Programmation Objet pour le Calcul scientifique

Master 1 Calcul haute performance et simulation

Sylvain Desroziers

2017-2018

IFP Energies nouvelles

Objectifs

Introduction générale à la programmation orientée objet :

- Concepts et paradigmes ;
- Quelques bonnes pratiques ;
- Ce n'est pas un cours de Génie Logiciel.

Approfondissement du langage C++ :

- Normes C++11/14 ;
- Recommandations pour la performance ;
- Tout n'est malheureusement pas abordé.

Application au calcul scientifique :

- Spécificités et exemples dans le cours ;
- Mise en pratique durant les TD/TP.

Détails pratique

Prérequis pour le cours :

- Avoir des bases de programmation procédurale,
- Avoir une machine sous `Linux`/`MacOS` :
  - Machine virtuelle : `http://www.cartnum.uvsq.fr`
- Avoir un compilateur `C++` (on y reviendra),
- Connaître un environnement de développement.

Pour me contacter : `sylvain.desroziers@ifpen.fr`

- Mettre dans l'objet `[CHPS-M1]`,
- Je ne suis pas un débugger et je ne lis pas dans les pensées.

Introduction

La problématique...

La complexité d'un logiciel est le facteur clé à maîtriser :

- La complexité est à la fois d'ordre technique et fonctionnelle ;
- La complexité d'un logiciel en fonction de sa taille n'est pas linéaire ;
- Le rôle de l'architecte est d'assurer que cette complexité reste le plus proche du linéaire.

Le génie logiciel apporte une solution concrète :

- Les processus de développements ont pour but de rationaliser cette complexité ;
- Le génie logiciel fournit des outils et méthodes pour :
  - Analyser les besoins ;
  - Concevoir le comportement du logiciel ;
  - Implanter (et maintenir) le système.

... ou comment mettre toutes les chances de son côté !

Mais il n'y a pas de processus idéal !

- La difficulté majeure consiste à s'adapter aux changements tout en garantissant l'avancement du développement.
- On se base sur des méthodes itératives et incrémentales fragmentant du logiciel en sous-produits :
  - Meilleure organisation ;
  - Augmentation de la réactivité ;
  - Prise de risques moins élevée.

Un modèle (ou langage) de programmation favorisant la réutilisabilité et la robustesse est un avantage dans ce cadre !

La modélisation

Modèle

Dans le langage des ingénieurs, un modèle est un assemblage de concepts représentant une chose *réelle* (ou *existante*) de manière simplifiée dans le but de la comprendre.

Un modèle a pour objectif de structurer les informations et activités d'une organisation : données, traitements, et flux d'informations entre entités.

Modélisation

L'art de distinguer le fondamental du superflu.

C'est le principe le plus scientifique de la création de logiciel :

1. Comprendre la réalité ;
2. Comprendre le domaine d'activité ;
3. Dégager les concepts fondamentaux pour le domaine.

Notion de paradigme

Paradigme

Représentation du monde, manière de penser, modèle cohérent de représentation du monde qui repose sur une base définie. En informatique, un paradigme correspond à un style fondamental de programmation qui traite de la manière dont les solutions aux problèmes doivent être formulées.

Il existe de nombreux paradigmes :

- Programmation procédurale ;
- Programmation orientée objet ;
- Programmation par contrat ;
- Programmation par aspect ;
- Programmation générique...

Le paradigme n'est pas lié à un langage !

Le paradigme procédural

Le logiciel est organisé autour de la notion de procédure. Le programme est une suite de procédures s'exécutant les unes à la suite des autres afin d'effectuer une tâche donnée :

- Les fonctionnalités sont placées dans les procédures ;
- Les procédures ont accès à la structure de données.

Pour réduire la complexité, on effectue pour un périmètre donné :

- Abstraction de procédures : on regroupe les procédures ;
- Abstraction de données : on regroupe les données.

C'est le paradigme le mieux maîtrisé et le plus employé en calcul scientifique.

Le paradigme *orienté objet*

Le logiciel est organisé autour de la notion d'*objet* :

- Un objet est une abstraction de données contenant des abstractions de procédures ;
- Le mécanisme de communication entre objets est l'envoi de message.

Le système est décrit sous forme d'un ensemble d'objets possédant leurs états propres et interagissant entre eux :

- C'est un style de programmation en pleine effervescence et parfaitement adapté au principe de modélisation ;
- Ce paradigme est de plus en plus employé en calcul scientifique.

Pourquoi l'*orienté objet* ?

La programmation orientée objet favorise :

- L'évolutivité ;
- La réutilisabilité ;
- L'encapsulation ;
- La robustesse.

Les impacts positifs sur un projet sont de :

- Faciliter la répartition des tâches ;
- Minimiser les coûts ;
- Réduire le « time to market ».

C'est une approche technique nécessitant une rigueur dans son application pour en tirer le meilleur !

Le modèle objet (1/2)

Pour modéliser un système complexe basé sur une réalité physique, il est courant de faire l'équivalence entre un objet et une entité physique.

Un objet est formé d'un état et d'un ensemble de comportement modélisés comme des réactions à des messages. Il a une identité, une durée de vie et endosse un ou plusieurs rôles dans le système.

Les principes fondamentaux sont :

- modularité : la logique interne de l'objet est décorrélée de son utilisation
- encapsulation : la seule façon d'influencer un objet est de lui envoyer un message
- abstraction : les objets sont classifiés suivant une relation de généralisation

Le modèle objet (2/2)

Ce modèle comporte des avantages certains :

- Les objets facilitent le raffinement local du modèle ;
- Les objets améliorent la réutilisabilité ;
- Le mécanisme d'héritage permet l'extension d'un composant.

Mais ce n'est pas une solution ultime :

- L'état d'un objet peut influencer sa réaction aux messages et ce n'est pas observable de l'extérieur ;
- Le mécanisme d'héritage est utilisé différemment en fonction du niveau d'abstraction ;
- Les messages sont ponctuels, non-quantifiables et compliquent les expressions calculatoires ;
- On aimerait pouvoir empêcher certaines extensions dangereuses.

En route vers le C++ !

Le C++ en comparaison du C

Quels sont les avantages (non exhaustifs) ?

- Le C est l'ancêtre du C++, interopérabilité complète ;
- Le C++ est rapide, au moins autant que le C ;
- Le C++ évolue, propose des paradigmes modernes ;
- Le C++ est la langage d'avenir en calcul scientifique ;
- Le C++ s'interface avec le `Fortran`.

Quels sont les désavantages (non exhaustifs) ?

- Le C++ n'est pas simple, encore moins pour la performance ;
- Le C++ n'aide pas intrinsèquement à écrire du code robuste !

Nous allons voir que le C++ est un langage puissant mais nécessite de l'expertise.

Le C++ et les autres langages

Et concernant `Matlab` ou `Python` ?

- Langages aut niveau adaptés aux applications scientifiques ;
- Développement / prototpyage rapide ;
- Passage à l'échelle difficile, trop haut niveau pour les performances et les modèles de programmation parallèle ;
- Peu interopérable (ou avec du travail).

Et par rapport au sacro saint `Fortran` ?

- Langage majeur et historique pour les applications scientifiques ;
- De très nombreuses libraires de calcul scientifique incontournables, i.e. `BLAS` / `LAPACK` ;
- Un langage vieillissant, peu d'évolution ;
- Risque sur les compilateurs.

Finalement, chaque langage a son périmètre d'utilisation qu'il faut connaître et respecter.

Un langage compilé multi-paradigme

Le C++ est un langage de programmation compilé, permettant la programmation sous de multiples paradigmes comme la programmation procédurale, la programmation orientée objet et la programmation générique.

source : `https://fr.wikipedia.org/wiki/C%2B%2B`

Extensions des fichiers (le plus souvent) :

- sources (compilables) : `.cc`, `.cpp`, `.c++`, `.cxx`
- entêtes (incluables) : `.h`, `.hh`, `.hpp`, `.h++`, `.hxx`

Principaux compilateurs et versions pour norme C++14 :

- `GNU g++` version 5.0 (latest 7.2)
- `INTEL icpc` 16.0 (incomplet, latest 17.0)
- `LLVM clang++` version 3.4 (latest 5.0)

Hello World

Fichier `hello.cpp` :

```cpp
#include <iostream>

using namespace std;
int main(int argc, char** argv)
{
  // print welcome message
  cout << "Hello World" << endl;

  return 0;
}
```

Introduction pédagogique

On retrouve des ingrédients classiques en programmation C++ :

- Les type de données (`int`) et variables (`argc`, `argv`) ;
- Les fonctions (`main`) et les blocs / portées ({ ... }) ;
- La bibliothèque standard (`std`) et les espaces de nommage (`namespace`) ;
- et les commentaires (`//` ou `/* ... */` à la C).

Pour compiler (norme C++98) :

```
desrozis> g++ -o hello hello.cpp
desrozis> ./hello
Hello World
```

Pour compiler avec les normes récentes, on utilise l'option de compilation `-std=c++<version>`

- c++11, c++14, c++17, etc.
- Une version tous les 3 ans.

Inclusion de fichiers

On distingue deux types de fichier :

- les fichiers *sources*, qui sont *compilés*,
- et les fichiers d'entêtes ou *headers*, qui sont *inclus* dans les autres fichiers, servent à la déclaration.

Par convention, les extensions des headers sont `.h`, `.H`, `.hh` ou `.hpp` et les extensions des sources sont `.cc` ou `.cpp`.

Les séquences `#include` sont remplacées par le contenu des fichiers au préprocessing.

Fichiers système (chemins prédéfinis) :

```
#include <filename>
```

Fichiers utilisateur :

```
#include "filename"
```

Protection des headers

L'inclusion mutuelle de fichiers peut créer des recursions infinies :

```cpp
// header foo.hpp
#include "bar.hpp"
... // sources
```

```cpp
// header bar.hpp
#include "foo.hpp"
... // sources
```

On protège alors les fichiers inclus avec des directives du préprocesseur :

```
#ifndef <nom>
#define <nom>
...
#endif
```

```cpp
// header foo.hpp
#ifndef _FOO_HPP_
#define _FOO_HPP_
#include "bar.hpp"
... // sources
#endif
```

```cpp
// header bar.hpp
#ifndef _BAR_HPP_
#define _BAR_HPP_
#include "foo.hpp"
... // sources
#endif
```

Types et variables

Un langage *typé*

En C++, les variables ont obligatoirement un type qui, une fois fixé, ne peut plus changer :

```
<type> <variable>;
```

Le langage n'est pas considéré *fortement* typé car il autorise la conversion implicite de type !

Exception : Pour ne rien déclarer, il existe le type vide `void` mais on ne peut déclarer de variable avec ! Principalement utilisé pour déclarer l'absence d'argument ou de retour dans les fonctions.

Règle de nommage

Les noms de variable peuvent être composés de caractères alphanumériques ainsi que de '_' mais :

- ne doivent pas commencer par un chiffre,
- attention à la casse,
- ne pas choisir des noms de mots clés existants !

Remarque : Nommer correctement est une chose importante, c'est un premier pas vers la qualité logicielle !

Il existe de nombreuses conventions et règles de codage :

- `http://www.possibility.com/Cpp/CppCodingStandard.html`
- `https://google.github.io/styleguide/cppguide.html`
- `http://www.boost.org/development/requirements.html`

Booléen

Pour déclarer une variable booléene, on utilise le mot clé `bool` :

```
bool b;
```

Une variable de type `bool` s'initialise avec les valeurs `true` ou `false` :

```
bool b1 = true; // copie de littéral
bool b2(false); // initialisation directe
bool b3 { true }; // initialisation uniforme (C++11)
```

Toute valeur entière différente de `0` est implicitement convertie à `true` :

```
bool b1 = 0; // false
bool b2 = 1; // true
bool b3 = 2; // true
```

Ici, l'opérateur d'affectation simple = sert à initialiser les variables.

Initialisation uniforme

L'initialisation uniforme a été introduite pour :

- éviter les ambiguïtés de construction pour les variables, les appels de fonctions, etc. ;
- uniformiser et faciliter la construction des variables.

En particulier, cela évite les conversions implicites malheureuses :

- On ne peut initialiser une valeur plus petite avec une plus grande avec l'initialisation uniforme.

```
bool b1 { 0 }; // false
bool b2 { 1 }; // true
bool b3 { 2 }; // ERREUR
```

Tableaux de données

Un tableau est une séquence d'éléments du même type (placés en mémoire de manière contigüe) où la taille est une constante connue à la compilation :

`<type-name> <variable-name>[<size>];`

Le tableau suivant est un tableau 5 éléments de type `bool` :

```
bool b[5];
```

Par défaut, les tableaux ne sont pas initialisés. Les tableaux peuvent être initialisés avec une liste de valeurs entre accolades :

```
bool b1[4] = { false, 0, true, 1 }; // initialisation uniforme !
bool b2[4] { false, 0, true, 1 }; // initialisation uniforme aussi !
bool b3[]  = { false, 0, true, 1 }; // taille 4 automatiquement déduite

bool b4[2] = { }; // initialisation à 0 0
bool b5[2] = { 0, 0, 0 }; // ERREUR trop de valeurs !
```

Accès de données

Chaque valeur du tableau peut être accédée individuellement en utilisant l'opérateur `[]` :

```
bool b[3];

b[0] = true; // en écriture
b[1] = true;
b[2] = false;
b[3] = false; // ERREUR débordement de tableau !

bool b2 = b[0]; // en lecture
```

Attention, en C++, l'indice des tableaux commence à `0`

Tableaux mutlidimensionnels

Un tableau multidimmensionnel est un *tableau de tableau* :

```
bool b[2][3][4]; // matrice 2x3x4

b[0][2][1] = true;
```

La première dimension peut être déduite à l'initialisation et l'initialisation par défaut opère :

```
bool b[][2] = { // matrice 2x2
  { 1 }, { }    // ligne 0 initialisée à {1, 0}
};              // ligne 1 initialisée à {0, 0}
```

L'initialisation peut également se faire en ligne :

```
bool b[3][2] = { // matrice 3x2
1, 3, 5, 2, 4, 6 // ligne 0 initialisée à {1, 3}
};               // ligne 1 initialisée à {5, 2}
                 // ligne 2 initialisée à {4, 6}
```

Caractères

Il y a 3 types *distincts* pour les caractères (codés sur 8-bits) :

- `char` : 'A', ..., 'Z', 'a', ..., 'z', '0', ... '9', etc.
- `signed char` : en nombre *signé*, i.e. -128, ... 127
- `unsigned char` : en nombre *non signé*, i.e. 0, ... 255

```
char c1;
char c2 = 'a'; // un unique caractère
unsigned char c3 = 100; // initialisation par valeur entière
```

Depuis, le C++11, il y a trois nouveaux types pour l'unicode (non abordé dans le cours) :

- `wchar_t`, `char16_t`, `char32_t`

Séquences de caractère

Une séquence de caractère est un tableau d'une taille donnée :

```
char c[20];
```

Ainsi, on peut stocker dans `c` la chaîne `"Hello"` ou `"Good Bye"`.
Toute chaîne de caractère est terminée par le caractère `'\0'` :

```
char c[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Les séquences de caractères entre " sont des *constantes littérales* et ont toujours le caractère de terminaison `'\0'` :

```
char c[] = "Hello";
c[0] = 'H';
c[1] = 'e';
c[2] = 'l';
c[3] = 'l';
c[4] = 'o';
c[5] = '\n';
```

Nombre entier

On dispose de représentations signées des entiers :

- `short` : codés sur 16 bits, valeurs de -2^{15} à $2^{15} - 1$
- `int` : codés sur 32 bits, valeurs de -2^{31} à $2^{31} - 1$
- `long` : codés sur 64 bits, valeurs de -2^{63} à $2^{63} - 1$

Remarque : On peut aussi écrire `signed short`, `signed int`, etc.

Mais aussi de représentations non signées d'entiers :

- `unsigned short` : codés sur 16 bits, valeurs de 0 à $2^{16} - 1$
- `unsigned int` : codés sur 32 bits, valeurs de 0 à $2^{32} - 1$
- `unsigned long` : codés sur 64 bits, valeurs de 0 à $2^{64} - 1$

Attention aux *dépassements d'entier* !

Un dépassement d'entier (integer overflow) est une condition qui se produit lorsqu'une opération mathématique produit une valeur numérique supérieure à celle représentable dans l'espace de stockage disponible.

source : `https://fr.wikipedia.org/wiki/D%C3%A9passement_d%27entier`

Ceci crée au mieux des bugs, au pire des failles de sécurité !

Voici un bug critique (mais fixé) dans `OpenSSH` :

```
unsigned int size = nresp * sizeof(char*);
response = xmalloc(size); // Aie !!
```

La taille maximale d'un `unsigned int` est $2^{32} - 1$, `sizeof(char*)` vaut 4 donc si `nresp` plus grand que 2^{30}, il y a dépassement, i.e. une troncature à 2^{32}.

De manière similaire, on peut définir les integer underflow.

Nombre réel à virgule flottante

Les nombres réels sont représentés (norme `IEEE 754`) par :

$$r = S \times M \times 2^{(E-B)}$$

où S est le signe, M la mantisse et E l'exposant de biais B.

Précision	Encodage	Exposant	Mantisse	Biais
simple	32 bits	8 bits	23 bits	127
double	64 bits	11 bits	52 bits	1023

Il y a deux types de nombres réels en `C++` :

- `float` : simple précision, $r \in [-3 \times 10^{38}, 3 \times 10^{38}]$;
- `double` : double précision, $r \in [-2 \times 10^{308}, 2 \times 10^{308}]$.

Précision de représentation

La précision machine est *définie* comme le plus petit nombre ϵ tel que $1 + \epsilon \neq 1$.

- `float` : $\epsilon \approx 5.96 \cdot 10^{-8}$;
- `double` : $\epsilon \approx 1.11 \cdot 10^{-16}$.

A cause de leur représentation limitée, les nombres réels sont arrondis. Ceci forme l'erreur de représentation et peut mener à des erreurs numériques se *propageant* lors des calculs.

Notation		
$\sqrt{2}$	Exact	1.414 213 562 373 095 048 80...
	`double`	1.414 213 562 373 095 145 47...
	`float`	1.414 213 538 169 860 839 84...
e	Exact	2.718 281 828 459 045 235 36
	`double`	2.718 281 828 459 045 090 79...
	`float`	2.718 281 745 910 644 531 25...

Attention à la perte de précision !

Soient

$$\overline{a} = (a + \Delta a) \qquad \text{et} \qquad \overline{b} = (b + \Delta b)$$

où

- a et b sont des valeurs exactes,
- $\overline{a}$ et $\overline{b}$ les approximations,
- Δa et Δb les erreurs.

L'erreur relative de $\overline{x} = \overline{a} - \overline{b} = (a - b) + (a\Delta a - b\ \Delta b)$ est

$$\frac{|x - \overline{x}|}{|x|} = \frac{|a\Delta a - b\ \Delta b|}{|a - b|}.$$

Ainsi, si $a \simeq b$, de petits Δa et Δb peuvent engendrer une grande erreur relative !

On parle de cancellation.

Constante littérale

Les constantes littérales réfèrent à des valeurs fixes, connues à la compilation, que le programme ne peut changer.

```
10          // int
10u         // unsigned int
10l         // long
10ul        // unsigned long
10lu        // unsigned long
1.23456L    // long double
1.23e45f    // float
1.23456     // double
1.23e45     // 1.23 x 10^45, double
1.2E-34     // 1.2 x 10^-34, double
```

Remarque : Chaque constante littérale écrite dans le code est en général propagée par le compilateur dans un souci de performance. C'est l'atout performance de la programmation générique !

Resource Acquisition Is Initialisation

Il y a toujours danger à déclarer une variable sans l'initialiser. Si on utilise une variable non initialisée, le comportement n'est pas défini mais en général, le compilateur émet un avertissement.

Resource Acquisition Is Initialisation (RAII) est un idiome de programmation pour les langages orientés objet. En substance, une variable doit toujours être initialisée au plus tôt, i.e. à la déclaration. On évite les zones de trouble, c'est une très bonne pratique !

```
double d1 = 1.0; // RAII ok
double d2; // ne pas utiliser !
           // ... zone instable ...
d2 = d1;   // ok maintenant
```

Attention, on ne peut utiliser que ce qui est déclaré :

```
double zero = 1.0 - one; // ERREUR, v n'est pas encore déclaré
double one  = 1.0;
```

const-correctness

Toute variable de tout type peut être déclarée non mutable, i.e. constante par l'utilisation du mot clé `const` devant le type :

`const <type-name> <variable-name> = <init-value>;`

```
const float v { 0.0f };
v = 1.0f; // ERREUR
```

Là encore, c'est une très bonne pratique :

- Cela force à programmer RAII car l'initialisation est ainsi obligatoire !
- On peut éviter des erreurs de programmation liées à la modification malheureuse de variables ;
- Le compilateur peut tirer parti de cette information pour l'optimisation.

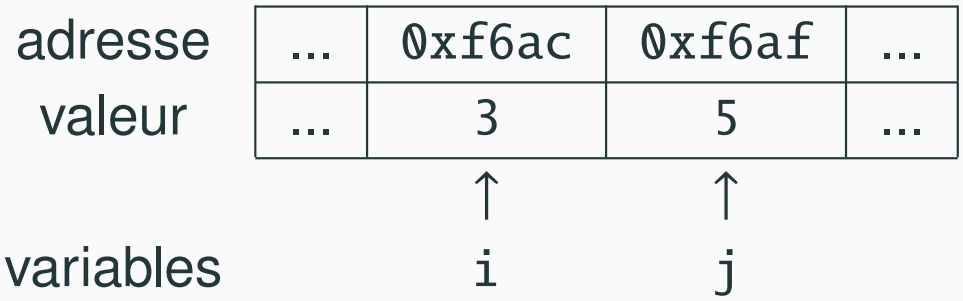
Remarque, le mot clé `const` agit par défaut *sur la gauche* :

```
float const v { 0.0f }; // ok
```

Variable et mémoire

Toute variable est stockée en mémoire à une adresse unique.

```c
int i = 3, j = 5;
```



C'est le système d'exploitation qui gère la mémoire, i.e. les adresses accessibles par le programme.

Attention : accéder à une zone mémoire non autorisée crée en général un arrêt brutal de l'execution.

Notion de *lvalue* et *rvalue*

Il y a principalement 2 types d'expressions en C++ :

- lvalue : expression qui réfère à un espace mémoire, persiste au delà d'une expression unique, peut apparaître à droite ou à gauche d'un assignement, porte un nom :

```
double one;
one = 1.0; // ok
```

- rvalue : expression qui réfère à la valeur d'une donnée temporaire non persistante au delà de l'expression, peut apparaître à uniquement à droite d'un assignement

```
1.0 = one;           // ERREUR
(one - 1.0) = 0.0; // ERREUR
```

Attention : En fait, il y a 5 types d'expressions en C++11, voir
http ://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3092.pdf

Pointeur

Un pointeur est une variable spécifique contenant l'adresse d'une autre variable d'un type donné :

`<type>* <pointeur>;`

```
int* pi;
```

Pour obtenir la valeur de la variable *pointée*, on utilise l'opérateur de déréférencement * et inversement, l'adresse d'une variable est donnée par l'opérateur & :

`* <pointeur>;`

`& <variable>;`

```
int i = 1;
int* pi = &i;
*pi = 3; // change la valeur de i qui vaut 3 maintenant
int* pi2 = pi; // un autre pointeur pointant i
```

Pointeur `nullptr`

Il existe une constante litérale spécifique aux pointeurs, quelque soit le type pointé sous-jacent, pour dire qu'un pointeur ne pointe sur aucune variable :

```
double* p = nullptr;
```

Ainsi, un pointeur doit toujours être initialisé soit à l'adresse d'une variable, soit à `nullptr`. On dit dans ce cas que le pointeur est *nul*.

Attention, manipuler des pointeurs est un risque similaire aux variables non initialisées.

```
int* pi = nullptr;

int j = *pi + 1;
```

Relation entre tableau et pointeur

En C++, il n'y a pas de distinction entre pointeur et tableau :

```
bool b[] = { false, false };

bool* p = b;

p[0] = true; // b[0] = true
p[1] = true; // b[1] = true
```

On peut aussi écrire en arithmétique dite *pointeur*. Les opérations possibles sur les pointeurs sont des additions et soustractions d'entier tels que `p + n` équivaut à l'adresse contenue dans `p` décalée de `n` éléments en mémoire.

```
bool b[] = { false, false };

bool* p = b;

* p      = true; // b[0] = true
*(P + 1) = true; // b[1] = true
```

Tableau de pointeurs

On peut utiliser les tableaux de pointeurs pour effectuer des tableaux multi-dimensions :

```
int n1[4], n2[4], n3[4], n4[4];

int* p1 = n1;
int* p2 = n2;
int* p3 = n3;
int* p4 = n4;

int* p[4] = { p1, p2, p3, p4 };
```

Remarque : on effectue de plusieurs accès en mémoire en utilisant les pointeurs et rien n'interdit que ces mémoires soient éloignées sur la machine.

Référence

La référence est une variante du pointeur et ne peut être initialisée uniquement avec des adresses de variables existantes :

`<type> & <reference>;`

```
int i = 5;

int &ri = i; // référence de i

ri = 3; // change la valeur de i qui vaut 3 maintenant

int &ri2 = ri; // une autre référence de i
```

Attention, à la différence des pointeur, on ne peut changer la variable référencée par une référence.

Remarque, on parle de référence *lvalue* car la mémoire est existante. En pratique, dès qu'une variable est nommée, une adresse est associée.

Pointeur / référence constant

On peut utiliser le qualifieur `const` avec les pointeurs et les références :

```
      int*       p1; // pointeur non const vers une variable non const
const int*       p2; // pointeur non const vers une variable const
      int* const p3; // pointeur const vers une variable non const
const int* const p4; // pointeur const vers une variable const
      int& r1; // référence vers une variable non const
const int& r2; // référence vers une variable const
```

Si le pointeur est constant, l'adresse qu'il pointe ne peut être modifiée. Si la variable pointée est constante, sa valeur ne peut être modifiée :

```
const int i = 1;
const int* pi1 = &i; // ok
      int* pi2 = &i; // ERREUR
const int& ri1 = i; // ok
      int& ri2 = i; // ERREUR
```

Inférence de type

Depuis C++11, les types peuvent être automatiquement déduits par le compilateur en utilisant le mot clé `auto` :

```
auto d = 3.0;          // d est double
auto i = 10u;          // i est unsigned int
const auto& rd = d;    // rd est une référence double
const auto* pi = &i; // pi est une pointeur unsigned int
```

Les types peuvent également être calculés en utilisant le mot clé `decltype` :

```
int x;
decltype(x) y;      // int y;
int& r = y;         // decltype(r)  = int&
const int& cr = r; // decltype(cr) = const int&
int* p = &y;        // decltype(p)  = int*
```

L'usage de `auto` réduit la conversion (involontaire) des types. Les codes se recentrent sur les algorithmes.

Inférence des expressions

L'inférence des expressions de type *rvalue* est obtenue par promotion stricte de type :

```
decltype(1) i;      // int
decltype(2+3) j;    // int
decltype(i + 2) k; // int
```

Si `v` est une variable alors `(v)` est une expression de type *lvalue*. Une référence est alors ajouté au calcul de type :

```
int i = 2;

decltype(i) d = i;
d = 3; // i = 2

decltype((i)) e = i; // e est int& !
e = 3; // i = 3
```

Conversions implicites

Des conversions implicites ont automatiquement lieu quand une valeur est copiée dans un type compatible :

```
const int one = 1.0;
```

Ici, le nombre `1.0` est une constante littérale de type `double`. Le résultat est alors converti implicitement en `int`.

Attention : les conversions implicites sont silencieuses et à éviter. Les conversions à la `C` sont également à proscrire.

Remarque : il faut gérer les overflow, troncatures, etc.

La conversion explicite de type est effectuée via

`typename( value )`

```
const float one = float(1.0);
```

Outils de conversion

Le C++ définit 4 opérateurs de conversion :

- `const_cast<T>(v)` : retire l'attribut `const` d'une variable v, peut s'avérer utile mais à manipuler avec précaution. En général, cela traduit un problème de conception ;
- `static_cast<T>(v)` : conversion classique de type avec vérification à la compilation (Je sais qui je suis) ;

```
int one = static_cast<int>(1.0);
```

- `reinterpret_cast<T>(v)` : conversion de type à l'execution, utilisé pour des conversions de pointeurs avec types avancés (`nullptr`) ;
- `dynamic_cast<T>(v)` : conversion de type avec vérification à l'execution (Qui suis-je ?), principalement utilisé avec des classes.

Définition de type

Il peut être utile de parfois renommer des types pour par exemple apporter un sens métier. Pour cela, on utilise le mot clé `typedef` :

```
typedef <type> <alias>;
```

```
typedef double type;
typedef type*  pointer;
typedef type&  reference;
```

De manière équivalente en C++11 on utilise le mot clé `using` :

```
using <alias> = <type>;
```

```
using type      = double;
using pointer   = type* ;
using reference = type& ;
```

Types manquants

En C++, le langage ne définit pas de type natif pour (par exemple) :

- manipuler les nombres complexes,
- manipuler les chaines de caractères,
- manipuler les pairs, tuples, etc.

Nous verrons dans la suite qu'il est possible de définir des *classes* d'objet permettant de définir nos propres types.

Remarque, la bibliothèque standard du C++ met à disposition un lot de types avancés `std::complex`, `std::string`, `std::pair`, `std::tuple`, etc.

Opérateurs

Opérateurs arithmétiques

Les opérateurs sont des symboles permettant de manipuler les variables. En C++, les opérateurs arithmétiques classiques pour les nombres entiers et flottants sont définis :

```
auto x = 2, y = 1;

auto r1 = x + y; // r1 = 3
auto r2 = x - y; // r2 = 1
auto r3 = x * y; // r3 = 2
auto r4 = x / y; // r4 = 2
```

Concernant les nombres entiers, l'opérateur *module* (reste de la division entière) est défini par % :

```
auto x = 5, y = 2;

auto r = x % y; // r = 1
```

Opérateurs d'affectation

L'opérateur = peut être combiné avec les opérateurs arithmétiques précédents définissant les opérateurs d'affectation *composés* :

```
auto x = 2, y = 1;

x += y; // x = x + y;
x -= y; // x = x - y;
x *= y; // x = x * y;
x /= y; // x = x / y;
x %= y; // x = x % y; pour les entiers
```

Les opérations d'affectation sont des expressions qui peuvent être évaluées :

```
auto x = 2, y = 1 + (x  = 3); // x = 3, y = 4
auto x = 2, y = 1 + (x *= 3); // x = 6, y = 7
```

Attention, cette écriture n'est pas intuitive, à éviter.

Opérateurs d'incrémentation / décrémentation

L'opérateur de *pré-incrementation* (respectivement *pré-décrementation*) incrémente (respectivement décremente) la valeur de la variable et retourne une référence du résultat.

```
auto i = 0;
++i; // i = 1;
--i; // i = 0;
auto j = 2 * (++i); // i = 1, j = 2
```

L'opérateur de *post-incrementation* (respectivement *post-décrementation*) crée une copie, incrémente (respectivement décremente) la valeur de la variable et retourne la copie.

```
auto i = 0;
i++; // i = 1;
i--; // i = 0;
auto j = 2 * (i++); // i = 1, j = 0
```

Valeurs `inf` et `nan`

En arithmétique entière, diviser par zéro arrête en général le programme :

```
auto i = 0, j = 1 / i; // ERREUR
```

En arithmétique flottante, il y a la valeur `inf` signifiant *infini* :

```
auto i = 0.0, j = 1.0 / i; // j = inf
```

Il y a également la valeur `nan` signifiant *not a number* pour les cas non définis, par exemple la racine carré d'un nombre négatif, la division de zéro par zéro, etc :

```
auto i = 0.0, j = i / i; // j = nan
```

Attention, en général, le programme continue même si des valeurs sont `nan` ou `inf`, d'où l'intérêt de bien initialiser les variables.

Opérateurs logiques

Les opérateurs logiques classiques sont définis pour le type `bool` :

```
bool b =  true && false; // et, false
bool b =  true || false; // ou, true
bool b = !true;          // négation, false
```

Remarque, les expressions booléennes sont évaluées jusqu'à l'obtention du résultat :

```
auto i = 0, j = 1, k = 2;

auto b = (i == 0 && j == 2 && k == 2); // k == 2 n'est pas évalué
                                       // car j == 2 faux

auto b = (i == 0 || j == 2 && k == 2); // seul i == 0 est testé
```

Associativité

Les opérations arithmétiques et logiques sont associatives *par la gauche* (en respectant la priorité des opérateurs) :

```
auto i = 1 + 2 + 3 + 4; // (((1 + 2) + 3) + 4)
auto j = 1 + 2 * 3 - 4; // ((1 + (2 * 3)) - 4)

auto a = true && false && true; // ((true && false) && true)
auto b = true || false && true; // (true || (false && true))
```

Tandis que les opérations d'affectations sont associatives *par la droite* :

```
int i, j, k;

i = j = k = 0; // (i = (j = (k = 0)));
```

Opérateurs de comparaison

En C++, les opérateurs classiques de comparaison sont définis :

```
bool b1 = (x >  y); // x plus grand que y
bool b2 = (x <  y); // x plus petit que y
bool b3 = (x >= y); // x plus grand ou égal à y
bool b4 = (x <= y); // x plus petit ou égal à y
bool b5 = (x == y); // x égal à y
bool b6 = (x != y); // x différent de y
```

Attention, il faut être vigilant concernant les comparaisons entre nombres flottants à cause de la représentation approchée et des éventuelles pertes de précisions :

```
auto i = 0.7f;

auto b1 = (i == 0.7); // comparaison double/float (voir IEEE) b1 = false

auto eps = 1e-7;

auto b2 = (i > 0.7 - eps) && (i < 0.7 + eps); // b2 = true
```

Les opérateurs en résumé

Priorité	Associativité	Opérateurs
haute	gauche	()
	droite	++, –, - (unaire), !, & (adresse), * (déréference)
	gauche	* (multiplication), /, %
	gauche	+, -
	gauche	>, <, >=, <=
	gauche	==, !=
	gauche	&&
	gauche	||
faible	droite	=, +=, -=, *=, /=, %=

Remarque : Les parenthèses sont prioritaires, il faut les utiliser en cas de doute !

Blocs et déclarations

Blocs de déclaration

Un *bloc de déclarations* est une région de code délimitée par des accolades contenant un ensemble de déclarations traité comme une unique déclaration par le compilateur :

```
{ // début de bloc
  // liste de déclarations
} // fin de bloc
```

Les blocs peuvent être imbriqués les uns dans les autres :

```
{     // début du bloc 1
  {   // début du bloc 2
  }   //   fin du bloc 2
  {   // début du bloc 3
    { // début du bloc 4
    } //   fin du bloc 4
  }   //   fin du bloc 3
}     //   fin du bloc 1
```

Remarque : l'indentation est une aide pour la lecture du code

Variables locales

Des variables peuvent être déclarées dans chaque bloc, les *variables locales*. Ces variables ont une portée (*scope*, durée de vie) liée au bloc, c'est-à-dire qu'elles sont crées (et éventuellement initialisées) à leur définition et automatiquement détruite à la sortie du bloc :

```
{
  auto i = 1;   // i crée et initialisé ici
  auto d = 0.0; // d crée et initialisé ici
} // sortie du bloc, i et d sont détruits
```

On ne peut pas avoir deux variables locales de même nom dans le même bloc :

```
{
  auto i = 1; // i crée et initialisé ici
  // ...
  auto i = 2; // ERREUR, i existe déjà
}
```

Portée et blocs imbriqués

Une variable locale déclarée est accessible dans les blocs imbriqués dans le bloc dans lequel elle est définie. Inversement, une variable locale déclarée dans un bloc encapsulé est détruite à la sortie du bloc et n'est plus accessible :

```
{
  auto i = 1;  // i crée et initialisé ici
  {
    auto j = i; // ok, j crée et initialisé ici
  } // sortie du bloc, j est détruit mais pas i

  auto k = j;  // ERREUR, j est inconnue
}
```

Shadowing

Une variable d'un bloc peut être masquée par une variable de même nom dans un bloc imbriqué :

```
{
  auto i = 1; // i crée et initialisé ici
  {
    auto i = 2; // ok, autre i crée et initialisé ici
    // i == 2 ici
  } // sortie du bloc, autre i est détruit

  // i == 1 ici
  {
    i = 2; // ok, affectation de i
  }
  // i == 2 ici
} // sortie du bloc, i est détruite
```

Attention, c'est typiquement une source de problèmes difficiles à détecter car la sémantique est bonne. Ces nommages sont à éviter.

En pratique

Il est fortement conseillé de définir les variables dans le scope le plus petit possible. Autrement dit, si une variable n'est utilisée que dans un bloc imbriqué, il faut la définir dans le bloc :

```
{
  auto i = 1; // i crée et initialisé ici
              // car on en a besoin dans ce bloc
  {
    auto j = 2; // j crée et initialisé ici
                // car on en a besoin dans ce bloc
                // et nul part ailleurs

  } // sortie du bloc, j est détruite

} // sortie du bloc, i est détruite
```

Espace de nommage

Le mot clé `namespace` est utilisé pour déclarer une portée qui contient un ensemble d'objets connexes. Vous pouvez utiliser un espace de noms pour organiser les éléments de code et créer des types globaux uniques.

source : `https://docs.microsoft.com/fr-fr/dotnet/csharp`

`namespace <nom> { <bloc> }`

Le namespace apporte de la cohérence en permettant de grouper, i.e. d'encapsuler :

```
namespace math {
  auto pi = 3.14;
  auto epsilon = 1.e-8;
}
```

Remarque, c'est un élément important en génie logiciel.

Espace de nommage

Un élément d'un namespace est utilisé de la manière suivante :

`<namespace>::<nom>`

```
auto pi = math::pi;
```

En utilisant le mot clé `using`, on accède aux éléments du namespace :

```
using namespace math; // on accede a tout math
```

Remarque, pas d'intérêt de grouper pour dégrouper ensuite.

Le namespace définit une technique pour éviter les conflits de nommage.

Remarque, bonne pratique à utiliser le plus possible.

Structures de contrôle

Sélecteur conditionnel `if`

En C++, l'expression conditionnelle par le mot clé `if` permet d'exécuter un bloc d'instructions si le résultat d'une condition est vraie :

```
if ( <condition> ) <bloc_true>
```

Un bloc d'instruction peut être executé si la condition est fausse :

```
if ( <condition> ) <bloc_true>
else               <bloc_false>
```

Remarque : toute valeur non nulle est considérée comme vraie

```cpp
if (i > 0) {
  auto j = -i; // variable locale j
  i = j;
} // j est détruit
else {
  i *= -1;
}
```

Enchainement de conditions

Il est possible d'enchainer les expressions conditionnelles en répétant les mots clés `else` et `if` :

```
if (i == 0) {
  i = 101;
} else if (i == 1) {
  i = 11;
} else if (i == 2) {
  i = 1;
} else {
  i++;
}
```

Boucle `for`

La boucle `for` est une structure permettant d'executer plusieurs fois la même série d'instructions jusqu'à ce qu'une condition de fin soit réalisée :

`for ( <init> ; <condition> ; <increment> ) <bloc>`

L'étape `init` est executé une fois en premier, puis `condition` est évaluée. Le bloc d'instruction `bloc` est ensuite executé. A la fin d'itération (i.e. le sortie du bloc), l'instruction `increment` est finalement executée :

```
auto n = 1;

for (auto i = 1; i < 10; ++i) {
  n *= i;
}
```

Attention, il est très facile d'écrire des boucles infinies.

Boucle `while`

La boucle `while` est une alternative à la boucle `for`. Ici, le bloc d'instruction est répété tant que la condition est vraie. Le condition est évaluée avant chaque itération :

`while ( <condition ) <bloc>`

```
auto n = 1, i = 1;

while (i < 10) {
  n *= i;
  ++i;
}
```

Boucle `do-while`

La boucle `do-while` est une variante à la boucle `while`. Le bloc d'instruction est aussi répété tant que la condition est vraie. La condition est par contre évaluée après chaque itération :

`do <bloc> while ( <condition> );`

```
int n = 1, i = 1;

do {
  n *= i;
  ++i;
} while (i < 10);
```

Interruption de boucle

Le mot clé `break` permet d'interrompre immédiatement une boucle sans tester la condition (ni la satisfaire). Seule la boucle courante est arrêtée.

Remarque : `break` s'utilise avec les trois formes de boucle `for`, `while` et `do-while`.

```
for (; true; ) { // boucle infinie...
  break; // ... mais interruption !
}

while (true) { // boucle infinie...
  break; // ... mais interruption !
}

do {
  break; // interruption...
} while(true); // ... sinon boucle infinie !
```

Interruption d'itération

Le mot clé `continue` permet d'interrompre immédiatement l'itération courante. La condition est alors testée.

Remarque : `continue` s'utilise avec les trois formes de boucle `for`, `while` et `do-while`.

```
for (; true; ) { // boucle infinie...
  continue;
  break; // ... et jamais d'interruption !
}

while (true) { // boucle infinie...
  continue;
  break; // ... et jamais d'interruption !
}

do {
  continue;
  break; // jamais d'interruption...
} while(true); // ... et boucle infinie !
```

Blocs implicites

Si une unique instruction est déclarée sans bloc, le bloc est implicitement ajoutée dans les structures de contrôle précédentes :

```
if (true)
  i++;

if (true) i++;

for (; true; ) break;

while (true) break;

do break; while(true);
```

Remarque : `continue` s'utilise avec les trois formes de boucle `for`, `while` et `do-while`.

Attention, ceci rend la lecture du code plus difficile. Il vaut toujours mieux maîtriser les scopes.

Sélecteur conditionnel `switch`

La structure `switch` est une alternative à la structure `if`. Ici, la valeur testée est une valeur entière permettant de multiples branchements. Le mot clé `break` (optionnel) sert à délimiter les différents cas :

```
switch ( <valeur> ) {
case <cas> : <bloc>
...
default : <default>
}
```

```
switch (i) {
case 0 : i = 101; break;
case 1 : i = 11;  break;
case 2 : i = 1;   break;
default: i++;
}
```

Remarque : le cas `default` est optionnel (ou presque).

Regroupement de cas

Si le mot clé `break` est absent pour un cas, les instructions du cas suivant sont exécutées :

```
switch (i) {
case 0 :           // le cas 1 est executé aussi pour le cas 0
case 1 : i = 11;   break;
case 2 : i = 1;    break;
default: i++;
}
```

Remarque : cette syntaxe particulière est historique. Elle est source importante de problème. Il faut être très vigilant à l'usage.

Gestion de la mémoire

Allocation

Durant l'exécution, le programme demande au système d'allouer et ensuite de libérer de la mémoire

- sur la pile (stack) : zone mémoire pour l'allocation *automatique* (par défaut) des variables, limitée au scope, les variables sont construites (détruites) dans l'ordre exact (inverse) de leur déclaration, taille très limitée ;

```
{
  int i[] = { 1, 2, 4 }; // allocation automatique sur la pile
                         // d'un tableau de 3 entiers
  auto d = 0.0; // allocation sur la pile d'un double
} // destruction du double d puis du tableau i
```

- sur la tas (heap) : zone mémoire gérée *explicitement* par l'utilisateur par les fonctions opérateurs `new` et `delete`, virtuellement illimitée, en pratique la mémoire RAM.

Tableaux dynamiques

On peut allouer de la mémoire (sur le tas) au cours de l'execution pour un variable d'un type donné et de taille arbitraire en utilisant un opérateur qui retourne l'adresse mémoire de l'espace alloué.

En C++, l'opérateur `new` sert à allouer la mémoire et l'opérateur `delete` à la libérer. Pour un unique élément :

```
<type>* p = new <type>;
delete p;
```

```
auto* p = new double;
delete p;
```

Pour plusieurs éléments (i.e. un tableau) :

```
<type>* p = new <type>[<taille>];
delete [] p;
```

```
auto* p = new double[4];
delete [] p;
```

Remarque : le pointeur joue un rôle central dans la gestion de la mémoire, c'est le type manipulé.

Retour sur `nullptr`

La constante `nullptr` est utilisée pour initialiser les pointeurs. Il convient de toujours remettre à `nullptr` tout pointeur désalloué :

```
auto* v = new double[1000];
delete [] v;
v = nullptr;
```

Remarque : l'opérateur `delete` appliqué à un pointeur initialisé à la valeur `nullptr` est autorisé et ne crée pas d'erreur :

```
int* v = nullptr;
delete v;    // ok
delete [] v; // ok
```

Les pointeurs sont dangereux (1/2)

Un pointeur est une adresse en mémoire. Il n'y a pas de notion de taille ni de vérification des bornes à l'utilisation. Il faut donc gérer à la main les éventuels débordements :

```
auto* v = new double[1000];

v[2000] = 1.0; // ERREUR débordement !
```

Il n'est pas possible de savoir si un pointeur est désalloué (sauf si on utilise `nullptr`) :

```
auto* v = new double[1000];
delete [] v;

v[0] = 1.0; // ERREUR déjà détruit !
```

Les pointeurs sont dangereux (2/2)

Le cycle de vie de la mémoire est à gérer avec attention pour éviter les fuites mémoires. En effet, la mémoire allouée est persistante et il n'y a pas de mécanisme de *garbage collector* :

```
{
  auto* v= new double[1000];
} // sortie du bloc, le pointeur v est détruit
  // PROBLEME la mémoire est perdue !
```

Allouer sans désallouer au préalable crée également des fuites mémoires :

```
auto* v= new double[1000];

v= new double[1000]; // PROBLEME la mémoire est perdue !
```

Remarque : l'utilisation de `nullptr` est une bonne pratique et permet (au moins) de tester. La gestion de la mémoire à la main est un point dur du C++.

Tableau de pointeurs

On peut utiliser les tableaux dynamiques de pointeurs pour effectuer des tableaux multi-dimensions :

```
auto* n1 = new int[4];
auto* n2 = new int[4];
auto* n3 = new int[4];
auto* n4 = new int[4];

auto** p[4] = new int*[4];

p[0] = n1;
p[2] = n2;
p[3] = n3;
p[4] = n4;
```

Exemple pratique (1/2)

Produit de deux matrices carrées $n \times n$:

```
// A[i*n + j] row major
// A[i + n*j] column major

auto* A = new double[n*n];
auto* B = new double[n*n];
auto* res = new double[n*n];

// initialisation des matrices

// en row major
for (auto i = 0; i < n; i++) {
  for (auto j = 0; j < n; j++) {
    res[j + i * n] = 0;
    for (auto k = 0; k < n; k++) {
      res[j + i * n] += A[k + i * n] * B[j + k * n];
    }
  }
}
```

Exemple pratique (2/2)

Matrice carrée $n \times n$ à partir d'un tableau :

```
auto* raw = new double[n*n];
auto** A = new double*[n];

for(auto i = 0; i < n; ++i) {
  A[i] = raw + i*n;
}

// remplissage 1D
for(auto i = 0; i < n*n; ++i) {
  raw[i] = i+1;
}

// remplissage 2D
for(auto i = 0; i < n; ++i) {
  for(auto j = 0; j < n; ++j) {
    A[i][j] = (i+1) + j*n;
  }
}
```

Fonctions

Définition de fonctions

Une fonction est une portion de code qui exécute une série d'instructions. En général, une fonction prend des paramètres en entrée et renvoie un résultat en sortie. La définition d'une fonction est la suivante :

```
<retour> <nom> ( <paramètres>... ) { <corps> }
```

Les variables définies dans le corps d'une fonction sont locales.

Une fonction est déclarée par son prototype :

```
<retour> <nom> ( <paramètres>... );
```

Attention : En C++, les fonctions ne peuvent être définies de manière imbriquée !

Remarque : Contrairement aux structures de contrôle, le corps définit un bloc qui ne peut être implicite.

Paramètres formels et effectifs

Les arguments d'une fonction sont appelés paramètres formels. Ces paramètres peuvent être de n'importe quel type et n'importent que dans la fonction. Si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clef `void`.

Les arguments avec lesquels la fonction est appelée sont appelés paramètres effectifs. L'ordre de ces paramètres doit correspondre avec la définition de la fonction. Les paramètres peuvent être des expressions et l'ordre d'évaluation dépend du compilateur. Des conversions sont menées quand les types ne correspondent pas aux types du prototype.

Remarque : Une très bonne pratique est d'utiliser l'attribut `const` pour les paramètres formels des fonctions.

Retour de fonctions

Quand une fonction ne retourne aucun résultat, on utilise le type `void`. Pour retourner une valeur, le mot clé `return` est utilisé dans le corps de la fonction :

```
double sqr ( const double x )
{
  return x*x;
}
```

Attention : Il faut s'assurer qu'une fonction à un appel à `return` pour tout chemin d'execution.

```
double abs( const double x ) // ok return sur tous les chemins
{
  if (x < 0.0)
    return -x;
  else
    return x;
}
```

Appel de fonctions

Une fonction est appelée de la manière suivante :

<nom> (<paramètres>...);

```
auto x = sqr(4.3);
auto y = abs(-2); // conversion de int en double
```

On peut imbriquer les appels de fonctions :

```
double fun( const double x )
{
  return sqr(abs(x) - x);
}
```

Pour être utilisée, une fonction doit être définie ou déclarée :

```
double fun( const double x ) { return sqr(abs(x) - x); } // ERREUR
double abs( const double x );
double sqr( const double x );
```

Forward declaration

Il est possible de dissocier la déclaration et la définition d'une fonction. On parle de *forward declaration* :

```
// forward declarations
double abs( const double x );
double sqr( const double x );

double fun( const double x ) { return sqr(abs(x) - x); } // ok

// implémentations
double sqr ( const double x ) { return x*x; }
double abs ( const double x )
{
  if (x < 0.0)
    return -x;
  else
    return x;
}
```

Arguments par défaut

Certains arguments d'une fonction peuvent avoir des valeurs par défaut. Ces valeurs sont utilisées à la place des paramètres effectifs manquants à l'appel de la fonction :

```
void foo ( int i = 1 ); // ok
foo( 2 ); // ok
foo();    // i.e. g ( 1 )

void bar ( int i, int j = 1 ); // ok
bar( 5, 8 ); // ok
bar( 3 );    // i.e. f( 3, 1 )
```

Attention : les arguments par défaut doivent être positionnés à la fin de la liste des paramètres :

```
void foo ( int i = 1, int j );        // ERREUR
void foo ( int i, int j = 1, int k ); // ERREUR
```

Appel par valeur

Un appel de fonction par valeur consiste à recopier les valeurs des arguments passés à la fonction dans les paramètres formels de la fonction :

```
int pow4( int i ) // paramètre non const
{
  // la valeur de i est explicitement changée
  i *= i; // i^2
  i *= i; // i^4
  return i;
}

auto i = 2;
auto j = pow4(i); // i = 2 et j = 16
```

Dans ce mode, le changement des paramètres dans la fonction n'ont pas d'effet sur le paramètre passé en argument.

Appel par référence

Un appel de fonction par référence consiste à recopier les références des arguments passés à la fonction dans les paramètres formels de la fonction :

```
int pow4( int& i ) // paramètre référence non const
{
  // la valeur de i est explicitement changée
  i *= i; // i^2
  i *= i; // i^4
  return i;
}

auto i = 2;
auto j = pow4(i); // i = 16 et j = 16
```

Dans ce mode, le changement des paramètres dans la fonction ont un effet sur le paramètre passé en argument.

Remarque : D'un certain point de vue, le paramètre peut être considéré comme un retour de la fonction.

Appel par pointeur

Un appel de fonction par pointeur consiste à recopier les adresses des arguments passés à la fonction dans les paramètres formels de la fonction :

```
int pow4( int* i ) // paramètre pointeur non const
{
  // la valeur de i est explicitement changée
  *i *= *i; // i^2
  *i *= *i; // i^4
  return *i;
}

auto i = 2;
auto j = pow4(&i); // i = 16 et j = 16
```

Ce mode est similaire au mode par référence.

Fonction principale `main`

La fonction principale `main` est la première fonction appelée par le système d'exploitation dans le programme. Chaque programme doit impérativement avoir exactement une fonction principale. C'est toutefois une fonction comme les autres.

En principe, seuls les codes contenus dans la fonction `main` et les fonctions appelées directement ou indirectement dans `main` seront exécutées. La fonction `main` peut être implémentée sans argument et doit retourner un entier :

```
int main ()
{
  std::cout << "hello world!" << std::endl;
  return 0;
}
```

La valeur de l'entier retourné est transmise à l'environnement d'exécution. La valeur zéro est utilisée en standard pour signaler qu'il n'y a pas d'erreur d'execution.

Appels récursifs

En C++, les fonctions peuvent être récursives, c'est-à-dire qu'elles sont autorisées à s'appeler elle-même dans leur définition :

```
int symmetric_factorial ( const int n )
{
  if (n == 0) {
    return 0;
  } else if (n > 1) {
    return n * symmetric_factorial(n - 1);
  } else if (n < 0) {
    return -symmetric_factorial(-n);
  } else {
    return 1;
  }
}
```

Remarque : la profondeur de la recursion est limitée par la taille de la pile.

Surcharge de fonctions

En C++, il est possible de définir des fonctions ayant le même nom, mais ayant une liste de paramètres différents. C'est le début d'une forme de polymorphisme :

```
int    symmetric_factorial ( const int    x );
float  symmetric_factorial ( const float  x );
double symmetric_factorial ( const double x );
```

Attention, le type de retour n'est pas discriminant :

```
         int foo ( const int x );
unsigned int foo ( const int x ); // ERREUR
```

Attention, les défauts n'impactent pas les listes de paramètres :

```
void foo ( const int i );
void foo ( const int i = 1); // ERREUR
```

Fonction `inline`

L'extension `inline`, ou *inlining*, est une optimisation d'un compilateur qui remplace un appel de fonction par le code de cette fonction.

source : `https://fr.wikipedia.org/wiki/Extension_inline`

Lorsque la fonction est petite, l'*inlining* peut augmenter énormément les performances :

```
inline double sqr ( const double x )
{
  return x*x;
}
```

Remarque : le mot clé `inline` est une indication pour le compilateur et non une obligation.

Attention, il n'y a pas de garantie, l'*inlining* peut être parfois contre productif, engendrer du code plus lent, un binaire plus gros ou n'avoir aucun effet.

Déduction automatique du type de retour

Au fil des normes, la déclaration de fonction a évolué vers la déduction calculée puis automatique du type de retour en utilisant les mots clés `auto` et `decltype` :

```
double sqr ( const double x ) { return x*x; }

// C++11 syntaxe intermédiaire
auto sqr ( const double x ) -> double         { return x*x; }
auto sqr ( const double x ) -> decltype(x*x) { return x*x; }

// C++14 syntaxe finale
auto sqr ( const double x ) { return x*x; }
```

Remarque : `auto` n'est pas (encore) autorisé dans la liste des paramètres formels. Nous verrons d'autres mécanismes.

Retour sur les namespaces

Comme pour les variables, une fonction peut appartenir à un namespace :

```
namespace foo {
  void bar();
}

foo::bar(); // ok
```

Le mot clé `using` permet d'accéder aux éléments du namespace :

```
using namespace foo;

bar(); // ok
```

Fonction anonyme

> Dans les langages de programmation fonctionnelle, les fonctions anonymes sont souvent appelées fonctions lambda, en référence au lambda-calcul, outil théorique dans lequel on programme tout sous forme d'appel de fonction, alors même qu'aucune fonction de base n'est définie.
>
> source : `https://fr.wikipedia.org/wiki/Fonction_anonyme`

Les fonctions anonymes sont des fonctions inline crées directement dans le code. La définition d'une fonction anonyme est la suivante :

```
[capture] ( <paramètres>... ) -> <retour> { <corps> }
```

```
[capture] ( <paramètres>... ) { <corps> }
```

Le type concret d'une fonction anonyme n'est pas défini par le standard, on utilise le mot clé `auto`.

Définition d'une fonction anonyme

L'utilisation d'une fonction anonyme est similaire à un objet fonction, i.e. à un *fonctor* :

```
auto sqr = [] (const double x) { return x*x; };

auto x2 = sqr(3.0);
```

La clause `capture` définit quelles variables extérieures sont utilisables dans la fonction. Une clause vide `[]` signifie ne rien utiliser.

Comme les variables locales, les fonctions anonymes sont déclarées dans les blocs. Ces fonctions ont une portée liée au scope du bloc, c'est-à-dire qu'elles sont crées et initialisées à leur définition et automatiquement détruite à la sortie du bloc.

```
{
  auto sqr = [] (const double x) { return x*x; };
} // sortie de bloc, sqr est détuite
```

Capture

Il existe deux modes de capture généralisée :

- `[=]` toutes les variables sont capturées par copie
- `[&]` toutes les variables sont capturées par référence

mais il est possible de spécifier par variable :

- `[x]` la variable `x` est capturée par copie
- `[&x]` la variable `x` est capturée par référence

Attention, il ne faut pas répéter les variables :

```
[x, y] // OK
[x, x] // ERROR x est répété
[&, x] // OK
[&,&x] // ERROR &x inclus dans &
[=, x] // ERROR x inclus dans =
[=,&x] // OK
```

Imbrication

Il est possible d'imbriquer les fonctions anonymes :

```
auto f =   [](int y) { return 2 * y; };
auto g = [&f](int x) { return f(x) + 1; };

auto r = g(5); // r vaut 11
```

```
auto g = [](int x) {
  auto f = [](int y) { return 2 * y; };
  return f(x) + 1;
};

auto r = g(5); // r vaut 11
```

```
auto r = [](int x) {
  return [](int y) {
    return y * 2;
  }(x) + 1;
}(5); // r vaut 11
```

Paramètres génériques

En utilisant le mot clé `auto`, les fonctions anonymes peuvent accepter des paramètres *génériques*. Le compilateur crée alors l'opérateur d'appel en fonction des paramètres :

```
auto sqr = [] (const auto x) { return x * x; };

auto r1 = sqr(1);   // r1 est int
auto r2 = sqr(1.0); // r2 est double
```

Chaque instance d'`auto` dans une liste de paramètres est équivalente à un paramètre de type distinct.

Attention, la validité est à la charge de l'utilisateur : non utilisé implique non compilé implique non validé !

L'approche historique de la généricité

Les *templates* (modèles ou patrons) sont à la base de la généricité en C++. L'idée est d'écrire des modèles génériques de fonction permettant de générer des fonctions concrètes.

Le mot clé `template` est utilisé pour définir une liste de types génériques paramétrant la fonction *template* :

`template<typename <type>...>`

```
template<typename T>
T sum(T a, T b) { return a + b; }
```

Si on utilise la fonction en utilisant deux paramètres effectifs de type `int`, le modèle est *spécialisé* de la manière suivante :

```
sum(1, 2); // int sum(int a, int b) { return a + b; }
```

Spécialisation explicite

Attention, il n'y a pas de conversion implicite de type mais une substitution stricte des paramètres effectifs, i.e. si on donne un `int` et un `double`, la fonction n'est pas définie !

```
sum(1, 2.5); // ERREUR  int != double !
```

On peut spécialiser explicitement si ambiguïté :

```
sum<double>(1, 2.5); // r = 3.5 ok, int converti en double
```

Pour répondre au problème de types multiples, on peut introduire un paramètre template pour chaque argument :

```
template<typename T, typename U>
T sum(T a, U b) { return a + b; }
```

Promotion de type

Le type de retour peut amener à une conversion problématique :

```
auto r = sum(1, 2.5); // r = 3, decltype(r) est int !
```

Une solution est l'utilisation de `auto` :

```
template<typename T, typename U>
auto sum(T a, U b) { return a + b; }

auto r = sum(1, 2.5); // r = 3.5 ok, decltype(r) = decltype(1+2.5)
```

On verra dans la suite du cours comment répondre à ce problème.

Fonctions mathématiques

Par héritage du C, le header <cmath> (ou <math.h>) déclare un ensemble de fonctions pour effectuer les opérations numériques classiques :

- trigonométrie : `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, etc.
- exponentiel & logarithme : `exp`, `log`, etc.
- puissance : `pow`, `sqrt`, `cbrt`, etc.
- valeur absolue : `abs`, etc.
- arrondi : `ceil`, `floor`, etc.

Remarque, les fonctions sont dans l'espace de nommage `std`.

```
auto x = std::sqrt(3.14);

auto sigmoid = 1. / (1. + std::exp(x));
```

Assertion

Par héritage du C, le header <cassert> (ou <assert.h>) déclare une fonction d'assertion pouvant être utilisée pour le débug :

```
void assert (int e);
```

Si la valeur de l'expression e est égale à 0 (i.e. l'expression est *false*), un message d'erreur est écrit et le programme se termine.

```
#include <assert.h>

int* p = nullptr;
assert (p == nullptr); // ok
int i = 3;
assert (i == 3); // ok
assert ( (i == 3) && (p == nullptr) ); // ok
assert (false); // ERREUR
```

Remarque, les assertions sont désactivées si la macro NDEBUG est définie.

Bases de la modélisation orientée objet

Objet en modélisation

Un objet représente une simplification de la réalité caractérisé par

- une identité distincte
- un état enregistré dans ses données internes
  - collection de variables privées
- un comportement décrit par ses méthodes
  - définit ses actions et réactions aux messages

Messages et méthodes

Un message indique le comportement que doit exécuter un objet

- c'est l'unique façon de communiquer avec un objet
- vision utilisateur !

Une méthode indique la réponse à un message

- implémentation cachée aux utilisateurs
- vision concepteur !

Les bénéfices de l'encapsulation

On a vu que l'objet encapsule ses données et ses traitements

- l'utilisateur ne connaît qu'une interface d'utilisation et fait abstraction de son implémentation
- l'objet est fiable car il maintient sa cohérence en contrôlant l'accès à ses données
- L'objet peut évoluer sans impacter ses utilisateurs en ne modifiant que son implémentation

L'encapsulation de l'état et du comportement facilite la réutilisabilité et la robustesse

Classe et instance

Une classe est une famille d'objets possédant les mêmes propriétés

- mécanisme de groupage
- permet de modéliser un concept
- différents niveaux de détails raffinés tout au long des activités
  - Classe conceptuelle d'analyse simple
  - Classe d'analyse détaillée
  - Classe de conception détaillée
- développée par des analystes, concepteurs et des développeurs

Un objet est une instance d'une classe

- chaque instance a sa propre identité et son propre état
- les instances d'une classe partagent le même comportement
- les instances peuvent partager l'état de la classe

Agrégation et composition

L'agrégation est une relation exprimant un lien entre instances :

- relation de type « ensemble / élément » ;
- l'élément agrégé peut être partagé ;
- association non symétrique.

La composition est une agrégation forte :

- description de la notion de « composite »
- le cycle de vie des composants et de l'agrégat sont liés ;
- un composant ne peut être lié qu'à un seul agrégat.

L'agrégation et la composition sont des vues subjectives !

Notion d'héritage

L'héritage permet la réutilisation de l'état et du comportement d'une classe par d'autres classes, acquisition automatique des biens (et des dettes)

On appelle super-classe une classe contenant les éléments communs à un ensemble de sous-classes

- Les sous-classes héritent toutes les variables et méthodes de la super-classe
- Les sous-classes raffinent, spécialisent la définition de leur super-classe
  - ajout de variables
  - ajout de méthodes
  - redéfinition de méthodes

L'héritage permet la classification des objets.

Relation de généralisation et spécialisation

La généralisation est une relation hiérarchique entre classes

- indique qu'une classe est plus générale qu'un autre
- au niveau conceptuel, traduit la relation « est un »
- au niveau de l'implémentation, peut être un mécanisme de factorisation
- relation descendante

Il convient alors de séparer les modèles

- logique : abstrait et simple, rendant compte du système
- d'implémentation : décrivant comme le code est structuré

La spécialisation consiste à enrichir la définition d'une classe

- extension par réutilisation
- relation ascendante

Notion d'interface

Une classe est dite abstraite si elle n'est pas instanciable

Une interface fournit une vue complète d'un ensemble de services offert par une classe

- joue un rôle de contrat
- définit un ensemble de méthodes publiques et abstraites
- mise en œuvre du concept de polymorphisme

Une classe implémentant une interface doit définir les méthodes de l'interface

- redéfinition totale : la classe est concrête
- redéfinition partielle : la classe est abstraite

La relation de réalisation a lieu entre une interface et la classe

- la classe d'implémentation réalise le contrat de l'interface
- l'interface définit un contrat garanti par la classe d'implémentation

Classes

Définition

En C++, le mot clé `class` permet de définir la notion de classe :

```
class <nom> {
<visibilité>:
  <données>
  <méthodes>
};
```

où :

- les *données* (i.e. variables ou attributs) et les *méthodes* (i.e. fonctions) sont les membres de la classe,
- la visibilité s'applique sur les membres, en determine l'accès par les utilisateurs des objets, mais aussi par les méthodes,
- des méthodes pour la construction et la destruction des objets de la classe sont définies.

Visibilité

La visibilité des membres peut être changée avec les mots clés :

- `public` : accès sans restriction,
- `protected` : accès uniquement par les fonctions membres et par les types dérivés,
- `private` : accès uniquement par les fonctions membres.

Attention, par défaut, la visibilité des membres de classe est *privée*. Ne rien mettre implique une classe inutilisable !

Remarque, la visibilité s'applique également lors de l'héritage.

```
class point_2d {
public:
  point_2d (); // constructeurs
  point_2d (const double x, const double y);
  ~point_2d (); // destructeur
  double norm (); // méthode
public: // attention, données publiques
  double m_x, m_y; // données membres ou attributs
};
```

Définition des méthodes

Une méthode de classe est une fonction ayant accès implicitement aux données membres de la classe (et aux autres méthodes) :

```
class point_2d
{
...
  double norm () {
    return std::sqrt(m_x*m_x + m_y*m_y);
  }
...
};
```

Pour l'implémentation hors du corps, on préfixe par le nom de la classe :

```
void point_2d::norm ()
{
  return std::sqrt(m_x*m_x + m_y*m_y);
}
```

Constructeurs & Destructeurs

Des méthodes *spécifiques* de classes pour leur gestion sont définies :

- constructeurs : fonctions automatiquement appelées à la construction des objets (i.e. par `new`). Un constructeur porte le nom de la classe :

```
point_2d ();
point_2d (const double x, const double y);
```

- destructeur : fonction automatiquement appelée à la destruction des objets (i.e. par `delete`). Le destructeur est unique et préfixé par ~ :

```
~point_2d ();
```

Remarque, constructeurs et destructeurs ne renvoient pas de type (y compris `void`), les destructeurs n'ont pas d'argument.

Constructeurs par défaut

Par défaut, deux constructeurs sont générés par le compilateur :

- le constructeur par défaut, sans argument, ne fait rien si non implémenté par l'utilisateur :

```
point_2d (); // constructeur par défaut
```

- le constructeur par copie, avec un argument référence constant du même type que la classe, dans le but de construire par *recopie* à partir d'un objet :

```
point_2d (const point_2d& p); // constructeur par recopie
```

Attention, les membres doivent être recopiable ! Dans le cas de pointeurs, seul la valeur du pointeur est recopié, non le contenu de l'adresse ! C'est très dangereux !

Gestion des constructeurs

Si un constructeur est défini par l'utilisateur, le constructeur par défaut n'est pas généré. On peut utiliser le mot clé `default` pour forcer la génération. De même, pour supprimer un constructeur, on utilise le mot clé `delete` :

```
class point_2d {
public:
  point_2d () = default; // construction par défaut
  point_2d (const point_2d& v) = delete; // pas de recopie
  point_2d (const double x, const double y);
...
};
```

Attention : le constructeur par défaut généré ne peut pas initialiser les membres constants (voir dans la suite).

Remarque : le constructeur par copie est par contre toujours généré

Liste d'initialisation

La liste d'initialisation suit immédiatement la signature du constructeur et initialise les données membres :

```
point_2d ()
  : m_x(0)
  , m_y(0) {}

point_2d (const double x, const double y)
  : m_x(x)
  , m_y(y) {}
```

Les constructeurs peuvent s'imbriquer par délégation dans la liste d'initialisation :

```
point_2d ()
  : point_2d(0,0) {} // constructeur point_2d (double x, double y)
```

Pourquoi la liste d'initialisation ?

La liste d'initialisation n'est pas obligatoire (sauf pour la délégation) :

```
point_2d (const double x, const double y)
{ // m_x et m_y initialisés par défaut
  m_x = x; m_y = y; // puis on recopie
}
```

Il faut éviter les constructions et affectations qui peuvent nuire aux performances et sont parfois difficiles à voir.

Attention, on peut vouloir factoriser du code et créer les mêmes problèmes :

```
point_2d (const double x, const double y)
{
  scale(x,y);
}
```

Remarque, la liste d'initialisation répond au principe d'initialisation RAII, à utiliser systématiquement.

Création d'objets

Un objet de classe est crée par l'appel au constructeur :

```
point_2d p; // création d'un objet par constructeur par défaut
```

Le destructeur est automatiquement appelé à la fin du scope du bloc initialisant l'objet :

```
{
  point_2d p (1.0, 2.0); // création d'un objet
  ...
} // sortie du bloc, appel au destructeur de p
```

Un pointeur peut être initialisé par l'opérateur new et détruit par delete. Pour un tableau d'objets, le constructeur par défaut doit être défini :

```
auto* p1 = new point_2d (1.0, 2.0); // création par un constructeur
delete p1; // appel au destructeur
auto* p2 = new point_2d [3]; // appel 3 fois du constructeur par défaut
delete [] p2; // appel 3 fois du destructeur
```

Appel des méthodes

Les méthodes publiques sont appelées en utilisant l'opérateur `.` :

```
point_2d p (1.0, 2.0);
auto n = p.norm();
```

Pour les pointeurs d'objet, les méthodes sont appelées soit en déréférençant par l'opérateur `*` et en utilisant l'opérateur `.`, soit en utilisant l'opérateur `->` (qui forme un accès simplifié) :

```
auto* p = new point_2d (1.0, 2.0);
auto n1 = (*p).norm();
auto n2 =   p->norm();
```

Attention, pour être accédée, la méthode doit être publique !

Accès aux données

De manière similaire aux méthodes, les données membres publiques d'objet sont appelées en utilisant l'opérateur `.` :

```
point_2d p (1.0, 2.0);
auto x = p.m_x;
```

L'accès aux données à partir d'un pointeur est similaire aux méthodes :

```
auto* p = new point_2d (1.0, 2.0);
auto x1 = (*p).m_x;
auto x2 =   p->m_x;
```

Attention, une donnée publique est un risque sérieux pour l'intégrité des objets. Une donnée publique doit être constante (voir la suite) !

Structure

Une *structure* est une classe particulière dont la visibilité est par défaut à publique. Il est possible (comme pour les classes) d'initialiser les variables directement à la déclaration :

```
struct point_2d {
  // données et méthodes publiques
  double norm() const { std::sqrt(x*x + y*y); }
  double x = 0;
  double y = 0;
};
```

Comme les tableaux, les structures ont une initialisation uniforme :

```
auto p = point_2d { 0.0, 0.0 };
p.y = p.x = 1.0;
auto r = p.norm();
```

Remarque, les structures sont des classes légères publiques.

const-correctness des méthodes

Les méthodes d'une classe qui ne changent pas les données membres peuvent être déclarées constantes par le mot clé `const` :

```
class point_2d {
public:
...
  double norm () const; // méthode const
...
};
```

Le compilateur vérifie que les données membres sont inchangées par la méthode, c'est un gain de robustesse :

```
double point_2d::norm() const
{
  auto n = std::sqrt(x*x + y*y);
  if (n < 1.e-10)
    n = y = x = 0; // ERREUR x et y constant
  return n;
}
```

const-correctness des données

Les données membres qui ne changent pas durant toute la vie des objets doivent être déclarées constantes par le mot clé `const` :

```
class point_2d
{
...
  const double x, y;
};
```

Ici encore, exprimer des invariants apporte de la robustesse :

```
double point_2d::norm() // non const
{
  auto n = std::sqrt(x*x + y*y);
  if (n < 1.e-10)
    n = y = x = 0; // ERREUR x et y constant
  return n;
}
```

const-correctness des objets

Les objets peuvent être déclarés constants comme les types simples par le mot clé `const` :

```
const point_2d p (1.0, 2.0);
```

Dans ce cas, seules les méthodes déclarées constantes sont appelables avec des objets déclarés constants :

```
const point_2d p (1.0, 2.0);

auto n = p.norm(); // ok si norm() const sinon ERREUR
```

Remarque, déclarer les méthodes, objets et membres constants prévient de nombreux bugs.

Pointer `this`

A l'intérieur d'une méthode de classe, un pointeur vers l'instance de la classe appelé `this` est accessible :

```
point_2d& p = *this;
```

- C'est un pointeur constant,qui est appelé de manière transparente à chaque appel de méthode,
- Il permet d'accéder aux données membres :

```
double point_2d::norm () const
{
  auto x = this->m_x;
  auto y = this->m_y;
  return std::sqrt(x*x + y*y);
}
```

Remarque, ce pointeur est utile quand on veut retourner une référence de l'objet. Il permet également de lever des ambiguïté liées à l'héritage.

Données et méthodes de classe

Le mot clé `static` permet de définir des données et méthodes appartenant à la classe et non aux objets :

```
class point_2d {
public:
...
  // éléments de classe partagés
  static const double origin_x = 0.0;
  static const double origin_y = 0.0;
  static double norm (const double x, const double y);
  double norm() { return norm(m_x, m_y); }
...
};
```

Les appels se font en préfixant par la classe :

```
auto ox = point_2d::origin_x;
auto oy = point_2d::origin_y;

auto r = point_2d::norm(ox, oy);
```

Surcharge d'opérateurs

En C++, les opérateurs +, -, *, /, =, [], () (et d'autres) peuvent être surchargés dans les classes :

```
struct point_2d {
...
  point_2d& operator+= (const point_2d& p)
  {
    m_x += p.m_x; // m_x non const
    m_y += p.m_y; // m_y non const
    return *this; // référence vers l'instance
  }
  point_2d operator+ (const point_2d& p) const
  {
    return point_2d(x+p.x, y+p.y); // nouvel objet
  }
...
};
```

C'est un fantastique moyen d'enrichissement des objets et de personnalisation de comportement.

Attention aux objets temporaires

Certains opérateurs engendrent la création d'objets temporaires. Ces objets impliquent des constructions, destructions, éventuellement copies, allocations, etc :

```
heavy_object operator+ (const heavy_object& u, const heavy_object& v)
{
  heavy_object h(u); // peut être des allocations
  h += v;            // peut être beaucoup d'opérations (ok)
  return h;          // peut être des recopies
}
```

Attention, il faut être très vigilant quant à la surcharge.

Remarque, il existe des techniques de programmation générique pour pallier les problèmes de performance.

Return Value Optimization

Optimisation du compilateur pour éliminer les objets temporaires en retour de fonction :

```
heavy_object foo()
{
  heavy_object h; // création d'un objet local
  h.do_something();
  return h;
}

auto h = foo ();
```

équivaut à :

```
void foo (heavy_object& h) { h.do_something(); }

heavy_object h;
foo (h);
```

Remarque, cette optimisation dépend du compilateur.

Affectation

L'opérateur d'affectation = est un opérateur très utilisé pour les conversions :

```
struct point_2d {
...
  point_2d& operator= (const point_2d& p)
  {
    m_x = p.m_x;
    m_y = p.m_y;
    return *this;
  }
  point_2d& operator= (const double d) const
  {
    m_x = m_y = d;
    return *this;
  }
...
};
```

Classe et argument de fonction

Les classes sont des variables qui peuvent naturellement être des arguments de fonctions, passées par valeur ou référence.

Quand un appel par valeur est utilisé, une copie de la classe est crée :

```
void foo (const point_2d p); // recopie
```

Pour des classes volumineuses (au sens mémoire), il peut y avoir un grand surcoût, surtout si de nombreux appels sont effectués.

Il faut privilégier l'appel par référence pour les arguments de type classe dans les fonctions :

```
void foo (const point_2d& p); // référence, pas de copie
```

Remarque, le passage par référence des classes est un argument de performance.

Résumé de bonnes pratiques

Pour respecter les principes *Resource Acquisition Is Initialisation* :

- les constructeurs créent les ressources nécessaires et le destructeur les libère,
- les listes d'initialisation des constructeurs limitent les zones d'instabilité durant la construction,
- Tout objet construit doit être utilisable !
- Ne pas avoir d'objet à état instable.

D'autre part :

- Le mot clé *const* apporte de la sécurité et de la robustesse,
- Le passage des objets par référence évite les problèmes de performances liées aux recopies,
- Il faut être vigilant à la création de temporaires lors des surcharges d'opérateurs.

Héritage

L'héritage est une technique qui permet de construire une classe à partir d'un modèle, i.e. une autre classe. On évite de ainsi réécrire du code et on capitalise les développements / validation / maintenance.

```
class point {
public:
  point(const unsigned dimension)
    : m_dimension(dimension) {}
  int dimension() const { return m_dimension; }
private:
  unsigned m_dimension;
};
```

```
class point_2d : public point {
public:
  point_2d() : point(2) {}
};
```

```
class point_3d : public point {
public:
  point_3d() : point(3) {}
};
```

L'héritage suit les règles de visibilité (*privé* par défaut)

Polymorphisme

Les méthodes définies dans la super-classe sont héritées aux classes dérivées :

```
point_2d p2;
auto dim2 = p2.dimension();

point_3d p3;
auto dim3 = p3.dimension();
```

Tout objet d'une classe dérivée est aussi un objet de la classe de base :

```
point* p;
p = &p2; p->dimension();
p = &p3; p->dimension();
```

On peut ainsi écrire des algorithmes indépendants des types dérivées.

Redéfinition de méthode

Certains comportements peuvent être redéfinis dans les classes dérivées :

```
class base {
public:
  base() {}
  std::string name() {
    return "base";
  }
};
```

```
class derived : public base {
public:
  derived() {}
  std::string name() {
    return "derived";
  }
};
```

Ici le comportement dépend de l'objet :

```
derived d;
std::cout << d->name() << std::endl; // derived

base& b = d;
std::cout << b->name(); // base
```

Méthode virtuelle

En utilisant le mot clé `virtual`, le comportement de la classe dérivée surcharge celui de la classe de base :

```
class base {
public:
  base() {}
  virtual std::string name() {
    return "base";
  }
};
```

```
class derived : public base {
public:
  derived() {}
  std::string name() {
    return "derived";
  }
};
```

Ici le comportement dépend de l'objet de classe dérivée :

```
derived d;
std::cout << d->name() << std::endl; // derived

base& b = d;
std::cout << b->name(); // derived
```

Destructeur virtuel

Corollaire : le constructeur de la classe de base doit être virtuel !
Sinon, si on détruit un objet de base, le destructeur de la classe dérivée non appelé :

```
class base {
public:
  base() {}
  ~base() {}
};

class derived : public base {
public:
  derived() {}
  ~derived() { std::cout << "derived dtor\n"; }
};

base*  b = new derived;
delete b; // Pas de message, possible fuite mémoire !
```

Méthode virtuelle pure

Certains comportements ne peuvent être définis que dans les classes dérivées. On définit alors une méthode *virtuelle pure*, i.e. sans définition :

```
class base {
public:
  base() {}
  virtual std::string name() = 0; // virtuelle pure
};
```

Ici le comportement dépend de l'objet de classe dérivée :

```
base* b = new derived;
std::cout << b->name(); // derived
```

La classe `base` est une classe dite *abstraite*, on ne peut l'instancier.

Attention : pour s'y retrouver dans les héritages, le programme garde une table virtuelle consultée à chaque appel. Les méthodes virtuelles ne doivent pas être appelées intensivement !

Vérification de surcharge

Le mot clé `override` permet :

- d'exprimer qu'une méthode surcharge une méthode virtuelle,
- d'enclencher de fait la vérification du compilateur :

```
class base {
public:
  virtual void foo() const = 0;
};

class derived : public base {
public:
  void foo() const override; // ok
};

class other : public base {
public:
  void foo() override; // ERREUR la méthode n'est pas const
};
```

Conversion dynamique

La conversion d'un objet de type dérivée en type de base est implicite, on parle de *upcast* :

```
point_2d p2;

point& p = p2; // upcast
```

Le mot clé `dynamic_cast` permet de convertir avec vérification les pointeurs et références de classe dans la hiérarchie d'héritage, on parle de *downcast* :

```
point_2d p2;

point& p = p2; // upcast, dynamic_cast possible mais non nécessaire

auto& p2 = dynamic_cast<point_2d&>(p); // downcast
```

Interface

En C++, il n'y a pas de mot clé pour définir une interface mais on peut utiliser les classes abstraites et se donner des contraintes de conception :

- toutes les méthodes sont virtuelles pures,
- il n'y a pas de donnée membre,
- le constructeur est protégé (peu important en pratique mais conceptuellement à faire).

```
class interface {
protected:
  interface() {} // force l'héritage
public:
  virtual ~interface() {}
  virtual void foo() = 0;
  ...
  // Pas de données !!
};
```

Classe générique

A la façon des fonctions, il est possible de définir une classe générique en utilisant les mots clés `template` et `typename` :

```
template<typename T>
class point_2d {
public:
  point_2d ();
  point_2d (const T x, const T y);
  ~point_2d ();
  T norm ();
private:
  T m_x, m_y;
};
```

que l'on instancie comme suit :

```
point_2d<float> p2f;
point_2d<double> p2d;
```

Attention, uniquement ce qui est déclaré est compilé !

Template de valeur

Les templates peuvent aussi être des valeurs entières ou booléens :

```
template<typename T, int N>
class point {
public:
  point ();
  point (const T x[N]);
  ~point ();
  T norm ();
private:
  T m_x[N];
};
```

que l'on instancie comme suit :

```
point<float,5> p5f;
point<double,5> p5d;
```

Attention, uniquement ce qui est déclaré est compilé !

Spécialisation de classe

Il est possible de spécialiser (même partiellement) les classes template pour définir des spécificités pour des types (ou valeurs) particuliers :

```
template<typename T, typename V, int I>
class A {};

// Spécialisations

template <class T1, class T2, int I>
class A<T1*, T2, I> {};

template <class T, int I>
class A<T, T*, I> {};

template <class T>
class A<int, T*, 5> {};

template <>
class A<int, double*, 3> {};
```

Méthode et spécialisation

La définition des méthodes d'une classe template nécessite le mot clé `template` et les arguments :

```
template<typename T, int N>
T point<T,N>::norm() const
{
  T n = T(0);
  for(auto i = 0; i < N; ++i)
    n += m_x[i]*m_x[i];
  return std::sqrt(n);
}
```

On peut également spécialiser les fonctions membres mais pas partiellement :

```
template<>
double point<double,1>::norm() const
{
  return std::abs(m_x[0]);
}
```

Les exceptions (en deux mots)

Les exceptions

En général, les codes de retour pour le succès ou les erreurs :

- sont faciles à oublier,
- occupent le retour des fonctions ou à défaut sont un paramètre de fonction à ajouter,
- sont à la charge du développeur (gestion et propagation).

En C++, les exceptions sont une solution de qualité. Le mécanisme consiste en l'envoi d'objets d'exception au travers de la pile d'appel du code par approche `try` / `catch` :

```
try {
  <bloc>
} catch(<exception>) {
  <bloc erreur>
}
```

Le mot clé `throw` est utilisé pour *envoyer* les objets exception.

Un exemple simple

```
class exception { // classe exception utilisateur
public:
  const char* error;
};

void throw_an_exception () { throw exception { "exception..." }; }

int main() {
  try { // bloc où une exception est éventuellement lancée
    throw_an_exception ();
  }
  catch (exception& e) { // une exception utilisateur est attrapée
    std::cout << e.error << std::endl;
    // on traite l'erreur
  }
  catch(...) { // une autre exception est attrapée
    std::cout << "exception inconnue" << std::endl;
    throw; // on remonte l'exception
  }
}
```

STL (en deux mots)

La bibliotèque standard

La *Standard Template Library* est une bibliothèque normalisée mise en œuvre à l'aide des templates fournissant (non exhaustif) :

- des fonctionnalités du `C` (maths, assertion, string, etc.),
- des générateurs de nombre aléatoire,
- des flux pour entrées / sorties,
- des outils de gestion mémoire,
- des conteneurs et algorithmes génériques,
- des mécanismes de programmation concurrente,
- des *timers* précis, etc.

Important, bien connaître la libraire standard est la base !

Attention, bibliothèque en pleine mutation ! Les normes futures du `C++` visent à y introduire de nombreuses fonctionnalités.

Remarque, le projet `boost` est considéré comme l'incubateur de la librairie (voir `www.boost.org`).

Flux I/O

La librairie standard fournit les outils de manipulations des chaines de caractères et des entrées/sorties :

- `<iostream>` : définit les flux standards `std::cin`, `std::cout` et `std::cerr`,
- `<iomanip>` : définit les services pour paramétrer les flux comme par exemple `std::setw` ou `std::setprecision`,
- `<fstream>` : définit les services pour manipuler les fichiers.

Outils de base pour les entrées / sorties :

```
extern std::ostream cout;
extern std::istream cin;
```

Approche par opérateur de flux « et » :

```
std::cout << "name ?" << std::endl;
std::string name;
std::cin >> name;
std::cout << "name is " << name << std::endl;
```

Les opérateurs de flux

Les opérateurs de flux sont des fonctions comme les autres :

```
std::cout << "ok";
```

équivaut à l'appel :

```
operator<<(std::cout, "ok");
```

Par exemple :

```
auto& o  = operator<<(std::cout,"hello ");
auto& o2 = operator<<(o,"world!");
std::endl(o2);

int a = 1, b = 2;
int c = operator+(a,b); // erreur sur les types simples
```

Surcharge des opérateurs de flux

Les opérateurs de flux peuvent être surchargés, par exemple pour l'affichage de classe :

```
struct A {
  std::string name() { return "A"; }
};

std::ostream& operator<<(std::ostream& nout, const A& a)
{
  nout << a.name();
  return nout;
}

A a;

std::cout << a << std::endl;
```

Pointeurs intelligents

Le cycle de vie des variables est un point critique à gérer dans un application. Le C++ permet une gestion fine de la mémoire dynamique assez brute.

Le C++ dispose de pointeurs *intelligents* (ou `smart`) implémentés avec les templates compatibles *RAII*. Principalement, deux catégories de pointeurs sont disponibles :

- `std::unique_ptr<T>` : ressource non partageable, libération automatique de la mémoire à la sortie du scope
- `std::shared_ptr<T>` : ressource partageable avec comptage de référence, libération automatique de la mémoire quand plus de référence

```
std::shared_ptr<int> p = std::make_shared<int>(5);
```

Important : Ne plus écrire de `new` et `delete` !

Conteneurs

Le C++ dispose de nombreux conteneurs génériques (non exhaustif) :

- les vecteurs `std::vector<T>`, `std::valarray<T>`
- les listes chainées et piles `std::list<T>`, `std::stack<T>`, `std::deque<T>`
- les tableaux associatifs type arbre binaire `std::map<Key,T>`, `std::set<T>` ou non ordonnés `std::unordered_map<Key,T>`, `std::unordered_set<T>`

```
#include <list>
std::list<double> l; // liste
l.push_back(4.0);

#include <map>
std::map<std::string, int> m; // tableau associatif string <-> int
m["ok"] = 3;

#include <vector>
std::vector<int> v(3); // vecteur de taille 3
v[1] = 1;
```

Itérateurs

Les conteneurs sont basés sur la notion d'itérateur pour leur parcours. Les itérateurs ont une syntaxe de pointeurs, peuvent se comparer (==, !=) et s'incrémenter (++, -) :

```
std::vector<int> v(5, 3); // i.e. [ 3 3 3 3 3 ]

auto sum = 0;
for(auto it = v.begin(); it != v.end(); ++it) {
  sum += *it; // syntaxe pointeur
}
```

Tout conteneur ayant une méthode `begin()` et `end()` peut être énuméré de manière simplifié (par *range-based loop* depuis C++11) :

```
std::vector<int> v(5, 3);

auto sum = 0;
for(auto& s : v) { // range-based loop
  sum += s;
}
```

Algorithmes

De nombreux algorithmes sont disponibles (non exhaustif) dans le header `<algorithm>` :

- tri `std::sort`, `std::stable_sort`, `std::partial_sort`
- recherche `std::binary_search`, `std::find`, `std::lower_bound`
- modification `std::copy`, `std::fill`, `std::replace`
- parcours et selection `std::for_each`, `std::find`, `std::count`

et le header `<numeric>` :

- `std::accumulation`, `std::inner_product`, etc.

```
std::vector<int> v(5, 3);

auto sum = 0;
std::for_each(v.begin(), v.end(), [&](int i) { // par lambda
    sum += i;
  });

auto sum = std::accumulate(v.begin(), v.end(), 0); // ok dans <numeric>
```

Complexité

Il faut choisir avec soin les conteneurs à utiliser en fonction des opérations à effectuer :

Conteneur	Opérations	Complexité
`std::vector`	Accès arbitraire	$O(1)$
	Insertion	$O(n)$
	Suppression	$O(n)$
`std::list`	Accès arbitraire	$O(n)$
	Insertion	$O(1)$
	Suppression	$O(1)$
`std::map`	Insertion	$O(logn)$
	Recherche	$O(logn)$

std::pair

La classe `std::pair` couple ensemble une paire de valeurs qui peuvent être de types différents :

```
template <class T1, class T2> class pair;
```

La classe `std::pair` a deux attributs `first` et `second` accessibles publiquement :

```
auto p = std::make_pair(1.0, 1); // std::pair<double,int>
auto r = p.first + p.second;
```

C'est le type d'insertion de la classe `std::map` :

```
std::map<std::string, int> m;
m.insert(std::make_pair("1", 1));
```

C'est également le type après déréférencement des itérateurs :

```
auto it = m.begin();
std::pair<std::string, int> p2 = *it;
```

std::tuple

La classe `std::tuple` couple ensemble une collection de valeurs qui peuvent être de types différents. C'est une généralisation de :

```
template <class... T> class tuple;
```

```
auto p = std::make_tuple(1, 1.f, 1.0); // std::tuple<int, float, double>
```

La classe `std::get` permet de extraire les valeurs :

```
auto r1 = std::get<0>(p); // 1
auto r2 = std::get<1>(p); // 1.f
auto r3 = std::get<2>(p); // 1.0
```

Remarque, `std::tuple` peut être utilisé en `retour de fonction` pour renvoyer un nombre multiple de variables :

```
std::tuple<std::string, int> foo () { return std::make_tuple("ok", 1); }
```

Timers

La partie <chrono> de la bibliothèque prend en charge la manipulation du temps :

```
#include <vector>
#include <numeric>
#include <chrono>
#include <iostream>

int main()
{
  std::vector<int> v(1000, 3);
  auto start = std::chrono::system_clock::now();
  auto sum = std::accumulate(v.begin(), v.end(), 0);
  auto end = std::chrono::system_clock::now();
  int elapsed_seconds =
    std::chrono::duration_cast<std::chrono::seconds>(end-start).count();

  std::cout << "sum = " << sum
            << ", elapsed time = " << elapsed_seconds << std::endl;
  return 0;
}
```

Programmation concurrente

Une gamme d'outils complète permet de manipuler les *threads* (basée sur `pthread`) dont (non exhaustif) :

- `std::thread` : classe représentant un thread d'execution,
- `std::mutex` : `std::lock _guard` : outils de synchronisation pour protéger les données partagées entre thread,
- `std::future` : mécanisme pour accéder aux résultats des opérations asynchrones,
- `std::async` : exécution de fonctions asynchrones,
- `std::atomic` : objets affranchis de *data race condition*.

```
int main()
{
  auto r = std::async([]() {
      return std::exp(1);
    });
  std::cout << r.get() << std::endl; // 2.71828
  return 0;
}
```

Programmation générique avancées

Méta-programmation

La *méta-programmation* est l'écriture et la manipulation de code à la compilation. En C++, cette technique repose sur les templates :

```
template<int N>
struct Factorial {
  static const int value = N * Factorial<N-1>::value;
};

template<>
struct Factorial<0> {
  static const int value = 1;
};
```

```
auto f = Factorial<5>::value;
```

Remarque, autrement dit, on utilise le langage C++ dans le langage !

Traits

Une classe de *traits* est une classe (souvent une structure) permettant d'apporter de la méta-information sur un type à la compilation. C'est une technique basée sur la programmation générique et la spécialisation :

```
template<typename U, typename V>
struct is_same { // faux par défaut
  static const bool value = false;
};

template<typename T>
struct is_same<T,T> { // spécialisation, vraie si même type
  static const bool value = true;
};

auto b1 = is_same<int,double>::value; // false
auto b2 = is_same<int,int>::value;    // true
```

Remarque, technique de base de la programmation générique.

Promotion de type

Application des *traits* à la promotion de type (à l'ancienne) :

```
template<typename U, typename V>
struct promote;

template<typename T>
struct promote<T,T> { using type = T; };

template<>
struct promote<double,int> { using type = double; };

template<>
struct promote<double,float> { using type = double; };
```

Exemple d'utilisation :

```
template<typename U, typename V>
typename promote<U,V>::type operator+ (const U& u, const V& v);
```

Remarque, `auto` et `decltype` font bien mieux !

<type_traits>

De nombreux traits sont définis dans la libraire standard :

- `std::is_class<T>` vérifie si le type est une classe,
- `std::is_integral<T>` vérifie si le type est intégral,
- `std::is_floating_type<T>` vérifie si le type est flottant,
- `std::is_pointer<T>` vérifie si le type est un pointeur,
- `std::is_same<T,U>` vérifie si ce sont les mêmes types,
- `std::add_const<T>` ajoute `const` au type,
- `std::remove_const<T>` retire `const` au type, etc.

Policy/Strategy Pattern

C'est une technique visant définir une politique (similaire à une interface) sans sacrifier la performance (i.e. sans méthode virtuelle) :

```
template<typename T>
struct policy {
  int foo () { return T::foo (); }
  int bar () { return T::bar (); }
};

struct A {
  int foo () { return 0; }
};

struct B {
  int foo () { return 1; }
  int bar () { return 2; }
};
```

Remarque, pas d'erreur si `policy<A>::bar ()` jamais appelé.

Substitution Failure Is Not An Error

SFINAE signifie que lorsque le compilateur substitue les types dans la déclaration d'une fonction template, s'il n'y arrive pas, ce n'est pas une erreur. Le compilateur recherche alors d'autres templates, c'est une erreur s'il n'y en a pas au final :

```
template<typename T>
void foo (typename T::type t);

template<typename T>
void foo (T t);

struct S {
  using type = int;
};

foo <S> (0); // foo (typename T::type) ok

foo<int> (0); // foo (typename T::type) ko
              // foo (T) ok
```

Généralisation

On peut définir un équivalent de `switch` pour les templates :

```
template <bool, typename T = void>
struct enable_if {};

template <typename T>
struct enable_if<true, T> {
  typedef T type;
};
```

Création d'une fonction `foo` définie pour les entiers et les classes :

```
template <typename T>
void foo (typename enable_if<std::is_integral<T>::value, T>::type &t);

template <typename T>
void foo (typename enable_if<std::is_class<T>::value, T>::type &t);
```

Remarque, le traits `std::enable_if` est dans la bibliothèque standard.

Curiously Recurring Template Pattern

CRTP est un idiome où une classe template dérive de son paramètre template. C'est une technique très utile pour retrouver un type en programmation générique :

```
template<typename T>
struct base : public T
{
  T& interface () { return static_cast<T&> (*this); }
};

struct derived {
  void foo ();
};
```

Exemple d'utilisation :

```
template<typename T>
void foo (base<T>& b) { interface ().foo (); } // T::foo () appelé
```

Remarque, technique de base des expressions templates.

Pour finir

Ce que l'on a pas vu

Plein de choses... en vrac :

- L'usage de `static`,
- Déplacement de mémoire pour les temporaires (mais pas que) :
  - `std::move` et `std::forward`, *Perfect Forwarding*,
- Les templates *variadiques* et la métaprogrammation en générale,
- Les mécanismes modernes d'énumérations :
  - `enum` et `scoped enum`,
- Les fonctions et classes *amies*,
- L'héritage multiple,
- L'incubateur `boost`,
- ... etc., etc., etc.

Littérature

Ce que je recommande pour aller plus loin :

- Bjarne Stroustrup : *Programming : Principles and Practice Using C++* (mis à jour pour C++11/C++14),
- Scott Meyers : *Effective C++*, *More Effective C++*, *Effective Modern C++*, *Effective Modern STL*,
- Herb Sutter : *Exceptional C++*, *More Exceptional C++*, *C++ Coding Standards*
- Andrei Alexandrescu : *Modern C++ Design*,
- David Abrahams : *C++ Template Metaprogramming*,
- Anthony Williams : *C++ Concurrency In Action*.