JUNIA Grande école d'ingénieurs

HEI · ISEN · ISA

CIR²

Le Langage C++

Une introduction rapide avant d'aborder la POO

Pascal MOSBAH

Object Oriented Programming : The C++ language

- Première version début des années 80.
- Proposée par Bjarne Stroustrup (AT&T Lab), https://en.wikipedia.org/wiki/Bjarne_Stroustrup
- Présentée comme une extension du langage C, avec:
  - Des améliorations syntaxiques : nouveaux types, opérateurs, mots clés… (garde une bonne compatibilité avec le langage C)
  - Introduction de la Programmation Orienté Objet
  - Multi Paradigme : A la fois un langage procédural et un langage orientée objet.
- Le nom original était "C with classes".
- Un des langages les plus populaires et des plus performants.

Object Oriented Programming : The C++ language

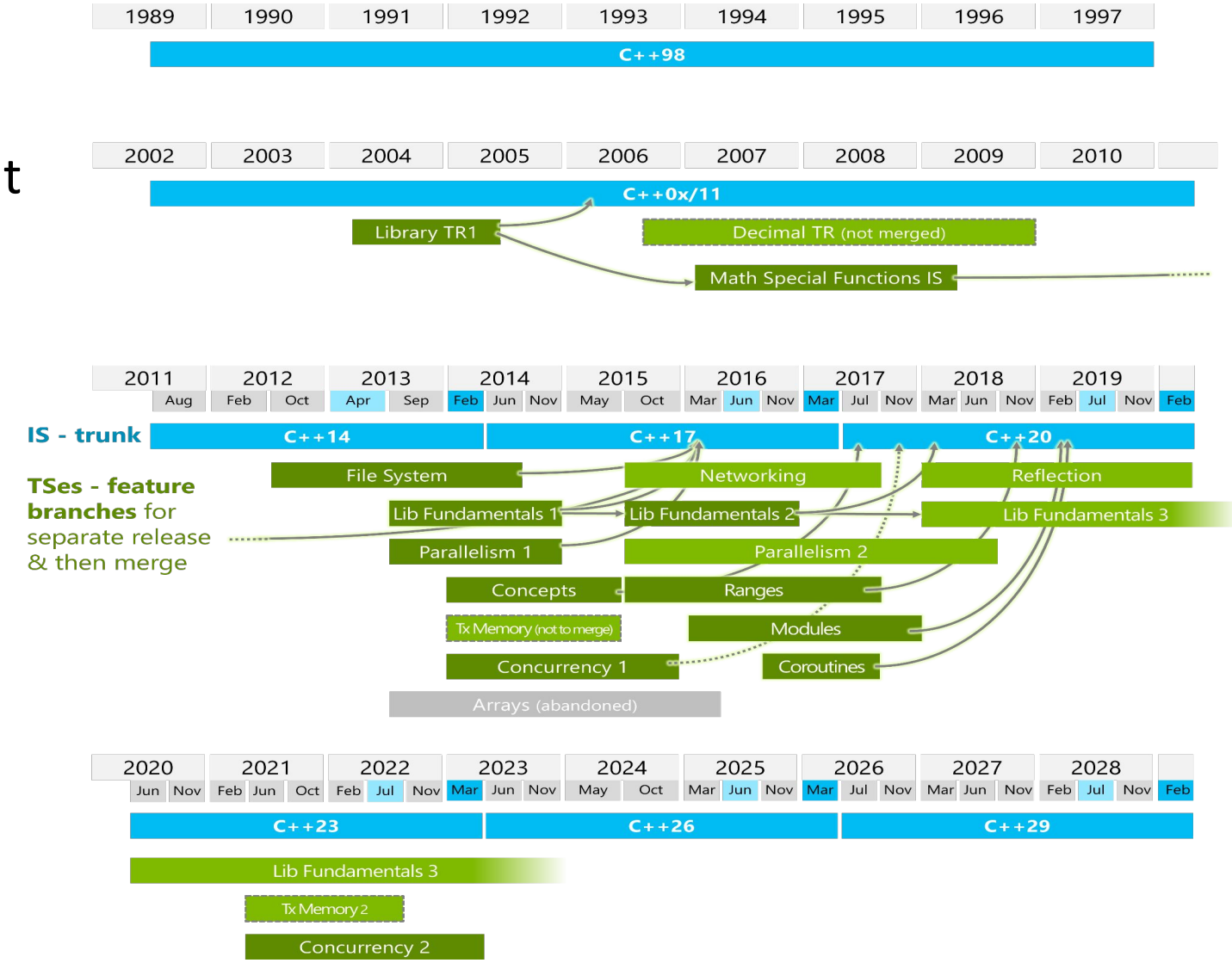
Particularités du Langage C++ :

C++ est un langage de programmation puissant et polyvalent qui peut être utilisé pour créer une variété d'applications. Il offre de nombreux avantages, notamment :

- Performances élevées : C++ est un langage de programmation compilé, ce qui signifie qu'il est converti en code machine (optimisé) avant d'être exécuté. Cela permet entre-autres d'obtenir des performances élevées car le code machine est directement exécuté par le processeur.
- Génère des programmes avec des consommations mémoire et énergétique les plus faibles de l'ensemble des langages de programmation pour une même tâche à exécuter (voir plus loin).
- Portabilité : Le code C++ repose sur des standards (actuellement C++23) et peut être compilé pour une variété de plates-formes, ce qui le rend portable. Cela signifie qu'un même code source peut être compilé et utilisé sur des ordinateurs de bureau, ordinateurs portables, smartphones, tablettes...
- Flexibilité : C++ offre une grande flexibilité et offre une palette d'utilisation assez large ce qui permet aux développeurs de créer des programmes personnalisés pour répondre à leurs besoins spécifiques.
- En revanche, le standard C++ se concentre principalement sur les éléments de langage. Par exemple, il ne comporte pas de bibliothèque graphique ou de système de programmation de fenêtrage (des bibliothèques externes existent pour ces utilisations) … Pour autant, il a été un des premiers à intégrer la programmation générique.

C++ : Les révisions et les standards

- Le C++ est régulièrement révisé.
- De nouvelles implémentations sont toujours en cours.



La compilation

- Extensions, le plus souvent:

  .hpp, .h++, .H ou sans extension (comme par exemple <iostream> pour les « classes » utilisées pour les entrées-sorties) pour les fichiers d'entêtes

  .cpp, .c++, .C pour le code source

- La fonction main() doit retourner un int

- Compilateur : cl, clang, g++, icl
  L'option définissant le standard -std=c++xx doit éventuellement être ajoutée pour utiliser un standard récent :
  g++ -std=c++20 filename.cpp avec gcc, ou set(CMAKE_CXX_STANDARD 20) dans CMakeLists.txt

- Les versions récentes des compilateurs implémentent la version standard ISO C++20

JUNIA Grande école d'ingénieurs

HEI · ISEN · ISA

Le Langage C++
Quelques nouveautés syntaxiques (1)

Les variables

Quelques nouveautés et améliorations syntaxiques

- Un vrai type bool
- Les mots clés auto et decltype
- Le mot clé const
- Les références
- La surcharge des fonctions
- Valeurs d'arguments par défaut
- Manipulation des flux avec << et >>
- Désallocation mémoire avec new et delete
- Les *Namespaces* et l'opérateur de portée ::

Le type bool

- C < 99
  - Pas de support des booléens
- C > 99
  - Définis dans stdbool.h
  - Basé sur un type int et des macros true et false (true si différent de zéro)
- C++
  - un vrai type booléen avec les mots clés bool, true et false
  - Conversion automatique à partir des autres types
  - Généralement stocké sur un octet (en fonction des systèmes)

Le type bool

```
bool b;
int i;

b = true;
i = b;          // i = 1

b = false;
i = b;          // i = 0

i = 3;
b = i;          // b = true

i = -2;
b = i;          // b = true

i = 0;
b = i;          // b = false
```

La declaration de variables

- Les variables locales peuvent être déclarées n'importe où dans un bloc.
- Un bloc est un ensemble d'instructions entouré de «{» et «}».
- Les blocs anonymes (non liés à une fonction ou une boucle) sont syntaxiquement corrects
- La portée (et la vie) de la variable s'arrête à la fin du bloc.

La déclaration et portée des variables : exemples

1)

```
#include <cstdio>       // for printf

int main(){
        int start = 4;
        int end = 10;
        for(int i = start; i < end; i++){
                int k = i/2;
                printf("%d\n", k);
        }
        int j = k;
        printf("%d\n", j);
        return EXIT_SUCCESS;
}
```

La variable k est utilisée en dehors de sa portée (compilation error : 'k' was not declared in this scope)

2)

```
#include <cstdlib>
#include <cstdio>
#include <ctime>
//C++17 tiré de https://www.tutorialspoint.com/cplusplus17-if-statement-with-initializer

int main() {
  srand(time(NULL));
  // C++17 if statement with initializer
  if (int random_num = rand(); random_num % 2 == 0) {
     printf("%i is an even number\n", random_num);
  } else {
      printf("%i is an odd number\n", random_num);
  }
  return EXIT_SUCCESS;
}
```

La portée de la variable random_num est limitée à l'opérateur conditionnel « if ».

3)

```
#include <cstdio>       // for printf
int main(){
   int start = 4;
   {
      int start = 8;
   }
   printf("%d\n", start);
   return 0;
}
```

start == 4

Les mots clés const, constexpr, consteval

- Le mot clé const a plusieurs utilisations. L'une d'entre elles est la déclaration de constantes (ou variables protégées contre la modification) .
- La valeur d'une variable constante est donnée à la déclaration de cette variable.
- Les tentatives de modification génèrent des erreurs de compilation.
- Exemple :

```
const int i = 1;   // OK
i++;    /* compilation error
         *error: increment of read-only variable 'i' */
et aussi:
    const int n = 10;
    int tab[n];
```

Les mots clés const, constexpr, consteval

- Le mot clé **constexpr** (c++11, c++14) étend l'utilisation de **const** aux fonctions et aux constructeurs. Il « peut » permettre l'évaluation d'expressions lors de la compilation ce qui rend le cas échéant le programme plus performant à l'exécution.
- La valeur de cette variable est indiquée lors de sa déclaration.
- Les tentatives de modification génèrent également des erreurs de compilation.
- Un contre-exemple :

```
constexpr float z = exp(5.3); (n'est pas évaluée lors
                    de la compilation dans ce cas)
```

Les mots clés const, constexpr, consteval

- Le mot clé **consteval** (c++20) rend impératif l'évaluation d'expressions lors de la compilation.

```
#include <cstdio>

consteval          // ou constexpr
long long unsigned factorial(long long unsigned n)
{
        return n <= 1 ? 1 : (n * factorial(n - 1));
}

int main(){

        long long unsigned total;
        for (int iter_total = 0; iter_total < 100000; ++iter_total){
                total = 0;
                for (int iter = 0; iter < 1000; ++iter){
                        constexpr long long unsigned n = 60;
                        total += factorial(n);
                }
        }
        printf("Factorial total %llu\n",total);
        return 0;
}
```

```
Message d'erreur quand l'évaluation à la compilation
n'est pas possible :

consteval float z = exp(5.3);
error: a variable cannot be declared 'consteval'
        consteval auto val = exp(5.3);
```

Boucle de calcul : 100000 * 1000 évaluations
de la fonction « factorial ».

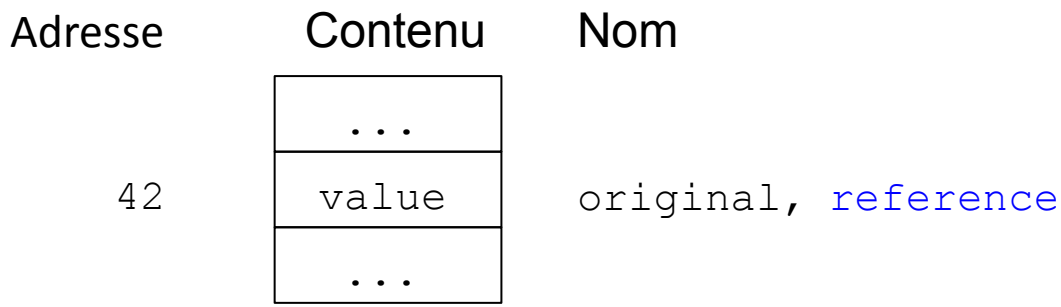
Avec le mot clé « consteval » (ou la fonction « factorial », avec la valeur constante n = 60, est évaluée lors de la compilation.

Avec le mot clé « consteval », l'exécution du programme est pratiquement instanée, alors qu'il est de l'ordre de 3 sec sans la directive « consteval ».

Les références

- Une référence peut être vu comme un alias (d'une autre variable).
- Le symbole **&** est utilisé (placé entre le type et le nom de la référence).
- Une référence est initialisée à sa déclaration (référence à quelle variable ?).
- La variable originale et sa référence pointent la même zone mémoire.
- Une réelle amélioration syntaxique pour réduire l'usage des pointeurs.

```
variable_type original = value;
variable_type& reference = original;
```



Les références : exemples

Ecriture simplifiée en utilisant des références

```
#include <stdio.h>
int main(){
    int i = 1;
    int *ptr = &i; // & -> adresse mémoire
    int &ref = i; // & -> référence
    printf("i: %p, ref: %p, ptr: %p\n", (void *) &i, (void *) &ref, (void *) ptr);

    printf("i: %d, ref: %d, ptr: %d\n", i, ref, *ptr);
    i++;
    printf("i: %d, ref: %d, ptr: %d\n", i, ref, *ptr);
    ref++;
    printf("i: %d, ref: %d, ptr: %d\n", i, ref, *ptr);
    (*ptr)++;
    printf("i: %d, ref: %d, ptr: %d\n", i, ref, *ptr);
    printf("i: %d, ref: %d, ptr: %d\n", i, ref, (*ptr)++);
    return EXIT_SUCCESS;
}
```

i: 0x7fff3152acf4, ref: 0x7fff3152acf4, ptr: 0x7fff3152acf4

i: 1, ref: 1, ptr: 1

i: 2, ref: 2, ptr: 2

i: 3, ref: 3, ptr: 3

i: 4, ref: 4, ptr: 4

?

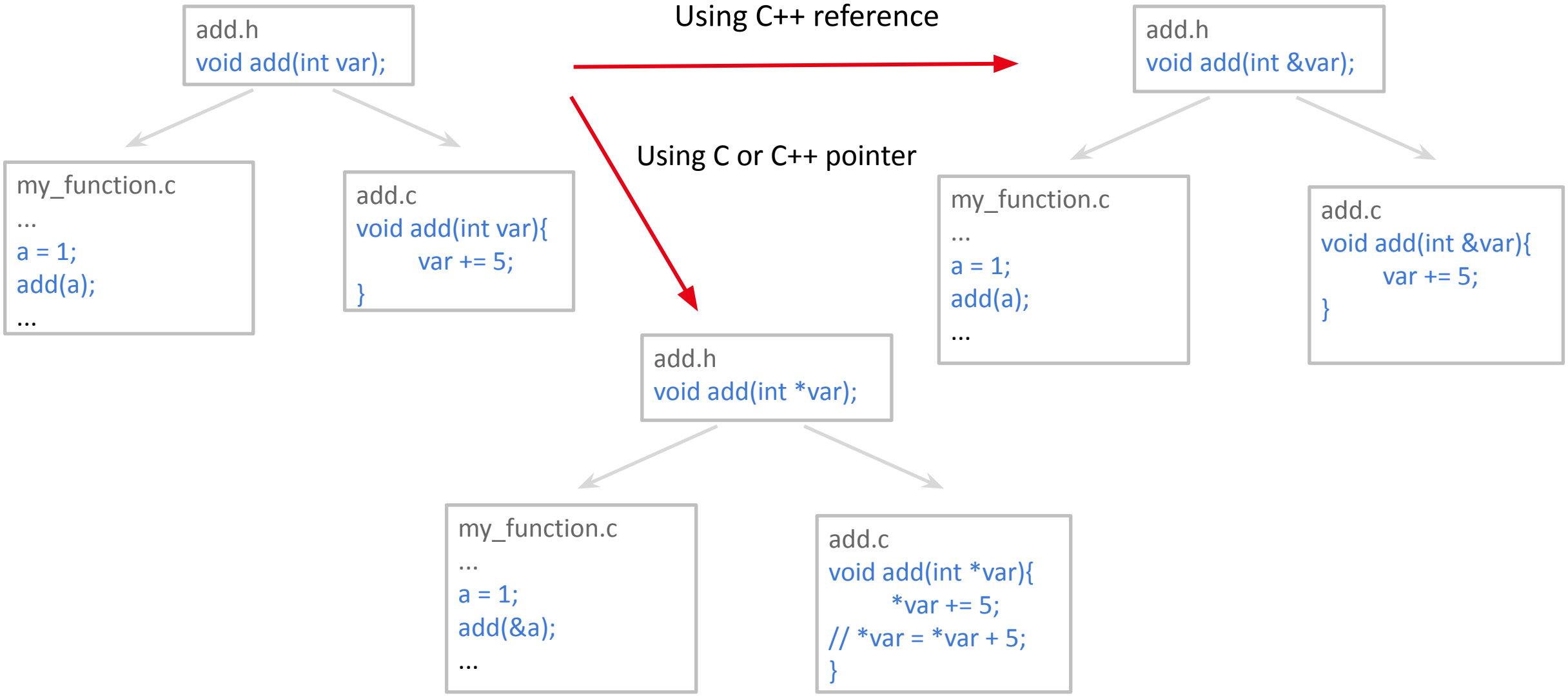
Passage d'arguments par référence

- Ainsi, en plus du passage par copie et du passage par adresse, le C++ permet de passer des paramètres par référence (en utilisant le symbole &) ;
- Un paramètre passé par référence peut modifier la variable originale (référencée) ;
- L'usage des références permet de lier les performances du passage par adresse au confort et la facilité d'écriture du passage par copie ;
- Placer le mot clé const devant le paramètre pour empêcher sa modification ;
- Il est aussi possible de retourner des variables par référence (attention).

```
// declaration
return_type function_name(arg1_type&, arg1_type&, ...);

// appel
function_name(arg1, arg2, ...); // aucun symbole lors de l'appel
```

Les références : exemples de passage de paramètres



Passage d'arguments par référence

Exemples passage d'arguments par référence

```
#include <iostream>
/*
int &plus_one(){
	int a = 5;
	return a; // warning: reference to local variable ‘a’ returned
}
*/
/*
int &plus_one(int a){
	return a; // warning: reference to local variable ‘a’ returned
}
*/
int &plus_one(int &a){
	return ++a;
}
/*
int &plus_one(int &a){
	return a++; // error: cannot bind non-const lvalue reference of type
‘int&’ to an rvalue of type ‘int’
}
*/
```

```
int main(){
	int val = 5;
	val = plus_one(val);
	std::cout<<val<<std::endl;
	plus_one(val) = 8;
	std::cout<<val<<std::endl;
	return EXIT_SUCCESS;
}
```

Valeur de val ?

Le mot clé auto

- Détection automatique du type de la variable (Depuis C++11)

```
auto i = 0;          // i is int
auto j = 2 + 3;      // j is int
auto d = 3.14;       // d is a double
auto f = 0.0f;       // f is a float
auto s = "string";   // s is a const char *
```

```
#include <cstdlib>
#include <cstdio>
#include <iostream>
auto plus_one(int a){
   return a * 5.;
}
//auto plus_one(auto a){
//    return a * 5.; //warning: use of 'auto' in parameter declaration (only available with '-std=c++20)'
//}
int main(){
        int a = 1;
        double b = plus_one(a);
   printf("%f\n", b);
   std::cout << b <<std::endl;
   return EXIT_SUCCESS;
}
```

JUNIA Grande école d'ingénieurs

HEI · ISEN · ISA

Le Langage C++
Quelques nouveautés syntaxiques (2)

Les fonctions

La surcharge de fonctions (Function Overloading)

- En C, chaque fonction doit avoir un nom différent.
- En C++:
  - Les fonctions peuvent avoir le même nom (dans le même espace) à conditions d'avoir des paramètres qui diffèrent en **types** ou en **nombre** (polymorphisme)
  - Le compilateur sélectionne la fonction adéquate en comparant la liste d'arguments passés avec la liste des paramètres de chaque fonction surchargée (et non suivant le paramètre de retour)

La surcharge de functions (Function Overloading)

- Exemple : Déclaration des fonctions - dans un fichier d'en-tête - overloaded_print.h

```
void print_params(const int &i1, const int &i2);

void print_params(const int &i, const float &f);

void print_params(const float &f);
```

```
int print_params(const int &i1, const int &i2);
```

Ambiguïté / Erreur de compilation
(différentiation par le type de retour de la fonction)

La surcharge de functions (Function Overloading)

Définition des fonctions - overloaded_print.cpp

```
void print_params(const int &i1, const int &i2){
   printf("Parameters: integer %d and integer %d\n", i1, i2);
}
void print_params(const int &i, const float &f){
   printf("Parameters: integer %d and float %f\n", i, f);
}
void print_params(const float &f){
   printf("Parameter: float %f\n", f);
}
```

Exemple d'appels – test.cpp

```
#include "overloaded_print.h"

int main(){
    int i1 = 1;
    int i2 = 2;
    float f = 3.0f;

    print_params(i1, i2);
    print_params(i1, f);
    print_params(f);

    return 0;
}
```

Fonctions: arguments par défaut

- Un argument par défaut : une valeur spécifique utilisée par défaut si la valeur d'un paramètre n'est pas fournie lors de l'appel.
- Les arguments par défaut sont obligatoirement placés en dernier dans la liste des paramètres de la fonction.
- Ils peuvent être spécifiés dans la déclaration de la fonction ou dans sa définition, mais pas dans les deux.

```
int increment(int i, int step = 1);
```
✔

```
int increment(int i = 0, int step = 1);
```
✔

```
int increment(int i = 0, int step);
```
✗

Mixer la surcharge et les arguments par défaut

- L'utilisation des arguments par défaut dans les fonctions surchargées peut générer des ambiguïtés.
- La règle est simple : toutes les versions induites par les arguments par défaut ainsi que les différentes surcharges de fonction ne doivent pas être en conflit.
- Exemple.

```
void increment(int i, int step = 1){}
void increment(int i, float step = 1.f){}
void increment(float step, int i = 1){}

int main(){
    increment(1,1);  ✔
    increment(1,1.f);✔
    ~~increment(1);~~     ✗
    increment(1.f);  ✔
}
```

JUNIA Grande école d'ingénieurs

HEI · ISEN · ISA

Le Langage C++
Quelques nouveautés syntaxiques (3)

Les espaces de noms (Namespaces)

L'opérateur de résolution de portée ::

- Syntaxe :

```
scope::variable
```

- `Scope` peut être
  - Vide (variables globales)
  - Un espace de nom
  - Le nom d'une classe
  - …

Les espaces de noms : Namespaces

- Définissent un contexte
- Réduisent les problèmes de conflits de noms
- Similaires au *packages* dans d'autres langages
- Les mots clés namespace et using sont utilisés pour déclarer des espaces de noms ou de les utiliser dans un programme.

Déclaration

```
namespace mynamespace{
   /* identifiers here */

}
```

Utilisation

```
mynamespace::identifier
or
using mynamespace::identifier
```

Utilisation du contexte

```
using namespace mynamespace;
```

Namespaces : Exemple

```
namespace f2{
   int factor = 2;

   int mult(int i){ return i*factor; }
}


namespace f4{
   float factor = 4.0;

   float mult(int i){ return i*factor; }
}
```

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


int main(){
   int i = 3;

   i = f2::mult(i);
   cout << "i = " << i << endl;

   i = f4::mult(i);
   cout << "i = " << i << endl;

   i = f2::factor;
   cout << "i = " << i << endl;
}
```

JUNIA Grande école d'ingénieurs

HEI · ISEN · ISA

Le Langage C++
Quelques nouveautés syntaxiques (4)

Les entrées sorties standards

Entrées/Sorties standards

- Les entrées/sorties standards du C peuvent toujours être utilisées.

En C

```
#include <stdio.h>
stdin, stdout, stderr
printf, scanf, …
```

En C++

```
#include <iostream>
std::cin, std::cout, std::cerr,
les opérateurs << et >>
```

Exemple 1

C

```c
#include <stdio.h>

int main(){
   int i = 1;
   printf("i is %d\n", i);
   return 0;
}
```

C++

```cpp
#include <iostream>
int main(){
 int i = 1;
 std::cout << "i is " << i <<
std::endl;
 return 0;
} (1)
```

```cpp
#include <iostream>
using namespace std;
int main(){
 int i = 1;
 cout << "i is " << i <<
endl;
 return 0;
} (2)
```

```cpp
#include <iostream>
using std::cout;
using std::endl;
int main(){
 int i = 1;
 cout << "i is " << i <<
endl;
 return 0;
} (3)
```

Exemple 2

C

```c
#include <stdio.h>

int main(){
 int i = 1;
 float f = 1.f;
 scanf("%d\n", &i);
 scanf("%f\n", &f);
 return 0;
}
```

C++

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

int main(){
 int i = 1;
 float f = 1.f;
 cin >> i; //pas de caractère de conversion à définir (polymorphisme)
 cin >> f;
 return 0;
}
```

Les fichiers

- `#include <fstream>`
- Types prédéfinis : ftream, ifstream, ofstream, …
- Des modes d'accès (qui peuvent être combinés avec |):
  `ios::in`
  `ios::out`
  `ios::app`
  `ios::trunc`
  `ios::binary`
- Des méthodes pour manipuler les fichier :
  `Open(), close(), is_open(), eof(), read(), write(), getline(), ...`
- Les flux fichiers sont compatibles avec les opérateurs << et >>.

Les fichiers : écriture dans un fichier

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

int main(){
    ofstream file; // or fstream file; ofstream, fstream are classes
    char filename[] = "example.txt";
    file.open(filename); // or file.open(filename, ios::out );

    if(file.is_open()){
        file << "First line" << endl;
        file << "Second line" << "\n";
        file.close();
    } else {
       cout << "The file " << filename <<" cannot be opened"<<endl;
    }
    return 0;
}
```

Les fichiers : lecture à partir d'un fichier

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;
int main(){
	char buffer[100];
	string cpp_buffer; //string is a c++ class
	ifstream file;
	file.open("example.txt");
	if(file.is_open()){
		while(!file.eof()){
			file.getline(buffer, 100);
		  //	getline(file, cpp_buffer); utilisant la classe string
			cout << buffer << endl;
		  //	cout << cpp_buffer << endl; utilisant la classe string
		}
	}
	file.close();
	return 0;
}
```

JUNIA Grande école d'ingénieurs

HEI · ISEN · ISA

Le Langage C++
Quelques nouveautés syntaxiques (5)

La mémoire

(Dés)Allocation de la mémoire

En C

```
#include <stdlib.h>
malloc(), calloc(), realloc(), free()
```

En C++

Rien à inclure

Les operateurs new et delete

Eviter de mixer les fonctions et les opérateurs C/C++ notamment parce que les fonctions malloc et free comme nous le verrons plus loin n'appellent respectivement pas les constructeurs et destructeurs des classes.

Exemples

```
auto i = new int;          /* allocates 1 int, left uninitialized */
auto j = new int();        /* equivalent syntax */
auto k = new int(3);       /* allocates 1 int, and initializes it to 3 */
auto f1 = new float[2];    /* allocates 2 floats, left uninitialized */
delete i, j, k;            /* free i, j ,k */
delete[] f1;               /* free f1, f1 is a array ([]) */
```

La classe « String »

Tout comme en C, en C++ une chaîne de caractères est une séquence de caractères. Elle est représentée par un objet de la classe std::string de la bibliothèque standard C++ et est un type de données abstrait qui fournit une interface simple pour manipuler ces séquences de caractères de taille arbitraire (la mémoire consacrée au stockage des caractères est prise en charge automatiquement par la classe « string ».

Exemples :

```
    // Initialisation-copie
std::string chr0 = "Hello world !"; // initialisation avec copie ou std::string chr = {Hello world}";
std::string chr1("Hello world !");  // initialisation directe ou std::string chr{"Hello world"};
std::string chr2(5, '0');          // chr contient "00000"
auto chr3 = chr2;                   // initialise et copie chr dans chr1
auto chr4(chr3);                   // initialisation directe

// Manipulations
std::string chr5{"Hello "};
chr5 += "world"; //Concaténation
std::cout << chr5[0]; // Les caractères d 'une chaîne C++ sont accessibles par l'opérateur d'indexation []

// Comparaison
if (chr5 == "Hello world") std::cout << "Chaines égales" << std::endl;
```

La classe « String »

Exemples :

```
std::cout << chr5 << std::endl; // Ecriture d'une chaîne
std::string nom;
std::cin >> nom; // Lecture d'une chaîne

// Quelques méthodes associées:
std::cout << chr5.size() << std::endl; // nombre de caractères de chr5
chr5 += " world"; //Concaténation
std::cout << chr5 << std::endl; //Hello world world

size_t found = chr5.find("world"); // Recherche de l'occurrence d'une chaîne dans une autre
if (found != std::string::npos) std::cout << "First occurrence is " << found << std::endl;

char arr[] = "world";
found = chr5.find(arr, found + 1);
if (found != std::string::npos)
{
    std::cout << "Prochaine occurrence " << found << std::endl;
}
```

JUNIA Grande école d'ingénieurs

HEI · ISEN · ISA

CIR²

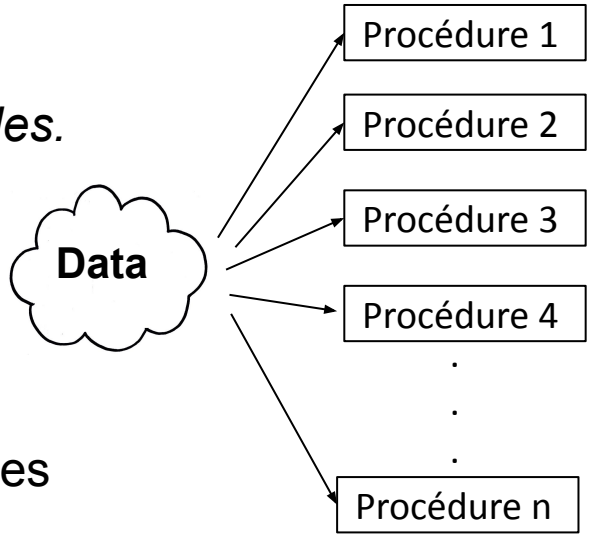
La programmation orientée objet (introduction)

Pascal MOSBAH

Programmation Orientée Objet : Généralités

Language C & Programmation Impérative

Les programmes sont décomposés en petits programmes, fonctions ou modules. Chaque module traite d'une tâche spécifique qui peut être testé et maintenu indépendamment sur un jeu de données.



Généralement, cette méthode consiste à séparer les structures de données modélisant l'information, des fonctions, modules … qui viennent traiter le comportement lié à ces structures de données. Les structures de données et les opérations sur ces structures sont donc le plus souvent séparés.

Disposant d'un paradigme de programmation simple, efficace, associé à des langages de programmation comme le langage C pourquoi en changer voire le faire évoluer (d'ailleurs, le langage C est utilisé dans de très nombreux projets dont un des plus emblématiques : le noyau Linux) ?

La POO peut proposer quelques solutions pour améliorer la modularité, la réutilisabilité, la sécurité, et la maintenance des codes.

Programmation Orientée Objet : Généralités

Par exemple, quelques défaillances courantes qui peuvent être éliminées par l'utilisation de la POO:

- Des données qui ne sont pas correctement initialisées ;
- Avec la séparation des données et des traitements, il y un risque de ré-écriture des fonctions et/ou que la fonction ne soit plus en adéquation avec la donnée à traiter ;
- L'« encapsulation » permet une relative protection des données qui évite que n'importe qu'elle fonction puisse changer la valeur d'une donnée en compromettant le comportement global du programme.

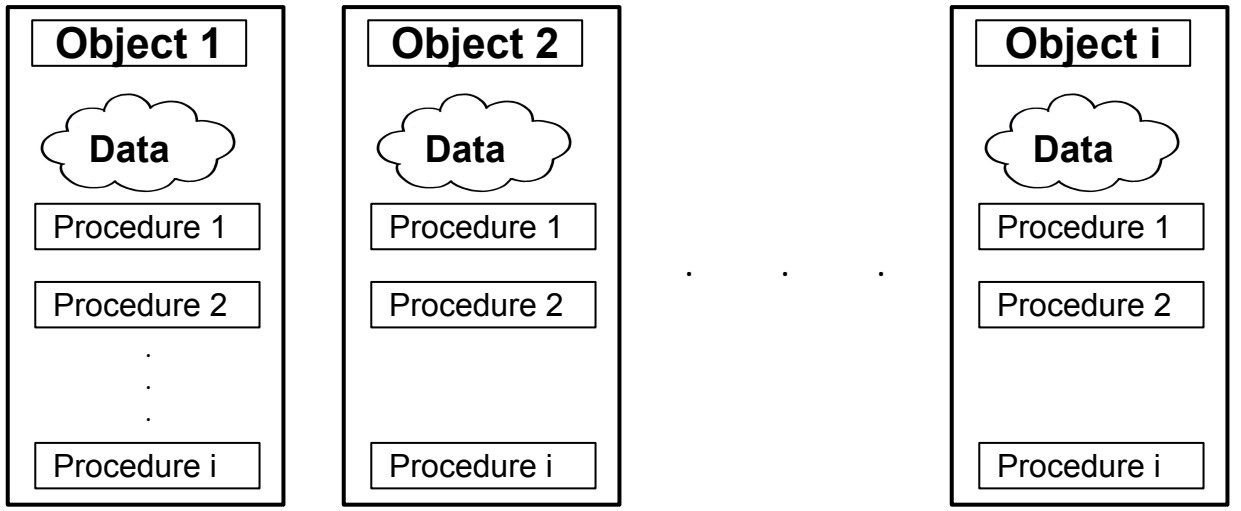
Programmation Orientée Objet : Généralités

1) https://en.wikipedia.org/wiki/Object-oriented_programming

La programmation orientée objet (POO) est un paradigme de programmation qui consiste à modéliser des objets représentant un concept, une idée ou toute entité du monde physique à l'aide de classes (structures). Une classe est une description d'un objet. Elle définit ses propriétés et ses méthodes. L'idée derrière la POO est de définir ces objets et leurs interactions.

La conception d'un programme se fait suivant les données et non par leurs traitements. Les procédures/fonctions existent toujours, mais elles sont attachées aux données. Dans l'idéal, les données sont encapsulées dans des entités appelées objets qui ne sont pas directement accessibles. Elles sont alors protégées contre toute manipulation « hasardeuse ». L'idée est d'exposer ces données aux travers de méthodes et de cacher le fonctionnement interne de ces entités.

Object-Oriented Programming (OOP)



Programmation Orientée Objet : Généralités

Les fondements de la POO sont généralement représentés par :

- Les **objets**, instances de **classes**, qui peuvent être vus comme une association de données (attributs) et procédures (méthodes).
  - Objet = attributs + méthodes
- **l'Encapsulation** : Les données et les comportements d'un objet sont encapsulés, ce qui signifie qu'ils sont (généralement) cachés des autres objets. Cela permet de protéger les données et de garantir que les objets ne peuvent être utilisés que de manière appropriée.
- **l'Héritage** : Un objet peut hériter des propriétés et des comportements d'un autre objet, appelé sa classe parent. Cela permet de réutiliser le code et de créer des objets plus complexes à partir d'objets plus simples.
- **le Polymorphisme** : Un objet peut avoir plusieurs méthodes avec le même nom, mais avec des signatures différentes. Cela permet de créer des objets qui peuvent être utilisés de manières différentes.
- **L'abstraction** : Quelquefois définie en surplus des critères ci-dessus. Il semble toutefois pouvoir découler des notions précédentes.

Quelques exemples de la mise en oeuvre de la POO (« dénichés » en partie par Bard-Google) :

- Une voiture est un objet qui peut avoir des données telles que le type, la marque, le modèle, le type de motorisation, la couleur. Elle peut avoir des comportements tels que conduire, tourner et freiner.
- Un compte bancaire est un objet qui peut avoir des données telles que le type de compte, le numéro de compte, le solde et le titulaire. Il peut avoir des comportements tels que déposer de l'argent, retirer de l'argent et consulter le solde.
- Un jeu vidéo est un ensemble d'objets qui interagissent entre eux. Les objets peuvent représenter des personnages, des objets, des bâtiments, une scène...

La POO est un paradigme de programmation puissant qui est utilisé dans de nombreux langages de programmation, (Java, C++, Python, JavaScript…). Elle facilite généralement la gestion de la complexité. Sa maîtrise demande généralement d'y consacrer du temps.

JUNIA Grande école d'ingénieurs

HEI · ISEN · ISA

CIR²

Programmation orientée objet en C++

Les classes

Pascal MOSBAH & Nacim IHADDADENE

Programmation orientée objet en C++ : les classes

Rappels :
Les fondements de la POO sont :

- Les **objets** qui peuvent être vus comme une association de données (attributs) et procédures (méthodes).
  - Objet = attributs + méthodes

- **l'Encapsulation** : Les données et les comportements d'un objet sont encapsulés, ce qui signifie qu'ils sont (généralement) cachés des autres objets. Cela permet de protéger les données et de garantir que les objets ne peuvent être utilisés que de manière appropriée.

- **l'Héritage** : Un objet peut hériter des propriétés et des comportements d'un autre objet, appelé sa classe parent. Cela permet de réutiliser le code et de créer des objets plus complexes à partir d'objets plus simples.

- **le Polymorphisme** : Un objet peut avoir plusieurs méthodes avec le même nom, mais avec des signatures différentes. Cela permet de créer des objets qui peuvent être utilisés de manières différentes.

Définition d'une classe en C++

Définition :

- Le mot clé class
- Les délimiteurs { } et un séparateur ; à la fin
- Une liste de membres
- Les membres d'une classe peuvent être des variables ou des fonctions, représentant les données et les opérations
- On appelle ces variables : attributs
- On appelle ces fonctions : méthodes
- Les membres (attributs et méthodes) peuvent être : private, public, ou protected (les membres d'une structure sont tous publics)

Définition d’une classe en C++

Trois types d’accesseurs sont distingués: private, public, ou protected :

- Les membres privées ne sont accessibles que pour la classe (“-” en UML).
- Les membres publics sont accessibles pour toute autre classe ou fonction (“+” en UML, ex. dans le programme principal via la fonction main())
- Les membres protégés sont accessibles pour la classe et ses classes enfants (“#” en UML, à détailler dans le chapitre « Héritage »)

To AI -> Please give me a definition of UML Diagram in french :

Un **diagramme UML (Unified Modeling Language) est une représentation graphique** normalisée utilisée en génie logiciel pour visualiser, spécifier, construire et documenter les différents aspects d'un système logiciel. Les **diagrammes UML permettent de modéliser la structure, le comportement et les interactions des éléments d'un système**, facilitant ainsi la communication et la compréhension entre les différentes parties prenantes d'un projet.

Le concept de classe

- Une classe est un ensemble d'objets qui partagent une même sémantique, des mêmes attributs et des opérations communes.
- Exemple classique :

  Toutes les fractions possède :
  un numérateur ("num_") et un dénominateur ("den_").

- Le mot clé private est la clé d'un mécanisme appelé : encapsulation
- L'encapsulation sépare :

  La structure interne de l'objet - dee l'interface (la manière de l'utiliser)

```
class Fraction
{
private:
  int num_ = 0, den_ = 1;

public:
  Fraction();
  Fraction(const int &num, const int &den);
  void set_num(const int &num);
  void set_den(const int &den);
  int get_num() const { return num_; };
  int get_den() const { return den_; };
};
```

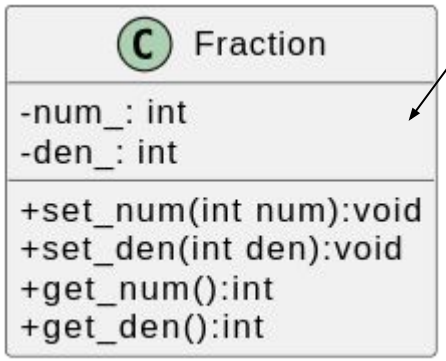


Diagramme UML

Objectifs de l'encapsulation

- Abstraction : l'utilisateur final n'a pas de vue sur l'implémentation des classes et leur structure interne. La manipulation se fait via les méthodes (interfaces).

- Protéger l'intégrité des objets: les méthodes peuvent vérifier que les différentes opérations effectuées préservent l'intégrité des objets.

  Exemple : le dénominateur ne doit pas être nul dans une fraction. Cette vérification se fera via la méthode set_den(…) (voir ci-dessous).

- Robustesse du code : la classe peut être modifiée sans que son utilisateur final soit impacté.

Définition des méthodes

- Les méthodes sont généralement écrites dans des fichiers sources (.cpp par exemple)
- On les fait précéder par le nom de la classe et l'opérateur de résolution de portée ::
  selon la syntaxe

```
return_type ClassName::methodName(type1 arg1, ...)
{

/* function body here */

}
```

Exemple d'une classe

fraction.hpp

```
#prama once

class Fraction {
        private:

        int num;
        int den;

        public:
        int getNum();
        int getDen();

        void setNum(const int&);
        bool setDen(const int&);
};
```

fraction.cpp

```
#include <iostream>
#include "fraction.hpp"
using namespace std

int Fraction::getNum(){
   return num;
}

int Fraction::getDen(){
   return den;
}

void Fraction::setNum(const int &newnum){
   num = newNum;
}

bool Fraction::setDen(const int &newden){
  if(newden == 0)
  {
     cerr << "den is 0!" << endl;
     return false;
  }
  else
  {
     den = newden;
     return true;
  }
}
```

Manipulation des instances

- Les **instances** sont des variables de type « class ».

- Allocation statique:

```
Fraction f;
```

- Allocation dynamique:

```
Fraction* pf = new Fraction();
ou
auto pf = new Fraction();
```

Analogie avec les types de base

```
int i;
```

```
auto pi = new int();
```

L'accès aux membres de la classe

- La syntaxe est similaire à celle des structures :
- Pour les instances : instance.membre

```
Fraction f;
f.setNum(3);
```

- Pour les pointeurs vers les instances : pinstance->membre

```
Fraction* pf1 = &f;
float num = pf1->getNum();

Fraction* pf2 = new Fraction();
pf2->setNum(num);
```

Les méthodes const

- 3ème usage du mot clé const
- Permet de spécifier des méthodes constantes : elles ne peuvent pas modifier les attributs de la classe (ils sont en « lecture seule »).
- Eviter au développeur de changer les attributs accidentellement.
- La syntaxe :

```
class MyClass
{
    ...
    returnType &methodName(...) const
    ...
}
```

- Attention :
  Le mot clé const doit être présent dans la déclaration et la définition.

Les méthodes const

```
Fraction f;
f.setNum(1);
cout << f.getNum();
```

```
class Fraction
{
    ...
    int getNum() const
    {
        num++;
        return num;
    }
    ...
};
```

Compiler output (g++)

```
fractionTest.cpp: In member function ‘float
Fraction::getNum() const’:
fractionTest.cpp:15:5: error: increment of data-member
‘Fraction::num’ in read-only structure
```

Le pointeur this (1/3)

- Il s'agit d'un pointeur passé automatiquement à toutes les méthodes d'une classe
- Il pointe l'adresse de l'instance ayant fait l'appel

- Exemple :

```
ThisDisplay* dp =
    new ThisDisplay();

cout << dp << endl;
dp->printThis();
```

Standard output

```
0x8242008
0x8242008
```

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

class ThisDisplay
{
    public:
    void printThis()
    { cout << this << endl; };
};
```

Le pointeur this (2/3)

- Le pointeur this permet par exemple de supprimer les conflits des identifiants entre les paramètres et les variables locales :

```
void Fraction::setNum(const int &newnum){
      num = newnum;
}
```

num reçoit la valeur de newnum

```
void Fraction::setNum(const int &num){
      num = num;
}
```

Le paramètre num reçoit sa propre valeur

```
void Fraction::setNum(const int &num){
      this->num = num;
}
```

L'attribut num reçoit la valeur du paramètre num

Pour éviter ces conflits, il est également possible d'utiliser des conventions de nommage pour les attributs des classes comme : m_num, num_, _num …

Le pointeur this (3/3)

- Retourner la valeur du pointeur this dans les méthodes permet de réaliser des appels chaînés (method chaining / fluent interface)

  - Example :

```
Fraction* Fraction::setNum(const int &num){
        this->num = num;
        return this;
}
```

Permet d'écrire :

```
Fraction* f = new Fraction();
f->setNum(3)->setDen(2);
```

Définition et fonctionnement

Un **constructeur** est une méthode particulière d'une classe. Il permet d'instancier et donc de créer un objet de sa propre classe.

Un **constructeur** porte toujours le nom de la classe et ne retourne rien.

Le constructeur

Le **constructeur** est appelé au moment de l'instanciation comme sur l'exemple suivant :

```
Fraction frac;
```

Ici, c'est le constructeur dit par défaut qui est appelé puisqu'il n'a pas de paramètre, et il est possible de le redéfinir et de développer de nouveaux constructeurs.

Définition et fonctionnement.

Un constructeur **par défaut** est généré automatiquement **si** le développeur n'en crée pas lui même.

Exemple : 3 constructeurs dans une seule classe

Fraction.hpp

```
class Fraction
{
private:
    int num_; // L'initialisation peut également
s'effectuer ici avec « int num = 0; int den = 1 »
    int den_;

public:
    Fraction();
    Fraction(int);
    Fraction(int, int);
};
```

Fraction.cpp

```
#include "Fraction.h"

Fraction::Fraction(){
 this->num_ = 0;
 this->den_ = 1;
}
Fraction::Fraction(const int &num){
 this->num_ = num;
 this->den_ = 1;
}
Fraction::Fraction(const int &num , const int &den ){
 this->num_ = num;
 this->den_ = den;
}
```

Le constructeur : Liste d'initialisateurs de membres

Initialise les membres de classe avant l'exécution du corps du constructeur :

```
#include "Fraction.hpp"
Fraction::Fraction(const int &num){
 this->num = num;
 this->den = 1;
}
Fraction::Fraction(const int &num , const int &den){
 this->num = num;
 this->den = den;
}
```

```
Fraction::Fraction(const int &num): num_(num), den_(1) {}
```

```
Fraction::Fraction(const int &num , const int &den): num_(num), den_(den) {}
```

Le constructeur : Liste d'initialisateurs de membres

Initialise les membres de classe avant l'exécution du corps du constructeur :

```
class Example{
  private:
  int &var_ref;
  const int var_const;
  public:
    Example(int &var_ref, const int &var_const) {
        var_ref = var_ref;
        var_const = var_const;
      }

    Example(int &var_ref, const int &var_const) : var_ref(var_ref), var_const(var_const){}
};
```

Ces variables, définies comme des références ou des constantes, doivent obligatoirement être initialisés avant l'exécution du corps du constructeur. C'est aussi le cas pour l'initialisation de classe n'ayant pas de constructeur par défaut.

Quelques exercices

Exercice 1 :

1) Ecrire un programme C++ qui affiche la table de multiplication d'un chiffre entre 1 et 9 saisi par l'utilisateur.

2) Ecrire un programme C++ qui affiche l'intégralité de la table de multiplication parfaitement alignée (en utilisant la librairie *<iomanip>*)

Exercice 2 :

1) Ecrire un programme C++ qui analyse un texte à partir d'un fichier en affichant le nombre de lignes, de mots et de lettres. Explorer l'utilisation de la classe «stringstream» pour effectuer cet exercice.

2) Modifier le programme pour qu'il affiche aussi le nombre d'occurrences de chaque lettre de l'alphabet (sans différencier les majuscules des minuscules).