4CPRG – Programmation C et C++

Thomas Ménès

Quels sont les objectifs de ce module ?

- Une connaissance de la problématique de l'analyse/conception objet
- La maîtrise des éléments de base d'UML
- La connaissance de l'implémentation en C++ de fragments UML
- La maîtrise du langage C++, de ses variantes et de ses particularités

Éléments du cours

- **1. Rappels sur le langage C :**

Compilation, Syntaxe, Structures, Algorithmes, Types, Pointeurs

- **2. Présentation du langage C++ :**

Abstractions, Iostream, Conteneurs, Fichiers, Exceptions, Concurrence

- **3. Présentation du langage UML :**

Objets, Contraintes, Encapsulation, Héritage, Cas d’utilisation, Interfaces

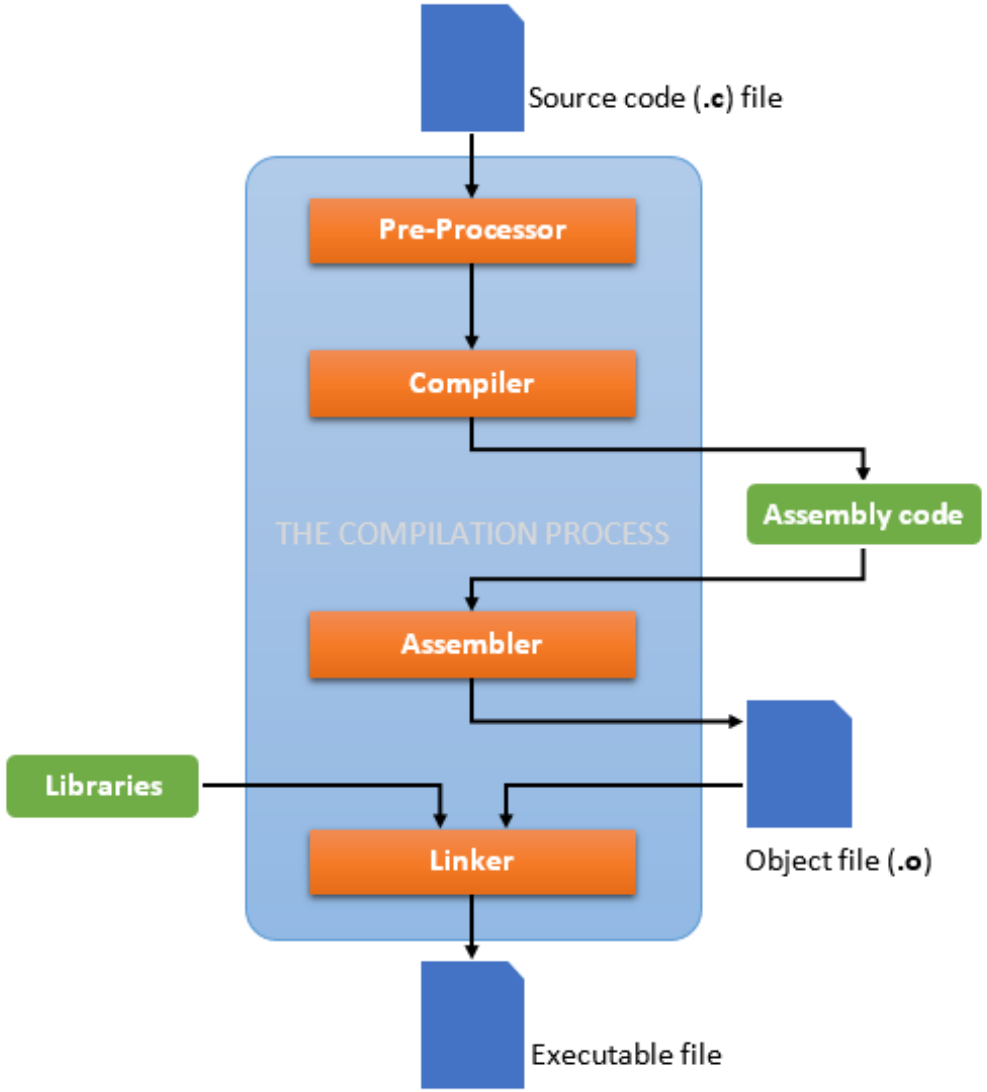
- **4. L’orienté objet en C++:**

Constructeurs, Templates, Design patterns, ORM

#1 Le langage C

Sciences informatiques

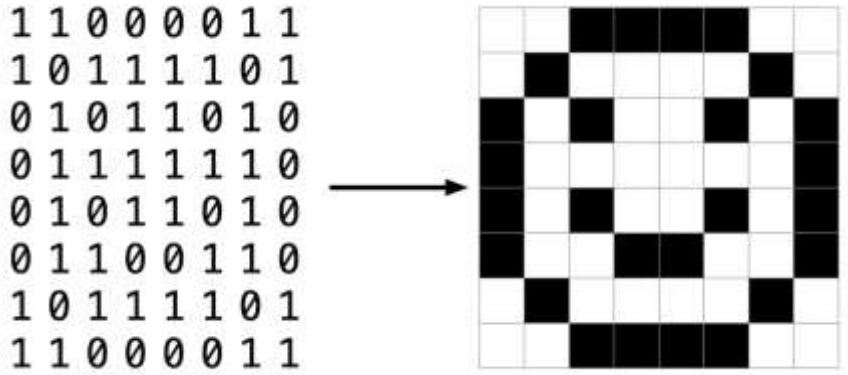
- Avant FORTRAN : 1 langage par machine.
- Assembleur, Unix, C, Compilation
  - 1972 Bell, Dennis Ritchie, Ken Thompson
  - Objdump
- Un ordinateur ne comprend que le binaire
  - Décimal, Binaire, Héxadécimal
  - Table ASCII
  - Int ou char
  - Portes logiques
    - https://www.nand2tetris.org/
- Algorithmes https://the-algorithms.com



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Images



- 24-bit color
- Rgb(140, 21, 21)
  - Red -> 140 = 10001100 en binaire
  - Green -> 21 = 00010101 en binaire
  - Blue -> 21 = 00010101 en binaire
- Tableau contenant les informations de chaque pixel

```
int red_screen_of_9_pixels[3][3][3] = {
    {{255, 0, 0}, {255, 0, 0}, {255, 0, 0}},
    {{255, 0, 0}, {255, 0, 0}, {255, 0, 0}},
    {{255, 0, 0}, {255, 0, 0}, {255, 0, 0}}
};
```

Programmer en C

- Fonction main :
  - Point d'entrée du programme
  - Peut prendre des arguments
  - Retourne un entier
    - Par convention 0 indique que le programme s'est déroulé correctement (code erreurs)

- Code source .c
- Fichier headers .h
  - #define : processeur
- Arguments
- Input et ouput

- Format :
  - %c : caractère simple
  - %d ou %i : entier
  - %f : nombre flottant
  - %lf : large flottant
  - %s : chaine de caractères

```c
int main(int argc, char const *argv[]) {
    /* code */
    int birthdate; // Commentaire
    printf("Année de naissance ? : \n");
    scanf("%i", &birthdate);
    printf("L'année est : %i.\n", birthdate);
    return 0;
}
```

Variables

- Types statiques
- Possibilité de déclarer plusieurs variables du même type sur la même ligne.

```c
int main() {
    int age = 32;
    char yes = 'y', no = 'n';

    age += 1;
    age ++;
    age -= 1;
    age --;
}
```

```c
enum API_options {
    CREATE = 'c',
    READ = 'r',
    UPDATE = 'u',
    DESTROY = 'd'
};
```

Opérateur	Exemple	
	Expression	Equivalent
+=	a += 8	a = a + 8
-=	a -= 8	a = a - 8
*=	a *= b	a = a * b
/=	a /= b+c	a = a /(b+c)
%=	a %= b*c	a = a % (b*c)

Structures conditionnelles et boucles

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    int x = 1, y = 2;
    if (x < y) {printf("x est inférieur à y.\n");}
    else if (x > y) {printf("x est supérieur à y.\n");}
    else {printf("x est égal à y.\n");}

    switch (x > y) {
        case true:
            printf("x est supérieur à y");
            break;
        case false:
            printf("x est inférieur à y");
            break;
        default:
           printf("x est égal à y");
            break;
    }
}
```

```c
#include <stdio.h>

int main() {
    for (int i = 0; i < 50; i++) {
        printf("i = %i\n", i);
    }

    while (x < 10) {
        printf("%i est inférieur à 10.\n", x);
        x ++;
    }

    do {
        printf("y est inférieur à 20");
        y ++;
    }
    while (y < 20);
}
```

Portées et copies

```c
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main(int argc, char const *argv[]) {
    int a = 1;
    int b = 2;
    swap(a, b);
    // Les valeurs modifiées dans swap
    // ne sont pas celles de main.
    printf("A: %i. B: %i.\n", a, b);
    return 0;
}
```

- Chaque élément déclaré ne reste accessible que dans sa portée representee par des accolades {}.
- Ainsi, une variable déclarée à l'intérieur d'une fonction, boucle ou tout bloc de code entouré d'accolades n'est accessible qu'à l'intérieur de celles ci.

```c
int main(int argc, char *argv[]) {
    {int a = 1;}
    int a = 1; // Pas d'erreur
    {a++;} // a est accessible
    int a = 1; // err: redef of ‘a’
    return 0;
}
```

Pointeurs et références

```
char c = 'c';
char *address_of_c = &c;
char c_ref = *address_of_c;
char *s = "string in C";
```

- Un pointeur (signe *) correspond à l'adresse mémoire dans l'ordinateur d'un élément donné.
- On doit toujours indiquer le type de l'élément vers lequel il pointe.
- On récupère l'adresse mémoire d'un élément avec le signe &.
- A partir d'un pointeur, on peut récupérer l'élément vers lequel il pointe avec le signe *. On créé une nouvelle reference du même élément, on parle de déréférencement du pointeur.
- On utilise des guillemets simples ‘’ pour les caractères et doubles “” pour les chaines de caractères.

Structures de données

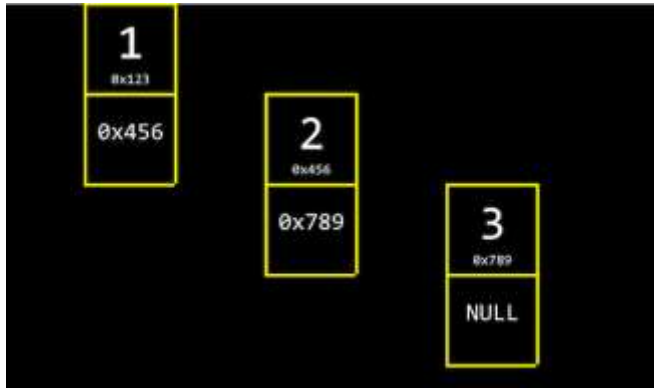
```c
struct Person {
    int id;
    char *name;
};
```

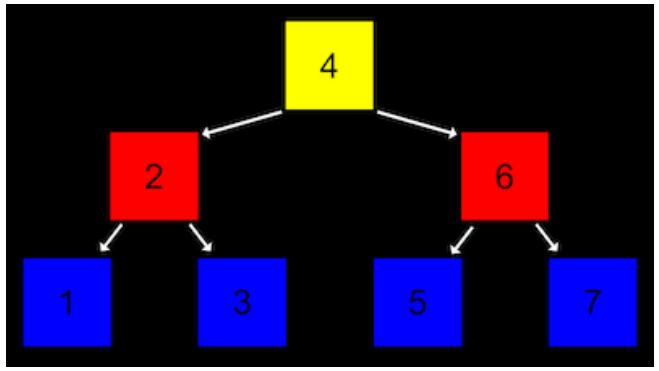
- Une structure de données est un conteneur composé d'autres éléments de différents types.
- On défini une structure avec le mot clé struct.
- On déclare le type des éléments qui composent la structure à l'intérieur d'accolades {}.
- Lors de l'initialisation d'une structure, les attributs à initialiser doivent être passés dans l'ordre.

```c
struct Person toto = {20, "toto"};
printf("%i, %s\n", toto.id, toto.name);
struct Person *ptr_to_toto = &toto;
printf("%i, %s\n", ptr_to_toto->id, ptr_to_toto->name);
struct Person copy_of_toto = toto; // Autre adresse mém
printf("%p, %p\n", &toto, &copy_of_toto);
struct Person reference_of_toto = *ptr_to_toto;
printf("%p, %p\n", &toto, &reference_of_toto);
```

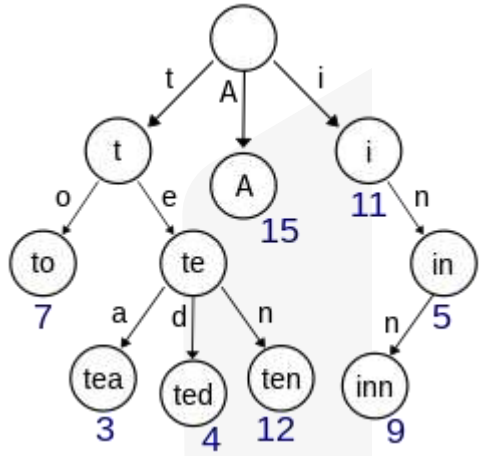
- On accède aux élements d'une structure avec le signe ".".
- Depuis un pointeur, on utilise le signe ->

Types





Trie pour les clés "A", "to", "tea", "ten", "ted", "i", "in", et "inn".




```c
typedef struct Person {
    int id;
    char *name;
    char genre;
    struct Person *m;
} Person;

Person p = {};
printf("%i, %s, %c, %p\n",
p.id, p.name, p.genre, p.m);
// 0, (null), , (nil)
```

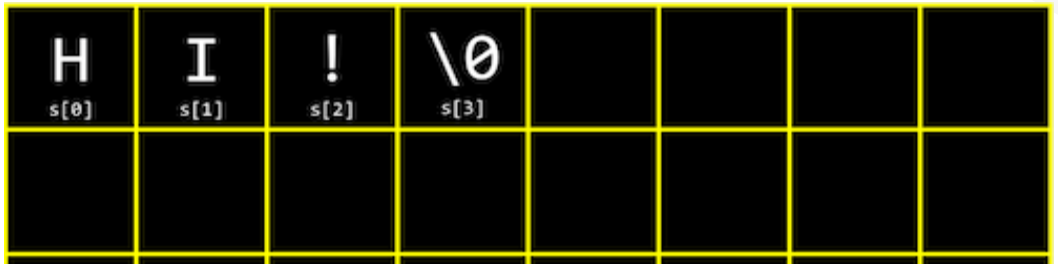
- On défini un nouveau type avec le mot clé typedef. On pourra alors déclarer de nouvelles structures de ce type sans passer par le mot struct.
- Il sera necessaire d'indiquer le nom de la structure après celle ci si elle est composée d'éléments de son propre type.
- Si un attribut n'est pas initialisé, il est mis à 0 pour un nombre et '' pour un caractère. Les pointeurs sont dirigés vers null pour les références et nil pour les pointeurs.
- Listes chainées, arbres, table de hashage (array de pointeurs), trie (arbres préfixes).

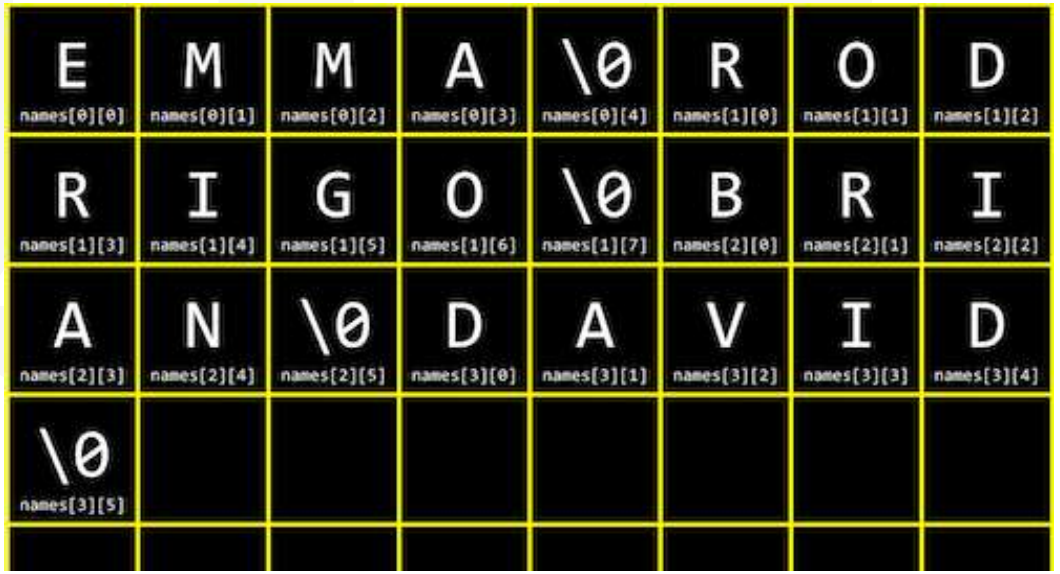
Arrays

```
char *names[] = {
    "EMMA", "RODRIGO", "BRIAN", "DAVID"
};
```

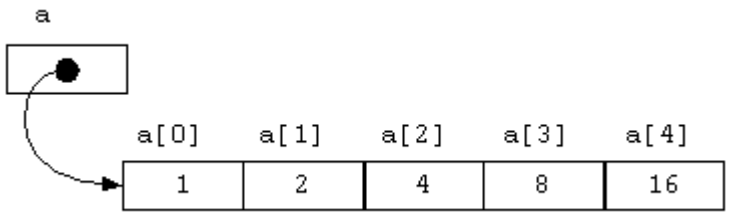
- Mémoire continue.
- Constitué d’élements de même type.
- String est en fait un array de caractères se terminant par \O pour indiquer sa fin.
- On peut lire une chaîne de caractères via string[index] mais pas la modifier.

```
char chars[100];
int numbers[] = {1, 2, 3};
char string[] = {'H', 'I', '!', '\O’};
char *s = "Hi!";
```



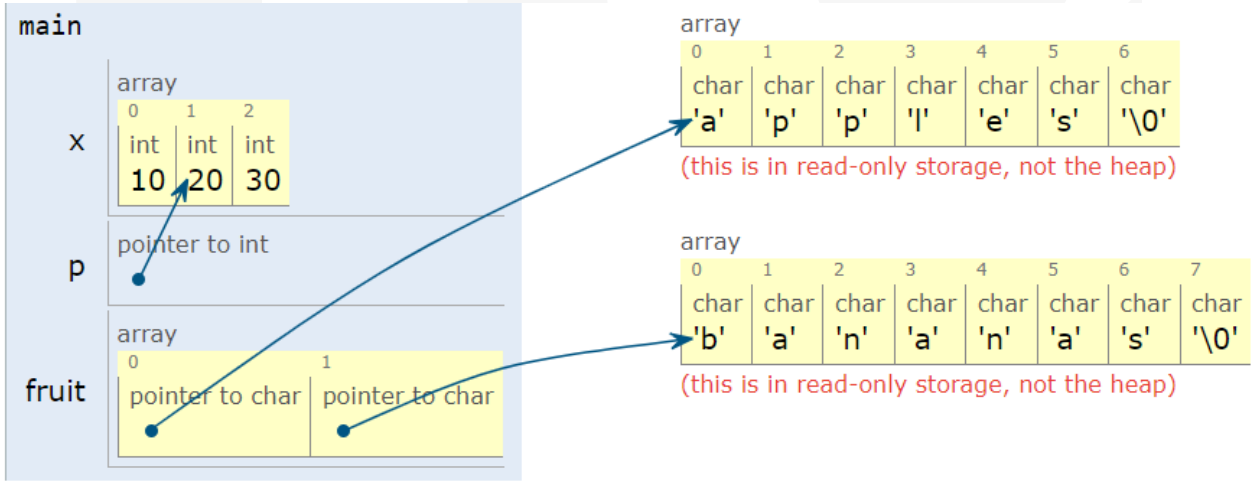


La mémoire



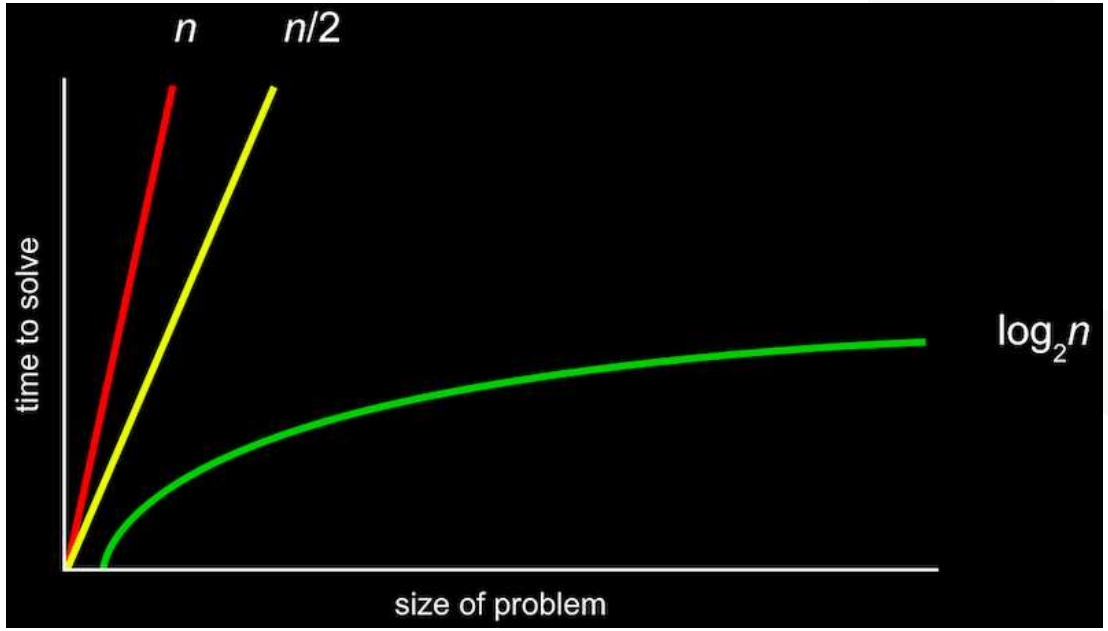
- Un array est représenté par un pointeur vers le premier block.
- https://pythontutor.com/
- Malloc et free : heap.
- On peut itérer sur un array car on connait la taille de ses élements.
- Afin de ne pas déborder de l'array, il nous faudra également le nombre d'élements dans l'array ou bien l'adresse du bloc mémoire de fin.

```
int main() {
  int x[] = {10, 20, 30};
  int* p = &x[1];
  char* fruit[2] = {"apples", "bananas"};
  return 0;
}
```



Algorithmes

- Structures de données de plus haut niveau composées d'autres structures.
- S'adapter le plus possible au problème.
  - Queue (FIFO) et Stack (LIFO).
  - Dictionnaire : mapping entre clés et valeurs ou strings et valeurs.
  - Algorithmes récursifs.



- Running times courants :
  - O(n2)
  - O(n log n)
  - O(n)
    - (linear search)
  - O(log n)
    - (binary search)
  - O(1)

Fonctions mathématiques

- Dans le header « math.h » (« cmath » en C++), il y a notamment les fonctions suivantes, de paramètre double et de résultat double :
  - floor : (resp.ceil) : partie entière par défaut
  - abs : valeur absolue
  - sqrt :racine carrée
  - pow : puissance (pow(x,y) renvoie xy)
  - exp, log, log10
  - sin, cos, tan, asin, acos, atang, sinh, cosh, tanh
- La librairie standard inclut entre autres la fonction rand(int min, int max);

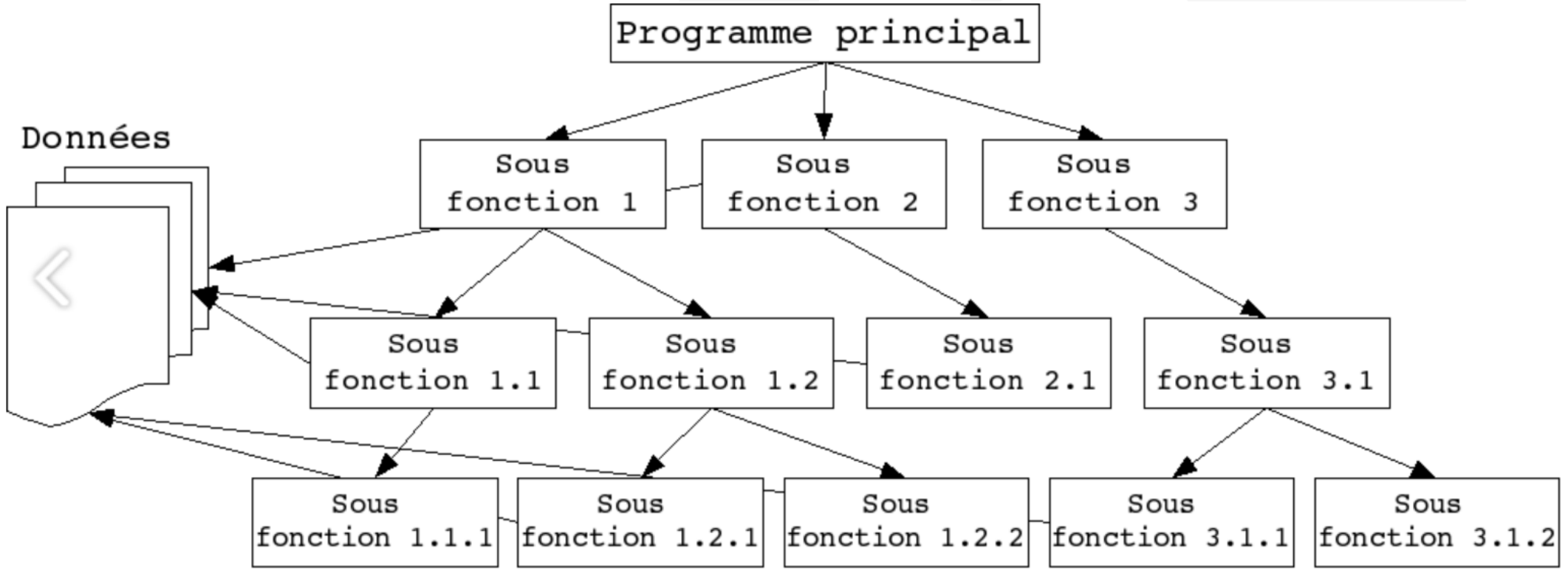
Fichiers

```
FILE *fopen(char *name, char *mode);
int fclose(FILE *f);
int remove(char *nom);
int rename(char *oldname, char *newname);
int putc(char c, FILE *index);
int fputs(char *chaîne, FILE *index);
char *fgets(char *chaine, int n, FILE *index);
```

- Pointeur vers le début du fichier.
- Les fichiers texte terminent par le caractère ASCII EOF (code –1).

- Modes pour les fichiers texte:
  - « r » lecture seule
  - « w » écriture seule (destruction de l'ancienne version si elle existe)
  - « w+ » lecture/écriture (destruction ancienne version si elle existe)
  - « r+ » lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version.
  - « a+ » lecture/écriture d'un fichier existant (mise à jour), pas de création d'une nouvelle version, le pointeur est positionné à la fin du fichier.

Programmation fonctionnelle



Fonctions variadiques

```c
#include <stdio.h>
#include <stdarg.h>

void print(int nb_arguments, ...) {
    va_list args;
    va_start(args, nb_arguments);
    for(int i = 0; i < nb_arguments; i++) {
        char *value = va_arg(args, char*);
        printf("%s\n", value);
    }
    va_end(args);
}
```

- Pour passer un nombre variable d'arguments à une fonction, on peut utiliser un array (comme pour les arguments optionnels dela fonction main) ou bien une fonction dite variadique.
- Dans stdargs.h la structure va_list est définie ainsi que les fonctions va_start, va_end et va_arg.
- De la même manière que pour un array classique, il nous faudra connaitre le nombre d'éléments.

Cyber sécurité

- La sécurité dans le monde reel ? En informatique ?
- https://www.safetydetectives.com/blog/the-most-hacked-passwords-in-the-world/
- Brute force un mot de passe :
  - 4 chiffres : 10.000 possibilités (10 * 10 * 10 * 10)
  - 4 caractères : 78.074.896 possibilités (94 * 94 * 94 * 94)
  - 8 caractères : plus de 6 quadrillons de possibilités
- Bloquer l'utilisateur X secondes s'il entre un mauvais mot de passe plus de n fois d'affilé.
  - On augmente uniquement le temps necessaire au hack.
  - Balance entre sécurité et possibilité que le "vrai" utilisateur se trompe.
- Password Managers, 2Factor authentication, hashing, cryptography, passkey, encryption
- Vision globale
- https://www.stroustrup.com/resource-model.pdf

Intelligence artificielle

- Arbres de decisions
- Jeux qui ont une réponse mathématique :
  - Les inputs et outputs peuvent être représentés de façon mathématique.
  - Jeu de morpion
    - Faire gagner X = 1
    - Faire gagner O = -1
    - Faire égalité = 0
  - Le but de la machine sera de maximiser ou minimiser une fonction.
  - Dans le cas du morpion, on a 255.168 parties possibles.
  - Dans une partie d'échecs, au 4ième coup, on a 288 millions de parties possibles.

- https://fr.wikipedia.org/wiki/Loi_de_Moore/

#2 Le langage C++

Généralités

- Bjarne Stroustrup en 1985.
- Le langage C++ est un des langages les plus célèbres au monde. Il est extrêmement rapide.
- Le C++ est un descendant du C. Il le complète en rajoutant un certain nombre de possibilités.
  - Tout code C est valide en C++.
  - Compilé et typé statiquement.
  - Langage de bas niveau (proche du langage machine) avec des librairies d'abstractions.
- Permet la programmation orientée objet.
  - Class pour définir un nouveau type (descendant spirituel de SIMULA)
  - Ecrire un programme avec des types appropriés
  - Abstraction et polymorphisme
  - Librairies : Structures, types, conteneurs, interfaces, fonctions, templates, regex, duck typing
    - https://cplusplus.com/reference/

Types

Type	Taille
bool	1 octet
char	1 octet
unsigned char	1 octet
short	2 octets
unsigned short	2 octets
int	4 octets
unsigned (int)	4 octets
long	4 octets
unsigned (long)	4 octets
float	4 octets
double	8 octets
long double	10 octets

Nom du type	Ce qu'il peut contenir
`bool`	Peut contenir deux valeurs "vrai" (`true`) ou "faux" (`false`).
`char`	Une lettre.
`int`	Un nombre entier.
`unsigned int`	Un nombre entier positif ou nul.
`double`	Un nombre à virgule.
`string`	Une chaîne de caractères. C'est-à-dire une suite de lettres, un mot, une phrase.

- String est ici un objet
- Ajout des tableaux dynamiques vector

Type auto

- Indique au compilateur d'utiliser l'expression d'initialisation d'une variable ou d'un paramètre d'expression lambda pour déduire son type.
- L'utilisation d'auto pour la plupart des situations (sauf volonté de conversion) offre ces avantages :
  - Robustesse : si le type de l'expression est modifié, y compris lorsqu'un type de retour de fonction est modifié, il fonctionne simplement.
  - Performances : vous êtes garanti qu'il n'y a pas de conversion.
  - Facilité d'utilisation : vous n'avez pas à vous soucier des difficultés d'orthographe de nom de type et des fautes de frappe.
  - Efficacité : votre codage peut être plus efficace.
- Cas de conversion dans lesquels vous ne souhaiterez peut-être pas utiliser auto :
  - Vous voulez un type spécifique et rien d'autre ne le fera.
  - Dans les types d'assistance de modèle d'expression.
- Utilisez auto au lieu d'un type pour déclarer une variable et spécifiez une expression d'initialisation. Vous pouvez modifier le mot clé à l'aide de spécificateurs et de déclarateurs tels que const , pointeur (*), référence (&) et référence rvalue (&&).
- Le compilateur évalue l'expression d'initialisation et utilise ensuite ces informations pour déduire le type de la variable.
- L'expression auto d'initialisation peut prendre plusieurs formes :
  - Syntaxe d'initialisation universelle, telle que auto a {42};
  - Syntaxe d'affectation, telle que auto b = 0;
  - Syntaxe d'affectation universelle, qui combine les deux formes précédentes, telles que auto c = {3.14159};
  - Initialisation directe ou syntaxe de style constructeur, telle que auto d(1.41421f);
- Lorsqu'il est utilisé pour déclarer le paramètre de boucle dans une instruction basée sur une itération, il utilise une syntaxe d'initialisation différente, par exemple for (auto& i : iterable) do_action(i);

Input et output

- Console in et out
- getline pour plusieurs mots
- Définies à l'intérieur du namespace std, dans iostream

CARACTERE	
'\n'	interligne
'\t'	tabulation horizontale
'\v'	tabulation verticale
'\r'	retour chariot
'\f'	saut de page
'\\'	backslash
'\''	cote
'\"'	guillemets

```cpp
#include <iostream>
#include <string>

int main(int argc, char const *argv[]) {
    std::cout << "argument count : " << argc << std::endl;
    for (auto i; i < argc; i++) {
        std::cout << "argument " << i << " value : " << argv[i] << std::endl;
    }
    std::string s;
    std::cout << "Entrez une chaine de caractères : " << std::endl;
    std::cin >> s;
    std::cout << "Valeur de s : " << s << std::endl;
    return 0;
}
```

Allocation de mémoire

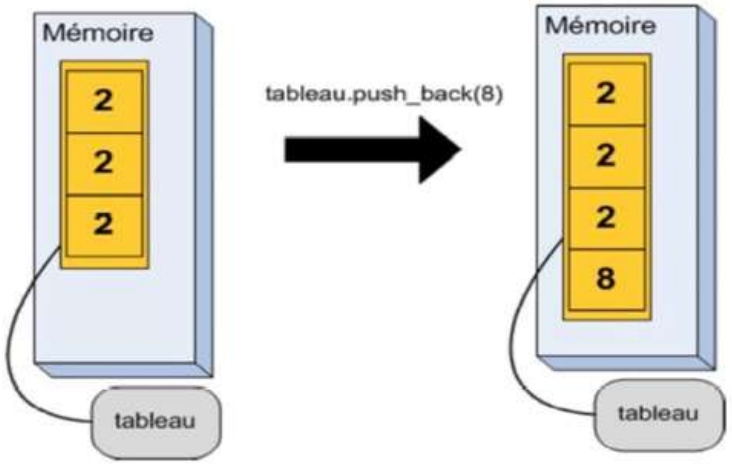
```
int main() {
    int *pointeur(0);
    pointeur = new int;
    delete pointeur;
    pointeur = 0;
}
```

- Le mot clé new renvoie un pointeur vers un block de mémoire disponible.

- Une fois que l'on a plus besoin de la case mémoire, il faut la rendre à l'ordinateur. Cela se fait via l'opérateur delete.
- La case est alors rendue à l'ordinateur qui pourra l'utiliser pour autre chose. Le pointeur, lui, existe toujours. Et il pointe toujours sur la case mais vous n'avez plus le droit de l'utiliser.
- Il est donc essentiel de supprimer cette "flèche" en mettant le pointeur à l'adresse 0 après avoir utilisé delete.
- Smart pointers:
  - Uniques, partagés ou faibles.
- Null pointer.

```
auto p1 = std::make_unique<int>(10);
auto p2 = std::make_shared<int>(10);
auto p3 = nullptr;
```

Array dynamique



- Initialisation
- Méthodes :
  - size()
  - push_back(value)
  - pop_back()
  - begin()
  - end()
- Tableau multidimensionnel

```
int main() {
    auto length(3), value(2);
    std::vector<int> tableau(length, value);
    tableau.push_back(8);
    return 0;
}
```

Les tests

```cpp
#include <cassert>
int main() {
    auto const a(1), b(2);
    assert(a < b);
}
```

- Unitaires :
  - Tester un composant individuellement et s'assurer qu'il fonctionne comme prévu.
  - Être très léger en termes de code et de performance.
  - Renvoyer un bon rapport en cas d'erreur ou de régression
- Fonctionnels :
  - Tester une fonctionnalité de bout en bout (End-to-end).
  - L'objectif ici est de vérifier le fonctionnement d'une fonctionnalité entière du point de vue de l'utilisateur.
- D’integration :
  - Vérifient que les différentes parties du programme, testées individuellement via des tests unitaires, fonctionnent bien une fois intégrées ensemble.
  - L'idéal est de tester des cas d'usages réels ou très proches du réel.

Fichiers

- C++ met à disposition différentes classes et fonctions pour abstraire le concept le fichier.
  - std::ofstream
  - std::ifstream
  - std::fstream
  - std::getline()
- Un fichier possède des méthodes qui le modifient ou produisent un résultat.

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

int main() {
    ofstream MyFile("filename.txt");
    MyFile << "Ajouté au fichier!";
    MyFile.close();

    string myText;
    ifstream MyReadFile("filename.txt");
    while (getline (MyReadFile, myText)) {
        cout << myText;
    }
    MyFile.close();

    return 0;
}
```

Exceptions

- try {expression}
- catch(erreur) {expression}
- On peut provoquer une exception avec le mot clé throw.
- std::cerr pour l'affichage standard des erreurs.
- catch(…) pour rediriger tous les types d'erreur possibles. Peut engendrer un comportement inattendu.

```cpp
#include <iostream>
#include <cassert>

int main() {
    try {
        int age = 15;
        assert(age > 18);

        if (age >= 18) {}
        else if(age == 18) {throw 400;}
        else {throw "LessThan18";}
    }
    catch(const std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
    catch (int error_code) {
        std::cerr << "Err " << error_code << std::endl;
    }
    catch (char *error_name) {
        std::cerr << "Err " << error_name << std::endl;
    }
    catch(...) {
        std::cerr << "Err" << std::endl;
    }
}
```

Concurrence

- Quand un programme lance plusieurs tâches (threads) en même temps.
- Il peut y avoir des dépendances dans les tâches :
  - Certaines tâches ne peuvent se faire avant d’autres.
  - Certaines tâches ne peuvent se faire en même temps.
- Parallélisme (ou asynchrone) : aucunes tâches interdépendantes.

```
#include <iostream>
#include <thread>

int main() {
    std::thread my_thread{[](int x) {
        std::cout << “Appelé depuis le thread " << x << std::endl;
    }, 25};
    my_thread.join();
    return 0;
}
```

Left et right values

```
std::string get_right_value() {
    return std::move("Not moving");
}
```

- Par défaut, tous les éléments créés sont supprimés lorsque l'on sort de leur niveau d'imbrication.
- Lorsque l'on affecte une valeur à un élément, c'est la partie à gauche du signe = qui "possède" l'adresse des blocs mémoire où seront enregistrées les valeurs renvoyées par la partie droite du signe.
- Pour éviter d'allouer de la mémoire pour un élément qui sera ensuite retourné (et donc de nouveau écrit là où la partie gauche du signe = s'est faite attribuer de la mémoire), on peut utiliser la fonction std::move.
- Cette fonction porte mal son nom car l'objet retourné n'est justement pas déplacé des blocs alloués initialement vers ceux alloués par la partie gauche du signe =.

```
auto left_value = get_right_value();
```

#3 Unified Modeling Language

Introduction

- Première version en 1997, v2 en 2005.
- Modélisation des systèmes et des processus de manière graphique.
- Basé sur l'approche objet :
  - Apparu bien avant l'UML (SIMULA dans les années 60 puis C++ ou Java).
- Permet une architecture "Model Driven".
- Permet de formuler les attentes du système (chefs de projets).
- Promu par l'OMG : Object Management Group, consortium de plus de 800 sociétés et universités actives dans les domaines technologiques de l'objet.
- Rend abstrait de nombreux aspects techniques.
- https://laurent-audibert.developpez.com/Cours-UML/

L’approche par objet

- Un objet est une entité identifiable du monde reel. Il peut être désigné de manière unique.
- Il peut avoir une existence physique (un livre, un ordinateur) ou pas (un texte de loi, un fichier, un dossier, une fonction ou encore une ligne de code).
- Possède un ensemble d’attributs (sa structure) et de méthodes (son comportement).
- Un attribut est une variable destinée à recevoir une valeur.
- Une méthode est un ensemble d’instructions (pouvant prendre des valeurs en entrée) qui modifie les valeurs des attributs ou bien produit un résultat.
- En UML, même les objets statiques sont perçus comme dynamiques : on considère qu’un livre peut s’ouvrir seul à la nième page.

Abstractions

- Pour un même objet, on ne retiendra pas les mêmes attributs et méthodes en fonction de l'application.
- On ne retient que les propriétés pertinentes pour un problème donné.

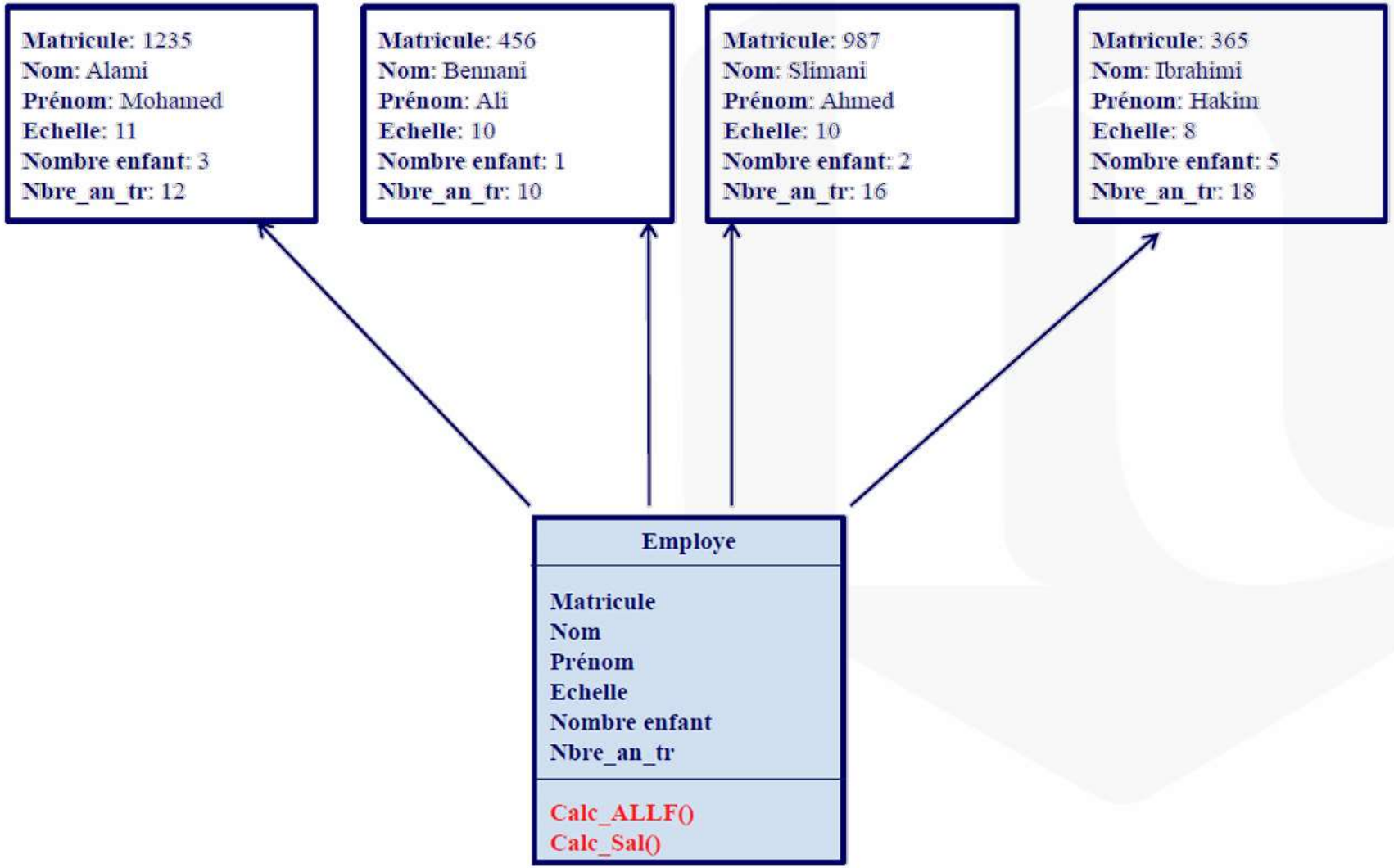

```
class Personne {
public :
        string nom;
        string societe;

                void presenteToi( ) {
                cout << "je suis " << nom << endl ;
                cout << "je travaille à " << societe << endl ;
                }
};
```

Classe d’objets

- Ensemble d’objets similaires, possédants la même structure et le même comportement.
- La structure et le comportement peuvent alors être définis au niveau de la classe.
- Chaque objet de la classe, appelé instance, se distingue par son identité unique et de valeurs spécifiques pour ses attributs.
- Chaque attribut peut avoir différentes contraintes (unique, positif etc)
  - OCL : Langage de contraintes des objets.

Class Employe



Encapsulation

- Masquer certains attributs ou méthodes de l'objet vis-à-vis des autres objets.
- Certains attributs ou méthodes ont pour unique objectif des traitements internes à l'objet et ne doivent pas être lus, exécutés ou modifiés depuis l'extérieur.
- C'est une abstraction car on simplifie la representation vis-à-vis des objets extérieurs.
- Cette representation simplifiée de l'objet, visible de l'extérieur est constitué uniquement de ses attributs et méthodes dites publiques.
- Exemple de la voiture : L'utilisateur ne peut faire tourner les roues plus vite autrement qu'avec la pédale qui active des fonctions internes non disponibles depuis l'extérieur.

```
class CreditCard {
    public:
        CreditCard(string number) : _number(number) {}
        string number() {return _number.substr(8);}

    private:
        string _number;};
```

Modélisation

```
class CoffeMachine {
    public:
        Coffe makeCoffe() {};
    protected:
        void on() {};
    private:
        void boilWater() {};
};
```

- Public : +
  - Non encapsule, visible par tous
- Protégé : #
  - Encapsulé, visible uniquement depuis la classe et ses sous classes
- Privé : -
  - Encapsulé, visible uniquement depuis la classe
- Package : ~
  - Encapsulé, visible uniquement dans les classes du même package

- En C++, par défaut, les attributs d'une structure sont publics et ceux d'une classe sont privés.
  - Les objets exposent un comportement et masquent les données.
  - Les structure exposent des données et n'ont pas de comportement significatif.

Attributs de classe et d’instance

- Par défaut, chaque instance d'une classe possède sa propre copie des attributs de la classe. Les valeurs des attributs peuvent donc différer d'un objet à un autre.
- Il est parfois nécessaire de définir un attribut de classe qui garde une valeur unique et partagée par toutes les instances de la classe.
  - Les instances ont accès à cet attribut, mais n'en possèdent pas une copie.
  - Un attribut de classe n'est pas une propriété d'une instance, mais une propriété de la classe.
  - L'accès à cet attribut ne nécessite pas l'existence d'une instance.

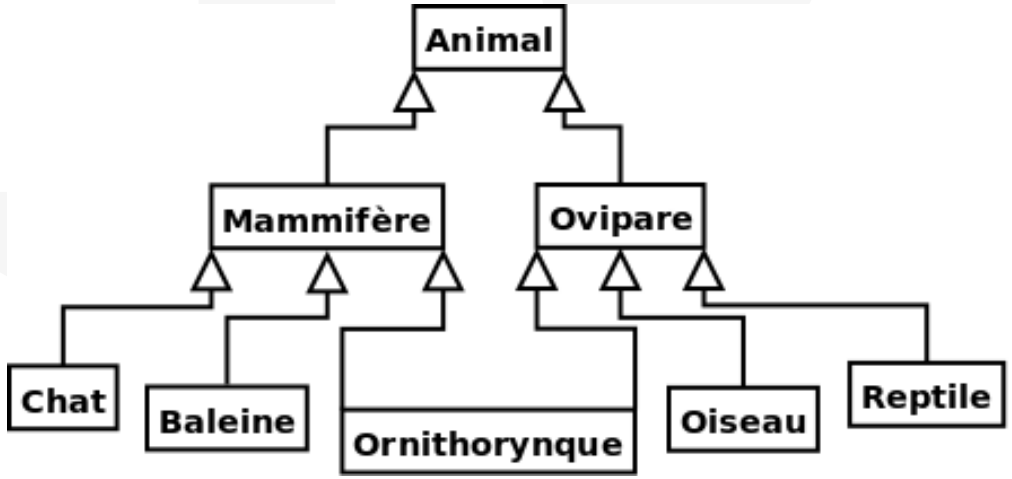
```
class User {
    public:
        static const vector<string> actions;
};
const vector<string> User::actions = {"login", "logout"};
```

Spécialisation et généralisation

- Une classe peut être définie comme étant un sous ensemble d'une autre.
- Elle constitue ainsi une spécialisation. On parle aussi de sous-classe.
- L'héritage fait bénéficier à la sous-classe de la structure de sa sur-classe.
  - Les instances d'une sous-classe sont aussi des instances de la sur-classe.
  - On parle aussi de classe enfant / parent.

```
// Classe de base
class Personne {
   private: string nom;
   public:
      Personne(string n) { nom = n }
};

// Classe dérivée
class Etudiant : public Personne {
   private: string ine;
   public:
      Etudiant(string nom, string ine) : Personne(nom), ine(ine) { }
};
```



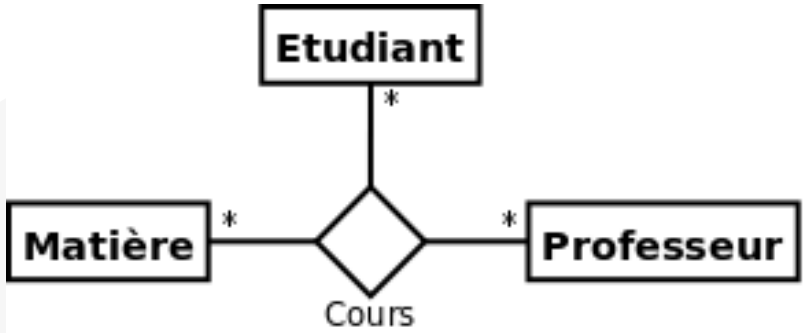
Classes abstraites et concrètes

- Une classe concrète possède des instances.
- Une classe abstraite ne possède pas directement d'instances :
  - Ne fait qu'abstraire une structure de données et un comportement.
  - A vocation à être utilisée en tant que classe parente.

- Interfaces :
  - Concept abstrait : Une interface ne s'instancie pas.
  - Ensemble de fonctionnalités qui pourront être données à des objets sans se soucier de leur type.
  - Exemples :
    - Les objets sur lesquels on peut itérer sont des itérables.
    - Une fonction "allumer" pourrait allumer une allumette comme un moteur.
      - On parlerait d'objet "allumable".

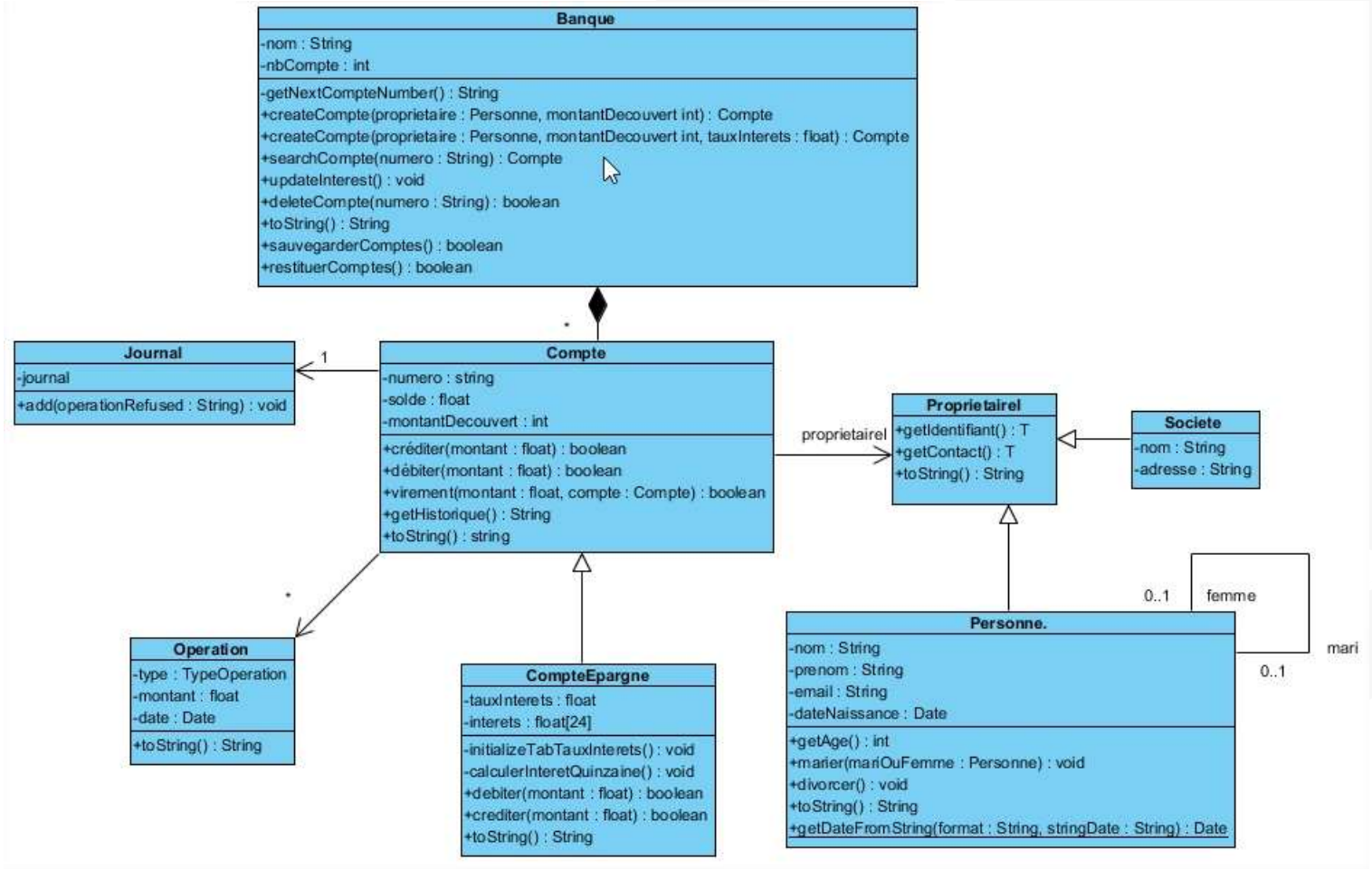
- Public :
  - Public –> public
  - Protected -> protected
- Protected :
  - Public -> protected
  - Protected -> protected
- Private :
  - Public -> private
  - Protected -> private

Aggrégation et composition

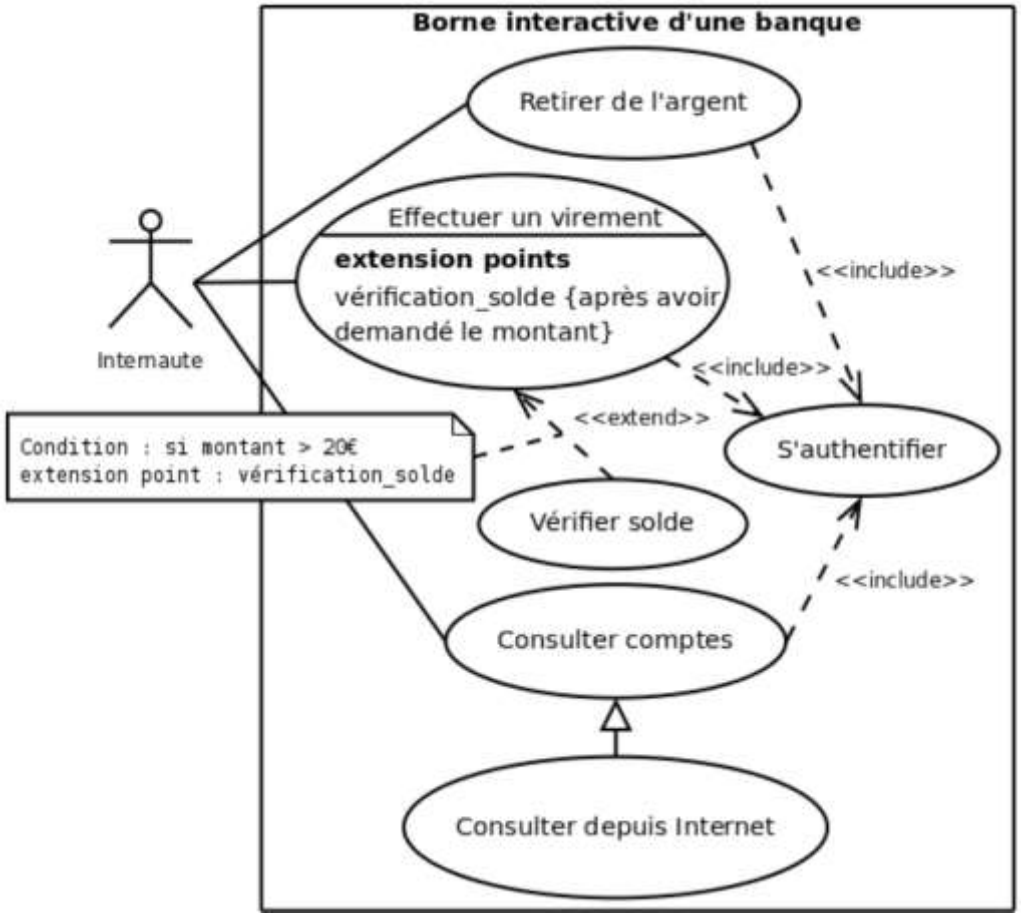


- Un objet peut être composé d'autres objets.
- Si l'objet aggrégé a pour unique but de faire partie du premier objet (il ne peut pas exister seul), on parle de composition.
- Association binaire :
  - Trait plein entre les classes associées.
  - Peut être ornée d'un nom, avec éventuellement une précision du sens de lecture (▸ ou ◂).
  - Quand les deux extrémités de l'association pointent vers la même classe, l'association est dite réflexive.
- Une association n-aire lie plus de deux classes. On parle aussi de classe d'association.
  - On représente une association n-aire par un grand losange avec un chemin partant vers chaque classe participante.
  - Le nom de l'association, le cas échéant, apparaît à proximité du losange.

Diagramme de classes

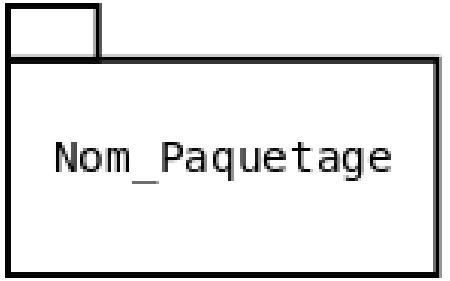


Cas d'utilisation



- Exprimer les exigences fonctionnelles.
- Vérifier que le système répond à ces exigences.
- Déterminer les frontières du système.
- Ecrire la documentation
- Construire les jeux de tests.
- <<include>>
  - Le cas A depend de B.
  - Lorsque A est sollicité, B a forcément déjà été ou sera sollicité comme une partie de A.
- <<extend>>
  - A peut être appelé au cours de l'exécution de B.
  - Exécuter B peut éventuellement entrainer l'execution de A.
  - Contrairement à l'inclusion, l'extension est optionnelle.

Package et namespace



- Regroupement d'éléments de modèles et de diagrammes.
- Organisation des éléments en groupes.
- Peut contenir tout type d'élément de modèle :
  - Classes, cas d'utilisations, interfaces, diagrammes…
  - Des paquetages imbriqués (décomposition hiérarchique)

```
namespace n {int a = 2;}
int main() {int b = n::a;}
```

- On détermine un élément donné de façon unique par son nom qualifié
  - Constitué de la série des noms des paquetages ou des autres espaces de noms depuis la racine jusqu'à l'élément en question.
  - Chaque espace de nom est séparé par deux doubles points (::).
  - Si un paquetage B est inclus dans un paquetage A et contient une classe X, il faut écrire A::B::X pour pouvoir utiliser la classe X en dehors du contexte du paquetage B.

Sources et headers

- La meilleure stratégie dépend de votre projet spécifique et de ses exigences. Si vous travaillez sur un grand projet avec de nombreuses dépendances, il peut être préférable d'opter pour une stratégie qui minimise les temps de compilation. D'un autre côté, si vous travaillez sur un petit projet ou si la lisibilité du code est une priorité, il peut être préférable d'inclure toutes les dépendances dans les fichiers d'en-tête.
- Tout inclure dans les fichiers d'en-tête (.hpp)
- Inclure le minimum possible dans chaque fichier
- Inclure le moins de fichiers possible (tout inclure d'un coup)
- Inclure les fichiers d'en-tête dans un fichier d'en-tête commun
- Utiliser des fichiers de précompilation
- Utiliser des modules C++ (C++20 et versions ultérieures)

#4 L’orienté objet en C++

Constructeurs et destructeurs

```
struct MyClass {
    MyClass() {
        cout << "Défaut\n";
    }
    explicit MyClass(int value) {
        cout << "Paramétré\n";
    };
    MyClass(MyCLass &other) {
        cout << "Copie\n";
    }
    MyClass(MyClass &&other) {
        cout << "Déplacement\n";
    }
    ~MyClass() {
        cout << "Destructeur\n";
    }
};
```

- Constructeur par défaut : Si aucun constructeur n'est défini pour une classe, le compilateur en génère un par défaut sans arguments.
- L'initialisation d'un objet avec {} est appelée initialisation par liste. Cela peut conduire à des comportements inattendus, comme l'initialisation des attributs à zéro. Il est donc préférable d'utiliser le constructeur pour initialiser les objets.
- Le mot-clé explicit est utilisé pour éviter les conversions implicites. Il est généralement utilisé avec le constructeur d'une classe.
  - Par exemple, T(" ") ou T(nullptr) au lieu de T(0) entraîneront des erreurs de type.
- Le destructeur est exécuté chaque fois qu'un objet de sa classe sort de la portée ou est explicitement supprimé. Un destructeur est appelé pour un objet après que son cycle de vie se termine.

Opérateurs

```cpp
bool Node::operator<(Node &other) {
    if (_lat != other._lat) {
        return _lat < other._lat;
    }
    return _long < other._long;
}

bool Node::operator==(Node &other) {
    return (
        (_lat == other._lat)
        and (_long == other._long)
    );
}
Node &Node::operator=(Node &other) {
    id = other.id;
    return *this;
}
```

- Permet aux développeurs de donner une signification spéciale aux opérateurs standard.
- + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= <=>(depuis C++20) && || ++ -- , ->* -> ( ) [ ]
- https://en.cppreference.com/w/cpp/language/operators
- L'opérateur d'affectation est utilisé pour remplacer les données d'un objet déjà initialisé par les données d'un autre objet alors que le constructeur de copie est utilisé pour initialiser un nouvel objet à partir des données d'un objet existant.

Polymorphisme

- Quand une classe représente un ensemble d'instances de classes différentes.
- Comportement different lors d'appels à des méthodes de même nom :
  - On login un Utilisateur mais le comportement n'est pas le même s'il est Admin ou Client (redirection vers la Home ou vers le back office).
  - On parle de surcharge de méthode.
- CharArray {'0'} vs IntArray {0}

```
class Person {
    public:
        virtual void say_hi() {
            cout << "HI" << endl;}
        void no_polymorphism() {
            cout << "HI" << endl;}
};
class Client : public Person {
    public:
        void say_hi() override {
            cout << "Client" << endl;}
        void no_polymorphism() {
            cout << "Client" << endl;}
};
class Admin : public Person {};

int main() {
    auto c = Client(); auto a = Admin();
    c.say_hi(); // -> Client
    admin.say_hi(); // -> HI

    Person *person = &c;
    person->say_hi() // -> Client
    person->no_polymorphism();  // -> HI}
```

Wrappers

```
template <typename AnyType>
class AddPrints {
    private:
        AnyType func;

    public:
        AddPrints(AnyType func):
            func(func) {}

        void operator ()() {
            printf("Before\n");
            func();
            printf("After\n");
        }
};
```

```
void some_task() {
    printf("Some task\n");}

void logger(void (*func)()) {
    printf("Before\n");
    func();
    printf("After\n");}

int main() {
    logger(&some_task);
    return 0;}
```

- Adaptateur : Convertit une interface en une autre (attendue par un Client).
- Façade : Fournit une interface simplifiée.
- Décorateur : Ajoute dynamiquement des responsabilités aux méthodes d'une interface sans la modifier.

Templates

```
class Card {
    public:
        virtual int max_withdrawal() const = 0;
};

class GoldCard : public Card {
    public:
        int max_withdrawal() const override {return 100;};
};


template<typename Card>
class Withdrawal {
    Card &creditcard;

    public:
        Withdrawal(Card &card, int amount) {
            if (amount > card.max_withdrawal()) {...}
        };
};
```

- Polymorphisme :
  - On donne à un objet de nouvelles fonctionnalités.
- Code générique et réutilisable.
- Interfaces en arguments.
- Cas d’utilisations.
- Permet de nombreux patterns :
  - Injection des dépendances
  - Décorateur
  - Fabrique et fabrique abstraite
  - Classes de tests
  - Composite
  - Singletons
  - Observateur
  - …

Durée de vie

```
HEAP SUMMARY:
in use at exit: 0 bytes in 0 blocks
total heap usage: 10,481 allocs, 10,481 frees, 857,053 bytes allocated
All heap blocks were freed -- no leaks are possible
ERROR SUMMARY: 0 errors from 0 contexts
```

- Les pointeurs intelligents sont des types abstraits de données qui simulent le comportement de pointeurs en ajoutant des fonctionnalités telles que la libération de la mémoire ou la vérification des bornes. Ils expriment au programmeur comment gérer les objets retournés.
  - Les unique_ptr ont la responsabilité de libérer la mémoire des objets qu'ils possèdent.
  - Les shared_ptr implémentent le comptage de références, ce qui permet de partager l'objet possédé entre plusieurs shared_ptr sans se soucier de comment libérer la mémoire associée. Lorsque le dernier shared_ptr est détruit, l'objet pointé est également détruit.
  - Les weak_ptr permettent d'accéder à une ressource possédée par un shared_ptr mais n'ont aucune influence sur sa destruction. Ils servent principalement en cas de références circulaires.
- https://learn.microsoft.com/fr-fr/cpp/standard-library/memory
- https://valgrind.org/

```
class Person {
    unique_ptr<Company> company = make_unique<Company>();
    void swap_company(Person &other)
       {company.swap(other.company);}};
```

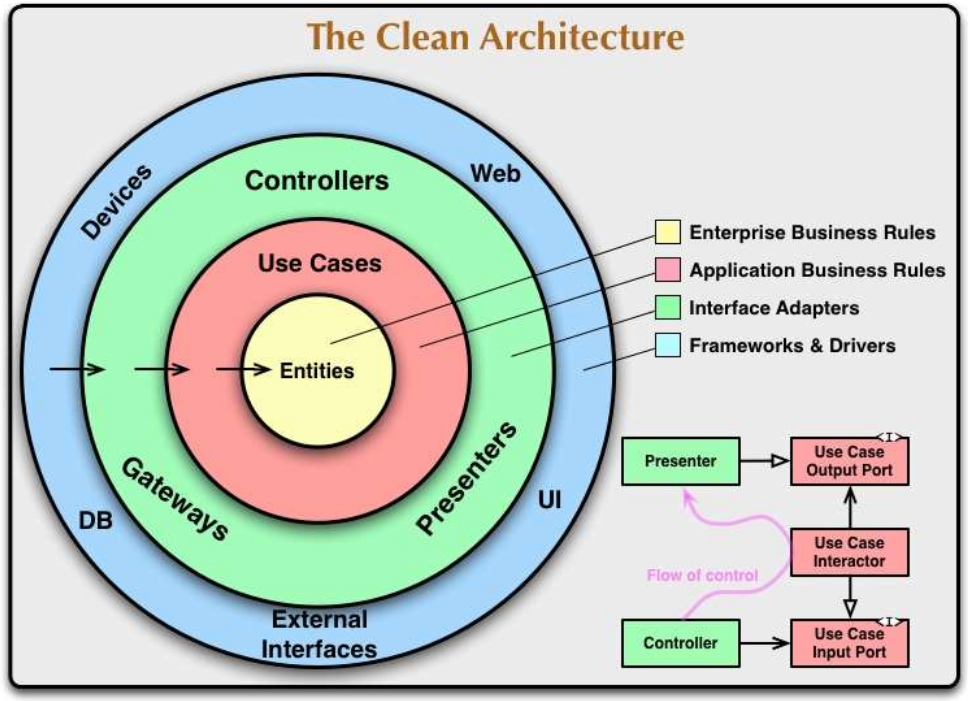
Object relational mapping

- Active record :
  - Les attributs d'une table ou d'une vue sont encapsulés dans une classe, chaque instance de la classe, est lié à un tuple de la base de données.
    - Après l'instanciation d'un objet, un nouveau tuple est ajouté à la base au moment de l'enregistrement.
    - Chaque objet récupère ses données en base ; quand un objet est mis à jour, le tuple l'est aussi.
    - La classe implémente des accesseurs pour chaque attribut.
  - On manipule un enregistrement actif comme une structure de données :
    - Variables publiques et fonctions de navigation (save, update…).
    - Injection d'objets séparés contenant les règles métier qui exposent un comportement mais cachent leur données internes (souvent une instance de l'enregistrement actif).
- Data mapper (repository pattern) :
  - Echange de données bidirectionnel entre un stockage de données persistant (souvent une base de données relationnelle) et une représentation de données en mémoire (la couche de domaine).
  - Indépendance entre la représentation en mémoire, le stockage de données et le Data Mapper lui-même.
  - Particulièrement utile lorsqu'il est nécessaire de modéliser et d'appliquer des processus métier rigoureux sur les données de la couche de domaine qui ne correspondent pas exactement au stockage de données persistant.
  - Un ou plusieurs mappers (ou Data Access Objects), qui effectuent le transfert de données.

SOLID

- Responsabilité unique (Single responsibility principle)
  - Une classe, une fonction ou une méthode doit avoir une et une seule unique raison d'être modifiée.
- Ouvert/fermé (Open/closed principle)
  - Une entité applicative (classe, fonction, module ...) doit être fermée à la modification directe mais ouverte à l'extension.
- Substitution de Liskov (Liskov substitution principle)
  - Si B est une sous-classe de A, alors tout objet de type A peut être remplacé par un objet de type B sans altérer les propriétés désirables du programme.
- Ségrégation des interfaces (Interface segregation principle)
  - Plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale.
- Inversion des dépendances (Dependency inversion principle)
  - Il faut dépendre des abstractions, pas des implémentations.
- https://fr.wikipedia.org/wiki/Patron_de_conception

Clean architecture



- https://blog.cleancoder.com
- Les frameworks avec dépendances « réinversées » plutôt que de tout configurer sont très puissants. Cela ne signifie pas que vous devez coupler votre système. Les conventions ne provoquent pas forcément de couplage.

- Découplage de l'interface utilisateur des règles métier :
  - Plus grande flexibilité et facilite les tests.
  - Le code de l'application est complètement découplé de l'interface utilisateur.
- « Cacher » le framework, l'interface utilisateur et la base de données du code de l'application.
- Le découplage n'entraîne pas de ralentissement, au contraire, il accélère le processus de développement :
  - Exécuter les tests sans délai
  - Permet de reporter les décisions concernant l'interface utilisateur, les framework, la base de données etc.
- Différer les décisions critiques :
  - Cela ne signifie pas que vous devez les reporter, mais si vous pouvez, cela signifie que vous disposez d'une grande flexibilité.