Programmation Objet & Modélisation

Loïc Demange
loic.demange@umontpellier.fr

Polytech, Montpellier, France

13 octobre 2025

Plan

Préliminaires

Nous allons étudier le C++, mais vu que c'est basé sur le langage C, nous allons faire des petits rappels dans un premier temps.

Le C est un langage de programmation impératif et compilé, inventé en 1972 dans les laboratoires Bells.

Il a continué à être mis à jour à travers des normes tels que le C89, C90, C95, C99, C11, C18 et C23.

Préliminaires - les types

Le C offre une liste de type primitif, permettant de représenter nos données.

Préliminaires - les types

Le C offre une liste de type primitif, permettant de représenter nos données.

- `short`,

Préliminaires - les types

Le C offre une liste de type primitif, permettant de représenter nos données.

- `short`,
- `int`,

Préliminaires - les types

Le C offre une liste de type primitif, permettant de représenter nos données.

- `short`,
- `int`,
- `long`,

Préliminaires - les types

Le C offre une liste de type primitif, permettant de représenter nos données.

- `short`,
- `int`,
- `long`,
- `char`,

Préliminaires - les types

Le C offre une liste de type primitif, permettant de représenter nos données.

- `short`,
- `int`,
- `long`,
- `char`,
- `float`,

Préliminaires - les types

Le C offre une liste de type primitif, permettant de représenter nos données.

- `short`,
- `int`,
- `long`,
- `char`,
- `float`,
- `double`,

Préliminaires - les types

Le C offre une liste de type primitif, permettant de représenter nos données.

- `short`,
- `int`,
- `long`,
- `char`,
- `float`,
- `double`,
- `bool` (depuis C23, `_Bool` de `stdbool.h` avant).

Préliminaires - les types

Si on souhaite stocker et manipuler des valeurs, il nous faut déclarer une variable, en spécifiant le type de données et un nom, par ex. `int x;` pour une variable `x` stockant un entier.

On peut aussi la définir à la déclaration, `int x = 4;`.

Ou affecter une valeur après déclaration,

```
int x;
x = 4;
```

Préliminaires - les types

Ces variables contiennent une valeur, et cette valeur est stockée dans un emplacement mémoire, identifiée par une adresse.

`&x` renvoie l'adresse où est stockée la valeur de `x`. Cette adresse peut être stockée elle-même dans une variable, de type pointeur.

Si on souhaite stocker l'adresse de `x`, on peut écrire `int *y = &x;`.
`y` est un pointeur d'entier, ce qui signifie qu'elle peut contenir l'adresse d'une variable de type entier. On peut récupérer la valeur présente dans l'adresse stockée dans `y` en le résolvant avec $*$. Donc `*y == x`.

Préliminaires - les types

Si on reprend nos types précédents,

Préliminaires - les types

Si on reprend nos types précédents,

- `short *` permet de stocker l'adresse d'un `short`,

Préliminaires - les types

Si on reprend nos types précédents,

- `short *` permet de stocker l'adresse d'un `short`,
- `int *` permet de stocker l'adresse d'un `int`,

Préliminaires - les types

Si on reprend nos types précédents,

- `short *` permet de stocker l'adresse d'un `short`,
- `int *` permet de stocker l'adresse d'un `int`,
- `long *` permet de stocker l'adresse d'un `long`,

Préliminaires - les types

Si on reprend nos types précédents,

- `short *` permet de stocker l'adresse d'un `short`,
- `int *` permet de stocker l'adresse d'un `int`,
- `long *` permet de stocker l'adresse d'un `long`,
- `char *` permet de stocker l'adresse d'un `char`,

Préliminaires - les types

Si on reprend nos types précédents,

- `short *` permet de stocker l'adresse d'un `short`,
- `int *` permet de stocker l'adresse d'un `int`,
- `long *` permet de stocker l'adresse d'un `long`,
- `char *` permet de stocker l'adresse d'un `char`,
- `float *` permet de stocker l'adresse d'un `float`,

Préliminaires - les types

Si on reprend nos types précédents,

- `short *` permet de stocker l'adresse d'un `short`,
- `int *` permet de stocker l'adresse d'un `int`,
- `long *` permet de stocker l'adresse d'un `long`,
- `char *` permet de stocker l'adresse d'un `char`,
- `float *` permet de stocker l'adresse d'un `float`,
- `double *` permet de stocker l'adresse d'un `double`,

Préliminaires - les types

Si on reprend nos types précédents,

- `short *` permet de stocker l'adresse d'un `short`,
- `int *` permet de stocker l'adresse d'un `int`,
- `long *` permet de stocker l'adresse d'un `long`,
- `char *` permet de stocker l'adresse d'un `char`,
- `float *` permet de stocker l'adresse d'un `float`,
- `double *` permet de stocker l'adresse d'un `double`,
- `bool *` permet de stocker l'adresse d'un `bool`.

Préliminaires - les types

Si on reprend nos types précédents,

- `short *` permet de stocker l'adresse d'un `short`,
- `int *` permet de stocker l'adresse d'un `int`,
- `long *` permet de stocker l'adresse d'un `long`,
- `char *` permet de stocker l'adresse d'un `char`,
- `float *` permet de stocker l'adresse d'un `float`,
- `double *` permet de stocker l'adresse d'un `double`,
- `bool *` permet de stocker l'adresse d'un `bool`.

En réalité, la taille d'une adresse est identique quel que soit le type. Il est important pour savoir ce qu'on manipule, et lorsqu'on récupère la valeur stockée à cette adresse.

Préliminaires - les types

On peut définir des tableaux, qui seront définis par un type et une taille.

Par exemple, si on souhaite un tableau de caractères de 10 cases, on écrit `char tab[10];`.

En réalité, toutes les cases sont contiguës en mémoire, et `tab` renvoie l'adresse mémoire du début du tableau, soit de la première case.
Pour accéder à sa valeur, on met l'opérateur crochet [] avec le numéro de case, allant de 0 à taille - 1.
`tab[7]` permet d'accéder à la valeur de la 8ème case de `tab`.

Préliminaires - les types

On a vu que lorsqu'on déclare un tableau `tab`, `tab` est en réalité l'adresse de début de tableau.

On peut donc la stocker dans un pointeur, tel que `char *t = tab`, ce qui nous sera utile par la suite, pour par exemple passer un tableau un paramètre de fonction.

Préliminaires - les structures de contrôle

On a différentes structures de contrôle, dont

Préliminaires - les structures de contrôle

On a différentes structures de contrôle, dont

- le `if(a) {} else {}`, qui permet de réaliser les instructions du premier bloc si la condition `a` du `if` est vraie (≥ 1), sinon les instructions du second bloc,

Préliminaires - les structures de contrôle

On a différentes structures de contrôle, dont

- le `if(a) {} else {}`, qui permet de réaliser les instructions du premier bloc si la condition `a` du `if` est vraie (≥ 1), sinon les instructions du second bloc,
- le `while(a){}`, qui réalise les instructions du bloc en boucle tant que la condition `a` du `while` est vraie,

Préliminaires - les structures de contrôle

On a différentes structures de contrôle, dont

- le `if(a) {} else {}`, qui permet de réaliser les instructions du premier bloc si la condition `a` du `if` est vraie (≥ 1), sinon les instructions du second bloc,
- le `while(a){}`, qui réalise les instructions du bloc en boucle tant que la condition `a` du `while` est vraie,
- le `for(a;b;c){}`, qui réalise d'abord les instructions `a`, puis réalise les instructions du bloc en boucle tant que la condition `b` est vraie. À chaque tour de boucle, les instructions `c` sont réalisées.

Préliminaires - les fonctions

Les fonctions permettent de factoriser une suite d'instructions en décomposant un algorithme en plusieurs sous-algorithmes.

En C, cela se forme de la façon suivante.

```
type nom_fonction(type param1, type param2, ...)
{
        ...
        return ...;
}
```

On donne le type de retour de la fonction, son nom, et la liste de ses paramètres, qui correspond à un couple type/nom.
Une fonction peut ne pas prendre de paramètres, alors on peut ne rien mettre entre les parenthèses (nombre indéterminé de paramètres), ou mettre `void`. De la même façon, une fonction peut ne rien retourner, et on peut mettre en type de retour `void`.

Préliminaires - les fonctions

Lorsqu'on fait appel à une fonction, nos paramètres vont être recopiés dans les variables locales à la fonction. On appelle cela le passage par recopie.

C'est pour cette raison que ce code ne fonctionne pas, car `x` est recopié dans `a`, et donc `x` n'est jamais incrémenté.

```
void add(int a)
{
        a = a + 1;
}

int main()
{
        int x = 4;
        add(x);
        return 0;
}
```

Préliminaires - les fonctions

Dans ce cadre, la solution est de passer l'adresse de `x`, ce qui permet de bien incrémenter la valeur de `x`.

```
void add(int *a)
{
        *a =  *a + 1;
}

int main()
{
        int x = 4;
        add(&x);
        return 0;
}
```

En réalité, il s'agit toujours d'un passage par recopie, sauf qu'au lieu de recopier la valeur, on recopie l'adresse, stockée dans un pointeur. L'adresse recopiée ne changeant pas, elle pointe toujours sur `x`, ce qui permet de réaliser l'incrémentation.

Plan

C++

Le C++ est apparu en 1979 et a été conçu par Bjarne Stroustrup.
L'objectif principal était d'être une surcouche au C permettant, en plus de permettre la programmation impérative, de pouvoir programmer en orienté objet.

Au fur et à mesure du temps et des normes, de multiples fonctionnalités ont été ajoutées, ainsi qu'une volonté de rendre le code plus simple, lisible, et fiable (par ex. la bibliothèque standard).
Le compilateur C++ (g++, `msvc`) est aussi connu pour être plus strict que le compilateur C (`gcc`, `msvc`).

Comme le C, le C++ continue à être mis à jour à travers des normes tels que C++98, C++03, C++11, C++17, C++20, C++23 et prochainement C++26.

Plan

C++ : différences

Bien que le C++ soit très similaire au C, il y a des éléments qui fonctionnent en C et ne fonctionnent plus en C++.

Nous allons commencer par voir ces éléments, puis ensuite voir les changements d'usage.

C++ : différences

I) Les `include`

Les entêtes et les bibliothèques en C finissent par l'extension `.h`, comme `header`.

```
#include <stdio.h>
#include "projet.h"
```

En C++, on aura pour extension `.hpp`, et les bibliothèques n'ont plus d'extension. Si on souhaite incorporer une bibliothèque C, on rajoutera `c` devant et sans l'extension `.h`.

```
#include <iostream>
#include <cstdio>
#include "projet.hpp"
```

De la même façon, nos fichiers principaux n'auront plus comme extension `.c` mais `.cpp`.

II) `const` **implicite**

En C, nous pouvons écrire

```
const c = 4;
```

car le compilateur affectera par défaut le type `int`.

En C++, nous devons préciser le type explicitement.

```
const int c = 4;
```

III) `const` initialisé

En C, nous pouvons écrire

```
const int x;
```

ce qui déclare une variable en lecture seule n'étant pas initialisée.

En C++, l'initialisation est obligatoire.

```
const int x = 2;
```

IV) Les chaînes de caractères

En C, nous pouvons écrire

```
char *s = "Bonjour";
```

avec `s` l'adresse de la première case du tableau contenant `Bonjour` suivi du caractère terminateur.

Cette chaîne n'étant pas modifiable en soi, nous devons préciser que le pointeur mène vers une chaîne en lecture seule avec le mot-clé `const`.

```
const char *s = "Bonjour";
```

IV) Les chaînes de caractères

De la même façon, en C, nous pouvons écrire

```
char s[7] = "Bonjour";
```

avec `s` l'adresse de la première case du tableau contenant `Bonjour` **sans** le caractère terminateur, car il manque une case.

En C++, il faut forcément l'espace pour ce-dit caractère.

```
char s[8] = "Bonjour";
```

V) Le caractère

En C, `sizeof('a')` retourne `sizeof(int)`, alors qu'en C++ `sizeof(char)`. Sur la plupart des systèmes, le `char` est stocké sur moins de bits que `int`.

VI) Les structures

En C, les structures peuvent servir avoir un groupe de variables qu'on peut manipuler en groupe. Il ne s'agit pas d'un type, et on ne peut pas avoir de méthodes au sein de la structure.

```
struct test {
        int x;
        int y;
};
int main()
{
        struct test a = {1,2};
        int b = a.x + a.y;
        return 0;
}
```

VI) Les structures

En C++, il s'agit d'un type, et on se rapproche du comportement d'une **classe**, qu'on définira plus tard. Ce qu'il faut comprendre, c'est que toutes les contraintes disparaissent : on peut avoir des méthodes, des variables initialisées, des variables statiques, et diverses choses.

```
struct test {
        int x;
        int y;
        static int z = 4;
        void setXY(int a, int b)
        {
                x = a;
                y = b;
        }
        int add()
        {
                return x + y;
        }

};
int main()
{
        test a;
        a.setXY(1,2);
        int b = a.add();
        return 0;
}
```

Plan

C++ : nouveautés

Maintenant que nous avons vu les différences, place aux nouveautés entre le C et le C++.

I) L'initialisation

En C, l'initialisation d'une variable se fait par l'opérateur =, comme `int x = 4;`.

En C++, on peut utiliser l'opérateur =, mais aussi les accolades ou les parenthèses.

```
int x = 4;
int x{4};
int x(4);
```

Nous n'allons pas détailler pour le moment les raisons et les implications. On peut utiliser ces trois méthodes, mais nous allons privilégier les {}.

II) Les expressions constantes

En C et C++ historique, on utilise la macro `#define` pour définir des instructions ou des valeurs constantes gérées à la compilation.

```
#define add10(x) x + 10
#define VAL 14
```

En C++ moderne, on utilise le mot-clé `constexpr`.

```
constexpr int add10(const int x) { return x + 10; }
constexpr int val{14};
```

Un élément `constexpr` est évalué à la compilation pour les variables. C'est la différence avec `const`, qui n'a pas d'obligation à être déterminé à la compilation. Sinon, il n'est évidemment pas modifiable à l'exécution.

Pour les fonctions, c'est aussi à la compilation si tous les arguments sont constants. Si ce n'est pas le cas, cela se comporte comme une fonction classique, à l'exécution.

III) L'opérateur ::

En C++, on va souvent rencontrer l'opérateur ::, qui est là pour accéder à un nom (variable, fonction, etc.) dans un contexte donné.

Premier cas : sans préciser de contexte, on est dans le global.

```
int x;
int main()
{
        int x;
        printf("%d\n", x); // local
        printf("%d\n", ::x); // global
}
```

III) L'opérateur : :

Deuxième cas : les structures.
Comme vu dans la partie différences, les structures en C sont assez différentes des structures en C++, qui pour ses dernières se rapprochent de classes.

En C++, nous pouvons concaténer des structures, et y accéder avec le fameux opérateur.

```
struct test {
        int x;
        struct test1 {
                int x;
        };
};
test t;
int main()
{
        test t;
        test::test1 t1;
        printf("%d\n", ::t.x); // global
        printf("%d\n", t.x); // local
        printf("%d\n", t1.x); // local
}
```

III) L'opérateur ::

Troisième cas : les espaces de noms.

Cela permet de regrouper thématiquement des variables ou des fonctions au sein d'un groupe nommé, et pour éviter des confusions en cas de noms de variables ou de fonctions similaires.

```
namespace test {
        int x;
}
int x;
int main()
{
        int x;
        printf("%d\n", x); // local
        printf("%d\n", ::x); // global
        printf("%d\n", test::x); // espace de nom test
}
```

IV) La surcharge

Contrairement au C, il est possible d'écrire ce code en C++.

```
void test(char c)
{
        printf("fonction␣1\n");
}

void test(int x)
{
        printf("fonction␣2\n");
}
```

Il s'agit de deux fonctions avec le même nom, mais pas les mêmes paramètres. Cela s'appelle la **surcharge**.

Dans cet exemple, si on appelle `test('a')`, cela affichera `fonction 1`, alors que si on appelle `test(4)`, cela affichera `fonction 2`.

Cela est très pratique lorsqu'on souhaite réaliser une même opération mais qu'en fonction des paramètres la marche à suivre est différente.

IV) La surcharge

Par exemple, l'addition.

```
float addition(float f1, float f2)
{
        return f1 + f2;
}

int addition(int x, int y)
{
        return x + y;
}

int main()
{
        float f1{4.0f}, f2{5.0f};
        int x{4}, y{5};
        addition(f1, f2);
        addition(x, y);
        return 0;
}
```

V) La référence

Le C++ ajoute le concept de référence, qui consiste en l'alias de variable.
Il y a maintenant moyen de créer deux variables qui ont la même **adresse**.

```
int x{4};
int & y = x;
```

Dans cet exemple, `x` est strictement équivalent à `y`, et si l'un est modifié l'autre l'est aussi.

V) La référence

Pour quelle utilité ? La principale utilité est d'outrepasser le fait que jusqu'à présent, le seul moyen de communiquer une variable avec une fonction était par recopie.

Si on reprend notre incrémentation,

```
void add(int &a)
{
        a =  a + 1;
}

int main()
{
        int x{4};
        add(x);
        return 0;
}
```

Cette fonction permet bien d'incrémenter, et sans aucune recopie. Ici, a est strictement équivalent à `x`, on ne recopie pas son adresse.

V) La référence

Modifions un peu notre fonction addition.

```
int add(int &a)
{
        return a + 1;
}

int main()
{
        int x = add(4);
        return 0;
}
```

Ce code donne une erreur, car 4 est une valeur littérale (appelée *r-value*), et la référence par simple esperluette ne permet que de prendre la référence d'une variable, soit une *l-value*.
Pour avoir la référence d'un littéral, il faut utiliser la double esperluette.

V) La référence

```
int add(int &&a)
{
        return a + 1;
}

int main()
{
        int x = add(4);
        return 0;
}
```

Ici, le code fonctionne parfaitement, tant qu'on donne une *r-value* en paramètres de add. On a parlé de littéral, donc on pense à des valeurs bruts, mais on parle en réalité de valeurs temporaires, qui ne sont référencées nul part auparavant.

```
int main()
{
        int x = 10;
        int y = add(x+4);
        return 0;
}
```

V) La référence

Bien que l'utilité de la référence avec la simple esperluette (*l-value reference*) semble bien établie, celle avec la double (*r-value reference*) semble plus compliquée à percevoir.

Nous aurons peut-être l'occasion de l'approfondir un jour.

VI) Le pointeur `NULL`

En C, un pointeur ne pointant vers rien pouvait être défini avec la macro `NULL`, qui vaut 0.

```
int* x = NULL;
```

En C++, on utilise le mot-clé `nullptr`.

```
int* x = nullptr;
```

VI) Le transtypage

En C, le transtypage entre un pointeur de `void` vers un pointeur d'un autre type ou inverse est un cast implicite et ne pose aucun souci. On rencontre souvent ce cas lors d'une allocation dynamique, qui retourne un `void *`.

```
int *x = malloc(sizeof(int) * 8);
```

En C++, il faut préciser le transtypage, pour que le compilateur accepte.

```
int *x{(int*)malloc(sizeof(int) * 8)};
```

Mais en réalité, nous n'utilisons plus ce genre de cast en C++, qu'on appelle des casts "type C".

VII) Le transtypage

En C++ nous avons différents opérateurs

VII) Le transtypage

En C++ nous avons différents opérateurs

- `const_cast<type>`, qui permet de transtyper un variable constante d'un type en variable non constante de même type ou inversement,

VII) Le transtypage

En C++ nous avons différents opérateurs

- `const_cast<type>`, qui permet de transtyper un variable constante d'un type en variable non constante de même type ou inversement,
- `static_cast<type>`, qui permet de passer d'un type à un autre à la compilation, tant que c'est considéré comme sécurisé par le compilateur (par ex. `float` $\Leftrightarrow$ `int`, ou `void*` $\Leftrightarrow$ `type*`),

VII) Le transtypage

En C++ nous avons différents opérateurs

- `const_cast<type>`, qui permet de transtyper un variable constante d'un type en variable non constante de même type ou inversement,
- `static_cast<type>`, qui permet de passer d'un type à un autre à la compilation, tant que c'est considéré comme sécurisé par le compilateur (par ex. `float` $\Leftrightarrow$ `int`, ou `void*` $\Leftrightarrow$ `type*`),
- `reinterpret_cast<type>`, qui permet de passer à un type à un autre à la compilation, sans aucune vérification de type. Très utile dans certains contextes, mais aussi le plus dangereux,

VII) Le transtypage

En C++ nous avons différents opérateurs

- `const_cast<type>`, qui permet de transtyper un variable constante d'un type en variable non constante de même type ou inversement,
- `static_cast<type>`, qui permet de passer d'un type à un autre à la compilation, tant que c'est considéré comme sécurisé par le compilateur (par ex. `float` $\Leftrightarrow$ `int`, ou `void*` $\Leftrightarrow$ `type*`),
- `reinterpret_cast<type>`, qui permet de passer à un type à un autre à la compilation, sans aucune vérification de type. Très utile dans certains contextes, mais aussi le plus dangereux,
- `dynamic_cast<type>`, qui permet de passer d'une référence ou d'un pointeur à un autre à l'exécution, et permet donc d'avoir un retour dans le cas où le transtypage a échoué. On approfondira ce type de cast dans la suite du cours.

VII) Le transtypage

```
int main()
{
        int w{4};
        const void* x = static_cast<const void*>(&w);
        void* y = const_cast<void*> (x);
        const void* z = const_cast<const void*> (y);
        float f = static_cast<float>(w);
        float *g = reinterpret_cast<float*>(&w);
        return 0;
}
```

Dans les faits, on pouvait aussi écrire `float f = w;`, puisque ce transtypage est sûr et réalisable avec un cast implicite.

VIII) Les allocations dynamiques

Fini les `malloc`, `calloc` ou `free`, maintenant on utilise `new` et `delete`.

En C, on écrivait

```
int *x = (int *)malloc(sizeof(int));
free(x);
```

pour définir un espace mémoire permettant de stocker un entier.

En C++, on écrit

```
int *x{new int};
delete x;
```

VII) Les allocations dynamiques

On peut aussi allouer une valeur directement à la case pointée par le pointeur en mettant cette valeur en paramètres.

```
int *x{new int(8)};
delete x;
```

Ici, `*x` vaut 8.

VIII) Les allocations dynamiques

On peut aussi réaliser la même chose avec l'allocation d'un tableau.
En C, on écrivait

```
int *x = (int *)malloc(sizeof(int) * 8);
free(x);
```

.

En C++, on écrit

```
int *x{new int[8]};
delete[] x;
```

On rajoute des crochets au delete pour souligner qu'il ne s'agit pas d'un unique élément mémoire alloué.

VIII) La bibliothèque standard

Le C++ offre la bibliothèque standard, qui permet d'utiliser un grand nombre d'objets avec de nombreuses fonctionnalités pour nous simplifier la tâche.

Nous n'allons pas rentrer dans le détail du fonctionnement pour le moment, ce sera abordé plus tard dans le cours.

IX) La bibliothèque standard : le tableau

En C et C++ historique, on déclare un tableau de taille statique de cette façon

```
int tab[20];
```

En C++ moderne, on peut utiliser la classe array qui permet d'instancier `array`.

```
#include <array>
std::array<int, 20> tab;
```

On peut accéder à `tab` et ses valeurs de la même manière qu'un tableau classique, avec l'opérateur crochet.

On remarque que la bibliothèque standard à son espace de nom `std`, demandant l'utilisation de l'opérateur `::`.

IX) La bibliothèque standard : le tableau

Pour l'allocation dynamique d'un tableau, en C on utilisait un `malloc` et un `free`.

```
int *tab = (int *)malloc(sizeof(int) * 20);
free(tab);
```

En C++ historique, on utilisait un `new` et un `delete`.

```
int *tab{new int[20]};
delete[] tab;
```

En C++ moderne, on utilise la classe vector qui permet d'instancier `vector`.

```
#include <vector>
std::vector<int> tab;
```

IX) La bibliothèque standard : le tableau

Lorsqu'on déclare un `vector`, on déclare un espace mémoire avec taille évolutive.
On peut réserver un espace précis (pour éviter des allocations intempestives) avec `.reserve()`. On peut aussi `.resize()`, qui lui va non seulement réserver mais initialiser cet espace à 0.

Pour accéder à une valeur, on peut utiliser les opérateurs crochets **uniquement** si une valeur a déjà été attribuée à cette case du tableau. On utilisera `.at()` par sécurité.
Pour ajouter une nouvelle valeur, on utilisera `.push_back()`.

Contrairement à l'allocation dynamique, inutile de libérer l'espace, la durée de vie du `vector` est cantonnée à son bloc, comme une variable locale classique.

IX) La bibliothèque standard : la chaîne de caractères

Pour l'allocation d'une chaîne de caractères modifiables, en C et C++ historique on utilisait un tableau d'une certaine taille pour stocker les lettres et le caractères terminateurs.

```
char str[8] = "Bonjour";
```

En C++ moderne, on utilise la classe `string` qui permet d'instancier `string`.

```
#include <string>
std::string str{"Bonjour"};
```

Pour obtenir la chaîne sous forme d'un tableau de `char` avec un caractère terminateur, on utilise la méthode `.c_str()`.

IX) La bibliothèque standard : la chaîne de caractères

Et pour la chaîne de caractères en lecture seule ? En C et C++ historique, on utilisait cette syntaxe.

```
const char *str = "Bonjour";
```

En C++ moderne (C++17), on utilise la classe `string_view` qui permet d'instancier `string_view`.

```
#include <string_view>
constexpr std::string_view str{"Bonjour"};
```

Pour obtenir la chaîne sous forme d'un tableau de `char` avec un caractère terminateur, on utilise la fonction `.data()`. Contrairement à `.c_str()` de `string`, le caractère terminateur n'est pas garanti par construction.

IX) La bibliothèque standard : les entrées-sorties

En C et C++ historique, pour lire une entrée sur le terminal, on utilisait `scanf` et pour écrire, on utilisait `printf`.

```
int x;
scanf("%d", &x);
printf("%d\n", x);
```

En C++ moderne, on utilise la classe `iostream` qui permet, à l'aide d'opérateurs, d'intéragir avec les entrées-sorties.

```
#include <iostream>
int x;
std::cin >> x;
std::cout << x << std::endl;
```

IX) La bibliothèque standard : les entrées-sorties

On remarquera qu'il n'y a plus de considérations de types, `cin` et `cout` acceptant *quasiment* n'importe quel type, dont des chaînes de caractères par exemple.

```
#include <iostream>
#include <string_view>

constexpr std::string_view str{"Bonjour"};
std::cout << str << std::endl;
```

Par ailleurs, le `endl` permet de revenir à la ligne et de **vider la mémoire tampon**. Si on ne souhaite que revenir à la ligne, on peut utiliser `\n`.

```
#include <iostream>
#include <string_view>

constexpr std::string_view str{"Bonjour"};
std::cout << str << '\n';
```

IX) La bibliothèque standard

Il existe plein d'autres éléments de la bibliothèque standard qui sont utiles mais ne vont pas être évoqués pour le moment.

L'un d'entre eux est le pointeur intelligent, qui permet de mieux gérer les pointeurs et leur durée de vie.
Par choix, nous n'allons pas approfondir pour le moment, en essayant au maximum d'éviter la manipulation de pointeurs qui pointent vers une mémoire allouée dynamiquement, et ce grâce aux objets vu précédemment.

Plan

Paradigme objet & modélisation

Le concept du paradigme objet est de pouvoir concevoir une modélisation d'un système d'informations sous la forme d'objets, qui est une représentation simplifiée d'entités concrètes du monde réel.

Un objet est défini par trois éléments : son état, qui correspond à l'ensemble des valeurs de tous ses attributs à un instant donné, son comportement, qui décrit les opérations réalisables par l'objet, et son identité, qui le caractèrise et est unique.

Paradigme objet & modélisation

Dans le monde réel, on regroupe et catégorise les éléments qui ont des points communs, on classe. Les classes sont des descriptions abstraites d'un ensemble d'objets. Ces objets peuvent aussi être appelés instance de classe.

Paradigme objet & modélisation

Pour mieux comprendre tout ça, nous allons nous concentrer sur les modèles et le langage de modélisation UML (*Unified Modeling Language*), permettant d'expliciter tout ça.

Un modèle est une description d'une partie du monde réel, permettant de classifier cette observation de la réalité. Pour le décrire, on répond aux questions suivantes :

Paradigme objet & modélisation

Pour mieux comprendre tout ça, nous allons nous concentrer sur les modèles et le langage de modélisation UML (*Unified Modeling Language*), permettant d'expliciter tout ça.

Un modèle est une description d'une partie du monde réel, permettant de classifier cette observation de la réalité. Pour le décrire, on répond aux questions suivantes :

- Quoi ? $\rightarrow$ les objets ou concepts

Paradigme objet & modélisation

Pour mieux comprendre tout ça, nous allons nous concentrer sur les modèles et le langage de modélisation UML (*Unified Modeling Language*), permettant d'expliciter tout ça.

Un modèle est une description d'une partie du monde réel, permettant de classifier cette observation de la réalité. Pour le décrire, on répond aux questions suivantes :

- Quoi ? $\rightarrow$ les objets ou concepts
- Qui ? $\rightarrow$ les personnes physiques ou morales

Paradigme objet & modélisation

Pour mieux comprendre tout ça, nous allons nous concentrer sur les modèles et le langage de modélisation UML (*Unified Modeling Language*), permettant d'expliciter tout ça.

Un modèle est une description d'une partie du monde réel, permettant de classifier cette observation de la réalité. Pour le décrire, on répond aux questions suivantes :

- Quoi ? $\rightarrow$ les objets ou concepts
- Qui ? $\rightarrow$ les personnes physiques ou morales
- Quand ? $\rightarrow$ éléments temporels

Paradigme objet & modélisation

Pour mieux comprendre tout ça, nous allons nous concentrer sur les modèles et le langage de modélisation UML (*Unified Modeling Language*), permettant d'expliciter tout ça.

Un modèle est une description d'une partie du monde réel, permettant de classifier cette observation de la réalité. Pour le décrire, on répond aux questions suivantes :

- Quoi ? $\rightarrow$ les objets ou concepts
- Qui ? $\rightarrow$ les personnes physiques ou morales
- Quand ? $\rightarrow$ éléments temporels
- Où ? $\rightarrow$ localisation

Paradigme objet & modélisation

Pour mieux comprendre tout ça, nous allons nous concentrer sur les modèles et le langage de modélisation UML (*Unified Modeling Language*), permettant d'expliciter tout ça.

Un modèle est une description d'une partie du monde réel, permettant de classifier cette observation de la réalité. Pour le décrire, on répond aux questions suivantes :

- Quoi ? $\rightarrow$ les objets ou concepts
- Qui ? $\rightarrow$ les personnes physiques ou morales
- Quand ? $\rightarrow$ éléments temporels
- Où ? $\rightarrow$ localisation
- Comment ? $\rightarrow$ actions réalisées par le Qui sur le Quoi

Paradigme objet & modélisation

Le langage de modélisation permet de décrire un système de différentes façons possibles (codes, images, diagrammes, etc.) et dont l'objectif est de représenter des concepts efficacement et sans ambiguitë, permettant de faciliter l'analyse.

Dans le cadre de l'UML, nous avons de nombreux diagrammes existants.

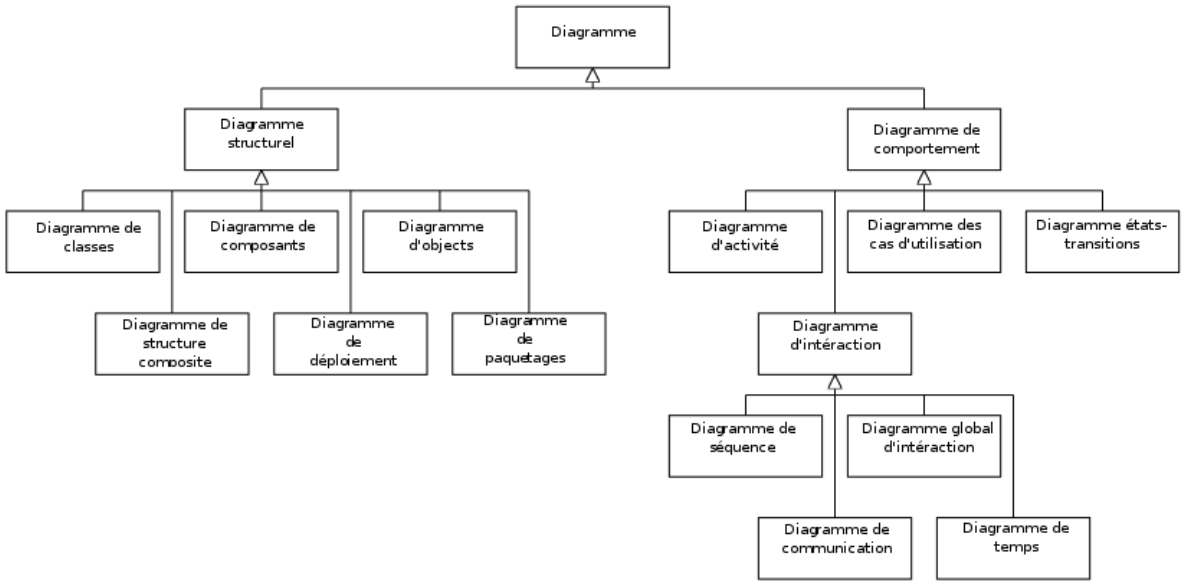


Figure: Liste des diagrammes UML, Wikipédia EN

Paradigme objet & modélisation

Nous n'allons pas étudier l'intégralité de ces diagrammes, mais seulement ces trois suivants

- le diagramme de classes, qui permet de modéliser un système avec des classes et des interactions entre elles,
- le diagramme de cas d'utilisation, qui permet de modéliser les interactions que peuvent avoir des utilisateurs avec un système,
- le diagramme de séquences, qui permet de modéliser la ligne de vie des différents acteurs et de capturer l'ordre des interactions entre ces derniers.

Ces modélisations permettent d'avoir une bonne idée du fonctionnement général du système et donc le concevoir correctement.

I) Diagramme de classes

Le diagramme de classes permet de décrire une structure à l'aide de classes ayant des relations entre elles.

Pour décrire une classe, nous avons des rectangles composés de trois parties : le nom de la classe, les attributs (variables), et les opérations (fonctions ou méthodes).

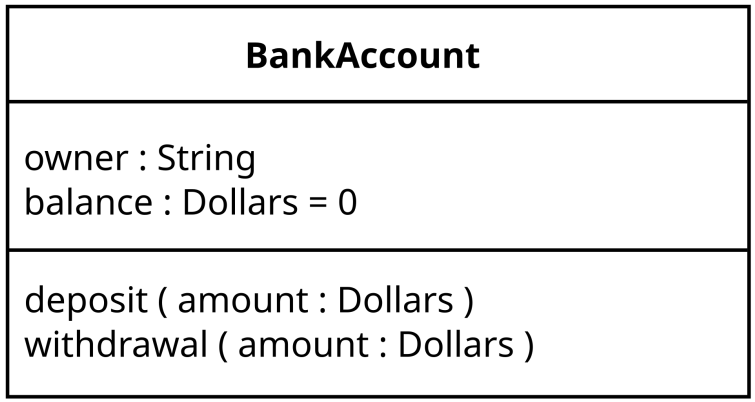


Figure: Modélisation d'un compte bancaire en classe, Wikipédia EN

I) Diagramme de classes

Il n'y a en pas mention sur cet exemple, mais on peut définir la visibilité des attributs et des opérations en ajoutant en prefixe un de ces symboles

- + pour l'accès public (accessible en dehors de cette classe)
- - pour l'accès privé (inaccessible en dehors de cette classe)
- # pour l'accès protégé (inaccessible en dehors de cette classe et ses sous-classes)

On peut aussi avoir le nom de la classe en italique ou le mot-clé *abstract* qui est indiqué, ce qui signifie que la classe est abstraite et qu'elle peut avoir des sous-classes mais ne pourra pas être instancié. Cela implique que cette classe a des opérations abstraites elles-aussi, qui seront spécifiées.

Mais, qu'est-ce qu'une sous-classe ?

I) Diagramme de classes

Une sous-classe est une classe qui descend d'une classe, qu'on appelle classe héritée ou super-classe.

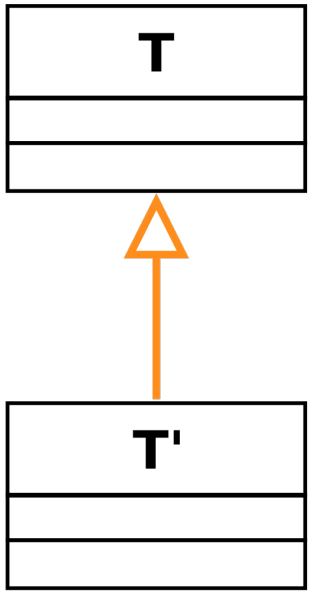


Figure: Héritage, Wikipédia

Ici, la classe T' est héritée de la classe T, et a donc accès aux attributs et opérations de la classe T qui ont une visibilité publique ou protégée.

I) Diagramme de classes

Mais l'héritage a quelle utilité pratique ?

On peut introduire cet exemple

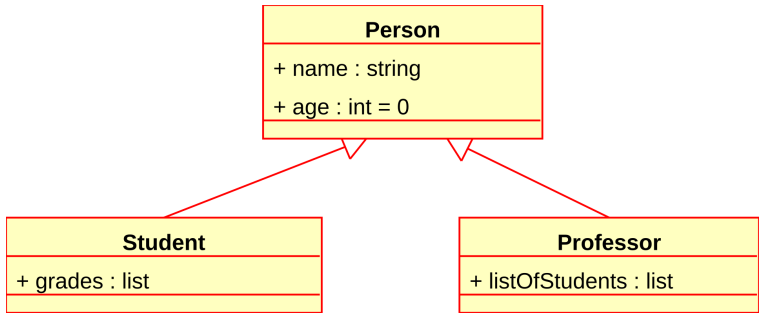


Figure: Héritage, Wikipédia EN

où on constate qu'un professeur ou un étudiant est bien une personne, et donc cela a du sens que ces classes héritent de cette façon.

I) Diagramme de classes

Mais on peut avoir des relations entre classes sans pour autant que l'une descende de l'autre. Le premier cas est l'association.

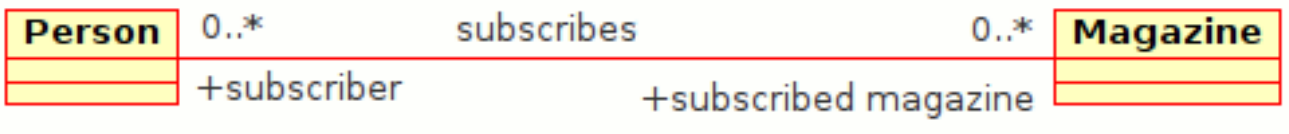


Figure: Association, Wikipédia EN

Elle est décrite par une ligne droite, et on constate des intervalles à chaque proximité de classe. Ici, cela signifie qu'une personne peut avoir entre 0 et une infinité d'abonnements à des magazines, et qu'un magazine peut avoir entre 0 et une infinité d'abonnés.

On peut flêcher cette association si elle est dans un sens unique. On peut aussi avoir une auto-association à une même classe.

I) Diagramme de classes

Quand on peut décrire une association avec un objet qui fait partie d'un autre objet, ou alors un objet est composé par un autre objet, on peut parler d'**agrégation**, qui est un cas particulier de l'association. On le décrit avec un losange vide de la classe qui possède.

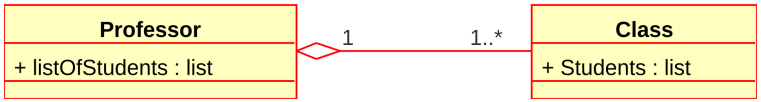


Figure: Agrégation, Wikipédia EN

Ici, cela signifie qu'un professeur possède entre 1 et une infinité de classes, alors qu'une classe ne possède qu'un seul professeur.

I) Diagramme de classes

Si lors de votre agrégation, votre objet ne peut pas vivre sans un autre, alors il s'agit d'une agrégation forte appelée **composition**. On le représente avec un losange noir du côté de la classe qui possède.

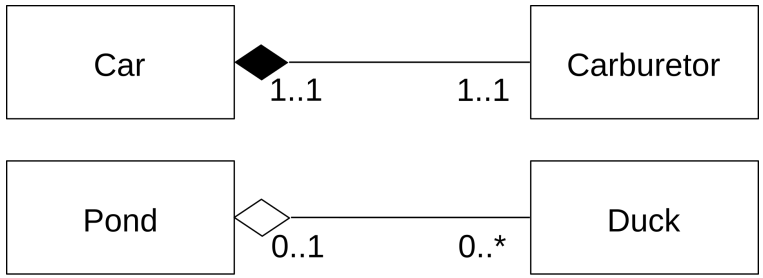


Figure: Agrégation et composition, Wikipédia EN

Ici, cela signifie qu'une voiture possède un unique carburateur et qu'un carburateur n'est possédée que par une unique voiture, et si cette voiture n'existe plus le carburateur n'existe plus non plus.
L'autre exemple est une agrégation, où un canard a soit une mare soit n'en a pas, et une mare compte entre 0 et une infinité de canard.

I) Diagramme de classes

Voici un récapitulatif de tous les liens possibles entre classes dans un diagramme de classes UML.

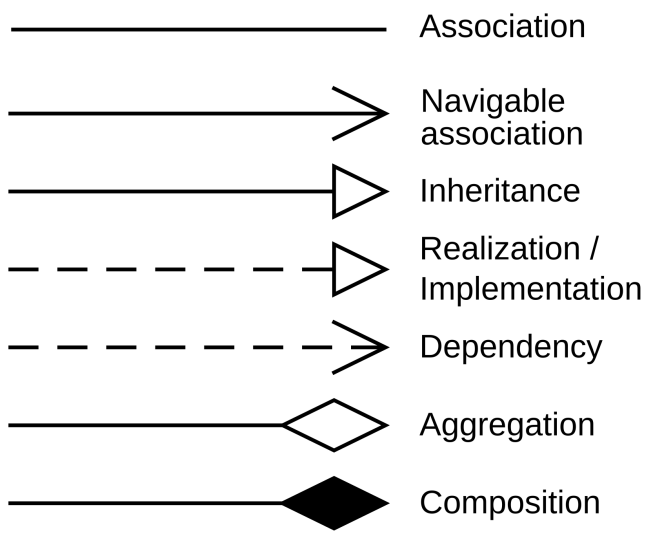


Figure: Liste des relations entre classes, Wikipédia EN

II) Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation permet de décrire les différentes façons dont les utilisateurs peuvent interagir avec un système.

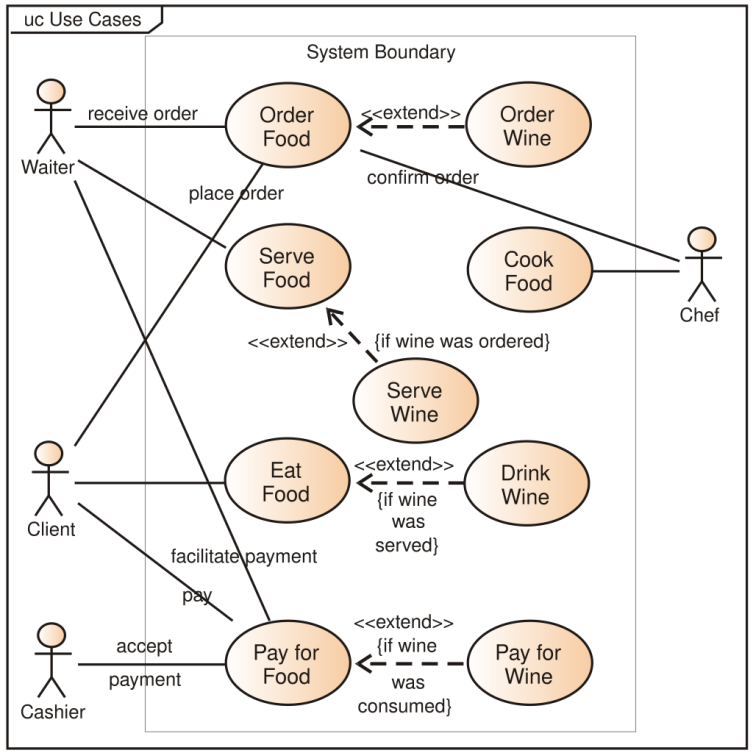


Figure: Diagramme de cas d'utilisations, Wikipédia EN

II) Diagramme de cas d'utilisation

II) Diagramme de cas d'utilisation

- Le cadre représente le système, et il a un nom.

II) Diagramme de cas d'utilisation

- Le cadre représente le système, et il a un nom.
- Les bonhommes correspondent aux différents utilisateurs, appelés acteurs.

II) Diagramme de cas d'utilisation

- Le cadre représente le système, et il a un nom.
- Les bonhommes correspondent aux différents utilisateurs, appelés acteurs.
- Les bulles au sein du système correspondent à des cas d'utilisation.

II) Diagramme de cas d'utilisation

- Le cadre représente le système, et il a un nom.
- Les bonhommes correspondent aux différents utilisateurs, appelés acteurs.
- Les bulles au sein du système correspondent à des cas d'utilisation.
- Les lignes entre les acteurs et les cas d'utilisation sont des associations (nous pouvons avoir des multiplicités, comme dans le diagramme de classe).

II) Diagramme de cas d'utilisation

- Le cadre représente le système, et il a un nom.
- Les bonhommes correspondent aux différents utilisateurs, appelés acteurs.
- Les bulles au sein du système correspondent à des cas d'utilisation.
- Les lignes entre les acteurs et les cas d'utilisation sont des associations (nous pouvons avoir des multiplicités, comme dans le diagramme de classe).
- Les flêches pointillées avec indiquée "include" sont des relations d'inclusion, ce qui signifie que pour réaliser une action on doit forcément passer par cette action incluse.

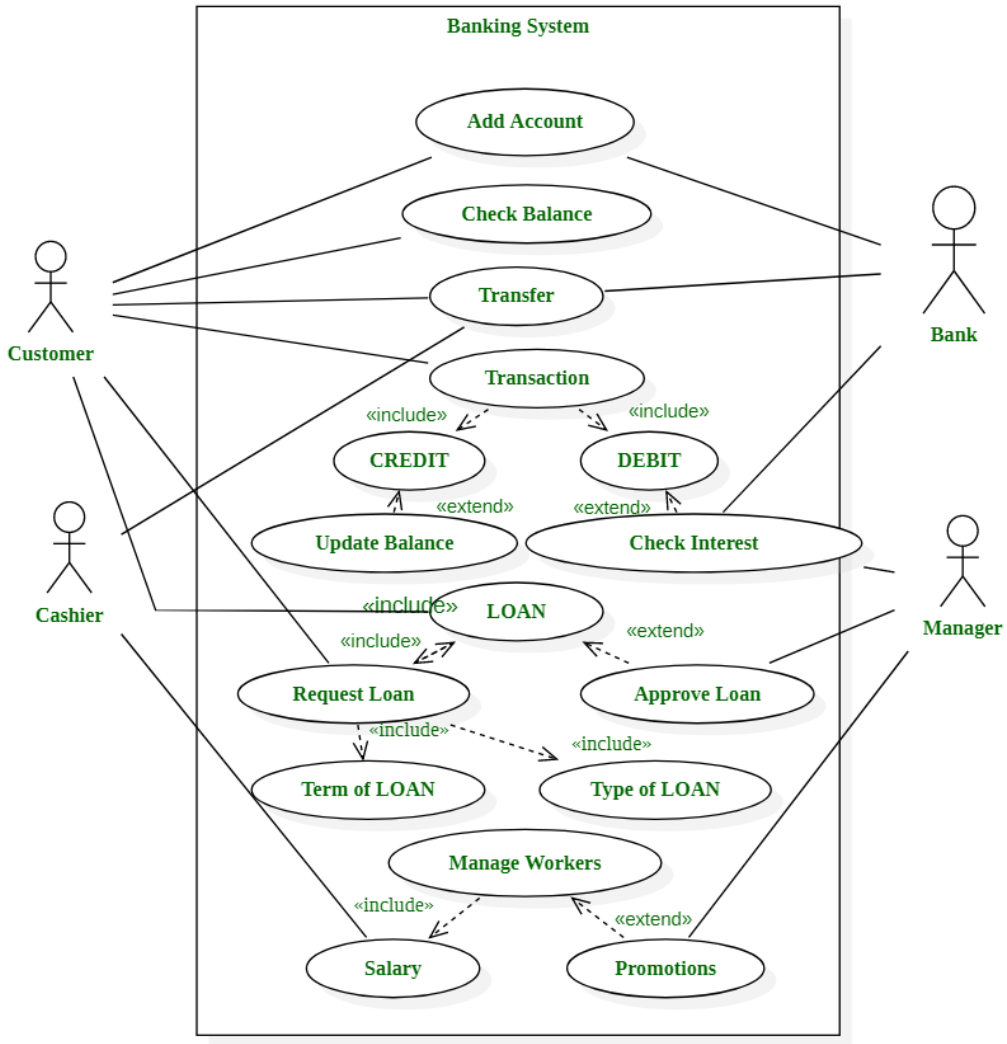
II) Diagramme de cas d'utilisation

- Le cadre représente le système, et il a un nom.
- Les bonhommes correspondent aux différents utilisateurs, appelés acteurs.
- Les bulles au sein du système correspondent à des cas d'utilisation.
- Les lignes entre les acteurs et les cas d'utilisation sont des associations (nous pouvons avoir des multiplicités, comme dans le diagramme de classe).
- Les flêches pointillées avec indiquée "include" sont des relations d'inclusion, ce qui signifie que pour réaliser une action on doit forcément passer par cette action incluse.
- Les flêches pointillées avec indiquée "extend" sont des relations d'extension, ce qui veut dire que pour réaliser une action on peut éventuellement réaliser cette action étendue.

II) Diagramme de cas d'utilisation

- Le cadre représente le système, et il a un nom.
- Les bonhommes correspondent aux différents utilisateurs, appelés acteurs.
- Les bulles au sein du système correspondent à des cas d'utilisation.
- Les lignes entre les acteurs et les cas d'utilisation sont des associations (nous pouvons avoir des multiplicités, comme dans le diagramme de classe).
- Les flêches pointillées avec indiquée "include" sont des relations d'inclusion, ce qui signifie que pour réaliser une action on doit forcément passer par cette action incluse.
- Les flêches pointillées avec indiquée "extend" sont des relations d'extension, ce qui veut dire que pour réaliser une action on peut éventuellement réaliser cette action étendue.
- On peut aussi avoir des relations entre les acteurs.

II) Diagramme de cas d'utilisation



III) Diagramme de séquences

Le diagramme de séquences permet de représenter les interactions entre les participants d'un système en fonction du temps. Il s'agit donc d'un diagramme dynamique.

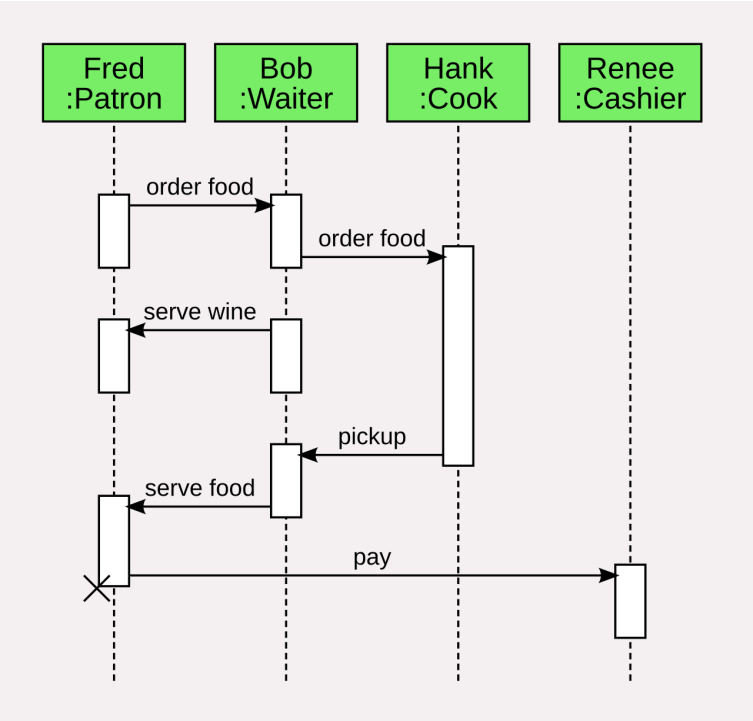


Figure: Diagramme de séquences, Wikipédia EN

III) Diagramme de séquences

III) Diagramme de séquences

- Les rectangles représentent les participants avec leur rôle.

III) Diagramme de séquences

- Les rectangles représentent les participants avec leur rôle.
- Les lignes en pointillées correspondent à la ligne de vie de ces acteurs.

III) Diagramme de séquences

- Les rectangles représentent les participants avec leur rôle.
- Les lignes en pointillées correspondent à la ligne de vie de ces acteurs.
- Les blocs rectangulaires blancs représentent la barre d'activité.

III) Diagramme de séquences

- Les rectangles représentent les participants avec leur rôle.
- Les lignes en pointillées correspondent à la ligne de vie de ces acteurs.
- Les blocs rectangulaires blancs représentent la barre d'activité.
- Des fragments (ressemblant à des blocs) peuvent être utilisés de façon à spécifier les opérations plus précisement (boucle, condition, parallélisme).

Paradigme objet & modélisation

Nous avons donc vu ici trois façons de modéliser des choses différentes, de façon à trouver la représentation adaptée à notre cas d'usage.

Il s'agit que d'un premier aperçu, il existe plein d'autres choses et l'objectif ici était de faire un premier lien avec ce monde.

Plan

Les classes en C++

En C++, programmer en orienté objet revient à manipuler des classes ; similaires à celles du diagramme de classes ; qui sont d'une certaine manière une évolution des structures.

On définit une classe par le mot-clé `class` et un nom, et cela va définir un nouveau type concret. Ce type peut contenir des variables ou des méthodes. On va préalablement déclarer cette classe au sein d'un fichier entête, avec la signature de tout son contenu.

On va donc créer un fichier `MaClasse.hpp` contenant ceci :

```
class MaClasse {
public:
        int x;
        int getX();
        void setX(int);
};
```

Les classes en C++

Une fois ce fichier crée, on peut construire le fichier `MaClasse.cpp`, qui va définir tous les éléments nécessaires déclarés dans l'entête.

```
#include "MaClasse.hpp"
int MaClasse::getX()
{
        return x;
}

void MaClasse::setX(int)
{
        x = a;
}
```

On remarque qu'on utilise le nom de classe comme espace de nom, pour bien spécifier quelles méthodes on définit.

Les classes en C++

Vu que notre classe est déclarée et que nos méthodes sont maintenant définies, on peut y faire appel. Par exemple, créons un fichier `main.cpp` contenant ceci :

```
#include <iostream>
#include "MaClasse.hpp"

int main()
{
        MaClasse mc;
        std::cout << mc.getX() << std::endl;
        mc.setX(10);
        std::cout << mc.getX() << std::endl;
        return 0;
}
```

Remarque Ici, `MaClasse` est une classe, `mc` est une instance de cette classe, soit un objet. Cet objet possède ses propres variables et méthodes, définit par la classe elle-même.

Cependant, on a diverses problématiques qui se posent :

Les classes en C++

Cependant, on a diverses problématiques qui se posent :

- Notre entête est ici incluse deux fois : dans `main.cpp`, puis dans `MaClasse.cpp`, ce qui peut causer une erreur à la compilation.

Cependant, on a diverses problématiques qui se posent :

- Notre entête est ici incluse deux fois : dans `main.cpp`, puis dans `MaClasse.cpp`, ce qui peut causer une erreur à la compilation.
- Toutes nos variables et toutes nos méthodes sont accessibles par tout le monde, et ce avec ce mot-clé `public`. Si on reprend notre `MaClasse mc;`, on peut directement modifier `x` en faisant `mc.x = 10`, sans utiliser la méthode `setX`.

Cependant, on a diverses problématiques qui se posent :

- Notre entête est ici incluse deux fois : dans `main.cpp`, puis dans `MaClasse.cpp`, ce qui peut causer une erreur à la compilation.
- Toutes nos variables et toutes nos méthodes sont accessibles par tout le monde, et ce avec ce mot-clé `public`. Si on reprend notre `MaClasse mc;`, on peut directement modifier `x` en faisant `mc.x = 10`, sans utiliser la méthode `setX`.
- Lorsque notre objet est crée, il n'y a aucune initialisation qui est réalisée. `x` faisant partie de la pile, on a aucun moyen de savoir ce qu'il vaut.

Les classes en C++

Pour notre premier problème, il suffit de rajouter une vérification, et de n'inclure notre entête uniquement si elle n'a jamais été inclue. Pour cela, on utilise les instructions pré-processeurs, qui nous permettent de définir une macro lors de la première inclusion, et si jamais celle-ci est définie on ne réinclut pas l'entête.

```
#ifndef MACLASSE_H
#define MACLASSE_H
class MaClasse {
public:
        int x;
        int getX();
        void setX(int);
};
#endif
```

Les classes en C++

Pour notre deuxième problème, on va voir succintement les protections d'accès.

Il existe différents mots-clés permettant de protéger l'accès aux variables et méthodes

- `public`, où les méthodes et variables sont accessibles par tout le monde,
- `private`, où les méthodes et variables ne sont accessibles qu'au sein de la même classe,
- `protected`, où les méthodes et variables ne sont accessibles qu'au sein de la même classe, ou des classes enfants.

Si on fait un lien avec le diagramme de classes en UML, il s'agit de l'équivalent des symboles +, - et #.

Les classes en C++

On va appliquer ces mots-clés en suivant la syntaxe `motclé:`, et la section sera indentée.

```
class MaClasse {
private:
        int x;
public:
        int getX();
        void setX(int);
};
```

En définissant ce code, `mc.x = 10` ne peut plus être réalisé sans erreur, car `x` est privé.

On constate la présence des fonctions `getX` et `setX`, qui sont publics et permettent d'accéder et modifier `x`, qui lui est privé.
Il s'agit d'une convention d'usage de construire ses classes ainsi, à l'aide d'accesseurs (aussi appelé getter/setter).

Bien qu'il puisse ne s'agir que d'une convention et d'une simple encapsulation d'un retour ou d'une affectation, cela peut permettre de réaliser des opérations en tant qu'observateur d'une accession ou d'une modification d'une variable.

Le fait d'utiliser des méthodes permet aussi d'utiliser un autre contrôle d'accès possible : l'amitié.

Les classes en C++

L'amitié consiste à dire qu'une classe ou une méthode peut accéder à des attributs privés même si on n'est pas au sein de la même classe s'il y a une amitié.
Pour cela, dans un premier temps, on doit définir dans notre classe une méthode ou une classe qui sera amie avec le mot-clé `friend`.

```
#include "MaClasse2.hpp"
class MaClasse {
private:
        int x;
public:
        friend class MaClasse2;
};
```

Prenons l'entête de `MaClasse2`.

```
#include "MaClasse.hpp"
class MaClasse2 {
public:
        int getX(MaClasse&);
};
```

Si nous essayons de compiler avec ces inclusions, cela va poser souci : nous avons une interdépendance.

En effet, `MaClasse` a besoin d'inclure `MaClasse2.hpp` pour avoir la définition de `MaClasse2` et déclarer l'amitié, et `MaClasse2` a besoin d'inclure `MaClasse.hpp` pour avoir la définition de `MaClasse` et faire la signature de la fonction.

On va donc réaliser ce qu'on appelle une déclaration anticipée, et ne pas inclure l'entête de `MaClasse`.

```
class MaClasse;
class MaClasse2 {
public:
        int getX(MaClasse&);
};
```

Les classes en C++

À partir de ce moment, on peut imaginer une opération au sein de la classe `MaClasse2` qui accède ou modifie `x`, sans souci. Inutile d'inclure `MaClasse2.hpp`, `MaClasse.hpp` l'inclut.

```
#include <iostream>
#include "MaClasse.hpp"

int MaClasse2::getX(MaClasse& mc)
{
        return mc.x;
}

int main()
{
        MaClasse mc;
        MaClasse2 mc2;
        std::cout << mc2.getX(mc) << std::endl;
}
```

Les classes en C++

On peut aussi rendre ami une méthode précise, plutôt que toute une classe.

```
class MaClasse {
private:
        int x;
public:
        friend int getX(MaClasse&);
};

int getX(MaClasse& mc)
{
        return mc.x;
}

int main()
{
        MaClasse mc;
        std::cout << getX(mc) << std::endl;
}
```

Ici il s'agit d'une fonction générale, mais on peut spécifier la fonction d'une autre classe.

Pour notre dernier problème d'initialisation, on va rentrer dans le monde des constructeurs et destructeurs.

Pour notre dernier problème d'initialisation, on va rentrer dans le monde des constructeurs et destructeurs.

- Un constructeur va être appelé lorsqu'un objet est déclaré. Par défaut, il y a un appel implicite au constructeur, et rien ne sera initialisé.

Les classes en C++

Pour notre dernier problème d'initialisation, on va rentrer dans le monde des constructeurs et destructeurs.

- Un constructeur va être appelé lorsqu'un objet est déclaré. Par défaut, il y a un appel implicite au constructeur, et rien ne sera initialisé.
- Un destructeur va être appelé lorsqu'un objet arrive en fin de vie (on sort de son bloc), ou volontairement.

Les classes en C++

Un constructeur est une méthode sans type de retour dont le nom est le nom de classe. Si on reprend notre entête précédente.

```
class MaClasse {
private:
        int x;
public:
        MaClasse();
        int getX();
        void setX(int);
};
```

Les classes en C++

On peut définir la méthode dans le fichier `.cpp`.

```
#include "MaClasse.hpp"
MaClasse::MaClasse()
{
        x = 0;
}
```

Ici, à la création de notre objet, `MaClasse()` sera appelé, et donc `x` sera défini à 0;

On remarque qu'ici, l'attribution de 0 à `x` a lieu en deux temps : on déclare, puis on affecte. Il y a moyen d'affecter directement en utilisant une liste d'initialisation, avec la syntaxe suivante

```
#include "MaClasse.hpp"
MaClasse::MaClasse() : x(0){}
```

Vu qu'ici on initialise directement, cela peut permettre aussi d'initialiser une variable `const`.

Les classes en C++

Comme toutes fonctions en C++, le constructeur accepte les arguments.

```
class MaClasse {
private:
        int x;
public:
        MaClasse(int);
        int getX();
        void setX(int);
};
```

et

```
#include "MaClasse.hpp"
MaClasse::MaClasse(int a) : x(a){}
```

Dans ce cas, la déclaration de l'objet contiendra un argument `MaClasse mc{4};`. Le constructeur vide existe toujours, donc `MaClasse mc;` ne lèvera pas d'erreurs, mais on ne saura pas ce que contient `x`.

Les classes en C++

Et comme toutes fonctions en C++, on peut surcharger le constructeur.

```
class MaClasse {
private:
        int x;
public:
        MaClasse();
        MaClasse(int);
        int getX();
        void setX(int);
};
```

et

```
#include "MaClasse.hpp"
MaClasse::MaClasse() : x(0){}
MaClasse::MaClasse(int a) : x(a){}
```

Et dans ce cas, avec `MaClasse mc{4};`, `x` vaut 4, et avec `MaClasse mc;`, `x` vaut 0.

On a vu que par défaut, nous avions un constructeur vide implicite. Il existe aussi un constructeur par recopie. Par exemple,

```
#include "MaClasse.hpp"
int main()
{
        MaClasse mc{4};
        MaClasse mc2{mc};
        return 0;
}
```

Ici, le contenu des variables de `mc` va être copié dans `mc2`, et donc `x` vaudra aussi 4.

Si on souhaite redéfinir ce constructeur par recopie, on ne peut pas définir le constructeur suivant

```
class MaClasse {
public:
        MaClasse(MaClasse);
};
```

Car lorsque le constructeur par recopie sera appelé, une recopie du paramètre sera faite, et donc le constructeur par recopie sera appelé, et donc une recopie du paramètre sera faite, et donc...

Pour éviter ce souci, on peut donc utiliser le passage par référence.

```
class MaClasse {
public:
        MaClasse(const MaClasse&);
};
```

De manière générale, on aura aussi tendance à ajouter le mot-clé `const` devant les passages par référence où on ne souhaite pas modifier par erreur son contenu.

Les classes en C++

Reprenons notre code et imaginons que nous modifions le nom de la variable pour notre méthode `setX`.

```
#include "MaClasse.hpp"
void MaClasse::setX(int x)
{
        x = x;
}
```

Ici, on va affecter `x` avec `x`. Comment éviter l'ambiguitë et savoir de quel variable on parle ? Avec le mot-clé `this`.

Les classes en C++

```
#include "MaClasse.hpp"
void MaClasse::setX(int x)
{
        this->x = x;
}
```

Le mot-clé `this`, qui est un pointeur vers l'objet en cours, permet entre autres de pointer vers les variables et méthodes au sein de ce dit objet. `this->x` correspond ici à la variable `x` de l'objet, l'opérateur `->` permettant d'accéder aux attributs de l'élément pointé. C'est l'équivalent de `(*this).x`.

Les classes en C++

À la fin de vie de l'objet, le destructeur est appelé. On peut le redéfinir pour spécifier des choses à réaliser à la destruction de l'instance.

Sa définition est toujours la même, dans l'entête.

```
class MaClasse {
public:
        virtual ~MaClasse();
};
```

Le mot-clé `virtual` est indispensable, mais sera explicité plus tard. Le destructeur est définit par la présence du $\sim$ devant le nom de la classe, sans paramètres ni retour.

Les classes en C++

Pour quelle utilité ? On peut, par exemple, vouloir réaliser une action à la destruction, comme la désallocation de la mémoire.

```
class MaClasse {
private:
        int* tab;
public:
        MaClasse();
        virtual ~MaClasse();
};
```

Les classes en C++

```
#include "MaClasse.hpp"
MaClasse::MaClasse()
{
        this->tab = new int[14];
}

virtual MaClasse::~MaClasse()
{
        delete[] this->tab;
}
```

Les classes en C++

Jusqu'à présent, on a étudié tout ce qu'on pouvait faire au sein des objets en eux-mêmes. Mais il est possible qu'on souhaite avoir des variables ou des méthodes qui ne sont pas propres à chaque objet mais appartiennent à une classe en elle-même, et donc en commun avec tous les objets de la même nature. On peut le voir comme une variable globale au sein d'une classe.

On appelle cela un attribut statique, et on le déclare avec le mot-clé `static`.

Comme précédemment, l'accessibilité de ces variables et méthodes dépendent du contrôle d'accès choisis : `public`, `protected` ou `private`.

```
class MaClasse {
private:
        static int x;
public:
        static int public();
};
```

Ici, `MaClasse::x` ne sera accessible qu'au sein de la même classe, alors que `MaClasse::public()` sera accessible partout.

Les classes en C++

Pour définir la méthode statique, c'est identique à une méthode classique.

```
#include "MaClasse.hpp"

int MaClasse::public()
{
        ...
}
```

Cette méthode ne nécessite pas la création d'une instance de la classe pour être appelée, et sera appelée de la même façon qu'une fonction dans un espace de noms `int x = MaClasse::public()`.
Comme elle n'est pas propre à chaque objet, elle ne peut pas faire appel à une instance. `this` n'est donc pas utilisable.

Pour initialiser `x`, on ne va pas le faire dans le constructeur, vu que `x` est global, et donc chaque nouvel objet réinitialiserait sa valeur. On va donc le faire en dehors de toute fonction.

```
#include "MaClasse.hpp"
int MaClasse::x = 10;
```

Les classes en C++

Aparté : la classe utilitaire.

Jusqu'à présent, on a utilisé les classes pour répondre à une modélisation objet et donc une utilisation à base d'instance de classe. En réalité, les classes peuvent aussi servir comme classe utilitaire : une sorte de boite à outils de méthodes statiques.

Par exemple, une classe Calcul.

```
class Calcul {
public:
        static int add(int, int);
        static int sub(int, int);
        static int mul(int, int);
        static int div(int, int);
};
```

Les classes en C++

En mettant toutes les méthodes en statique, ces dernières peuvent être appelée sans création d'instance, par exemple
`int x = Calcul::add(4,5);`.

Souci : rien ne nous empêche de créer une instance `Calcul c;`.
Solution : mettre le constructeur en privé.

Les classes en C++

```
class Calcul {
private:
        Calcul();
public:
        static int add(int, int);
        static int sub(int, int);
        static int mul(int, int);
        static int div(int, int);
};
```

De cette façon, il n'y a aucun moyen de construire l'instance.

Il s'agit d'un exemple où le constructeur peut être mis en privé.
Un autre exemple est si on souhaite que la création de notre objet ne soit autorisé qu'au sein d'une autre classe ou méthode précise, avec l'amitié.

Les classes en C++

```
class MaClasse {
private:
        MaClasse();
public:
        friend class MaClasse2;
        friend void MaClasse3::create_instance();
};
```

Avec ce genre de design, seules la classe `MaClasse2` ou la méthode `create_instance()` de `MaClasse3` peuvent construire une instance de `MaClasse`.

Plan

L'héritage & polymorphisme

On a vu dans la partie modélisation qu'une classe peut descendre d'une autre classe, parce qu'il y a un sens thématique. Par exemple, un chien peut descendre d'une classe animal.

En C++, on utilise la syntaxe suivante.

```
class MaSousClasse : public MaClasse {
};
```

Ici, `MaSousClasse` hérite de la classe `MaClasse`, avec un contrôle d'accès `public`.

L'héritage & polymorphisme

Le contrôle d'accès `public` signifie que l'on garde les mêmes contrôles d'accès que la classe héritée (aussi appelée super-classe). Si on réalise un héritage privé `private`, toutes les variables et méthodes publiques ou protégées passent en privé. Les éléments privés de la super-classe ne sont pas accessibles. Quant à l'héritage protégé, toutes les variables et méthodes publiques ou protégées passent en protégé. Les éléments privés de la super-classe ne sont toujours pas accessibles.

Les cas d'utilisation n'étant pas évident à ce stade, nous allons garder l'héritage public.

L'héritage & polymorphisme

Lorsqu'on réalise cet héritage, ma sous-classe hérite de toutes les variables et méthodes de la classe héritée.

Attention cependant à l'accessibilité, une sous-classe ne peut accéder qu'aux variables et fonctions `public` ou `protected`, les `private` n'étant accessibles qu'au sein de la classe propre elle-même.

L'héritage & polymorphisme

Lorsqu'on réalise une sous-classe, l'objectif est souvent de spécialiser, et on a donc un besoin à ajouter ou modifier des variables et des méthodes. La redéfinition de méthode consiste à reprendre la même signature de méthode que la classe à laquelle on hérite et de modifier son comportement.

```
#include <string>
class MaClasse {
public:
        std::string getClasse() { return "MaClasse"; }
};

class MaSousClasse : public MaClasse {
public:
        std::string getClasse() { return "MaSousClasse";
};
```

Ici, `getClasse` est une redéfinition au sein de la classe `MaSousClasse` (à ne pas confondre avec la **surcharge**).

L'héritage & polymorphisme

Lorsqu'on crée une instance d'une sous-classe et qu'on souhaite accéder à une méthode, cela va d'abord parcourir les méthodes de la sous-classe qui correspondrait à la méthode appelée (que les paramètres correspondent ou non), et si introuvable ça remonte aux méthodes de la classe supérieure.

Attention, si on fait une redéfinition, cela masque les surcharges de la super-classe.

L'héritage & polymorphisme

```
#include <string>
class MaClasse {
public:
        void methode(float);
        void methode(std::string);
        void methode2(float);
        void methode2(std::string);

};

class MaSousClasse : public MaClasse {
public:
        void methode(float);
};
```

Ici, appeler `methode` avec un objet de `MaSousClasse` va appeler quoi qu'il arrive la méthode de la sous-classe, ce qui va empêcher d'utiliser `void methode(std::string);` de la classe principale.

La `methode2` fonctionnera quoi qu'il arrive, car cela appelera la méthode de `MaClasse`.

L'héritage & polymorphisme

Lorsqu'un objet issu d'une sous-classe est crée, le constructeur de la sous-classe puis de la classe est appelé. Si ce n'est pas spécifié, le constructeur de la classe appelé sera le constructeur vide.

```
#include <string>
class MaClasse {
private:
        std::string str;
public:
        MaClasse();
        MaClasse(std::string);
};

class MaSousClasse : public MaClasse {
public:
        MaSousClasse(std::string);
};
```

Ici, l'appel au constructeur de `MaSousClasse` avec une chaine de caractères va appeler `MaClasse()` et non `MaClasse(std::string)`.

Il faut donc l'appeler à la main, et pour cela on passe par la liste d'initialisation, lors de la définition de la méthode.

```
#include <MaSousClasse.hpp>
MaSousClasse::MaSousClasse(std::string str) : MaClasse(str) {}
```

Ici, l'appel au constructeur de `MaSousClasse` avec une chaine de caractères va bien appeler `MaClasse(std::string)`, de manière explicite.

Cela permet aussi que le constructeur initialise `str`, puisque `MaSousClasse` n'a pas les droits d'accès à la variable `private` de la classe `MaClasse`.

L'héritage & polymorphisme

Cette considération doit être gardée lorsqu'on souhaite faire une construction par recopie.

```
class MaClasse {
public:
        MaClasse(const MaClasse&);
};

class MaSousClasse : public MaClasse {
public:
        MaSousClasse(const MaSousClasse&);
};

MaClasse::MaClasse(const MaClasse& mc) {...}
MaSousClasse::MaSousClasse(const MaSousClasse& msc) : MaClasse(msc) {...}
```

Cela va bien appeler `MaClasse(const MaClasse&)`, et ce malgré le fait que `msc` soit un objet `MaSousClasse` et non `MaClasse`.

Par défaut, si `MaClasse(const MaClasse&)` n'est pas défini, cela va appeler le constructeur par recopie par défaut et donc copier les valeurs des variables communes entre les deux classes.

L'héritage & polymorphisme

En terme de compatibilité des types, on peut toujours transtyper une sous-classe avec une classe supérieure, que ce soit l'objet en lui-même ou son adresse, **si on est en héritage public.**

```
#include <string>
#include <iostream>
class MaClasse {
public:
        std::string getClasse() { return "MaClasse"; }
};

class MaSousClasse : public MaClasse {
public:
        std::string getClasse() { return "MaSousClasse"; }
};


int main()
{
        MaSousClasse msc;
        std::cout << msc.getClasse() << std::endl;

        MaClasse mc{msc};
        std::cout << mc.getClasse() << std::endl;

        MaClasse *mc_ptr{&msc};
        std::cout << mc_ptr->getClasse() << std::endl;

        return 0;
}
```

Cela est dû au fait que l'héritage est une spécialisation, et donc qu'on ajoute ou redéfinit uniquement des choses. Cela apporte une compatibilité avec la super-classe. L'héritage privé ou protégé modifiant les accès aux éléments, ils ne peuvent pas être utilisé comme sous-typage.

À l'exécution du code précédent, on obtient la sortie suivante.

```
MaSousClasse
MaClasse
MaClasse
```

Cela signifie que quel que soit notre manière de transtyper, l'objet est par la suite considéré comme une instance de la classe transtypée.
Cela est dû au fait que la méthode qui sera appelée est déterminée à la compilation, et donc un transtypage revient à utiliser la méthode de la classe transtypée, et non la méthode de la nature réelle de l'objet.

L'héritage & polymorphisme

Pour corriger cela, on peut demander à ce que la détermination de la méthode soit faite à l'exécution, à l'aide du mot-clé `virtual`.

```
#include <string>
class MaClasse {
public:
        virtual std::string getClasse() { return "MaClasse"; }
};

class MaSousClasse : public MaClasse {
public:
        virtual std::string getClasse() { return "MaSousClasse"; }
};
```

L'héritage & polymorphisme

Cette fois-ci, l'exécution donne ce résultat.

```
MaSousClasse
MaClasse
MaSousClasse
```

Le transtypage direct d'un objet va le convertir en l'objet souhaité, là où le transtypage d'un pointeur ne va convertir que le type de l'adresse et donc laisser l'objet intact, mis à part la manière dont on le considère.

À savoir que le transtypage de référence fonctionne de la même façon.

```
int main()
{
        MaSousClasse msc;
        MaClasse &mc_ptr{msc};
        return 0;
}
```

L'héritage & polymorphisme

Avec l'utilisation de méthodes virtuelles, on touche à la notion de **polymorphisme**.
Le polymorphisme est le concept de pouvoir fournir une interface commune à des entités de différents types.

Dans notre cas, cette fonction `getClasse` consiste bien en une interface unique pour traiter différents types, qu'ils soient de type `MaClasse` ou `MaSousClasse`.

En réalité, il s'agit ici d'un polymorphisme dynamique. On peut considérer la surcharge comme étant un polymorphisme statique.
En C++, on appelle **classe polymorphe** tout classe ayant au moins une méthode virtuelle.

L'héritage & polymorphisme

Nous avions évoqué rapidement dans la modélisation UML qu'une classe ou des opérations pouvaient être *abstraites*. Dans les faits, il suffit d'une opération abstraite pour que la classe le devienne. Une classe abstraite ne peut pas être instanciée, mais aura des sous-classes. Ces dernières devront définir proprement les opérations abstraites pour pouvoir être concrètes.

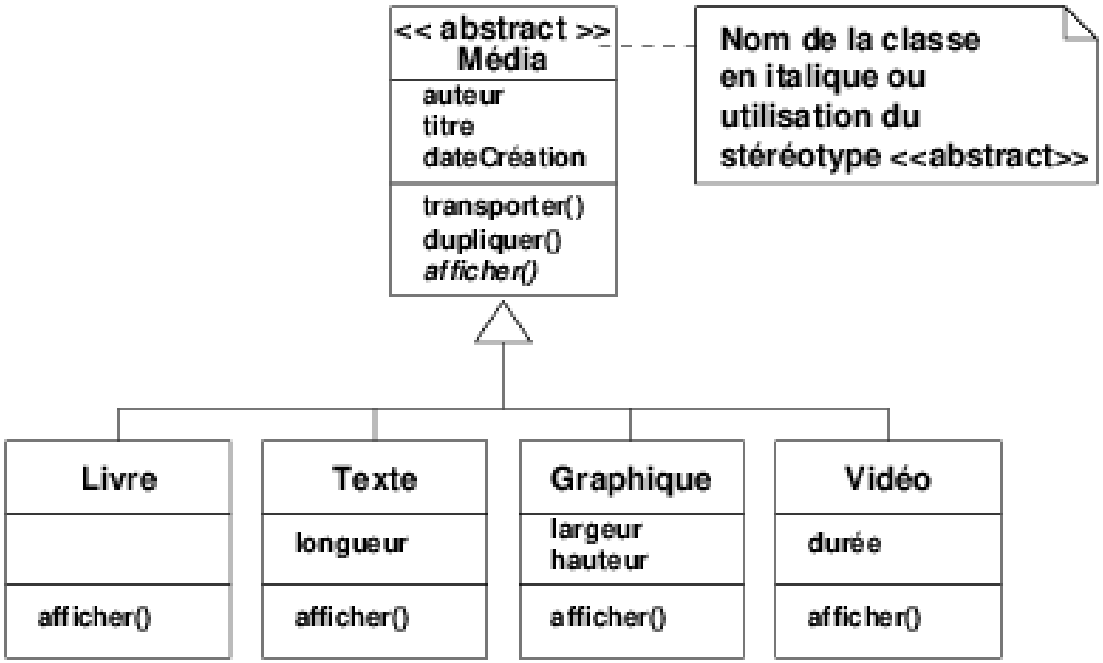


Figure: Diagramme de classe, classe abstraite, D. Conan, C. Taconet, C. Bac, Télécom SudParis, CSC 4002, Octobre 2015

L'héritage & polymorphisme

On constate ici que l'opération `afficher` est abstraite et donc, pour que les sous-classes soient concrètes, il faut qu'elle la redéfinisse.

Par contre, les attributs sont bien réels, et les opérations `transporter` et `dupliquer` sont bien concrètes, et n'ont pas nécessairement besoin d'être surchargé.

L'utilisation de cette classe abstraite a donc permis de mieux classifier les sous-classes en trouvant leur point commun, et in fine factoriser les opérations, là où sans cette classe Média, on aurait eu de la redondance dans les attributs et les opérations.

Cela permet aussi un polymorphisme, puisqu'on peut imaginer une collection de Média, qui peuvent être indifféremment des Livre, Texte, Graphique ou Vidéo, et les traiter via une unique interface.

L'héritage & polymorphisme

En C++, tout comme dans notre modélisation, une classe devient abstraite si une des méthodes l'est. Une méthode abstraite est appelée une méthode virtuelle pure, et peut être définit de la façon d'une méthode virtuelle, mais en ajoutant `= 0` avant le point virgule.

```
class MaClasse {
public:
        virtual int abstraite(int) = 0;
};
```

À partir de ce moment là, `MaClasse` n'est plus instanciable. Si on souhaite définir une sous-classe, il faut absolument définir cette méthode.

```
class MaSousClasse : public MaClasse {
public:
        virtual int abstraite(int a)  { return a; }
};
```

L'héritage & polymorphisme

Même si on ne peut pas l'instancier, on peut toujours transtyper un pointeur pointant vers une instance d'une sous-classe de `MaClasse` en pointeur de `MaClasse`, ou l'équivalent en référence.

```
int main()
{
        MaClasse mc; // non
        MaSousClasse msc; // oui
        MaClasse* mc_ptr = &msc; // oui
        MaClasse& mc_ref = msc; // oui
        std::cout << mc_ptr->abstraite(4) << '\n';
        std::cout << mc_ref.abstraite(4) << std::endl;
        return 0;
}
```

Pourquoi cela fonctionne-t-il ? Car le fait que les méthodes soient virtuelles ont fait que la détermination des méthodes a été réalisée à l'exécution, et donc la fonction `abstraite` qui pourrait être appelée par le pointeur ou la référence `msc_ptr` et `msc_ref` va bien être celle de `MaSousClasse`.

L'héritage & polymorphisme

Dernier aparté sur le mot-clé `virtual` : le destructeur.

```
class MaClasse {
        virtual ~MaClasse();
};
```

Pourquoi le destructeur est-il virtuel ?

L'héritage & polymorphisme

Dernier aparté sur le mot-clé `virtual` : le destructeur.

```
class MaClasse {
        virtual ~MaClasse();
};
```

Pourquoi le destructeur est-il virtuel ?
Pour pouvoir faire appel au destructeur adapté en cas de polymorphisme.

L'héritage & polymorphisme

Voici une manière d'allouer dynamiquement un objet de type `MaSousClasse`, mais dans un pointeur de type `MaClasse`.

```
int main()
{
        MaClasse* mc = new MaSousClasse();
        delete mc;
}
```

Sans `virtual` sur le destructeur, c'est le destructeur de `MaClasse` et non `MaSousClasse` qui sera appelé.

L'héritage & polymorphisme

Dernier mot sur le polymorphisme : les conversions polymorphes.

On a vu qu'un objet d'une sous-classe pouvait être transtypé sans souci vers une instance d'une classe supérieure (en convertissant et en tronquant les informations), et qu'on pouvait aussi transtyper les pointeurs et références.

Dans l'autre sens c'est plus compliqué : on ne peut pas convertir un objet d'une classe en l'objet d'une sous-classe, mais on peut, dans certains cas, le faire sur des pointeurs et des références. Pour cela, on utilise le `dynamic_cast`.

Comme introduit en première partie du cours, il s'agit d'un transtypage qui a lieu à l'exécution, et qui nous donne donc un retour en cas de succès ou d'échec.

- Renvoie le transtypage si tout se passe bien.
- Renvoie `nullptr` si cela échoue et qu'on a voulu transtyper un pointeur.
- Renvoie une **exception** si cela échoue et qu'on a voulu transtyper une référence.

L'héritage & polymorphisme

Dans quel cas l'utiliser ? Lorsqu'on souhaite ramener un type de pointeur ou de référence à celui qui correspond au type réel de l'objet alors qu'on avait déjà transtypé vers une super-classe auparavant, mais **qu'on n'est pas sûr de quel type il est réellement**.

Si on reprend notre exemple précédent, ici on sait de quel type il est réellement, et donc on peut utiliser un `static_cast`.

```
int main()
{
        MaSousClasse msc;
        MaClasse* mc_ptr = &msc;
        MaClasse& mc_ref = msc;
        MaSousClasse* msc_ptr = static_cast<MaSousClasse*>(mc_ptr);
        MaSousClasse& msc_ref = static_cast<MaSousClasse&>(mc_ref);

        return 0;
}
```

L'héritage & polymorphisme

Mais imaginons que nous avons cette construction.

```
class MaClasse {
public:
        virtual int abstraite(int) = 0;
};
class MaSousClasse : public MaClasse {
public:
        virtual int abstraite(int a)  { return a; }
};
class MaSousClasse2 : public MaClasse {
public:
        virtual int abstraite(int a)  { return a+1; }
};
```

L'héritage & polymorphisme

On peut imaginer transporter un `vector` rempli de pointeurs de `MaClasse`, sans savoir s'il s'agit d'un objet de type `MaSousClasse` ou `MaSousClasse2`. C'est dans ce cadre qu'on utilise le `dynamic_cast`.

```
int main()
{
        std::vector<MaClasse*> collec= .......:
        for(int i = 0; i < collec.size(); i++)
        {
                MaSousClasse* msc{dynamic_cast<MaSousClasse*>(collec[i])};
                if(msc == nullptr)
                // c'est un objet de type MaSousClasse2
                else
                // c'est un objet de type MaSousClasse
        }
        return 0;
}
```

L'héritage & polymorphisme

On peut aussi l'utiliser sur des références, mais cela demande une gestion des **exceptions**, que l'on verra plus tard.