Programmation Impérative II
Structures de données

Florent Hivert

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

Organiser ses données

Pour le moment, on a 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...)
- Une discothèque doit modéliser des disques (auteur, titre, année, genre musical) qui contiennent des chansons (titre, durée)...

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(etu);
```

plutôt que

```
affiche(nom, prenom,
        annee_naiss, mois_naiss, jour_naiss,
        numero_rue, nom_rue, code_postal, ville,
        note info111, note info121);
```

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)

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**.

Type de données et **objets**

Remarque

Bien que l'on soit en C++ qui est un langage orienté objet, dans ce cours, nous n'allons **pas utiliser la programmation objet**.

En pratique, les programmes que l'on va écrire ressembleront plus à ce que l'on aurait écrit en C (avec `cout`, `cin` et les références en plus).

Remarque

Pour ce qui est de la définition de nouveaux types, **le C et le C++ utilisent des syntaxes qui ne sont pas toujours compatibles**.

Type de données et **objets**

Remarque

Bien que l'on soit en C++ qui est un langage orienté objet, dans ce cours, nous n'allons **pas utiliser la programmation objet**.

En pratique, les programmes que l'on va écrire ressembleront plus à ce que l'on aurait écrit en C (avec `cout`, `cin` et les références en plus).

Remarque

Pour ce qui est de la définition de nouveaux types, **le C et le C++ utilisent des syntaxes qui ne sont pas toujours compatibles**.

Type de données et **objets**

Remarque

Bien que l'on soit en C++ qui est un langage orienté objet, dans ce cours, nous n'allons **pas utiliser la programmation objet**.

En pratique, les programmes que l'on va écrire ressembleront plus à ce que l'on aurait écrit en C (avec `cout`, `cin` et les références en plus).

Remarque

Pour ce qui est de la définition de nouveaux types, **le C et le C++ utilisent des syntaxes qui ne sont pas toujours compatibles**.

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 certain cas).

Type de données

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 `&`
- les pointeurs `*`

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 `v.size()` .

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 `&`
- les pointeurs `*`

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 `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` $\mapsto$ `float`
- `float` $\mapsto$ `int` *(arrondi vers 0)*
- `bool` $\mapsto$ `int` (`false` $\rightarrow$ *0*, `true` $\rightarrow$ *1*)
- `int` $\mapsto$ `bool` (*0* $\rightarrow$ `false`, $\neq 0 \rightarrow$ `true`).

On peut demander une ***conversion explicite*** *avec :*

```
a = type(b);
```

Le compilateur signale une erreur si la conversion est impossible.

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` $\mapsto$ `float`
- `float` $\mapsto$ `int` *(arrondi vers 0)*
- `bool` $\mapsto$ `int` (`false` $\rightarrow$ *0*, `true` $\rightarrow$ *1*)
- `int` $\mapsto$ `bool` (*0* $\rightarrow$ `false`, $\neq 0 \rightarrow$ `true`).

On peut demander une ***conversion explicite*** *avec :*

```
a = type(b);
```

Le compilateur signale une erreur si la conversion est impossible.

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

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 :*

```
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
```

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 :*

```
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`).

Types énumérés

Syntaxe

*Un **type énuméré** est un type **dont on connaît l'ensemble des valeurs** :*

```
enum nom {val1, val2, ...};
```

Par exemple :

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

jour_semaine j;
...
j = mardi;
if (j == samedi) ...;
```

Type énuméré et chaîne

Retenir

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

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 ‘samedi’ not handled in switch
```

Type énuméré et chaîne

Retenir

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

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 ‘samedi’ not handled in switch
```

Exemple d'affichage d'un type énuméré

enum.cpp

```
1  string jour_string_switch(jour_semaine j) {
2      string res;
3      switch (j) {
4      case lundi :     res = "lundi"; break;
5      case mardi :     res = "mardi"; break;
6      case mercredi :  res = "mercredi"; break;
7      case jeudi :     res = "jeudi"; break;
8      case vendredi :  res = "vendredi"; break;
9      case samedi :    res = "samedi"; break;
10     case dimanche :  res = "dimanche"; break;
11     }
12     return res;
13 }
14 //...
15     jour_semaine j = lundi;
16     cout << jour_string_switch(j) << "  " << j << endl;
```

Exemple de saisie d'un type énuméré

enum.cpp

```
1   jour_semaine saisie_jour_switch() {
2       jour_semaine res;
3       char c;
4       bool ok;
5       do {
6           ok = true;
7           cout << "Quel jour (l)undi, m(a)rdi, m(e)rcredi, (j)eudi, (v)endredi, (s)amedi, (d)imanche ? ";
8           cin >> c;
9           switch (c) {
10          case 'L' : case 'l' : res = lundi; break;
11          case 'A' : case 'a' : res = mardi; break;
12          case 'E' : case 'e' : res = mercredi; break;
13          case 'J' : case 'j' : res = jeudi; break;
14          case 'V' : case 'v' : res = vendredi; break;
15          case 'S' : case 's' : res = samedi; break;
16          case 'D' : case 'd' : res = dimanche; break;
17          default : ok = false;
18          }
19      } while (not ok);
20      return res;
21  }
```

Codage des types énumérés

Retenir

*Les valeurs d'un type énuméré sont en fait **codées par des entiers**. Le C++ fait une **conversion implicite** vers l'entier.*

```
jour_semaine j = lundi;
cout << j;  // affiche le code de lundi c'est-à-dire 0
```

Retenir

*La conversion dans l'autre sens (entier $\mapsto$ type énuméré) doit être demandée **explicitement** (après avoir vérifié que l'entier est bien le code d'une valeur du type énuméré).*

Codage des types énumérés

Retenir

*Les valeurs d'un type énuméré sont en fait **codées par des entiers**. Le C++ fait une **conversion implicite** vers l'entier.*

```
jour_semaine j = lundi;
cout << j;  // affiche le code de lundi c'est-à-dire 0
```

Retenir

*La conversion dans l'autre sens (entier $\mapsto$ type énuméré) doit être demandée **explicitement** (après avoir vérifié que l'entier est bien le code d'une valeur du type énuméré).*

Conversion type énuméré vers `string`

enum.cpp

```
1  const string nom_du_jour[7] = {
2      "lundi", "mardi", "mercredi", "jeudi",
3      "vendredi", "samedi", "dimanche"};
4
5  string jour_string(jour_semaine j) {
6      return nom_du_jour[j];
7  }
```

Exemple de saisie d'un type énuméré

enum.cpp

```
1  jour_semaine saisie_jour() {
2      int res;
3      do {
4          cout << "Quel jour (0=lundi, 6=dimanche) ? ";
5          cin >> res;
6      } while (res < lundi or res > dimanche);
7      return jour_semaine(res);
8  }
```

Structures

Quand plusieurs variables modélisent le même objet du monde réel, il est très pratique de les regrouper.

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.

Structures

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 :

```
struct dureeHMS {
  int heure;
  int minute;
  float seconde;
};
```

Nombre complexe :

```
  struct complexe {
    float part_re, part_im;
  };
```

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

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

Type «assuré social»

```
struct AssureSocial {
  string nom, prenom, Nsecu;
  date date_naissance;
};
```

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 :

```
date aujourdhui;
aujourdhui.mois = 2;
aujourdhui.annee = 2020;
```

Structures imbriquées :

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

Manipulations des structures

Retenir

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

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

En particulier, on ne peut pas lire ou afficher une structure directement.

Manipulations des structures

Retenir

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

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

En particulier, on ne peut pas lire ou afficher une structure directement.

Exemple : Affichage d'une date

date.cpp

```
 1  #include<iostream>
 2  #include<iomanip>
 3  using namespace std;
 4
 5  struct date {
 6    int jour, mois, annee;
 7  };
 8
 9  /** Affiche une date sous le format jj/mm/aaaa
10  * @param[in] d :  la date
11  **/
12  void afficheDate(date d) {
13    cout << setfill('0') << setw(2) << d.jour << "/"
14         << setfill('0') << setw(2) << d.mois << "/" << d.annee;
15  }
```

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 }
```

Manipulation d'une date

date.cpp

```
1  /** Augmente la date d'un jour
2  *  @param[in/out] d : une date
3  **/
4  void avanceDate(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 }
```

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.

a	nom		"John"
	prenom		"Doe"
	Nsecu		"1 74 34 234 123"
	date_naissance	annee	1974
		mois	11
		jour	3

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

Initialisation d'une `struct`

Syntaxe

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

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

init-struct.cpp

```
1  struct date {
2    int jour, mois, annee;
3  };
4  date d = {3, 11, 1974};
5
6  // fonction qui retourne le 1er janvier 2000
7  date bugday() {
8    return {1, 1, 2000};
9  }
```

Passage d'une structure en paramètre par valeur

Retenir

Tous les champs *de la structure paramètre réel doivent être recopiés dans le paramètre formel :*

Avant l'appel ligne 6

```
1  void afficheDate(date d) {
2    cout << ...
3  }
4  [...]
5    date rdv;
6    rdv = {12, 3, 2019};
7    afficheDate(rdv);
```

Pile :

	14	?
	13	?
	12	?
	11	?

...

rdv.annee	6	**2019**
rdv.mois	5	**3**
rdv.jour	4	**12**

...

Passage d'une structure en paramètre par valeur

Retenir

Tous les champs *de la structure paramètre réel doivent être recopiés dans le paramètre formel :*

Appel de `afficheDate` ligne 7

```
1  void afficheDate(date d) {
2    cout << ...
3  }
4  [...]
5    date rdv;
6    rdv = {12, 3, 2019};
7    afficheDate(rdv);
```

Pile :

afficheDate	14	ADM
d.annee	13	2019
d.mois	12	3
d.jour	11	12

...

6	2019
5	3
4	12

...

Passage d'une structure en paramètre par référence

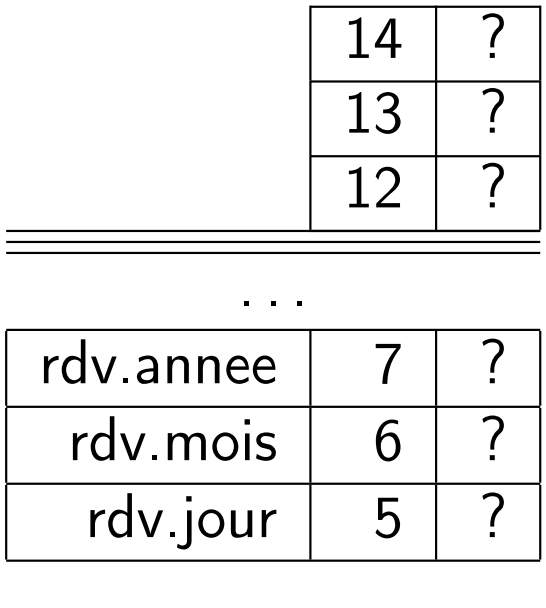
Retenir

*Comme on stocke les champs **toujours dans le même ordre**, il suffit de passer l'**adresse du premier champ**. Les autres seront dans les cases mémoires suivantes.*

Avant l'appel ligne 5

```
1  void lireDate(date &d) {
2    [...]
3    cin >> d.annee;
4    [...]
5  }
6  [...]
7    date rdv;
8    lireDate(rdv);
```

Pile :



Passage d'une structure en paramètre par référence

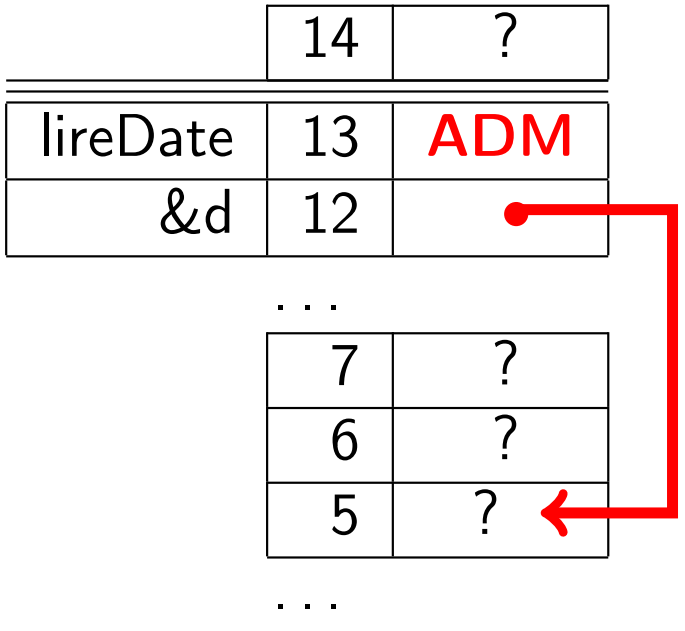
Retenir

*Comme on stocke les champs **toujours dans le même ordre**, il suffit de passer l'**adresse du premier champ**. Les autres seront dans les cases mémoires suivantes.*

Appel de `lireDate` ligne 6

```
1  void lireDate(date &d) {
2    [...]
3    cin >> d.annee;
4    [...]
5  }
6  [...]
7    date rdv;
8    lireDate(rdv);
```

Pile :



Passage d'une structure en paramètre par référence

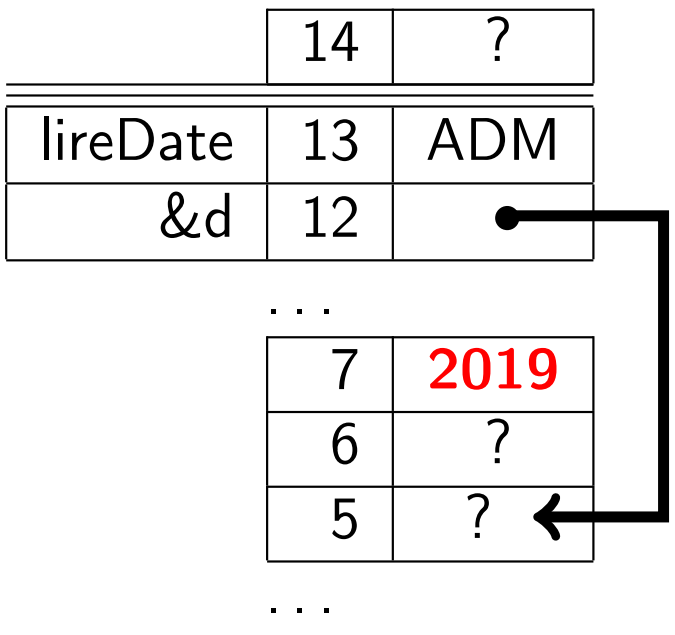
Retenir

*Comme on stocke les champs **toujours dans le même ordre**, il suffit de passer l'**adresse du premier champ**. Les autres seront dans les cases mémoires suivantes.*

Affectation de `d.jour` ligne 1

```
1  void lireDate(date &d) {
2    [...]
3    cin >> d.annee; // -> 2019
4    [...]
5  }
6  [...]
7    date rdv;
8    lireDate(rdv);
```

Pile :



Une structure comme résultat

L'affectation des structures est possible ! Une fonction peut donc retourner une structure.

Retenir

*La case mémoire `return` du tableau d'activation est en fait composée de **plusieurs cases mémoires**.*

Mais, cela peut forcer (pas toujours...) le compilateur à faire une copie de la structure qui peut être coûteuse.

Retenir

*Il peut être plus efficace de **passer le résultat dans un paramètre par référence** (mais attention à la lisibilité du code)...*

Plutôt que : `AssureSocial LitAssureSocial();`

Écrire : `void LitAssureSocial(AssureSocial &A);`

Une structure comme résultat

L'affectation des structures est possible ! Une fonction peut donc retourner une structure.

Retenir

*La case mémoire `return` du tableau d'activation est en fait composée de **plusieurs cases mémoires**.*

Mais, cela peut forcer (pas toujours...) le compilateur à faire une copie de la structure qui peut être coûteuse.

Retenir

*Il peut être plus efficace de **passer le résultat dans un paramètre par référence** (mais attention à la lisibilité du code)...*

Plutôt que : `AssureSocial LitAssureSocial();`

Écrire : `void LitAssureSocial(AssureSocial &A);`

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`.

```
float moyenne(vector<float> const &notes)
```

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`.

```
float moyenne(vector<float> const &notes)
```

Les tableaux

Le **type tableau** est un type prédéfini qui permet de créer un grand nombre de variables du même type à des emplacements consécutifs de la mémoire.

Syntaxe

Pour définir une variable de type tableau, il faut préciser sa taille ainsi que le type de ses éléments :

```
type_des_elements nom_du_tableau[taille];
```

On peut aussi donner un nom au type tableau :

def-tableau.cpp

```
1 const int taille=10;
2 using tab = int[taille];
3 tab t;
```

Tableau, C et C++

Retenir

Les tableaux définis ici sont les ***tableaux bas niveau*** *hérités du langage C.*
En C++ moderne, on utilise plutôt les classes `array` *pour les tableaux de taille fixe et* `vector` *pour les tableaux de taille variable.*
Ces classes sont construites en utilisant les tableaux du C.

def-tableau.cpp

```
1
2  using namespace std;
3  #include<array>
4  using tabcpp = array<int, taille>;
5
6  #include<vector>
7  using vectcpp = vector<int>;
```

Type tableau...

Syntaxe

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

```
t[indice]
```

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

Retenir

*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...).*

Type tableau...

Syntaxe

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

```
t[indice]
```

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

Retenir

*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...).*

Tableau et mémoire

Retenir

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 :

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

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

```
int t[10];
```

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

Tableau et mémoire

Retenir

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 :

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

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

```
int t[10];
```

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

Passage d'un tableau en paramètre

Retenir (**Attention !**)

*Les tableaux ont un comportement particulier : lors d'un passage d'un tableau en paramètre, on ne passe que l'**adresse du premier élément**.*

En quelque sorte, le tableau est passé par référence. Il n'est pas possible de passer un tableau simple par valeur. En pratique, on veut rarement passer un tableau par valeur car la recopie est coûteuse. Solutions (voir `exemple/tab-const.cpp`) :

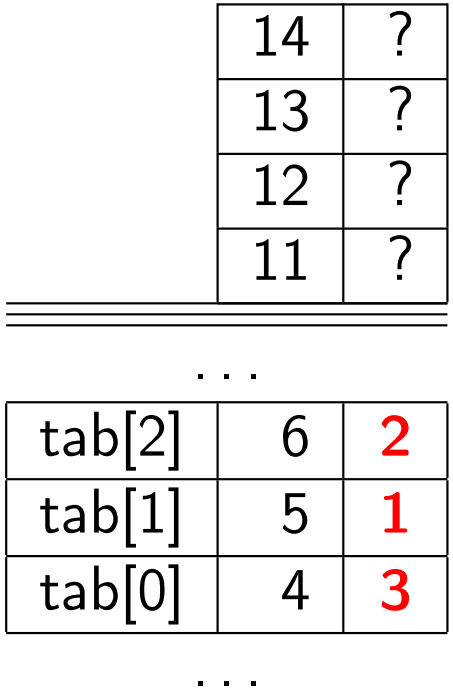
- Utiliser un tableau constant : `const int t[3]`
- Utiliser les `array` du `C++` : `array<int, 3> t`
- Mettre le tableau à l'intérieur d'une structure (voir plus loin)

Mécanisme du passage en paramètre d'un tableau

Avant l'appel ligne 6

```
1  void afficheTab(int t[3]) {
2    for (int i = 0; i < 3; i++)
3        cout << t[i] << " ";
4  }
5  [...]
6    int tab[3] = {3,1,2};
7    afficheTab(tab);
```

Pile :

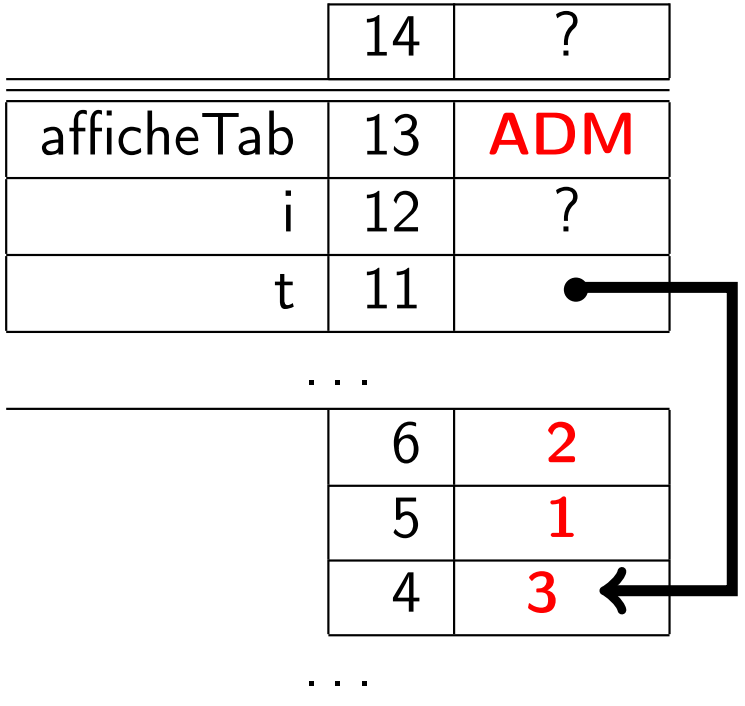


Mécanisme du passage en paramètre d'un tableau

Appel ligne 7

```
1  void afficheTab(int t[3]) {
2    for (int i = 0; i < 3; i++)
3        cout << t[i] << " ";
4  }
5  [...]
6    int tab[3] = {3,1,2};
7    afficheTab(tab);
```

Pile :



Rappel (Manipulation de tableau)

tab-acc.cpp

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

const int taille = 10;
using tab = int[taille];

void lireTab(tab t) {
 for (int i=0; i<taille; i++)
    cin >> t[i];
}

void afficheTab(const tab t) {
  for (int i=0; i<taille; i++)
    cout << setw(3) << i;
  cout << endl;
  for (int i=0; i<taille; i++)
    cout << setw(3) << t[i];
  cout << endl;
}

float moyenneTab(const tab t) {
  int somme = 0;
  for (int i=0; i<taille; i++)
    somme += t[i];
  return float(somme) / float(taille);
}
```

tab-acc.cpp

```cpp

int maxTab(const tab t) {
  int maxi = 0;
  for (int i=0; i<taille; i++)
    maxi = max(maxi, t[i]);
  return maxi;
}

int main() {
  tab notes;

  lireTab(notes);
  cout << endl << endl << endl;

  afficheTab(notes);

  cout << "moyenne = " << moyenneTab(notes) << endl;

  cout << "maxi = " << maxTab(notes) << endl;

}
```

Passer un tableau de taille inconnue

Retenir

Dans le mécanisme de passage de paramètres précédent, on peut passer un tableau dont la ***fonction ne connaît pas la taille à la compilation****.*

tab-ukn-size.cpp

```
1  void afficheTab(int taille, const int t[]) {
2    for (int i = 0; i<taille; i++)
3      cout << t[i] << " ";
4  }
5
6  int main() {
7    int tab[3] = {12,43,13};
8    afficheTab(3, tab); cout << endl;
9    int tab2[4] = {1,34,54,12};
10   afficheTab(4, tab2); cout << endl;
11 }
```

Différences tableau/vecteur

- La taille d'un tableau est **fixée statiquement** et ne peut pas évoluer, alors que celle d'un vecteur peut évoluer en fonction des besoins
- Contrairement aux vecteurs, un tableau n'a pas de **méthode prédéfinie** :
  - calcul du nombre d'éléments : `v.size()`
  - ajout d'un élément à la fin : `v.push_back(n)`
- Il faut représenter, dans une variable annexe, le nombre d'éléments présents dans le tableau
- Les tableaux sont **toujours passés par un pointeur sur le premier élément** alors que les vecteurs sont passés normalement (par valeur, ou par référence avec `&`)
- Pas de distinction entre les phases de **déclaration et d'allocation** initiales comme pour les vecteurs
- **Pas d'affectation directe possible** d'un tableau dans un autre

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*

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  const int taille = 1000;
9  using carnet = personne[taille];
10
11 void lirePersonne(personne &p) {
12   cin >> p.nom >> p.prenom;
13   p.naissance = lireDate();
14   cin >> p.telephone;
15 }
16
17
18 void affichePersonne(personne p) {
19   cout << p.nom << " " << p.prenom << " ";
20   afficheDate(p.naissance);
21   cout << " " << p.telephone;
22 }
23
24 void initCarnet(carnet c) {
25   for (int i=0; i< taille; i++)
26     c[i].nom = "";
27 }
```

carnet.cpp

```
1  void afficheCarnet(carnet c) {
2    for (int i=0; i< taille; i++)
3      if (c[i].nom != "") {
4        cout << i << " : ";
5        affichePersonne(c[i]);
6        cout << endl;
7      }
8    cout << endl;
9  }
10
11
12 int main() {
13   carnet addr;
14
15   initCarnet(addr);
16
17   addr[5] = {"toto", "dupont", {3, 12, 1998}, 1345782348};
18   afficheCarnet(addr);
19
20   int i;
21   cout << "Case a modifier ? ";
22   cin >> i;
23   lirePersonne(addr[i]);
24   cout << endl;
25
26   afficheCarnet(addr);
27 }
```

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**.

Syntaxe

- *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.

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.