IN202 - Programmation orientée objet pour les systèmes embarqués

Gabriel Busnot

CEA, List
ENSTA ParisTech

gabriel.busnot@cea.fr

23 mars 2020

Sommaire I

Sommaire II

Section 1

Introduction

Subsection 1

Le logiciel dans les systèmes embarqués

Système embarqué - Définition

Définition

Un *système embarqué* est un *système informatique spécialisé* évoluant dans un *environnement contraint* (consommation énergétique, enveloppe thermique, compatibilité électromagnétique, encombrement, etc.). Il est généralement en interraction avec d'autres systèmes embarqués ou des capteurs et actionneurs. On parle alors de *système cyberphysique*.

Système embarqué - Exemples

Quelques exemples de systèmes embarqués :

- Smartphone
- Aviation
- Spatial
- GPS
- Montre connectée
- Borne WiFi
- Automobile
  - Contrôleur moteur (ECU)
  - Système multimédia
  - Calculateurs d'aide à la conduites (Régulateur de vitesse, freinage automatique, surveillance angles morts, etc.)
  - Calculateur de suspension pilotée
  - Vision automatique (Pilote automatique, détection de piétons, lecture de panneaux)

Composition d'un système embarqué

Les 2 composantes d'un système embarqué

- Le *matériel* (hardware) : Les composants électroniques (microprocesseur, mémoire, DSP, CAN/CNA, contrôleur réseau, etc.) assemblés pour former la carte électronique en charge d'exécuter le logiciel et de s'interfacer avec l'environnement.
- Le *logiciel* (software) : Le code exécuté par les différents éléments programmables du système embarqué. Il s'agit le plus souvent de microprocesseurs mais aussi de GPU ou encore de FPGA.

Spécificité du logiciel embarqué

Les logiciels embarqués présentent des spécificités par rapport aux logiciels destinés aux ordinateurs conventionnels :

- **Performances limitées** : mémoire en Ko, Fréquence en MHz
- **Temps réel** : réaction à des événements externes dans un temps imparti
- **Fiabilité** : des vies ou du matériel coûteux peuvent être menacés par un bogue
- **Outils de développement restreints** : langages, debogueur, bibliothèques, etc.

Subsection 2

De l'assembleur au C au langages orientés objet

L'assembleur, la base du logiciel embarqué

L'*assembleur* est une *représentation lisible* du *code exécuté physiquement par un microprocesseur*. Il requiert une grande expertise, présente un risque d'erreur et de bogue élevé, limite la complexité des programmes écrivables et est souvent moins bien optimisé que le code généré par un compilateur depuis un langage tel que le C ou le C++.

```
putc:
        li a2, UART_BASE
1:      lbu a3, REG_IIR(a2)
        andi a3, a3, IIR_TX_RDY
        beqz a3, 1b
        sb a1, REG_TBR(a2)
        ret
```

Listing 1: Un exemple de code assembleur RISC-V

Le C dans l'embarqué

Le C est le langage historique de l'embarqué et est, encore aujourd'hui, le langage dominant de ce secteur. Il s'agit d'un langage *bas niveau*, en ce sens qu'il est proche du langage machine (assembleur).

```c
int main() {
    printf("hello, world");
    return 0;
}
```

Listing 2: Le programme 'Hello World !' en C

Avantages du C pour l'embarqué

Le C présente de nombreux avantages face à l'assembleur qui expliquent son adoption universelle par les développeurs embarqués :

- **Le C est portable** : un même programme peut être compilé pour plusieurs microprocesseurs différents
- **Le C est plus haut niveau que l'assembleur** : les fonctions, structures, tableaux et pointeurs sont des outils puissants absents de l'assembleur.
- **Le C est bas niveau** tout de même : Le C offre un contrôle presque aussi fin des accès mémoire que l'assembleur.
- **Le C supporte l'intégration d'assembleur** : Il est toujours possible d'écrire un morceau d'assembleur dans un programme C pour implémenter certains comportements spécifiques.

Subsection 3

Le choix du C++ dans l'embarqué

Que manque-t-il au C ?

Bien que suffisant pour programmer à peu près tout type de système, le C présente des inconvénients majeurs. Outre de nombreux détails tels que les conversions de valeurs dangereuses (typage faible), les pointeurs "bruts", les macros ou encore la gestion difficile de la mémoire allouée dynamique (malloc),

le C ne permet pas facilement d'écrire des programmes orientés objet.

La Programmation Orientée Objet (POO)

La POO est un paradigme de programmation, c'est à dire un style d'écriture de programmes. Des langages orientés objet sont C++, Java ou encore PHP. Ils offrent des *fonctionnalités* simplifiant l'utilisation du *paradigme objet*.

D'autres paradigmes (non mutuellement exclusifs) sont :

- La programmation impérative (C, C++, Python, Java, Assembleur)
- La programmation fonctionnelle (Lisp, OCaml, SCala, Python)
- La programmation logique (Prolog)

Pourquoi la POO ? I

L'objectif de la POO est de permettre d'*établir une correspondance* entre :

- des concepts réels
- et l'organistation du programme qui les manipule.

Ces concepts sont alors représentés par des *objets* qui *intéragissent entre eux via un ensemble de fonctions qui leurs sont propres*.
Ce mode d'organisation des programmes vise à :

- *simplifier leur organisation*
- en facilitant le *découpage du programme*
- et en *délimitant les modes d'interraction* entre ses parties.

Pourquoi la POO ? II

Critique d'un code C :

```
typedef struct rational{
    long num;
    long den;
} Rational; //'Rational' is the same type as 'struct rational'

Rational addRationals(Rational a, Rational b){
    Rational res;
    res.num = a.num * b.den + b.num * a.den;
    res.den = a.den * b.den;
    return res;
}

int main(){
    Rational a = {1, 2}, b = {3, 0}; // a = 1/2 ; b = 3/0
    Rational c = addRationals(a, b); // c = 6/0
    int cIntegerPart = c.num / c.den; // crash
    return 0;
}
```

Pourquoi la POO ? III

Faiblesses du code précédent :

- `b` a pu être initialisé avec un dénominateur nul qui cause un crash par la suite. Solution : aucune car l'utilisateur est maître de l'initialisation de `Rational`. Éviter ce type d'erreur relève exclusivement de sa vigilence et de sa bonne lecture de la documentation de `Rational`, s'il y en a une.
- `Rational` ne peut plus être modifié sans demander de potentiellement grands changements aux utilisateurs de cette structure. Solution : aucune car si un utilisateur peut utiliser une structure en C, alors il peut utiliser chacun de ses champs directement. Toute modification de la structure l'obligerait alors à modifier son code.

Pourquoi la POO ? IV

Faiblesses du code précédent (suite) :

- Il serait logique de nommer `addRationals` simplement `add`. C'est risqué car si, par exemple, une fonction `addComplex` existe ailleurs et qu'elle est également renommée `add`, il y aurait un conflit de nom. solution : aucune car il n'y a aucun moyen d'avoir deux fonctions avec le même nom en C.
- Il est impossible de contraindre l'utilisation de `Rational` pour empêcher sa mauvaise utilisation (e.g. dénominateur nul). L'utilisateur pourrait alors corrompre son état, c'est à dire donner aux champs de la structure des valeurs incohérentes (volontairement ou pas).
- Par conséquent, il n'est pas possible de faire d'hypothèse sur les valeurs contenues par Rational. Par exemple, il est impossible de supposer qu'un Rational est toujours sous forme réduite.

Pourquoi la POO ? V

Le même code en orienté objet (C++) :

```
class Rational{
private: //Everything after this line is inaccessible to code outside Rational
    long num;
    long den;
public: //Everything after this line is accessible to code outside Rational
    Rational(long n, long d){ //This is the function called at initialization
        if(d == 0){exit(-1);} //Initializing with nul denominator gives an error
        long p = pgcd(n, d);  //We assume that function pgcd exists
        num = n / p;
        den = d / p;
    }
    Rational add(Rational r){
        long n = num * r.den + num * r.den;
        long d = den * r.den;
        return Rational(n, d);
    }
    long integerPart(){
        return(num / den); //Never fails as den cannot equal 0
    }
};

int main(){
    Rational a(1, 2), b(3, 4); // a = 1/2 ; b = 3/4
    Rational c = a.add(b); // c = 10/8
    long cIntegerPart = c.integerPart();
    return 0;
}
```

Pourquoi la POO ? VI

Les faiblesses énumérées précédement ont été corrigées par l'approche objet :

- Le dénominateur d'un `Rational` ne peut plus être nul.
- Les champs contenus par la classe `Rational` peuvent être modifiés ou ramplacés au besoin sans que l'utilisateur de la classe ne soit impacté car il n'y a pas accès. Seules les fonctionnalités de la classe (qui restent inchangées) lui sont accessibles.
- La fonction d'addition s'appelle seulement `add`, ce qui est suffisant puisqu'elle ne peut être appellée que sur des `Rational`. En pratique, il est même possible de définir l'opérateur '`+`' pour pouvoir écrire '`a + b`', où `a` et `b` sont des `Rational`.

Pourquoi la POO ? VII

- Il est impossible de mal utiliser la classe `Rational` puisque les possibilités d'actions sur `Rational` sont limitées aux fonctions membres publiques de la classe. En particulier, il n'est pas possible d'utiliser `num` et `den` individuellement.
- Un `Rational` est toujours mis sous forme réduite lors de sa construction. Il est ensuite possible de supposer cela sans risque dans les fonctions membres de `Rational`. Celles-ci doivent cependant garantir qu'elles préserveront cet hypothèse lorsqu'elles sont appelées.

L'encapsulation

Un principe fondamental de la POO a été appliqué dans l'exemple précédent : l'encapsulation.

Encapsulation

Consiste à rendre innaccessibles (`private`) les données relatives à l'implémentation d'une fonctionnalité pour ne laisser accès (`public`) qu'à des fonctions spécifiques constituant l'interface. Les objets deviennent ainsi des boîtes noires ne laissant accessible que leur interface et pas leurs composants internes. Ce principe doit être appliqué systématiquement en POO !

La compréhension du principe d'encapsulation est le coeur de ce cours et prévaudra sur la technicité dont vous sauriez faire preuve par ailleurs.

Section 2

Rappels de C valables en C++

Subsection 1

Hello world

Hello World !

Programme minimal en C.
Compilation : `gcc main.c -o hello_world`

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    //argc is the number of arguments passed to the program
    //argv is the list of arguments as strings of characters
    //they are optional
    printf("Hello World!");
    return 0;
}
```

Subsection 2

Variables

Variables

```c
#include <stdio.h>

int main()
{
    char c = 'A'; //8 bits signed integer representing the character 'A'
    int a = -42; //Usually a 32 bits signed integer
    unsigned int b = 42; //Usually a 32 bits unsigned integer
    long l = 100; //usually 32 or 64 bits signed integer (can be unsigned)
    float f = 1.5; //A 32 bits floating point value
    double d = 0.001 //A 64 bits floating point value

    printf("Hello World!");
    return 0;
}
```

Subsection 3

Structures de contrôle

Structure de contrôle : if else

```c
#include <stdio.h>

int main()
{
    int a = 1, b = 2;
    //Conditions can be combined using && and ||
    if(a > b){
        printf("|a - b| = %d\n", a - b);
    } else if(a < b) {
        printf("|a - b| = %d\n", b - a);
    } else {
        printf("|a - b| = 0\n");
    }
    return 0;
}
```

Structure de contrôle : switch

```c
#include <stdio.h>
enum gender{
    MALE, FEMALE
};
int main()
{
    char name[] = "Smith";
    gender g = FEMALE;
    //The switch can only be used on integers or enums
    switch(g){
    case MALE:
        printf("Hello Mr. %s!\n", name);
        break;
    case FEMALE:
        printf("Hello Mrs. %s!\n", name);
        break;
    default:
        printf("Hello Mr. or Mrs. %s!\n", name);
    }
    return 0;
}
```

Subsection 4

Boucles

Boucle : while

```c
#include <stdio.h>

int main()
{
    int n = 2048, temp = n;
    int log2n = 0;
    while(temp > 1){
        temp /= 2; //temp = temp / 2
        log2n++;   //log2n = log2n + 1
    }
    printf("log_2(%d) = %d\n", n, log2n);
}
```

Boucle : for

```c
#include <stdio.h>

int main()
{
    int t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int size = sizeof(t) / sizeof(t[0]); //size = 10
    int sum = 0;
    //for(init ; continuation condition ; inter-loop action)
    for(int i = 0 ; i < size ; ++i){
        sum += t[i];
    }
    printf("avg = %f\n", ((float)sum) / size);
}
```

Subsection 5

Fonctions

Fonctions

```c
#include <stdio.h>

//returnType funcName(argType argName, ...)
unsigned long pow(long n, unsigned long p){
    if(p == 0) return 1;
    unsigned long val = pow(n, p/2);
    val *= val;
    if(p % 2 == 0) return val;
    else return val * n;
}

int main()
{
    long n = 10;
    unsigned long p = 5;
    printf("%ld^%lu = %lu\n", n, p, pow(n, p));
}
```

Subsection 6

Pointeurs

Les pointeurs

- Un pointeur est une variable contenant l'adresse en mémoire d'une autre variable.
- Un pointeur est toujours de taille 32 bits sur les systèmes 32 bits et 64 bits sur les systèmes 64 bits.
- Un pointeur sur une variable de type `T` est de type `T*`.
- L'adresse d'une variable `x` est obtenue par l'opérateur préfixe `&` : `&x`.
- La valeur pointée par un pointeur `p` est obtenues par l'opérateur préfixe `*` : `*p`.
- Par convention, un pointeur ne pointant sur rien doit avoir la valeur `NULL` en C et `nullptr` en C++.

Les pointeurs - exemple

```
int main(){
    int n = 42;
    int* nptr = NULL; //NULL is 0 in C
    nptr = &n;
    printf("n equals %d\n", n); //42
    printf("n is at address %p\n", &n); //0x7f9e480990ac
    printf("nptr points to address %p\n", nptr); //0x7f9e480990ac
    printf("Value pointed by nptr is %d\n", *nptr); //42
    *nptr = 43; // n = 43
    printf("Value pointed by nptr is %d\n", *nptr); //43
    printf("n equals %d\n", n); //43
    return 0;
}
```

Les pointeurs - Utilisation

Utilité des pointeurs en C :

- Passer une structure de grande taille à une fonction sans avoir à la copier.
- Permettre à une fonction de modifier les variables passées en argument.
- Accéder à la mémoire alloué dynamiquement.
- Implémenter les tableaux.

Allocation dynamique I

En C, l'allocation dynamique de mémoire se fait via `malloc` et sa libération via `free`. En C++, deux mots clefs remplacent ces fonctions : **`new`** et **`delete`**.

Par exemple, "`int* p = new int(10);`" alloue un entier initialisé à 10 et "`delete p;`" le libère.
L'exemple suivant montre leur utilisation dans leur variante tableau (avec `[]`).

Allocation dynamique II

```
//The modifications made to t will be visible by the caller
void rotate(int* t, int size){
    if(size <= 1) return;
    int temp = t[size-1];
    for(int i = size-1 ; i > 0 ; i--){
        t[i] = t[i-1];
    }
    t[0] = temp;
}
int main(){
    int* tab = new int[10]; //Dynamically allocated array of 10 int
    for(int i = 0 ; i < 10 ; i++) tab[i] = i;
    rotate(tab, 10); //tab = [9,1,2,3,4,5,6,7,8]
    delete[] tab;
    return 0;
}
```

Subsection 7

Fichier d'en-tête et fichier d'implémentation

Fichier d'en-tête et fichier d'implémentation I

La déclaration des fonctions (prototype) et leur définition (corps de la fonction) sont généralement séparées dans deux fichiers différents : le *fichier d'en-tête (header)* et le *fichier d'implémentation*.

```
//func.h (header)
int foo(int a, float b); //Declaration

//func.c (implementation)
int foo(int a, float b){
    //Definition
}

//main.c
#include "func.h"
int main(){
    int n = foo(1, 42.0);
    return 0;
}
```

Fichier d'en-tête et fichier d'implémentation II

Un fichier d'en-tête doit toujours être protégé contre l'*inclusion multiple*.
On utilise pour cela une garde d'inclusion comme suit :

```
//header.h
#ifndef HEADER_H
#define HEADER_H

//Source code here

#endif
```

Si le fichier `heaeder.h` est inclus plusieurs fois, son comptenu ne sera effectivement inclus que lors de la première directive `#include`.

Subsection 8

Compilation

Options de compilation utiles

Pour compiler un ensemble de fichiers : `g++ <src0> <src1> -o <out>`
Options utiles :

- `-Wall -Wextra -pedantic` : Plus de warning, pour votre bien. Un warning n'est jamais anodin.
- `-c` : Générer un fichier objet (`.o`), c'est à dire un produit de compilation intermédiaire qui pourra être joint à d'autres pour former un exécutable. Permet de diviser la compilation en étapes pour ne pas recompiler l'intégralité d'un projet à chaque changement. Voir Makefile.
- `-std=c++14` : Utiliser le standard C++14 qui est plus moderne.
- `-O3` : Optimisation de niveau 3 (0 par défaut, 3 maximum)
- `-g` : Insérer les informations de déboguage. Voir techniques de debug.

Section 3

Les classes en C++

Subsection 1

Les struct du C améliorées

Les classes, concept de base du C++

Une classe en C++ est une structure du C à laquelle ont été ajoutés, entre autre, les éléments suivants :

- Fonctions membres (ou méthodes) : des fonctions pouvant être appellées *sur* les objets instanciés à partir de cette classe pour les manipuler.
- Visibilité des champs : il est possible de cacher ou non les variables et fonctions membres d'une classe via les mots clefs `public` et `private`.
- Relation d'héritage : une classe peut hériter des composants d'une autre.
- Constructeur et destructeur : fonctions spéciales en charge de l'initialisation et de la destruction d'un objet respectivement lors de sa création et de sa suppression.

Subsection 2

Déclaration de classe et instanciation d'objet

Déclaration et instanciation de classe

```
//myclassname.hpp
class MyClassName{
private: //Optional, members are private by default in a class
    int aPrivateMemberVariable;
    float anotherPrivateMemberVariable;
public:
    bool aPublicMemberVariable; //Public member variable, very rare

    int aPublicMemberFunction(char str[]);
};

//main.cpp
#include <myclassname.hpp>
int main(){
    // MyClassName is the 'class' of the 'object' anInstance
    MyClassName anInstance;
}
```

Subsection 3

Constructeur et destructeur

Constructeur

Un objet doit être initialisé lorsqu'il est créé :

1. Initialization des variables membres (attributs)
2. Initialisation des classes héritées
3. Réalisation d'opérations supplémentaires
    1. Allocation de mémoire
    2. Ouverture de fichier
    3. etc.

C'est le rôle du *constructeur*. Il s'agit d'une fonction membre spéciale ne retournant aucune valeur et portant le même nom que la classe qui le contient.

Constructeur - Exemple

```
//rational.hpp
class Rational{
    long a;
    unsigned long b;
public:
    //Here is the constructor (no return type, same name as class)
    Rational(long aVal, unsigned long bVal);
};

//rational.cpp
#include "rational.hpp"
//The intializer list is used to initialize a to aVal and b to bVal
// '::' specifies the namespace : Rational inside Rational
Rational::Rational(long aVal, unsigned long bVal):
    a(aVal), b(bVal)
{
    //Here is an extra action, for the sake of example
    printf("Rational initialized to %l/%lu\n", aVal, bVal);
}
```

Constructeur - Liste d'initialisation I

Dans l'exemple précédent, la *liste d'initialisation* a été utilisée pour initialiser les variables membres de `Rational`. Elle est située entre le symbole ":" et le corps du constructeur dans l'exemple ci-dessous.

```
Rational::Rational(long aVal, unsigned long bVal):
    a(aVal), b(bVal) //initializer list
{
    //Would work in this particular case but probably slower
    // a = aVal;
    // b = bVal;
}
```

Constructeur - Liste d'initialisation II

L'utilisation de la liste d'initialisation est toujours préférable à l'initialisation dans le *corps* du constructeur (i.e. entre les {}) car :

- Lors de l'entrée dans le corps du constructeur, les variables sont toutes initialisées. Les "réinitialiser" ici revient parfois à écrire une seconde fois dessus.
- Certains éléments d'une classe ne peuvent pas être initialisés autrement :
  - Les attributs constants
  - Les attributs objets (i.e. dont le type est une classe) : l'appel de leur constructeur n'est possible que dans la liste d'initialisation
  - Les attributs référence : abordés plus tard
  - Les classes héritées : abordées plus tard

Constructeur - Initialisation d'attributs objets

```
class Coordinates{
    Rational x, y; //Must be initialized with their constructors
public:
    //Inline declaration (i.e. inside the class definition in the header)
    Coordinates::Coordinates(long x1, long y1, long x2, long y2):
        x(x1, y1), y(x2, y2) //Calls the constructor of Rational
    {
        //Nothing to do in the body
    }
}
```

Destructeur

De même qu'un objet est initialisé lors de sa création, il est détruit lors de sa suppression. Cette étape permet généralement de libérer des ressources acquises lors de l'initialisation de l'objet :

- Libération de mémoire
- Fermeture de fichier
- etc.

Ce rôle est confié à une fonction membre spéciale appelée le `destructeur`. Cette fonction spéciale ne retourne rien, ne prend pas d'argument et possède le même nom que la classe à laquelle elle appartient préfixée d'un `~`.

Destructeur - Exemple

```
//rational.hpp
class Rational{
    long a;
    unsigned long b;
public:
    Rational(long aVal, unsigned long bVal);
    //Here is the destructor
    //(no return type, no arguments, same name as class with ~)
    ~Rational()
};

//rational.cpp
#include "rational.hpp"
//The intializer list is used to initialize a to aVal and b to bVal
Rational::Rational(long aVal, unsigned long bVal):
    a(aVal), b(bVal)
{}
Rational::~Rational()
{
    //For the sake of example, just an informative print.
    printf("Rational deleted\n");
}
```

Appels de constructeur et du destructeur

Le constructeur est appelé lors de la déclaration d'un objet. Le destructeur est appelé lors de la sortie du *scope* auquel appartient l'objet.

Le *scope* d'une variable correspond généralement à un *bloc* délimité par des accolades `{...}`. Par exemple, le corps d'une fonction ou d'un `if` est un *bloc*. Il est aussi possible de créer un *bloc* n'importe où en ouvrant des accolades.

```
#include "rational.hpp"
int main(){
    Rational r1(1, 2); //The constructor is called here for r1
    { //Block creation
        Rational r2(2, 3); //The constructor is called here for r2
    } //The destructor is called here for r2
} //The destructor is called here for r1
```

Subsection 4

Visibilité des membres d'une classe

Visibilité des membres d'une classe

Les membres d'une classe ne sont pas tous *visibles* depuis l'extérieur de cette classe. L'utilisation de la visibilité est fondamentale à l'encapsulation.

Il existe trois niveaux de visibilité :

- `private:` Seules les fonctions membres de la classe y ont accès.
- `public:` Tout le monde y a accès.
- `protected:` Seule les fonctions membres de la classe et de ses classes dérivées y ont accès. Utilisation découragée.

Visibilité des membres d'une classe - Exemple

```
//rational.hpp
class Rational{
private: //Optional as the members of a class are private by default
    long a;
    unsigned long b;
public: //Everything is public below this line
    Rational(long aVal, unsigned long bVal);
    Rational mult(Rational other);
};

//rational.cpp
#include "rational.hpp"
//The constructor of Rational belongs to Rational: it can access its private members
Rational::Rational(long aVal, unsigned long bVal):
    a(aVal), b(bVal)
{}
Rational Rational::mult(Rational other){
    //private members of other objetcs can also be accessed
    return Rational(a * other.a, b * other.b);
}

//main.cpp
#include "rational.hpp"
int main(){
    Rational r1(1, 2);
    //long num = r1.a; //Won't compile as 'a' is private in Rational
    Rational r2 = r1.mult(r1); //ok: mult is public
}
```

Subsection 5

Le pointeur this

Le pointeur `this` I

Lors de l'appel d'une méthode sur un objet, les variables membres de cet objet sont directement accessibles à cette méthode. Cependant, comment accéder à l'objet lui-même ? On pourrait vouloir le passer en argument d'une autre fonction par exemple.

C'est le rôle du pointeur `this`. Ce pointeur spécial (mot clef du langage C++) est utilisable dans le corps des fonctions membres d'une classe. `this` est alors l'adresse de l'objet sur lequel a été appelée la fonction membre (et donc `*this` est l'objet lui même).

Le pointeur this II

```
class Something{
public:
    void printAddress(){
        printf("Something stored at address %p\n", this);
    }
};
int main(){
    Something s;
    s.printAddress();
}
```

Subsection 6

Membres statiques

Membres de classe statiques

Les membres d'une classe ne sont accessibles que par le biais d'un objet instancié à partir de cette classe via le ".". Il est cependant possible de déclarer des variables et des fonctions membres d'une classe accessibles sans disposer d'un objet de cette classe : ce sont les *membres statiques*.

Il sont alors *uniques* et *globaux* (partagés par tout le programme). Leur visibilité peut également être modifiée via les mots clef `private` et `public`.

Membres de classe statiques - Exemple

```
class SelfCountingClass{
    static int i;
public:
    SelfCountingClass(){
        //i is global so this modification will be persistent
        i++;
    }
    static int getCount(){
        return i;
    }
};
//Static class member initialization
//must be put somewhere once and only once outside of a header
int SelfCountingClass::i = 0;

int main(){
    //getCount can be called without an object
    int c0 = SelfCountingClass::getCount();
    SelfCountingClass scc;
    int c1 = SelfCountingClass::getCount();
    printf("Count before objet instanciation: %d\n", c0);
    printf("Count after objet instanciation: %d\n", c1);
}
```

Subsection 7

Héritage simple

Héritage simple

Une autre notion primordiale de la POO est l'*héritage*. Il s'agit pour une classe d'être définie sur la *base* d'une autre classe. Elle *hérite* alors de ses membres (variables et fonctions).

L'héritage a, au même titre que les membres, une visibilité parmis `public`, `protected` et `private`. L'usage courant est l'héritage public qui préserve strictement la visibilité des membres.

Si la classe `Derived` hérite publiquement de la `Base` :

- `Derived` peut accéder à tous les membres non privés de `Base` (i.e. `public` ou `protected`).
- Les membres publics de `Base` et `Derived` sont accessibles de l'extérieur.

Il est aussi possible d'hériter de plusieurs classes en même temps : c'est l'*heritage multiple*. Il présente un certain nombre de subtilités et ne nous sera pas utile.

Héritage Simple - Exemple

```
//base class
class Animal{
    char mName[32];
public:
    //The list initializer cannot be used here because we need strcpy
    Animal(char* name){strcpy(mName, name);}
    char* getName(){return mName;}
};
//Derived class
class Cat: public Animal{
public:
    //The constructor of Cat calls the constructor of Animal
    Cat(char* name): Animal(name){}
    //meow has to use the inherited method getName
    void meow(){printf("%s: meow meow\n", getName());}
};

int main(){
    Cat felix("Felix");
    //Both getName and meow can be called on a Cat
    char* name = felix.getName();
    felix.meow();
}
```

Section 4

Plus de C++

Subsection 1

Introduction à la Standard Template Library (STL)

La STL

La STL est la bibliothèque standard du langage C++. Elle fournit entre autre :

- Des conteneurs génériques (pour tous types de données) :
  - Chaîne de caractère (`string`)
  - Tableau dynamique (`vector`)
  - Liste chaînée (`list`)
  - Conteneur associatif (type dictionnaire en Python) (`map`)
  - Ensemble (`set`)
  - etc.
- Des algorithmes à appliquer dessus :
  - Tri (`sort`)
  - Recherche (`find`)
  - Comptage (`count`)
  - etc.
- D'autres fonctionnalités spécifiques
  - Threads (`thread`)
  - Manipulation de fichiers (`fstream`)
  - etc.

Le vecteur, roi de la STL I

Le conteneur le plus utilisé et le plus polivalent de la STL est `vector`. Il s'agit d'une classe représentant un tableau dynamique, c'est à dire une tableau dont la taille peut changer au cours de l'exécution d'un programme. Afin de créer un vecteur, une syntaxe particulière doit être utilisée :

- Le préfixe `std::` doit être utilisé car toutes les classes et fonctions de la STL sont dans l'*espace de nommage (namspace)* `std`.
- Le type d'objets contenus par le vecteur doit être précisé entre <>. La classe `std::vector` est en effet écrite de manière à fonctionner avec n'importe quel type fourni en paramètre. Il s'agit d'un paramètre dit "de *template*".

```
#include <vector>
//Create an empty vector of int
std::vector<int> vi;
//Create a vector of 10 doubles filled with 42
std::vector<double> vd(10, 42);
```

Le vecteur, roi de la STL II

Il est alors possible d'ajouter des éléments dans un vecteur ou d'en retirer.

```
#include <vector>
#include <cstdio>

int main(){
    std::vector<int> vi(3, 0);  //vi = [0, 0, 0]
    vi.push_back(1);            //vi = [0, 0, 0, 1]
    vi.push_back(-1);           //vi = [0, 0, 0, 1, -1]
    vi.pop_back();              //vi = [0, 0, 0, 1]
    vi.push_back(42);           //vi = [0, 0, 0, 1, 42]
    for(int i(0) ; i < vi.size() ; i++){
        printf("vi[%d] = %d\n", i, vi[i]);
    }
}
```

Subsection 2

Les références

Les références

En C++, les pointeurs sont souvent remplacés par des *références*.

- Une référence est un *alias*, i.e. un autre nom donné à un objet existant.
- Une référence sur un objet de type `T` est de type `T&`.
- Une référence s'utilise ensuite exactement comme l'objet qu'elle référence.

```
int main(){
    int i(42); //An "object" of type int
    int& ri(i); //A reference to i
    // ri can be used wherever i can (and vice versa)
    printf("i = %d, ri = %d\n", i, ri);
    printf("&i = %p, &ri = %p\n", &i, &ri);
}
```

Les références

Les références sont le plus souvent utilisées en paramètre de fonction pour éviter des copies inutiles.

```
#include <vector>
//v is not copied but a reference is created
float avg(std::vector<int>& v){
    long sum(0);
    for(int i(0) ; i < v.size() ; i++){
        sum += v[i];
    }
    return sum / float(v.size());
}
int main(){
    std::vector<int> v;
    for(int i(0) ; i < 1000 ; i++){
        v.push_back(i*i);
    }
    float avg_v(avg(v));
}
```

Les références - pourquoi ?

Pourquoi utiliser des références alors que les pointeurs permettent de faire à peu près la même chose ?
Car les références sont beaucoup plus difficiles à mal utiliser et plus simple à bien utiliser :

- Une référence est toujours valide [1]. Alors qu'un pointeur peut être nul, une référence doit être initialisée lors de sa déclaration à l'aide d'un objet valide.
- Une référence ne peut pas changer d'objet référencé.
- Lorsqu'on manipule une référence, c'est toujours l'objet référencé qui est en fait manipulé. Pas de confusion entre objet et adresse de l'objet.

1. Sauf manipulations volontaires très hazardeuses.

Subsection 3

La const correctness

La const correctness, qu'est-ce ?

Une notion importante de la POO est la *const correctness*. Il s'agit de déclarer constantes toutes les variables et méthodes qui... sont constantes.

- Une variable constante est une variable qui ne change pas de valeur.
- Une méthode constante est une méthode qui :
  1. Ne modifie pas la valeur des variables membres de `*this`.
  2. N'appelle aucune méthode non constante sur `*this`.
  3. Ne passe *this et ses membres à d'autres fonctions que par valeur ou référence constance.

L'application de la const correctness offre des garanties utiles à l'utilisateur et diminue le risque d'erreur de programmation.

La const correctness, exemple I

```
#include <vector>
//v cannot be const as it is modified
void pushMultiple(std::vector<int>& v, int val, int n){
    for(int i(0) ; i < n ; i++){
        v.push_back(val);
    }
}
int main(){
    const std::vector<int> vconst;
    std::vector<int> v;
    // Will not compile as it "discards the const qualification of vconst"
    // pushMultiple(vconst, 10, 20);
    pushMultiple(v, 10, 20); //Ok
}
```

La const correctness, exemple II

```
//v must be const or the function cannot be called on const vectors
float avg(const std::vector<int>& v){
    long sum(0);
    for(int i(0) ; i < v.size() ; i++){
        sum += v[i];
    }
    return sum / float(v.size());
}
int main(){
    std::vector<int> v(10, 20);
    //vconst is copied from v at declaration but won't be modifiable
    const std::vector<int> vconst(v);
    avg(v); //ok
    avg(vconst); //ok
}
```

La const correctness, exemple III

```
class Complex{
    double a, b;
public:
    Complex(double a, double b): a(a), b(b){}
    // These two methods do not modify the object,
    // then they are const or they cannot be called on const Complex
    double real() const {return a;}
    double imag() const {return b;}
};
```

Subsection 4

La surcharge de fonctions

La surcharge de fonctions I

En C++, plusieurs fonctions ou méthodes (y compris constructeurs) peuvent avoir le même nom à condition qu'elles aient des arguments différents. La détermination de la fonction appelée sera alors faite d'après le type des arguments.

Une règle d'or : Deux fonctions avec le même nom doivent faire la même chose !

La surcharge de fonctions II

```
void print(const Complex& c){ //A
    printf("%f + i%f", c.real(), c.imag());
}

void print(const std::vector<int>& v){ //B
    for(int i(0) ; i < v.size() ; ++i){
        printf("%d, ", v[i]);
    }
}

int main(){
    Complex c(1, 2);
    std::vector v;
    v.push_back(1);
    v.push_back(2);
    print(c); //Calls A
    print(v); //Calls B
}
```