Programmation Modulaire
Structures de données

Florent Hivert

Mél : `Florent.Hivert@lri.fr`
Adresse universelle : `http://www.lri.fr/~hivert`

1 Organiser ses données : les types
- Types de base
- Alias de types

2 Les types structurés
- Les paires
- Les `structs`

3 Types énumérés

4 Tableaux : tableau C, `array` et `vector`

5 Le mécano
- Combinaisons de tableaux, vecteurs et structures
- Compléments : fonctions et paramètres

6 Annexe : tableaux dynamiques

Plan

1 Organiser ses données : les types

2 Les types structurés

3 Types énumérés

4 Tableaux : tableau C, `array` et `vector`

5 Le mécano

6 Annexe : tableaux dynamiques

Organiser ses données

Au premier semestre, vous avez vu essentiellement des programmes qui travaillent avec des nombres. Dans beaucoup de programmes, on modélise des **objets compliqués ayant de nombreuses caractéristiques**. Voici quelques exemples :

- Une banque doit modéliser un compte en banque avec le journal de toutes les opérations
- Un simulateur du système solaire doit modéliser les différents corps célestes (position, vitesse, masse, aplatissement, axe et vitesse de rotation sur lui-même...)
- Un catalogue musical doit modéliser des disques (auteur, titre, année, genre musical) qui contiennent des chansons (titre, durée, sons)...

Pour ceci, le `C++` permet de **créer ses propres types de variables**.

Organiser ses données

Remarque

Créer de nouveaux types de variables devient indispensable en pratique quand on cherche à faire des programmes plus complexes :

On veut pouvoir écrire

```
affiche(etudiant);
```

plutôt que

```
affiche(nom, prenom,
        annee_naiss, mois_naiss, jour_naiss,
        numero_rue, nom_rue, code_postal, ville,
        note_info1, note_info2, note_info3, note_info4);
```


Types de données

Retenir

*Avoir choisi les **bons types de données** permet d'avoir un programme*

- *plus lisible car auto-documenté*
- *plus facile à maintenir*
- *souvent plus rapide, en tout cas plus facile à optimiser*

" I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. " — Linus Torvalds (creator of Linux) « J'affirme que l'on fait la différence entre un mauvais et un bon programmeur, selon qu'il considère comme plus important son code ou bien ses

Type de données et **objets**

Compléments

Dans beaucoup de langages modernes (C++, Java, Python, par exemple), on a décidé de placer les **données** (par opposition aux fonctions dans les langages procéduraux) au centre des programmes. C'est la notion de **programmation orientée objet**.

C'est un changement radical sur la manière de penser la programmation. On parle de **paradigme de programmation**.

Remarque

Dans un premier temps, nous n'allons pas faire de programmation objet.

On verra cela dans le deuxième chapitre. . .

Rappel : Types de données

Définition (Notion de type)

Système de typage *: ensemble de règles qui associent aux constructions d'un programme (variables, expressions, fonctions. . .) une propriété nommée* ***type*** *dans le but de* ***vérifier (partiellement) la cohérence des programmes****.*

En C++ :

- **typage statique** : le contrôle de type est effectué à la compilation (dans la plupart des cas).
- **typage explicite** : le type des éléments du programme doit être déclaré explicitement (mais il peut être déduit automatiquement dans certains cas).

Plan

1 Organiser ses données : les types
- Types de base
- Alias de types

2 Les types structurés

3 Types énumérés

4 Tableaux : tableau C, array et vector

5 Le mécano

6 Annexe : tableaux dynamiques

Types de base

Le C++ définit plusieurs types de base :

- les valeurs de vérité `bool`
- les entiers et leurs variantes :
  `char`, `int`, `short int`, `long int`, `signed`, `unsigned`...
- les nombres à virgule flottante : `float`, `double`
- les références & et les pointeurs * : voir l'UE Algorithmes et Structures de Données (ASD)

Remarque

Les types avancés de la bibliothèque standard `string`, `array`, `vector` et autres conteneurs, sont des **types composés** qui sont définis à partir des types simples ci-dessus, et qui utilisent le paradigme de la **programmation objet**. D'où la notation particulière pour les appels de méthode (par ex. `v.size()`).

Conversion de type

Retenir

En général, une affectation `a = b;` *n'a pas de sens si* `a` *et* `b` *n'ont pas le même type. Dans certains cas, C++ fait une* ***conversion implicite*** *(on dit aussi coercion) :*

- `int` ↦ `float`
- `float` ↦ `int` *(arrondi vers 0)*
- `bool` ↦ `int` *(*`false` → *0,* `true` → *1)*
- `int` ↦ `bool` *(0 →* `false`*, ≠ 0 →* `true`*).*

On peut faire une ***conversion explicite*** *avec*

`nom_du_type(b)`

Le compilateur signale une erreur si la conversion est impossible.

Note : Le C++ recommande `static_cast<nom_du_type>(b)`.

Exemple de coercions

coercions.cpp

```
1   int i;
2   float f;
3   bool b;
4
5   i = 5; f = i;
6   cout << i << " " << f << endl;   // affiche  5 5
7
8   f = 6.79; i = f;
9   cout << f << " " << i << endl;   // affiche  6.79 6
10
11  b = true; i = b;
12  cout << b << " " << i << endl;   // affiche  1 1
13
14  i = 5; b = i;
15  cout << i << " " << b << endl;   // affiche  5 1
```

Exemple de conversion explicite

convexpl.cpp

```
1  int main() {
2    int i, j, k;
3
4    i = 1; j = 2;
5    k = i / j;
6    cout << 1/2 << " " << k << endl;   // affiche  0 0
7
8    float f;
9    f = i / j;
10   cout << f << endl;   // affiche  0
11
12   f = float(i) / j;
13   cout << f << endl;   // affiche  0.5
14 }
```


Piège des divisions entières

Attention

La division entre deux entiers est une division entière (même si le résultat est mis dans un `float`).

Exemple :

```
float f = 1 / 3;     //  f recoit la valeur 0
```

Plusieurs possibilités :

```
float f = 1. / 3;              float f = 1 / float(3);
float f = 1 / 3.;
```

Retenir

Si `float`, il vaut mieux mettre systématiquement le «.».

```
float f = 1. / 3.;
```


Plan

1 Organiser ses données : les types
- Types de base
- Alias de types

2 Les types structurés

3 Types énumérés

4 Tableaux : tableau C, array et vector

5 Le mécano

6 Annexe : tableaux dynamiques

Alias de types

Pour les besoins d'auto-documentation du programme, on peut donner un **nom nouveau** à un type préexistant.

Syntaxe

- *Syntaxe C classique (encore très souvent utilisée) :*

```
typedef float masse;   // exprimée en Kg
```

- *Syntaxe C++ 2011 :*

```
using longueur = float;   // exprimée en mètre
```

Uniquement documentation, pas de contrôles supplémentaires :

```
1  masse m;
2  longueur l;
3  l = m;   // valide, les deux sont des floats
```

Unités physiques avec contrôle

Compléments

Il est possible d'écrire de l'**analyse dimensionnelle** en `C++` en utilisant les *templates* (patrons) et la **méta-programmation** (méthode de programmation qui consiste à faire des calculs qui vont eux-mêmes écrire des programmes pendant la compilation). Il faut en effet avoir des types qui sont capables d'effectuer les calculs d'analyse dimensionnelle à la compilation.

Il en existe une implantation (relativement standard) dans la bibliothèque `units` de la collection de bibliothèques `boost` (voir `http://www.boost.org`).

Plan

1 Organiser ses données : les types
2 Les types structurés
3 Types énumérés
4 Tableaux : tableau C, array et vector
5 Le mécano
6 Annexe : tableaux dynamiques

Structures

Quand plusieurs variables modélisent différents aspects d'un même objet du monde réel, il est très pratique de les regrouper dans un nouveau type de variables.

Retenir

Selon les langages de programmation, on appelle ces types ***structure, produit (product), enregistrement (record)****.*

Les ***composants*** *d'une structure s'appellent des* ***champs*** *et chaque champ possède un* ***nom et un type*** *(simple ou composé).*

Il peut y avoir autant de champs que l'on veut, et les types des différents champs peuvent être égaux ou différents.

Plan

1 Organiser ses données : les types
2 Les types structurés
- Les paires
- Les structs
3 Types énumérés
4 Tableaux : tableau C, array et vector
5 Le mécano
6 Annexe : tableaux dynamiques

Les paires

Le C++ fournit un type composite élémentaire : la paire. C'est utile quand une fonction doit retourner deux résultats.

Syntaxe

Déclaration (il faut inclure le fichier `<utility>`) :

```
std::pair<type1, type2> nom_de_variable;
```

Construction :

```
std::make_pair(val1, val2);
```

On extrait les deux composantes avec :

```
nom_de_variable.first     nom_de_variable.second
```


Les paires : exemple

Fonction qui renvoie deux valeurs :

estpuiss2-pair.cpp

```
/** Teste si un entier est une puissance de 2
 ** et calcule l'exposant.
 * @param[in]  n un nombre entier positif
 * @return une paire (b, e) où
 *        b : booleen
 *        e : exposant de la plus petite
 *            puissance de 2 supérieure ou égale à n
 **/
pair<bool, int> estPuissanceDe2(int n);
```

Remarque

On peut aussi faire ça avec un passage de paramètre par référence (voir l'UE Algorithmes et Structures de Données).

Les paires : exemple

estpuiss2-pair.cpp

```
1 pair<bool, int> estPuissanceDe2(int n) {
2   int exp = 0;
3   int puiss = 1;
4   while (puiss < n) {
5     exp++;
6     puiss *= 2;
7   }
8   return make_pair(puiss == n, exp);
9 }
```

estpuiss2-pair.cpp

```
1   pair<bool, int> p = estPuissanceDe2(a);
2   // Le booléen b est affiché avec 0 pour false et 1 pour true
3   cout << p.first << " " << p.second << endl;
```


Les paires : exemple (2)

La fonction standard `minmax` retourne une paire

minmax.cpp

```
1  // minmax example
2  #include <iostream>     // std::cout
3  #include <algorithm>    // std::minmax
4  using namespace std;
5
6  int main () {
7    auto result = minmax({1,2,3,4,5});
8
9    cout << "minmax({1,2,3,4,5}): ";
10   cout << result.first << ' ' << result.second << endl;
11   return 0;
12 }
```

Complément : les tuples, extraction

Compléments

- Le C++ définit un type `tuple` si l'on veut plus de deux champs.
- On extrait le *i*-ème champ avec `get<i>(t)`
- On peut extraire d'un coup tous les champs d'un tuple dans des variables `a,b,c,...` (précédemment déclarées) avec

  `tie(a, b, c, ...) = mon_tuple;`
- si l'on veut ignorer l'un des champs, on le remplace par `ignore`

Complément : les tuples, extraction

tuple.cpp

```
1  // Déclaration et initialisation directe d'un tuple:
2  tuple<int,char> foo(10, 'x');
3  // Si l'on ne veut pas (flemme, lisibilité) écrire les types
4  // on peut demander au compilateur de les déduire !
5  // Déclaration et initialisation avec déduction des types
6  auto bar = make_tuple("test", 3.1, 14, 'y');
7
8  cout << get<1>(foo) << endl; // accès en lecture, affiche 'x'
9  get<2>(bar) = 1415;  // accès en écriture
10
11 int myint; char mychar;
12 // Même effet que myint = get<0>(foo); mychar = get<1>(foo);
13 tie(myint, mychar) = foo;
14 // On ignore les deux premiers champs dans l'extraction:
15 // Même effet que myint = get<2>(bar); mychar = get<3>(bar);
16 tie(ignore, ignore, myint, mychar) = bar;
```


Plan

1 Organiser ses données : les types

2 Les types structurés
- Les paires
- Les `structs`

3 Types énumérés

4 Tableaux : tableau C, `array` et `vector`

5 Le mécano

6 Annexe : tableaux dynamiques

Pourquoi les structures ?

Remarque

On préfère souvent donner des noms plus pratiques que `first`, `second` ou des numéros aux différents champs.

- pas besoin de retenir l'ordre
- beaucoup plus **lisible**

Définition d'un type structure

Syntaxe

Définition d'un type structure :

```
struct NomDuType {
  type1 champ1;
  type2 champ2;
  ...
  typeN champN;
};
```

Après une telle définition chaque variable de type `NomDuType` aura *N* champs, chacun du type indiqué.

Exemples de définition de type structure

Durée en heures, minutes et secondes :

```
1  struct DureeHMS {
2    int heure;
3    int minute;
4    float seconde;
5  };
```

Nombre complexe :

```
1  struct Complexe {
2    float part_re, part_im;     // Syntaxe abrégée.
3  };
```


Exemples de définition de type structure (2)

Attention

Ne pas oublier le «;» à la fin de la définition.

Sinon le compilateur pense que l'on est en train de déclarer une variable dont le type est la structure :

```
error: expected ';' after struct definition
 }
```

Selon ce qui suit la définition, le message d'erreur peut être plus ou moins clair :

```
error: expected initializer before 'd'
 date d;
```


Un premier LEGO®

Retenir

Un champ d'une structure peut lui aussi être une structure !

Type «date» composé d'un jour, un mois, une année

```
1  struct Date {
2    int jour, mois, annee;
3  };
```

Type «assuré social»

```
1  struct AssureSocial {
2    string nom, prenom, Nsecu;
3    Date date_naissance;
4  };
```

Accès aux champs d'une structure

Syntaxe

*La **notation pointée** permet d'accéder (en lecture et en écriture) aux **champs d'une struct** :*

```
nom_de_la_variable.nom_du_champ
```

Par exemple, pour leur affecter une valeur :

```
1    Date aujourdhui;
2    aujourdhui.mois = 2;
3    aujourdhui.annee = 2020;
```

Structures imbriquées :

```
1    AssureSocial a;
2    if (a.date_naissance.annee >= 2003)
3      cout << a.prenom << " " << a.nom << " est mineur";
```


Initialisation d'une struct

Syntaxe

En C++, on peut initialiser une `struct` avec les accolades :

```
{ val1, val2, ..., valN }
```

dans ce cas les champs sont initialisés dans l'ordre.

Exemple, pour la structure :

date.cpp

```
struct Date {
  int jour, mois, annee;
};
```

Décl. et init. d'une variable :

date.cpp

```
Date d = {3, 11, 1974};
```

Dans un return :

date.cpp

```
Date bugday() {
  return {1, 1, 2000};
}
```


Manipulations des structures

Retenir

*À part l'extraction des champs, les **seules manipulations possibles** sur une variable de type structuré sont*

- *l'**affectation à une structure du même type** (tous les champs sont recopiés)*
- *le **calcul de l'adresse** (adresse du premier champ)*

Note : Le calcul d'adresse sera utile par exemple pour le passage par référence (voir l'UE ASD).

Affectation et retour de structures

L'affectation des structures est possible !

date.cpp

```
1  Date d = {3, 11, 1974};
2
3  Date aujourdhui;
4  aujourdhui = lireDate();
```

Une fonction peut donc retourner une structure :

date.cpp

```
1  /** Le bug de l'an 2000
2  * @return le premier janvier 2000
3  **/
4  Date bugday() {
5    return {1, 1, 2000};
6  }
```

Affectation et retour de structures (2)

Retenir

L'affectation de structures ***recopie tous les champs****, ce qui peut être très coûteux s'il y en a beaucoup.*

Compléments

Dans le cas où l'on renvoie une structure déclarée dans une fonction, la plupart des compilateurs modernes savent optimiser la copie de la structure (**Return Value Optimisation**, nombreux cas garantis depuis C++17) !

Manipulations des structures

Attention

On ne peut pas lire ou afficher une structure directement (voir cours suivant à propos de la surcharge).

Le message d'erreur n'est pas toujours très lisible. Le compilateur essaye plein de fonctions (avec des coercions) pour l'affichage :

```
date.cpp:107:8: error: invalid operands to binary expression
    ('std::ostream' (aka 'basic_ostream<char>') and 'Date')
cout << d;
~~~~ ^  ~
```

Ensuite il donne la liste (très longue) de toutes les conversions qu'il a essayées.

Note : voir cours suivant pour la **surcharge d'opérateur**.

Exemple : Affichage d'une date

date.cpp

```
1  struct Date {
2    int jour, mois, annee;
3  };
4
5  /** Affiche une Date sous le format jj/mm/aaaa
6  * @param[in] d : la date
7  **/
8  void afficheDate(Date d) {
9    cout << setfill('0') << setw(2) << d.jour << "/"
10        << setfill('0') << setw(2) << d.mois << "/" << d.annee;
11 }
```


Exemple de saisie d'une date

date.cpp

```
1  /** Demande une Date à l'utilisateur
2  *   Si la date n'est pas correcte, une nouvelle date est demandée
3  * @return une Date
4  **/
5  Date lireDate() {
6    Date res;
7    bool erreur;
8    do {
9      cout << "jour ? "; cin >> res.jour;
10     cout << "mois ? "; cin >> res.mois;
11     cout << "annee ? "; cin >> res.annee;
12     erreur = not estCorrecteDate(res);
13     if (erreur) cout << "Date incorrecte !" << endl;
14   } while (erreur);
15   return res;
16 }
```

date.cpp

```
1  /** Teste si une année est bissextile
2  * @param[in] annee : un entier
3  * @return le booléen correspondant au test
4  **/
5  bool estBissextile(int annee) {
6    return (annee % 4 == 0 and annee % 100 != 0) or (annee % 400 == 0);
7  }
8
9  /** Le nombre de jours qu'il y a dans un mois
10 * @param[in] annee : un entier
11 * @param[in] mois : un entier entre 1 et 12
12 * @return le nombre de jours du mois
13 **/
14 int nbJourMois(int mois, int annee) {
15   switch (mois) {
16   case 1 : case 3 : case 5 :
17   case 7 : case 8 : case 10 : case 12 : return 31;
18   case 4 : case 6 : case 9 : case 11  : return 30;
19   case 2 :
20     if (estBissextile(annee)) return 29;
21     else                      return 28;
22   default: return -1;
23   }
24 }
25
26 /** Teste si une Date est correcte
27 * @param[in] d : une date
28 * @return le booléen correspondant au test
29 **/
30 bool estCorrecteDate(Date d) {
31   if (d.mois <= 0 or d.mois > 12) return false;
32   if (d.jour <= 0) return false;
33   return d.jour <= nbJourMois(d.mois, d.annee);
34 }
```


Deux manières de renvoyer une date au programme

date.cpp

```
1  /** Le bug de l'an 2000
2  * @return le premier janvier 2000
3  **/
4  Date bugday() {
5    return {1, 1, 2000};
6  }
7
8  /** Le bug de l'an 2000
9  * @param[out] d reçoit le bug day
10 **/
11 void bugdayRef(Date &d) {
12   d = {1, 1, 2000};
13 }
```

Voir l'UE ASD pour l'explication de la deuxième méthode.

Manipulation d'une date

date.cpp

```
1  /** Lendemain d'une Date
2  *  @param[in] d : une date
3  **/
4  Date lendemain(Date d) {
5    d.jour++;
6    if (d.jour > nbJourMois(d.mois, d.annee)) {
7      d.jour = 1;
8      d.mois++;
9      if (d.mois == 13) {
10       d.mois = 1;
11       d.annee++;
12     }
13   }
14   return d;
15 }
```


Stockage en mémoire d'une structure

Retenir

Le compilateur réserve les emplacements nécessaires pour stocker les différents champs de la structure. Ils correspondent à des emplacements mémoires consécutifs différents.

<table>
<tr><td rowspan="6">a</td><td colspan="2">nom</td><td>"John"</td></tr>
<tr><td colspan="2">prenom</td><td>"Doe"</td></tr>
<tr><td colspan="2">Nsecu</td><td>"1 74 34 234 123"</td></tr>
<tr><td rowspan="3">date_naissance</td><td>annee</td><td>1974</td></tr>
<tr><td>mois</td><td>11</td></tr>
<tr><td>jour</td><td>3</td></tr>
</table>

Note : dans la réalité, le type `string` est lui aussi composé et comporte plusieurs sous-variables (c'est un objet).

Référence vers un objet constant

Dans le cas d'un passage par valeur, la copie peut être coûteuse si l'objet passé est complexe (structure, vecteur...).

Compléments

Dans certains cas, on fait un **passage par référence pour éviter la copie**. Le C++ permet alors d'**interdire la modification** en déclarant l'objet passé par référence constant grâce au mot clé `const`.

Voir le cours d'ASD pour l'explication du passage par référence.

```
1    void affiche_etudiant(const Etudiant &etudiant);
2
3    float moyenne(const vector<float> &notes);
```


Plan

1 Organiser ses données : les types
2 Les types structurés
3 Types énumérés
4 Tableaux : tableau C, array et vector
5 Le mécano
6 Annexe : tableaux dynamiques

Types énumérés

Syntaxe

*Un **type énuméré** est un type **dont on connaît l'ensemble des valeurs**. On le déclare par la commande*

```
enum class Nom {val1, val2, ...};
```

Les valeurs sont accessibles par

```
Nom::valn
```


Exemple de types énumérés

```
1  enum class Jour {lundi, mardi, mercredi, jeudi,
2                   vendredi, samedi, dimanche};
3
4  Jour j;
5  ...
6  j = Jour::mardi;
7  if (j == Jour::samedi) ...;
```

Manipulations des types énumérés

Attention

Comme les structures, on ne peut pas lire ou afficher un type énuméré directement (voir cours suivant à propos de la surcharge).

```
Jour j = Jour::lundi;
cout << j;
```

Fait une erreur comme dans le cas des structures :

```
enum.cpp:85:8: error: invalid operands to binary expression
('std::ostream' (aka 'basic_ostream<char>') and 'Jour')
  cout << j << endl;
  ~~~~ ^  ~
```

Ici aussi, le message d'erreur n'est pas très lisible, car le compilateur essaye plein de fonctions et donne la liste de toutes ce qu'il a essayées.

Type énuméré et chaîne de caractères

Attention

Les valeurs (`Jour::lundi`, `Jour::mardi`...) que l'on vient de définir sont des **constantes d'énumération**. Leur type est le type `Jour` que l'on a défini et non des chaînes de caractères (`string`).

Si on veut obtenir une chaîne de caractère (par exemple pour affichage), il faut écrire explicitement la fonction de conversion.

L'instruction `switch` est alors très pratique : le compilateur vérifie que l'on n'a pas oublié de cas, et sinon affiche un message d'avertissement : `warning: enumeration value 'Jour::samedi' not handled in switch`

Parenthèse : le switch

Objectif : Exécution de code selon la valeur d'un entier, d'un caractère ou d'un type énuméré (mais pas `float` ou `string`).

Syntaxe

```
switch (val) {
  case val1 : ...
  case val2 : ...
  ...
  [default: ...]
}
```

- Selon la valeur de `val`, **saute au cas correspondant**.
- Si l'on ne veut pas que l'exécution continue dans les autres cas, mettre une instruction `break;` **pour sortir du** `switch`.
- optionnellement mettre un cas `default`.

Exemple de switch (cas d'un entier)

switch.cpp

```
1  switch (i) {
2    case 1: cout << "cas 1"; break;
3    case 2: cout << "cas 2"; // pas de break;
4    case 3: cout << "cas 3"; break;
5    case 4: case 5: case 6:
6      cout << "cas 4, 5 ou 6"; break;
7    default : cout << "autre cas";
8  }
9  cout << endl;
```

- Si `i` vaut 1 affiche «cas 1»
- Si `i` vaut 3 affiche «cas 3»
- Si `i` vaut 2 affiche «cas 2cas 3»
- Si `i` vaut 9 affiche «autre cas»

Exemple d'affichage d'un type énuméré avec switch

enum-class.cpp

```
33 string jour_string_switch(Jour j) {
34   string res;
35   switch (j) {
36   case Jour::lundi:    res = "lundi"; break;
37   case Jour::mardi:    res = "mardi"; break;
38   case Jour::mercredi: res = "mercredi"; break;
39   case Jour::jeudi:    res = "jeudi"; break;
40   case Jour::vendredi: res = "vendredi"; break;
41   case Jour::samedi:   res = "samedi"; break;
42   case Jour::dimanche: res = "dimanche"; break;
43   }
44   return res;
45 }

85   Jour j = Jour::lundi;
86   cout << jour_string_switch(j) << endl;
```


Exemple de saisie d'un type énuméré (cas d'un caractère)

enum-class.cpp

```
49 Jour saisie_jour_switch() {
50   Jour res; bool ok;
51   do {
52     char c; ok = true;
53     cout << "Quel jour (l)undi, m(a)rdi, m(e)rcredi, (j)eudi, (v)endredi, "
54             "(s)amedi, (d)imanche ? ";
55     cin >> c;
56     switch (c) {
57     case 'L': case 'l': res = Jour::lundi; break;
58     case 'A': case 'a': res = Jour::mardi; break;
59     case 'E': case 'e': res = Jour::mercredi; break;
60     case 'J': case 'j': res = Jour::jeudi; break;
61     case 'V': case 'v': res = Jour::vendredi; break;
62     case 'S': case 's': res = Jour::samedi; break;
63     case 'D': case 'd': res = Jour::dimanche; break;
64     default: ok = false;
65     }
66   } while (not ok);
67   return res;
68 }
```


Codage des types énumérés

Rappel : l'affichage direct fait une erreur :

```
Jour j = Jour::lundi;
cout << j;    // Erreur (invalid operands)
```

Retenir

*Les valeurs d'un type énuméré sont en fait **codées par des entiers**. En C++, on obtient l'entier avec une **conversion explicite**.*

```
Jour j = Jour::lundi;
cout << int(j);  // affiche le code de lundi : 0
```


Codage des types énumérés

Attention

La conversion dans l'autre sens (entier $\mapsto$ type énuméré) est possible. Il faut d'abord avoir **vérifié** que l'entier est bien le **code d'une valeur du type énuméré**.

```
Jour j = Jour(0);  // lundi
```

Affichage et saisie type énuméré (variante sans switch)

Affichage par conversion en chaîne de caractères :

enum-class.cpp

```
14 // Mieux : utiliser un array (voir plus loin dans le cours)
15 const vector<string> nom_du_jour { { "lundi", "mardi",
16         "mercredi", "jeudi", "vendredi", "samedi", "dimanche" } };
17
18 string jour_string(Jour j) { return nom_du_jour[int(j)]; }
```

Saisie :

enum-class.cpp

```
22 Jour saisie_jour() {
23   int res;
24   do {
25     cout << "Quel jour (0=lundi, 6=dimanche) ? ";
26     cin >> res;
27   } while (res < int(Jour::lundi) or res > int(Jour::dimanche));
28   return Jour(res);
29 }
```


Boucle sur un type énuméré

Retenir

Pour faire une boucle sur un type énuméré, on boucle sur les entiers et on convertit.

Voici par exemple comment afficher les jours de la semaine :

enum-class.cpp

```
73 void affiche_semaine() {
74   for (int i=0; i < 7; i++) {
75     Jour j = Jour(i);
76     cout << jour_string(j) << " ";
77   }
78   cout << endl;
79 }
```


Type énumérés et comparaisons

Retenir

*Les **comparaisons** valeurs d'un type énuméré sont possibles.*

*Les valeurs d'un type énuméré sont ordonnées dans l'**ordre de la déclaration** et peuvent être comparées avec les opérateurs d'inégalité (par exemple <=, >).*
En conséquence, on peut aussi utiliser les fonctions standards `min`, `max`, `minmax` *(déclarées dans* `<algorithm>`*).*

enum-class.cpp

```
98    // Affiche 0 pour faux
99    cout << (Jour::lundi == Jour::jeudi) << endl;
100   // Affiche 1 pour vrai
101   cout << (Jour::lundi <= Jour::jeudi) << endl;
102
103   // Affiche jeudi
104   cout << jour_string(max(Jour::lundi, Jour::jeudi)) << endl;
```


Type énuméré à la C

Compléments

On trouve souvent les anciens types énumérés qui sont déclarés sans le mot clé `class` :

enum.cpp

```
enum jour_semaine {lundi, mardi, mercredi, jeudi,
                   vendredi, samedi, dimanche};
```

Très courant dans les codes, mais déconseillé car

- pollution de l'espace de nom global avec les constantes
- conversion implicite vers les entiers

Exemple :

```
1   jour_semaine j = lundi;
2   if (j == true) { ... }  // compile car les deux sont convertis
3                           // implicitement en entier !!!
```

Plan

1 Organiser ses données : les types

2 Les types structurés

3 Types énumérés

4 Tableaux : tableau C, `array` et `vector`

5 Le mécano

6 Annexe : tableaux dynamiques

Tableaux

Rappel : un tableau est une valeur composite formée de plusieurs valeurs du même type.

Attention

Il y a plusieurs types de «tableaux» en `C++`. Pour faire la différence, ceux que vous avez vus au premier semestre seront appelés **vecteurs**.

Rappel sur les vecteurs

Retenir (Les trois étapes pour utiliser un vecteur)

- *Un **vecteur** se construit en trois étapes :*
  1. *Déclaration :* `vector<int> t;`
  2. *Allocation :* `t = vector<int>(3);`
  3. *Initialisation :* `t[0] = 3; t[1] = 0;`

  Utilisation : `t[i] = t[i]+1; t.size(); t.push_back(3);`
- *Un vecteur est une valeur comme les autres ;*
- *Il peut être passé en paramètre à ou renvoyé par une fonction.*

Différentes sortes de tableaux

Retenir

En `C++`, il y a trois sortes de tableaux :

1. *Les vecteurs* `std::vector<int> t` *:*
   - *réservation de mémoire manuelle ;*
   - *méthode (*`t.size()`, `t.at()`, `t.push_back(2)`*...)*
2. *Les tableaux* `std::array<int, 5> t` *:*
   - *Taille connue à la compilation, mémoire automatique ;*
   - *méthode (*`t.size()`, `t.at()`*) ;*
   - *taille fixe (pas de* `t.push_back(2)`*...)*
3. *Les tableaux bas niveau* `int t[5];` *:*
   - *hérités du C, pas de méthode (*`size`, `push_back(2)`, `at`*) ;*
   - *pas des valeurs (pas de retour, ni passage de paramètre)*
   - *gestion manuelle de la mémoire ;*
   - *peu utilisés en C++*

Tableaux : bas niveau et haut niveau

Compléments

Les `vector` et `array` sont des objets (voir la suite de ce cours) qui cachent un tableau bas niveau. Avantages :

- réservation simplifiée de mémoire
- libération automatique de la mémoire
- méthode...

Les tableaux bas niveau sont utilisés en C++ seulement quand on veut vraiment gérer sa mémoire.

Attention

Dans ce cours, on n'utilisera pas les tableaux bas niveau (voir l'UE ASD).

Déclaration tableaux et vecteurs

Retenir

Différence et similarité entre tableau et vecteur :

	tableau	vecteur
Entête	`#include<array>`	`#include<vector>`
Déclaration	`array<int, 5> ar;`	`vector<int> v;`
Réservation	Automatique	`v = vector<int>(5);`
Accès	`ar[i]`	`v[i]`
Accès contrôlé	`ar.at(i)`	`v.at(i)`
Taille	`ar.size()`	`v.size()`
Ajout de n	Impossible	`v.push_back(n);`
Modif. taille	Impossible	`v.resize(8);`
Suppr. dernier	Impossible	`v.pop_back();`
Vider	Impossible	`v.clear();`

Accès

Syntaxe

On accède, en lecture comme en écriture, à la valeur de chaque élément d'un tableau/vecteur en utilisant la ***notation indicée*** *:*

```
t[indice]
```

où `indice` *est une expression (constante, variable, calcul...).*

Attention

L'accès n'est possible que si l'**indice est compris entre 0 et la taille du tableau - 1**. Le comportement d'un **accès en dehors des bornes du tableau** est **INDÉFINI** (segfault, rien, modification d'une autre variable, accès super utilisateur...).

Accès contrôlé

Syntaxe

Si l'on veut vérifier que l'accès est bien valide :

```
t.at(indice)
```

où `indice` *est une expression (constante, variable, calcul...).*

Attention

En cas d'accès hors des bornes, une erreur (exception) est déclenchée.

Tableau et mémoire

Compléments

Pour stocker un tableau, le compilateur réserve des **emplacements mémoire consécutifs**.
Le calcul de l'adresse du i-ème élément du tableau peut donc se faire avec :

$$\text{Adresse}(\,T[i]\,) = \text{Adresse}(\,T[0]\,) + i * \text{Taille d'un élément}$$

Accès très rapide : **ne demande pas de parcourir le tableau**.

```
int t[10];      array<int, 10> t;     vector<int> v(10);
```

	0	1	2	3	4	5	6	7	8	9	
?	1	8	5	13	9	11	13	1	19	2	?

Rappel : tableau, accumulation, recherche

Retenir

Différentes manipulations

- *Initialisation, saisie, affichage ;*
- ***Accumulation** : on parcourt tout le tableau ; pour chaque case, on met à jour une valeur (par exemple : somme, produit, maximum...)*
- ***Recherche** : on parcourt le tableau ; si l'on a trouvé on retourne immédiatement la réponse ; il faut **attendre la fin pour répondre que l'on a pas trouvé**.*

Rappel : Manipulation de tableau

tab-acc-rech.cpp

```
1  // Déclaration de type
2  using tab = array<int, 10>;
3
4  tab lireTab() {
5    tab res;
6    for (int i = 0; i < res.size(); i++)
7      cin >> res[i];
8    return res;
9  }
10
11 void afficheTab(tab t) {
12   for (int i = 0; i < t.size(); i++)
13     cout << setw(3) << i;
14   cout << endl;
15   for (int i = 0; i < t.size(); i++)
16     cout << setw(3) << t[i];
17   cout << endl;
18 }
19
```


Rappel : Accumulation dans un tableau

tab-acc-rech.cpp

```
1  float moyenneTab(tab t) {
2    int somme = 0;
3    for (int i = 0; i < t.size(); i++)
4      somme += t[i];
5    return float(somme) / float(t.size());
6  }
7
8  int maxTab(tab t) {
9    int maxi = t[0];
10   for (int i = 1; i < t.size(); i++)
11     maxi = max(maxi, t[i]);
12   return maxi;
13 }
14
15 int maxTab_variante(tab t) {
16   int maxi = t[0];
17   for (int i = 1; i < t.size(); i++)
18     if (maxi < t[i]) maxi = t[i];
19   return maxi;
20 }
```

Rappel : Recherche dans un tableau

tab-acc-rech.cpp

```
1  /** Recherche dans un tableau
2   * @param t un tableau
3   * @param x l'entier à chercher
4   * @return la position de la première occurence de x
5   *         -1 si x n'apparaît pas dans le tableau
6   */
7  int chercheTab(tab t, int x) {
8    for (int i = 0; i < t.size(); i++)
9      if (t[i] == x) return i;
10   return -1;
11 }
```


Avantage des tableaux par rapport aux vecteurs

Tout ce que l'on peut faire avec un tableau peut être fait avec un vecteur. On ne peut pas changer la taille d'un tableau.

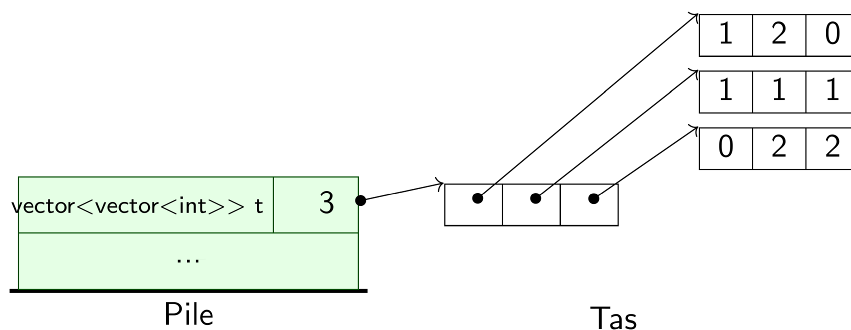
Pourquoi les tableaux ?

Retenir

- *on dit clairement que la taille est fixe*
- *pas besoin de réserver la mémoire*
- *plus efficaces en mémoire, donc légèrement plus rapides*

Organisation mémoire d'un vecteur

Organisation de la mémoire pour un vecteur à deux dimensions :



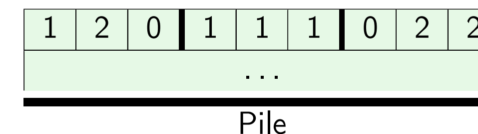
Organisation mémoire d'un tableau

Organisation de la mémoire pour un tableau à deux dimensions :

Retenir

Le tableau est stocké directement dans la pile !
Les lignes sont collées les unes après les autres.

Exemple pour le tableau `array<array<int,3>, 3> t` :



Exemple : Vecteurs et tableaux 2D

Exemple : tableau à deux dimensions pour un jeu de morpion :
Mise en place des cases du tableau :

morpion-vecteur.cpp

```
1  #include<iostream>
2  #include<iomanip>
3  #include<vector>
4  using namespace std;
5
6  enum class Case { vide, croix, rond };
7
8  char charCase(Case c) {
9    switch (c) {
10     case Case::vide  : return '.';
11     case Case::croix : return 'x';
12     case Case::rond  : return 'o';
13   }
14 }
```


Exemple : Vecteurs et tableaux 2D

Remplissage du tableau et affichage :

morpion-vecteur.cpp

```
1    // Initialization
2    t[0][0] = Case::croix; t[0][1] = Case::rond;  t[0][2] = Case::vi
3    t[1][0] = Case::croix; t[1][1] = Case::croix; t[1][2] = Case::cr
4    t[2][0] = Case::vide;  t[2][1] = Case::rond;  t[2][2] = Case::ro
5
6    // Affichage
7    for (int l = 0; l < 3; l++) {
8      for (int c = 0; c < 3; c++) {
9        std::cout << charCase(t[l][c]) << " ";
10     }
11     std::cout << std::endl;
12   }
```


Exemple : Vecteurs et tableaux 2D

Déclaration et allocation :
Vecteur :

morpion-vecteur.cpp

```
1    // Declaration
2    vector<vector<Case>> t;
3    // Allocation
4    t = vector<vector<Case>>(3);
5    // Allocation des sous-tableaux
6    for ( int i = 0; i < t.size(); i++ )
7      t[i] = vector<Case>(3);
```

Tableau :

morpion-tableau.cpp

```
1    // Declaration et Allocation
2    array<array<Case, 3>, 3> t;
```


Exemple : Vecteurs et tableaux 2D

Compléments

Il est en fait possible de raccourcir syntaxiquement les trois lignes :

morpion-vecteur.cpp

```
vector<vector<Case>> t;
t = vector<vector<Case>>(3);
for ( int i = 0; i < t.size(); i++ )
  t[i] = vector<Case>(3);
```

en

```
vector<vector<Case>> t (3, vector<Case>(3));
```

Cependant, le travail effectué par la machine et en particulier la boucle est le même.

Tableau ou vecteur

Retenir

Bilan :

- *On utilisera un tableau* `array` *si l'on connaît la taille à la compilation ;*
- *Sinon on utilisera un vecteur* `vector`.

Plan

1. Organiser ses données : les types
2. Les types structurés
3. Types énumérés
4. Tableaux : tableau C, array et vector
5. **Le mécano**
6. Annexe : tableaux dynamiques

Plan

1. Organiser ses données : les types
2. Les types structurés
3. Types énumérés
4. Tableaux : tableau C, array et vector
5. **Le mécano**
   - **Combinaisons de tableaux, vecteurs et structures**
   - Compléments : fonctions et paramètres
6. Annexe : tableaux dynamiques

Le principe de l'idée récursive

«Quand on a une bonne idée, en l'appliquant récursivement on obtient très souvent une bien meilleure idée.»

- Les structures et les tableaux sont de bonnes idées !
- En **combinant** les deux, on a encore beaucoup plus de possibilités.

Retenir

- *Il est possible qu'**un champ d'une structure soit un tableau***
- *On peut faire des **tableaux dont les éléments sont des structures***

Exemple de tableaux de structures

carnet.cpp

```
1  struct Personne {
2    string nom, prenom;
3    Date naissance;
4    long int telephone;
5    // int numero de rue; string nom de rue ...
6  };
7
8  Personne lirePersonne() {
9    Personne res;
10   cout << "Nom et prénom ? "; cin >> res.nom >> res.prenom;
11   res.naissance = lireDate();
12   cout << "Numéro de tél ? "; cin >> res.telephone;
13   return res;
14 }
15
16 void affichePersonne(Personne p) {
17   cout << p.nom << " " << p.prenom  << " ";
18   afficheDate(p.naissance);
19   cout << " " << p.telephone;
20 }
```


Exemple de tableaux de structures(1)

carnet.cpp

```
1  const int taille = 1000;
2  using Carnet = array<Personne, taille>;
```

carnet.cpp

```
1  Carnet initCarnet() {
2    Carnet res;
3    for (int i=0; i< taille; i++)
4      res[i].nom = "";
5    return res;
6  }
7
8  void afficheCarnet(Carnet c) {
9    for (int i=0; i< taille; i++)
10     if (c[i].nom != "") {
11       cout << i << " : ";
12       affichePersonne(c[i]);
13       cout << endl;
14     }
15   cout << endl;
16 }
```


Exemple de tableaux de structures (2)

carnet.cpp

```
1  int main() {
2    Carnet addr;
3
4    addr = initCarnet();
5
6    addr[5] = {"Toto", "Dupont", {3, 12, 1998}, 1345782348};
7    afficheCarnet(addr);
8
9    while (1) {
10     int i;
11     cout << "Case a modifier ? ";
12     cin >> i;
13     addr[i] = lirePersonne();
14     cout << endl;
15     afficheCarnet(addr);
16   }
17 }
```


Un début de structuration d'un jeu (1)

struct-jeu.cpp

```
1  // Exemple de déclaration de structures pour l'organisation d'un jeu
2  struct Coord { int x, y; };
3
4  enum class Couleur { noir, blanc, rouge, bleu, vert };
5  enum class Metier { guerrier, magicien, ranger, druide };
6  enum class TypeArme { epee, masse, arc, arbalete, couteau };
7
8  struct Arme {
9    TypeArme type;
10   int munitions;  // -1 si infinie ou pas
11 };
12
13 struct Personnage {
14   string Nom;
15   Metier metier;
16   int ptVie, bouclier;
17   vector<Arme> armes;
18   Coord position;
19 };
20
21 struct Armee {
22   Couleur coul;
23   vector<Personnage> pers;
24 };
```

Un début de structuration d'un jeu (2)

struct-jeu.cpp

```
1  struct Arme {
2    TypeArme type;
3    int munitions;  // -1 si infinie ou pas
4  };
5
6  struct Personnage {
7    string Nom;
8    Metier metier;
9    int ptVie, bouclier;
10   vector<Arme> armes;
11   Coord position;
12 };
13
14 struct Armee {
15   Couleur coul;
16   vector<Personnage> pers;
17 };
```

Déclaration d'un vecteur d'armées :

struct-jeu.cpp

```
vector<Armee> armees;
```

Exemple d'utilisations :

struct-jeu.cpp

```
1  // Le nombre de points de vie du
2  // personnage no 8 de l'armée no 2
3  cout << armees[2].pers[8].ptVie;
4  // son abscisse
5  cout << armees[2].pers[8].position.x;
6  // nbr de munitions son arme principale
7  cout <<
8      armees[2].pers[8].armes[0].munitions;
```


Plan

1 Organiser ses données : les types

2 Les types structurés

3 Types énumérés

4 Tableaux : tableau C, array et vector

5 Le mécano
- Combinaisons de tableaux, vecteurs et structures
- Compléments : fonctions et paramètres

6 Annexe : tableaux dynamiques

Valeurs

Retenir

Les vecteurs, tableaux, structures, énumérations et combinaisons sont des ***valeurs****. Il est donc possible de*

- *les affecter d'une variable dans une autre ;*
- *les passer en paramètre à une fonction ;*
- *les retourner dans une fonction.*

Problème des copies...

Affectation, paramètre, return...

Attention

Sans optimisations, ces opérations font une copie intégrale de toute la structure de données :

- recopie de tous les champs d'une structure
- recopie de toutes les cases d'un tableau
- recopie récursive des sous-tableaux et sous-structures

Cela peut être très lent et coûteux si l'on a de grosses structures.

Passages par référence

Voir cours ASD pour les détails.

Compléments

Pour éviter d'avoir à retourner une grosse structure, on envoie sa position en mémoire (référence) et la fonction travaille directement à la bonne place.

Au lieu de

```
1  GrosseStruct maFonction() {
2    GrosseStruct res;
3    res.machin = ...
4    return res;
5  }
6
7  int main {
8    GrosseStruct g = maFonction();
9  }
```

On écrit

```
1  void maProcedure(GrosseStruct &res) {
2    res.machin = ...
3  }
4
5  int main {
6    GrosseStruct g;
7    // Ici, la position de g en mémoire
8    // est transmise à maProcedure
9    maProcedure(g);
10 }
```


Passages par référence constante

Voir cours ASD pour les détails.

Compléments

Pour éviter d'avoir à passer une grosse structure qui ne sera pas modifiée, on envoie sa position en mémoire (référence).

Au lieu de

```
1  ... maFonction(GrosseStruct g) {
2    g.machin
3  }
4
5  int main {
6    GrosseStruct g;
7    maFonction(g);
8  }
```

On écrit

```
1  ... maFonction(const GrosseStruct &g) {
2    g.machin
3  }
4
5  int main {
6    GrosseStruct g;
7    maFonction(g);
8  }
```


Plan

1 Organiser ses données : les types

2 Les types structurés

3 Types énumérés

4 Tableaux : tableau C, array et vector

5 Le mécano

6 Annexe : tableaux dynamiques

Annexe

Attention

Ce qui suit est un extrait du cours de l'année dernière

Ce n'est pas au programme de «programmation modulaire» cette année. Les détails seront données en ASD.

Tableau et vecteur (1)

Contrairement aux vecteurs, les tableaux ont une taille fixe. On ne peut pas changer la taille. Pour résoudre ce problème, on réserve de la mémoire dont on utilise seulement une partie.

Retenir (Version statique)

*Dans cette solution, on fixe **statiquement** la quantité de mémoire à réserver que l'on appelle la **capacité**. En cas de débordement de capacité, on stoppe le programme avec un message d'erreur.*

tab-size.cpp

```
1 const int capacite = 10;
2 struct Tableau {
3   int taille;
4   int tab[capacite];
5 };
```


Initialisation et ajout d'un élément à la fin du tableau

tab-size.cpp

```
1  void initTab(Tableau &t) {
2    t.taille = 0;
3  }
4
5  void ajouteFinTab(Tableau &t, int i) {
6    if (t.taille == capacite) {
7      cout << " taille maximal atteinte " << endl;
8      exit(1);
9    }
10   t.tab[t.taille] = i;
11   t.taille++;
12 }
```


Problème de la solution statique

- Si l'on veut changer la capacité, il faut **recompiler le programme**
- Si l'on utilise plusieurs tableaux avec des tailles différentes, on **gaspille** beaucoup de mémoire.

Retenir

*On veut pouvoir **changer dynamiquement la capacité**.*

Complément : Allocation dynamique manuelle de mémoire

Syntaxe

*On peut à tout moment demander au C++ **d'allouer un emplacement mémoire** avec le mot clé `new` :*

- *Le résultat est un **pointeur sur la mémoire réservée**.*
- *La mémoire est allouée dans un segment appelé le **tas**.*

*Il faut toujours **libérer l'emplacement mémoire** quand on n'en a plus besoin avec le mot clé `delete`.*

alloc.cpp

```
1   int *pi;
2   pi = new int; // Réserve de la mémoire pour un entier
3   *pi = 5;
4   cout << *pi << endl;
5   delete pi; // libère la mémoire réservée
```

Allocation dynamique manuelle de mémoire

Retenir

L'allocation dynamique manuelle permet donc d'avoir des variables

- ***anonymes*** *(on y accède seulement par un pointeur)*
- ***dont la durée de vie est décidée par le programme*** *(création par le* `new`*, destruction par le* `delete`*).*

Allocation d'un tableau

Il est possible d'**allouer plusieurs emplacements mémoire consécutifs d'un coup**.

Compléments

- Allocation d'un tableau :

```
pointeur = new type_des_elements[nombre_d_elements];
```

- Libération d'un tableau :

```
delete [] pointeur;
```


Exemple d'allocation d'un tableau

alloc-tab.cpp

```
1   int *tab;
2   int nbelem, i;
3   cin >> nbelem;
4   tab = new int [nbelem]; // Réserve de la mémoire
5   // Utilisation du tableau comme un tableau normal
6   for (i=0; i < nbelem; i++) tab[i] = i*i;
7   for (i=0; i < nbelem; i++) cout << tab[i] << " ";
8   cout << endl;
9   // Fin de l'utilisation du tableau
10  delete [] tab; // libère la mémoire réservée
```


Tableau et vecteur (2)

Remarque (Version dynamique)

On va ajouter la capacité dans la structure de données et maintenir la cohérence entre la capacité et la mémoire allouée.

tab-size-dyn.cpp

```
1  struct Tab {
2    int taille;
3    int capacite;
4    int *tab;
5  };
6
7  void initTab(Tab &t) {
8    t.taille = 0;
9    t.capacite = 0;
10   t.tab = NULL;
11 }
```

Ajout à la fin d'un tableau dynamique

tab-size-dyn.cpp

```
1  void changeCapacite(Tab &t, int nouvcapa) {
2    t.taille = min(t.taille, nouvcapa);
3    int *nouvtab = new int [nouvcapa];
4    for (int i = 0; i < t.taille; i++) nouvtab[i] = t.tab[i];
5    delete [] t.tab; // pas de problème si t.tab est NULL
6    t.capacite = nouvcapa;
7    t.tab = nouvtab;
8  }
9
10 void ajouteFinTab(Tab &t, int elem) {
11   if (t.taille == t.capacite) {
12     changeCapacite(t, max(t.capacite+1, t.capacite*2));
13   }
14   t.tab[t.taille] = elem;
15   t.taille++;
16 }
```


Problème de la copie d'un tableau dynamique

Compléments

Il ne suffit pas de copier la structure, il faut aussi

- **allouer** un emplacement pour la copie
- **copier** les données du tableau

tab-size-dyn.cpp

```
1 void copieTab(const Tab &src, Tab &dst) {
2   dst = src;
3   dst.tab = new int [dst.capacite];
4   for (int i = 0; i < src.taille; i++) dst.tab[i] = src.tab[i];
5 }
```


Problème de la destruction d'un tableau dynamique

Compléments

Si l'on stocke un tableau dynamique dans une variable locale, et que l'on en a plus besoin, il ne faut pas oublier de **libérer la mémoire**.

tab-size-dyn.cpp

```
1 void detruitTab(Tab &t) {
2   delete [] t.tab;
3   t.tab = NULL;
4   t.taille = 0;
5   t.capacite = 0;
6 }
```


Copie d'un tableau dynamique

Compléments (Sémantique de copie)

La programmation objet permet en `C++` de

- **initialiser automatiquement** la structure
- **redéfinir l'opérateur d'affectation** pour faire la copie automatiquement.
- **libérer automatiquement** la mémoire si le tableau est une variable locale qui est détruite.

C'est le cas pour les `vector` et `array` du `C++`, mais pas pour les `arrayList` de Java et les `list` de Python.

Le type `vector` dans la vraie vie

Sur ma machine, dans le fichier `/usr/include/c++/5/bits/stl_vector.h`, on peut lire la définition des vecteurs. Il n'y a que trois champs :

```
1  typedef [...] pointer;
2  struct _Vector_impl
3  [...]
4  {
5      pointer _M_start;
6      pointer _M_finish;
7      pointer _M_end_of_storage;
8  [...]
9  };
```

Il y a deux fonctions `size` et `capacity` qui font essentiellement

```
1  return size_type(v._M_finish - v._M_start);
2  return size_type(v._M_end_of_storage - v._M_start);
```


Le type `vector` dans la vraie vie

Compléments

En pratique, les vecteurs sont implantés avec une structure comme la notre, à la différence que plutôt que de **stocker la taille et la capacité**, le développeur a préféré stocker **deux pointeurs sur la fin de la zone utilisée et la fin de la zone allouée.**

Bien sûr, il y a aussi toute la programmation objet qui permet de simplifier et d'automatiser un certain nombre d'opérations.