
 * @brief Test de pile pleine
 * @param[in] p : la pile testée
 * @return true si p est pleine, false sinon
 */
bool estPleine(const Pile& p);

/**
 * @brief Test de pile vide
 * @param[in] p : la pile testée
 * @return true si p est vide, false sinon
 */
bool estVide(const Pile& p);

/**
 * @brief Lire l'item au sommet de pile
 * @param[in] p : la pile
 * @return la valeur de l'item au sommet de pile
 * @pre la pile n'est pas vide
 */
Item sommet(const Pile& p);
```


Annexe du programme documenté

Spécification du composant (de) Pile – Pile.h (3/3)

```
/**
 * @brief Empiler un item sur une pile
 * @param[in,out] p : la pile
 * @param[in] it : l'item à empiler
 * @pre p n'est pas pleine
 */
void empiler(Pile& p, const Item& it);

/**
 * @brief Dépiler l'item au sommet de pile
 * @param[in,out] p : la pile
 * @pre p n'est pas vide
 */
void depiler(Pile& p);

#endif
```


Annexe du programme documenté

Corps du composant (de) Pile – Pile.cpp (1/3)

sem04-cours-Cpp2

```
/**
 * @file Pile.h
 *
 * @brief Composant de pile à capacité paramétrée
 */

#include <iostream>
#include <cassert>
#include "Pile.h"

 /**
 * @brief Initialiser une pile vide
 * la pile est allouée en mémoire dynamique
 * @see detruire, pour une désallocation en fin d'utilisation
 * @param[out] p : la pile à initialiser
 * @param[in] c : capacité de la pile (nb maximum d'items stockés)
 */
void initialiser(Pile& p, unsigned int c) {
   p.capacite = c;
   p.tab = new Item[c];
   p.sommet = -1;
}
/**
 * @brief Désallouer une pile
 * @see initialiser, la pile a déjà été initialisée
 * @param[in,out] p : la pile à désallouer
 */
void detruire(Pile& p) {
   delete [] p.tab;
   p.tab = NULL;
}
```

Annexe du programme documenté

Corps du composant (de) Pile – Pile.cpp (2/3)

```
/**
 * @brief Test de pile pleine
 * @param[in] p : la pile testée
 * @return true si p est pleine, false sinon
 */
bool estPleine(const Pile& p) {
   return (p.sommet == (p.capacite-1));
}

/**
 * @brief Test de pile vide
 * @param[in] p : la pile testée
 * @return true si p est vide, false sinon
 */
bool estVide(const Pile& p) {
   return (p.sommet == -1);
}
```


Annexe du programme documenté

Corps du composant (de) Pile – Pile.cpp (3/3)

```
/**
 * @brief Lire l'item au sommet de pile
 * @param[in] p : la pile
 * @return la valeur de l'item au sommet de pile
 * @pre la pile n'est pas vide
 */
Item sommet(const Pile& p) {
   assert(!estVide(p));
   return p.tab[p.sommet];
}
/**
 * @brief Empiler un item sur une pile
 * @param[in,out] p : la pile
 * @param[in] it : l'item à empiler
 * @pre p n'est pas pleine
 */
void empiler(Pile& p, const Item& it) {
   assert(!estPleine(p));
   p.sommet++;
   p.tab[p.sommet] = it;
}
/**
 * @brief Dépiler l'item au sommet de pile
 * @param[in,out] p : la pile
 * @pre p n'est pas vide
 */
void depiler(Pile& p) {
   assert(!estVide(p));
   p.sommet--;
}
```


Annexe du programme documenté

Test du composant (de) Pile – testPileDates.cpp (1/2)

```
/**
 * @file testPileDates.cpp
 * …
 * @brief Test du composant de pile
 */

#include <iostream>
using namespace std;

#include "Pile.h"

/* Test d'une pile d'éléments de type Date */
int main() {

   Pile pDates; // Déclaration de la pile de dates
   Date d;

   /* Initialisation de la pile */
   initialiser(pDates, 3);

   cout << "Test de pile de dates" << endl;

   /* Test de pile vide */
   cout << "Début de l'application : ";
   if (estVide(pDates)) cout << "la pile est vide" << endl;
   else cout << "la pile n'est pas vide" << endl;
```

sem04-cours-Cpp2

Item.h

```
#ifndef _ITEM_
#define _ITEM_
#include "Date.h"
typedef Date Item;
#endif
```


Annexe du programme documenté

Test du composant (de) Pile – testPileDates.cpp (2/2)

```
   /* Ajout de dates dans la pile jusqu'à la saisie
    * d'une date d'année nulle (date non empilée) */
   cout << "Saisir des dates jusqu'à la saisie d'une année nulle\n";
   cout << "Les dates (à l'exception de celle de l'année nulle) \n ";
   cout << "seront ajoutées à la pile de capacité " << pDates.capacite;
   unsigned short nbSaisies =0;
   do {
      d = saisir();
      if (d.annee != 0) {
         empiler(pDates, d);
         nbSaisies++; //Compteur de dates stockées
      }
   } while (d.annee!=0);

   cout << "\nEtat de la pile après " << nbSaisies << " dates empilées : ";
   if (estVide(pDates)) cout << "la pile est vide" << endl;
   else cout << "la pile n'est pas vide" << endl;

   /* Dépiler la pile avec la trace des éléménts dépilés */
   cout << "Pile dépilée avec affichage des éléments dépilés" << endl;
   while (!estVide(pDates)) {
      cout << "Elément dépilé : ";
      afficher(sommet(pDates)); cout << endl; //affiche le sommet de pile
      depiler(pDates); // depile le sommet de pile
   }
   cout << "La pile est vide";

   detruire(pDates);

   return 0;
}
```

Annexe du programme documenté

Traces du test du composant (de) Pile – testPileDates.cpp

```
Test de pile de dates
Début de l'application : la pile est vide
Saisir des dates jusqu'à la saisie d'une année nulle
Les dates (à l'exception de celle de l'année nulle)
seront ajoutées à la pile de capacité 3
Date (jour? mois? annee?) ? 25 12 2009
Date (jour? mois? annee?) ? 26 12 2009
Date (jour? mois? annee?) ? 27 12 2009
Date (jour? mois? annee?) ? 28 12 2009
Assertion failed: !estPleine(p), file Pile.cpp, line 76

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

```
Test de pile de dates
Début de l'application : la pile est vide
Saisir des dates jusqu'à la saisie d'une année nulle
Les dates (à l'exception de celle de l'année nulle)
seront ajoutées à la pile de capacité 3
Date (jour? mois? annee?) ? 25 12 2009
Date (jour? mois? annee?) ? 26 12 2009
Date (jour? mois? annee?) ? 27 12 2009
Date (jour? mois? annee?) ? 28 12 0
Etat de la pile après 3 dates empilées : la pile n'est pas vide
Pile dépilée avec affichage des éléments dépilés
Elément dépilé : 27/12/2009
Elément dépilé : 26/12/2009
Elément dépilé : 25/12/2009
La pile est vide
```


Ce que vous avez appris aujourd'hui…

- La notion d'**abstraction des données** avec l'approche des **Types Abstraits de Données** (TAD)
- **Définition d'un TAD**
  L'exemple de la définition du TAD de pile
- **Implémentation d'un TAD**
  Comment implémenter un type abstrait de données
  - Choix d'une donnée concrète
  - Compilation séparée et principe de structuration des sources
  - Assertion pour le contrôle des préconditions

  Indépendance de la **définition** et de **l'implémentation** du TAD
- Un **exemple d'implémentation** du TAD de pile
- Un **exemple de réutilisation** du TAD de pile