



# Langage C/C++

*De la syntaxe à la programmation orientée objet*

T. Grenier

O. Bernard, N. Ducros, T. Redarce

January 16, 2025

Contributeurs:

- T. Redarce, N. Ducros, O. Bernard
- Bryan Debout, étudiant Génie Electrique, INSA Lyon, 2015

Copyright © 2013 INSA Lyon

PUBLISHED BY INSA LYON

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

*First printing, March 2015*

# Contents

<b>1</b>	<b>Introduction</b> .....	<b>7</b>
<b>1.1</b>	<b>Il était une fois...</b>	<b>8</b>
<b>1.2</b>	<b>Utilisations actuelles</b>	<b>9</b>
1.2.1	Popularité et utilisation des langages .....	9
1.2.2	Relation avec les performances des processeurs .....	10
<b>1.3</b>	<b>Le C et le C++</b>	<b>12</b>
1.3.1	Histoires compilées .....	12
1.3.2	Utilisation .....	13

I

## Partie A: How to "Développer"

<b>2</b>	<b>Développement en C/C++</b> .....	<b>17</b>
<b>2.1</b>	<b>Rappels sur la compilation</b>	<b>17</b>
2.1.1	Pré-processeur .....	18
2.1.2	Compilateur .....	18
2.1.3	Éditeur de liens .....	18
<b>2.2</b>	<b>Exemple simple</b>	<b>18</b>
<b>3</b>	<b>Environnement de développement</b> .....	<b>21</b>
<b>3.1</b>	<b>Environnement QtCreator</b>	<b>21</b>
<b>3.2</b>	<b>Installation</b>	<b>22</b>
3.2.1	Téléchargement .....	22
3.2.2	Installation .....	23

<b>3.3</b>	<b>Création d'un projet</b>	<b>24</b>
<b>3.4</b>	<b>Compilation</b>	<b>26</b>
<b>3.5</b>	<b>Exécutions des programmes</b>	<b>27</b>
<b>3.6</b>	<b>Interactions utilisateur lors de l'exécution en mode console</b>	<b>28</b>
3.6.1	Execution dans un terminal	28
3.6.2	Passage de paramètres lors de l'exécution	28
3.6.3	Terminal interne ou externe	28
<b>3.7</b>	<b>Débogage</b>	<b>29</b>
<b>3.8</b>	<b>Paramétrage du compilateur</b>	<b>30</b>

## II

## Partie B: Le langage

<b>4</b>	<b>Introduction à la programmation</b>	<b>35</b>
<b>5</b>	<b>Mémento de la syntaxe C++</b>	<b>37</b>
<b>5.1</b>	<b>Syntaxe élémentaire</b>	<b>37</b>
5.1.1	Instructions	37
5.1.2	Commentaires	38
5.1.3	Casse	38
5.1.4	Symboles	38
<b>5.2</b>	<b>Mots clés et mots réservés</b>	<b>39</b>
5.2.1	Mots clés	39
5.2.2	Directives	39
<b>5.3</b>	<b>Variables et types</b>	<b>40</b>
5.3.1	Types fondamentaux en C++	40
5.3.2	Déclaration de variables	41
5.3.3	Déclaration de variables avec affectation	41
<b>5.4</b>	<b>Bases et systèmes de numération</b>	<b>42</b>
<b>5.5</b>	<b>Opérateurs</b>	<b>42</b>
5.5.1	Opérateurs unaire, binaire et ternaire	42
5.5.2	Priorité des opérateurs	43
5.5.3	Associativité des opérateurs	43
<b>5.6</b>	<b>Opérations binaires et logiques</b>	<b>43</b>
5.6.1	Opérateurs binaires	43
5.6.2	Opérateurs logiques	43
<b>5.7</b>	<b>Structures conditionnelles</b>	<b>46</b>
5.7.1	if / else	46
5.7.2	switch / case	47
<b>5.8</b>	<b>Structures de boucles</b>	<b>48</b>
5.8.1	for	48
5.8.2	while	49
5.8.3	do / while	49
<b>5.9</b>	<b>Pointeurs et Références</b>	<b>50</b>
5.9.1	Pointeurs	50
5.9.2	Références	51

<b>5.10</b>	<b>Tableaux</b>	<b>52</b>
5.10.1	Allocation statique	53
5.10.2	Allocation dynamique	53
5.10.3	Tableaux et pointeurs	55
5.10.4	Tableaux multidimensionnels	55
<b>5.11</b>	<b>Fonctions</b>	<b>56</b>
5.11.1	Prototype d'une fonction	56
5.11.2	Définition d'une fonction	56
5.11.3	Appel de fonction	57
5.11.4	Passage d'arguments à une fonction	57
5.11.5	Cas des paramètres avec allocation dynamique dans une fonction	59
5.11.6	Surcharge de fonction	61
<b>6</b>	<b>Flux et interactions utilisateurs</b>	<b>63</b>
<b>6.1</b>	<b>Flux d'entrée cin et de sortie cout</b>	<b>63</b>
<b>6.2</b>	<b>Flux de sortie pour l'affichage: cout</b>	<b>63</b>
<b>6.3</b>	<b>Flux d'entrée clavier: cin</b>	<b>64</b>
<b>6.4</b>	<b>Cas des chaines de caractères</b>	<b>65</b>
<b>7</b>	<b>Flux et Fichiers</b>	<b>67</b>
<b>7.1</b>	<b>Fichiers</b>	<b>67</b>
<b>7.2</b>	<b>Mode texte</b>	<b>67</b>
<b>7.3</b>	<b>Mode binaire</b>	<b>68</b>
<b>7.4</b>	<b>Et en C?</b>	<b>69</b>

### III

## Partie C: Orienté Objet

<b>8</b>	<b>Programmation Orientée Objet en C++</b>	<b>73</b>
<b>8.1</b>	<b>UML</b>	<b>73</b>
8.1.1	Diagramme de classe	74
8.1.2	UML et C++	75
<b>8.2</b>	<b>Concept de classe pour l'encapsulation</b>	<b>76</b>
8.2.1	Définition de classe	77
8.2.2	Visibilité	77
8.2.3	Données membres	78
8.2.4	Fonctions membres	79
8.2.5	Objets	80
8.2.6	Surcharge de méthodes	81
<b>8.3</b>	<b>Méthodes particulières</b>	<b>82</b>
8.3.1	Constructeurs	82
8.3.2	Constructeur de copie	83
8.3.3	Destructeurs	83
8.3.4	Opérateurs	84
<b>8.4</b>	<b>Héritage</b>	<b>89</b>
8.4.1	Type d'héritage	89
8.4.2	Appel des constructeurs et des destructeurs	91

<b>8.5</b>	<b>Polymorphisme</b>	<b>93</b>
8.5.1	Définition .....	93
<b>8.6</b>	<b>Généricité</b>	<b>95</b>

## IV

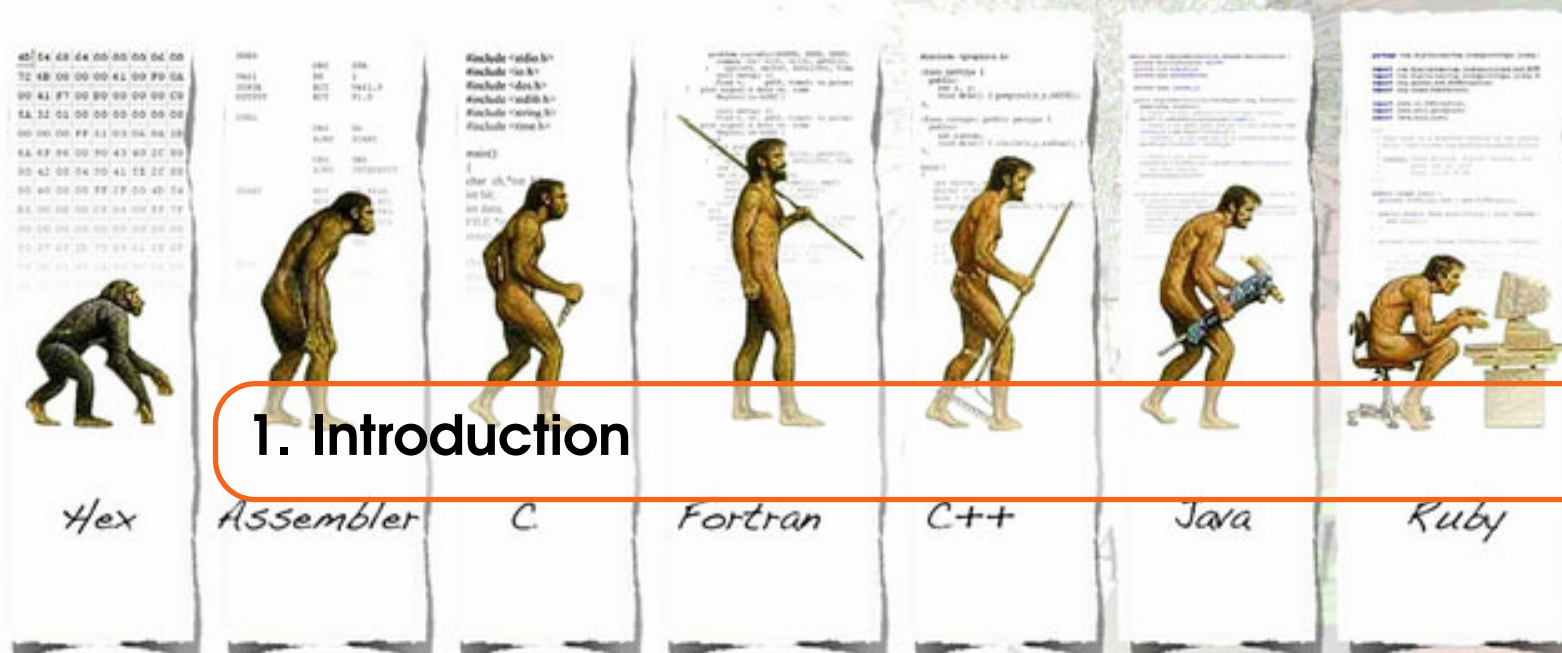
## Part C: Thèmes choisis

<b>9</b>	<b>Syntaxe moderne du C++ .....</b>	<b>99</b>
9.1	C++11, C++14 et C++17	99
9.2	Mot clés	100
9.3	Messages d'aide au débogue	108

## Annexes

<b>Bibliographie .....</b>	<b>113</b>
<b>Index .....</b>	<b>115</b>

# The Evolution Of Computer Programming Languages



## 1. Introduction

Apprendre, connaître et savoir utiliser le C et le C++, oui mais pourquoi? Matlab, GNU Octave, R, Scilab sont très adaptés pour mes calculs numériques et toutes les fonctions 'compliquées' sont connues ou existent! Puis pour le web, Php et javascript suffisent, non? Et si par hasard je veux faire une application Android, et ben : java! Pour le reste Python est très souple et permet de quasiment tout faire: du calcul numérique (scipy, numpy) à l'intelligence artificielle (Tensorflow/Keras, PyTorch), en passant par des interfaces graphiques, l'accès aux bases de données, des pages pour les serveurs web, et du traitement d'image et à la radio numérique<sup>1</sup> (gnuradio et ses scripts python)! Il n'y a que si j'ai besoin de développer un driver pour linux, ou d'écrire un programme bas niveau pour un système embarqué que j'ai besoin du C?

C'est pas faux: il faut utiliser les langages les plus adaptés à son problème. Ceci sous entend de bien évaluer le problème et l'environnement du problème. Mais il y a au moins deux pièges.

D'abord, quand le concept "qui peut le plus peut le moins" n'est pas adapté. Effectivement, ne jurer que par Matlab (par exemple!), c'est comme croire qu'un automate s'impose toujours pour piloter un système... n'importe quel système. Même pour faire clignoter une LED lorsqu'un bouton est enfoncé. C'est aussi se restreindre à penser que les AOP permettent de tout amplifier, rendant inutiles les transistors; que seules les machines synchrones sont pertinentes pour la mise en rotation, etc.

Non, il faut bien sur considérer les ressources disponibles, de coûts, etc. Ainsi les contextes de chaque problème imposent que la solution doit être réfléchie et adaptée.

En informatique, c'est évidemment pareil. Il y a des cas où écrire un programme en assembleur est trop laborieux, non portable et non réalisable en temps raisonnable (c'est d'ailleurs l'origine du C), et des cas où utiliser des langages très haut niveau interprétés n'est pas en adéquation avec le matériel ou la consommation de ressources (qu'elles soient financières, énergétiques, calculatoires, mémoires, encombrements... et souvent: toutes à la fois!). Dans certains de ces cas le C/C++ est une alternative à considérer, surtout en terme de formation car ces deux langages, exigeants, permettent de couvrir la totalité des enjeux précédents et offrent aussi une bonne base de connaissances pour passer à d'autres langages (qui souvent utilisent C/C++).

<sup>1</sup>Plus exactement SDR : *Software Defined Radio*.



Le second piège est la méconnaissance du système sur lequel on exécute son programme.

- Quelles sont les limites des ressources de ce système? Que va-t-il se passer si elles sont dépassées?
- Pourquoi d'ailleurs mon programme plante de temps en temps, ou rend le système très lent?
- Quels sont les impacts des défaillances de l'exécution de mon programme? Existe-il une solution minimisant ces défaillances ou garantissant leur "bonne prise en compte"?

On peut tenter le parallèle avec un manager qui budgétiserait, planifierait et exécuterait ses projets sans connaître son équipe opérationnelle. Parfois il faudra aller voir qui exécute le programme et vérifier que celui-ci est bien adapté (sans nécessairement être l'optimal). Or, dans le cas des langages et plus largement de la programmation informatique, cette vérification n'est pas forcément aisée et pourra déboucher sur une réponse "*on ne peut pas faire mieux avec ce langage*" en cas de connaissance trop superficielle du système.

Il faudra aussi se pencher sur l'algorithme lui-même: l'efficacité d'un algorithme (coût temporel et mémoire) et son exactitude (donne-t-il toujours le bon résultat?) sont des concepts à maîtriser et capitaux pour optimiser l'utilisation des ressources (voir le livre [Cor+09]). Il faudra aussi analyser comment le programme est exécuté sur tel ou tel processeur et comment il interagit avec les autres programmes.

Cette introduction continue avec un petit historique des langages, avant de faire un tour des utilisations actuelles des langages puis de se focaliser sur le C et le C++.

## 1.1 Il était une fois...

Historiquement, le premier programme informatique est attribué à Ada Lovelace qui, vers 1843, décrit une méthode pour calculer les nombres de Bernoulli avec la machine *Analytical Engine* imaginée par Charles Babbage. Cette machine conceptualise le calculateur tout usage (légère adaptation du terme *general purpose computer*). Le premier ne sera réalisé qu'en 1940. À partir de cette date, plusieurs langages de programmation vont voir le jour afin d'exprimer ce que l'on veut faire faire à ce calculateur et comment. Les calculateurs étant de plus en plus performants, les programmes deviennent de plus en plus ambitieux et doivent être décrits de manière plus concise. Les calculateurs se diversifient rapidement, les manières de les programmer suivent. Des domaines d'application apparaissent et des langages spécifiques sont proposés. Encore une fois, il est difficile de séparer langage, processeur et problème.

La figure 1.1 donne l'ordre chronologique d'apparition de quelques langages.

Cette figure est à mettre en parallèle avec l'évolution des langages: un langage s'appuie généralement sur un ou plusieurs langages existants. Ainsi, un nouveau langage complète l'offre existante en ajoutant ou regroupant des fonctionnalités issues d'autres langages. Parfois, l'essentiel de la syntaxe provient d'un langage existant (C, C++, java, C#, ...) facilitant ainsi l'adhésion (puis la migration) d'une communauté de développeurs à ce nouveau langage.

Plusieurs critères décrivent les spécificités d'un langage par rapport à un autre. Les deux critères les plus utilisés pour classer les langages en catégories sont:

- soit un langage est **impératif** (langage qui décrit une suite d'instructions à exécuter) soit il est **déclaratif** (langage qui décrit le problème plutôt que définir une solution: Prolog, SQL,...) .
- soit un langage est **interprété** soit il est **compilé**.

On se focalise par la suite que sur les langages impératifs. Notons que l'essentiel des processeurs est impératif.

Un langage compilé permet de créer un programme réalisé par un compilateur qui va transformer le code source en code machine, c'est à dire en un code exécutable par un processeur: les instructions, les variables, etc sont comprises et réalisées par un processeur. Ce programme peut ensuite être exécuté. Par exemple : ALGOL, BASIC, COBOL, C, C++, C#, Delphi, ...



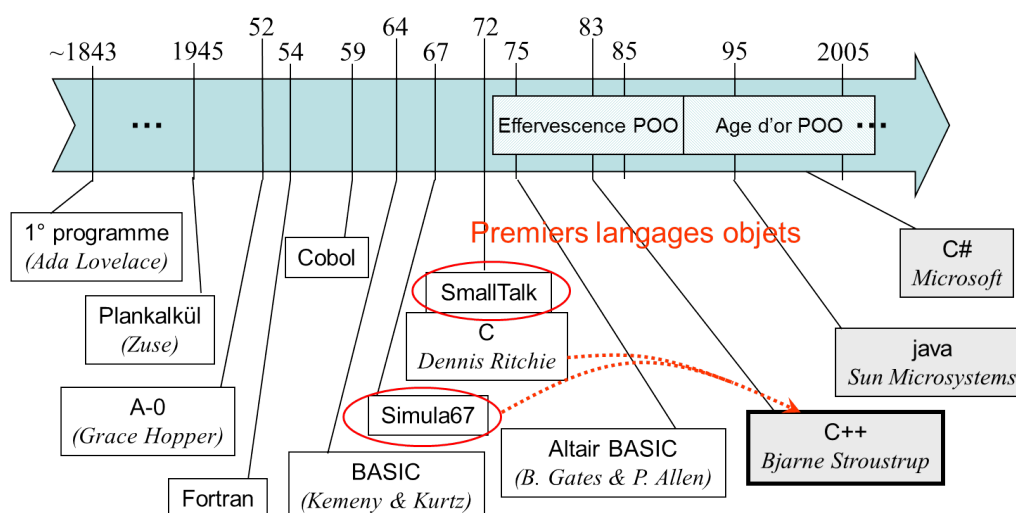


Figure 1.1: Historique des langages. (POO : Programmation Orientée Objet)

Un langage interprété a une exécution interprétée (souvent ligne à ligne à partir du code source) par un interpréteur qui est chargé de transformer dynamiquement le code en langage machine (au travers d'une *machine virtuelle*). Les langages suivants sont généralement interprétés : BASIC, JavaScript, Maple, PHP, Python, ...

## 1.2 Utilisations actuelles

En 2015, on dénombrait environ 700 langages <sup>2</sup>, Xavier Leroy <sup>3</sup> en recense entre 2000 et 3000. Une vingtaine de langages concentre plus de 80% de la popularité. Il n'est pas évident de faire un lien rigoureux entre cette popularité et l'utilisation du langage. Le terme même d'utilisation est très ambiguë (nombre de lignes, nombre de programmes, nombre de forums, nombre d'ingénieurs, nombre d'embauches,...).

Un autre amalgame fréquent est la relation entre un langage et la performance du programme. Ces deux points -popularité et performances- sont traités successivement dans la suite.

### 1.2.1 Popularité et utilisation des langages

Le site web Tiobe propose un classement des langages en fonction de leur popularité. Il s'agit en fait d'un ratio mesuré en analysant le nombre de cours, de forums, de pages web, de vendeurs de parties tierces et d'ingénieurs se revendiquant compétents dans le langage à partir des plateformes de recherches les plus connues (Youtube, Google, Wikipedia, Amazon,...). La figure 1.2 donne l'évolution de la popularité relative de 20 langages de programmation sur la période 06/2002 - 08/2021.

Plusieurs autres indicateurs existent avec différentes sources et critères:

- **PYPL** figure 1.3 : [pypl.github.io](http://pypl.github.io) basé sur le nombre de fois où des tutoriels sont recherchés sur Google.
- **RedMonk** figure 1.4: [redmonk.com](http://redmonk.com) basé sur la corrélation de l'utilisation des langages sur GitHub et les discussions sur Stack Overflow.
- **Trendy Skills** figure 1.5 : [trendyskills.com](http://trendyskills.com) qui déduit son indice à partir des compétences recherchées sur les sites d'embauches populaires et permet aussi de mettre en correspondance

<sup>2</sup>En négligeant notamment de nombreux dialectes du BASIC, *Wikipedia*.

<sup>3</sup>Nommé à la chaire de Sciences du logiciel comme professeur au collège de France depuis 2018

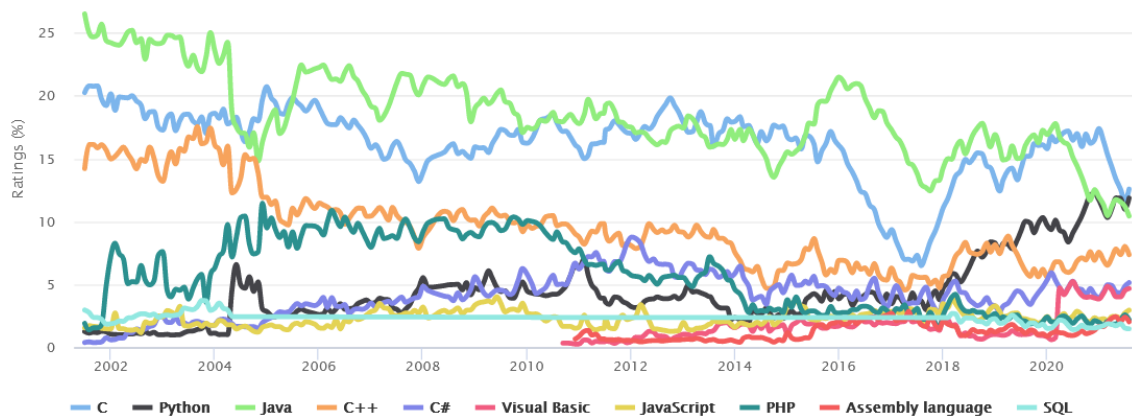


Figure 1.2: Évolutions de la popularité des 20 langages les plus utilisés. Ces 20 langages correspondent à 70% de la popularité totale. Source TIOBE, Septembre 2021.

le salaire proposé.

L'analyse de ces indicateurs montrent que python, java, javascript, C et C++ sont les langages de programmation les plus utilisés et les plus présents. Les progressions de Kotlin et Rust sont à considérer pour les développements Android et sécurisés, respectivement.

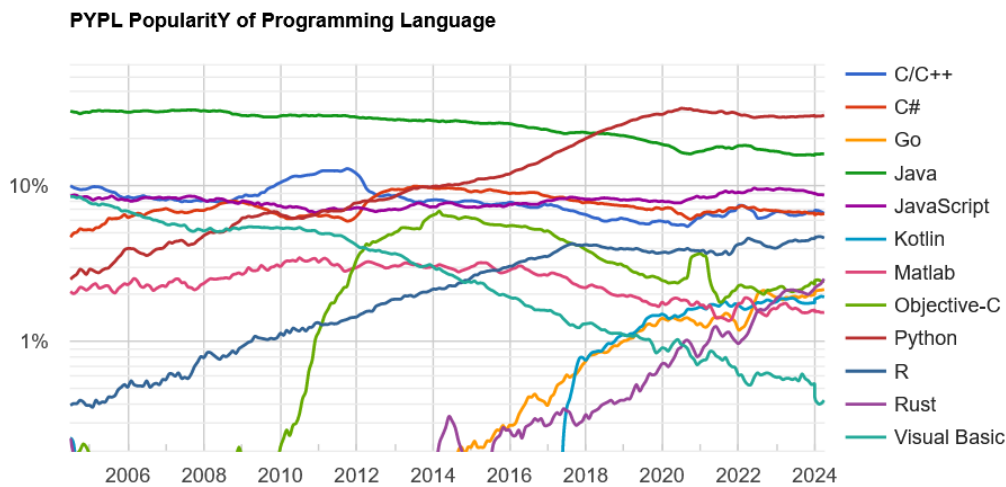


Figure 1.3: Popularité des langages par la recherche de tuto sur Google. Source PYPL, Avril 2024.

### 1.2.2 Relation avec les performances des processeurs

Il est très vraisemblable que la prédiction de Gordon Moore en 1975 (sur le doublement du nombre de transistors dans les processeurs tout les ans) ne puisse encore tenir longtemps, si elle tient encore! Depuis 2002, elle n'est plus en adéquation avec le doublement de la performance de calcul. Il y a même une relative stagnation des performances des processeurs depuis 2004. C'est aussi en 2004 que l'un des plus grands acteurs du domaine (Intel) a dû admettre que les futures améliorations de performances passeraient plus par les multi-coeurs et que par le processeur en lui même. Cependant, malgré les multi-coeurs il y a stagnation de l'évolution des performances calculatoires. Il y a un goulot d'étranglement des performances quelque part. Les bandes passantes des bus et des mémoires ont été pointées du doigts pendant longtemps. Elles ont cependant

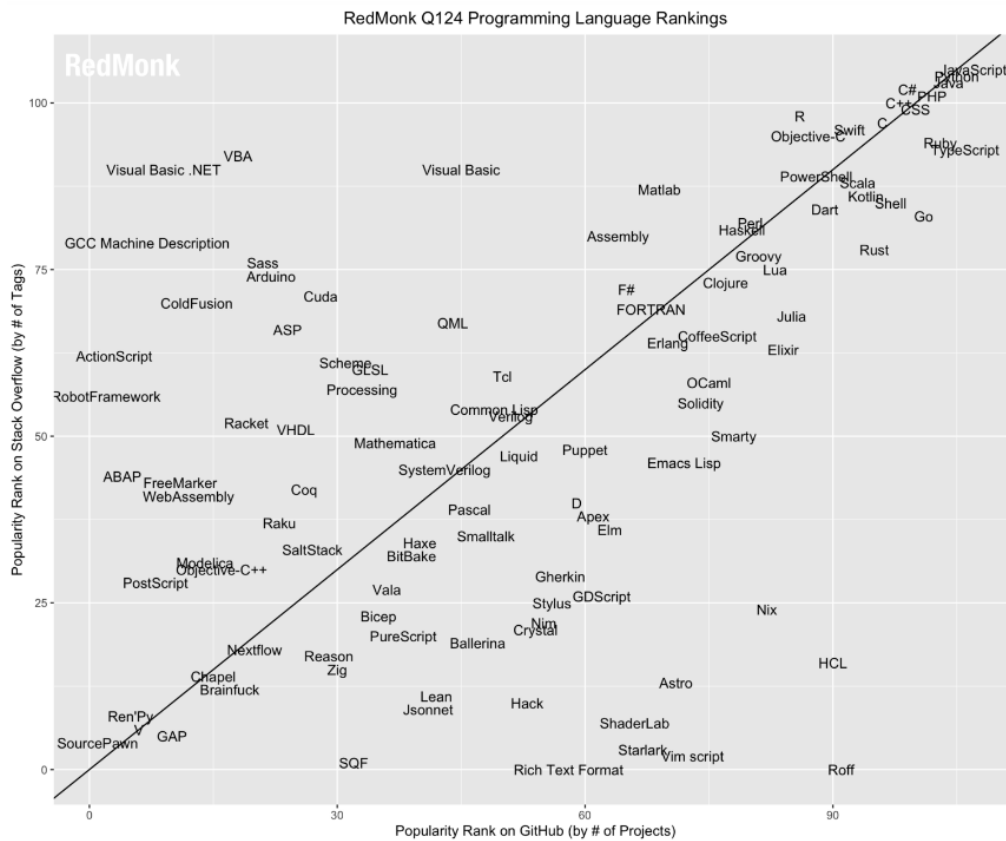


Figure 1.4: Corrélation des discussions Stack Overflow et projets GitHub. Source *RedMonk*, Avril 2024.

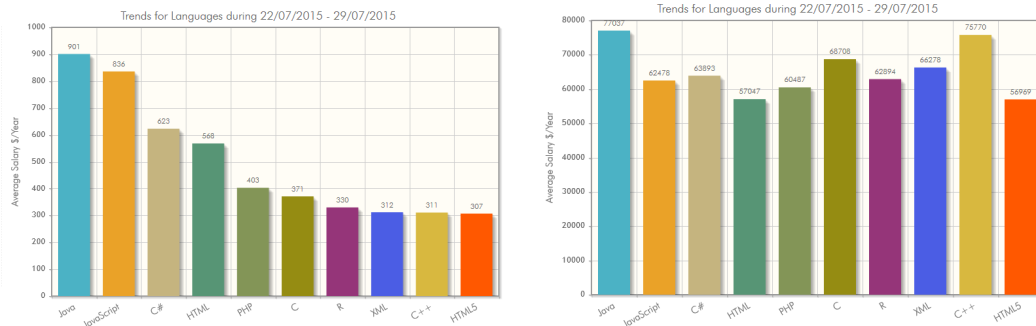


Figure 1.5: Compétences cherchées (à gauche) et salaire (à droite) sur des profils de postes dans 14 pays (sans la France). Source *Trendy Skills*, Juillet 2015.

beaucoup progressé et ne semblent pas être la cause du manque de performances des processus calculatoires. Plusieurs auteurs concluent qu'en fait, la performance vient du logiciel [HP11]. Ils pointent du doigt directement l'utilisation non optimale que les programmeurs font des ressources en ne maîtrisant que trop faiblement l'impact de leur lignes de code sur les performances des systèmes et aussi les stratégies de programmation utilisées (dont la négligence de l'algorithmie). Tout ceci est lié: de nombreux langages poussent et mettent en avant leur simplicité d'utilisation pour obtenir très rapidement et sans trop de connaissances ni d'efforts des développements intéressants: ils complaisent très souvent les développeurs dans une superficialité de maîtrise et ne poussent pas à comprendre plus en profondeur les systèmes. Ainsi, ils peuvent rapidement faire prendre de

mauvaises habitudes de programmation aux développeurs peu expérimentés et peu avertis qui deviennent alors complètement dépendant du langage-outils.

Souvent, il y a des enjeux commerciaux ou financiers et de compétitivité à courts termes : développer vite avec le moins de ressource possible et le moins qualifiée possible.

L'enquête "Embedded Market Study" présentée en 2013 au *Design West* montre que le C et C++ sont les deux langages privilégiés pour les systèmes embarqués, et qu'à priori cela ne changerait pas tout de suite comme le montre les figures 1.6. Ces deux langages sont proches du matériel et permettent une utilisation très juste des ressources. Encore faut-il les maîtriser (langages et ressources). Effectivement, ces deux langages demandent des efforts de compréhension et de rigueur pour développer des applications fiables et sûres.

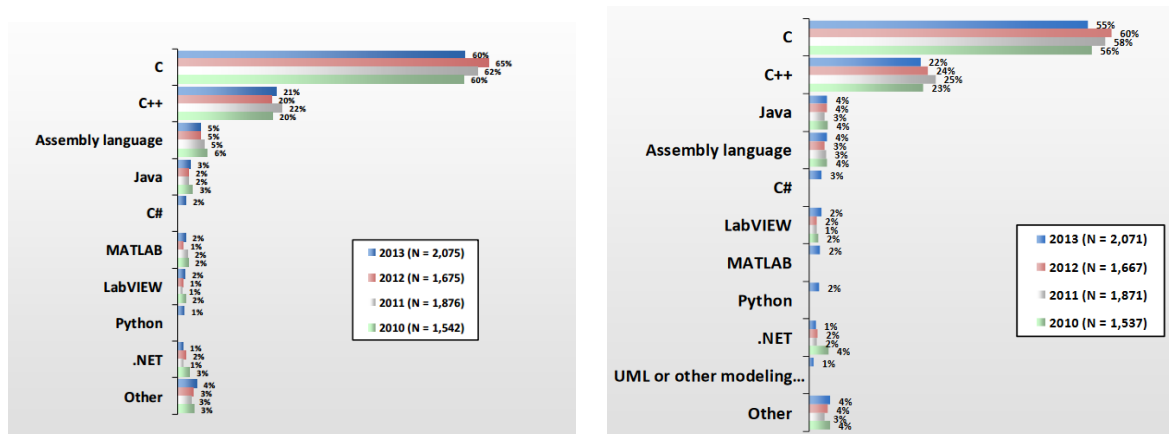


Figure 1.6: Langues pour les systèmes embarqués qui sont (à gauche) et seront (à droite) utilisés par les développeurs pour leurs prochains projets en embarqué. Source 2013 EMBEDDED MARKET STUDY Design WEST.

## 1.3 Le C et le C++

### 1.3.1 Histoires compilées

Denis Ritchie travaillant aux laboratoires Bell a créé le langage C pendant l'année 1972. Le langage C s'est très vite imposé pour principalement trois raisons.

- Ce langage a été créé pour le re-développement du système Unix qui était devenu une référence de système d'exploitation multi-tâches et multi-utilisateurs mais impossible à maintenir car initialement écrit en assembleur.
- La diffusion du compilateur C au milieu universitaire eut la même stratégie que pour la diffusion d'Unix: une licence à coût très modeste (~300\$) à acquérir par les universités et valable pour tous leurs étudiants.
- **Sa puissance.** C'est le premier langage de haut niveau qui grâce aux pointeurs permet d'être au plus proche du processeur et d'interagir avec tous les éléments du systèmes.

Une extension Orientée Objet au langage C a été conçue par Bjarne Stroustrup (au Bell Labs d'AT&T) dans les années 1980. Il s'agit du C++. Cette extension s'inspire d'autres langages<sup>4</sup> dont

<sup>4</sup>C, **Ada 83** (dont le nom est un hommage au premier programmeur informatique (Ada Lovelace), ce langage se voulait proche du matériel pour pouvoir être embarqué sur des systèmes temps-réels), **Algol 68** (l'utilisateur peut définir de nouveaux types de donnée, de nouveaux opérateurs ou surcharger des existants), **CLU** (pour *CLUster*, dont l'élément novateur essentiel était *cluster* et qui correspondait au concept de classe, mais où intentionnellement de nombreux paradigmes objets avaient été omis), **ML** (*Meta Language*, qui est un langage fonctionnel impur permettant des adaptations et détermination de type)

**Simula67** (version sortie en 1967 du langage *Simula*) qui est le premier langage à *classes* c'est à dire le premier langage orienté objet.

Le C comme le C++ sont normalisés par l'ISO et continuent d'évoluer. La prochaine révision du langage C++ est celle de 2017.

### 1.3.2 Utilisation

Chaque langage a ses spécificités et des prédispositions à exprimer certaines solutions. En clair, pour un type d'application donné, tous les langages ne se valent pas... et les développeurs l'ont bien compris<sup>5</sup>. Ainsi, Matlab n'est pas vraiment adapté à l'écriture d'un site web marchand ou à l'écriture d'un driver pour une carte graphique.

Il existe cependant des langages dont l'usage s'est révélé général, voir à *tout faire*, ou presque. L'un des plus général actuellement est le Java dont les domaines d'utilisation sont: l'écriture d'applications, le développement sur plateformes embarquées, Web, l'écriture d'applications coté serveur et coté client, la programmation générale,...

Le C est aussi utilisé pour la programmation générale, la programmation système, les opérations de bas niveau et l'écriture d'applications. Ce dernier point est de moins en moins représenté. Le C++ est lui plus utilisé pour la programmation d'applications et la programmation des systèmes. De nombreuses bibliothèques de calcul sont écrites en C++ (Eigen, ITK, VTK, PCL, Boost, GMP, OpenGL, ImageMagick, ...) de même pour les IHM (Qt, wxWidgets, GTKmm...).

Notons qu'il ne s'agit pas ici des aptitudes des langages mais bien de leurs utilisations les plus courantes.

---

<sup>5</sup>Ils sont même à l'origine de l'écriture d'un nouveau langage ou des évolutions d'un langage afin de répondre à une nouvelle demande.





# Partie A: How to "Développer"

<b>2</b>	<b>Développement en C/C++ .....</b>	<b>17</b>
2.1	Rappels sur la compilation	
2.2	Exemple simple	
<b>3</b>	<b>Environnement de développement ...</b>	<b>21</b>
3.1	Environnement QtCreator	
3.2	Installation	
3.3	Création d'un projet	
3.4	Compilation	
3.5	Exécutions des programmes	
3.6	Interactions utilisateur lors de l'exécution en mode console	
3.7	Débogage	
3.8	Paramétrage du compilateur	



## 2. Développement en C/C++

### 2.1 Rappels sur la compilation

Avant de donner plus de détails sur les notions de base en C++, il est important de rappeler les différentes étapes intervenant lors de la création d'un exécutable (ou d'une librairie) à partir de fichiers C++. La figure 2.1 donne un schéma simplifié de la création d'un exécutable lors de la compilation en C++.

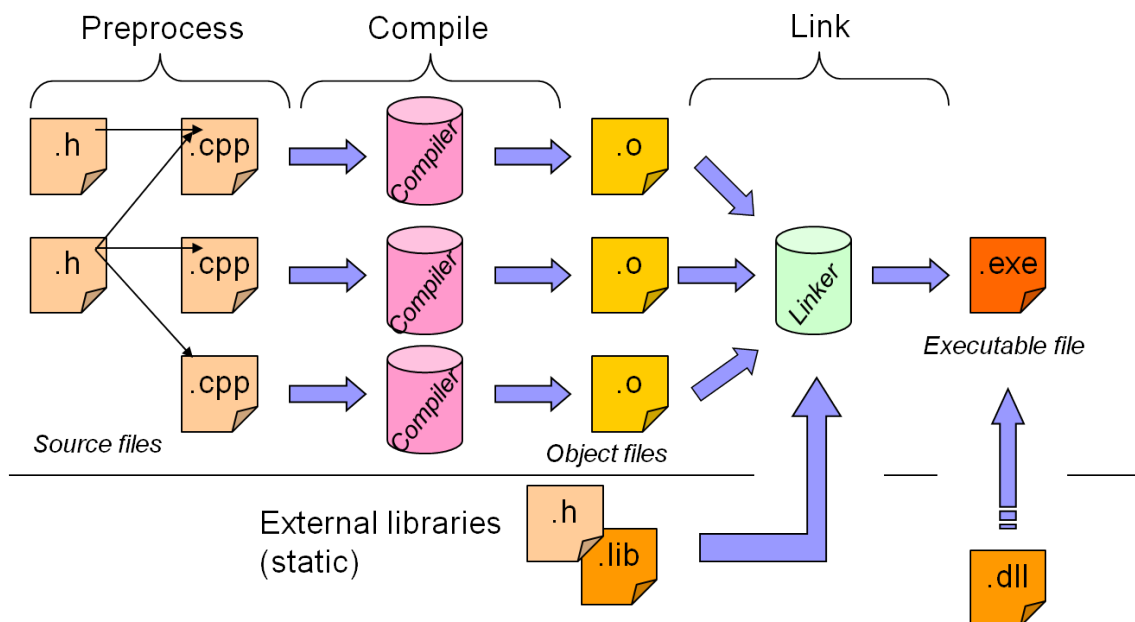


Figure 2.1: Création d'un exécutable lors de la compilation en C++

De façon simplifiée, trois étapes sont réalisées lors de la création d'un exécutable.

### 2.1.1 Pré-processeur

Le pré-processeur ou pré-compilateur est un utilitaire qui traite le fichier source avant le compilateur. C'est un manipulateur de chaînes de caractères. Il retire les commentaires, prend en compte les lignes du texte source ayant un # en première colonne, etc, afin de créer le texte que le compilateur analysera. Ses possibilités sont de trois ordres:

- inclusion de fichiers (mot clé `#include`). Lors de cette étape, le pré-compilateur remplace les lignes `#include` par le fichier indiqué;
- définition d'alias et de macro-expression (mots clés `#define`, `macro`);
- sélection de parties de texte (mot clé `inline`).

A la fin de cette première étape, le fichier `.cpp` (ou `.c`) est complet et contient tous les prototypes des fonctions utilisées.

### 2.1.2 Compilateur

L'étape de compilation consiste à transformer les fichiers sources en code binaire compréhensible par un processeur. Le compilateur compile chaque fichier `.cpp` (ou `.c`) un à un. Les fichiers d'entête `.h` (ou *header*) ne sont pas compilés. Le compilateur génère un fichier objet `.o` (ou `.obj`, selon le compilateur) par fichier `.cpp`. Ce sont des fichiers binaires temporaires. Après la création du binaire final (l'exécutable), ces fichiers pourront être supprimés ou gardés selon les besoins de re-compilation. Enfin, il est à noter que ces fichiers peuvent contenir des références insatisfaites à des fonctions. Elles devront être résolues par l'éditeur de liens.

### 2.1.3 Éditeur de liens

L'éditeur de liens prend chaque fichier objet généré par le compilateur (`.o`) et les associe pour créer un fichier binaire (l'exécutable ou la librairie). Il se sert de bibliothèques pour résoudre les références insatisfaites. Il doit donc connaître l'emplacement et les bibliothèques à utiliser... Cela demande parfois de longues configurations!

## 2.2 Exemple simple

Voici un exemple de codage et compilation simple. On utilise ici des outils élémentaires. Les outils présentés dans la prochaine partie permettront d'effectuer ces tâches plus facilement.

La première étape est d'écrire un programme : un fichier source qui contiendra les instructions que l'on veut faire faire au processeur. Pour cela, utiliser un éditeur de texte orienté programmation (*notepad++*, *gedit*, ...). La figure 2.2 illustre la création du fichier `main.cpp` sous l'éditeur *vim*.

Il s'agit ensuite de compiler le programme. Pour cela il faut disposer d'un logiciel permettant de compiler. La figure 2.2 donne la ligne de commande linux permettant d'invoquer le compilateur `g++` pour compiler le fichier `main.cpp` et créer l'exécutable `TD1`. La dernière ligne de la figure permet d'exécuter le programme `TD1` qui affiche un message.

```

Terminal
Fichier Éditer Paramètres Aide
#include <iostream>
using namespace std;

int main(void)
{
    cout << " Hello " << endl;
    cout << "   World ! \n";

    return 0;
}
~
~
~
~
~
~
~
~
~
~
~
~
~
"main.cpp" 10L, 129C 1

Terminal
Fichier Éditer Paramètres Aide
bash-2.04$ vim main.cpp
bash-2.04$ g++ main.cpp -o TD1
bash-2.04$ ./TD1
  Hello
  World !
bash-2.04$ █

```

Figure 2.2: Création d'un code C++ sous *vim* (à gauche) et les commandes linux permettant l'édition du fichier *main.cpp*, la compilation avec *g++* puis l'exécution du programme (à droite).





## 3. Environnement de développement

Un environnement de développement (ou IDE, de l'anglais *Integrated Development Environment*) est un programme regroupant généralement tous les outils nécessaires au développement. Les IDE les plus efficaces sont ceux qui permettent:

- la création et la gestion de projets (plusieurs fichiers, plusieurs exécutables, plusieurs bibliothèques, ...),
- l'édition des fichiers avec une reconnaissance de la syntaxe et la complétion de code,
- la compilation, l'exécution et le débogue des programmes,
- le *refactoring* et le *versionning* (git, svn, hg , ...) de code.

Il existe de nombreux IDE pour travailler en C++. Voici une liste non exhaustive d'IDE gratuits (compatibles Linux, Windows et Mac):

- **QtCreator** que l'on va utiliser par la suite,
- Microsoft Visual Studio et Microsoft Visual Studio Code,
- Code::Blocks,
- Eclipse, supportant de nombreux langages (télécharger *Eclipse IDE for C/C++ Developers*)
- ...

Aujourd'hui, il est peu envisageable de se passer d'un IDE pour la réalisation d'un programme. Les IDE vont automatiser l'enchaînement des tâches de construction de l'exécutable, faciliter l'édition des fichiers (navigation, refactoring, analyse de la syntaxe, auto-complétion,...) et simplifier le débogue. Cependant, cela ne sera possible que si l'IDE a les informations sur ce que l'on souhaite développer: quels fichiers? avec quelles bibliothèques? pour quel processeur?... Toutes ces précisions seront à donner et elles se font au travers des projets (paragraphe 3.3).

### 3.1 Environnement QtCreator

Nous présentons ici un environnement de développement *Qt Creator*. Cet environnement présente une interface très représentative des IDE. QtCreator fonctionne sous Windows, Linux/X11 et Mac OS. En plus des OS précédents, il permet d'écrire des applications pour Android, iOS et supporte

la compilation croisée. Cet IDE s'appuie sur la librairie Qt<sup>1</sup>.

La figure 3.1 donne un aperçu de son interface. La disposition des outils peut être différente en fonction de la version.

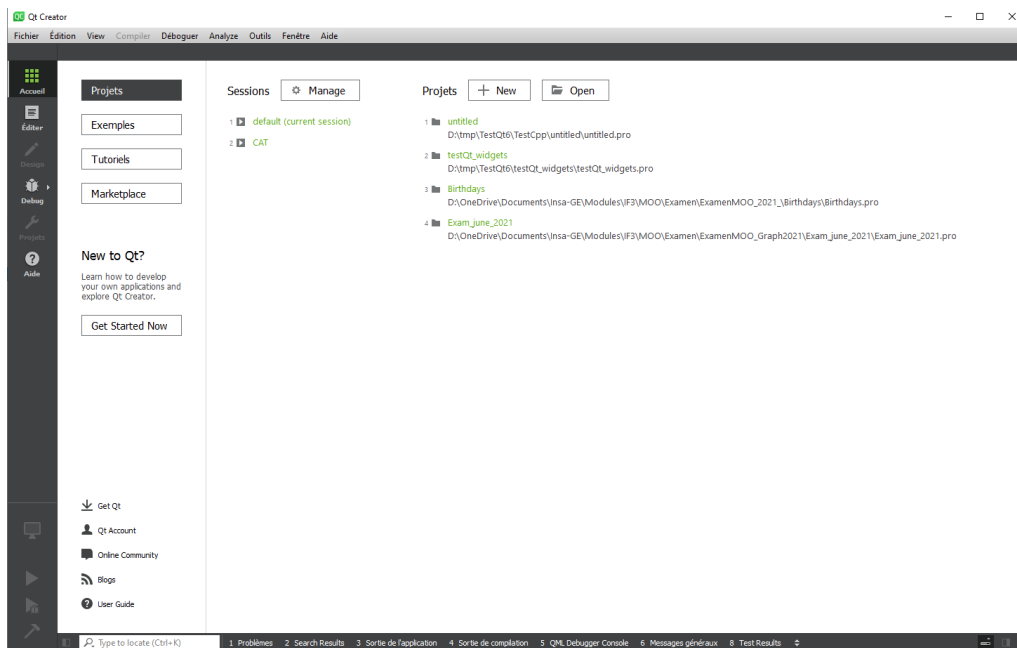


Figure 3.1: interface type de QtCreator

Dans ce qui suit, nous décrivons les étapes clés afin d'être rapidement autonome sur cet IDE. Notons que la démarche et les termes présentés ci-après sont les mêmes pour les autres IDE.

## 3.2 Installation

L'environnement de développement QtCreator est compatible Linux, Mac et Windows.

Cet environnement peut s'installer sans la librairie Qt. Cependant, ceci n'est pas recommandé pour un utilisateur débutant. Une telle installation nécessiterait de disposer :

- une chaîne de compilation (compilateur, éditeur de lien, bibliothèques, ...),
- un débogueur (optionnel),
- un générateur de *Makefile*: CMake.

**Il est donc conseillé de télécharger et d'installer Qt qui permettra de disposer de tous les éléments nécessaires.**

### 3.2.1 Téléchargement

La librairie Qt a subi de nombreux rebondissements de licence. Elle est maintenant disponible sous forme commerciale et communautaire (LGPL v2/v3<sup>2</sup>).

Le téléchargement de la version communautaire de Qt et de QtCreator se fait à partir du site [www.qt.io/download-qt-installer-oss](http://www.qt.io/download-qt-installer-oss). Il est possible de télécharger les sources de Qt puis de les compiler soi-même. Cet exercice est très intéressant, mais peut se révéler particulièrement complexe à réaliser.

L'utilisation d'une version compilée (ou binaires) est donc plus rapide à mettre en œuvre sur son système. De plus, les versions compilées (ou binaires) de Qt intègrent QtCreator et parfois

<sup>1</sup> Prononcer 'quioute' ... de l'anglais *cute*, ou "Q.T." à la française ...

<sup>2</sup> Licence publique générale limitée GNU

différentes chaînes de compilation propres au système d'exploitation et au processeur. Bien que disponibles, les versions pour MCU, "Android" et autres OS embarqués ne sont pas à installer sauf si vous projetez de compiler des applications pour ces plateformes.

**R** Pour installer Qt, il faudra vous créer un compte Qt.

### 3.2.2 Installation

#### Linux

Pour le système d'exploitation Linux, l'installation par le gestionnaire de paquets de votre distribution est plus simple que le téléchargement du binaire proposé sur le site web. Solution à privilégier sauf si vous avez besoin de la toute dernière version et que vous ne pouvez pas attendre la disponibilité du paquetage pour votre distribution. Les différents éléments de Qt sont à installer individuellement par le gestionnaire de paquets de votre distribution.

#### Mac OS X

Pour Mac, l'installation du paquet `qt-opensource-mac-x64-clang-X.X.X.dmg` (les X représentent la version) sur le site devrait suffire. Il faudra disposer de `clang`. Le plus simple est d'installer `XCode` et de le lancer une fois, afin d'accepter les licences, avant d'installer `qt-opensource-mac-x64-clang-X.X.X.dmg`.

#### Windows

Pour Windows, les versions actuelles de Qt intègrent la totalité des outils permettant le développement d'application Qt: la librairie elle-même, QtCreator, une chaîne de compilation complète (`mingw`) ainsi que le débogueur. Lors de l'installation, on peut choisir les éléments à installer. Pour limiter la taille sur le disque dur à moins de 4Go, choisir "*Qt 6.X for desktop development*" ou X représente la dernière version, ou celle demandée pour votre projet de développement. Un exemple est donné sur la figure 3.2.

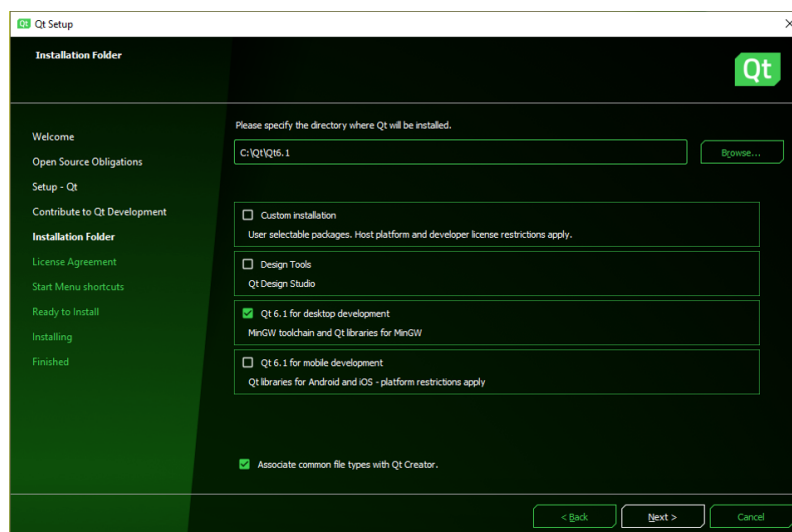


Figure 3.2: Elements pour réduire l'empreinte de l'installation de Qt. Le numéro de version peut varier!

### 3.3 Création d'un projet

Les IDE imposent la création d'un projet pour proposer l'automatisation des constructions et débogage de vos programmes. En fait ces projets permettent de configurer chacune des étapes de construction, le compilateur et ses options, les bibliothèques et leurs chemins. Heureusement, les IDE proposent de nombreux modèles de projets répondant à la plus part des configurations classiques.

La suite de ce paragraphe illustre la création d'un projet C++ standard.

La première étape consiste à créer un projet. Pour cela, il faut suivre les instructions suivantes:

- à partir du menu faire: **File** → **New file or project**;
- dans la fenêtre qui apparait choisir: **Project non-Qt**;
- dans le cadre d'une application basique choisir **C++ Project**;
- enfin, cliquer sur **Choose**.

La figure 3.3 donne un aperçu de la fenêtre de dialogue de sélection du type de projet.

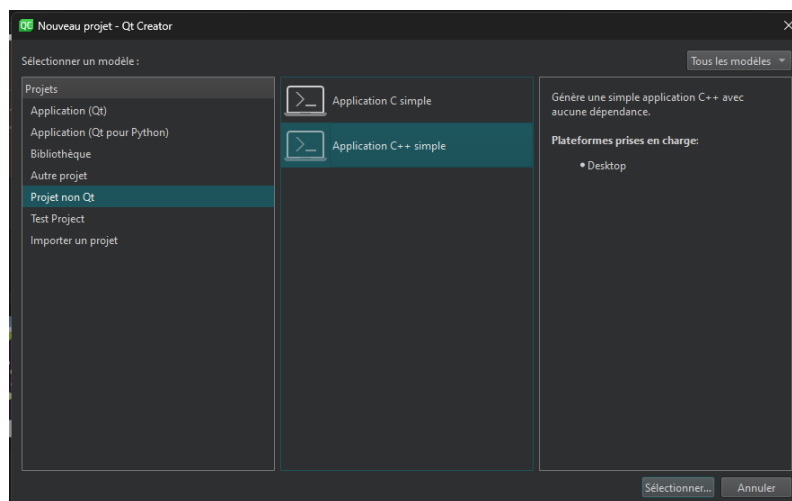


Figure 3.3: Fenêtre de dialogue pour la sélection du type de projet sous QtCreator

La deuxième étape consiste à donner un emplacement et un nom à votre projet (figure 3.4).

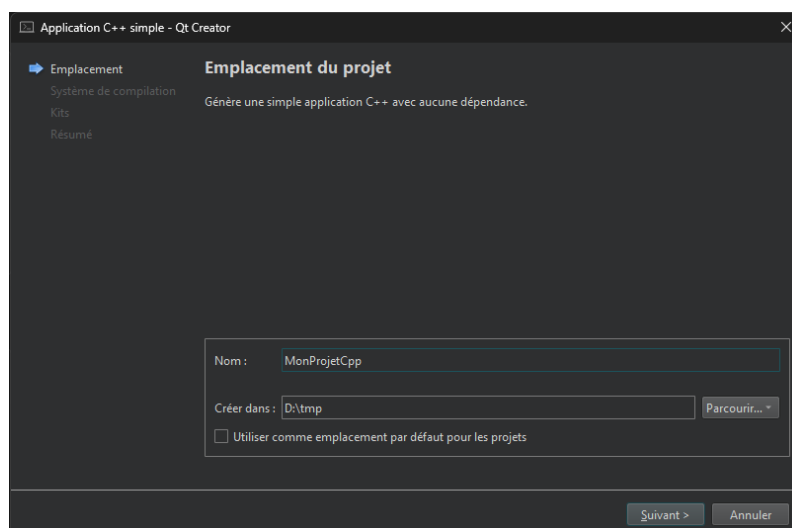


Figure 3.4: Fenêtre de dialogue pour l'emplacement et le nom du projet

La fenêtre qui suit doit être similaire à celle de la figure 3.5. Elle présente les 'kit' disponibles et utilisés pour le projet en cours de création. Dans l'exemple de la figure, tout est opérationnel et on peut laisser la sélection effectuée.

Cependant, il est possible qu'aucun kit n'apparaisse. Dans ce cas, votre environnement de développement n'est pas correctement configuré ou installé et il faut se reporter au paragraphe 3.8.

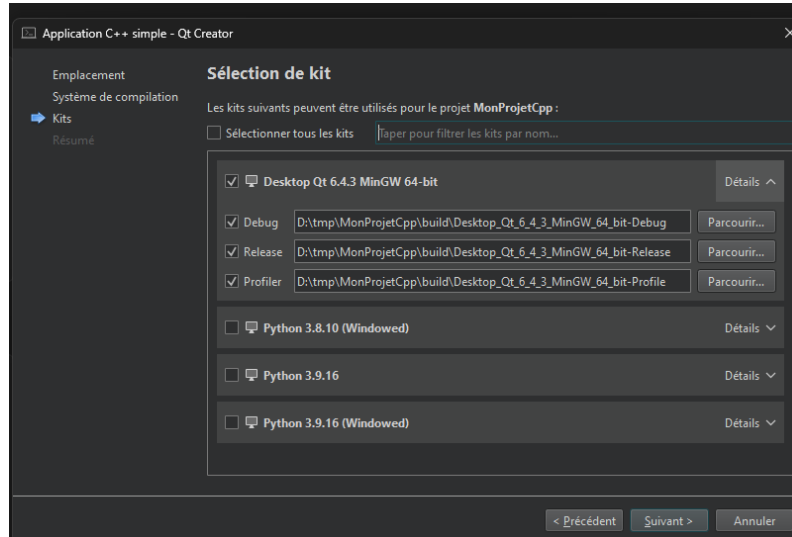


Figure 3.5: Fenêtre de création de projet, où l'environnement est opérationnel.

Ensuite vient la sélection du système gestionnaire de projet. La figure 3.6 illustre le choix de qmake qui est souvent recommandé<sup>3</sup>.

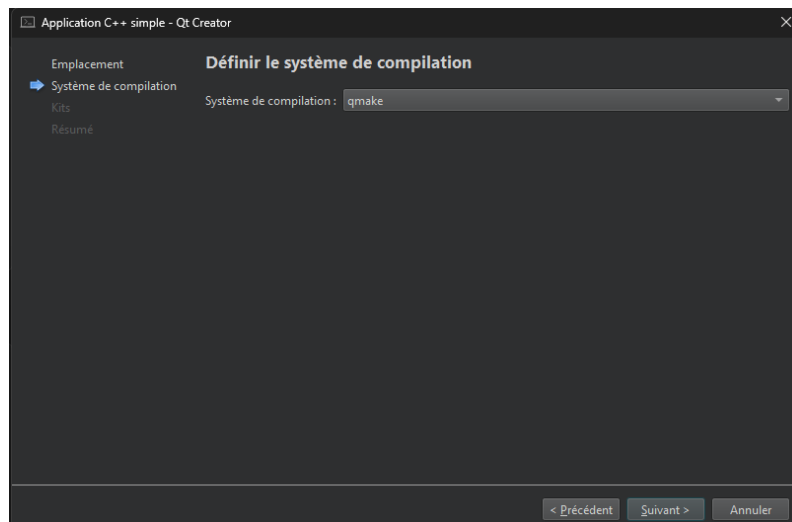


Figure 3.6: Fenêtre de création de projet, choix du système de gestion de projet. Ici qmake.

L'étape suivante est le résumé de la création du projet et le choix d'un gestionnaire de version<sup>4</sup>. Si l'option "Terminer" est disponible vous pouvez la choisir (figure 3.7).

<sup>3</sup>L'alternative CMake est très intéressante, dans les nouvelles version de Qt, qmake génère un CMakeList.txt automatiquement

<sup>4</sup>Un système de gestion de version permet d'archiver les modifications successives faites à vos fichiers par un ou plusieurs utilisateurs. Un des outils indispensables pour le travail collaboratif! Les plus connus sont CVS, SVN, GIT,

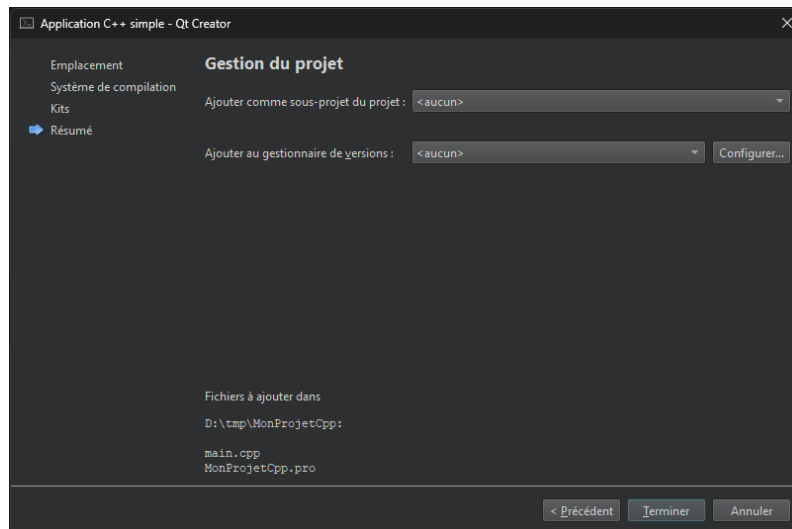


Figure 3.7: Fenêtre de résumé de création de projet et choix d'un outils de gestion de version (aucun).

### 3.4 Compilation

Il existe différentes échelles de compilation par rapport à votre projet:

- compilation d'un fichier. Ceci permet de corriger les erreurs de syntaxe éventuelles. Cette étape n'existe pas dans QtCreator. Les erreurs de syntaxe sont soulignées en rouge pendant l'écriture du code (en vert : les *warnings*). Cette étape est néanmoins présente dans de nombreux autres IDE, en général accessible de la façon suivante: à partir du menu **Build** → **Compile**;
- compilation de l'ensemble du projet. Un projet est généralement constitué d'un ensemble de fichiers (.h, .cpp). Cette compilation permet de tester la syntaxe de l'ensemble des fichiers ainsi que leurs interactions. De plus, avant la compilation, tous les fichiers sont enregistrés et les modifications récentes sont prises en compte par le compilateur. Cette étape s'effectue de la façon suivante: à partir du menu faire: **Build** → **Build Project xxx**, ou via le raccourci *ctrl+B*;
- compilation de l'ensemble des projets. Il est possible qu'une application soit basée sur plusieurs projets. La compilation de l'ensemble des projets se fera par **Build** → **Build all**, raccourci *ctrl+Maj+B*, ou via l'icône en bas à gauche de l'application (marteau) (figure 3.8);

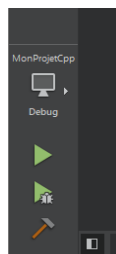


Figure 3.8: Boutons de exécution rapide. De haut en bas : choix du projet en cours, exécution en mode *release*, exécution en mode *debug* et tout compiler

Lors de la compilation, les erreurs de syntaxe détectées sont signalées dans l'onglet *Building Errors*



(figure 3.9). Il est important de noter que l'erreur est systématiquement décrite et qu'un double clic sur cette description envoie le curseur à la ligne correspondant à l'erreur détectée dans le fichier concerné. Ceci facilite grandement la résolution d'erreur de syntaxe dans les fichiers contenant le code. La figure 3.9 donne un exemple de résultat de compilation avec un message d'erreur.

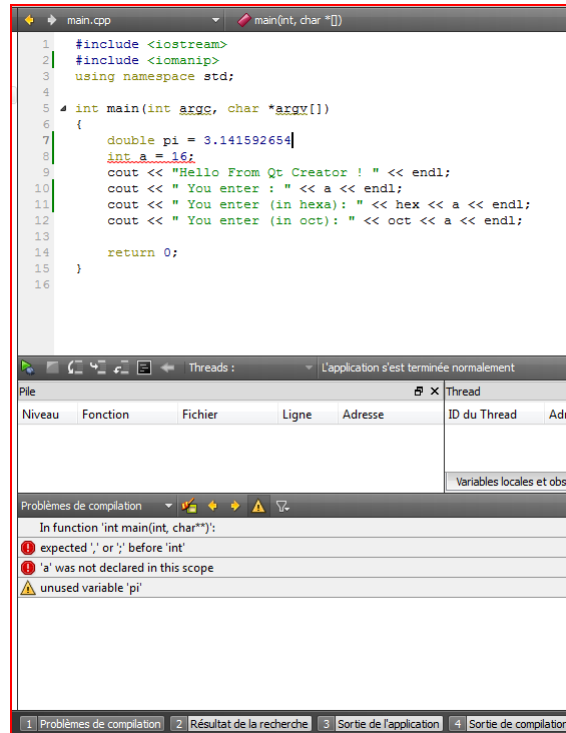


Figure 3.9: Exemple d'erreurs de compilation. remarquer le souligné rouge. Ici il ne manque qu'un ; à la fin de la ligne 7 comme indiqué par le message d'erreur.

Une fois l'ensemble du projet compilé (aucune erreur de compilation, ni de liage), il est possible d'exécuter le programme.

- R** Pendant l'édition, une analyse de syntaxe est faite. Cela est très pratique pour corriger les petites erreurs avant la compilation. Cependant, il est possible que des erreurs "fantômes" apparaissent (symbolisées par un cercle rouge dans la marge, notamment sur les cin, cout, etc). Elles viennent du fait que l'analyseur de syntaxe à la volée n'est pas C++ mais le Clang. Pour désactiver le Clang : aller dans *Aide* → "**A propos des plug-ins...**" et désactiver dans C++ le *ClangCodeModel* comme montré sur la figure 3.10. Il faudra redémarrer QtCreator pour que la modification soit prise en compte.

### 3.5 Exécutions des programmes

Il est possible d'exécuter un programme compilé de plusieurs façons :

- à partir du menu **Build** → **Execute**,
- par le raccourci **ctrl+R**,
- en cliquant sur le bouton de compilation rapide **play** (ou debug)(cf. figure 3.8);

Ces méthodes font toutes les mêmes opérations: enregistrer les fichiers sources, compiler le programme s'il n'existe pas ou qu'il n'est pas à jour (prise en compte de vos dernières modifications), exécuter le programme.

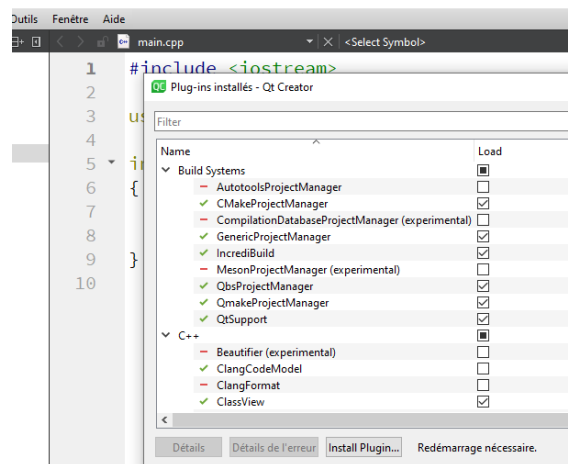


Figure 3.10: Désactivation du modèle Clang pour l'analyse de syntaxe à la volée.

### 3.6 Interactions utilisateur lors de l'exécution en mode console

Il est commun qu'un programme affiche des informations ou demande des valeurs à l'utilisateur. Quand le programme est un programme dit *console*, c'est à dire sans interface graphique, et que l'on souhaite interagir avec lui lors de son exécution (*cin*), il faut expliciter à QtCreator que l'on souhaite utiliser une console pour l'exécution. Car par défaut, sous QtCreator, un programme va afficher dans l'onglet 3 "Sortie de l'application" ses informations (voir le bas de la figure 3.9). Et dans cet onglet, par défaut, rien n'est prévu pour que l'utilisateur entre des données.

#### 3.6.1 Execution dans un terminal

Pour rendre ceci possible, il faut exécuter le programme dans un terminal (ou une console) qui sera capable de prendre en compte les interventions de l'utilisateur. Ceci est possible depuis QtCreator. Il s'agit de modifier les paramètres d'exécution du projet: **Projets** → **Paramètres d'exécution** cocher "Exécuter dans un terminal" (figure 3.11). Sur cette figure, les indications en rouge correspondent à l'ordre des clics pour accéder à cette page.

#### 3.6.2 Passage de paramètres lors de l'exécution

Il est possible de transmettre des arguments à un programme dès son lancement. Ces arguments sont vus comme des paramètres de la fonction `main`. On peut ensuite les utiliser via des variables dans le reste du programme. Cela permet d'automatiser des exécutions, sans demander à l'utilisateur d'entrer à chaque exécution ces valeurs.

Pour préciser ces arguments à l'exécution, il faut compléter le cadre "Arguments" de la fenêtre "Paramètres d'exécution" comme indiqué dans la figure 3.11.

**R** Pour vos projets en mode console, il est recommandé de toujours cocher la case "Exécuter dans un terminal".

#### 3.6.3 Terminal interne ou externe

Dans les versions récentes de QtCreator, un terminal interne est proposé. Lors de l'exécution du programme, il permet d'interagir avec l'utilisateur en restant dans QtCreator et supporte ainsi des fonctionnalités avancées pour l'affichage de caractères et d'historique. Il est aussi très utile quand aucun terminal n'existe sur la plateforme ciblée pour les développements.

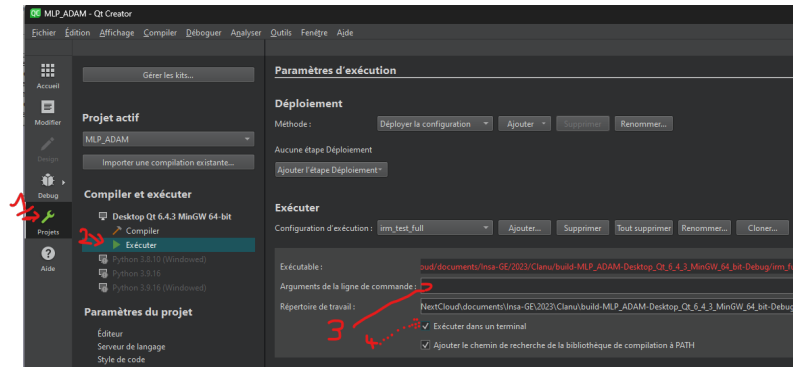


Figure 3.11: Exécution dans un terminal du programme et passage d'arguments en ligne de commande.

Cependant, quand un terminal existe sur le système, il peut être plus réaliste d'exécuter le programme comme il le sera après les développements : sans l'environnement QtCreator. Pour utiliser un terminal système au lieu de celui "interne" proposé par QtCreator, il faut se rendre dans les préférences de QtCreator *Édition* → *Préférences* puis, dans "Terminal", décocher la case "Utiliser le terminal interne" (voir la figure 3.12). Ainsi, le terminal par défaut du système d'exploitation sera utilisé pour les exécutions de programmes.

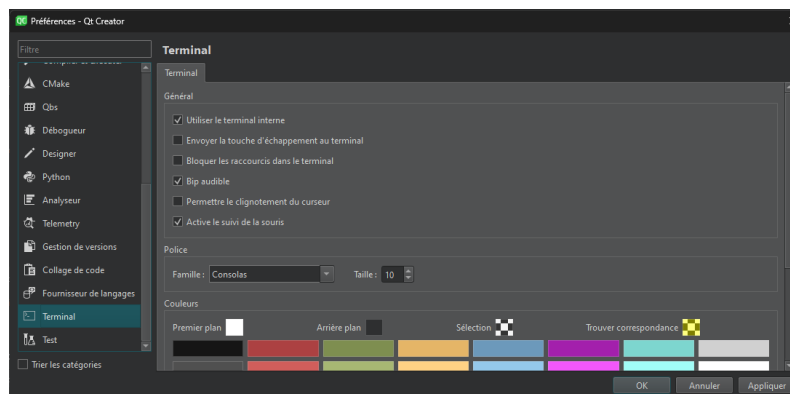


Figure 3.12: Choix du terminal, ici interne, pour les exécutions de programmes dans les préférences de QtCreator.

### 3.7 Débogage

Les deux principaux modes de compilation sont *release* et *debug*. Sous QtCreator, seuls ces deux modes sont disponibles. Pour chaque projet, et à chaque compilation, il est possible de choisir le mode de compilation. Contrairement au mode *release*, le mode *debug* permet d'effectuer le débogage d'un programme. Le débogage d'un projet permet d'interagir avec le programme pendant son exécution. Il est ainsi possible de surveiller les valeurs des variables, contrôler les passages dans les tests, l'exécution de boucles, l'allocation des objets, modifier les valeurs de variables ...

Le débogage d'un programme s'effectue de la façon suivante:

- positionner dans le code un ou plusieurs points d'arrêt *breakpoint*. Cette étape s'effectue en cliquant (bouton gauche) juste avant le numéro de la ligne où l'on souhaite insérer un breakpoint, ou en pressant **F9** sur la ligne voulue;
- exécuter le programme en mode *debug*: à partir du menu faire *Debug* → *Start Debugging*

→ **Start Debugging**, ou **F5** ou en cliquant sur le bouton de compilation rapide debug (figure 3.8).

L'exécution du programme démarre normalement (ainsi que la compilation si cela est nécessaire). Mais l'exécution sera alors mise en pause, juste avant que la ligne où est positionné le *breakpoint* ne soit exécutée.

Une fois le programme mis en pause, on peut interagir avec les variables et la progression de l'exécution en utilisant les boutons de la barre d'outil **Step Over** (**F10**) et **Step Into** (**F11**). La figure 3.13 donne un exemple de débogage d'un programme.

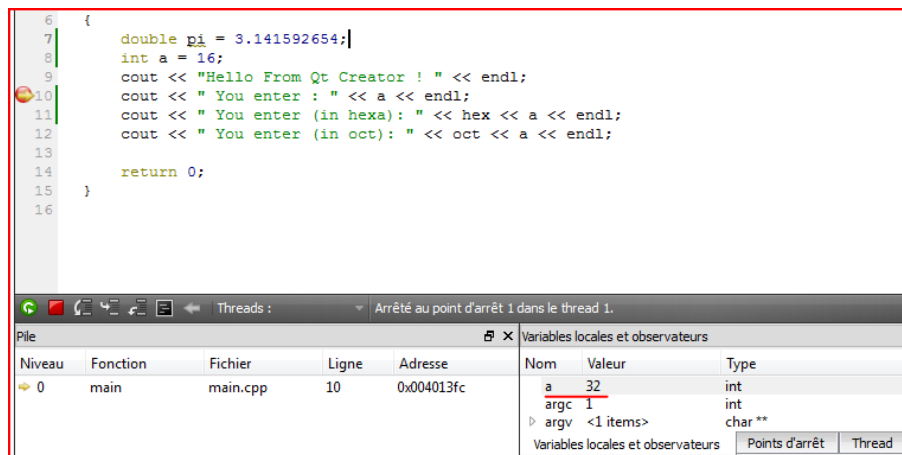


Figure 3.13: Exemple de débogage d'un programme sous QtCreator

### 3.8 Paramétrage du compilateur

Il est possible de changer les options de compilation de l'environnement. Dans QtCreator, il s'agit des "kits". Pour être fonctionnel, un kit doit disposer d'un compilateur C, d'un compilateur C++ (et d'un éditeur de lien si non pris en charge par le "compilateur"), optionnellement d'un débogueur et d'un générateur de Makefile (gmake de Qt, cmake, ...).

Ces réglages se font via "Kits" de la fenêtre "*Préférences*" accessible par **Édition** → **Préférences**. La figure 3.14 montre un exemple pour Windows de cette fenêtre.

**R** Si vous avez des problèmes de création de projet ou de compilateur non trouvé, et ce, dès votre première utilisation de QtCreator, vérifier :

- que vous disposez bien d'un compilateur et que celui-ci est reconnu et vu dans QtCreator (vous pouvez spécifier le chemin des compilateurs C++ et C au besoin). Très fréquemment, une absence de compilateur vient que vous avez installé QtCreator et non Qt comme recommandé dans le paragraphe 3.2 et suivants.
- qu'une version de Qt soit présente. Si aucune n'est disponible, aller dans l'onglet *Version de Qt* pour en trouver une. De même, s'il n'y en a pas, ce problème vient souvent du fait que vous avez juste installé QtCreator et non Qt avec les éléments recommandés.
- que l'architecture de votre PC corresponde à celle du compilateur. Ceci peut arriver quand on utilise différents compilateurs et plateformes. Il faut dans ce cas bien vérifier ceux utilisés et au besoin ajouter s'il en manque.

**Sans ceci, vous ne pourrez pas compiler, voire ne même pas pouvoir créer de projet.**

Il peut être recommandé d'ajouter le chemin vers votre chaîne de compilation dans la variable d'environnement PATH si elle n'est pas présente (souvent nécessaire sous Windows avec cmake).

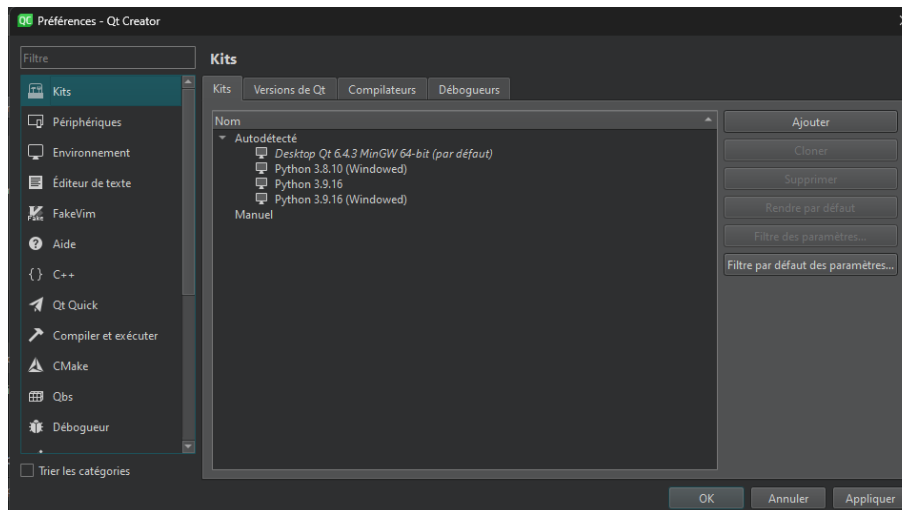


Figure 3.14: Fenêtre de configuration des outils de compilation sous QtCreator. Il s'agit ici d'un environnement Windows. Pour le développement C++, il faudra veiller à avoir un environnement avec Qt et un compilateur (ici MingGW). Ces éléments sont normalement installés et reconnus automatiquement par QtCreator.

Sans suppression de votre part de fichiers, une fois un environnement correctement configuré et fonctionnel est pérenne.





# Partie B: Le langage

<b>4</b>	<b>Introduction à la programmation</b> . . . .	<b>35</b>
<b>5</b>	<b>Mémento de la syntaxe C++</b> . . . . .	<b>37</b>
5.1	Syntaxe élémentaire	
5.2	Mots clés et mots réservés	
5.3	Variables et types	
5.4	Bases et systèmes de numération	
5.5	Opérateurs	
5.6	Opérations binaires et logiques	
5.7	Structures conditionnelles	
5.8	Structures de boucles	
5.9	Pointeurs et Références	
5.10	Tableaux	
5.11	Fonctions	
<b>6</b>	<b>Flux et interactions utilisateurs</b> . . . . .	<b>63</b>
6.1	Flux d'entrée cin et de sortie cout	
6.2	Flux de sortie pour l'affichage: cout	
6.3	Flux d'entrée clavier: cin	
6.4	Cas des chaînes de caractères	
<b>7</b>	<b>Flux et Fichiers</b> . . . . .	<b>67</b>
7.1	Fichiers	
7.2	Mode texte	
7.3	Mode binaire	
7.4	Et en C?	





## 4. Introduction à la programmation

Un grand nombre de langages de haut-niveaux comme le C++ sont des langages *impératifs* : ils décrivent une suite d'instructions qui doivent être exécutées consécutivement par le processeur. C'est un paradigme auquel nous sommes très familier: une recette de cuisine, un entraînement sportif, une partition de musique, un trajet en voiture, ... s'apparentent à une suite de 'choses à faire' avant de passer à la suivante et dans le but d'atteindre un objectif.

Pour revenir à la programmation, les instructions exécutées interagissent et modifient le contenu mémoire, les registres, les entrées/sorties, etc du système. On dit que l'état du système, à un instant donné, est défini par le contenu de la mémoire, des registres et autres entrées/sorties à cet instant et ainsi que le comportement de chaque instruction va dépendre de l'état du système au moment de l'exécution de l'instruction. Cette exécution va elle même faire évoluer l'état du système.

La base impérative dispose de 4 types d'instructions:

- **la séquence d'instructions ou bloc d'instructions** §5.1.1 qui décrit successivement une instruction, puis une autre, et ainsi de suite :  
`SortirPoelle; PrendreBeurre; FaireChaufferBeurreDansPoelle;`
- **les instructions d'assignation ou d'affectation** §5.5 qui permettent de manipuler des contenus mémoires et d'enregistrer un résultat en vue d'une utilisation ultérieure. Par exemple la séquence d'instructions  $a \leftarrow 2; x \leftarrow 2 + a;$  permet d'affecter 2 dans la variable nommée  $a$ , et ensuite de mettre 4 dans  $x$ . Les opérateurs arithmétiques et ceux du langage (et leurs combinaisons) peuvent être utilisés pour l'affectation.
- **les instructions conditionnelles** §5.7 qui permettent d'exécuter un bloc d'instructions que si une condition préalable est remplie:  
`si PoelleChaude alors MettreOeufsDansPoelle;`
- **les instructions de boucle** §5.8 qui vont permettre de répéter un bloc d'instructions un nombre déterminé de fois ou tant qu'une condition est vraie:  
`tant que OeufsPasCuits alors LaisserCuireUnPeuPlus;`

Un langage de haut niveau va permettre au programmeur de disposer: de plusieurs instructions de chaque type, de très nombreux opérateurs permettant de décrire des expressions très complexes, des outils pour développer de grosses applications et pour communiquer avec d'autres outils existants,

...

Le memento qui suit essaie de présenter tout ce que le C++ offre au programmeur. Il restera au programmeur le soin de faire sa propre recette.

## 5. Mémento de la syntaxe C++

### 5.1 Syntaxe élémentaire

Dans cette section, nous donnons les principales règles d'écriture d'un programme en C++. En effet, pour qu'un compilateur puisse interpréter un code en C++, il est nécessaire de respecter différentes règles résumées dans les trois paragraphes suivants.

#### 5.1.1 Instructions

**R** Toutes les instructions en C++ se terminent par ;. Le symbole ; ne changera jamais de rôle ou de sens.

Les instructions sont des suites de symboles (+, -, =, ( )) ou de lettres pouvant avoir un sens (*for*, *MaFonction()*, *double*, ...), aussi appelés opérateurs, permettant de manipuler des nombres ou des variables. Voici un exemple pour le calcul de  $s = \text{sinc}_\pi(x) = \frac{\sin(\pi x)}{\pi x}$

```
1 x = 1.1234 ;
2 pi = 3.141592 ;
3 s = sin( x * pi ) / ( x * pi );
```

**R** Le saut de ligne "entrée" ne permet pas de signaler la fin d'une instruction.

De manière générale, le saut de ligne, les espaces ainsi que la tabulation, peuvent être ajoutés n'importe où, sauf pour:

- l'écriture d'un chiffre (100000 et non 100 000);
- le nom des fonctions, opérateurs à deux symboles (== par exemple) et variables;
- les mots clés.

Un bloc d'instructions permet de grouper plusieurs instructions. La définition d'un bloc d'instructions (ou *scope*) se fait avec des **accolades**:

```

1  {
2      instruction1;
3      instruction2;
4      ...;
5  }
```

La durée de vie (création et destruction) ainsi que la portée (possibilité d'utilisation / d'accès ...) de tout ce qui est défini à l'intérieur d'un bloc d'instructions est limitée à ce bloc. Ces blocs sont particulièrement utiles pour définir les instructions des fonctions ou des structures de boucles ou de tests.

**R** Les espaces et tabulations, on parle d'indentation du code, ne permettent pas de créer un bloc d'instructions. Ils sont cependant bienvenus pour faciliter la lecture.

### 5.1.2 Commentaires

Au même titre que l'indentation, il est important de commenter son code, c'est à dire d'ajouter des informations non compilées dans le code source à destination des programmeurs ou utilisateurs afin de mieux faire comprendre et se souvenir de ce qui a été programmé.

Deux styles de commentaire sont disponibles en C++:

- `//` permet de mettre en commentaire tout ce qui est après jusqu'au prochain saut de ligne;
- `/* ... */` permettent de délimiter plusieurs lignes de commentaires (un bloc de commentaires).

Voici un exemple d'utilisation de commentaires en C++:

```

1  int b;    // variable utilisée pour compter des notes
2  nb = 10; /* par défaut on connaît
3             le nombre de notes qui est :
4             10... et toutes ces lignes sont en commentaires */
```

### 5.1.3 Casse

Le compilateur prend en compte la casse (majuscule ou minuscule) pour

- les noms des fichiers,
- les noms des variables et des fonctions,
- les mots clés du langage C++ et les directives du préprocesseur.

**R** Pour résumer, **la casse est toujours prise en compte.**

### 5.1.4 Symboles

La diversité des instructions nécessaires en informatique et le faible nombre de symboles communs font que les symboles sont utilisés pour plusieurs instructions différentes. Ainsi, le rôle et le sens des symboles changent en fonction du contexte de leur utilisation! Cependant, il ne doit toujours exister qu'**une unique manière de comprendre l'instruction** et l'utilisation du symbole.

Les accents, cédilles et autres subtilités ne sont pas acceptées par le compilateur. Ils sont donc interdits dans les noms de fonctions, variables, classes, ... On peut cependant les utiliser dans les commentaires et les chaînes de caractères. Cependant, attention à leur affichage à l'écran: il pourra être erroné si le caractère n'est pas pris en charge par le périphérique d'affichage.

Le C, C++ et d'autres langages utilisent abondamment les accolades, crochets, dièse. Noter que les caractères { et } peuvent être remplacés par les digraphes `<%` et `%>`. Les caractères [ et ] par respectivement `<:` et `:>`. Puis le caractère # par le digraphe `%:`.

Enfin, le symbole `\` ne peut être utilisé que dans les chaînes de caractères et les commentaires. Il a un rôle particulier de caractère d'échappement, notamment pour représenter des caractères non affichables (entrée, tabulation, ...). Ainsi, sous Windows, il est recommandé de ne pas l'utiliser pour donner le chemin d'accès à un fichier via une chaîne de caractères (préférer `/`).

## 5.2 Mots clés et mots réservés

Les mots clés (ou mots réservés) constituent, avec les opérateurs, le vocabulaire natif du langage, c'est à dire l'ensemble initial des mots à partir desquels il est possible de faire vos phrases.

### 5.2.1 Mots clés

Les mots clés sont les mots reconnus par le compilateur.

Le tableau 5.1 fournit la liste complète des mots clés.

<code>alignas</code>	<code>alignof</code>	<code>and</code>	<code>and_eq</code>	<code>asm</code>	<code>auto</code>	
<code>bitand</code>	<code>bitor</code>	<code>bool</code>	<code>break</code>			
<code>case</code>	<code>catch</code>	<code>char</code>	<code>char16_t</code>	<code>char32_t</code>	<code>class</code>	<code>compl</code>
<code>const</code>	<code>constexpr</code>	<code>const_cast</code>	<code>continue</code>			
<code>decltype</code>	<code>default</code>	<code>delete</code>	<code>do</code>	<code>double</code>	<code>dynamic_cast</code>	
<code>else</code>	<code>enum</code>	<code>explicit</code>	<code>export</code>	<code>extern</code>		
<code>false</code>	<code>float</code>	<code>for</code>	<code>friend</code>			
<code>goto</code>						
<code>if</code>	<code>inline</code>	<code>int</code>				
<code>long</code>						
<code>mutable</code>						
<code>namespace</code>	<code>new</code>	<code>noexcept</code>	<code>not</code>	<code>not_eq</code>	<code>nullptr</code>	
<code>operator</code>	<code>or</code>	<code>or_eq</code>				
<code>private</code>	<code>protected</code>	<code>public</code>				
<code>register</code>	<code>reinterpret_cast</code>	<code>return</code>				
<code>short</code>	<code>signed</code>	<code>sizeof</code>	<code>static</code>	<code>static_assert</code>	<code>static_cast</code>	
<code>struct</code>	<code>switch</code>					
<code>template</code>	<code>this</code>	<code>thread_local</code>		<code>throw</code>	<code>true</code>	<code>try</code>
<code>typedef</code>	<code>typeid</code>	<code>typename</code>				
<code>union</code>	<code>unsigned</code>	<code>using</code>				
<code>virtual</code>	<code>void</code>	<code>volatile</code>				
<code>wchar_t</code>	<code>while</code>					
<code>xor</code>	<code>xor_eq</code>					

Table 5.1: Liste des mots clés du langage C++

### 5.2.2 Directives

Les directives du préprocesseur ne sont pas des instructions du langage. Elles permettent des adaptations du code ou de la compréhension du code par le compilateur. Ces directives commencent par le symbole `#`. Le tableau 5.2 liste les directives du préprocesseur.

Il est évident que les mots clés et les directives du préprocesseur ne peuvent pas être utilisés par le programmeur pour nommer ses propres variables, ses objets et ses fonctions<sup>1</sup>.

Voici un exemple d'utilisation de mots clés et de directives du préprocesseur:

<sup>1</sup>Il pourra cependant en surcharger quelques uns...

#define			
#elif	#else	#endif	#error
#if	#ifdef	#ifndef	#include
#line			
#pragma			
#undef			

Table 5.2: Liste des 12 directives du préprocesseur

```

1 // Ces 3 directives sont necessaire a tous fichiers ".h", la
2 // variable "_MaClasse_H_" doit etre differente pour chaque fichier
3 #ifndef _MaClasse_H_
4 #define _MaClasse_H_
5
6 #define PI 3.14 // pas de type devant la variable ni de "=" ni de ";"
7
8 // le reste du fichier .h
9 class MaClasse
10 {
11 ...
12 };
13
14 #endif

```

### 5.3 Variables et types

Dans cette section, nous donnons les principales notions à connaître sur les types et la déclaration de variables. Le C et le C++ sont des langages fortement typés. Cela signifie que le langage s'appuie sur les types des données et fonctions pour adapter son comportement.

En C et C++, le type d'une donnée (ou fonction) ne pourra pas évoluer au cours du programme et le type de chaque variable et fonction doit être connu à la compilation.

#### 5.3.1 Types fondamentaux en C++

Le tableau 5.3 fournit une liste des types fondamentaux utilisés en programmation C/C++.

Type	Taille	Intervalle de valeurs	Précision
bool	1 octet	<i>false</i> ou <i>true</i>	-
char	1 octet	-128 à 127	-
unsigned char	1 octet	0 à 255	-
short	2 octets	-32768 à 32767	-
unsigned short	2 octets	0 à 65535	-
int	4 octets	-2147483648 à 2147483647	-
unsigned (int)	4 octets	0 à 4294967295	-
long	4 octets	-2147483648 à 2147483647	-
unsigned (long)	4 octets	0 à 4294967295	-
float	4 octets	+/- 3.4 * 10 <sup>-38</sup> à 3.4 * 10 <sup>38</sup>	6 chiffres
double	8 octets	+/- 1.7 * 10 <sup>-308</sup> à 1.7 * 10 <sup>308</sup>	15 chiffres
long double	10 octets	+/- 1.2 * 10 <sup>-4932</sup> à 1.2 * 10 <sup>4932</sup>	18 chiffres

Table 5.3: Types fondamentaux en C++

La taille des variables de type `short`, `int` et `long` étant sujette à l'architecture du processeur, de nouveaux types plus précis sont proposés dans la bibliothèque `cstdint` (ou `stdint.h` pour le C) afin de garantir la taille des entiers. Ceci est très précieux pour la programmation des systèmes embarqués. La table 5.4 donne ces types spécifiques.

Type signé	Type non-signé	Taille	Remarque
<code>int8_t</code>	<code>uint8_t</code>	1 octet	-
<code>int16_t</code>	<code>uint16_t</code>	2 octets	-
<code>int32_t</code>	<code>uint32_t</code>	4 octets	-
<code>int64_t</code>	<code>uint64_t</code>	8 octets	-
<code>intmax_t</code>	<code>uintmax_t</code>	≤ 8 octets	entier de taille maximum supportée (max 64)
<code>int_leastX_t</code>	<code>uint_leastX_t</code>	X = 8, 16, 32, 64	entier de taille minimum X
<code>int_fastX_t</code>	<code>uint_fastX_t</code>	X = 8, 16, 32, 64	entier de taille minimum X, pouvant être surdimensionné si les performances sont identiques
<code>intptr_t</code>	<code>uintptr_t</code>	-	codage et manipulation des adresses ( <code>void *</code> )

Table 5.4: Types entiers de `cstdint`

### 5.3.2 Déclaration de variables

La syntaxe de déclaration d'une variable en C++ est la suivante:

```
1  Type variable;
2  Type var1, var2, var3;
```

Les noms de variables doivent commencer par une lettre ou par `_`. Les accents sont interdits. La casse (majuscule ou minuscule) est prise en compte (`var1` est différent de `Var1`). Exemples:

```
1  int i,j,k;
2  float Valeur;
```

### 5.3.3 Déclaration de variables avec affectation

La déclaration d'une variable avec affectation d'une valeur (ou initialisation) se fait de la façon suivante:

```
1  type var1 = 0;
```

qui est équivalent à<sup>2</sup>:

```
1  type var1;
2  var1 = 0;
```

Voici une liste d'exemples de déclaration de variables avec affectation:

```
1  float a=3.002;
2  unsigned short b=1,c=2;
3  double un_mille(1.609e3);
```

L'opération d'affectation peut se faire entre des variables de même type:

```
1  int a,b;
2  a=4;
3  b=a; // la valeur de a est copiée dans b: b vaut 4
```

<sup>2</sup>algorithmiquement... pas forcément en terme d'exécution



ou entre des variables de types différents (dans ce cas, attention aux pertes lors de la conversion):

```

1  float a=3.1415;
2  char b;
3  double c;
4  b = a;    // b vaut 3, un warning est genere a la compilation
5  c = a;    // c vaut 3.1415, pas de warning ici car la precisison
6            // du type double est superieure a la precision du
7            // type float (pas de perte de precision)

```

## 5.4 Bases et systèmes de numération

En C/C++ il est possible de décrire les nombres dans quatre bases : binaire (0 ou 1), octal (de 0 à 7), décimal (de 0 à 9) et hexadécimal (de 0 à 9 puis 'a' à 'f'). Une syntaxe particulière permet d'identifier la base utilisée pour représenter un nombre: il s'agit de caractères présents au début du nombre. Voici la même valeur représentée dans ces quatre systèmes:

- Le système décimal est celui utilisé par défaut et ne nécessite aucun symbole particulier:

```
int d = 1978;
```

- Le système hexadécimal sera utilisé pour les nombres commençant par les symboles "zéro x" : 0x

```
int h = 0x7BA; //ou 0x7ba
```

- Le système octal sera utilisé pour les nombres commençant par le symbole zéro : 0

```
int o = 03672;
```

- Le système binaire sera utilisé pour les nombres commençant par les symboles "zéro b" : 0b

```
int b = 0b11110111010;
```

Depuis le C++14, il est possible d'ajouter des apostrophes comme séparateur de nombre. Ainsi `int b = 0b111'1011'1010;` est plus facile à lire.

L'affichage des nombres dans des bases spécifiques est traité au paragraphe 6.2.

## 5.5 Opérateurs

Nous abordons dans cette partie les définitions et les différentes propriétés des opérateurs du C++. Comme dit précédemment, un symbole (par exemple \*) peut avoir différentes significations en fonction de son contexte d'utilisation.

### 5.5.1 Opérateurs unaire, binaire et ternaire

Un opérateur unaire est un opérateur ne possédant qu'un seul opérande. Voici un exemple d'un tel opérateur:

```

1  int a; // declaration de a
2  &a;    // le "et commercial" est un operateur unaire,
3         // il renvoie l'adresse memoire de a

```

Un opérateur binaire possède deux opérandes (à ne pas confondre avec les opérateurs réalisant des opérations en binaire, qui sont parfois (et, ou ...) des opérateurs binaires). Voici un exemple d'un tel opérateur:

```

1  int a; // declaration de a
2  a = 1; // = est un operateur binaire, il affecte la valeur 1 a "a"

```

Il n'existe qu'un seul opérateur ternaire, l'opérateur de condition ?: qui est semblable à la structure conditionnelle `if else` (voir paragraphe 5.7.1).



### 5.5.2 Priorité des opérateurs

La priorité fixe l'ordre dans lequel les opérateurs seront interprétés par le compilateur. Les opérateurs en C++ sont répartis selon 17 niveaux de priorité. La priorité la plus basse vaut 1 et la priorité la plus haute vaut 17. La règle de priorité des opérateurs est la suivante: si dans une même instruction sont présents des opérateurs avec des niveaux de priorité différents, les opérateurs ayant la plus haute priorité sont exécutés en premier. Par exemple:

```
1   A op1 B op2 C;
```

si la priorité de *op2* est supérieure à la priorité de *op1*, on a:

```
1   A op1 ( B op2 C );
```

De manière générale, l'ajout de parenthèses dans les expressions enlève les ambiguïtés de priorité. Le tableau 5.5.3 fournit la liste des opérateurs en C++ avec leur niveau de priorité.

### 5.5.3 Associativité des opérateurs

L'associativité désigne la direction dans laquelle sont exécutés les opérateurs d'un même niveau de priorité. Par exemple, la multiplication et la division ont le même niveau de priorité et leur associativité est de gauche à droite:

```
1   int a = 3 * 4 / 2 * 3; // a = 18
2   int b = 3 * 4 / ( 2 * 3); // b = 2
```

De manière générale, l'ajout de parenthèses dans les expressions enlève les ambiguïtés d'associativité. Le tableau 5.5.3 fournit la liste des opérateurs en C++ avec leur niveau d'associativité.

## 5.6 Opérations binaires et logiques

Les opérateurs vus précédemment s'appliquent aux différentes bases. Cependant, il est souvent nécessaire d'effectuer des opérations *booléennes*. Il faut noter dans la table 5.5.3 l'existence d'opérateurs assez proches (souvent confondus): les opérateurs binaires et les opérateurs logiques.

### 5.6.1 Opérateurs binaires

Les opérateurs binaires permettent de modifier le contenu d'une variable au niveau de ses bits. Par exemple, le code suivant réalise un masque en OU sur le premier bit de *a*:

```
1   a = a | 0x01; // qui peut s'écrire : a |= 0x01;
```

Ici l'opérateur `|` correspond au "OU" binaire (se reporter au tableau 5.5.3), c'est à dire bit à bit.

De même similaire le code suivant réalise un masque en ET:

```
1   b &= ~0x01;
```

L'utilisation du `~`, qui permet d'inverser les valeurs des bits, a l'avantage de s'effectuer sur l'ensemble des bits utiles. Ce nombre dépend du type de la variable.

L'opérateur binaire pour le "OU EXCLUSIF" est le symbole `^`! Donc  $4^2$  en C/C++ vaut 6.

### 5.6.2 Opérateurs logiques

Les opérateurs logiques permettent d'effectuer une équation logique sur des variables booléennes et ne modifient pas les variables.

Par exemple pour savoir si *A* et *B* (que l'on note *A.B*) est vraie, on écrira *A && B*. Ce test va retourner `true` ou `false`, c'est à dire respectivement 0 ou 1, en fonction des valeurs de *A* et de *B*. Toutes variable de type entier (char, short, int, unsigned int ...) peuvent être utilisées pour des tests logiques. Si la valeur d'une variable est égale à 0 alors elle est considérée comme `false`, sinon (1, 1000, -2, ...) comme `true`.

Prio.	Opér.	Signification	Associativité	Exemple
18	::	Résolution de portée (unaire)	(aucune)	
18	::	Résolution de portée (binaire)	(aucune)	
17	++	incrément suffixe	gauche à droite	a++x
17	--	décrément suffixe	gauche à droite	a--x
17	()	Appel de fonction	gauche à droite	MaFonction(x,y)
17	[]	Indexation de tableau	gauche à droite	Tab[i]
17	.	Sélection de composant par référence	gauche à droite	objet.methode();
17	->	Sélection de composant par pointeur	gauche à droite	this->methode();
17	typeid()	information de type (run-time)	gauche à droite	typeid(a); typeid(type);
17	const_cast	Cast constant	gauche à droite	
17	dynamic_cast	Cast dynamic	gauche à droite	
17	reinterpret_cast	Cast avec ré interprétation	gauche à droite	
17	static_cast	Cast static	gauche à droite	
16	(type)	Conversion de type explicite	droite à gauche	(double) x;
16	sizeof	Taille en octets	droite à gauche	sizeof(int);
16	&	Fournit l'adresse mémoire	droite à gauche	&a
16	*	Indirection (déréférence)	droite à gauche	
16	~	Négation binaire	droite à gauche	~a;
16	!	Négation logique	droite à gauche	!a;
16	+	Addition (unaire)	droite à gauche	+a;
16	-	Soustraction (unaire)	droite à gauche	-a;
16	++	Incrément préfixe	droite à gauche	++a;
16	--	Décrément préfixe	droite à gauche	--a;
16	new, new[]	Allocation mémoire dynamique	droite à gauche	
16	delete, delete[]	Dé-allocation mémoire dynamique	droite à gauche	delete[] t;
15	.*	Sélection de composant	gauche à droite	
15	->*	Sélection de composant (pointeur)	gauche à droite	
14	*	Multiplication	gauche à droite	a*b;
14	/	Division	gauche à droite	a/b;
14	%	Modulo	gauche à droite	a%b;
13	+	Addition (binaire)	gauche à droite	a+b;
13	-	Soustraction (binaire)	gauche à droite	a-b;

Prio.	Opér.	Signification	Associativité	Exemple
12	>>	Décalage des bits à droite	gauche à droite	a>>2;
12	<<	Décalage des bits à gauche	gauche à droite	a<<2;
11	>	Test strictement supérieur	gauche à droite	a>2;
11	>=	Test supérieur ou égal	gauche à droite	a>=2;
10	<	Test strictement inférieur	gauche à droite	a<2;
11	<=	Test inférieur ou égal	gauche à droite	a<=2;
10	==	Test d'égalité	gauche à droite	if (choix=='o')
10	!=	Test de non égalité	gauche à droite	if (pt!=NULL)
9	&	ET binaire	gauche à droite	a=1&3; //1
8	^	OU exclusif binaire	gauche à droite	a=1^3; //2
7		OU binaire	gauche à droite	a=1 2; //3
6	&&	ET logique	gauche à droite	if (a==1&&b<3)
5		OU logique	gauche à droite	if (a==1  b<3)
4	?:	Opérateur de condition (ternaire)	droite à gauche	
3	=	Affectation simple	droite à gauche	a=b=2;
3	+=	Affectation combinée +	droite à gauche	a+=2; //a=a+2
3	-=	Affectation combinée -	droite à gauche	a-=2; //a=a-2
3	*=	Affectation combinée *	droite à gauche	a*=2; //a=a*2
3	/=	Affectation combinée /	droite à gauche	a/=2; //a=a/2
3	%=	Affectation combinée %	droite à gauche	a%=2; //a=a%2
3	<<=	Affectation combinée <<	droite à gauche	a<<=2; //a*=4
3	>>=	Affectation combinée >>	droite à gauche	a>>=1; //a/=2
3	&=	Affectation combinée &	droite à gauche	a&=1;
3	^=	Affectation combinée ^	droite à gauche	a^=0xFF;
3	=	Affectation combinée	droite à gauche	a =0xFE;
2	throw	Envoie d'exception	droite à gauche	
1	,	Séquence d'expressions	gauche à droite	int a,b,c;

Table 5.5: Liste et propriétés des opérateurs en C++

Voici une utilisation des opérateurs logiques permettant de reproduire la première loi De Morgan  $\overline{A+B} \Leftrightarrow \overline{A}.\overline{B}$ :

```
1  !(A || B) est equivalent !A && !B
```

A noter que tous les opérateurs de tests et les opérateurs logiques du tableau 5.5.3 retournent une valeur booléenne: 0 pour `false` et 1 pour `true`. Ainsi `A == 0` retourne vrai si A est égale à zéro.

## 5.7 Structures conditionnelles

Les conditions servent à comparer (ou évaluer) des variables ou des valeurs. Une condition est soit vraie (`true`) ou fautive (`false`). Pour donner des exemples de conditions :

- `a > b`, sera vraie si  $a > b$
- `a == 1`, sera vraie si  $a = 1$
- `a`, sera vraie si  $a \neq 0$
- `!a`, sera vraie si  $a = 0$

### 5.7.1 if / else

La syntaxe d'une condition `if` est la suivante:

```
1  if ( condition ) // "condition" est une variable booléenne (true ou false)
2  {
3  ...           // instructions effectuees si test est vrai
4  }
```

La structure `if` peut être complétée d'un `else` permettant de traiter les cas où la condition sera fautive:

```
1  if ( condition ) // "condition" est une variable booléenne (true ou false)
2  {
3  ...           // instructions effectuees si test est vrai
4  }
5  else         // bloc else: n'est pas obligatoire
6  {
7  ...           // instructions effectuees si test est faux
8  }
```

La variable ou valeur `condition` est forcément de type booléen (`bool`). Après l'instruction `if`, les accolades sont nécessaires si plusieurs instructions sont à exécuter. Le bloc `else` n'est pas obligatoire. Il est à noter que la syntaxe `if(a)` est équivalent à `if( a!=0 )`.

```
1  bool a,b;
2  if( a==true ) // il s'agit bien de 2 signes "=="
3  {
4  a = false;
5  // std::cout permet d'afficher a l'ecran un message
6  std::cout << "Initialisation de la variable a false";
7  }
```

Les structures des tests peuvent s'imbriquer au tant que nécessaire. Bien penser aux accolades quand elles sont nécessaires (ou bien dans le doute!) et à l'indentation pour avoir un code lisible. Voici un exemple d'utilisation de structures `if` imbriquées en C++:

```
1  double moyenne;
2  cin >> moyenne; // demande a l'utilisateur sa moyenne
3  if ( moyenne >= 16.0 )
4  std::cout << "Mention tres bien";
5  else
```

```

6     if ( moyenne >= 14.0 )
7         std::cout << "Mention bien";
8     else
9         if ( moyenne >= 12.0 )
10            std::cout << "Mention assez bien";
11        else
12            if ( moyenne >= 10.0 )
13                std::cout << "Mention passable";
14            else
15                std::cout << "Non admis...";

```

L'opérateur ?: est une structure du type `if` puis `else` mais qui ne permet d'exécuter qu'une seule instruction qui est généralement un retour de valeur.

```

1     (condition) ? instruction si vrai : instruction si faux ;

```

Voici un exemple d'utilisation de cet opérateur où la variable `nb_point_en_moins` vaut 3 si la variable `vitesse` est strictement supérieure à 50, et 0 sinon.

```

1     #include <iostream>
2     int main (void)
3     {
4         int limitation = 50 ;
5         int vitesse;
6         int nb_point_en_moins;
7         // demander la vitesse
8         std::cin >> vitesse;
9         nb_point_en_moins = (vitesse > 50) ? 3 : 0;
10        // affichage
11        std::cout <<" Nombre de points en moins : ";
12        std::cout << nb_point_en_moins << std::endl;
13        return 0;
14    }

```

### 5.7.2 switch / case

L'imbrication de `if / else` peut avantageusement être remplacée par une structure `switch case` quand la **variable à tester est de type entier**. C'est à dire que la variable à tester est de type: `char`, `short`, `int`, `long` et `string`. Dans l'exemple de `if else` imbriqués (5.7.1), il n'est donc pas possible d'utiliser cette structure.

La syntaxe d'une condition `switch case` est la suivante:

 **Les crochets signalent des blocs optionels, ne pas écrire les crochets en C++!**

```

1     switch ( variable )
2     {
3         case constante_1:    // faire attention au ":"
4             instructions executees si variable == constante_1
5             [ break ; ] // le break n'est pas obligatoire... mais surement
                       // necessaire !
6         case constante_2:
7             instructions executees si variable == constante_2
8             [ break ; ]
9         ...
10        [ default: // l'etiquette default et ses instructions sont optionnelles
11            instructions executees si aucun des

```

```

12     cas precedents ne sont executes ]
13 }

```

Les blocs d'instructions n'ont pas besoin d'accolades. Généralement chaque bloc se termine par l'instruction `break;` qui saute à la fin de la structure `switch` (l'accolade fermante). Si un bloc `case` ne se termine pas par une instruction `break;`, à l'exécution le programme passera au bloc `case` suivant. Voici un exemple d'utilisation de structure `switch case` en C++:

```

1  char choix;
2  // cout permet d'afficher a l'ecran un message
3  cout << "Etes vous d'accord ? O/N";
4  // cin permet de recuperer une valeur saisie au clavier
5  cin >> choix;
6  switch( choix )
7  {
8  case 'o':
9  case 'O':
10     cout << "Vous etes d'accord";
11     break;
12 case 'n':
13 case 'N':
14     cout << "Vous n'etes pas d'accord";
15     break;
16 default:
17     cout << "Repondre par o/n ou O/N";
18 }

```

Dans cet exemple, l'utilisateur va rentrer 'O' ou 'N', mais s'il tape 'o' les instructions du cas 'O' (ligne 10 et 11) seront exécutées (respectivement pour 'n' et les lignes 14 et 15).

## 5.8 Structures de boucles

Les boucles en C++ servent à répéter un ensemble d'instructions. Comme en C, il existe en C++ 3 structures de boucle: `for`, `while` et `do / while`.

### 5.8.1 for

La boucle `for` est à privilégier quand on connaît la valeur initiale, la valeur finale et l'incrément à utiliser (notamment pour les tableaux). La syntaxe d'une boucle `for` en C++ est la suivante:

```

1  for ( initialisations; condition; post-instructions )
2  {
3  // Instructions de boucle r'ep'et'ees tant que la condition est vraie
4  }

```

Il n'y a pas de ; à la fin de la ligne du `for`. On rappelle que les crochets ne sont pas à écrire et permettent dans ce document de désigner les blocs optionnels. On remarque que tous les blocs de la boucle `for` sont optionnels! Voici le détail de l'exécution et la signification de chacun de ces blocs. L'exécution d'une boucle `for` se déroule en 3 étapes.

1. Le bloc [initialisations] est toujours exécuté et une seule et unique fois,
2. Le bloc [condition] est évalué à chaque itération,
3. Si la condition est vraie, les instructions de boucle puis le bloc [post-instructions] sont exécutés puis l'étape 2 est répétée. Si la condition est fausse, ni les instructions de boucle ni le bloc [post-instructions] ne sont exécutés et la boucle se termine (on sort de la boucle et les instructions qui suivent sont exécutées).

Les blocs [initialisations] et [post-instructions] peuvent contenir aucune ou plusieurs instructions séparées par des virgules. Le bloc [condition] peut être vide, dans ce cas on a généralement affaire à une boucle sans fin.

Voici trois exemples d'utilisation de boucles `for`:

```

1   int Taille=3, titi=1, toto;
2
3   // qu'une instruction de boucle: accolades non obligatoires
4   for ( int i=0; i < Taille; i++ )
5       cout << i << endl; // Afficher la valeur de i a l'ecran: 012
6
7   //-----
8   for ( int i=0, int j=0; i < Taille; i++, j=2*i, toto=titi )
9   { //exemple plus complique...
10      titi = i + j;
11      cout << i << endl; // Afficher la valeur de i a l'ecran
12      cout << toto << endl; // Afficher la valeur de toto a l'ecran
13  }
14
15  for( ; ; ) // boucle infinie
16      cout << "afficher sans fin" << endl;

```

### 5.8.2 while

La syntaxe d'une boucle `while` est la suivante:

```

1   while ( condition )
2   {
3       // Instructions de boucle repetees tant que la condition est vraie
4   }

```

Il n'y a pas de ; à la fin de la ligne du `while`.

Il est à noter que le test du bloc [condition] se fait en début de boucle. Voici deux exemples d'utilisation d'une boucle `while` en C++:

```

1   int i=0, Taille=3, j;
2   // Resultat identique a la boucle \verb$for$ ci-dessus
3   while ( i < Taille )
4   {
5       cout << i << endl; // Afficher i a l'ecran
6       i++;
7   }
8   //-----
9   // Multi conditions
10  while ( ( i < Taille ) && ( j != 0 ) )
11  {
12      cout << "Ok" << endl;
13      j = 2 * i - i * ( Taille - i );
14      i++;
15  }

```

### 5.8.3 do / while

Contrairement aux boucles `for` et `while`, la boucle `do while` effectue les instructions de boucle **avant** d'évaluer l'expression d'arrêt (le bloc [condition]). La syntaxe d'une boucle `do while` est la suivante:

```

1  do
2  {
3      // Instructions de boucle repetees tant que la condition est vrai
4  }
5  while ( condition ); // le ; apres le while est obligatoire!

```

Il n'y a pas de ; après le **do** mais il y en a un après le **while**.

Voici un exemple d'utilisation d'une boucle **do while** en C++:

```

1  char choix;
2  do
3  {
4      cout << "Quel est votre choix: (q pour quitter) ?";
5      cin >> choix;
6      [...] // autres instructions
7  }
8  while ( choix != 'q' );

```

## 5.9 Pointeurs et Références

### 5.9.1 Pointeurs

En langage C et C++, chaque variable est stockée à une (et unique) adresse physique. Une variable *normale*, c'est à dire ni pointeur ni référence, permet de symboliser une valeur contenue en mémoire (par exemple `double y=3.14;`). Un pointeur est une variable contenant une adresse mémoire, principalement celle d'une autre variable.

Un pointeur est typé: il pointe vers des variables d'un certain type et ainsi permet de manipuler correctement ces variables. Avant d'être utilisé, un pointeur doit obligatoirement être initialisé à l'adresse d'une variable ou d'un espace mémoire. Il est possible d'obtenir l'adresse d'une variable à partir du caractère `&`:

```

1  int a;
2  cout << &a; // affichage de l'adresse de a

```

Les pointeurs ont un grand nombre d'intérêts :

- ils permettent de manipuler des contenus de variables et ces modifications de contenu sont pérennes (pratique quand un fonction doit "retourner" ou plutôt modifier plusieurs variables),
- ils permettent de manipuler de façon efficace en mémoire des données pouvant être importantes: au lieu de passer un élément de grande taille par copie à une fonction, on pourra lui fournir un pointeur vers cet élément. On gagne ainsi en mémoire et en vitesse d'exécution,
- ils permettent de manipuler les tableaux (un tableau est en fait un pointeur sur un espace mémoire réservé),
- en C++ (programmation orientée objet) les pointeurs permettent aussi de réaliser des liens entre objets (cf. cours).

Un pointeur est une variable qui doit être définie, précisant le type de variable pointée, de la manière suivante:

```

1  type * Nom_du_pointeur;

```

Le type de variable pointée peut être aussi bien un type primaire (tel que `int`, `char`...), qu'un type élaboré (tel qu'une `struct` ou une `class`). La taille des pointeurs, quel que soit le type pointé, est toujours la même. Elle est de 4 octets avec un OS<sup>3</sup> 32 bits (et 8 en 64 bits).

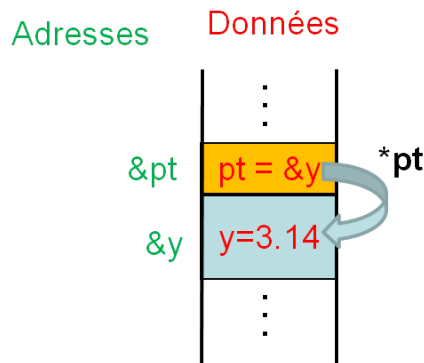
<sup>3</sup>Système d'Exploitation



Un pointeur est typé. Cependant, il est toutefois possible de définir un pointeur sur `void`, c'est-à-dire sur quelque chose qui n'a pas de type prédéfini (`void * toto`). Ce genre de pointeur sert généralement de pointeur de transition, dans une fonction générique, avant un transtypage qui permettra d'accéder aux données pointées. Le polymorphisme (cf. cours) proposé en programmation orientée objet est souvent une alternative au pointeur `void *`.

Pour initialiser un pointeur, il faut utiliser l'opérateur d'affectation = suivi de l'opérateur d'adresse & auquel est accolé un nom de variable (voir la figure 5.9.1):

```
1 double *pt;
2 double y=3.14;
3 pt = &y; // initialisation du pointeur!
```



Après (et seulement après) avoir déclaré et initialisé un pointeur, il est possible d'accéder au contenu de l'adresse mémoire pointée par le pointeur grâce à l'opérateur \*. La syntaxe est la suivante:

```
1 double *pt;
2 double y=3.14;
3 cout << *pt; // Affiche le contenu pointe par pt, c'est a dire 3.14
4 *pt=4; // Affecte au contenu pointe par pt la valeur 4
5 // A la fin des ces instructions, y=4;
```

Attention à ne pas mélanger la signification des symboles \*:

- il sert à **déclarer** un pointeur (`double *pt;`),
- il sert à accéder au contenu pointé par un pointeur (`*pt`), il s'agit de l'in-direction ou de la dé-référence,
- il s'agit de l'opérateur de multiplication (mettre `*pt` au carré ...  $(*pt) * (*pt)$ <sup>4</sup>)

## 5.9.2 Références

Par rapport au C, le C++ introduit un nouveau concept: les références. Une référence permet de faire **référence** à des variables. Le concept de référence a été introduit en C++ pour faciliter le passage de paramètres à une fonction, on parle alors de passage par référence. La déclaration d'une référence se fait simplement en intercalant le caractère &, entre le type de la variable et son nom:

```
1 type & Nom_de_la_variable = valeur;
```

Il existe deux contraintes à l'utilisation des références :

- une référence doit obligatoirement être initialisée lors de sa déclaration,
- une référence ne peut pas être dé-référencée d'une variable à une autre.

Voici un exemple de déclaration d'une référence suivie d'une **affectation**:

<sup>4</sup>KISS est un bon principe: *Keep it Simple, Stupid*

```

1  int A = 2;
2  int &refA = A; //initialisation obligatoire a la declaration!
3  refA++; // maintenant A=3
4
5  // dereference impossible ==> modification de la valeur contenue:
6  int B = 5;
7  refA = B; // signifie A = 5 !!!!
8  refA++;
9  // on a : A = 6, B = 5, et refA = 6

```

Les références permettent d'alléger la syntaxe du langage C++ vis à vis des pointeurs. Voici une comparaison de 2 codes équivalents, utilisant soit des pointeurs soit des références.

<pre> 1 2  /* Code utilisant un pointeur */ 3  int main() 4  { 5      int age = 21; 6      int *ptAge = &amp;age; 7      cout &lt;&lt; *ptAge; 8      *ptAge = 40; 9      cout &lt;&lt; *ptAge; 10 } </pre>	<pre> 1  //Code utilisant une reference 2  int main() 3  { 4      int age = 21; 5      int &amp;refAge = age; /* l' 6      affectation a la 7      declaration */ 8      cout &lt;&lt; refAge; 9      refAge = 40; 10     cout &lt;&lt; refAge; </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Bien que la syntaxe soit allégée, les références ne peuvent pas remplacer les pointeurs. En effet, contrairement aux pointeurs, les références ne peuvent pas faire référence à un tableau d'éléments et ne peuvent pas changer de référence pour une nouvelle variable. Ainsi, en fonction des situations, il sera plus avantageux d'utiliser soit des pointeurs soit des références.

**R** L'usage le plus commun des références est le passage de paramètres aux fonctions.  
Conseil : n'utiliser les références que pour le passage de paramètres aux fonctions ( 5.11.4 )

## 5.10 Tableaux

Un tableau est une variable composée de données de même type, stockées de manière contiguë en mémoire (les unes à la suite des autres, 5.1). Un tableau est donc une suite de cases (espaces mémoires) de même taille. La taille de chacune des cases est conditionnée par le type de donnée que le tableau contient. Les éléments du tableau peuvent être :

- des données de type simple : int, char, float, long, double...;
- des pointeurs, des tableaux, des structures et des classes

En C/C++, les éléments d'un tableau sont indicés à partir de 0. La lecture ou l'écriture du ième élément d'un tableau `tableau` se fait de la façon suivante:

```

1  A = tableau[i]; // lecture du ieme element
2  tableau[i] = A; // ecriture du ieme element

```

Connaitre la taille d'un tableau est primordial pour savoir jusqu'à où il est possible de lire ou écrire en mémoire. Si un tableau contient  $N$  éléments, le dernier élément est accessible par `tableau[N-1]`. Le nombre d'éléments d'un tableau (sa *taille*) est capital car conditionne directement l'utilisation de la mémoire.

Il existe deux familles de tableaux différencier par la manière de gérer l'utilisation (ou l'allocation) mémoire:

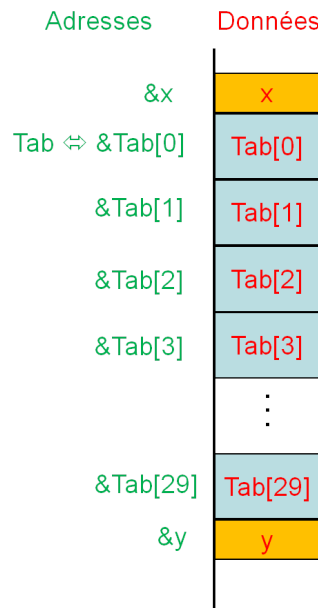


Figure 5.1: Organisation en mémoire d'un tableau

1. les tableaux alloués statiquement où la taille réservée en mémoire est connue (fixée et constante) **à la compilation**,
2. les tableaux alloués dynamiquement qui permettent d'allouer et de libérer l'espace au gré du programme. Ils permettent d'optimiser l'occupation de la mémoire mais demande au programmeur de gérer lui même les réservations et libérations de mémoires.

### 5.10.1 Allocation statique

Lorsque la taille du tableau est connue par avance (à la compilation du programme et non lors de son exécution), il est possible de définir un tableau de la façon suivante:

```

1 Type variable_tableau[nb_elements];
2 Type variable_tableau[] = {element1, element2, element3...};

```

Voici un exemple d'une telle initialisation:

```

1 float notes[10];
2 int premier[] = {4,5,6}; // le nombre d'elements est 3

```

Les tableaux statiques sont alloués et détruits de façon automatique.

La taille de tableaux déclarés ainsi (avec les []) peut être obtenue avec `sizeof( t )` qui retourne le nombre d'octets utilisés par la variable `t`. Ainsi pour connaître le nombre d'éléments d'un tableau:

```

1 float t[] = {4,5,6}; // le nombre d'elements est 3
2 unsigned int t_size = sizeof( t ) / sizeof(float); // t_size vaudra 3

```

**R** Remarque: La norme ISO 1999 du C++ interdit l'usage des *variable-size array* ou *variable-length array* mais de nombreux compilateurs la tolère (via des extensions propres au compilateur). Ce qui n'arrange pas les choses est que de nombreux sites web utilisent ces VLA.

### 5.10.2 Allocation dynamique

Lorsque la taille d'un tableau n'est pas connue lors de la compilation, il est possible en C++ d'allouer un espace mémoire à la taille voulue en cours de programme. On parle d'allocation

dynamique. Pour faire un tableau dynamique, il faut d'abord créer un pointeur, puis allouer l'espace mémoire souhaité par la commande `new`. A la fin de l'utilisation du tableau, il ne faut pas oublier de libérer l'espace alloué par la commande `delete[]`. Voici un récapitulatif des instructions à utiliser en C++ pour la gestion dynamique de tableaux:

- Déclaration d'un pointeur

```
1 type *variable_tableau;
```

- Allocation puis affectation

```
1 variable_tableau = new type[nb_element];
2 //ou en une ligne avec la declaration
3 type *variable_tableau = new type[nb_element];
```

- Libération de l'espace alloué

```
1 delete[] variable_tableau;
```

Voici un exemple de comparaison de déclaration, allocation, destruction d'un tableau dynamique en C et en C++.

<pre>1 /* Allocation dynamique en C */ 2 { 3 int Nb = 4; 4 int *Array = NULL; 5 // Allocation de la memoire 6 Array = malloc(sizeof(int)*Nb); 7 // liberation de la memoire 8 free(Array); 9 }</pre>	<pre>1 /* Allocation dynamique en C++ */ 2 { 3 int Nb = 4; 4 int *Array = NULL; 5 // Allocation de la memoire 6 Array = new int[Nb]; 7 // liberation de la memoire 8 delete[] Array; 9 }</pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

#### Cas particulier: tableau dynamique avec un seul élément:

Pour allouer dynamiquement l'espace pour un seul élément, l'instruction:

```
1 Type *variable_unique = new type[1];
```

se simplifie:

```
1 Type *variable_unique = new type;
```

de même pour la suppression:

```
1 delete variable_unique;
```

L'intérêt de l'écriture `Type *variable_unique = new type;` est de pouvoir initialiser la variable avec des valeurs particulières<sup>5</sup>, par exemple:

```
1 float *tab = new float(1.3); // un seul objet de type float
2 // est alloue en memoire
3 // et sa valeur est initialisee a 1.3
```

L'allocation dynamique permet au programmeur de gérer la mémoire. Ceci est utile lorsque le programme est gourmand en ressource mémoire (empreinte mémoire importante) ou lors de partage de ressources entre processus par exemple.

<sup>5</sup>`newtype` exécute le constructeur par défaut, `newtype(arg1,arg2,...)` exécute un constructeur utilisateur, cf section 8.3.1

```
double t[] = {12, 32, 4, 3.14, 7.02, 32, 7, 5.5};
t → 

|    |    |   |      |      |    |   |     |
|----|----|---|------|------|----|---|-----|
| 12 | 32 | 4 | 3.14 | 7.02 | 32 | 7 | 5.5 |
|----|----|---|------|------|----|---|-----|



char tab[] = {'T', 'o', 't', 'o'};
tab → 

|     |     |     |     |
|-----|-----|-----|-----|
| 'T' | 'o' | 't' | 'o' |
|-----|-----|-----|-----|



char chaine[] = "Toto" ;
chaine → 

|     |     |     |     |   |
|-----|-----|-----|-----|---|
| 'T' | 'o' | 't' | 'o' | 0 |
|-----|-----|-----|-----|---|


```

### 5.10.3 Tableaux et pointeurs

La syntaxe suivante:

```
1 int tab[] = {4,5,6};
2 cout << tab;
```

affiche une adresse. Ceci montre qu'il existe un lien étroit entre les tableaux et les pointeurs. Plus précisément une variable tableau (dans l'exemple précédent `tab`) renvoie comme valeur l'adresse de la première case du tableau. Ainsi, le code:

```
1 int tab[] = {4,5,6};
2 cout << *tab; // affiche le contenu pointe par tab, c'est a dire
3 // la valeur de la premiere case du tableau
```

renvoie la valeur de la première case du tableau, ici 4. Dans l'exemple suivant, nous proposons différentes opérations sur les éléments de tableaux:

```
1 int tab[] = {4,5,6};
2 cout << &(tab[1]); // affiche l'adresse de la deuxieme case du tableau
3 cout << tab+1; // affiche l'adresse de la deuxieme case du tableau
4 cout << tab[2]; // affiche 6
5 cout << *(tab+2); // affiche 6
```

Voici (figure 5.10.3) quelques exemples de tableaux en mémoire.

- Un tableau de double initialisé à la création
- Un tableau de char initialisé à la création (figure 5.10.3)
- Une chaîne de caractère (figure 5.10.3)

Dans le cas des chaînes de caractères (uniquement pour des tableaux de type `char`), le dernier élément est un **zéro terminal**. Il ne s'agit pas d'un caractère affichable. Il permet de détecter la fin du tableau de caractère. Ainsi, les chaînes de caractères sont les seuls tableaux qui ne nécessitent pas de connaître le nombre d'éléments contenus dans le tableau (il suffit de trouver le zéro terminal).

### 5.10.4 Tableaux multidimensionnels

Les tableaux multidimensionnels sont des tableaux de tableaux. Par exemple, en dimension 3, un tableau statique sera déclaré de la façon suivante (usage déconseillé):

```
1 type volume[L][C][P]; // L,C,P, nb de lignes, colonnes et profondeur
```

En dimension 2, un tableau dynamique sera déclaré de la façon suivante:

```
1 type **matrice;
2 matrice = new type*[L];
3 for( int i=0; i<L; i++ )
4 matrice[i] = new type[C];
```

et sera détruit de la façon suivante:

```

1 // A partir de l'exemple precedent
2 for( int i=0; i<L; i++ )
3     delete[] matrice[i];
4     delete[] matrice;

```

## 5.11 Fonctions

Une fonction est un sous-programme qui permet d'effectuer un ensemble d'instructions par simple appel à cette fonction. Les fonctions permettent d'exécuter dans plusieurs parties du programme une série d'instructions, ce qui permet de simplifier le code et d'obtenir une taille de code minimale. Il est possible de passer des variables aux fonctions. Une fonction peut aussi retourner une valeur (au contraire des procédures, terminologie oblige...). Une fonction peut faire appel à une autre fonction, qui fait elle même appel à une autre, etc. Une fonction peut aussi faire appel à elle-même, on parle alors de fonction récursive (il ne faut pas oublier de mettre une condition de sortie au risque de ne pas pouvoir arrêter le programme).

### 5.11.1 Prototypé d'une fonction

Comme les variables, avant d'être utilisée, une fonction doit être déclarée. Le prototype (ou déclaration) d'une fonction correspond à cette déclaration. Il s'agit de la description de la fonction, c'est à dire donner son nom, indiquer le type de la valeur renvoyée et les types d'arguments. Cette fonction sera définie (les instructions qu'elle exécute) plus loin dans le programme. On place le prototype des fonctions en début de programme (généralement le plus tôt possible). Cette déclaration permet au compilateur de *vérifier* la validité de la fonction à chaque fois qu'il la rencontre dans le programme. Contrairement à la définition de la fonction, le prototype n'est pas suivi du corps de la fonction (contenant les instructions à exécuter), et ne comprend pas obligatoirement le nom des paramètres (seulement leur type). Une fonction est déclarée de la façon suivante:

```

1 type_retour NomFonction( type Arg1, type Arg2,... );

```

Le prototype est une déclaration. Il est suivi d'un point-virgule. Les entêtes (fichiers .h) contiennent les déclarations des fonctions. Par exemple `#include <math.h>` permet de déclarer toutes les fonctions de la librairie math.

### 5.11.2 Définition d'une fonction

La définition d'une fonction est l'écriture du code exécuté par cette fonction. Une fonction est définie par l'ensemble *nom de fonction, nombre d'arguments en entrée, type des arguments en entrée et liste d'instructions*. La définition d'une fonction se fait selon la syntaxe suivante:

```

1 type_retour NomFonction( type Arg1, type Arg2,... )
2 {
3     // liste d'instructions
4 }

```

Remarques:

- `type_retour` représente le type de valeur que la fonction peut retourner (char, int, float...), il est obligatoire même si la fonction ne renvoie rien<sup>6</sup>;
- si la fonction ne renvoie aucune valeur, alors `type_retour` est `void`;
- s'il n'y a pas d'arguments, les parenthèses doivent rester présentes.

De plus, le nom de la fonction suit les mêmes règles que les noms de variables:

<sup>6</sup>exception faite des constructeurs et destructeurs

- le nom doit commencer par une lettre,
- un nom de fonction peut comporter des lettres, des chiffres et le caractère `_` (les espaces ne sont pas autorisés),
- le nom de la fonction, comme celui des variables est sensible à la casse (différenciation entre les minuscules et majuscules).

### 5.11.3 Appel de fonction

Pour exécuter une fonction, il suffit de lui faire appel en écrivant son nom (en respectant la casse) suivie de parenthèses (éventuellement avec des arguments):

```
1  NomFonction(); // ou
2  NomFonction1( Arg1, Arg2,... );
```

### 5.11.4 Passage d'arguments à une fonction

Il existe trois types de passage de paramètres à une fonction: le passage par valeur, par pointeur et par référence. Ces passages sont décrits ci-dessous.

#### Passage par valeur

Les arguments passés à la fonction sont copiés dans de nouvelles variables propres à la fonction. Les modifications des valeurs des arguments ne sont donc pas propagés au niveau de la portée de la fonction appelante. Ce type de passage de paramètre par valeur est aussi appelé passage par copie.

Voici la syntaxe du passage par valeur:

- Déclaration:

```
1  type_retour NomFonction( type Arg1, type Arg2,... );
```

- Définition:

```
1  type_retour NomFonction( type Arg1, type Arg2,... )
2  {
3      // instructions de la fonction
4      // pouvant utiliser les valeurs des paramètres
5      // et déclarer d'autres variables
6      return variable_de_type_retour;
7  }
```

Voici un exemple de telles fonctions:

```
1
2  void Affichage(int a)
3  {
4      cout<<"Valeur = "<<a<<endl;
5  }
6
7  int Somme(int a, int b)
8  {
9      int c = a + b;
10     return c;
11 }
12
1 // suite du programme
2
3 int main()
4 {
5     int nb1=2, nb2=3, nb3=4;
6     int somme;
7     Affichage( nb1 );
8     Affichage( nb2 );
9     Affichage( Somme(nb1,nb2) );
10    somme = Somme( nb2, nb3);
11    return 0;
12 }
```

#### Passage par pointeur

Dans le cas où l'on souhaite modifier dans la fonction la valeur d'une variable passée en paramètre, il est nécessaire d'utiliser en C++ soit un passage par pointeur soit un passage par référence. Dans le cas du passage par pointeur, les variables passées en arguments de la fonction correspondent à des adresses de variables (pointeur ou `&variable`). Voici la syntaxe du passage par pointeur:

- Déclaration de fonction:

```
1 type_retour NomFonction( type *Arg1, type *Arg2,... );
```

Définition de fonction:

```
1 type_retour NomFonction( type *Arg1, type *Arg2,... )
2 {
3     // instructions de la fonction
4     // les variables passees par pointeur s'utilisent
5     // comme les autres variables
6     return variable_de_type_retour;
7 }
```

Voici un exemple d'utilisation:

```
1
2 void CopieTab(double *origine,
3             double *copie,
4             int taille)
5 {
6     for (int i=0; i<taille, i++)
7         copie[i] = origine[i];
8 }
9
10 void ModifTailleTab(double **tab,
11                   int taille)
12 {
13     // suppression du tableau
14     if ( *(tab) != NULL )
15         delete[] *(tab);
16     *(tab) = new double[taille];
17     // il faut definir **tab pour
18     // que l'allocation dynamique
19     // soit prise en compte au
20     // niveau de la fonction
21     // appelante
22 }
```

```
1 // suite du programme
2 int main()
3 {
4     int dim=20;
5     double tab1[20];
6     double tab2[20];
7     double *p = NULL;
8     // Initialisation
9     // de tab2
10    CopieTab(tab1,tab2,dim);
11    // Allocation
12    // dynamique de p
13    ModifTailleTab(&p,dim);
14    delete[] p;
15    return 0;
16 }
17 }
```

### Passage par référence

Un paramètre passé par référence n'est pas copié en local pour la fonction: l'adresse mémoire du paramètre (sa référence) est passée à la fonction. Pour l'utilisateur de la fonction ainsi que pour le programmeur, ce processus est complètement transparent et est syntaxiquement identique au passage par valeur (à un & près...). Le passage par référence est utile pour:

- modifier et préserver les changements d'un paramètre passé en argument;
- gagner du temps en ne copiant pas le paramètre pour la fonction;
- ne pas exécuter le constructeur de copie de l'objet passé.

Voici la syntaxe du passage par référence:

- Déclaration:

```
1 type_retour NomFonction( type &Arg1, type &Arg2,... );
```

- Définition:

```
1 type_retour NomFonction( type &Arg1, type &Arg2,... )
2 {
3     // instructions de la fonction
4     // les variables passees par reference s'utilisent
```



```

5     // comme les autres variables
6     return variable_de_type_retour;
7 }

```

Voici un exemple de comparaison d'utilisation de fonctions avec passage par pointeur ou avec passage par référence.

<pre> 1  /* Passage par pointeur */ 2  struct Point 3  { 4      int x,y; 5  }; 6 7  void remiseAzero(Point *ptNew) 8  { 9      ptNew-&gt;x = 0; 10     ptNew-&gt;y = 0; 11 } 12 13 int main() 14 { 15     Point pt; 16     remiseAzero(&amp;pt); 17     return 0; 18 } </pre>	<pre> 1  /* Passage par reference */ 2  struct Point 3  { 4      int x,y; 5  }; 6 7  void remiseAzero(Point &amp;ptNew) 8  { 9      ptNew.x = 0; 10     ptNew.y = 0; 11 } 12 13 int main() 14 { 15     Point pt; 16     remiseAzero(pt); 17     return 0; 18 } </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 5.11.5 Cas des paramètres avec allocation dynamique dans une fonction

Il peut être commode de créer des fonctions qui vont allouer (et respectivement dé-allouer) des espaces mémoires. Pour les fonctions d'allocation, il y a deux approches :

- soit le nouvel espace est retourné par le **paramètre de retour de la fonction**, cas simple.
- soit le nouvel espace est un **argument** qui doit être modifié, cas plus complexe. Ce cas peut se présenter quand on doit modifier un espace déjà existant ou quand on souhaite retourner une autre valeur par retour de fonction (par exemple un booléen qui précise si l'allocation a pu se faire correctement).

Dans les deux cas, il faudra passer par des pointeurs. Pour les fonctions de libération de l'espace mémoire, seul le cas par argument est possible.

#### Retour de fonction d'un espace mémoire dynamique

Cette approche est relativement simple et ne comporte pas de piège particulier: on retourne un pointeur alloué dynamiquement. L'exemple suivant est donné pour une allocation d'un simple tableau de `double`, mais il pourra s'agir d'allocations dynamiques plus complexes comme des matrices ou des objets plus spécifiques.

```

1  double * AllouerVecteur(int n)
2  {
3      double *r = new double[n];
4      return r;
5  }

```

Son utilisation dans un main sera :

```

1  int main()
2  {
3      double *t = 0;
4      t = AllouerVecteur(100);
5      // ... utilisation de t

```

```

6   delete[] t;
7   return 0;
8   }

```

Parfois il n'est pas possible de retourner un pointeur sur l'espace mémoire. Dans ce cas il faut considérer l'approche présentée dans le paragraphe suivant.

### Allocation dynamique dans une fonction par un argument

Voici la même fonction mais avec la variable à allouer passée en argument. Il y a deux difficultés pour cette fonction. D'une part il s'agit de modifier l'adresse stockée par le pointeur et non le contenu pointé. Le type de cette variable est un pointeur de pointeur soit `type **` s'il s'agit d'allouer un espace pour un tableau mono-dimensionnel (type `***` pour une matrice).

D'autre part, les règles de priorité des opérateurs `*` et `[]` imposent une rigueur d'écriture et une bonne connaissance du langage notamment si on souhaite modifier les valeurs du tableau.

```

1 void AllouerVecteur(double **t, int n)
2 {
3   *t = new double[n];
4   for(int i=0; i<n; i++) // exemple de mise a 0 des valeurs
5     (*t)[i] = 0; // noter les parentheses obligatoires!
6 }

```

L'initialisation des valeurs n'est pas forcément utile mais permet d'illustrer l'impact de la priorité des opérateurs `*` et `[]` (voir paragraphe 5.5.2). Cependant, dans le cas des matrices, il faudra nécessairement utiliser cette syntaxe :

```

1 void AllouerMatrice(int l, int c, double ***res)
2 {
3   *res = new double*[l]; // allocation des lignes
4   for(int i=0; i<l;i++)
5     (*res)[i] = new double[c]; // allocation des colonnes
6 }

```

L'utilisation de la fonction *AllouerVecteur* dans un main sera :

```

1 int main()
2 {
3   double *t = 0;
4   AllouerVecteur(&t, 100);
5   // ... utilisation de t
6   delete[] t;
7   return 0;
8 }

```

Dans l'exemple précédent, on suppose que l'argument ne pointe vers aucun espace mémoire préalablement alloué. Dans un cadre plus général, il faudrait d'abord vérifier si cela est vrai et au besoin libérer l'espace mémoire utilisé avant d'en allouer un autre. Sans cela, le premier espace mémoire sera perdu si aucune autre variable pointe vers cet espace. La fonction du paragraphe suivant pourra être appelée au début de la fonction *AllouerVecteur*.

### Fonction de libération d'espace mémoire

La libération d'une variable allouée par une fonction nécessite le passage de la variable par un argument. Cette fonction commence par vérifier que l'adresse pointée ne soit pas zéro qui correspond à une zone mémoire qui ne peut pas être allouée.

```

1 void LiberationVecteur(double **t)
2 {
3   if( *t != 0 )

```

```
4  delete[] *t;
5  *t = 0;
6  }
```

Et son utilisation:

```
1  int main()
2  {
3  double *t = 0;
4  t = AllouerVecteur(100);
5  // ... utilisation de t
6  LiberationVecteur(&t);
7  return 0;
8  }
```

### 5.11.6 Surcharge de fonction

Il est possible de nommer plusieurs fonctions avec le même nom, à condition que celles-ci aient leurs arguments différents (en type et/ou en nombre). Ce principe est appelé surcharge de fonction. Plusieurs fonctions portant le même nom et ayant des paramètres différents peuvent co-exister. Chaque fonction réalisera des opérations différentes dépendant des arguments qui lui sont passés.

```
1  void Affiche(int a)
2  { cout << a << endl;}
3
4  void Affiche(double *t, int size)
5  {
6  for(int i=0; i<size; i++)
7  cout << t[i]<< " ";
8  cout << endl;
9  }
10
11 int main()
12 {
13 double pi=3.14;
14 double v[]={1,2,3,4,5};
15
16 Affiche(pi); // Affiche avec un double
17 Affiche(v, sizeof(v)/sizeof(double) );
18
19 return 0;
20 }
```

**R** Pour savoir si une surcharge est légitime ou non, il faut se mettre à la place du compilateur et vérifier que l'appel de fonction n'est pas ambiguë c'est à dire qu'il n'y a aucun doute sur quelle fonction utiliser. A noter que le type de retour d'une fonction n'est pas discriminant.



## 6. Flux et interactions utilisateurs

### 6.1 Flux d'entrée cin et de sortie cout

En C++, la manière la plus efficace pour gérer les entrées/sorties est l'utilisation de flux de données (stream). Les flux d'entrée/sortie ne sont pas inclus dans le langage natif du C++. Ils se trouvent dans la *standard stream input/output library* aussi appelé *iostream*. Pour utiliser les flux dans un programme, il faut inclure le fichier d'entête de cette librairie (cette librairie est liée par défaut au compilateur):

```
1  #include <iostream> // utilisation de cin et cout
2  #include <iomanip> // utilisation des manipulateurs
3                      // (hex, bin, setw, ...)
4
5  using namespace std; // pour ne pas ecrire "std:."
6                      // devant chaque fonction/classe de la std
```

### 6.2 Flux de sortie pour l'affichage: cout

`cout` désigne le flux de sortie standard qui est dirigé par défaut sur l'écran. Il va permettre d'afficher ce que l'on veut à l'écran. La syntaxe d'utilisation est la suivante:

```
1  [std::]cout << donnee_a_afficher [ << autres_donnees << encore_des_donnees
   ];
```

Voici des exemples d'utilisation de flux de sortie:

```
1  cout << "Hello world";
2  cout << "Hello " << "world"; //idem que precedent
3  cout << "Hello world" << endl; //idem que precedent + saut de ligne
4  int longueur = 12;
5  cout << "La longueur est de : " << longueur << " metres \n";
6  double longueur = 12.323;
7  cout << "La longueur est de : "
8      << longueur
```

```

9         << " metres"
10        << endl;
11        // Affichage en bases differentes
12        cout << dec << 16 << endl; // base decimale
13        cout << hex << 16 << endl; // base hexadecimal
14        cout << oct << 16 << endl; // base octale
15        cout << 24; // la derniere base utilisee reste active !!!

```

Les 4 dernières lignes provoquent l’affichage suivant:

```

1    16
2    10
3    20
4    30

```

Pour modifier la précision à l’affichage des valeurs, il faut utiliser la fonction `setprecision`.

```

1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  int main () {
6      double pi = 3.141592654;
7      cout << setprecision (3) << pi << endl;
8      cout << setprecision (9) << pi << endl;
9      cout << fixed;
10     cout << setprecision (3) << pi << endl;
11     cout << setprecision (9) << pi << endl;
12     return 0;
13 }

```

Le code précédent donne l’affichage suivant:

```

1    3.14
2    3.14159265
3    3.142
4    3.141592654

```

Pour ajouter des caractères avant l’affichage d’une valeur, utiliser la fonction `setw`.

```

1    cout << ":" << setw (10) << 77 << endl;

```

Ce qui donne l’affichage suivant:

```

1    :          77

```

### 6.3 Flux d’entrée clavier: `cin`

`cin` est le flux d’entrée standard sur lequel est dirigé par défaut le clavier. Il va permettre de récupérer des valeurs entrées par l’utilisation au clavier. La syntaxe est la suivante:

```

1    [std::]cin >> variable;
2    cin >> variable1 >> variable2; // pas recommande

```

Voici des exemples d’utilisation de flux d’entrée:

```

1    #include <iostream>
2    using namespace std;
3    int main()
4    {
5        int age;

```

```

6     cout << "Entrer votre age ";
7     cin >> age;
8     cout << "Vous avez " << age << " ans" << endl;
9     return 0;
10  }

```

Pour insérer une pause (attente que l'on appuie sur la touche *Entree*), il est conseillé de d'abord purger la mémoire tampon lue par `cin` qui contient souvent le dernier caractère de la saisie précédente (le deuxième symbole de la touche *Entree*).

```

1   cin.sync(); // tous les caracteres de la memoire tampon non lus sont
           ignores
2   cin.get(); // attente de la touche Entree

```

## 6.4 Cas des chaînes de caractères

Les chaînes de caractères sont un groupe de lettres et permettent de représenter des mots, phrases, etc. Une chaîne de caractère est un tableau de `char` dont la dernière valeur est égale à la valeur 0 (zéro). En C++ il existe deux manières de gérer les chaînes de caractères

- un tableau de caractère : `char ch[50]` ;
- un objet : `string st` ;

Voici des exemples d'utilisation de chaînes de caractères.

```

1  #include <iostream>
2  using namespace std;
3
4  int main(int argc, char *argv[])
5  {
6      int i;
7
8      cout << " Chaîne de Caracteres : ";
9      char ch[50];
10     cin >> ch ;
11     cout << " La chaîne est :" << ch << endl;
12     for( i=0; ch[i]!=0 ; i++) // jusqu'a la valeur 0
13         cout << "," <<ch[i] <<"," ";
14     cout << endl;
15
16     cout << " String : ";
17     string st;
18     cin >> st;
19     cout << " Le string est :" << st << endl;
20     for( i=0; i<st.size() ; i++) // permet de connaitre le nombre de
           caracteres
21         cout << "," <<st[i] <<"," ";
22     cout << endl;
23
24 }

```

Si on souhaite demander à l'utilisateur d'entrer une chaîne comportant des espaces, l'utilisation du code suivant est nécessaire (même recommandé dans de nombreux cas pour éviter les failles de type *buffer-overflow*).

```

1  char ch[50];
2  cin.get(ch,50);

```

Avec un objet `string` ceci est possible grâce au code suivant (utilisation de la fonction globale `getline`: ajouter `#include <string>`):

```
1 string str;  
2 getline( cin, str);
```



## 7. Flux et Fichiers

### 7.1 Fichiers

En C++, les flux génériques permettent d'accéder et de manipuler de nombreux contenus. Les fichiers en font partis. Il y a cependant des limites à la généralité des flux dues aux différents types d'accès (lecture seule, écriture seule, ...) et au mode des fichiers (**binaire** ou **texte**). L'implémentation en C d'accès en lecture/écriture est tout aussi dépendante de ces différences. Celle-ci est donnée dans le paragraphe 7.4.

### 7.2 Mode texte

Le mode texte correspond à l'enregistrement de données dans leur forme ASCII. Par exemple, la valeur `double pi = 3.141592` sera enregistrée sous forme d'une chaîne de caractères: `'3' '.' '1' '4' '1' '5' '9' '2' '0'`. Les fichiers en mode texte peuvent être visualisés par n'importe quel logiciel d'édition de texte (*vim*, *notepad*, ...). Leur taille dépend de la précision affichée et est souvent largement supérieure à la quantité de mémoire nécessaire au stockage des valeurs.

Pour disposer des flux sur les fichiers, il faut inclure `fstream`. Ceci permet d'accéder en lecture ou en écriture ou en lecture/écriture à un fichier.

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
```

Ensuite, dans le `main`, il faut créer une variable de flux sur le fichier. Son type est `fstream`.

```
1 int main(int argc, char *argv[])
2 {
3     fstream fileT;
```

Puis ouvrir le fichier avec la fonction `open`. `open` est une fonction qui ne peut être exécutée qu'à partir d'un objet (une variable) de type `fstream`.

```
1     fileT.open ("test.txt", fstream::in | fstream::out | fstream::trunc);
2
```

```

3     if( fileT.fail() )
4     {
5         cout << "Text File cannot be opened/created !" << endl;
6         return -1; // fin avec un retour d'erreur
7     }

```

Écriture de quelques informations:

```

1     fileT << "Hello_SansEspace" << " " << 3.141592654 << " " << 2.718281 <<
        endl;
2     fileT << 6.022141 << " " << 1.380650 << endl;

```

Puis la lecture de quelques informations. D'abord un retour au début du fichier:

```

1     //read Something
2     fileT.clear(); // mise a 0 des flags
3     fileT.seekg(0, fstream::beg); // retour au debut du fichier

```

Et enfin la lecture des informations. A noter le respect des types et de l'ordre des informations!

```

1     string stT;
2     double piT, eT, aT, bT;
3
4     fileT >> stT >> piT >> eT >> aT >> bT; // lecture
5
6     cout << stT << endl;
7     cout << "PI : " << piT << " E : " << eT << endl;
8     cout << "Avogadro : " << aT << " Boltzmann : " << bT << endl;
9     fileT.close(); // close file
10    return 0;
11 }

```

### 7.3 Mode binaire

Le mode binaire correspond à l'enregistrement brute des données en format binaire (forme du stockage en mémoire). Par exemple, la valeur `double` `pi` = 3.141592 sera enregistrée sous sa forme `double` (virgule flottante format IEEE 754). Le contenu de ces fichiers est illisible par les logiciels d'édition de texte. L'accès aux fichiers binaires est assez similaire à l'accès aux fichiers texte. Voici les principales différences.

Pour lire ou écrire en mode binaire le drapeau (*flag* en anglais) `fstream::binary` doit être ajouté lors de l'ouverture (fonction `open`).

L'écriture dans un fichier ouvert en écriture (flag `fstream::in`) se fait par la fonction `write`. Cette fonction enregistre le contenu d'une variable quelconque **octet par octet**. Il s'agira donc de convertir le contenu de la variable en un tableau de caractère (`char *`) grâce à la fonction `reinterpret_cast<char *>(&variable)` et de spécifier à `write` le nombre d'octets à écrire grâce à la fonction `sizeof(type_de_variable)`.

La lecture de donnée d'un fichier en mode binaire (et ouvert en lecture : `fstream::out`) se fait via la fonction `read`. Elle fonctionne de la même manière que la fonction `write`. Il n'y a pas de symbole de séparation entre différente variable puisque à chaque lecture, un nombre précis d'octet est lu.

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main(int argc, char *argv[])

```

```

6  {
7      fstream fileB;
8      fileB.open ("test.bin", fstream::binary | fstream::in | fstream::out |
9          fstream::trunc );
10
11     if( fileB.fail() )
12     {
13         cout << "Binary File cannot be opened/created !" << endl;
14         return -1; // end
15     }
16
17     // write something
18     double pi = 3.1415192;
19     double e = 2.718281;
20     int c = 299999;
21     fileB.write(reinterpret_cast<char *>(&pi), sizeof(double) );
22     fileB.write(reinterpret_cast<char *>(&e), sizeof(double) );
23     fileB.write(reinterpret_cast<char *>(&c), sizeof(int) );
24
25     //read Something
26     fileB.clear();
27     fileB.seekg(0, fstream::beg);
28     double piB, eB;
29     int cB;
30
31     fileB.read( reinterpret_cast<char *>(&piB), sizeof(double) );
32     fileB.read( reinterpret_cast<char *>(&eB) , sizeof(double) );
33     fileB.read( reinterpret_cast<char *>(&cB) , sizeof(int) );
34
35     cout << " PiB : " << piB << " EB : " << eB << " cB : " << cB << endl;
36
37     fileB.close(); // close file
38
39     return 0;
40 }

```

L'utilisation de structure ou de classe pour l'enregistrement de données organisées est fortement conseillée pour le mode binaire.

## 7.4 Et en C?

Il pourrait arriver que vous ayez besoin de manipuler fichiers sans disposer du C++ mais uniquement du C... Or, en C, les flux ne sont pas disponibles. Pour accéder aux fichiers, il faut utiliser le type `FILE *` et les fonctions:

- `fopen` pour ouvrir un fichier (mode binaire ou texte, lecture/écriture)
- `fclose` pour fermer un fichier ouvert
- `fseek` pour se déplacer dans un fichier
- `fread` pour lire une quantité de donnée dans le fichier (unité : l'octet ou le `char`)
- `fwrite` pour écrire dans un fichier (unité : l'octet ou le `char`)

Voici une fonction en C qui permet la lecture générique de tous les fichiers (mode binaire). Il faudra changer le type (`cast`) de `message` pour obtenir le type de données voulu<sup>1</sup>. Pour les formats complexes de données, il est conseillé d'utiliser des structures. Le message est un tableau de donnée. Ce tableau est alloué dynamiquement (en C) par la fonction `LireFichier`.

<sup>1</sup>le type de message de la fonction pourrait être `void *`

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int LireFichier( const char fichier[], unsigned char **message, int *taille,
4                 int offset)
5 {
6     int nb_lu;
7     FILE* fich = fopen(fichier,"rb");
8     if(!fich) return 1;
9
10    // Lecture de la taille du fichier
11    fseek( fich, 0, SEEK_END); // Positionnement du pointeur a la fin du
12    // fichier
13    *taille = ftell(fich) - offset; // Lecture de la position du pointeur (=
14    // taille)
15    fseek( fich, offset, SEEK_SET); // rePositionnement du pointeur au debut
16    // du fichier (offset)
17
18    *message = (char*) malloc (*taille);
19    if( *message == NULL ) return 3; // PB allocation
20
21    nb_lu = fread( *message, sizeof(char), *taille, fich); // Lecture
22    if ( nb_lu != *taille ) return 1;
23
24    fclose(fich);
25    return 0; // pas de probleme
26 }

```

La fonction générique pour écrire dans un fichier en mode binaire est la suivante. Si le message est la version chaîne de caractère de données (fprintf), on créera un fichier lisible en mode texte.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int EcrireFichier( const char fichier[], const unsigned char message[], int
5                   taille)
6 {
7     FILE *f_dest;
8     f_dest = fopen(fichier,"wb");
9     if(!f_dest) return 1; //probleme d'accès en écriture
10    fwrite( message, sizeof(char), taille, f_dest);
11
12    fclose(f_dest);
13    return 0; //pas de probleme
14 }

```



# Partie C: Orienté Objet

## **8** Programmation Orientée Objet en C++ 73

- 8.1 UML
- 8.2 Concept de classe pour l'encapsulation
- 8.3 Méthodes particulières
- 8.4 Héritage
- 8.5 Polymorphisme
- 8.6 Généricité



## 8. Programmation Orientée Objet en C++

Le langage C est un langage procédural, c'est-à-dire un langage permettant de définir des données grâce à des variables, et des traitements grâce aux fonctions. L'apport principal du langage C++ par rapport au langage C est l'intégration des concepts "objet", afin d'en faire un langage orienté objet. Les approches orientées objets (programmation, conception, ...) sont nombreuses et s'appliquent à de nombreux domaines autre que l'informatique (par exemple en gestion de stock, l'électronique, l'automatisme, ...). De nombreux outils sont disponibles pour représenter et concevoir en orienté objet. L'UML est particulièrement bien adapté à cette tâche, notamment grâce aux diagrammes de classes et d'objets pour décrire les constituants du système puis grâce aux diagrammes d'interactions et de machines d'états pour décrire les comportements d'un système.

Ce qui caractérise les méthodes "orientées objet" est l'utilisation d'un ou plusieurs paradigmes objets dont les plus importants sont:

- Encapsulation: regrouper données et opérations
- Héritage : généralisation de classes
- Polymorphisme: découle de l'héritage, permet aux classes les plus générales d'utiliser les spécifications des classes plus spécifiques (nécessite la surcharge de fonction membre)
- Généricité (extension de l'encapsulation) : Modèle d'encapsulation, quelque soit le type des données.

Après avoir présenté rapidement l'UML, ces 4 paradigmes sont développés.

### 8.1 UML

L'UML (*Unified Modeling Language*, que l'on peut traduire par *langage de modélisation unifié*) est une notation permettant de modéliser une application ou un système sous forme de concepts abstraits (les **classes**) et d'interactions entre les instances de ces concepts (les objets). Cette modélisation consiste à créer une représentation des éléments du monde réel auxquels on s'intéresse, sans se préoccuper de leur réalisation (ou implémentation en informatique). Ainsi, pour l'informatique, cette modélisation est indépendante du langage de programmation.

L'UML est depuis 1997 un standard industriel (Object Management Group) initié par Grady Booch, James Rumbaugh et Ivar Jacobson. L'UML est un langage formel, ce qui se traduit par une



concision des descriptions mais exige une bonne précision (rigueur) pour la représentation des des systèmes. Il se veut aussi intuitif et graphique: la totalité des représentations d'un système se fait par des diagrammes.

L'UML 2 dispose de 13 diagrammes différents permettant de modéliser l'ensemble d'un système (encore une fois, pas seulement informatique!) et de décrire totalement un système et son déploiement, à tel point que l'UML 2.0 est compilable et permet de générer des exécutables pour les applications informatiques. Ces diagrammes sont classiquement regroupés par vues (il y a 5 vues) pour faciliter la modélisation des systèmes dont la représentation est donnée sur la figure 8.3.

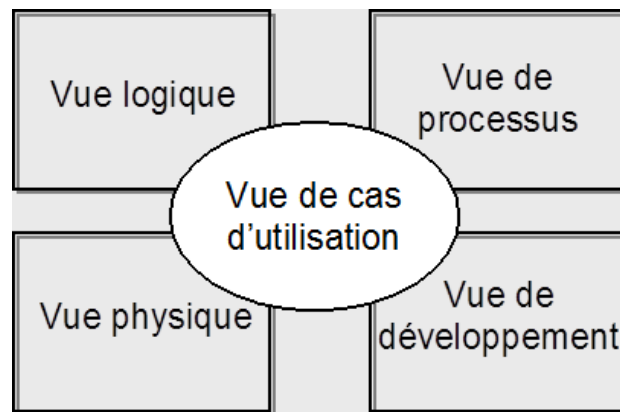


Figure 8.1: Modèle de vue 4+1 de P. Kruchten.

Par la suite, nous nous focaliserons sur le diagramme de classe - contenu dans la *Vue Logique* - qui permet de décrire le contenu et l'interface de chaque classe et représenter les liens entre les classes d'un système.

Modéliser un système n'est pas évident d'autant plus quand il s'agit d'un système à concevoir. La première difficulté est de déterminer les objets (et les concepts) présents (une sorte de regroupement pertinent de données et fonctions du système) puis les interactions qui existent entre les différents concepts. Ensuite il faut passer à la description précise de chaque concept: décrire les données et les fonctions qu'il utilise pour fonctionner correctement jusqu'au déploiement de l'application.

### 8.1.1 Diagramme de classe

Le diagramme de classe est un des 13 diagrammes de l'UML. Il s'agit d'un diagramme de la *vue logique* qui décrit de manières abstraites les parties d'un système et qui sert à modéliser les composant du système et leurs interactions.

On peut ainsi modéliser un programme en plusieurs classes qui peuvent avoir des relations entre elles. Le principal intérêt de ce diagramme est le fait d'associer ou de faire interagir les différentes classes pour obtenir un programme qui peut être complexe mais facilement réalisable à l'aide de ces dites classes qui auront un code plus simple.

En UML, pour le diagramme de classe, une classe est définie par son nom, ses attributs (ou champs, ou données membres) et ses opérations (ou méthodes, ou fonctions membres). Il n'y a pas besoin d'indiquer le code réaliser par chaque méthodes: les noms des champs et des méthodes, ainsi que les types et paramètres doivent être suffisant pour comprendre ce qui sera réalisé par les fonctions et géré par la classe. La figure 8.2 présente un digramme de classe minimaliste permettant de représenter des nombre complexes et d'effectuer des additions entre des nombres complexes.

La première partie du diagramme correspond simplement au nom de la classe, sans espaces ni ponctuations et qui doit être le plus explicite possible. La seconde partie est dédiée aux attributs, on y retrouve le nom de l'attribut et son type. La dernière partie contient le nom des fonctions



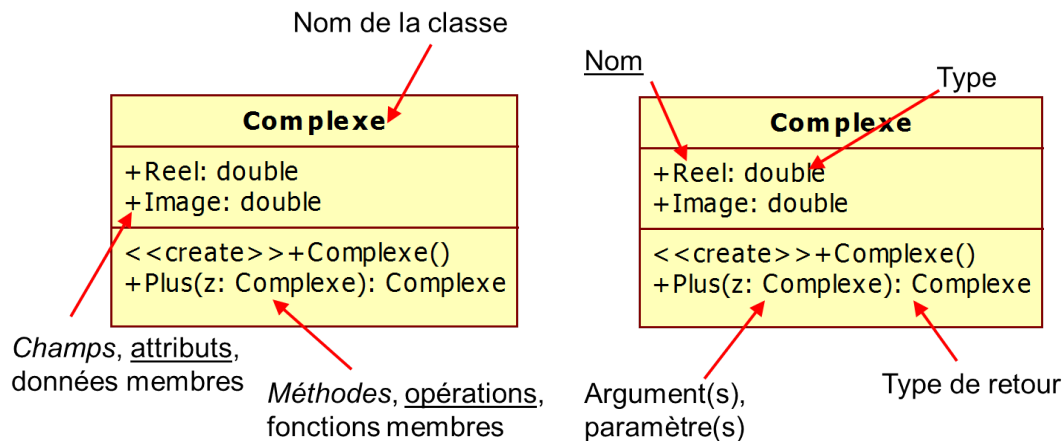


Figure 8.2: Diagramme de classe pour des nombres complexes, avec les terminologies usuelles.

membres avec leurs arguments et le type de retour de ces dites fonctions. Pour résumer cela, la figure 8.3 donne un exemple générique d'une classe avec les syntaxes pour chaque partie.

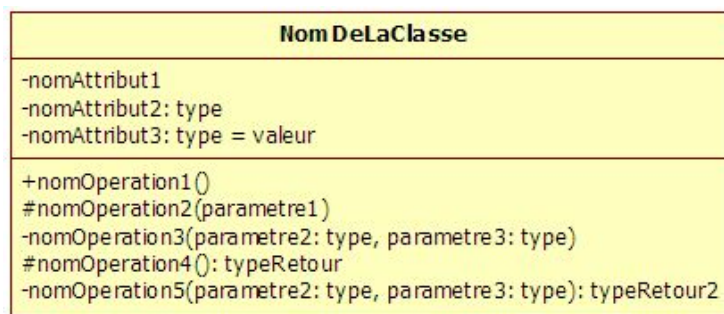


Figure 8.3: Syntaxes de chaque partie constituant la description d'une classe par son diagramme.

Un point important est la visibilité qui permet de préciser la restriction aux attributs et aux méthodes par les autres éléments du système (les autres classes). Cette visibilité est indiquée au début de chaque ligne de la classe par les symboles "+", "-", "#" ou "*sim*". La visibilité est détaillée dans le paragraphe 8.2.2.

### 8.1.2 UML et C++

Le C++ étant un langage orienté objet, l'UML permet bien évidemment de décrire les classes et objets qui seront implémentés en C++. Cependant, le langage C++ possède quelques mécanismes et fonctionnalités spécifiques qui ne sont pas décrits en conception UML. Il s'agit notamment de:

- la construction et la destruction des objets;
- les processus de copie d'objet (constructeur de copie et opérateur d'affectation);
- les règles d'héritage (en UML `public` uniquement, le C++ ajoute `protected` et `private`).
- le champs sur l'objet courant (`this`)

Pour les deux premiers points, le compilateur C++ ajoute automatiquement des méthodes par défaut (et naïves!) si celles-ci ne sont pas déjà présentes:

- le constructeur de copie;
- le constructeur par défaut (sans argument);
- l'opérateur d'affectation (ou de copie) =;

- le destructeur (par défaut déclaré avec le mot clé `virtual`).

Il arrive très souvent que ces méthodes par défaut soient utilisées par l'application alors que la conception UML ne les faisait pas apparaître. Ainsi, passer de la conception UML à une implémentation C++ nécessitera des ajustements de la conception. Afin d'illustrer ce paragraphe, voici une comparaison de deux diagrammes UML (l'un naturel, l'autre correspondant aux ajouts du C++) d'une même classe `Point` permettant de représenter et de manipuler (addition) des points du plan.

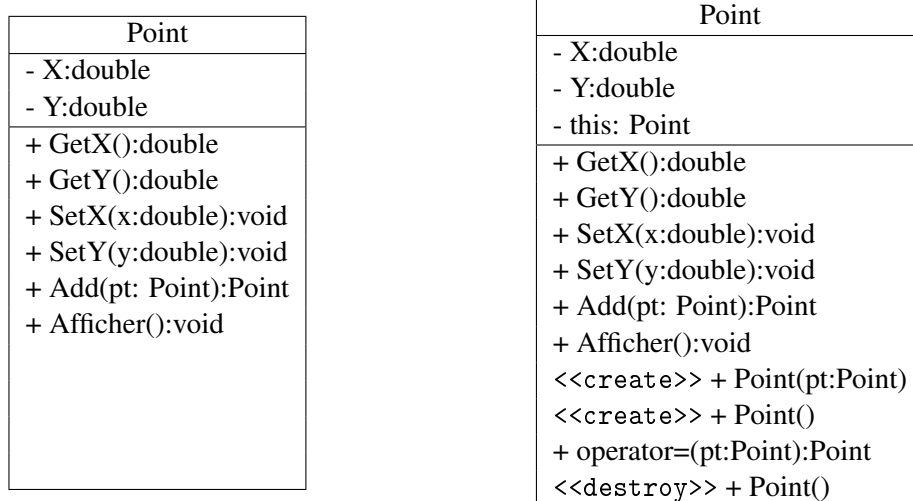


Table 8.1: Diagramme de classe UML (à gauche) et diagramme complet de l'implémentation C++ (à droite). Les trois dernières méthodes du deuxième diagramme sont automatiquement créées en implémentation C++.

La déclaration en C++ de la classe `Point` du diagramme UML (Table 8.1) est la suivante:

```

1  class Point
2  {
3  private:
4      double X;
5      double Y;
6  public:
7      Point Add(const Point &pt);
8      void Afficher();
9      double GetX();
10     void SetX(const double &x);
11     double GetY();
12     void SetY(const double &y);
13 };

```

## 8.2 Concept de classe pour l'encapsulation

On appelle *classe* la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc *issu* d'une classe. Plus rigoureusement, on dit qu'un objet est une instance (une réalisation) d'une classe. Pour désigner un objet, on parle aussi d'instance ou d'occurrence. L'intérêt des classes est de regrouper les données devant fonctionner ensemble (les coordonnées d'un point par exemple) et les fonctions associées à ces données (addition, distance entre 2 point, ...). Il s'agit du premier paradigme objet : l'encapsulation des données et des fonctions.

### 8.2.1 Définition de classe

Avant de manipuler un objet, il faut définir la classe dont il provient, c'est-à-dire décrire de quoi est composé la classe: fonctions et données (cf. paradigme d'encapsulation du cours). La définition d'une classe s'effectue dans un fichier d'entête (*header*) dont l'extension commence généralement par un **h** (.h, .hxx). Le nom de ce fichier est souvent le même que celui de la classe qu'il définit. Cette définition en C++ se fait de la manière suivante:

**R** Les crochets signalent des blocs optionnels.

```

1  class NomClasse [ : type_heritage NomClasseMere]
2  {
3  [ public:
4      // definition des champs et methodes publics
5  ]
6  [  NomClasse(); ] // Constructeur
7  [  ~NomClasse(); ] // Destructeur
8  [ protected:
9      // definition des champs et methodes proteges
10 ]
11 [ private:
12     // definition des champs et methodes privees
13 ]
14 };

```

Le mot clé `type_heritage` permet de définir un type d'héritage. Il existe trois types d'héritage possibles: `public`, `protected` ou `private` (cf. paragraphe 8.4). Le point virgule situé à la fin du bloc de définition de la classe (ligne 14) est obligatoire. Par défaut la visibilité des champs et méthodes d'une classe est `private`<sup>1</sup>.

### 8.2.2 Visibilité

L'utilisateur d'une classe n'a pas forcément besoin de pouvoir accéder directement aux données d'un objet ou à certaines méthodes. En fait, un utilisateur n'est pas obligé de connaître tous les détails de l'implémentation des méthodes et doit être considéré comme potentiellement maladroit: les méthodes de la classe doivent garantir le bon fonctionnement du système. En interdisant l'utilisateur de modifier directement les données, il est obligé d'utiliser les fonctions définies pour les modifier. Si ces fonctions sont bien faites, il est alors possible de garantir l'intégrité des données et un fonctionnement cohérent de la classe et ainsi du système. Différents niveaux de visibilité permettent de restreindre l'accès à chaque donnée (ou fonction) suivant que l'on accède à cette donnée par une méthode de la classe elle-même, par une classe fille (ou dérivée), ou bien d'une classe ou fonction quelconque.

En UML, il existe 4 niveaux de visibilité des éléments de la classe:

- '+' correspondant à `public`
- '#' correspondant à `protected`
- '~' correspondant à paquet (pas de correspondance dans les classes du C++)
- '-' correspondant à `private`.

La figure 8.4 liste les différents types de visibilité en UML et C++.

En C++, il existe 3 niveaux de visibilité des éléments de la classe:

1. `public` Aucune restriction. Il s'agit du plus bas niveau de restriction d'accès, toutes les fonctions membres (de n'importe quelle classe) peuvent accéder aux champs ou aux méthodes `public`,

<sup>1</sup>Contrairement aux structures : `struct`

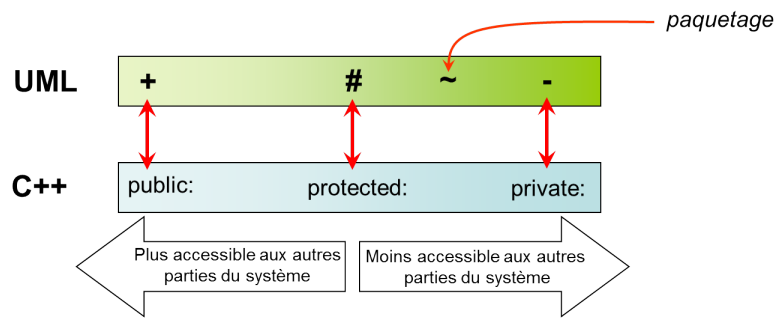
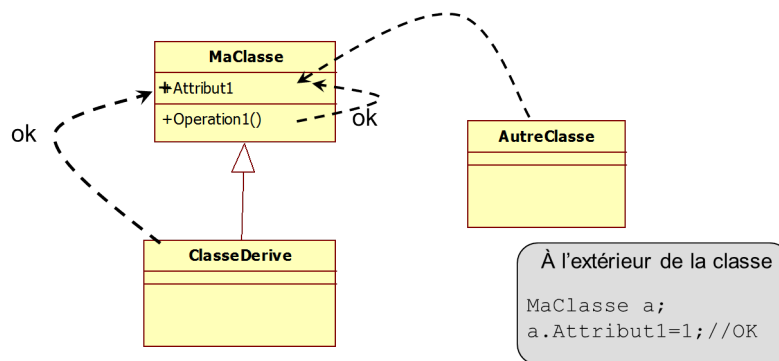
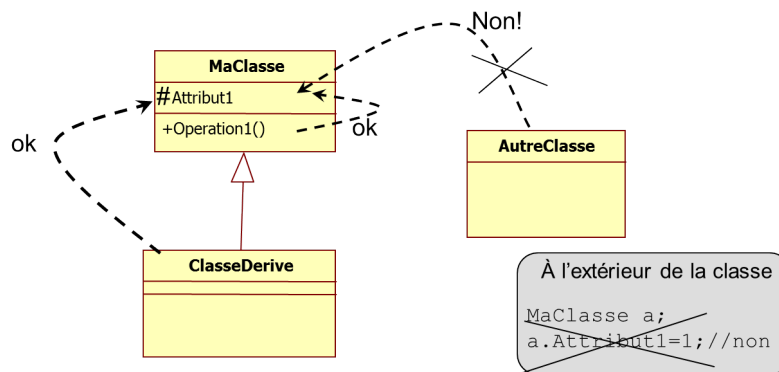


Figure 8.4: Symboles UML et C++ pour spécifier la visibilité

Figure 8.5: Illustration de la visibilité *public*

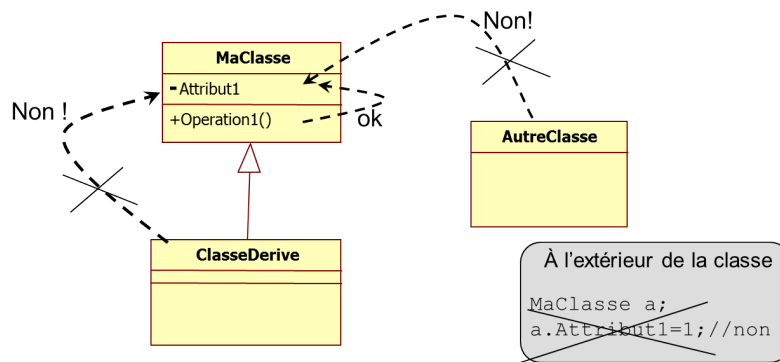
2. **protected** : l'accès aux variables et fonctions est limité aux fonctions membres de la classe et aux fonctions des classes filles,

Figure 8.6: Illustration de la visibilité *protected*

3. **private** Restriction d'accès le plus élevé. L'accès aux données et fonctions membres est limité aux méthodes de la classe elle-même.

### 8.2.3 Données membres

Les données membres (ou attributs ou champs) sont les variables principales d'un objet. Ces données sont le propre du concept de classe sous-jacent (un jour, un mois et une année pour le concept de Date par exemple, elles prendront une valeur bien particulière pour un objet qui sera VotreDateAnniversaire).

Figure 8.7: Illustration de la visibilité *private*

Pour les définir en UML, elles doivent être précédées de leur type et de leur visibilité (**public**, **protected** ou **private**) permettant de préciser quelles autres éléments du système peuvent accéder à ces valeurs (portée des champs). Il est recommandé d'utiliser des champs **private** ou **protected** pour les champs ayant un impact sur le bon fonctionnement des objets et du système. Ainsi l'accès à ces données ne peut se faire qu'au moyen de fonctions membres (ou méthodes) qui s'assurent de la bonne utilisation de ces variables. La visibilité de type **public** facilite la programmation de la classe (moins de méthodes à écrire) mais doit être réservée pour des champs dont le rôle n'est pas critique et n'affecte pas le fonctionnement de l'objet (la partie réelle et imaginaire d'une classe complexe par exemple).

Voici un exemple de déclaration d'une classe Point avec des attributs ayant différentes visibilités:

```

1  class Exemple
2  {
3      double X; // par défaut dans une classe, la visibilité est private
4      double Y;
5  protected:
6      char *Txt;
7  public:
8      double Val;
9  private:
10     Exemple *A; // champs privé, pointeur sur objet de type Exemple
11 };
  
```

Il est à noter que toute classe peut être utilisée comme type d'attribut. Cependant il est impossible de créer un objet du même type que la classe en construction (dans ce cas il faut passer par un pointeur ou une référence) ou un objet d'un type de la hiérarchie avale (car les objets fils n'existent pas encore).

### 8.2.4 Fonctions membres

Les données membres permettent de conserver des informations relatives à la classe, tandis que les fonctions membres (ou méthodes) représentent les traitements qu'il est possible de réaliser avec les objets de la classe. On parle généralement de fonctions membres ou méthodes pour désigner ces traitements. Il existe deux façons de définir des fonctions membres:

- en définissant la fonction (prototype et corps) à l'intérieur de la classe en une opération;
- en déclarant la fonction à l'intérieur de la classe et en la définissant à l'extérieur (généralement dans un autre fichier).

Pour des questions de lisibilité, la seconde solution est préférée. Ainsi, puisque l'on définit la fonction membre à l'extérieur de sa classe, il est nécessaire de préciser à quelle classe cette dernière

fait partie. On utilise pour cela l'opérateur de résolution de portée, noté `::`. A gauche de l'opérateur de portée figure le nom de la classe, à sa droite le nom de la fonction. Cet opérateur sert à lever toute ambiguïté par exemple si deux classes ont des fonctions membres portant le même nom. Voici un exemple de déclaration d'une classe `Valeur` avec attribut et méthodes:

```

1 //fichier Valeur.h
2 //Declaration des methodes a l'
  //interieur de la classe
3 class Valeur
4 {
5 private:
6     double X;
7 public:
8     void SetX(double);
9     double GetX();
10 };

```

```

1 // fichier Valeur.cpp
2 // Definition des methodes a l'
  //exterieur de la classe
3 void Valeur::SetX(double a)
4 {
5     X = a;
6 }
7
8 double Valeur::GetX()
9 {
10     return X;
11 }

```

### 8.2.5 Objets

En C++, il existe deux façons de créer des objets, c'est-à-dire d'instancier une classe:

- de façon statique,
- de façon dynamique.

#### Création statique

La création statique d'objets consiste à créer un objet en lui affectant un nom, de la même façon qu'une variable:

```
1 NomClasse NomObjet;
```

Ainsi l'objet est accessible à partir de son nom. L'accès à partir d'un objet à un attribut ou une méthode de la classe instanciée se fait grâce à l'opérateur `.`:

```

1 // Fichier Valeur.h
2 class Valeur
3 {
4 private:
5     double X;
6 public:
7     void SetX(double a)
8     { X = a; }
9     double GetX()
10    { return X; }
11 };

```

```

1 #include "Valeur.h"
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     Valeur a;
8     a.SetX(3);
9     cout << "a.X=" << a.GetX();
10    return 0;
11 }

```

#### Création dynamique

La création d'objet dynamique se fait de la façon suivante:

- définition d'un pointeur du type de la classe à pointer,
- création de l'objet *dynamique* grâce au mot clé `new`, renvoyant l'adresse de l'objet nouvellement créé,
- affecter cette adresse au pointeur,
- après utilisation: libération de l'espace mémoire réservé: `delete`.

Voici la syntaxe à utiliser pour la création d'un objet dynamique en C++:

```

1 NomClasse *NomObjet;
2 NomObjet = new NomClasse;

```

ou directement

```
1   NomClasse *NomObjet = new NomClasse;
```

Tout objet créé dynamiquement devra impérativement être détruit à la fin de son utilisation grâce au mot clé `delete`. Dans le cas contraire, une partie de la mémoire ne sera pas libérée à la fin de l'exécution du programme. L'accès à partir d'un objet créé dynamiquement à un attribut ou une méthode de la classe instanciée se fait grâce à l'opérateur `->` (il remplace `(*pointeur).Methode()`):

```
1   // Fichier Valeur.h
2   class Valeur
3   {
4   private:
5       double X;
6   public:
7       void SetX(double a)
8       { X = a; }
9       double GetX()
10      { return X; }
11  };

1   #include "Valeur.h"
2
3   int main()
4   {
5       Valeur *a = new Valeur;
6       a->SetX(3); // ou (*a).SetX(3);
7       delete a;
8       return 0;
9   }
```

### 8.2.6 Surcharge de méthodes

Toute méthode peut être surchargée<sup>2</sup>. On parle de surcharge de méthode lorsque l'on déclare au sein d'une classe une méthode ayant le même nom qu'une autre méthode (voir paragraphe 5.11.6). La différence entre les deux méthodes (ou plus) réside dans les arguments de ces méthodes: chaque fonction doit avoir une particularité dans ces arguments (type, nombre ou protection constante) qui permettra au compilateur de choisir la méthode à exécuter. Le type de retour d'une fonction ne constitue pas une différence suffisante pour que le compilateur fasse la différence entre plusieurs fonctions. Voici un exemple de surcharge de méthode:

```
1   class Point
2   {
3   private:
4       double X;
5       double Y;
6   public:
7       void SetX(int);           // définition d'une fonction
8       void SetX(int,int);      // bonne surcharge
9       void SetX(double);       // bonne surcharge
10      double SetX(double);      // mauvaise surcharge
11      void Set(Point &a);       // bonne surcharge
12      void Set(const Point &a); // bonne surcharge
13  };
```

Il faut noter qu'on peut distinguer 3 formes de surcharge en C++:

- la surcharge de méthodes utilisateur, comme dans l'exemple précédent.
- la surcharge de fonction par défaut comme le constructeur par défaut, le constructeur de copie, l'opérateur d'affectation (`operator=`) et le destructeur. Dans ce cas la fonction écrite par le développeur remplace celle par défaut du compilateur.
- la surcharge de fonction à l'héritage. Ceci sera vu plus loin (§ 8.4) et est un élément qui permettra de réaliser le polymorphisme d'objets (§ 8.5).

<sup>2</sup>Il n'existe qu'une seule méthode qui ne peut être surchargée: le destructeur. Cette méthode ne possède pas d'arguments... assez logique, non ?

## 8.3 Méthodes particulières

### 8.3.1 Constructeurs

Le constructeur est la fonction membre appelée automatiquement lors de la création d'un objet (en statique ou en dynamique après un `new`). Cette méthode est la première fonction membre à être exécutée. Elle est utilisée pour initialiser les attributs de la classe afin d'être dans un état déterminé. Un constructeur possède les propriétés suivantes:

- un constructeur porte le même nom que la classe dans laquelle il est défini;
- un constructeur n'a pas de type de retour (même pas `void`);
- un constructeur peut avoir des arguments.

La figure 8.8 illustre un code C++ avec un constructeur et la modélisation UML de ce constructeur.

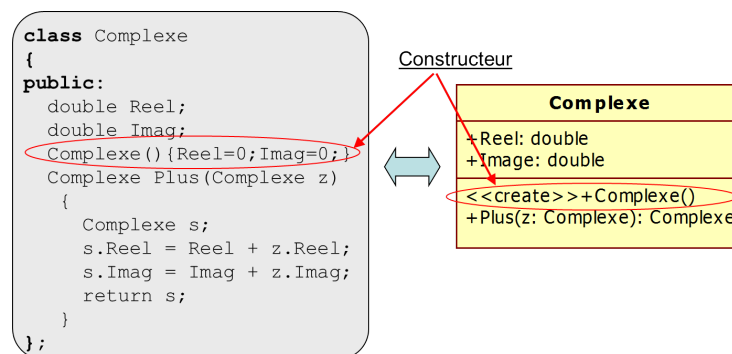


Figure 8.8: Constructeur en C++ et UML

La définition de cette fonction membre spéciale **n'est pas obligatoire**, notamment si vous ne souhaitez pas initialiser les données membres par exemple, et dans la mesure où un constructeur par défaut (appelé parfois constructeur sans argument) est défini par le compilateur C++ si la classe n'en possède pas. Voici un autre exemple d'utilisation d'un constructeur pour initialiser les attributs d'une classe:

```

1  class Point
2  {
3  private:
4      double X;
5      double Y;
6  public:
7      Point(double a=0,double b=0); // valeurs par défaut {0,0}
8  };
9
10 Point::Point(double a, double b)
11 {
12     X = a;
13     Y = b;
14 }
  
```

Comme pour n'importe quelle fonction membre, il est possible de surcharger les constructeurs, c'est-à-dire définir plusieurs constructeurs avec un nombre/type d'arguments différents. Ainsi, il sera possible d'avoir des comportements d'initialisation différents en fonction de la méthode de construction utilisée.

On distingue 3 formes de constructeurs:

- le constructeur par défaut, qui ne prend aucun paramètre: `Point p1, p2()`. Les champs sont initialisés à une valeur par *défaut*.



- les constructeurs utilisateurs, qui prennent différents arguments et permettant d'initialiser les champs en fonction de ces paramètres: `Point p3(1), p4(1,2);`
- le constructeur de copie, qui permet de créer un nouvel objet à partir d'un objet existant: `Point p5(p2);`. Ce type de constructeur est détaillé dans le paragraphe 8.3.2.

Il faut aussi noter qu'un seul constructeur est appelé et uniquement lors de la création en mémoire de l'objet. Il n'y a pas d'appel de constructeur pour les pointeurs et références. Ainsi `Point *pt;` n'appelle pas de constructeur (il sera appelé lors d'un `pt = new Point(1,2);`). De même, `Point rp=&p1;` n'appelle pas de constructeur pour la création de la référence `rp`.

Un constructeur peut être rendu particulièrement efficace en terme de cout mémoire et de temps pour l'initialisation des champs de la classe. En effet il est possible de passer des valeurs d'initialisation aux champs lors de leur création en mémoire. Ceci est fait ainsi sur l'exemple précédent de la classe `Point`:

```
1 Point::Point(double a, double b):X(a),Y(b)
2 { } // plus rien a mettre! X vaudra "a", et Y sera egale a "b"
```

### 8.3.2 Constructeur de copie

Le but de ce constructeur est d'initialiser un objet lors de son instantiation à partir d'un autre objet appartenant à la même classe (ou du même type). Toute classe dispose d'un constructeur de copie par défaut (et naïf) généré automatiquement par le compilateur. Naïf car il copie une à une les valeurs des champs de l'objet à copier dans les champs de l'objet à instancier. Toutefois, ce constructeur par défaut ne conviendra pas toujours, et le programmeur devra parfois en fournir un explicitement, typiquement quand la classe possède des champs dynamiques. La définition du constructeur de copie se fait comme celle d'un constructeur normal. Le nom doit être celui de la classe, et il ne doit y avoir aucun type de retour. Dans la liste des paramètres cependant, il devra toujours y avoir une référence sur l'objet à copier. Voici un exemple d'utilisation d'un tel constructeur:

```
1 class Valeur                                1 #include "Valeur.h"
2 {                                           2
3 private:                                     3 int main()
4     double X;                               4 {
5 public:                                       5     // constructeur
6     // constructeur                         6     Valeur v1(2.3);
7     Valeur(double a);                       7     // constructeur par copie
8     { X = a; }                               8     Valeur v2(v1);
9     // constructeur de copie               9     // constructeur par copie
10    Valeur(const Point &pt)                 10    Valeur v3 = v2;
11    { X = pt.X; }                            11    return 0;
12 };                                          12 }
```

### 8.3.3 Destructeurs

Le destructeur est une fonction membre qui s'exécute automatiquement lors de la destruction d'un objet (pas besoin d'un appel explicite à cette fonction pour détruire un objet). Cette méthode permet d'exécuter des instructions nécessaires à la destruction de l'objet. Cette méthode possède les propriétés suivantes:

- un destructeur porte le même nom que la classe dans laquelle il est défini et est précédé d'un tilde `~`;
- un destructeur n'a pas de type de retour (même pas `void`);
- un destructeur n'a pas d'argument;
- la définition d'un destructeur n'est pas obligatoire lorsque celui-ci n'est pas nécessaire.

Le destructeur, comme dans le cas du constructeur, est appelé différemment selon que l'objet auquel il appartient a été créé de façon statique ou dynamique.

- le destructeur d'un objet créé de façon statique est appelé de façon implicite dès que le programme quitte la portée dans lequel l'objet existe;
- le destructeur d'un objet créé de façon dynamique sera appelée via le mot clé delete.

Il est à noter qu'un destructeur ne peut être surchargé, ni avoir de valeur par défaut pour ses arguments, puisqu'il n'en a tout simplement pas. Voici un exemple de destructeur:

```

1  class Point
2  {
3  private:
4      double X;
5      double Y;
6      double *tab; // champs dynamique !!!
7  public:
8      Point(double a=0,double b=0); // constructeur
9      ~Point(); // destructeur
10 };
11
12 Point::Point(double a, double b)
13 {
14     X = a;
15     Y = b;
16     tab = new double[2];
17 }
18
19 Point::~~Point()
20 {
21     delete[] tab;
22 }
```

### 8.3.4 Opérateurs

Par convention, un opérateur *op* reliant deux opérands *a* et *b* est vu comme une fonction prenant en argument un objet de type *a* et un objet de type *b*. Ainsi, l'écriture *a op b* est traduite en langage C++ par un appel *op(a, b)*. Par convention également, un opérateur renvoie un objet du type de ses opérands. Cela permet pour de nombreux opérateurs de pouvoir intervenir dans une chaîne d'opérations (par ex. *a=b+c+d*).

La fonction C++ correspondant à un opérateur est représentée par un nom composé du mot *operator* suivi de l'opérateur (ex: *+* s'appelle en réalité *operator+( )*).

Un opérateur peut être défini en langage C++ comme une fonction membre, ou comme une fonction amie indépendante. Ici, nous nous limiterons au cas des fonctions membres. Tableau (8.2) donne une liste de l'ensemble des opérateurs qui peuvent être surchargés en C++:

#### Surcharge d'opérateur avec une fonction membre

Le langage C++ étend la notion de surcharge de fonctions aux opérateurs du langage. Dans ce type de surcharge d'opérateur, le premier opérande de notre opérateur, correspondant au premier argument de la fonction, va se trouver transmis implicitement. Ainsi l'expression *a op b* sera interprétée par le compilateur comme *a.operator op(b)* (exemple: *a+b*; est en fait *a.operator +(b)* ;). Le prototype de surcharge d'opérateur avec une fonction membre est le suivant (les crochets signalent des blocs non nécessaires):

+	-	*	/	%	^	&
	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*, ,	
->	[]	()	new	delete	type()	

Table 8.2: Liste des opérateurs que l'on peut surcharger en C++

```

1  NomClasse operator SymboleOperateur( const NomClasse & operande);
2  AutreClasse operator SymboleOperateur( const NomClasse & operande);
3  AutreClasse operator SymboleOperateur( const AutreClasse & operande);
4  //...
```

Voici un exemple de surcharge de l'opérateur +:

```

1  class Valeur
2  {
3  private:
4      double X;
5  public:
6      Valeur(double a=0) { X=a; }
7      //attention exemple PAS optimal!
8      Valeur operator+(Valeur pt)
9      {
10         Valeur result;
11         result.X = X + pt.X;
12         return result;
13     }
14 };

1  #include "Valeur.h"
2
3  int main()
4  {
5      Valeur a(2);
6      Valeur b(3);
7      // surcharge de l'operateur
8      Valeur c = a + b;
9      return 0;
10 }
```

### Transmission par référence

Dans le paragraphe précédent, nous avons présenté comment effectuer une surcharge d'opérateur avec une transmission d'argument et de valeur de retour par copie. Dans le cas de manipulation d'objet de grande taille, il est préférable d'utiliser une transmission d'arguments (voir de valeur de retour) par référence (gain en temps et économie de mémoire). Le prototype de surcharge d'opérateur avec une fonction membre, avec une transmission par référence, est le suivant (les crochets signalent des blocs optionnels):

```

1  NomClasse operator SymboleOperateur( [const] NomClasse & );
```

Voici un exemple de surcharge de l'opérateur +:

```

1  class Valeur
2  {
3  private:
4      double X;
5  public:
6      Valeur(double a=0) { X=a; }
7      Valeur operator+(
8          const Valeur &pt)
9      {
10         Valeur result;
11         result.X = X + pt.X;
12         return result;
13     }
14 };

```

```

1  #include "Valeur.h"
2
3  int main()
4  {
5      Valeur a(2);
6      Valeur b(3);
7      // surcharge de l'operateur
8      Valeur c;
9      c = a + b;
10     return 0;
11 }

```

Comme toutes fonctions, les opérateurs peuvent être surchargés. On peut donc définir un autre opérateur + entre un objet Valeur et un double :

```

1  class Valeur
2  {
3  private:
4      double X;
5  public:
6      Valeur(double a=0) { X=a; }
7      Valeur operator+( const Valeur &pt)
8      {
9          Valeur result;
10         result.X = X + pt.X;
11         return result;
12     }
13     Valeur operator+( const double &a)
14     {
15         Valeur result;
16         result.X = X + a;
17         return result;
18     }
19 };

```

Ainsi on pourra écrire:

```

1  #include "Valeur.h"
2
3  int main()
4  {
5      Valeur a(2);
6      // surcharge de l'operateur
7      Valeur c;
8      c = a + 3.14;
9      return 0;
10 }

```

### Surcharge d'opérateur avec un opérateur dyadique, et une fonction amie

Dans l'exemple précédent, on peut écrire  $c = a + 3.14$ ; mais pas  $c = 3.14 + a$ ; . On rappelle que l'écriture  $c = a + 3.14$ ; est l'écriture abrégée de  $c = a.operator+(3.14)$ , ce qui implique que  $c$ 'est un objet (ici  $a$ ) qui appelle son opérateur pour traiter l'argument (3.14). Dans le cas

$c = 3.14 + a$ ; 3.14 (donc le type double) qui appelle son opérateur sur  $a$  (classe Valeur). Le problème serait comment expliquer à "double" qu'il existe une opération "+" avec un objet de type Valeur?

Il faut pour cela passer par l'autre forme des opérateurs : la forme dyadique, c'est à dire qui prend en paramètres les 2 arguments. L'ordre des paramètres étant important, il y a deux fonctions:

```

1 Valeur operator+( const double &a, const Valeur &pt); // permettra c = 3.13
    + a; OK !
2 Valeur operator+( const Valeur &pt, const double &a); // permettra c = a
    +3.14 ; PB !
3                                     // en redondance avec "Valeur operator+( const
                                     double &a)"

```

Pour être appelée à partir de n'importe quel type d'objet, ces fonctions ne doivent pas appartenir à une classe: il s'agira de fonctions classiques (pas de fonction membre d'une classe: pas de Valeur:: dans l'exemple). **Ainsi pour l'écriture des instructions d'opérateur dyadique il ne sera pas possible d'accéder aux champs privés ou protégés directement.** Dans notre exemple (pas de fonctions d'accès au champ  $X$  qui est privé) il est impossible d'écrire l'opérateur dyadique +.

Pour résoudre ce problème, afin d'accéder directement aux champs privés ou protégés de la classe, l'utilisation du mot clé `friend` devra être utilisé. On déclare ainsi une fonction *amie* à la classe, sous entendue que cette fonction pourra accéder aux champs des objets.

```

1 class Valeur
2 {
3 private:
4 double X;
5 public:
6 Valeur(double a=0) { X=a; }
7 // fonction amie, operateur dyadique : double + Valeur
8 friend Valeur operator+( const double &a, const Valeur &pt);
9
10 // operateur classique Valeur + double
11 Valeur operator+( const double &a)
12 {
13 Valeur result;
14 result.X = X + a;
15 return result;
16 }
17 };
18
19 // Noter que pour la denomination d'une fonction amie,
20 // il n'y a pas l'appartenance a la classe
21 // ici pas de "Valeur::"
22 Valeur operator+( const double &a, const Valeur &pt)
23 {
24 Valeur result;
25 result.X = a + pt.X; // acces autorise au champ prive pour les fonctions
    amies
26 return result;
27 }

```

Cet exemple avec les opérateurs pourra être généralisé grâce à l'utilisation des modèles ou *patrons* (`template`) 8.6.

Un dernier exemple particulièrement utile est la surcharge de l'opérateur << pour effectuer un affichage avec cout:

```
1 class Valeur
2 {
3 private:
4 double X;
5 public:
6 Valeur(double a=0) { X=a; }
7 friend std::ostream & operator<<(std::ostream &os, const Valeur &pt)
8 {
9 return os << pt.X;
10 }
11
12 // ... suite de la classe
13 };
```

Ces deux fonctions amies permettent d'écrire:

```
1 #include <iostream>
2 #include "Valeur.h"
3
4 using namespace std;
5
6 int main()
7 {
8 Valeur a(2);
9 Valeur c;
10 c = 3.14 + a ; // appel de : Valeur operator+( const double &a,
11 const Valeur &pt)
12 cout << " c = " << c << endl; // appel de : std::ostream & operator<<( ... )
13
14 return 0;
15 }
```

Le mot clé Friend est aussi détaillé au paragraphe 9.2.

## 8.4 Héritage

L'héritage est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Le nom d'héritage (pouvant parfois être appelé dérivation de classe) provient du fait que la classe dérivée (la classe nouvellement créée, ou classe fille) contient les attributs et les méthodes de sa classe mère (la classe dont elle dérive). L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles hérités.

Par ce moyen, une **hiérarchie de classes** de plus en plus spécialisées est créée. Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante. De cette manière il est possible de récupérer des bibliothèques de classes, qui constituent une base, pouvant être spécialisées à loisir.

Un particularité de l'héritage est qu'un objet d'une classe dérivée est aussi du type de la classe mère...

### 8.4.1 Type d'héritage

En C++, il existe trois types d'héritage définis par les mots clés `public`, `protected`, `private`. Un type d'héritage définit les accès possibles qu'une classe dérivée a vis à vis des attributs et méthodes de la classe de base. Le prototype de l'héritage se fait à la définition de la classe:

```
1  class NomClasseDerivee : TypeHeritage NomClasseDeBase
2  {
3  // definition des attributs et methodes
4  // de la classe derivee
5  ...
6  };
```

#### Héritage public

L'héritage public se fait en C++ à partir du mot clé `public`. C'est la forme la plus courante de dérivation (et la seule décrite en UML). Les principales propriétés liées à ce type de dérivation sont les suivantes:

- les membres (attributs et méthodes) publics de la classe de base sont accessibles par **tout le monde**, c'est à dire à la fois aux fonctions membres de la classe dérivée et aux utilisateurs de la classe dérivée;
- les membres protégés de la classe de base sont accessibles aux fonctions membres de la classe dérivée, mais pas aux utilisateurs de la classe dérivée;
- les membres privés de la classe de base sont inaccessibles à la fois aux fonctions membres de la classe dérivée et aux utilisateurs de la classe dérivée.

De plus, tous les membres de la classe de base conservent dans la classe dérivée le statut qu'ils avaient dans la classe de base. Cette remarque n'intervient qu'en cas de dérivation d'une nouvelle classe à partir de la classe dérivée.

#### Héritage privé

L'héritage privé se fait en C++ à partir du mot clé `private`. Les principales propriétés liées à ce type de dérivation sont les suivantes:

- les membres (attributs et méthodes) publics de la classe de base sont inaccessibles à la fois aux fonctions membres de la classe dérivée et aux utilisateurs de cette classe dérivée;
- les membres protégés de la classe de base restent accessibles aux fonctions membres de la classe dérivée mais pas aux utilisateurs de cette classe dérivée. Cependant, ils seront considéré comme privés lors de dérivation ultérieures.

## Héritage protégé

L'héritage protégé se fait en C++ à partir du mot clé `protected`. Cette dérivation est intermédiaire entre la dérivation publique et la dérivation privée. Ce type de dérivation possède la propriété suivante: les membres (attributs et méthodes) publics de la classe de base seront considérés comme protégés lors de dérivation ultérieures.

## Résumé des héritages

Le tableau 8.3 récapitule toutes les propriétés liées aux différents types de dérivation.

Classe de base	Dérivée publique		Dérivée protégée		Dérivée privée	
	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur
public	public	O	protégé	N	privé	N
protégé	protégé	N	protégé	N	privé	N
privé	privé	N	privé	N	privé	N

Table 8.3: Les différents types de dérivation en C++

Voici un exemple d'utilisation d'un héritage public:

```

1 //fichier Point.h
2
3 class Point
4 {
5     protected:
6         float X,Y;
7     public:
8         Point(float a, float b)
9         {
10            X=a;
11            Y=a;
12        }
13        void SetX(double a) {X=a;}
14        void SetY(double b) {Y=b;}
15 };

```

```

1 //fichier PointColor.h
2
3 class PointColor : public Point
4 {
5     private:
6         short couleur;
7     public:
8         void SetCouleur(short a)
9         { couleur=a; }
10        void Afficher()
11        {
12            cout << "La couleur en ";
13            cout << X << " " << Y;
14            cout << " = " << couleur;
15        }
16 };

```

```

1 #include "PointColor.h"
2 #include <iostream>
3
4 using namespace std;
5
6 int main()
7 {
8     PointColor pt; // Objet instancie
9
10    pt.setX(3.4); // Utilisation d'une methode
11    pt.SetY(2.6); // de la classe de base
12
13    pt.SetCouleur(1); // Utilisation de methodes
14    pt.Afficher(); // de la classe derivee
15
16    return 0;
17 }

```



### 8.4.2 Appel des constructeurs et des destructeurs

#### Hiéarchisation des appels

Soit l'exemple suivant:

```

1
2 class A
3 { .....
4 public:
5     A()
6     ~A()
7     .....
8 };

```

```

1
2 class B : public A
3 { .....
4 public:
5     B()
6     ~B()
7     .....
8 };

```

Pour créer un objet de type B, il faut tout d'abord créer un objet de type A, donc faire appel au constructeur de A, puis le compléter par ce qui est spécifique à B en faisant appel au constructeur de B. Ce mécanisme est pris en charge par le C++: il n'y aura pas à prévoir dans le constructeur de B l'appel au constructeur de A.

La même démarche s'applique aux destructeurs: lors de la destruction d'un objet de type B, il y aura automatiquement appel du destructeur de B, puis appel de celui de A. Il est important de noter que les destructeurs sont appelés dans l'ordre inverse de l'appel des constructeurs.

Voici un exemple illustrant l'ordre des appels des constructeurs. Considérons les 2 classes suivantes:

```

1
2 class A
3 {
4 double X;
5 public:
6 A()
7 { cout << "A:Default" << endl; }
8 A(double a)
9 { X=a;
10 cout << "A:User" << endl; }
11 };

```

```

1 class B: public A
2 {
3 public:
4 B()
5 { cout << "B:Default" << endl; }
6 B(double a):A(a)
7 { cout << "B:User" << endl; }
8 };

```

La construction d'un objet B `ob1`; déclenche l'appel de `B()` **sans l'exécuter**. A cause de l'héritage, ce constructeur va immédiatement faire appel au constructeur par défaut de sa classe mère : `A::A()` qui lui va s'exécuter (ici allouer l'espace pour X puis faire l'affichage). Ensuite, le constructeur de la classe B est exécuté. Au final, X n'est pas initialisé et on obtient l'affichage suivant :

```

1 A:Default
2 B:Default

```

Si on construit l'objet B `ob2(3)`; les processus seront les mêmes sauf que le paramètre 3 lance l'appel du constructeur utilisateur `B::B(double a)` qui fait un appel explicite au constructeur utilisateur `A::A(double a)` (ce qui initialise X à 3). Ainsi X est initialisé à 3 et on obtient l'affichage suivant :

```

1 A:User
2 B:User

```

Ce mécanisme de transmission d'information entre constructeurs est décrit dans le paragraphe suivant.

#### Transmission d'informations entre constructeurs

En C++, il est possible de spécifier, dans la définition d'un constructeur d'une classe dérivée, les informations que l'on souhaite transmettre au constructeur de la classe de base. Voici un exemple d'héritage avec transmission d'informations entre constructeurs:

```
1  class Point
2  {
3  protected:
4      float X,Y;
5  public:
6      Point(float a, float b)
7      {
8          X=a;
9          Y=a;
10     }
11 };

1  class PointColor : public Point
2  {
3  private:
4      short couleur;
5  public:
6      PointColor(float a, float b,
7                  short c) : Point(a,b)
8      {
9          couleur=c;
10     }
11 };
```

On rappelle aussi que l'on peut aller plus loin pour le passage des paramètres entre constructeur: un constructeur peut appeler explicitement chacun des constructeurs de ces champs. Voici comment la classe *Point* pourrait être écrite.

```
1  class Point
2  {
3  protected:
4      float X,Y;
5  public:
6      Point(float a, float b): X(a), Y(b)
7      { } // il n'y a plus d'instructions !
8  };
```

Attention de bien respecter le même ordre pour la déclaration et l'initialisation des champs dans la classe, sinon un avertissement pourrait être donné par certains compilateurs.



5

Figure 8.9:

Figure 8.10:

## 8.5 Polymorphisme

Parfois appelé polymorphisme d'héritage. L'intérêt du polymorphisme est d'exécuter la méthode la plus appropriée à l'objet quand on manipule celui-ci indirectement (pointeur ou référence sur une classe parente). Il s'agit de changer le comportement d'un objet en fonction de sa spécialisation (héritage). L'intérêt principal est pour les objets *conteneurs* (qui contiennent des objets) comme les tableaux, les listes, ... quelque soit le type de l'objet stocké (pour peut qu'il appartienne à la même hiérarchie de classe) le comportement adapté à l'objet stocké.

La Programmation Orienté Objets autorise, sous certaines règles, les objets à changer de forme ou plutôt de type. Rappelons que le C++ est un langage fortement typé: le type des objets est obligatoirement connu à la compilation et sert à identifier les opérations à réaliser. Les règles qui vont permettre aux objets du C++ de changer de formes sont très strictes.

Voici les trois ingrédients indispensables à la mise en place du polymorphisme:

- une hiérarchie de classes (au moins une classe héritant d'une autre)
- une surcharge de fonction dans cette hiérarchie de classes avec de la virtualisation (`virtual`)
- un pointeur ou une référence de type amont et initialisé sur un objet de type aval de la hiérarchie

### 8.5.1 Définition

Le polymorphisme signifie ici "*peut prendre plusieurs formes*". D'abord, ce concept est relatif à la surcharge des méthodes (ou opérations) des classes d'une même hiérarchie. Ceci implique qu'une fonction soit présente dans la classe fille et dans la classe mère sous la même forme (nombre et type d'arguments similaires) comme illustré dans la figure 8.10

Ensuite, il permet d'étendre les possibilités du mécanisme d'héritage qui induit qu'**un objet d'une classe fille est du même type que sa classe mère**. En utilisant des pointeurs ou des référence, on va pouvoir accéder aux différents types de l'objet. Ainsi, les codes suivants sont exécutables:

```

1 A a1; // ok
2 B b1; // ok
3 A *pt_a = &b1; // ok!
4 A &ref_a = b1; // ok!
5
6 a1.afficher(); // appel de la fonction A::Afficher()
7 b1.afficher(); // appel de la fonction B::Afficher()
8 pt_a->Afficher(); // appel de la fonction A::Afficher()
9 ref_a.Afficher(); // appel de la fonction A::Afficher()
  
```

Cependant, les deux derniers affichages ne conduisent pas à l'appel de `B::Afficher()` pourtant ils travaillent avec un objet de type B: ils ne sont pas adaptés.

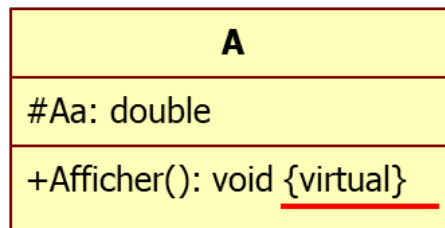


Figure 8.11: Mot clé virtual pour l'UML et le c++

C'est le dernier élément de la mise en oeuvre du polymorphisme qu'il faut ajouter : le mot clé `virtual` qui explique que la méthode `A::Afficher()` peut être surchargée dans une classe fille et qu'en fonction du type de l'objet appelant, il faudra faire appel à la fonction la plus adaptée. La figure 8.11 illustre son utilisation pour la classe A précédente.

```
1  class A
2  {
3  protected:
4  double Aa;
5  public:
6  A(double a=1):Aa(a) {}
7  virtual void Afficher()
8  { cout << Aa; }
9  };
```

## 8.6 Généricité

La généricité, parfois appelée polymorphisme paramétrique, n'a rien à voir avec le polymorphisme du C++ (le polymorphisme dit d'*héritage*). Le but de la généricité est de rendre générique les traitements aux **types** des variables. Ces variables concernées peuvent être des champs d'une classe ou les paramètres d'une fonction.

Le rappel précédent sur le langage fortement typé vaut aussi ici: le type d'une variable est d'une importance capitale. En C++, il va être possible de décrire des traitements sur des types de données quelconques (c'est ce qui est détaillé ici), il n'est cependant pas possible de les compiler. Ceci a deux conséquences:

- les méthodes génériques ne peuvent être écrites que dans des entêtes (.h par exemple) car ces fichiers ne sont pas compilés,
- lors de la production d'un exécutable, les types doivent être connus et fixés. Dès lors, les types génériques sont remplacés par le type fixé et les fonctions ou classes génériques peuvent alors être compilées.



# IV

## Part C: Thèmes choisis

<b>9</b>	<b>Syntaxe moderne du C++</b> .....	<b>99</b>
9.1	C++11, C++14 et C++17	
9.2	Mot clés	
9.3	Messages d'aide au débogue	





## 9. Syntaxe moderne du C++

### 9.1 C++11, C++14 et C++17

Le langage C++ est un langage vivant et normé. Ainsi de nouvelles fonctionnalités sont ajoutées progressivement au langage. Ces nouveautés émergent des problèmes récurrents observés ou de nouveauté provenant de la théorie (modélisation).

La norme en vigueur est celle publiée en septembre 2011 et formellement appelée ISO/IEC 14882:2011, et de manière informelle : C++11. Le C++14 sera une évolution mineure du C++11 (quelques précisions de la norme, finalisation de mécanismes). Une nouvelle norme est attendue en 2017 (C++17).

Les changements et apports les plus importants de cette norme sont les suivants:

1. déduction automatique du type à la compilation : mot clé `auto`,
2. parcours de boucle automatique: les *range-for*, toujours avec le mot clé `for`<sup>1</sup>,
3. les références universelles : `&&`,
4. les fonctions lambdas. Il s'agit de fonctions anonymes temporaires, définies à la volée dans un contexte (scope et class/struct/union) capable d'interagir (intercepter) avec des objets du contexte: `[]`, `[&]`, `[=]`, `[this]`, ... On peut les voir comme une alternative élégante aux pointeurs de fonctions.
5. extensions des classes de la std: `chrono`, `tuple`, `thread` ...
6. extensions de la gestion des *template* (`decltype`...) et de la mémoire (`mutex`, pointage unique, ...)
7. initialisation des champs lors de leur définition ( usage à réserver aux classes très simples, ou très (très) complexes (avec de nombreux constructeurs différents) ).

Les évolutions liées au matériel (multi-coeur des processeurs par exemple) sont rarement couverts par les normes du langage mais plus par celles des compilateurs. Ainsi le compilateur gcc 5.0 utilise par défaut le C++11 et la bibliothèque OpenMP en version 3 pour les optimisations des processeurs (dont les instructions SIMD)<sup>2</sup>

<sup>1</sup>en complément des boucles `for_each`

<sup>2</sup>Pour voir les macros supportées par le compilateur gnu : `g++ -E -dM - </dev/null` et la version du compilateur `gcc -v`

## 9.2 Mot clés

### friend

Une classe peut avoir des fonctions amies ou des classes amies. On détail ici les fonctions amies, l'extension aux classes amies d'une classe est logique.

Une fonction amie est obligatoirement déclarée dans une classe, cependant elle n'appartient pas à (ou ne compose pas) cette classe et n'est donc pas nécessairement appelée à partir d'un objet de la classe. En conséquence, elle ne dispose pas de `this`. Elle pourra cependant accéder à tous les champs et méthodes de la classe, même ceux dont la visibilité est limitée comme `private` ou `protected` à condition d'avoir passer un argument du type de la classe.

L'utilisation la plus fréquente des fonctions amies est la surcharge des opérateurs d'une classe. En effet, il est très commode de surcharger les opérateurs (notamment arithmétiques) par des fonctions amies pour résoudre le problème de commutativité. Attention cependant au fait qu'une fonction amie n'est pas une fonction membre, donc pas de `this` et ainsi : il faut écrire les opérateurs n'ont plus sous leur forme interne (ou implicite) mais sous leur forme externe. Généralement, les deux formes ne sont pas présentes simultanément.

```

1  class Complexe
2  {
3  public:
4  double Re, Im; // Parties reelles et imaginaires
5  Complexe(double x=0, double y=0);
6
7  // Operateur + en forme externe (equivalent a Complexe operator+(const
   // Complexe &z2)
8  friend Complexe operator+(const Complexe &z1, const Complexe &z2);
9
10 friend Complexe operator+(const double &a, const Complexe &z); // pour
   // 3.14 + Complexe(1,2)
11 friend Complexe operator+(const Complexe &z, const double &a); // pour
   // Complexe(1,2) + 3.14
12 };

```

Pour les opérateurs arithmétiques, l'utilisation des `template` est à considérer pour limiter le nombre de fonction amies. Par exemple:

```

1  template<class T> friend Complexe operator+(const T& a, const Complexe &z);

```

permettra d'adapter l'addition complexe à n'importe quel type d'objet... même les non-numériques (considérer `std::enable_if`).

A noter que la qualification `friend` ne se propage pas lors de l'héritage: une fonction amie d'une classe mère n'est pas amie pour les classes filles.

### volatile

Exemple

### inline

Pour les fonctions, permet de copier le code de la fonction à l'endroit de l'appel, à la place de faire un appel à la fonction. Ceci permet d'optimiser le temps d'exécution,

- dans le cas où la fonction `inline` est très petite (le cout de l'appel est aussi long que le code de la fonction)
- dans le cas où la préparation des paramètres pour l'appel de la fonction est trop couteux en temps (copie en mémoire).

Exemple, voici le fichier `ex_inline.cpp`:

```

1  #include <iostream>
2  using namespace std;
3
4  // La declaration __attribute__((always_inline)) n'est necessaire que pour
   // cet exemple simple
5  // afin d'etre certain que quelque soit l'optimisation de compilation la
   // fonction soit bien inline.
6  inline float squareme_inline(float a) __attribute__((always_inline));
7
8  inline float squareme_inline(float a)
9  {
10     return a*a;
11 }
12
13 float squareme(float a)
14 {
15     return a*a;
16 }
17
18 int main(int argc, char *argv[])
19 {
20     float a, b, c;
21     float a2, a2_f_inline, a2_f;
22     a = atof( argv[1] );
23     b = atof( argv[2] );
24     c = atof( argv[3] );
25
26     a2          = a * a;
27     a2_f        = squareme( b );
28     a2_f_inline = squareme_inline( c );
29
30     return 0;
31 }

```

Après la compilation par `g++ -g ex_inline.cpp -c -o ex_inline.o` voici un extrait du contenu du fichier `ex_inline.lst` obtenu avec `objdump -S ex_inline.o > ex_inline.lst` pour un processeur intel:

```

1  a2          = a * a;
2  7d:  f3 0f 10 45 fc      movss  -0x4(%rbp),%xmm0 // -0x4(%rbp) = 'a'
3  82:  f3 0f 59 45 fc      mulss  -0x4(%rbp),%xmm0
4  87:  f3 0f 11 45 ec      movss  %xmm0,-0x14(%rbp) // -0x14(%rbp) = 'a2'
5  a2_f        = squareme( b );
6  8c:  8b 45 f8            mov    -0x8(%rbp),%eax // -0x8(%rbp) = 'b'
7  8f:  89 45 dc            mov    %eax,-0x24(%rbp)
8  92:  f3 0f 10 45 dc      movss  -0x24(%rbp),%xmm0
9  97:  e8 64 ff ff ff      callq  0 <_Z8squaremef> //appel de squareme()
10 9c:  66 0f 7e c0         movd   %xmm0,%eax
11 a0:  89 45 e4            mov    %eax,-0x1c(%rbp) // -0x1c(%rbp) = 'a2_f'
12
13 a3:  f3 0f 10 45 f4      movss  -0xc(%rbp),%xmm0 // -0xc(%rbp) = 'c'
14 a8:  f3 0f 11 45 f0      movss  %xmm0,-0x10(%rbp) // -0x10(%rbp) = copie
   // de 'c'
15 return a*a;
16 ad:  f3 0f 10 45 f0      movss  -0x10(%rbp),%xmm0

```

```

17   b2:  f3 0f 59 45 f0          mulss  -0x10(%rbp),%xmm0
18   a2_f_inline = squareme_inline( c );
19   b7:  f3 0f 11 45 e8          movss  %xmm0,-0x18(%rbp) //-0x14(%rbp) = '
      a2_f_inline'

```

On remarquera que le code des lignes '7d:' à '87:' est similaire à celui de 'ad:' à 'b7:': il s'agit respectivement des codes pour l'appel direct à la multiplication et à la fonction `inline`.

### static

La seule différence avec une variable normale, est qu'une variable `static` ne va être créée qu'une seule fois. Il peut s'agir d'une variable à l'intérieur d'une fonction ou bien d'un champ d'une classe (ou encore d'une variable globale). Cette variable n'est accessible que dans son contexte de définition: le code (scope de définition) dans une fonction ou sa visibilité dans une classe. Il s'agit d'un liage interne. Voici un exemple de variable statique dans une fonction.

```

1  #include <iostream>
2  using namespace std;
3
4  int countme()
5  {
6  static int j=0;
7  j++;
8  return j;
9  }
10
11 int main()
12 {
13 cout << countme() << endl; //affiche 1
14 cout << countme() << endl; //affiche 2
15 return 0;
16 }

```

On peut ainsi voir les variables `static` comme des variables globales (une seule instance, initialisables, jamais détruites) visibles et accessibles comme des variables locales. Il s'agit donc d'une alternative à l'utilisation de variable globale.

### extern

Une variable `extern` est assez similaire à une variable `static` sauf qu'il s'agit d'utiliser une variable déclarée dans un autre contexte (autre scope). C'est à dire soit une autre fonction ou un autre fichier. Une fonction peut aussi être `extern`. Il s'agit d'un liage externe : la variable ou la fonction est dans un autre fichier objet... possiblement venant d'un autre langage de programmation!

Voici le premier fichier `ex_extern_def.cpp` qui ne contient pour cet exemple que la déclaration et initialisation de la variable `j`:

```

1  int j = 0;

```

Le second fichier `ex_extern.cpp` implémente un compteur et un affichage:

```

1  #include <iostream>
2  using namespace std;
3
4  int countme()
5  {
6  extern int j;
7  j++;
8  return j;
9  }

```

```

10
11 int main()
12 {
13 int j=10;
14 cout << countme() << endl; // affiche 1
15 cout << countme() << endl; // affiche 2
16 return 0;
17 }

```

Il faut compiler ces fichiers puis les lier ensemble. Ceci peut être fait simplement par `g++ -g ex_extern.cpp ex_extern_def.cpp -o ex_extern.o`.

L’affichage sera bien 1 puis 2 car la variable *j* utilisée par la fonction *countme* est celle qui est globale (définie dans *ex\_extern\_def.cpp*). On rappelle que la variable *j* déclarée dans le *main* n’est pas visible par la fonction *countme*. Enfin, il sera déconseillé d’utiliser des noms communs pour les variables globales afin d’éviter les ambiguïtés... *j* est ici un très mauvais exemple.

### C++ cast : conversion de types

Les conversions de type apparaissent fréquemment. S’il est facile de prévoir ce qui sera fait avec des types simples (`float` vers `double`, `double` vers `int`...), ceci est moins évident avec des objets plus complexes, ou issus de classes héritées, ou encore des `template`.

Il existe plusieurs manières de convertir des objets:

1. **les conversions implicites.** Elles ne demandent aucun opérateur ou symbole particuliers et se produisent automatiquement lors de copie d’une valeur vers un type compatible. Quand il y a perte de précision, le compilateur peut le signaler par un avertissement.

```

1 int v_i = 12;
2 double v_d;
3 v_d = v_i; // conversion implicite int -> double

```

Pour les classes, il est possible de favoriser une conversion implicite en écrivant un constructeur d’une classe B dont le paramètre est d’un autre type (A). On permet ainsi la conversion implicite de A vers B.

```

1 class A {
2 public:
3 double Aa;
4 };
5 class B {
6 double Ba;
7 public:
8 B(A a) { Ba = Aa;}
9 };
10 int main()
11 {
12 A a;
13 B b = a; // conversion de A -> B
14 return 0;
15 }

```

2. **les conversions explicites : fonctionnelles et C-style.** Lorsque l’on précise explicitement la conversion, on réalise une conversion explicite. Ces conversions n’engendrent pas d’avertissement en cas de perte de précision.

```

1 int v_i = 12;
2 double v_d;
3 v_d = (double) v_i; // conversion c-like int -> double

```

```
4   v_d = double(v_i); // notation fonctionnelle (par constructeur) int
    -> double
```

Ce type de conversion est très efficace et suffit dans de nombreux cas avec les types standards. Cependant, il est tout a fait possible d'utiliser ces syntaxes avec des objets de classes et des pointeurs d'objets. Mais malgré l'absence d'erreur de syntaxe, l'exécution peut donner des résultats inattendus.

```
1   class A {
2   public:
3   double Aa;
4   };
5   class B {
6   public:
7   int Ba;
8   B(int b) { Ba = b;}
9   int GetBa() { return Ba; }
10  };
11  int main()
12  {
13  A a; a.Aa = 3.14;
14  B *b = (B*) &a; // c-like A* -> B*
15  std::cout << b->GetBa() << std::endl; // ne marche pas!
16  return 0;
17  }
```

L'instruction `b->GetBa()` est légale mais utilise un espace mémoire inapproprié. C'est pour résoudre ces problèmes que les conversions spécifiques qui suivent ont été introduites.

3. les conversions `static_cast`
4. les conversions `dynamic_cast`
5. les conversion `reinterpret_cast`
6. les conversions constantes `const_cast`

#### `explicit`

Ce mot clé est très utilisé pour les constructeurs et, depuis le C++11, pour les opérateurs de conversions. Par défaut, toutes les fonctions sont implicites: le compilateur est autorisé à convertir implicitement le type des paramètres et à exécuter les initialisations par copie. En ajoutant devant un constructeur, ou une fonction de conversion de type, le mot clé `explicit` ces mécanismes deviennent interdits (erreur de compilation).

```
1   class A
2   {
3   private:
4   int m_A;
5   public:
6   A(int a): m_A(a) {} // constructeur avec un seul parametre int
7   int GetA() {return m_A;}
8   };
9
10  void DoSomething(A a)
11  { int i = a.GetA (); }
12
13  int main()
14  {
15  DoSomething(42); // l'int '42' est converti implicitement en objet de type
    A
```

```

16     // DoSomething( A(42) );
17 }

```

Exemple avec une conversion de type:

```

1  class A
2  {
3  public:
4      A(int a) {}
5      operator int() const { return 0; }
6  };
7
8  class B
9  {
10 public:
11     explicit B(int b) {}
12     explicit operator int() const { return 0; }
13 };
14
15 int main()
16 {
17     // A n'a pas de constructeur explicit ou de conversion
18     // Tout est ok
19     A a1 = 1;
20     A a2 ( 2 );
21     A a3 { 3 };
22     int na1 = a1;
23     int na2 = static_cast<int>( a1 );
24
25     B b1 = 1; // Erreur: conversion implicite de int vers B
26     B b2 ( 2 ); // OK: appel explicite au constructeur
27     B b3 { 3 }; // OK: appel explicite au constructeur
28     int nb1 = b2; // Erreur: conversion implicite de B vers int
29     int nb2 = static_cast<int>( b2 ); // OK: explicit cast
30 }

```

Il est utile d'interdire les conversions implicites quand l'utilisation devient ambiguë. Par exemple, avec un constructeur `MyString(int : size)` qui construit une chaîne de caractères de taille `size` et une fonction d'affichage `print(MyString: &)`, un appel à `print(3)` ne produira pas l'affichage "3" mais celui d'une chaîne vide de taille 3...

### using

Le mot `using` a vocation de créer des raccourcis pour les noms de fonctions, de fonctions dans un `namespace` ou dans une classe mère, ou de type (nouveau avec le C++11).

Le mot clé `using` s'utilise de 3 manières différentes:

- comme directive pour l'utilisation de `namespace` (exemple : `using namespace std;`) ou de membre de `namespace` (*using-declaration*),
- la déclaration d'utilisation de membres de classe (*using-declaration*),
- déclaration d'alias de type (C++11): il remplace `typedef`.

Détaillons rapidement les deux modes *using-declaration*. Tout d'abord pour un `namespace`:

```

1  #include <iostream>
2  #include <string>
3  #include <vector>
4
5  // using namespace std; // exemple familier

```



```

6 using std::string; // on precise qu'on utilisera std::string
7
8 int main()
9 {
10     std::vector<double> a; // designation complete
11     string s = "Hello"; // designation raccourcie grace au using std::string
12
13     // a la volée:
14     using std::cout; // ici le cout
15     cout << s << std::endl; // mais pas le endl
16
17     return 0;
18 }

```

Ensuite dans une classe:

```

1 class A
2 {
3     protected:
4         int Aa;
5     public:
6         A(int a);
7         // A dispose des constructeurs suivants:
8         // 1- A(int)
9         // 2- A(const A&)
10        // 3- A(A&&) C++11
11    };
12
13    class B: public A
14    {
15        public:
16        using A::Aa; // le champs Aa est maintenant public a partir de B!
17        using A::A; // B herite des constructeurs de A (disposant des memes et
18        // bons arguments)
19        // B dispose des constructeurs:
20        // 1- B() : constructeur par default
21        // 2- B(const B&)
22        // 3- B(B&&) C++11
23        // 4- B(int) : herite de A!
24    };

```

### auto

Le mot clé `auto` a subi un lifting avec la version C++11. Avant il permettait de spécifier la durée de vie d'une variable déclarée dans un bloc d'instructions ou dans une liste de paramètres d'une fonction. Il indiquait qu'une telle variable avait un comportement par défaut (normal...) : allouée au début du bloc et supprimée (dé-allocation) à la fin du bloc.

Depuis le C++11, `auto` permet de déclarer une variable (ou fonction) en spécifiant que le type de la variable sera automatiquement déduit à partir de son initialisation. Attention, le C++ étant un langage fortement typé, ceci implique que l'initialisation soit connue à la compilation. A noter que ce mot clé pousse à déclarer les variables à la volée dans le code.

```

1 #include <iostream>
2 #include <typeinfo>
3
4 int main()
5 {

```



```

6  auto a = 1 + 2; // int
7  std::cout << " type de a : " << typeid(a).name() << std::endl;
8  // --> type de a : int
9
10 auto l = { 1, 2, 3}; // une liste d'entier, et non un tableau
11 std::cout << " type de l : " << typeid(l).name() << std::endl;
12 // --> type de l : std::initializer_list<int>
13
14 return 0;
15 }

```

### boucle for sur plage

Une nouvelle syntaxe pour la boucle `for` permet d'alléger l'écriture des boucles, notamment pour les objets complexes (vector, list, map, ...). La nouvelle syntaxe est :

```

1  for( type variable_evoluant : variable_a_balayer)
2  { instructions de boucles }

```

Il faut préciser le type et le nom de la variable évoluant. C'est celle-ci qui va prendre les une après les autres, toutes les valeurs contenues dans `variable_a_balayer`. Il ne faut pas la confondre avec l'indice de la syntaxe classique. La variable à balayer est soit un **tableau statique** (par exemple `float v[10]`;) ou soit n'importe quel **objet conteneur de la std** (`std::vector<double>` par exemple). Le type de la variable évoluant est donc le type des éléments stockés dans le tableau. Voici un exemple simple d'affichage d'un tableau statique:

```

1  int tab[]={1,2,3,4};
2  for (const int& val : tab) // acces par reference constante
3  std::cout << val << " "; // pas de modification de val possible

```

Un autre exemple de mise à 0 des éléments d'un tableau

```

1  float tab[10];
2  for( float& val : tab) // acces par reference aux elements de tab
3  {
4  val=0; // mise a 0 des valeurs de tab et ne nuit pas aux iterations!
5  std::cout << val << " "; // puis affichage
6  }

```

Cette nouvelle syntaxe se prête tout particulièrement à l'utilisation du mot clé `auto`. La syntaxe recommandée est alors:

```

1  float tab[10];
2  for( auto&& val : tab) // acces par reference aux elements de tab (val est
3  de type float &)
4  val=0; // mise a 0 des valeurs de tab

```

Et pour un objet vector

```

1  std::vector<double> tab={3.14, 4, 5.5, 10, 2, 4};
2  for( auto&& val : tab) // acces par reference aux elements de tab (val est
3  de type double &)
4  val=0; // mise a 0 des valeurs de tab

```

Un dernier exemple avec les map, chaque partie d'élément est toujours accessible avec `.first` et `.second`.

```

1  std::map<char, int> letter_counts {'a', 27}, {'b', 3}, {'c', 1}};
2  for( auto && f : letter_counts)
3  std::cout << "f:" << f.first << " -s:" << f.second << std::endl;

```

Les limitations de cette syntaxe sont :

- ne pas utiliser sur des tableaux dynamiques: il n'est pas toujours possible que le compilateur trouve les extrémités du tableau,
- il n'est pas possible naturellement de faire de modifications sur l'indice (passer les éléments de 2 en 2 par exemple)
- sur les objets simples (tableaux statiques, `vector`, `list`), on ne peut pas récupérer l'indice de l'élément en cours,
- si vous écrivez votre propre classe conteneur, pour être utilisable il faudra implémenter un `begin` et un `end` soit sous forme de données membre, soit sous forme de fonction.

### 9.3 Messages d'aide au débogue

Pour faciliter la mise au point de gros programme, la fonction du C `assert` peut être très utile. Elle permet de réaliser un test et si celui-ci est faux de quitter le programme (`abort()`) en envoyant un message sur la sortie standard d'erreur incluant: le nom du programme, le fichier et le numéro de ligne de la fonction `assert` en défaut, le nom de la fonction incluant l'`assert` et le test qui a échoué.

Considérons le programme `test_assert.cpp` suivant:

```

1  #include <iostream>
2  #include <cassert>
3
4  using namespace std;
5
6  void Affiche(double *v, int t)
7  {
8  assert(t>0); // si la taille n'est pas >0 assert!
9  for(unsigned int i=0; i<t; i++)
10     cout << v[i] << " ";
11  cout << endl;
12  }
13
14  int main()
15  {
16  double a[ ]={2,3,4,5};
17  Affiche(a, sizeof(a)/sizeof(double) );
18  Affiche(a,0);
19  cout << "Terminons!" << endl;
20  return 0;
21  }
```

Voici ce qui sera affiché lors de l'exécution:

```

2 3 4 5
test_assert: test_assert.cpp:9: void Affiche(double*, int): Assertion 't>0' failed.
Abandon
```

Les `assert` peuvent être désactivés grâce à la définition de la variable `NDEBUG`:

- soit dans le programme **avant** d'inclure `cassert` (ou `assert.h`): `#define NDEBUG`
- soit à la compilation : `g++ test_assert.cpp -o test_assert -DNDEBUG`


Dans ce cas voici la sortie du programme:

```

2 3 4 5
```

```

Terminons!
```

-  A noter que le C++11 et le C++17 ajoutent le mot clé `static_assert` au C++. L'objectif n'est pas le même que pour `assert`: `static_assert` doit avoir un test *constant*, c'est à dire connu au moment de la compilation. `static_assert` permettra de tester des compatibilités de types, de gestion d'exception, de fonctionnalités sur les types, etc... voir `#include <type_traits>` pour ce qui est testable.







## Bibliographie

- [Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848 (cited on page 8).
- [HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728 (cited on page 11).







## Index