Le langage C++

Junia

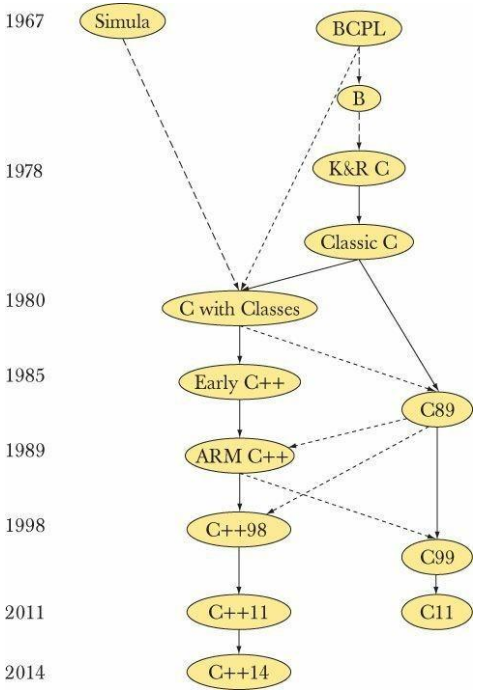
2022 - 2023

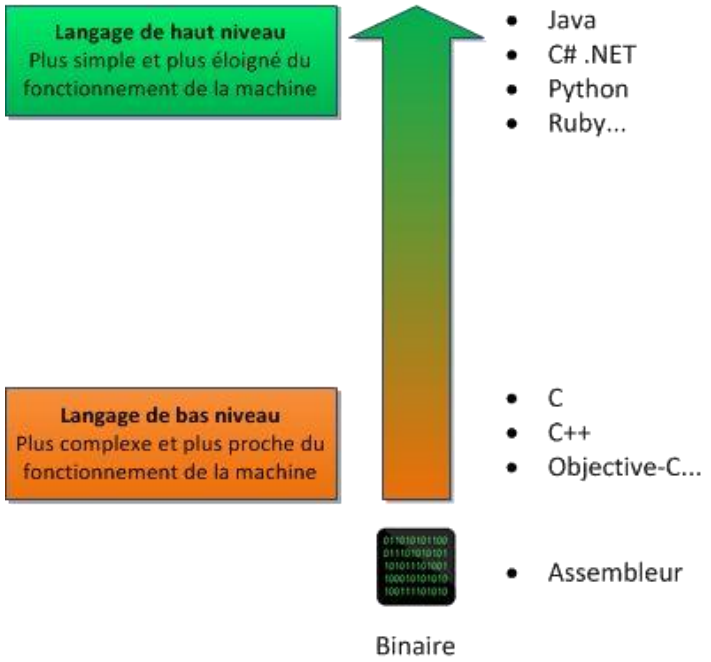
Intervenants

Hanaë Rateau & Yassine Benabbas

hanae.rateau@junia.com

Langage C++ - Introduction





Langage C++ - Outils

Utiliser 3 programmes séparément :

- ➔ éditeur de texte (nano/vim/emacs, Atom, VSCode);
- ➔ compilateur (gcc, g++) et outil make;
- ➔ débugger (gdb, ddd)

Utiliser un programme 3-en-1 / un environnement de développement intégré :

- ➔ Visual Studio
- ➔ CLion
- ➔ Qt Creator
- ➔ CodeBlocks

Pour ce cours

Compilateur:

g++ via Cygwin

- Téléchargez Cygwin https://x.cygwin.com/docs/ug/setup.html
- Pendant la sélection de paquets choisir gcc-g++ et make

Editeur:

Visual Studio Code + extensions pack C/C++

Langage C++ - Hello World!

```cpp
#include <iostream> // directive de préprocesseur qui permet d'inclure la bibliothèque iostream

using namespace std; // indique que nous utilisons l'espace de noms std

/*
Fonction principale "main"
Tous les programmes commencent par la fonction main et se termine à la fin de celle-ci
*/
int main()
{
    cout << "Hello world!" << endl; // flux sortant qui permet d'afficher un message
    return 0; // instruction qui termine la fonction main et donc le programme et indique que tout s'est bien passé
}
```

Compilation

"`gcc -o hello.exe hello.c`" se passe comme suit:

1. **Pre-processing:** le pré-processeur cpp rajoute tous les fichiers headers (`#include`) et les macros (`#define`). Le résultat est un fichier “hello.i”contenant le code “déplié”.

```
> cpp hello.c > hello.i
```

2. **Compilation**: Le compilateur, ici GCC compile le code préprocessé de l’étape d’avant en code assembleur. Ici l’option –S permet d’indicquer au compilateur de s’arrêter à l’étape d’assembleur et non objet. Le résultat est un fichier assembleur "`hello.s`"

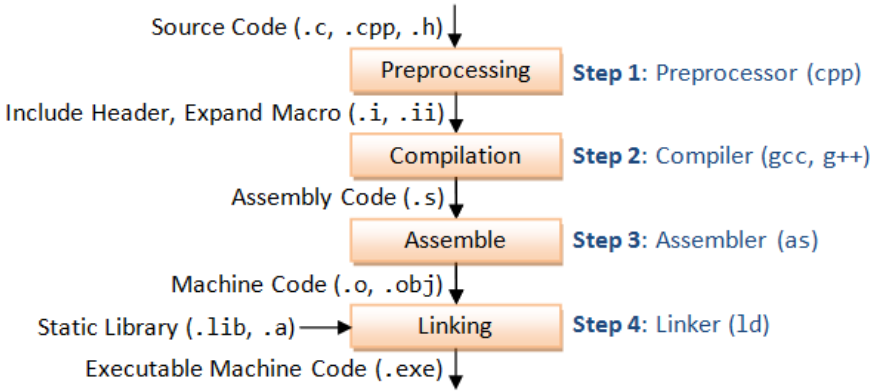
```
> gcc -S hello.i
```

3. **Assembleur**: L’assembleur converti le .i en code machine dans un fichier code objet .o

```
> as -o hello.o hello.s.
```

4. **Linker**: Le linker est la dernière étape dans laquelle le code objet est lié aux différentes libraires statiques utilisées pour formée un code exécutable. .exe.

```
> ld -o hello.exe hello.o ...libraries...
```



https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

Commandes Compilation G++

Compilation simple

```
g++ myfile.cpp
g++ mufile.cpp –o out.exe
g++ -Wall myfile.cpp // -Wall to output all warnings
```

Compilation multiple

```
g++ -o out.exe file1.cpp file2.cpp
```

Compilation et Linkage séparés (conseillé pour plusieurs fichier)

```
g++ -c file1.cpp
g++ -c file2.cpp
g++ -o out.exe file1.o file2.o
```

Makefile

Faciliter la compilation pour fichiers multiples.
Fichier texte formatté qui définit différentes commandes appelées “target”.

```
target: pre-req1 pre-req2 …
    command
```

On peut ensuite appeler ces commandes avec la commande make suivie de la target.

```
all: hello.exe

hello.exe: fichiers_multiples.o hello.o
    g++ -o hello.exe hello.o fichiers_multiples.o

hello.o: hello.cpp
    g++ -c hello.cpp

fichiers_multiples.o: fichiers_multiples.cpp
    g++ -c fichiers_multiples.cpp

clean:
    rm hello.o hello.exe
```

Programmation Procédurale / Programmation Objet

« **La programmation procédurale** est un paradigme de programmation, basé sur le concept de l' *appel de procédure* .

Les procédures, également appelées routines, sous-programmes ou fonctions, contiennent simplement une série d'étapes de calcul à exécuter. » - *Wikipédia*

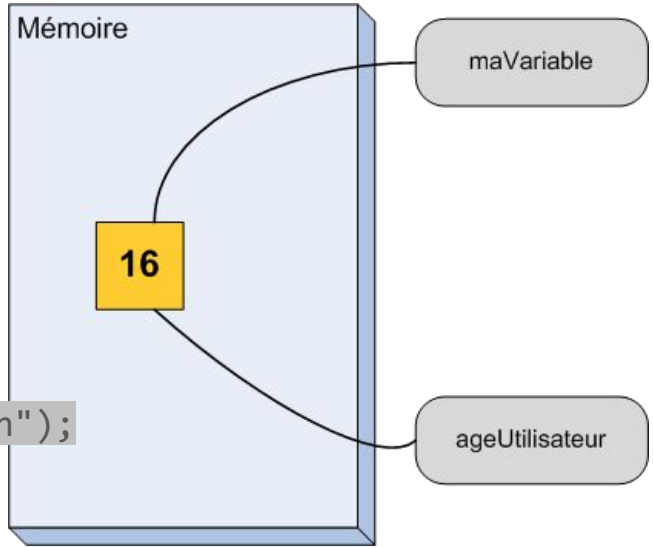
« **La programmation orientée objet** (**POO**) est un paradigme de programmation basé sur le concept d'« objets », qui peut contenir des données, sous la forme de champs, souvent appelés *attributs;* et du code, sous forme de procédures, souvent appelées *méthodes. » - Wikipédia*

Bases du langage

Langage C++ - Mémoire

Les tailles dependent de l'architecture du CPU

Bool	vrai (true) ou faux (false)	= 1 octet
Char	un caractère	= 1 octet
Int	un nombre entier	= 4 octets
unsigned int	un nombre entier positif ou nul	= 4 octets
Double	un nombre à virgule	= 8 octets
String	une chaîne de caractères	



`type nom (valeur);` pour allouer et initialiser une variable ou comme en C : `type nom = valeur`

déclaration à la chaine : `string prenom("Albert"), nom("Einstein");`

références/alias = une seule case mémoire mais deux étiquettes

```
int const ageUtilisateur(16);
int const& maVariable(ageUtilisateur);
```

Langage C++ - Opérateurs

- ➔ **arithmétiques** (+, -, /, *, %, ...)
- ➔ **comparaison** (==, >, <, >=, <=, !=, ...)
- ➔ **logiques** (&&, ||, !, ...)
- ➔ **assignation** (=, .=, +=, -=, *=, /=, ...)
- ➔ **incrémentation/décrémentation** (++, --)

fonctions mathématiques plus complexes avec l'en-tête <cmath>

➔Lecture/Écriture depuis le terminal

```
cin >> maVariable;
cin.ignore();
Stringm
maVariable("blablabla");
getline(cin, maVariable);
cout << “Hello World! ” << maVariable << endl;
```

concaténation de string avec l'opérateur +

Langage C++ - Structure de contrôle

```
if(condition) {...}
else if (condition) {...}
else {...}
```

```
switch(var)
{
     case 0:
       // instructions
       break;
     case 1:
     case 2:
       // instructions
       break;
     default:
       break;
}
```

➔ teste uniquement l'égalité;
➔ uniquement avec des nombres entiers;

```
do {...}
while(condition);
```

➔assure que la boucle sera lue au moins une fois;

```
while(condition) {...}
```

```
for (initialisation ; condition ; incrementation) {...}
```

➔ boucle for quand on connaît le nombre de répétitions;
➔ boucle while quand on ne sait pas combien de fois la boucle va être effectuée.

Langage C++ - Fonctions

Les fonctions permettent de mieux organiser son code. Elles peuvent prendre un ou plusieurs paramètres et retourner une valeur.

```
type nomFonction(type argument1, type argument 2, ...)
{
      // instructions effectuées par la fonction
      // return valeur; si type != void
}
```

Différentes manières de passer des paramètres à une fonction :

```
void maFonction(int a) {} // passage par valeur => copie
void maFonction(int& a) {} // passage par référence
         // => aucune copie n'est effectuée
void maFonction(int const& a) {}  //pour empêcher
         // la modification du paramètre
```

➔ monFichier.cpp le code source de la fpnction

```
#include "monFichier.h"
```

➔ monFichier.h contient uniquement le prototype de la fonction

```
#ifndef MONFICHIER_H_INCLUDED
#define MONFICHIER_H_INCLUDED

type nomFonction(type argument1, type argument 2, ...);

#endif // MONFICHIER_H_INCLUDED
```

pas d'espace de nom dans les fichiers d'en-tête
possibilité de définir des valeurs par défaut pour chaque argument qui devient alors facultatif
ils doivent obligatoirement se trouver à la fin

```
void maFonction(int param1, int param2 = 0,
int param2 = 0);
```

Langage C++ - Tableaux

➔ **Les tableaux statiques (à la compilation)**

```
type nom[taille];
```

// impérativement utiliser une constante comme taille du tableau.

```
nomDuTableau[numeroDeLaCase]
```

// pour accéder aux éléments, 0…(taille - 1)

```
void fonction(double tableau[], int tailleTableau)
```

//tableau passé en tant que pointeur vers son premier élément

Les chaînes de caractères sont en fait des tableaux ! On ne peut pas créer un tableau de références !

➔ **Les tableaux dynamiques (à l'execution)**

```
#include <vector>
vector<type> nom(taille, valeur);

monTableau.push_back(valeur) // pour ajouter un élément
monTableau.pop_back() // pour supprimer le dernier élément
monTableau.size() // pour connaître la taille

void fonction(vector<int> const& a) // évite une copie du tableau
```

Langage C++ - Tableaux

➔ **Les tableaux multi-dimensionnels**

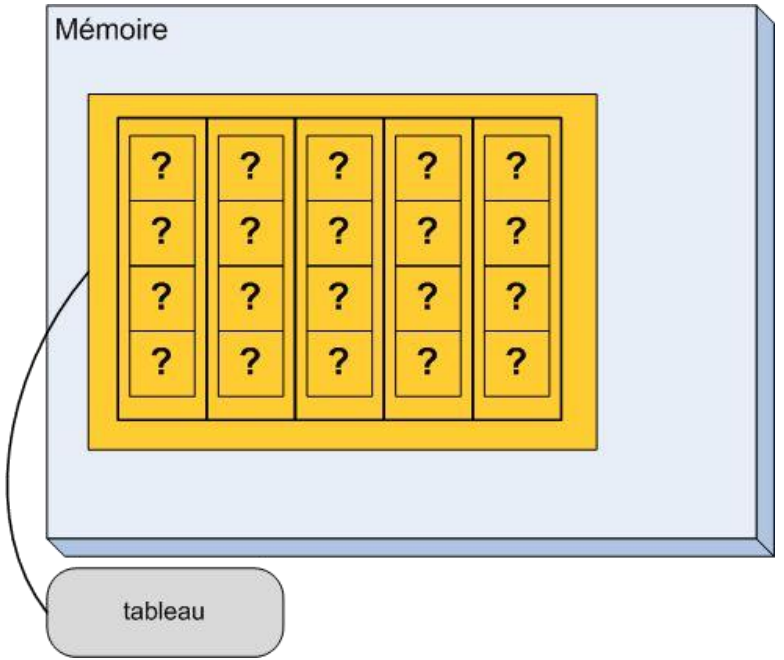
statique : type nomTableau[tailleX][tailleY]

dynamique :

```
vector<vector<int> > grille;
grille[2][3] = 9;
grille.push_back(vector<int>(5));
grille[0].push_back(8);
```

possibilité de créer des grilles tri-dimensionnelles, voire même plus :

```
double grilleExtreme[5][4][6][2][7];
```



Langage C++ - Lecture/modification des fichiers

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

using namespace std;

int main()
{
   ifstream fichier("data.txt");

   if(fichier)
   {
     // Lire caractère par caractère
     char c;
     fichier.get(c);
     // Lire mot par mot
     string mot;
     fichier >> mot;
     // Purge le buffer
     fichier.ignore();
     // Lire ligne par ligne
     string ligne;
     getline(fichier, ligne);
   }
   else
   {
      cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;
   }

   return 0;
```

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

using namespace std;

int main()
{
    string const nomFichier("data.txt");
    ofstream fichier(nomFichier.c_str(), ios::app);

    if(fichier)
    {
        fichier << "Hello World!" << endl;
    }
    else
    {
        cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;
    }

    return 0;
}
```

Langage C++ - Lire et modifier des fichiers

Récupérer la position (int) du curseur :

➔ `flux.tellg()` pour ifstream;

➔ `flux.tellp()` pour ofstream.

Déplacer le curseur :

➔ `flux.seekg(nombreCaracteres, position)` pour ifstream;

➔ `flux.seekp(nombreCaracteres, position)` pour ofstream.

3 positions possibles :

➔ début du fichier `ios::beg`

➔ fin du fichier `ios::end`

➔ position actuelle `ios::cur`

Possibilité de gérer manuellement l'ouverture et la fermeture des fichiers :

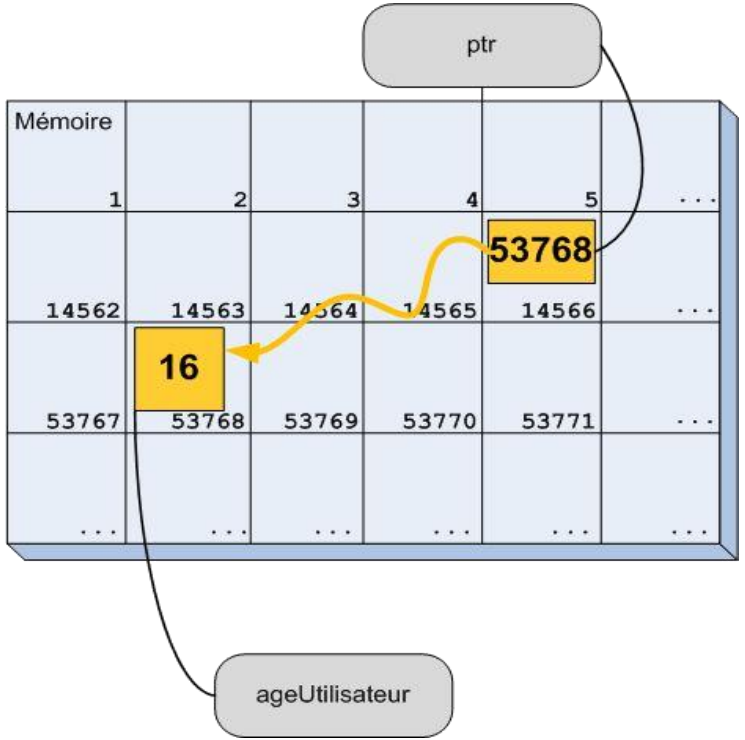
```
ofstream flux;
flux.open("data.txt");
// traitement
flux.close();
```

Langage C++ - Pointeurs

➔ **pointeur = adresse mémoire**

permet d'accéder à une variable grâce à son adresse

```
1  int ageUtilisateur(16);
2  int *ptr(0);
3
4  ptr = &ageUtilisateur;
5
6  cout << ageUtilisateur << endl;
7  cout << &ageUtilisateur << endl;
8
9  cout << ptr << endl;
10 cout << *ptr << endl;
```



➔ **reference (&) = pointeur constant**

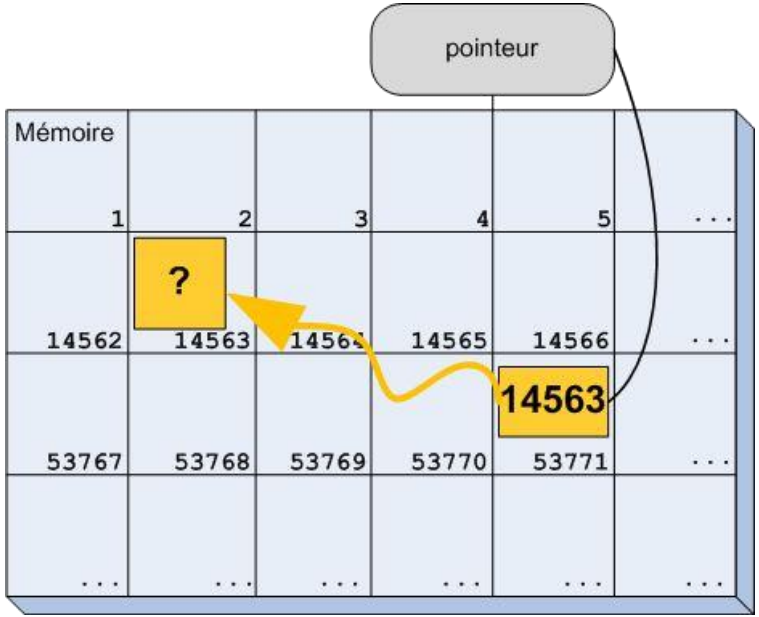
Ex: `int& idUser = 1234;`

Langage C++ - Allocation dynamique

➔ opérateur **new** pour demander manuellement une case dans la mémoire;

➔ opérateur **delete** pour rendre la mémoire;

```
1  int *pointeur(0);
2  pointeur = new int;
3
4  *pointeur = 2;
5
6  delete pointeur;
7  pointeur = 0;
```



➔ si changement de valeur du pointeur, plus possible d'utiliser la mémoire ni de la libérer = **fuite mémoire // OOM Killer / Out Of Memory**

Langage C++ - Bibliothèque standard (SL)

La bibliothèque standard est principalement composée de trois grandes parties :

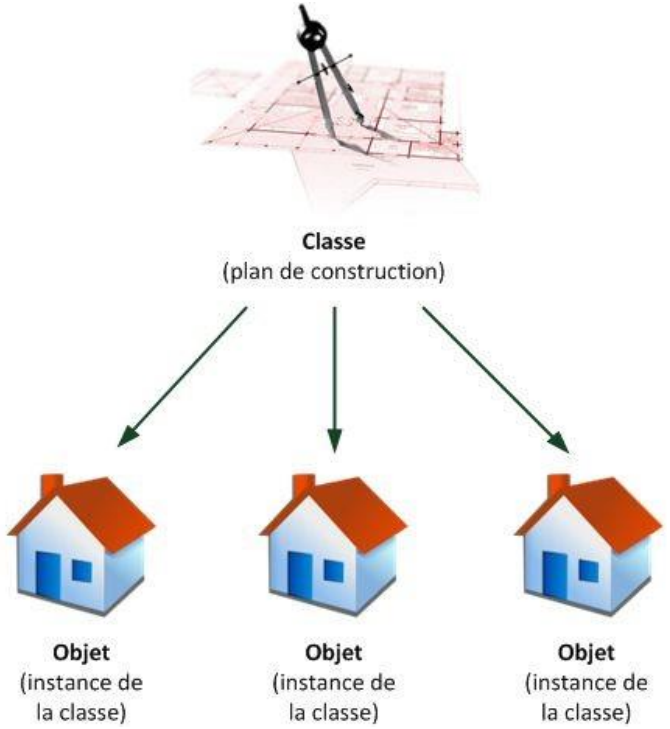
- ➔ l'ensemble de la bibliothèque standard du C (cmath, cctype, ctime, cstdlib, cassert, …), de cette manière, les programmes écrits en C peuvent (presque tous) être réutilisés tels quels en C++;
- ➔ les flux qui permettent d'afficher ou d’écrire des messages dans la console, d'effectuer des opérations sur les fichiers...;
- ➔ la Standard Template Library (STL) qui propose des conteneurs, tels que les vector, des algorithmes standard comme la recherche d'éléments dans un conteneur ou le tri d'éléments, des itérateurs, des foncteurs, des prédicats, des pointeurs intelligents...

http://www.cplusplus.com/reference/
http://yosefk.com/c++fqa/index.html

Programmation Orientée Objet

Langage C++ - Programmation orientée objet

- ➔ dans la programmation orientée objet on manipule des objets;
- ➔ un objet est constitué d'attributs et de méthodes, c'est-à-dire de variables et de fonctions membres;
- ➔ on crée des classes pour définir le fonctionnement des objets, elles correspondent aux plans pour construires des objets;
- ➔ les objets sont la matérialisation concrète des classes, on parle d'instance de classe;
- ➔ la programmation orientée objet impose un code très structuré ce qui rend le code souple, pérenne et réutilisable.



Langage C++ - Classes, attributs et méthodes

➔ un objet peut contenir un autre objet au sein de ses attributs;

➔ possibilité de définir les droits d'accès des méthodes et attributs d'une classe (public, private, protected);

➔ tous les attributs d'une classe doivent toujours être privés, ce qui oblige à utiliser uniquement les méthodes pour manipuler les attributs = **principe d'encapsulation;**

- écrire des accesseurs pour accéder aux attributs;
- **type getX() const** / const indique au compilateur que la méthode ne modifie pas l'objet;
- **void setX(param)** / permet de faire des tests pour vérifier qu'on ne met pas n'importe quoi dans l'attribut;

➔ Séparer prototypes et définitions;

- un header (*.h) qui contiendra les attributs et les prototypes de la classe (singatures des méthodes);
- un fichier source (*.cpp) qui contiendra la définition des méthodes et leur implémentation;
- Dans les .h, il est recommandé de ne **jamais** mettre la directive using namespace (conflits de nom).

Langage C++ - Classes, attributs et méthodes

```cpp
#ifndef DEF_PERSONNAGE
#define DEF_PERSONNAGE

#include <string>

class Personnage
{
    public:
      void recevoirDegats(int nbDegats);
      void attaquer(Personnage &cible);
      void boirePotionDeVie(int quantitePotion);
      void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
      bool estVivant();

    private:
      int m_vie;
      int m_mana;
      std::string m_nomArme;
      int m_degatsArme;
};

#endif
```

```cpp
#include "Personnage.h"

using namespace std;

void Personnage::recevoirDegats(int nbDegats)
{
    m_vie -= nbDegats;
    if (m_vie < 0)
    {
        m_vie = 0;
    }
}

void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_degatsArme);
}

void Personnage::boirePotionDeVie(int quantitePotion)
{
    m_vie += quantitePotion;
    if (m_vie > 100)
    {
        m_vie = 100;
    }
}

void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
{
    m_nomArme = nomNouvelleArme;
    m_degatsArme = degatsNouvelleArme;
}

bool Personnage::estVivant()
{
    return m_vie > 0;
}
```

Langage C++ - Classes, attributs et méthodes

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

using namespace std;

int main()
{
    Personnage david, goliath;
    goliath.attaquer(david);
    david.boirePotionDeVie(20);
    goliath.attaquer(david);
    david.attaquer(goliath);
    goliath.changerArme("Hache", 40);
    goliath.attaquer(david);
    return 0;
}
```

Langage C++ - Constructeur/Destructeur

Le constructeur

➔ méthode appelée automatiquement à chaque fois que l'on crée un objet basé sur une classe;
➔ un constructeur par défaut, vide, est automatiquement créé par le compilateur;
➔ le rôle principal du constructeur est d'initialiser les attributs;
➔ il faut que la méthode ait le même nom que la classe, sans aucun type de retour, même pas void;
➔ il est possible de surcharger le constructeur, permet de créer un objet de plusieurs façons différentes.

dans le .h il faut ajouter le prototype : Personnage();

puis dans le .cpp :

```
//Constructeur par défaut

Personnage::Personnage() {
            m_vie = 100;
            m_mana = 100;
            m_monArme = "Épée rouillée";
            m_degatsArme = 10;
      }
```

équivalent à : Personnage::Personnage() : m_vie(100), m_mana(100), m_nomArme("Épée rouillée"), m_degatsArme(10) {}

Langage C++ - Constructeur/Destructeur

Pour surcharger le constructeur

dans le .h, on ajoute le nouveau prototype : `Personnage(std::string nomArme, int degatsArme);`
puis dans le .cpp : `Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100), m_mana(100), m_nomArme(nomArme), m_degatsArme(degatsArme) {}`
et enfin dans le main : `Personnage david, goliath("Épée", 20);`

Le constructeur par copie

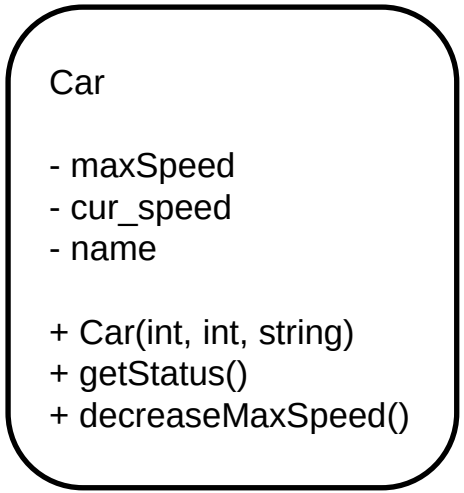
- ➔ surcharge particulière qui permet de créer un objet en lui affectant un autre objet;
- ➔ Le rôle du constructeur de copie est de copier la valeur de tous les attributs du premier objet dans le second;
- ➔ il est appelé lors de la création, l'affectation, et le passage par copie à une fonction;
- ➔ celui généré automatiquement se contente de copier la valeur de tous les attributs, même des pointeurs...
- ➔ Si l'on surcharge le constructeur de copie, alors il faut aussi obligatoirement écrire une surcharge de operator=

dans le .h, on ajoute le prototype : `Personnage(Personnage const& autre);`

puis dans le .cpp : `Personnage::Personnage(Personnage const& autre): m_vie(autre.m_vie), m_mana(autre.m_mana), m_nomArme(autre.m_nomArme), m_degatsArme(autre.m_degatsArme) {}`

Exercice

Coder une classe voiture ayant une vitesse maximale, une vitesse courante et un nom. Pour les méthodes, un constructeur permettant d'initialiser les attributs, une fonction qui affiche les attributs de la voiture et une fonction permettant de baisser la vitesse maximale de la voiture.



Langage C++ - Constructeur/Destructeur

Le destructeur

➔ méthode appelée automatiquement lorsqu'un objet est détruit, lors d'un delete par exemple;
➔ son principal rôle est de désallouer la mémoire;
➔ c'est une méthode qui commence par un tilde (~) suivi du nom de la classe et qui ne renvoie aucune valeur
➔ le destructeur ne peut prendre aucun paramètre, il ne peut pas être surchargé.

dans le .h, on ajoute le prototype :

```
~Personnage();
```

puis dans le .cpp :

```
Personnage::~Personnage() {
// delete et autres vérifications si necessaire
}
```

Langage C++ - Surcharge d’opérateurs

Comme un objet est bien plus complexe qu'un nombre, il faut expliquer à l'ordinateur comment effectuer l'opération.

```
bool operator==(Objet const& a, Objet const& b)
{
     return a.estEgal(b);
}
```

```
bool operator!=(Objet const& a, Objet const& b)
{
     return !(a==b);
}
```

Même si l'on parle de classe, ceci n'est pas une méthode. C'est une fonction normale située à l'extérieur de toute classe.

3 solutions aux problèmes d’accès des attributs depuis l'extérieur de la classe :

➔ création d’accesseurs (aussi appelés getters)
➔ utilisation du concept d’amitié;
➔ création d’une méthode dans la classe qui fera le traitement et appeler cette méthode depuis l’opérateur.

Ré-utiliser du code pré-existant permet de s'assurer que le code fait ce que l'on souhaite et que l'on a pas fait d'erreur stupide.

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Langage C++ - Surcharge d’opérateurs

Les opérateurs raccourcis vont devoir modifier l'objet qui les utilise. Pour respecter l'encapsulation, il va donc falloir déclarer ces opérateurs comme étant une méthode de la classe et pas comme une fonction externe. On peut ensuite réutiliser les opérateurs raccourcis dans nos opérateurs arithmétiques.

```
Objet& Objet::operator+=(const Objet& a)
{
     m_attribut1 += a.m_attribut1;
     m_attribut2 += m_attribut2;
     return *this;
}
```

```
Objet operator+(Objet const& a, Objet const& b)
{
     Objet copie(a);
     copie += b;
     return copie;
}
```

```
resultat = objet1 + objet2 + objet3;
resultat = operator+(operator+(objet1, objet2), objet3);
```

On n'est pas obligé d'additionner des objets de même nature, du moment que cela reste logique et compatible.
Le C++ ne vous permet pas de changer la priorité des opérateurs.

Langage C++ - Surcharge d’opérateurs

Définir ses propres opérateurs pour cout
cout ne connaît pas votre classe, et il ne possède donc pas de fonction surchargée pour les objets de ce type.

```
ostream &operator<<(ostream &flux, Objet const& objet)
{
  objet.afficher(flux);
  return flux;
}
```

Le premier paramètre est l'objet cout. Le second paramètre est une référence constante vers l'objet que vous tentez d'afficher en utilisant l'opérateur <<.

```
void Objet::afficher(ostream &flux) const
{
  flux << attribut1 << "attribut1";
}
```

Langage C++ - Surcharge d’opérateurs

Surcharge globale de new & delete

```
void* operator new(size_t sz)
{
      printf("operateur new: %d octets\n", sz);
      void* m = malloc(sz);
      if(!m) {
            puts("plus de memoire");
      }
      return m;
}
```

```
void operator delete(void* m)
{
      puts("operateur delete");
      free(m);
}
```

Langage C++ - Surcharge d’opérateurs

Surcharge locale (classe) de new & delete

Prototype de méthode à ajouter dans le .h

- ➔ void* operator new(size_t);
- ➔ void operator delete(void*);

L'opérateur new surchargé peut prendre plus d'un argument, cela peut permettre de :

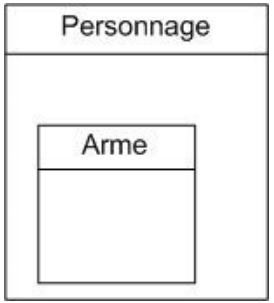
- ➔ placer un objet dans un emplacement spécifique de la mémoire;
- ➔ choisir entre différents allocateurs quand vous appelez new.

Si vous créez un tableau d'objets d’une classe, l'opérateur new global est appelé. Vous pouvez contrôler l'allocation de tableaux d'objets en surchargeant la version spéciale tableaux des opérateurs :

- ➔ void* operator new[](size_t sz);
- ➔ void operator delete[](void* p);

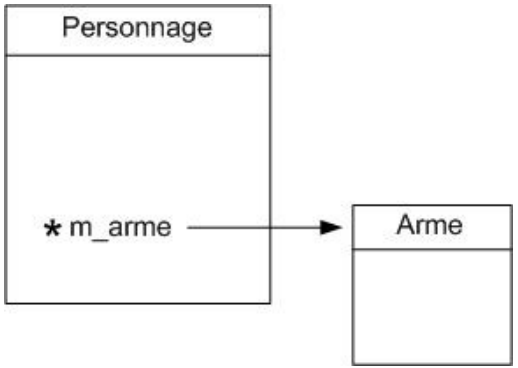
Langage C++ - Classes et pointeurs

Il y a plusieurs façons différentes d'associer des classes entre elles :
- ➔ intégrer directement une classe à ses attributs;
- ➔ utiliser un pointeur d'une classe vers une autre classe.



Un pointeur offre plus de souplesse et de possibilités :
- ➔ inventaire, changer d'arme à tout moment en modifiant le pointeur;
- ➔ donner son arme;
- ➔ ne pas avoir d'arme.

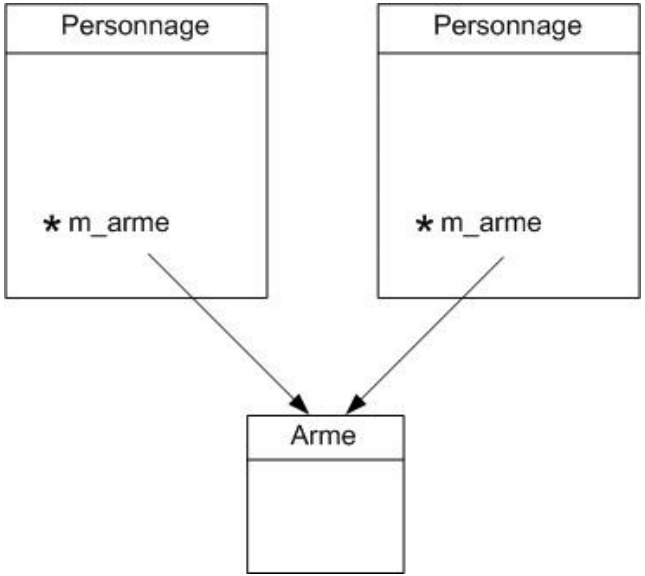
Un pointeur nécessite toutefois de gérer l'allocation dynamique de la mémoire :
- ➔ m_arme = new Arme(); dans le constructeur;
- ➔ delete m_arme; dans le destructeur.



Langage C++ - Classes et pointeurs

Le constructeur par copie est indispensable dans une classe contenant des pointeurs !

```
Personnage::Personnage(Personnage const& personnageACopier)
    : m_vie(personnageACopier.m_vie), m_mana(personnageACopier.m_mana), m_arme(0
{
    m_arme = new Arme(*(personnageACopier.m_arme));
}

Personnage& Personnage::operator=(Personnage const& personnageACopier)
{
    if(this != &personnageACopier)
    {
        m_vie = personnageACopier.m_vie;
        m_mana = personnageACopier.m_mana;
        delete m_arme;
        m_arme = new Arme(*(personnageACopier.m_arme));
    }
    return *this;
}
```



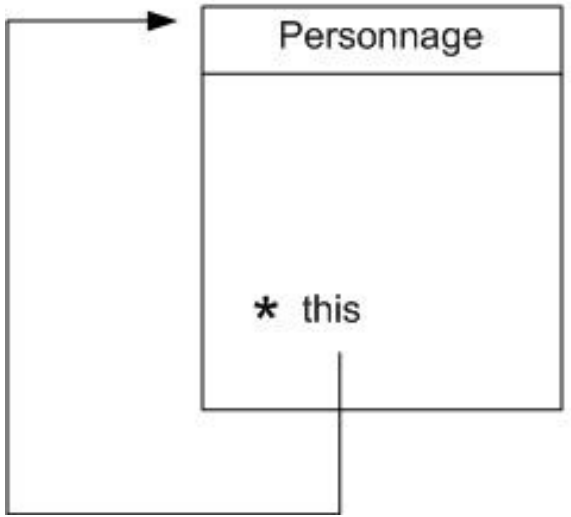
Langage C++ - Classes et pointeurs

Le pointeur **this**

➔ il pointe vers l'objet actuel;

➔ ***this** est l'objet lui-même.

Peut être utile lorsque vous êtes dans une méthode de votre classe et que cette méthode doit renvoyer un pointeur vers l'objet auquel elle appartient.



Langage C++ - Éléments statiques et amitié

Les méthodes statiques : static void maMethode();

➔ méthodes qui appartiennent à la classe mais pas aux objets instanciés à partir de la classe;
➔ elles n’ont pas accès aux attributs de la classe;
➔ utilisées pour regrouper les fonctions dans des classes, par thème, et aussi éviter des conflits de nom.

Les attributs statiques : static int monAttribut;

➔ attributs qui appartiennent à la classe et non aux objets créés à partir de la classe;
➔ initialisation dans l'espace global, en dehors de toute classe ou fonction, en dehors du main();
➔ se comportent comme des variables globales, des variables accessibles partout dans le code;
➔ utilisés pour la création d'un compteur d'instances.

Langage C++ - Éléments statiques et amitié

Dans les langages orientés objet, l'amitié est le fait de donner un accès complet aux éléments d'une classe.

Si je déclare une fonction f amie de la classe A, la fonction f pourra modifier les attributs de la classe A même si les attributs sont privés ou protégés. La fonction f pourra également utiliser les fonctions privées et protégées de la classe A.

En déclarant une fonction amie d'une classe, on casse complètement l'encapsulation de la classe puisque quelque chose d'extérieur à la classe pourra modifier ce qu'elle contient !

➔ les fonctions amies ne sont pas des méthodes de la classe;
➔ une fonction amie ne doit pas, en principe, modifier l'instance de la classe;
➔ les fonctions amies ne doivent être utilisées que si vous ne pouvez pas faire autrement.

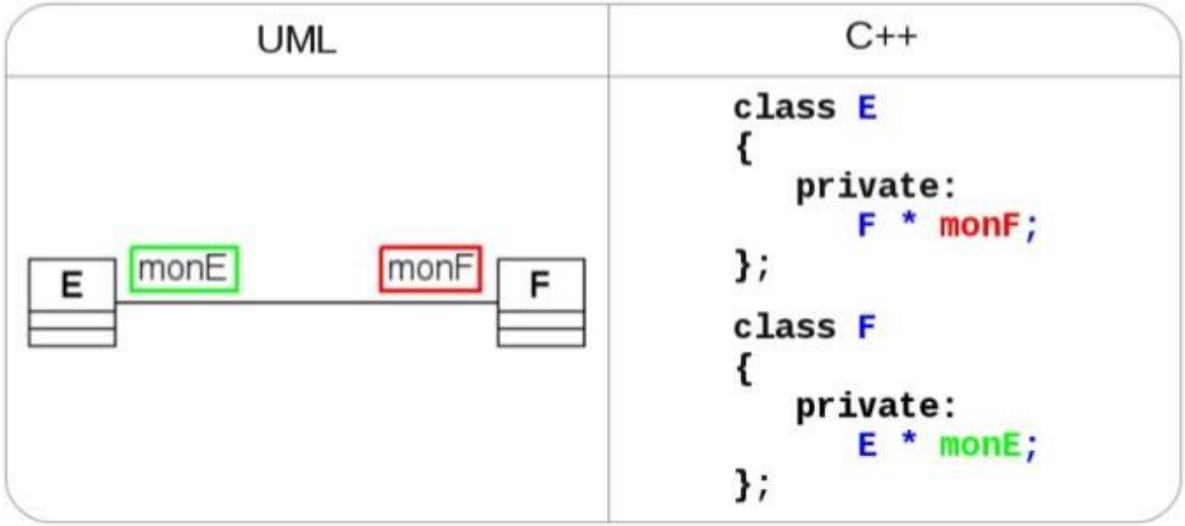
On écrit **friend** suivi du prototype de la fonction et on place le tout à l'intérieur de la classe :

```
friend std::ostream& operator<< (std::ostream& flux, Duree const& duree);
```

Langage C++ - Association, agrégation, composition

Les relations d'association, d'agrégation et de composition illustre une relation de type "avoir".

Une **association** représente une relation sémantique durable entre deux classes.
La relation est bidirectionnelle A connaît B et B connaît A.



Langage C++ - Association, agrégation, composition

Une **agrégation** est un cas particulier d'association non symétrique exprimant une relation de contenance. A est le composite et B le composant. Dans une agrégation, le composant peut être partagé entre plusieurs composites ce qui entraîne que, lorsque le composite A sera détruit, le composant B ne le sera pas forcément.

UML	C++

	`class A`
	`{`
	`  private:`
	`  B * monB;`
	`};`
	
	`class B`
	`{`
	
	`};`

Langage C++ - Association, agrégation, composition

Une **composition** est une agrégation plus forte impliquant le non partage du composant et la destruction de celui-ci lorsque le composite est détruit.

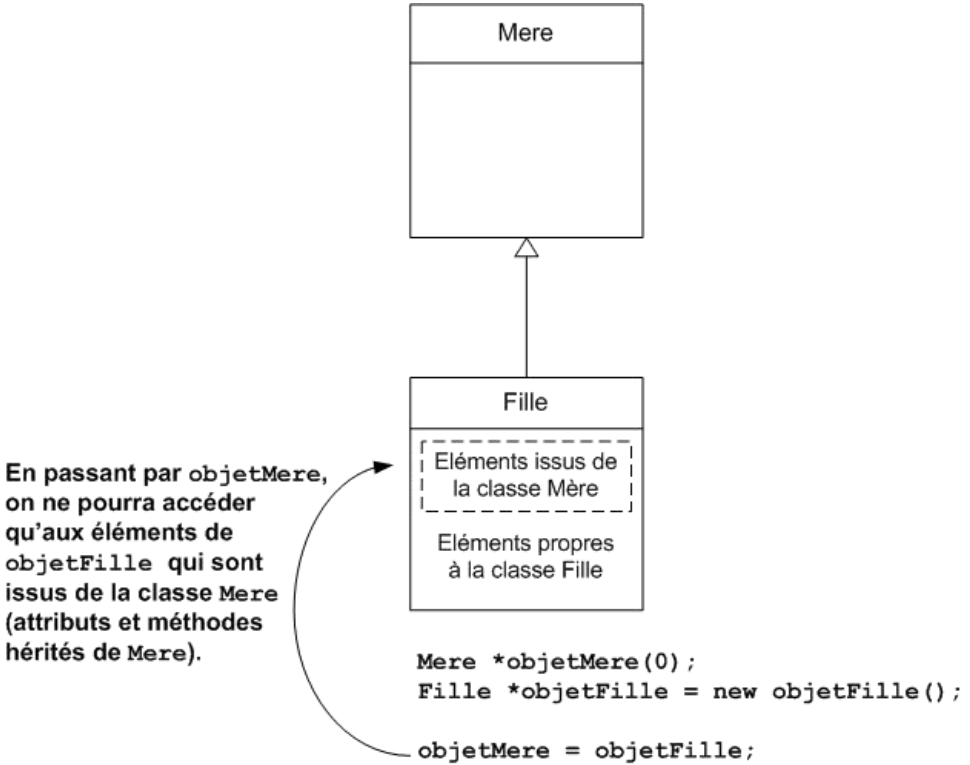
UML	C++

	`class C`
	`{`
	`  private:`
	`  D monD;`
	`};`
	
	`class D`
	`{`
	
	`};`

Langage C++ - Héritage

L’héritage illustre une relation de type “être”. Cela revient à créer une classe à partir d'une autre classe. Elle lui sert de modèle, de base de départ. Cela permet d'éviter d'avoir à réécrire un même code source plusieurs fois.

- ➔ La classe Fille possède toutes les caractéristiques (méthodes et des attributs) de la classe Mere mais elle possède en plus ses propres caractéristiques.
- ➔ On peut affecter un objet enfant à un objet parent ! Par contre, l'inverse est faux !



Langage C++ - Héritage

Dans le .h :

class Fille : public Mere
// créer une classe Fille qui hérite de la classe Mere

- ➔ Les attributs protected ne sont pas accessibles depuis l'extérieur de la classe, sauf si c'est une classe fille. Pour respecter l'encapsulation et l'héritage, toujours la portée protected aux attributs des classes.
- ➔ Il est possible de masquer une fonction de la classe Mere, en créant dans la classe fille une fonction possédant le même nom. Le nombre et le type des arguments ne joue aucun rôle.
- ➔ Economiser des lignes de code en appelant si besoin la fonction masquée grâce à l'opérateur de résolution de portée : Mere::fonction()

- ➔ Le compilateur appelle d'abord le constructeur de la classe mère, puis appelle ensuite le constructeur de la classe fille.

```
Magicien::Magicien() : Personnage(), m_mana(100)
{

}
```

- ➔ Transmission de paramètres

```
Magicien::Magicien(string nom) : Personnage(nom),
m_mana(100)
{

}
```

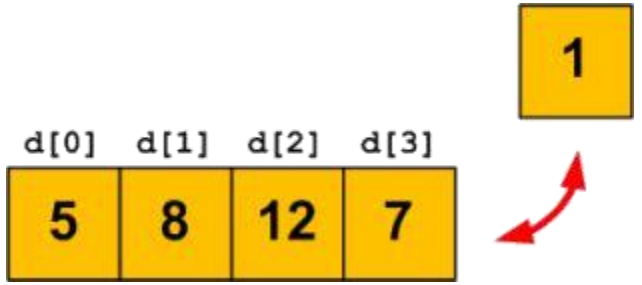
Langage C++ - Les conteneurs (ou les collections)

vector<int>

➔éléments stockés côte-à-côte ;

➔optimisé pour l'ajout en fin de tableau ;

➔éléments indexés par des entiers.

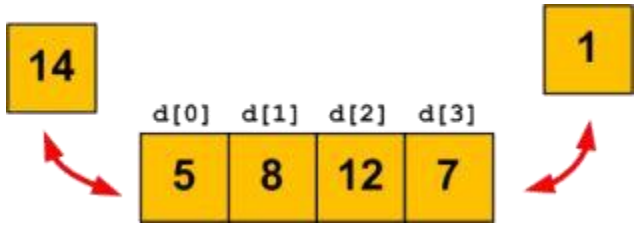


deque<int>

➔éléments stockés côte-à-côte (non garanti) ;

➔optimisé pour l'ajout en début et en fin de tableau ;

➔éléments indexés par des entiers.



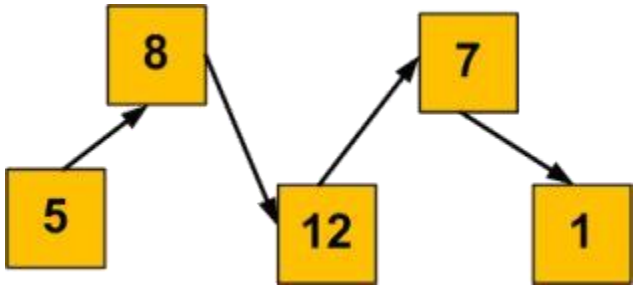
Langage C++ - Les conteneurs (ou les collections)

list<int>

➔éléments stockés de manière « aléatoire » dans la mémoire ;

➔ne se parcourt qu'avec des itérateurs ;

➔optimisé pour l'insertion et la suppression au milieu.



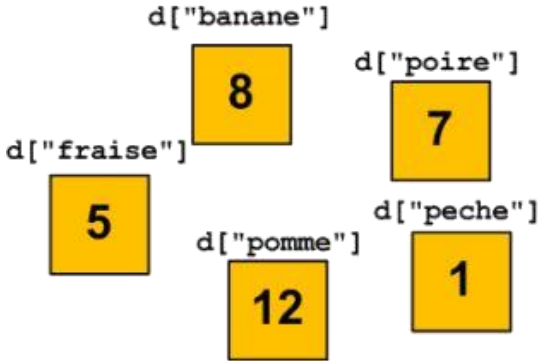
Langage C++ - Les conteneurs (ou les collections)

map<string,int>

➔ éléments indexés par ce que l'on veut ;

➔ éléments triés selon leurs index ;

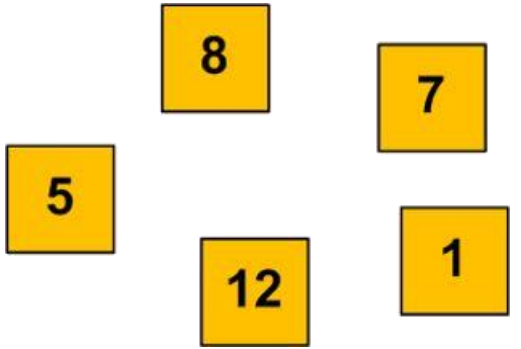
➔ ne se parcourt qu'avec des itérateurs.



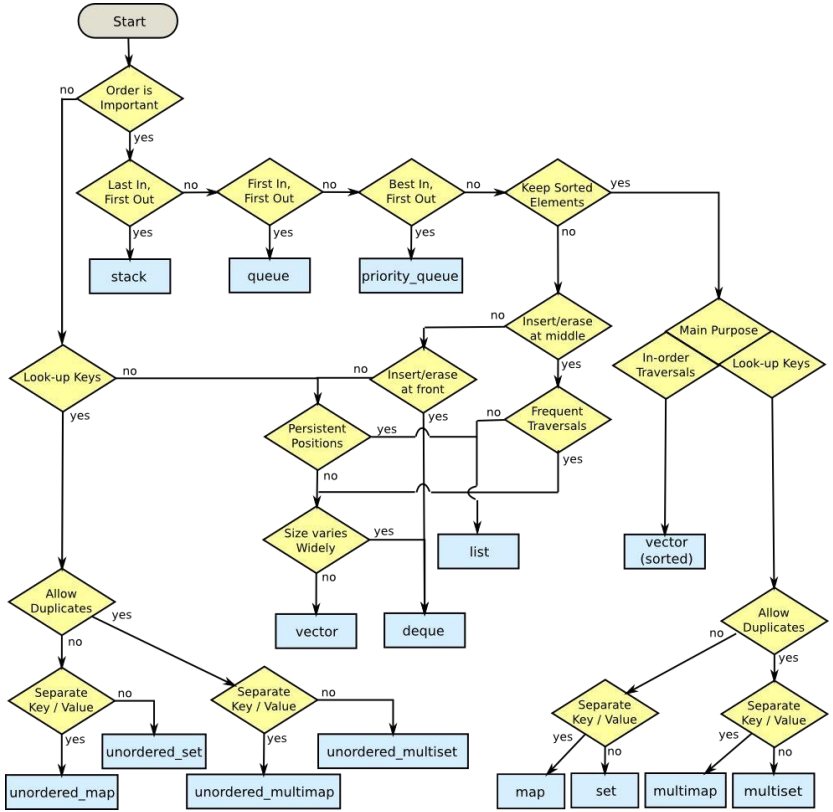
set<int>

➔ éléments triés ;

➔ ne se parcourt qu'avec des itérateurs.



Langage C++ - Les conteneurs (ou les collections)



Langage C++ - Polymorphisme

Résolution statique des liens
C'est le type de la variable qui détermine quelle fonction membre appeler et non sa vraie nature.

```
class Vehicule
{
	public:
	void affiche() const;
};

class Voiture : public Vehicule
{
	public:
	void affiche() const;
};

class Moto : public Vehicule
{
	public:
	void affiche() const;
};
```

```
void presenter(Vehicule v)
{
	v.affiche();
}


int main()
{
	Vehicule v;
	presenter(v);

	Moto m;
	presenter(m);

	return 0;
}
```

La fonction reçoit un Vehicule, c'est donc toujours la « version Vehicule» des méthodes qui sera utilisée.

Langage C++ - Polymorphisme

Résolutions dynamique

➔ utiliser un pointeur ou une référence;

➔ utiliser des méthodes virtuelles (mot-clé “virtual” dans le .h);

➔ il n'est pas nécessaire de mettre «virtual» devant les méthodes des classes filles (automatiquement virtuelles par héritage);

➔un constructeur virtuel n'a pas de sens;

➔ on ne peut pas appeler de méthode virtuelle dans un constructeur;

➔ un destructeur doit toujours être virtuel si on utilise le polymorphisme.

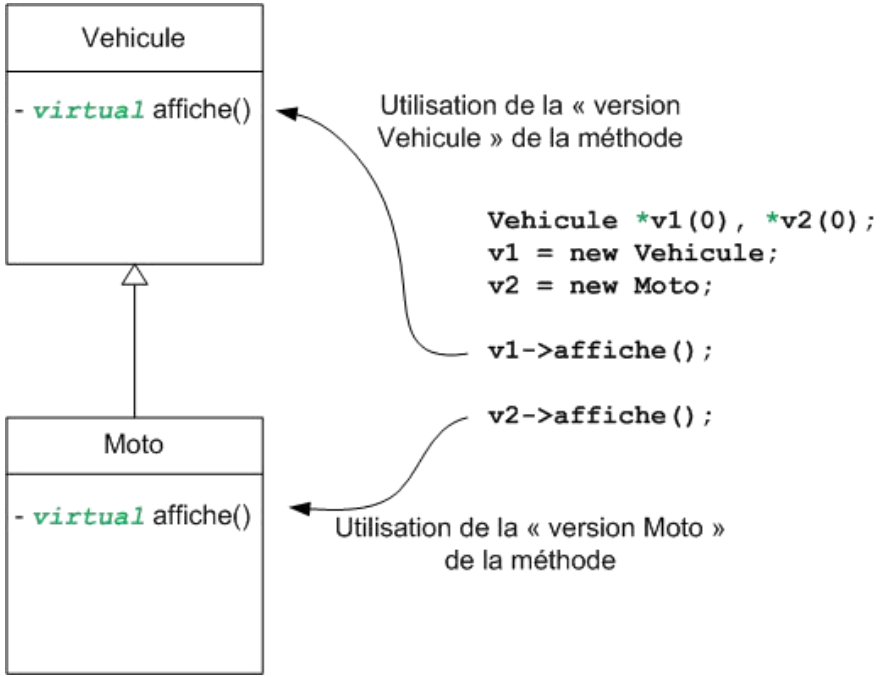
```
void presenter(Vehicule const& v)
{
        v.affiche();
}

int main()
{
        Vehicule v;
        presenter(v);

        Moto m;
        presenter(m);

        return 0;
}
```

La fonction presenter connait la vraie nature du Vehicule. Un même morceau de code a deux comportements différents suivant le type passé en argument. C'est ce qu’on appelle du polymorphisme.

Langage C++ - Polymorphisme

Collections hétérogènes

```
int main()
{
        vector<Vehicule*> listeVehicules;

        listeVehicules.push_back(new Voiture());
        listeVehicules.push_back(new Voiture());
        listeVehicules.push_back(new Moto());

        listeVehicules[0]->affiche();
        listeVehicules[2]->affiche();

        for(int i(0); i < listeVehicules.size(); ++i) {
                delete listeVehicules[i];
                listeVehicules[i] = 0;
        }
        return 0;
}
```



Langage C++ - Polymorphisme

- ➔ une méthode virtuelle peut être redéfinie dans une classe fille;
- ➔ une méthode virtuelle pure doit être redéfinie dans une classe fille;
- ➔ une classe qui possède au moins une méthode virtuelle pure est une classe abstraite;
- ➔ on ne peut pas créer d'objet à partir d'une classe abstraite.

virtual int nbrRoues() const = 0; // **méthode virtuelle pure** à ajouter dans Vehicule.h

Cela revient à dire au compilateur : Dans toutes les classes filles de Vehicule, il y a une fonction nommée nbrRoues() qui renvoie un int et qui ne prend aucun argument mais, dans la classe Vehicule, cette fonction n'existe pas.

Si on ne déclare pas la fonction dans la classe mère, alors on ne pourra pas l'utiliser depuis notre collection hétérogène.

Langage C++ - Template

```
int maximum(int a, int b)
{
    if(a > b)
        return a;
    else
        return b;
}
```

```
double maximum(double a, double b)
{
    if(a > b)
        return a;
    else
        return b;
}
```

point commun = corps de la fonction strictement identique => simplifier les tâches répétitives

➔ ~~pointeur générique void + cast vers un autre type de pointeur~~

➔ template

Langage C++ - Template

template/modèle = permet d'autoriser une fonction ou une classe à utiliser des types différents/générique

➔ **la STL utilise énormément ce concept**

```
template <typename T>
T maximum(const T& a, const T& b)
{
	if(a > b)
		return a;
	else
		return b;
}

maximum<double>(3.11, 2.67)
```

➔ T = type générique ou plus exactement tout type possédant l'opérateur >

➔ mécanisme statique => compilation plus lente, mais exécution plus rapide

➔ tout doit obligatoirement se trouver dans le fichier.h

➔ fonctions ordinaires prioritaires

Langage C++ - Template

➔ Possibilité de spécialiser un template

```
// spécialisation pour choisir le critère de comparaison pour le type string
template <>
string maximum<string>(const string& a, const string& b)
{
	if(a.size() > b.size())
		return a;
	else
		return b;
}
```

Respecter un ordre particulier :

1. la fonction générique
2. les fonctions spécialisées

Langage C++ - Template

➔Tous les arguments d'une fonction ne doivent pas forcément être des types génériques

➔Il est possible de déclarer plusieurs types génériques

```
template<typename T, typename S>  S moyenne(T tableau[], int taille)
{

S somme(0);
  for(int i(0); i < taille; ++i) {
    somme += tableau[i];
  }
return somme / taille;
}

int tab[5];
moyenne<int,double>(tab,5)
```

Langage C++ - Template

Ecrire un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque, le résultat possèdera le même type.

Langage C++ - Template

Ecrire un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque, le résultat possèdera le même type.

template <typename T>

Langage C++ - Template

Ecrire un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque, le résultat possèdera le même type.

```
template <class T>
T carre (T a)
```

Langage C++ - Template

Ecrire un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque, le résultat possèdera le même type.

```
template <class T>
T carre (T a)
{
	return a * a;
}
```

Langage C++ - Template

Ecrire un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque le résultat possèdera le même type.

```
template <class T>
T carre (T a)
{
	return a * a;
}

main()
{
	int n = 5;
	float x = 1.5;
	cout << "carre de " << n << " = " << carre (n) << endl;
	cout << "carre de " << x << " = " << carre (x) << endl;
}
```

Langage C++ - Template

Soit cette définition de patron de fonctions :

```
template <typename T, typename U>
T fct(T a, U b, T c)
{ … }
```

Avec les déclarations suivantes :

```
int n, p, q;
float x;
char t[20];
char c;
```

Quels sont les appels corrects et, dans ce cas, quels sont les prototypes des fonctions instanciées ?

```
fct (n, p, q);        // A1
fct (n, x, q);        // A2
fct (x, n, q);        // A3
fct (t, n, &c);       // A4
```

Langage C++ - Template

Soit cette définition de patron de fonctions :
template <typename T, typename U>
T fct(T a, U b, T c)
{ … }

Avec les déclarations suivantes :
int n, p, q;
float x;
char t[20];
char c;

Quels sont les appels corrects et, dans ce cas, quels sont les prototypes des fonctions instanciées ?
fct (n, p, q); // A1 => int fct (int, int, int)
fct (n, x, q); // A2
fct (x, n, q); // A3
fct (t, n, &c); // A4

Langage C++ - Template

Soit cette définition de patron de fonctions :

```
template <typename T, typename U>
T fct(T a, U b, T c)
{ … }
```

Avec les déclarations suivantes :

```
int n, p, q;
float x;
char t[20];
char c;
```

Quels sont les appels corrects et, dans ce cas, quels sont les prototypes des fonctions instanciées ?

```
fct (n, p, q);      // A1 => int fct (int, int, int)
fct (n, x, q);      // A2 => int fct (int, float, int)
fct (x, n, q);      // A3
fct (t, n, &c);     // A4
```

Langage C++ - Template

Soit cette définition de patron de fonctions :

```
template <typename T, typename U>
T fct(T a, U b, T c)
{ … }
```

Avec les déclarations suivantes :

```
int n, p, q;
float x;
char t[20];
char c;
```

Quels sont les appels corrects et, dans ce cas, quels sont les prototypes des fonctions instanciées ?

```
fct (n, p, q);      // A1 => int fct (int, int, int)
fct (n, x, q);      // A2 => int fct (int, float, int)
fct (x, n, q);      // A3
fct (t, n, &c);     // A4
```

Langage C++ - Template

Soit cette définition de patron de fonctions :

```
template <typename T, typename U>
T fct(T a, U b, T c)
{ … }
```

Avec les déclarations suivantes :

```
int n, p, q;
float x;
char t[20];
char c;
```

Quels sont les appels corrects et, dans ce cas, quels sont les prototypes des fonctions instanciées ?

```
fct (n, p, q);      // A1 => int fct (int, int, int)
fct (n, x, q);      // A2 => int fct (int, float, int)
fct (x, n, q);      // A3
fct (t, n, &c);     // A4 => char * fct (char *, int, char *)
```

Langage C++ - Template

➔ template de classe = classes dont le type des attributs peut varier.

```
template <typename T>
class Rectangle
{
	private:
		T m_gauche;
		T m_droite;
		T m_haut;
		T m_bas;

	public:
		Rectangle(T gauche, T droite, T haut, T bas) : m_gauche(gauche), m_droite(droite), m_haut(haut), m_bas(bas) {}
		T hauteur() const;

		bool estContenu(T x, T y) const
		{
			return (x >= m_gauche) && (x <= m_droite) && (y >= m_bas) && (y <= m_haut);
		}
};
```

Langage C++ - Template

➔ séparer le prototype de la définition => indiquer à nouveau le type T

➔ tout doit se trouver dans le fichier.h

```
template<typename T>
T Rectangle<T>::hauteur() const
{
	return m_hau t- m_bas;
}

Rectangle<double> monRectangle(1.0, 4.5, 3.1, 5.2);
```

Langage C++ - Template

Soit la définition suivante d'un patron de classes :

```
template <class T, int n> class essai
{
	T tab [n];
	public:
		essai(T a);
};
```

A) Donnez la définition du constructeur essai, en supposant :
- qu'elle est fournie "à l'extérieur" de la définition précédente ,
- que le constructeur recopie la valeur reçue en argument dans chacun des éléments du tableau tab.

Langage C++ - Template

Soit la définition suivante d'un patron de classes :

```
template <class T, int n> class essai
{
        T tab [n];
        public:
                essai(T a);
};
```

A) Donnez la définition du constructeur essai, en supposant :
- qu'elle est fournie "à l'extérieur" de la définition précédente ,
- que le constructeur recopie la valeur reçue en argument dans chacun des éléments du tableau tab.

```
template <class T, int n>
```

Langage C++ - Template

Soit la définition suivante d'un patron de classes :

```
template <class T, int n> class essai
{
        T tab [n];
        public:
                essai(T a);
};
```

A) Donnez la définition du constructeur essai, en supposant :
- qu'elle est fournie "à l'extérieur" de la définition précédente ,
- que le constructeur recopie la valeur reçue en argument dans chacun des éléments du tableau tab.

```
template <class T, int n> essai<T,n>::essai(T a)
```

Langage C++ - Template

Soit la définition suivante d'un patron de classes :

```
template <class T, int n> class essai
{
	T tab [n];
	public:
		essai(T);
};
```

A) Donnez la définition du constructeur essai, en supposant :
- qu'elle est fournie "à l'extérieur" de la définition précédente ,
- que le constructeur recopie la valeur reçue en argument dans chacun des éléments du tableau tab.

```
template <class T, int n> essai<T,n>::essai(T a)
{
	for (int i = 0; i < n; i++)
		tab[i] = a;
}
```

Langage C++ - Template

B) Disposant de la définition précédente du patron essai, de son constructeur et de ces déclarations :

```
template <class T, int n> class essai
{
        T tab [n];
        public:
                essai(T);
};

const int n = 3;
int p = 5;
```

Quelles sont les instructions correctes et les classes instanciées :

```
essai<int, 10> ei(3);          // A1
essai<float, n> ef(0.0);       // A2
essai<double, p> ed(2.5);      // A3
```

Langage C++ - Template

B) Disposant de la définition précédente du patron essai, de son constructeur et de ces déclarations :

```
template <class T, int n> class essai
{
        T tab [n];
        public:
                essai(T);
};

const int n = 3;
int p = 5;
```

Quelles sont les instructions correctes et les classes instanciées :

```
essai<int, 10> ei(3);          // A1
essai<float, n> ef(0.0);       // A2
essai<double, p> ed(2.5);      // A3
```

```
class essai
{
        int tab [10];
        public:
                essai(int);
};
essai::essai(int a)
{
        for(int i = 0; i < n; i++)
                tab[i] = a ;
}
```

Langage C++ - Template

B) Disposant de la définition précédente du patron essai, de son constructeur et de ces déclarations :

```
template <class T, int n> class essai
{
        T tab [n];
        public:
                essai(T);
};

const int n = 3;
int p = 5;
```

Quelles sont les instructions correctes et les classes instanciées :

```
essai<int, 10> ei(3);          // A1
essai<float, n> ef(0.0);       // A2
essai<double, p> ed(2.5);      // A3
```

```
class essai
{
        float tab [n];
        public:
                essai(float);
};
essai::essai(float a)
{
        for(int i = 0; i < n; i++)
                tab[i] = a;
}
```

Langage C++ - Template

B) Disposant de la définition précédente du patron essai, de son constructeur et de ces déclarations :

```
template <class T, int n> class essai
{
        T tab [n];
        public:
                essai(T);
};

const int n = 3;
int p = 5;
```

Quelles sont les instructions correctes et les classes instanciées :

```
essai<int, 10> ei(3);          // A1
essai<float, n> ef(0.0);       // A2
essai<double, p> ed(2.5);      // A3 => incorrect car p n'est pas une expression constante
```

Langage C++ - Template

STL

➔ structures de données génériques (conteneurs) **= modèles de classe**

➔ algorithmes génériques sur conteneurs **= modèles de fonction**

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
int main()
{
        int input;
        vector<int> ivec;

        while(cin >> input)
                ivec.push_back(input);

        sort(ivec.begin(), ivec.end());

        vector<int>::iterator it;
        for(it = ivec.begin(); it != ivec.end(); ++it)
                cout << *it << " ";

        cout << endl;
        return 0;
}
```

Langage C++ - Assertions

Assertions = mécanisme de détection et de gestion des erreurs issu du langage C

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

int main()
{
	int a(5);
	int b(5);

	assert(a == b);

	return 0;
}
```

```
monProg: main.cpp:9: int main(): Assertion `a == b' failed.
Abandon
```

➔ A utiliser durant la phase de création d'un programme puis désactiver (option -DNDEBUG)

Langage C++ - Exceptions

Exceptions = moyen de gérer efficacement les erreurs qui pourraient survenir dans un programme en les prévoyant à l'avance

- **try { … }** signale une portion de code où une erreur peut survenir ;
- **throw** signale l'erreur en lançant un objet ;
- **catch(...) { … }** introduit la portion de code qui récupère l'objet et gère l'erreur.

On parle d'exception et pas d'erreur puisque, si on la traite, ce n'est plus une erreur.

```
int division(int a, int b) {
    if(b == 0)
        throw string("ERREUR : Division par zéro !");
    else
        return a / b;
}

int main() {
    int a, b;

    cin >> a;
    cin >> b;

    try {
        cout << division(a,b) << endl;
    } catch(string const& chaine) {
        cerr << chaine << endl;
    }

    return 0;
}
```

Langage C++ - Exceptions

1. On arrête l'exécution du programme au point du "throw";
2. Le programme teste s'il se trouve dans un bloc "try" (si ce n'est pas le cas, fin de programme);
3. On cherche un bloc "catch" compatible avec le type de l'exception levée (si aucun, fin de programme);
4. Le code qui se trouve dans le bloc "catch" est exécuté et le destructeur de tous les objets déclarés à l'intérieur du bloc "try" est appelé;
5. Le programme reprend après le bloc "catch" et non pas au point du "throw".

➔ Il ne faut pas lever une exception dans le destructeur
➔ Attention à l'ordre des blocs "catch"
➔ Plus d'information sur la pile pour la gestion de l'exception

Il est possible, en utilisant throw sans expression derrière, de relancer une exception reçue par un bloc catch afin de la traiter une deuxième fois, plus loin dans le code.

Langage C++ - Exceptions

La classe exception de la bibliothèque standard est la classe de base de toutes les exceptions lancées par la bibliothèque standard. Elle est aussi spécialement pensée pour qu'on puisse la dériver afin de réaliser notre propre type d'exception.

```
class exception {
    public:
        exception() throw() {}
        virtual  exception() throw();

        virtual const char* what() const throw();
};
```

➔ throw indique que ces méthodes ne vont pas lancer d'exceptions (noexcept en CPP11)

On peut alors lancer un objet qui contiendrait plusieurs attributs comme une phrase décrivant l'erreur, le numéro de l'erreur, le niveau de l'erreur (erreur fatale, erreur mineure...), l'heure à laquelle l'erreur est survenue, ...

Langage C++ - Exceptions

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

class Erreur: public exception {
    public:
        Erreur(int numero=0, string const& phrase="", int niveau=0) throw()
            : m_numero(numero),m_phrase(phrase),m_niveau(niveau) {}

        virtual const char* what() const throw() {
            return m_phrase.c_str();
        }

        int getNiveau() const throw() {
            return m_niveau;
        }

        virtual ~Erreur() throw() {}

    private:
        int m_numero;
        string m_phrase;
        int m_niveau;
};
```

```cpp
int division(int a,int b) {
    if(b == 0)
        throw Erreur(1, "Division par zéro", 2);
    else
        return a / b;
}

int main() {
   int a, b;

   cin >> a;
   cin >> b;

    try {
        cout << division(a, b) << endl;
    } catch(std::exception const& e) {
        cerr << "ERREUR : " << e.what() << endl;
    }

    return 0;
}
```

➔ exceptions attrapées par référence constante afin d'éviter une copie et de préserver le polymorphisme de l'objet reçu

Langage C++ - Exceptions

La bibliothèque standard peut lancer 5 types d'exceptions différents :

bad_alloc	Lancée s'il se produit une erreur en mémoire.
bad_cast	Lancée s'il se produit une erreur lors d'un dynamic_cast.
bad_exception	Lancée si aucun catch ne correspond à un objet lancé.
bad_typeid	Lancée s'il se produit une erreur lors d'un typeid.
ios_base::failure	Lancée s'il se produit une erreur avec un flux.

Toutes les exceptions lancées par les fonctions standard dérivent de la classe exception ce qui permet, avec un code générique, de rattraper toutes les erreurs qui pourraient arriver : catch(exception const& e)

Langage C++ - Exceptions

Le fichier std except contient 9 classes d'exceptions séparées en 2 catégories pour les cas les plus courants.

domain_error	logique	Erreur de domaine mathématique.
invalid_argument	logique	Argument invalide passé à une fonction.
length_error	logique	Taille invalide.
out_of_range	logique	Erreur d'indice de tableau.
logic_error	logique	Autre problème de logique.
range_error	exécution	Erreur de domaine.
overflow_error	exécution	Erreur d'overflow.
underflow_error	exécution	Erreur d'underflow.
runtime_error	exécution	Autre type d'erreur.

Langage C++ - RAII

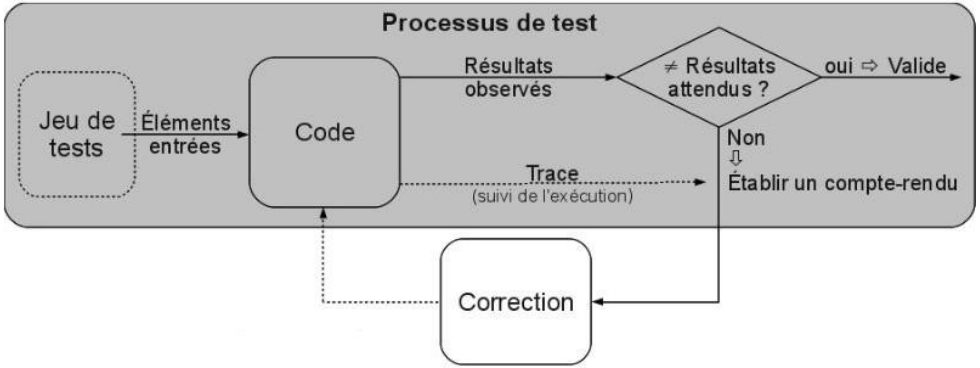
RAII = Resource Acquisition Is Initialization = Acquisition de Ressources lors de l'Initialisation

- ➔ encapsule la gestion d'une ressource au sein d'un objet qui va acquérir cette ressource lors de son initialisation et la libérer lors de sa destruction;
- ➔ se résumer en « tout faire dans le constructeur et le destructeur »;
- ➔ code exception-safe sans aucune fuite de mémoire;
- ➔ rendu possible en C++ par le fait que les classes disposent d'un destructeur qui est appelé dès qu'un objet sort de son bloc de portée (alternative à finally);
- ➔ utilisé intensivement dans la bibliothèque standard : gestion des fichiers, des chaînes de caractères, des tableaux dynamiques, des pointeurs;
- ➔ principe de base des pointeurs intelligents qui permettent d'envelopper toutes sortes de ressources (std::unique_ptr, std::shared_ptr, std::weak_ptr).

Langage C++ - RAII

```
class FileHandle {
    public:
        FileHandle(const char* name, const char* mode) {
            f_ = fopen(name, mode);
        }

        FILE* file() {
            return f_;
        }

        ~FileHandle() {
            if (f_ != nullptr)
                fclose(f_);
        }

        private:
            FILE* f_;
};


string do_stuff_with_file(string filename) {
    FileHandle handle(filename.c_str(), "r");
    int firstchar = fgetc(handle.file());

    if(firstchar != '$') {
        return "bad bad bad";
    }

    return string(1, firstchar);
}
```

Langage C++ - Tests unitaires



tests unitaires => tests d'intégration => tests de validation => tests de recette
On évalue à environ 40% la part des tests dans le coût d'un logiciel (et plus pour des logiciels critiques).

tests unitaires = chaque module du logiciel est testé séparément par rapport à ses spécifications, c'est-à-dire, les plus petites unités testables (classes/méthodes)

mocks = objets simulés qui reproduisent le comportement d'objets réels de manière contrôlée

Google Test + Google Mock
https://github.com/google/googletest

Boost
http://www.boost.org/

Catch2
https://github.com/catchorg/Catch2

Langage C++ - Thread

Calcul concurrent = forme de calcul dans laquelle plusieurs calculs sont exécutés pendant des périodes de temps qui se chevauche.

- ➔ Un thread est une petite partie du processus (programme) qui peut être exécutée en même temps que d'autres parties du processus;
- ➔ Les threads s'exécutent dans un espace mémoire partagé.

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

void f() { cout << "Hello"; }

struct F {
    void operator()() { cout << "Parallel World!\n"; }
};

void user() {
    thread t1{f};
    thread t2{F()};

    t1.join();
    t2.join();
}
```

Langage C++ - Thread

Calcul concurrent = forme de calcul dans laquelle plusieurs calculs sont exécutés pendant des périodes de temps qui se chevauche.

- ➔ Un thread est une petite partie du processus (programme) qui peut être exécutée en même temps que d'autres parties du processus;
- ➔ Les threads s'exécutent dans un espace mémoire partagé.

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

void f() { cout << "Hello"; }

struct F {
    void operator()() { cout << "Parallel World!\n"; }
};

void user() {
    thread t1{f};
    thread t2{F()};

    t1.join();
    t2.join();
}
```

- ➔ sortie imprévisible et pouvant varier d'une exécution à l'autre : PaHerallllel o World!
- ➔ contrôle par serrures ou autres mécanismes afin d’empêcher les accès simultanés

Langage C++ - Thread

Passage d'arguments

t1 = accepte une séquence arbitraire d'arguments (variadic template)

ref = fonction de <functional> pour traiter some_vec comme un wrapper de référence

Autoriser la modification à l'aide d'une non const ref ou d'une const ref et d'un second argument pour déposer le résultat

```
void f(vector<double>& v)

struct F {
    vector<double>& v
    F(vector<double>& vv) : v{vv} {}
    void operator()();
};

void user() {
    vector<double> some_vec {89, 76, 56};
    vector<double> vec2 {10,2,1,4};

    thread t1{f, ref(some_vec)};
    thread t2{F(vec2)};

    t1.join();
    t2.join();
}
```

Langage C++ - Thread

Partage des données

synchronisation avec les mutex
(mutual exclusion/exclusion mutuelle)

1. Un thread acquiert un mutex en utilisant une opération lock();
2. Si le mutex est déjà acquis, le thread attend jusqu'à ce que l'autre clôture son accès.

➔ Attention au deadlock en cas d'utilisation de multiple lock; Si thread1 acquiert mutex1 et essaye ensuite d'acquérir mutex2 alors que thread2 acquiert mutex2 et essaie ensuite d'acquérir mutex1…

➔ locking/unlocking parfois plus cher que la copie;

➔ lock inutile pour les variables atomic (voir std::atomic).

```cpp
#import <mutex>
#include <thread>
using namespace std;

mutex m;
int sh;

void f() {
    unique_lock<mutex> lck{m}; // acquire mutex
    sh += 7; // manipulation shared data
    // release mutex implicitly
}

// ---multiple lock---

void f() {
    // don't yet try to acquire the mutex
    unique_lock<mutex> lck1{m1, defer_lock};
    unique_lock<mutex> lck2{m2, defer_lock};
    unique_lock<mutex> lck3{m3, defer_lock};

    lock(lck1, lck2, lck3); //acquire all three locks
}
```

Langage C++ - Thread

Attendre des événements

Parfois un thread doit attendre un certain type d'événement externe :

- ➔ le temps qui passe;
- ➔ un autre thread avec certaines conditions.

```
#import <chrono>

using namespace std::chrono;

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20}); // refers to the one and only thread
auto t1 = high_resolution_clock::now();

cout << duration_cast<nanoseconds>(t1-t0).count() << " nanoseconds passed\n";
```

Langage C++ - Thread

condition_variable = mécanisme permettant à un thread d'en attendre un autre avec certaines conditions, e.g., assurer qu'une file d'attente n'est pas vide.

#include <condition_variable>

```
class Message {
    ...
};

queue<Message> mqueue;
condition_variable mcond;
mutex mmutex;
```

```
void consumer()
{
    while(true) {
        unique_lock<mutex> lck{mmutex}; // acquire mmutex
        mcond.wait(lck)); // release lck and wait
        // re acquire lck upon wale up
        auto m = mqueue.front(); // get message
        mqueue.pop();
        lck.unlock(); // release lck
        //process m
    }
}
```

```
void producer()
{
    while(true) {
        Message m;
        unique_lock<mutex> lck {mmutex};
        mqueue.push(m);
        mcond.notify_one();
    }
}
```

Langage C++ - Lambda Function

Le concept C++ de fonction lambda ou fonction anonymes trouve son origine dans le lambda-calcul et la programmation fonctionnelle. Une telle fonction peut être utilisée pour des petits morceaux de code qui ne valent pas la peine d'être nommées.

[] () mutable -> T { }
[] correspond à la liste de capture, () la liste d'argument, mutable autorise la modification des variables capturées par valeur, T correspond au type de retour et {} au corps de la fonction.

La liste de capture définit ce qui est disponible à l'intérieur du corps de la fonction et comment. Il peut s'agir soit d'une variable [x], d'une référence [&x], de toute variable actuellement dans la portée par référence[&] ou par valeur[=].

Langage C++ - Thread

C++ propose dans <future> quelques moyens pour opérer au niveau conceptuel des tâches : promise/futures, packaged_task, async

- ➔ promise/future permet le transfert d'une valeur entre deux tâches sans utilisation explicite d’une serrure;
- ➔ lorsqu'une tâche veut transmettre une valeur à une autre tâche, elle la met dans une promise, la valeur apparaîtra dans le future correspondant;
- ➔ get() => future (task1) <=> value <=> promise (task2) <= set_value/set_exception.

```
void f(promise<X>& px) {
    try {
        X res;
        px.set_value(res);
    } catch() {
        px.set_exception(current_exception());
    }
}

void g(future<X>& fx) {
    try {
        // if necessary wait, or throw an exception
        X v = gx.get();
    } catch() {

    }
}
```

Langage C++ - Thread

packaged_task permet de paramétrer les tâches liées par des futures et promises afin d'être exécuté sur thread en se concentrant sur les tâches à accomplir plutôt que sur les mécanismes utilisés pour gérer leur communication.

- ➔ absence de mention explicite des serrures
- ➔ packaged_task ne peut pas être copié car il gère des ressources
- ➔ move = permet d'éviter une copie

```
double accum(double* beg, double* end, double init) {
    return accumulate(beg, end, init);
}

double comp2(vector<double>& v) {
    using Task_type = double(double*, double*, double);

    packaged_task<Task_type> pt0 {accum};
    packaged_task<Task_type> pt1 {accum};

    future<double> f0 {pt0.get_future()};
    future<double> f1 {pt1.get_future()};

    double* first = &v[0];
    thread t1 {move(pt0), first, first+v.size()/2, 0};
    thread t2 {move(pt1), first+v.size()/2, first+v.size(), 0};

    return f0.get() + f1.get();
}
```

Langage C++ - Thread

Lancer des tâches de manière asynchrone

➔ utiliser async;

➔ pas besoin de penser aux threads et aux locks;

➔ async décide du nombre de thread en fonction des ressources système disponibles.

```
double comp4(vector<double>& v) {
    if(v.size() < 10000)
        return accum(v.begin(), v.end(), 0.0);

    auto v0 = &v[0];
    auto sz = v.size();

    auto f0 = async(accum, v0, v0+sz/4, 0.0); // first quarter
    auto f1 = async(accum, v0+sz/4, v0+sz/2, 0.0); // second quarter
    auto f2 = async(accum, v0+sz/2, v0+sz*3/4, 0.0); // third quarter
    auto f3 = async(accum, v0+sz*3/4, v0+sz, 0.0); // fourth quarter

    return f0.get() + f1.get() + f2.get() + f3.get();
}
```