TP Programmation C/C++
Les structures de données

© 2017 *tv* <tvaira@free.fr> v.1.1

Sommaire

Bilan **23**

Les objectifs de ce tp sont de comprendre et mettre en oeuvre des structures de données. Certains exercices sont extraits du site www.france-ioi.org.

Les critères d'évaluation de ce TP sont :

- le minimum attendu : on doit pouvoir fabriquer un exécutable et le lancer !
- le programme doit énoncer ce qu'il fait et faire ce qu'il énonce !
- le respect des noms de fichiers
- le nommage des fonctions et des variables, l'utilisation de constantes
- une fonction doit énoncer ce qu'elle fait et faire seulement ce qu'elle énonce !
- les fonctions ne dépassent pas 15 lignes
- le code est fonctionnel
- la simplicité du code

Types composés

Les types composés permettent de regrouper des variables au sein d'une même entité :

- Il est possible de regrouper des variables de types différents dans des **structures de données** ;
- Il est possible de regrouper des variables de types identiques dans des **tableaux**.

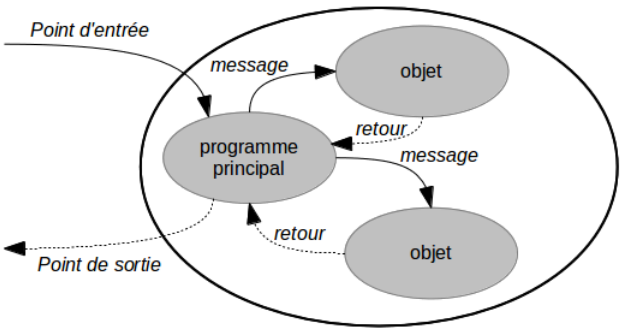
✍ *Une chaîne de caractères peut être considérée comme un type composé.*

En C, on dispose de trois types de structures de données :

- les structures (`struct`)
- les unions (`union`)
- les champs de bits

Le C++ ajoute la notion de type **classe** qui permet de réaliser des programmes orientés objet (POO). La classe est le modèle (le « moule ») pour créer des objets logiciels.

✍ *En POO, un programme est vu comme un ensemble d'objets interagissant entre eux en s'échangeant des messages.*



Structures

➟ *Besoin : Il est habituel en programmation d'avoir besoin d'un mécanisme permettant de grouper un certain nombre de variables de types différents au sein d'une même entité.*

Exemple : On travaille par exemple sur un fichier de personnes et on voudrait regrouper une variable de type chaîne de caractères pour le nom, une variable de type entier pour le numéro d'employé, etc.

La réponse à ce besoin est le concept d'**enregistrement** : un enregistrement est un ensemble d'éléments de types différents repérés par un nom. Les éléments d'un enregistrement sont appelés des **champs**.

En langage C, on utlise le vocabulaire suivant :

– enregistrement → **structure**
– champ d'un enregistrement → **membre** d'une structure

Une **structure** est donc un **objet agrégé comprenant un ou plusieurs membres** d'éventuellement différents types que l'on regroupe sous un seul nom afin d'en faciliter la manipulation et le traitement.

Chacun des membres peut avoir n'importe quel type, y compris une structure, à l'exception de celle à laquelle il appartient.

Déclaration de structure

Il y a plusieurs méthodes possibles pour déclarer des structures.

```
struct [etiquette]
{
   type champ_1;
   ...
   type champ_n;
} [identificateur];
```

Pour déclarer une structure, on utilise le mot clé `struct` suivi d'une liste de déclarations entre accolades. Il est possible de faire suivre le mot `struct` d'un nom baptisé etiquette de la structure. Cette etiquette désigne alors cette sorte de structure et, par la suite, peut servir pour éviter d'écrire entièrement toute la déclaration. Il est aussi possible d'instancier directement une variable de nom `identificateur`.

```
// Déclaration d'une type struct Date :
struct Date
{
   int   jour;
   int   mois;
   int   annee;
};

// Instanciation d'une variable de type struct Date :
struct Date dateNaissance;

// Ou directement :
struct
{
   int   jour;
   int   mois;
   int   annee;
} dateDeces; // dateDeces est une variable de type structuré
```

Exemple de déclaration d'un type structuré

En C, le type s'appelle en fait `struct Date`. On préfère souvent créer un **synonyme** avec `typedef` :

```
// Déclaration d'une type struct Date :
struct Date
{
   int   jour,
         mois,
         annee;
};

// Création d'un type synonyme :
typedef struct Date Date;

// Ou directement :
typedef struct
{
   int   jour,
         mois,
         annee;
} Date;

// Une variable de type Date :
Date dateNaissance;
```

Un synonyme de type structuré

✍ *En C++, ce synonyme est créé naturellement avec la structure. Le* ***`typedef`*** *est donc inutile dans ce langage.*

Initialisation d'une structure

Comme pour les tableaux, les accolades peuvent être utilisées pour indiquer les valeurs initiales des membres d'une variable de type structuré. Cela ne fonctionne qu'à l'initialisation.

☮ Ce n'est pas utilisable pour une affectation.

```
// Une variable de type Date :
Date naissanceMarie = {7, 11, 1867}; // initialisation de la structure naissanceMarie

printf("La structure struct Date occupe une taille de %d octets\n", sizeof(struct Date));
```

Initialisation d'une structure

✍ *La taille d'une structure est la somme des tailles de tous les objets qui la compose (cf.* ***`sizeof()`****). Dans notre exemple, la structure aura une taille de* 3×4 *(****`int`****) soit 12 octets.*

Accès aux membres

Pour accéder aux membres d'une structure, on utilise :

– l'opérateur d'accès `.` (point) pour une variable de type structuré :

```
Date naissanceMarie = {7, 11, 1867}; // initialisation de la structure naissanceMarie

printf("Marie Curie est née le %02d/%02d/%4d\n", naissanceMarie.jour, naissanceMarie.mois,
     naissanceMarie.annee);
```


– l'opérateur d'indirection `->` (flèche) pour un pointeur sur un type structuré :

```
struct Date naissanceMarie = {7, 11, 1867}; // initialisation de la structure
    naissanceMarie
struct Date *p_naissanceMarie = &naissanceMarie;

printf("Marie Curie est née le %02d/%02d/%4d\n", p_naissanceMarie->jour, p_naissanceMarie
    ->mois, p_naissanceMarie->annee);

// Ou :
printf("Marie Curie est née le %02d/%02d/%4d\n", (*p_naissanceMarie).jour->jour, (*
    p_naissanceMarie).mois, (*p_naissanceMarie).annee);
```

Affectation de structure

Il suffit d'accéder aux membres d'une structure pour leurs affecter une valeur.

```
struct Date naissanceMarie = {7, 11, 1867}; // initialisation de la structure naissanceMarie
struct Date uneDate;

// Quelques affectations :
uneDate.jour = 1;
scnaf("%d", &uneDate.mois);
uneDate.jour = naissanceMarie.annee;
```

Il est aussi possible d'affecter des structures de même type entre elles :

```
struct Date naissanceMarie = {7, 11, 1867};
struct Date copie;

copie = uneDate;
```

Dans ce cas, les valeurs sont copiées.

Ainsi, lorsque vous prenez une structure en paramètre d'une fonction, la valeur donnée à l'appel de la fonction est copiée dans l'emplacement mémoire du paramètre. Le paramètre a ainsi son propre emplacement mémoire. Si la structure est très grosse, appeler la fonction va donc nécessiter un espace supplémentaire, et une copie qui peut prendre du temps. Pour éviter cela, on pourra utiliser un **passage par adresse** en C/C++ ou **par référence** en C++.

☮ Aucune comparaison n'est possible sur les structures, même pas les opérateurs `==` et `!=`.

Tableaux de structures

Une déclaration de tableau de structures se fait selon le même modèle que la déclaration d'un tableau dont les éléments sont de type simple :

```
struct Date t[100]; // 100 éléments de type struct Date

// Ou :
Date t[100]; // 100 éléments de type Date

// Chaque élément du tableau est de type Date :
for(i=0<100;i++)
   printf("%02d/%02d/%4d\n", t[i].jour, t[i].mois, t[i].annee);
```


Liste de structures

Une des utilisations fréquentes des structures, est de créer des listes de structures chaînées. Pour cela, il faut que chaque structure contienne un membre qui soit de type **pointeur** vers une structure du même type. Cela se fait de la façon suivante :

```
struct personne
{
   ... /*  les différents membres */

   struct personne *suivant;
};
```

✍ *La dernière structure de la liste devra avoir un membre suivant dont la valeur sera le pointeur* `NULL` *pour indiquer la fin.*

Quand on crée une liste chaînée, on ne sait généralement pas à la compilation combien elle comportera d'éléments à l'exécution. Pour pouvoir créer des listes, il est donc nécessaire de pouvoir allouer de l'espace dynamiquement :

```
#include <stdlib.h>

struct personne *p;

// Allocation :
p = malloc(sizeof(struct personne));

// Libération :
free(p);
```

Allocation dynamique d'une structure

Union

➠ *Besoin : Il est parfois nécessaire de manipuler des variables auxquelles on désire affecter des valeurs de type différents.*

Une **union** est conceptuellement identique à une structure mais peut, à tout moment, contenir n'importe lequel des différents champs.

Une union définit en fait **plusieurs manières de regarder le même emplacement mémoire**. A l'exception de ceci, la façon dont sont déclarés et référencées les structures et les unions est identique.

```
union [etiquette] {
   type champ_1;
   ...
   type champ_n;
} [identificateur];
```

Exemple : On va déclarer une `union` pour conserver la valeur d'une mesure issue d'un capteur générique, qui peut par exemple fournir une mesure sous forme d'un `char` (-128 à +127), d'un `int` (-2147483648 à 2147483647) ou d'un `float`. `valeurCapteur` pourra ,à tout moment, contenir SOIT un entier, SOIT un réel, SOIT un caractère.


```
union mesureCapteur
{
   int   iVal;
   float fVal;
   char  cVal;
} valeurCapteur;
```

✍ *La taille mémoire de la variable* ***valeurCapteur*** *est égale à la taille mémoire du plus grand type qu'elle contient (ici c'est* ***float****).*

```
#include <stdio.h>
typedef union mesureCapteur
{
   int   iVal;
   float fVal;
   char  cVal;
} Capteur;

int main()
{
   Capteur vitesseVent, temperatureMoteur, pressionAtmospherique;
   pressionAtmospherique.iVal = 1013; /* un int */
   temperatureMoteur.fVal = 50.5; /* un float */
   vitesseVent.cVal = 2; /* un char */

   printf("La pression atmosphérique est de %d hPa\n", pressionAtmospherique.iVal);
   printf("La température du moteur est de %.1f °C\n", temperatureMoteur.fVal);
   printf("La vitesse du vent est de %d km/h\n", vitesseVent.cVal);

   printf("Le type Capteur occupe une taille de %d octets\n", sizeof(Capteur));

   return 0;
}
```

Utilisation d'une ***union***

L'exécution du programme d'essai permet de vérifier cela :

```
La pression atmosphérique est de 1013 hPa
La température du moteur est de 50.5 °C
La vitesse du vent est de 2 km/h

Le type Capteur occupe une taille de 4 octets
```

Champs de bits

➟ *Besoin : Il est parfois nécessaire pour un programmeur de décrire en termes de bits la structure d'une information.*

Les **champs** de bits ("Drapeaux" ou "*Flags*"), qui ont leur principale application en informatique industrielle, sont des **structures** qui ont la possibilité de regrouper (au plus juste) plusieurs valeurs. La taille d'un champ de bits **ne doit pas excéder celle d'un entier**. Pour aller au-delà, on créera un deuxième champ de bits.

On utilisera le mot clé `struct` et on donnera le type des groupes de bits, leurs noms, et enfin leurs tailles :


```
struct [etiquette]
{
   type champ_1 : nombre_de_bits;
   type champ_2 : nombre_de_bits;
   [...]
   type champ_n : nombre_de_bits;
} [identificateur];
```

Si on reprend le type structuré `Date`, on peut maintenant décomposer ce type en trois groupes de bits (`jour`, `mois` et `annee`) avec le nombre de bits suffisants pour coder chaque champ. Les différents groupes de bits seront tous accessibles comme des variables classiques d'une structure ou d'une union.

```
struct Date
{
  unsigned short jour : 5; // 2^5 = 0-32 soit de 1 à 31
  unsigned short mois : 4; // 2^4 = 0-16 soit de 1 à 12
  unsigned short annee : 7; // 2^7 = 0-128 soit de 0 à 99 (sans les siècles)
};

int main (int argc, char **argv)
{
   struct Date naissanceRitchie = {9, 9, 41};
   struct Date naissanceThompson = {4, 2, 43};
   struct Date mortRitchie = {12, 10, 11};

   printf("Dennis Ritchie est né le %02d/%02d/%2d\n", naissanceRitchie.jour,
      naissanceRitchie.mois, naissanceRitchie.annee);
   printf("Dennis Ritchie est mort le %02d/%02d/%2d\n\n", mortRitchie.jour, mortRitchie.mois
      , mortRitchie.annee);
   printf("Ken Thompson est né le %02d/%02d/%2d\n", naissanceThompson.jour,
      naissanceThompson.mois, naissanceThompson.annee);

   printf("La structure champs de bits date occupe une taille de %d octets\n", sizeof(struct
       date));
   return 0;
}
```

Le champ de bits `Date`

✍ *Il est autorisé de ne pas donner de nom aux champs de bits qui ne sont pas utilisés.*

L'exécution du programme d'essai permet de vérifier cela :

```
Dennis Ritchie est né le 09/09/41
Dennis Ritchie est né le 09/09/41
Dennis Ritchie est mort le 12/10/11
Ken Thompson est né le 04/02/43

La structure champs de bits date occupe une taille de 2 octets
```

✍ *La taille mémoire d'une variable de ce type sera égale à 2 octets (5 + 4 + 7 = 16 bits).*

Classes et objets

➟ *Besoin : Il est intéressant de pouvoir représenter une entité du monde physique, comme une voiture, une personne ou encore une page d'un livre, sous la forme d'une structure comprenant des données et un comportement.*

Un **objet** est caractérisé par le rassemblement d'un ensemble de **propriétés** (constituant son **état**) et d'un **comportement** :

- La notion de propriété est matérialisée par un **attribut** qui est une variable locale membre d'un objet
- La notion de comportement est matérialisée par un ensemble de **méthodes** qui sont ses fonctions membres

Une **classe** déclare des propriétés communes à un ensemble d'objets. Elle représente donc une catégorie d'objets.

Exemple : un objet lampe

- Une lampe est caractérisée par :
  - sa **puissance** (une **propriété** ➟ un **attribut**)
  - le **fait qu'elle soit allumée ou éteinte** (un **état** ➟ un **attribut**)
- Au niveau **comportement**, les **actions possibles** sur une lampe sont :
  - l'**allumer** (une **méthode**)
  - l'**éteindre** (une autre **méthode**)

✍ *Les langages orientés objets les plus utilisés sont : C++, C#, Java, PHP, Python, ...*

Déclaration d'une classe

Déclarer une classe revient à créer un nouveau **type** (un *moule*) à partir duquel il sera possible de créer des objets.

Pour déclarer une classe, on utilise le mot clé `class` (le mot clé `struct` est autorisé) suivi d'une liste de déclarations entre accolades :

```
class etiquette
{
   [private|public|protected:]
      type attribut_1;
      ...
      type attribut_n;

      [type] methode_1([type parametre_1, ...]);
      ...
      [type] methode_n([type parametre_1, ...]);
};
```

```
class Point
{
   double x, y; // des attributs : nous sommes des propriétés de la classe Point
   void afficher(); // une méthode : je suis un comportement de la classe Point
};
```

Une classe Point

☞ *Il existe d'autres formes de déclaration notamment pour l'héritage que l'on verra plus tard.*

Le C++ permet de préciser le **type d'accès aux membres** (**attributs** et **méthodes**) d'un objet. Cette opération s'effectue au sein des classes de ces objets :

- **Accès `public` :** on peut utiliser le membre de n'importe où dans le programme.
- **Accès `private` :** seule une fonction membre de la même classe peut utiliser ce membre ; il est invisible de l'extérieur de la classe.
- **Accès `protected` :** ce membre peut être utilisé par une fonction de cette même classe, et pas ailleurs dans le programme (ressemble donc à `private`), mais il peut en plus être utilisé par une classe dérivée.

```
class Lampe
{
    private: // des membres privés à la classe
        int puissance;
        bool estAllumee;

    public: // des membres publiques
        void allumer();
        void eteindre();
};
```

Déclaration de la classe Lampe

✍ *Comme pour les autres déclarations, celle d'un type* ***`class`*** *se fera dans un fichier d'en-tête (*`.h`*) dont le nom est généralement celui de la classe (par exemple* ***`lampe.h`****).*

En résumé, une classe est une structure C avec des fonctions dedans (ou pas).

✍ *En fait, en C++, même les* ***`struct`*** *sont des classes. La seule différence entre* ***`struct`*** *et* ***`class`*** *en C++ est que les membres de* ***`struct`*** *sont* ***`public`*** *par défaut et* ***`class`****,* ***`private`****.*

Définition d'une classe

La définition d'une classe revient à définit l'ensemble de ses méthodes. On doit faire précéder chaque méthode de l'opérateur de résolution de portée :: précédé du nom de la classe (`Lampe`) pour préciser au compilateur que ce sont des membres de cette classe :

```
void Lampe::allumer()
{
   estAllumee = true;
}

void Lampe::eteindre()
{
   estAllumee = false;
}
```

Définition de la classe Lampe

✍ *Comme pour les autres définitions, celle d'un type* ***`class`*** *se fera dans un fichier source (*`.cpp`*) dont le nom est généralement celui de la classe (par exemple* ***`lampe.cpp`****).*

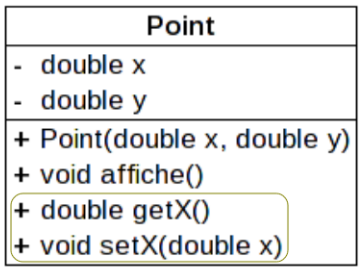
Notion d'encapsulation

L'encapsulation est l'idée de **protéger l'accès aux variables** contenues dans un objet et de ne proposer que des méthodes pour les manipuler. En respectant ce principe, toutes les variables (les attributs) d'une classe seront donc privées.

L'objet est ainsi vu de l'extérieur comme une "boîte noire" possédant certaines propriétés et ayant un comportement spécifié (les méthodes). L'ensemble des méthodes publiques forme l'interface. C'est le comportement d'un objet qui modifiera son état.

Il faut donc créer des **méthodes publiques** pour **accéder** aux **attributs privés** :

UML



C++

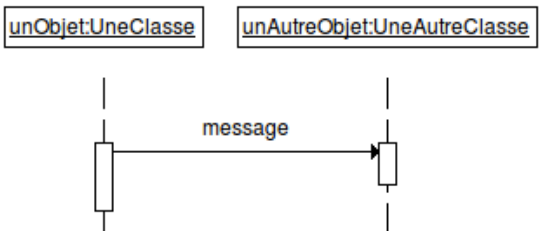
```cpp
class Point
{
   private:
      double x, y;

   public:
      Point(double x, double y);
      void affiche();
      double getX() const;
      void setX(double x);

};

// L'accesseur get du membre x
double Point::getX() const
{
   return x;
}

// Le manipulateur set du membre x
void Point::setX(double x)
{
   this->x = x;
}
```

La méthode `getX()` est déclarée constante (`const`). Une méthode constante est tout simplement une méthode qui ne modifie aucun des attributs de l'objet. Il est conseillé de qualifier `const` toute fonction qui peut l'être car cela garantit qu'on ne pourra appliquer des méthodes constantes que sur un objet constant.

La méthode publique `getX()` est un accesseur (*get*) et `setX()` est un manipulateur (*set*) de l'attribut `x`. L'utilisateur de cet objet ne pourra pas lire ou modifier directement une propriété sans passer par un accesseur ou un manipulateur.

Notion de messages

Les objets interagissent entre eux en **s'échangeant des messages**. Un objet doit être capable de répondre un **ensemble de messages**. L'ensemble des messages forme ce que l'on appelle l'**interface de l'objet**.



La réponse à la réception d'un message par un objet est appelée **une méthode**. Une **méthode est donc la mise en oeuvre du message** : elle décrit la réponse qui doit être donnée au message.

```
// L'envoie d'un message correspond à l'appel de la méthode du même nom :
unAutreObjet.message();
```

Construction d'objets

Pour **créer des objets à partir d'une classe**, il faut ... **un constructeur** :

- Un constructeur est chargé d'**initialiser un objet de la classe** ;
- Il est appelé **automatiquement au moment de la création** de l'objet ;
- Un constructeur est une **méthode qui porte toujours le même nom que la classe** ;
- Il peut avoir des paramètres, et des valeurs par défaut ;
- Il peut y avoir plusieurs constructeurs pour une même classe (surcharge) ;
- Il n'a jamais de type de retour.

On le **déclare** de la manière suivante :

```
class Point
{
   private:
      double x, y; // nous sommes des attributs de la classe Point

   public:
      Point(double x, double y); // j'ai le même nom que la classe, je suis donc le
          constructeur de la classe Point
};
```

Point.h

Il faut maintenant **définir** ce constructeur afin qu'il **initialise tous les attributs de l'objet au moment de sa création** :

```
// Je suis le constructeur de la classe Point
Point::Point(double x, double y)
{
   // je dois initialiser TOUS les attributs de la classe
   this->x = x; // on affecte l'argument x à l'attribut x
   this->y = y; // on affecte l'argument y à l'attribut y
}
```

Point.cpp

✍ *Le mot clé "**this**" permet de désigner l'adresse de l'objet sur laquelle la fonction membre a été appelée. On peut parler d'"auto-pointeur" car l'objet s'auto-désigne (sans connaître son propre nom). Ici, cela permet de différencier les attributs **x** et **y**) des paramètres **x** et **y**.*

Constructeur par défaut

Si vous essayer de créer un objet sans lui fournir une abscisse `x` et une ordonnée `y`, vous obtiendrez le message d'erreur suivant :

```
erreur: no matching function for call to Point::Point()'
```


Ce type de constructeur (`Point::Point()`) se nomme un **constructeur par défaut** (il ne possède pas de paramètres). Son rôle est de créer une instance non initialisée quand aucun autre constructeur fourni n'est applicable.

Le constructeur par défaut de la classe `Point` sera :

```
Point::Point() // Sans aucun paramètre !
{
   x = 0.;
   y = 0.;
}
```

✍ *Il est aussi possible de déclarer un constructuer par défaut en utilisant des paramètres par défaut.*

Instancier des objets

Instancier un objet revient à créer une variable d'un type classe (`class`). Une instance de classe est donc un objet.

On pourra alors **créer nos propres points** :

UML	C++
p1:Point x = 1 y = 2	`Point p1(1, 2);`
p2:Point x = 4 y = 0.	`Point p2(4, 0.);`

Les objets `p1` et `p1` sont des instances de la classe `Point`. Un objet possède sa propre existence et un état qui lui est spécifique (c.-à-d. les valeurs de ses attributs).

✍ *On accède aux membres comme pour un structure : avec l'opérateur d'accès* `.` *ou l'opérateur d'indirection* `->` *si on manipule une adresse.*

```
Lampe lampe;

// Pour allumer la lampe :
lampe.allumer();

// Et pour l'éteindre :
lampe.eteindre();
```

Allocation dynamique d'objet

Pour allouer dynamiquement un objet en C++, on utilisera l'opérateur `new`. Celui-ci renvoyant une adresse où est créé l'objet en question, il faudra un pointeur pour la conserver. Manipuler ce pointeur, reviendra à manipuler l'objet alloué dynamiquement.

Pour libérer la mémoire allouée dynamiquement en C++, on utilisera l'opérateur `delete`.

```
Point *p3; // je suis un pointeur (non initialisé) sur un objet de type Point

p3 = new Point(2,2); // j'alloue dynamiquement un objet de type Point

cout << "p3 = ";
p3->affiche(); // Comme p3 est une adresse, je dois utiliser l'opérateur -> pour accéder aux
    membres de cet objet

cout << "p3 = ";
(*p3).affiche(); // cette écriture est possible : je pointe sur l'objet puis j'appelle sa
   méthode affiche()

delete p3; // ne pas oublier de libérer la mémoire allouée pour cet objet
```

Tableau d'objets

Il est possible de conserver et de manipuler des objets dans un tableau :

```
// typiquement : les cases d'un tableau de Point
Point tableauDe10Points[10]; // le constructeur par défaut est appelé 10 fois (pour chaque
   objet Point du tableau) !
int i;

cout << "Un tableau de 10 Point : " << endl;
for(i = 0; i < 10; i++)
{
   cout << "P" << i << " = "; tableauDe10Points[i].affiche();
}
cout << endl;
```

Les objets constants

Les règles suivantes s'appliquent aux objets constants :
- On déclare un objet constant avec le modificateur `const`
- On ne peut appliquer que des méthodes constantes sur un objet constant
- Un objet passé en paramètre sous forme de référence constante est considéré comme constant

Une **méthode constante** est tout simplement une méthode qui ne modifie aucun des attributs de l'objet.

```
class Point
{
   private:
      double x, y;

   public:
      Point(double x=0, double y=0) : x(x), y(y) {}

      double getX() const; // une méthode constante
      double getY() const; // une méthode constante
      void setX(double x);
      void setY(double y);
```



```
};

// L'accesseur get du membre x
double Point::getX() const
{
   return x;
}

// Le manipulateur set du membre x
void Point::setX(double x)
{
   this->x = x;
}
```

Destruction d'objets

Le **destructeur** est une méthode membre appelée **automatiquement** lorsqu'une instance (objet) de classe cesse d'exister en mémoire :

- Son rôle est de **libérer toutes les ressources** qui ont été acquises lors de la construction (typiquement libérer la mémoire qui a été allouée dynamiquement par cet objet) ;
- Un destructeur est une **méthode qui porte toujours le même nom que la classe, précédé de "~"** ;
- Il ne possède aucun paramètre et il n'a jamais de type de retour.
- Il n'y en a qu'un et un seul.

La forme habituelle d'un destructeur est la suivante :

```
class T
{
   public:
      ~T(); // destructeur
};
```


Exercices : un long voyage

Votre voyage se passe bien mais il vous reste encore de longs jours de marche avant d'arriver à votre destination. Courage !

☞ Objectif : vous allez découvrir les structures de données en C et C++

Inscription d'étudiants

Vous voilà enfin arrivé dans la capitale régionale. Vous décidez de passer quelques temps au sein de l'université. Comme chaque année, lors de la rentrée universitaire, de nombreux étudiants viennent s'inscrire à la bibliothèque et une longue file d'attente se forme. Afin d'essayer d'accélérer les choses, les fiches d'inscription de tous les étudiants seront informatisées grâce à votre robot.

Chaque fiche contient le nom, le prénom, l'âge et le sexe (f/m) d'un étudiant. Les noms et prénoms des étudiants font moins de 50 caractères de long et doivent commencer par une lettre majuscule. On ne tiendra pas compte des espaces.

☞ Ce que doit faire votre programme :

Le programme lit le nombre de fiches à saisir, les lit et les stocke dans un tableau. On considèrera que l'on peut avoir un maximum de 500 fiches. Après avoir lu les fiches, le programme affiche le nombre de femmes, puis les informations relatives à la plus jeune personne.

Vous devez définir un type « Fiche » permettant de représenter la fiche d'un étudiant telle que décrite ci-dessus.

✍ *Le fichier d'en-tête* `ctype.h` *fournit les déclarations des fonctions* `tolower()` *et* `toupper()` *qui permettent de convertir un caractère en minuscule/majuscule.*

☞ Exemples :

```
$ ./inscription-etudiants
6
Nom : Baley
Prénom : Elijah
Sexe (f/m) : m
Age : 20
Nom : Darell
Prénom : Arcadia
Sexe (f/m) : f
Age : 21
Nom : Darell
Prénom : Bayta
Sexe (f/m) : f
Age : 27
Nom : Branno
Prénom : Harlan
Sexe (f/m) : f
Age : 25
Nom : Hardin
Prénom : Salvor
```



```
Sexe (f/m) : m
Age : 22
Nom : Riose
Prénom : Bel
Sexe (f/m) : m
Age : 22

Nombre de femmes : 3
Etudiant le plus jeune : Baley Elijah 20 m
Liste des étudiants :
Baley Elijah 20 m
Darell Arcadia 21 f
Darell Bayta 27 f
Branno Harlan 25 f
Hardin Salvor 22 m
Riose Bel 22 m
```

Question 1. Écrire le programme `inscription-etudiants-v1.c`. Tester et valider.

Afin de rendre pérenne les données stockées et pour éviter des mises à jour annuelles, on souhaite stocker la date de naissance en lieu et place de l'âge. Vous devez définir un type « Date » et modifier le type « Fiche » en conséquence. Votre programme doit continuer à fonctionner avec ce type de donnée en lieu et place de l'âge.

Question 2. Écrire le programme `inscription-etudiants-v2.c`. Tester et valider.

✍ *Le fichier d'en-tête* `time.h` *fournit les déclarations des fonctions* `time()` *et* `localtime()` *pour obtenir l'horodatage courant.*

Recette secrète

Vous voici arrivé(e) tout en haut de la montagne. Vous allez enfin pouvoir libérer le chef du village ! Vous tombez des nues lorsque vous l'apercevez en train de discuter tranquillement avec le Grand Sorcier. Loin de s'être fait kidnapper par ce dernier, il l'a rejoint pour préparer une mixture en vue de la célébration qui a lieu dans quelques jours.

La mixture en question est composée de trois ingrédients à mélanger en proportions parfaitement exactes : 5 volumes d'huile, 4 volumes d'eau, et 3 volumes d'un ingrédient secret. Le chef et le Grand Sorcier disposent de deux tonneaux non gradués de contenances 5 litres et 3 litres, avec lesquels ils pourront facilement doser l'huile et l'ingrédient secret. Mais il leur manque le tonneau de 4 litres car le chef l'a oublié au village !

Si l'on transfère le contenu d'un tonneau dans l'autre, jusqu'à avoir vidé le premier ou rempli le second, par le calcul, on peut savoir précisément combien d'eau se trouve dans chacun des deux tonneaux. Ainsi, vous vous dites qu'il doit bien y avoir un moyen d'utiliser les tonneaux disponibles pour mesurer exactement 4 litres d'eau. Vous utilisez votre robot pour chercher la solution.

☞ Ce que doit faire votre programme :

Vous vous trouvez devant une source d'eau qui jaillit de la montagne, et vous disposez de deux tonneaux vides de capacités 5 litres et 3 litres. Écrivez un programme qui effectue une série de transvasements permettant d'obtenir exactement 4 litres d'eau dans le plus grand tonneau.

Vous disposez d'une classe `Tonneau`. Quand on instancie un objet de cette classe, il faut préciser en paramètre du constructeur sa contenance en litres. Une fois créé, le tonneau est vide.

```
#include <iostream>
#include "tonneau.h"

using namespace std;

int main()
{
    Tonneau tonneau(2); // un tonneau de 2 L

    cout << "Tonneau " << tonneau.contenance() << "L : " << tonneau3.quantite() << endl;
    // Affiche : Tonneau 2L : 0

    return 0;
}
```

Instancier un tonneau

Pour doser l'eau dans les tonneaux, vous disposez de ces trois instructions :

- Remplir tonneau
- Transférer tonneauSource → tonneauDestination
- Vider tonneau

Quand on transvase un tonneau dans l'autre, on s'arrête lorsque le tonneau source est vide ou lorsque le tonneau destination est plein à ras bord. Ainsi, après chaque opération, on peut savoir exactement combien de litres d'eau se trouvent dans les deux tonneaux.

En C++, les trois instructions s'écrivent comme suit :

```
#include <iostream>
#include "tonneau.h"

using namespace std;

int main()
{
    Tonneau tonneau1(2); // un tonneau de 2 L
    Tonneau tonneau2(2); // un tonneau de 2 L

    tonneau1.remplir(); // on remplit le tonneau1

    tonneau2.remplir(tonneau1); // on transvase le tonneau1 dans le tonneau2

    tonneau2.vider(); // on vide le tonneau2

    return 0;
}
```

☞ Exemples :

Arrêtez-vous dès que le grand tonneau de 5 litres contient exactement 4 litres.

```
$ make

$ ./recette-secrete
Tonneau 5L : 4 l
```

Question 3. Écrire le programme `main.cpp`. Tester et valider.

Mastermind

Vous vous octroyez un petit moment de détente et décidez de programmer une partie de *Mastermind* avec votre robot.

Le *Mastermind* est un jeu de société pour deux joueurs dont le but est de trouver un code. C'est un jeu de réflexion, et de déduction, inventé par Mordecai Meirowitz dans les années 1970 alors qu'il travaillait comme expert en télécommunications. Il se présente généralement sous la forme d'un plateau perforé de 10 rangées de quatre trous pouvant accueillir des pions de couleurs.

Le nombre de pions de couleurs différentes est de 8 et les huit couleurs sont généralement : rouge ; jaune ; vert ; bleu ; orange ; blanc ; violet ; fuchsia. Il y a également des pions blancs et rouges (ou noirs) utilisés pour donner des indications à chaque étape du jeu.

Un joueur commence par placer son choix de pions sans qu'ils soient vus de l'autre joueur à l'arrière d'un cache qui les masquera à la vue de celui-ci jusqu'à la fin de la manche. Le joueur qui n'a pas sélectionné les pions doit trouver quels sont les quatre pions, c'est-à-dire leurs couleurs et positions. Pour cela, à chaque tour, le joueur doit se servir de pions pour remplir une rangée selon l'idée qu'il se fait des pions dissimulés.

Une fois les pions placés, l'autre joueur indique :

– le nombre de pions de la bonne couleur bien placés en utilisant le même nombre de pions rouges ;
– le nombre de pions de la bonne couleur, mais mal placés, avec les pions blancs.

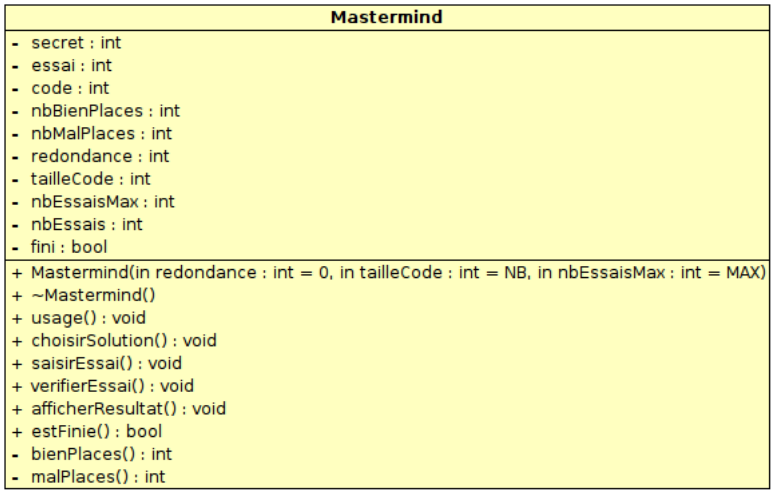
Il arrive donc surtout en début de partie qu'il ne fasse rien concrètement et qu'il n'ait à dire qu'aucun pion ne correspond, en couleur ou en couleur et position. La tactique du joueur actif consiste à sélectionner en fonction des coups précédents, couleurs et positions, de manière à obtenir le maximum d'informations de la réponse du partenaire puisque le nombre de propositions est limité par le nombre de rangées de trous du jeu.

☞ Ce que doit faire votre programme :

Les couleurs seront remplacées par des chiffres de 1 à 8. Dans ce jeu de société pour deux joueurs, on distinguera les rôles :

– le joueur qui devra deviner la combinaison secrète (couleur et position des pions) ;
– la “machine” qui assurera logiciellement le déroulement d'une manche en établissant la combinaison secrète puis en déterminant le nombre de pions de la bonne couleur bien placés et mal placés par rapport à la proposition indiquée par le joueur “humain” à chaque tour.

On vous fournit la classe `Mastermind` qui implémente le jeu :



La déclaration de cette classe se trouve dans le fichier en-tête (*header*) `Mastermind.h`. Cette classe respecte le principe d'encapsulation : l'ensemble des attributs sont déclarés privés (`private`, indiquées par un `-` dans le diagramme ci-dessus). Les méthodes publiques sont repérées par un `+`.

✍ *Les méthodes peuvent aussi être déclarées privées (c'est le cas de* `bienPlaces()` *et* `malPlaces()` *qui sont à usage interne à la classe).*

Voici une brève description de ses méthodes publiques :

– `usage()` : affiche un texte informatif sur le jeu ;
– `choisirSolution()` : détermine aléatoirement la combinaison secrète et initialise une nouvelle manche ;
– `saisirEssai()` : assure la saisie d'une combinaison proposée par le joueur ;
– `verifierEssai()` : vérifie si la combinaison proposée par le joueur correspond à la combinaison secrète et détermine le nombre de pions de la bonne couleur bien placés et mal placés ;
– `afficherResultat()` : affiche l'état du tour (le numéro de tour, le nombre de tours restant et le nombre de pions de la bonne couleur bien placés et mal placés) et de la manche si elle est finie (en dévoilant la combinaison secrète en cas de défaite du joueur) ;
– `estFinie()` : retourne vrai (`true`) si la manche est fini sinon faux (`false`).

La classe `Mastermind` possède un **constructeur** qui permet de paramètrer le type de manche à jouer en indiquant :

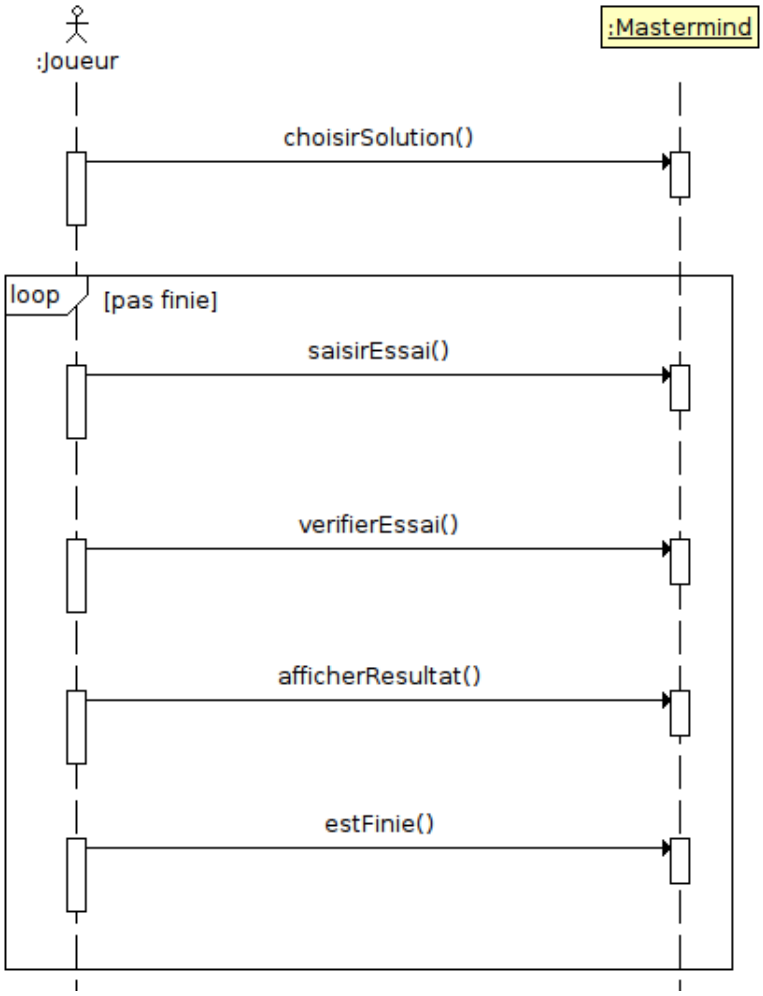
– si la redondance des couleurs est admise dans le code ;
– le nombre de pions dans le code ;
– le nombre d'essais maximum pour deviner la combinaison secrète en une manche.

```
// Valeurs par défaut
#define NB 4  // nombre de pions dans le code secret
#define MAX 10 // nombre d'essais maximum
#define REDONDANCE 1 // la redondance des couleurs dans le code est admise

Mastermind(int redondance=0, int tailleCode=NB, int nbEssaisMax=MAX);
```

La déclaration du constructeur de la classe Mastermind

Le constructeur par défaut initialisera donc une manche avec 4 pions de couleurs uniques à deviner en 10 tours maximum.

Le diagramme de séquence pour “jouer une manche” est le suivant :



☞ *Un diagramme de séquence est un diagramme d'interaction. Le but est de décrire comment les objets collaborent au cours du temps et quelles responsabilités ils assument. Il décrit un scénario d'un cas d'utilisation ou le cas d'utilisation lui-même. Un diagramme de séquence représente donc les interactions entre objets, en insistant sur la chronologie des envois de message. C'est un diagramme qui représente la structure dynamique d'un système car il utilise une représentation temporelle. Les objets, intervenant dans l'interaction, sont matérialisés par une « ligne de vie », et les messages échangés au cours du temps sont mentionnés sous une forme textuelle. Les messages sont des méthodes d'une classe et l'envoi d'un message correspond donc à l'appel de cette méthode.*

Le diagramme de séquence ci-dessus utilise un **fragment combiné** de type *loop* (une boucle) qui permet de répéter le fragment tant que la condition indiquée entre crochets (`[]`) est vérifiée. Évidemment, il existe d'autres type de fragments combinés comme *alt* (alternatif) qui est équivalent à la structure conditionnelle “`SI ... ALORS ... SINON ... FSI`”.

☞ Exemples :

Avant de jouer la partie, il est conseillé d'afficher les règles du jeu.

```
$ make
```

```
$ ./mastermind
Le Mastermind est un jeu de société dont le but est de trouver un code secret.
```



```
C'est un jeu de réflexion et de déduction, inventé par Mordecai Meirowitz dans les années
    1970.
Le nombre de couleurs différentes est de 8 (de 1 à 8).
Vous avez 10 essais pour deviner un code de 4 pions de couleurs strictement différentes.

Entrez votre essai : 1 2 3 4
Essai 1/10 : bien places 1, mal places 1
Entrez votre essai : 1 5 6 2
Essai 2/10 : bien places 0, mal places 3
Entrez votre essai : 5 2 1 7
Essai 3/10 : bien places 2, mal places 1
Entrez votre essai : 5 2 3 4
Essai 4/10 : bien places 2, mal places 0
Entrez votre essai : 5 2 8 1
Essai 5/10 : bien places 4, mal places 0
```

Question 4. Écrire le programme `main-v1.cpp` en respectant les spécifications exprimées précédemment. Tester et valider.

Question 5. Écrire le programme `main-v2.cpp` afin de jouer une manche autorisant la redondance des couleurs pour une combinaison de 5 pions en 12 tours maximum. Tester et valider.

Des histoires pour les enfants

Vous avez raccompagné le chef au village, à la grande joie de ses habitants qui vous sont très reconnaissants. Vous décidez de passer un peu de temps avec ces villageois. Les enfants du village sont très intrigués par votre langue si différente de la leur, à tel point qu'ils insistent pour que vous leur donniez des cours. Pour les y aider, vous souhaitez imprimer des histoires. Les intervenants de vos histoires sont tous des humains. Un humain est caractérisé par son nom, sa boisson favorite et sa popularité (`0` pour commencer). La boisson favorite d'un humain est, par défaut, de l'eau. Dans vos histoires, un humain pourra parler de la manière suivante :

```
(nom de l'humain) -- texte
```

Un humain pourra également se présenter : il dira bonjour, son nom, et indiquera sa boisson favorite. Il pourra boire : il dira alors « Ah ! un bon verre de (sa boisson favorite) ! GLOUPS ! ».

☞ Ce que doit faire votre programme :

Le programme doit raconter une histoire sur *Lucky Luke*. Vous devez définir une classe « Humain » pour les personnages de vos histoires. Vous utiliserez le type `string` pour les chaînes de caractères et respecterez le principe d'encapsulation.

☞ Exemples :

```
$ make

$ ./histoire-enfants
Une histoire sur Lucky Luke
(Lucky Luke) -- Bonjour, je suis Lucky Luke et j'aime le coca-cola
(Lucky Luke) -- Ah ! un bon verre de coca-cola ! GLOUPS !
```

Question 6. Écrire la classe `Humain` (`humain.h` et `humain.cpp`) afin de valider le programme fourni.


```cpp
#include <iostream>
#include "humain.h"

using namespace std;

int main()
{
   Humain lucky("Lucky Luke", "coca-cola");

   cout << "Une histoire sur " << lucky.getNom() << endl;

   lucky.sePresente();

   lucky.boit();

   return 0;
}
```

Une histoire sur Lucky Luke

Bilan

Conclusion : les types sont au centre de la plupart des notions de programmes corrects, et certaines des techniques de construction de programmes les plus efficaces reposent sur la conception et l'utilisation des types.

Rob Pike (ancien chercheur des *Laboratoires Bell* et maintenant ingénieur chez *Google*) :

> « Règle n°5 : Les données prévalent sur le code. Si vous avez conçu la structure des données appropriée et bien organisé le tout, les algorithmes viendront d'eux-mêmes. La structure des données est le coeur de la programmation, et non pas les algorithmes. »

Cette règle n°5 est souvent résumée par « Écrivez du code stupide qui utilise des données futées ! »

