Synthèse du cours **INFOF-105**
Langages de Programmation
Théorie des langages

Table des matières

Chapitre 1

Programmation et langage

1.1 programme et algorithme

L'informatique est la *science du traitement automatique de l'information par une machine*. La machine en question est soit une abstraction (machine de Turing), soit un ordinateur réel avec certaines caractéristiques. On peut donc définir ce qu'est **programmer** comme *la réalisation d'une séquence d'opérations élémentaires définissant un traitement particulier sur une machine visée*. Afin de programmer, il faut deux étapes distinctes : premièrement, il faut concevoir l'algorithme (structures de données et principe de fonctionnement), et il faut ensuite implémenter cet algorithme dans un contexte particulier pour le faire fonctionner sur une machine visée.

Un **programme** est donc une *traduction contextualisée* d'un algorithme. La notion de programmation recèle alors plus de l'art que de la science (notons le livre *The Art of Computer Programming* de Donald Knuth) car elle n'a rien d'exact et s'apprend par l'exercice.

Contrairement à un algorithme, un programme doit être complet (environnement d'exécution, point d'entrée, etc.) alors qu'un algorithme ne concerne habituellement qu'une partie de la gestion du traitement.

1.2 Langage de programmation

Un langage de programmation est une interface entre le code binaire directement compréhensible par la machine et ce que produit l'humain. Un langage de programmation doit être intelligible par un humain, défini de manière univoque et non ambiguë, et standardisé. Un programme n'est autre que du texte suivant certaines conventions (celles du langage). La définition d'un langage de programmation est donc *une convention d'écriture compréhensible par l'humain et traduisible de manière univoque par une machine*.

1.3 Haut niveau et bas niveau

Les langages sont souvent classés selon leur paradigmes, mais il sont également séparés dans deux grandes familles : les langages de **haut niveau** qui disposent d'une abstraction élevée du type de machine visée, et les langages de **bas niveau** (ou d'assemblage) qui sont totalement dépendants des machines visées à l'exécution.

1.4 Compilation

Bien que le programme soit du **texte** exprimé dans un langage particulier, son but est d'être exécuté sur une machine. La traduction du code source (fichier contenant le programme) en code exécutable par un type de machine visé est appelée **compilation**. Le logiciel qui se charge de cette traduction est appelé **compilateur** et transforme le code source en **code objet** (format propre à la machine visée). Le compilateur peut soit refuser le code source en provoquant une erreur (le programme n'est pas conforme à la définition du langage) soit l'accepter et le traduire littéralement. Attention cependant, le fait qu'un code soit accepté par le compilateur ne le rend pas correct pour autant. Il l'est syntaxiquement mais pas obligatoirement sémantiquement. Une fois le code objet produit, il n'est toujours pas exécutable. Pour ce faire, il faut passer par l'**édition des liens**, la phase qui gère les éventuels liens entre le programme et des parties externes de bibliothèques ou avec d'autres codes objets.

Un programme ne doit pas obligatoirement être compilé entièrement en une fois : il existe un principe appelé *compilation séparée* qui permet de compiler plusieurs **unités de compilation** différentes à différents moments et de les assembler seulement lors de

l'édition des liens (création du binaire exécutable). Cela permet de ne pas avoir à recompiler tout un projet conséquent à chaque modification : seules les entités modifiées sont recompilées, les anciennes unités de compilations sont conservées.

1.4.1 Conséquences de la compilation

La compilation transforme le programme (séquence de texte) en une séquence binaire exécutable plus tard. Une conséquence de ce traitement est (entre autres) que tous les noms donnés aux différentes entités sont perdus lors de la compilation, tout ce qui reste est un ensemble d'adresses binaires.

1.4.2 Étapes de la compilation

Le procédé de compilation est découpé en plusieurs étapes. La première est **l'analyse lexicale** qui consiste en la découpe du code source en **lexèmes** compris par le compilateur. La suivante est **l'analyse syntaxique** qui consiste en la phase de regroupement des lexèmes en **instructions**. La dernière est **l'analyse sémantique** qui consiste en la *compréhension* des instructions, la levée des sens multiples, l'association des noms (du code source) aux adresses (du binaire), etc.

C'est lors de ces différentes phases que le compilateur peut sortir des messages d'erreurs. Les messages varieront en fonction de la phase à laquelle l'erreur est détectée. Ces trois phases se chargent d'une traduction littérale du code source, les optimisations se font après. Ces phases peuvent nécessiter plusieurs exécutions pour pouvoir traduire le code source, c'est pourquoi les compilateurs *transforment* en interne le code afin de le structurer de manière plus efficace pour le compilateur.

1.4.3 Langages interprétés

Tous les langages ne passent pas par ces différentes étapes. Certains programmes sont écrits dans des langages dits **interprétatifs**. Le principe est que ces programmes sont dépendants d'un programme (un **interpréteur**) car c'est ce dernier qui lit le code source (caractère par caractère) et qui l'interprète instruction par instruction une fois qu'il détient suffisamment d'informations. Par exemple, `while b:` ne s'exécutera pas directement, l'interpréteur attendra d'avoir le contenu de l'*indented block* avant d'exécuter tout le contenu de la boucle autant de fois que nécessaire.

Contrairement aux langages compilés, les langages interprétés font une exécution **synchrone** car le programme est analysé et directement exécuté alors que les langages compilés font une exécution **asynchrone** car ils sont compilés en code objet qui est transformé en exécutable et *stocké* dans un fichier en attendant une exécution.

On appelle **réflexion** la propriété de ces langages qui les rend **introspectifs** car ils sont capables d'examiner leur état courant et **intercessifs** car ils peuvent interagir avec eux-mêmes en se modifiant.

1.5 L'alphabet d'un langage

L'alphabet d'un langage est l'ensemble des caractères autorisés pour un code source. Cet alphabet contient tant les caractères visibles que les invisibles (ou de contrôle). Le plus souvent, les langages ont un alphabet basé sur l'alphabet latin restreint (sans caractère accentués, ligatures, etc.) augmenté par les caractères algébriques, les chiffres (arabes), des symboles de ponctuation et des blancs.

Certains langages ont la particularité de ne pas être **case-sensitive**. Il n'y a donc pas de distinction entre les caractères minuscules et majuscules. La plupart des langages actuels n'ont pas cette propriété, ce qui permet de faire une distinction syntaxique intelligente entre minuscules et majuscules. Identiquement, certains langages considèrent `NL`, le caractère de nouvelle ligne, comme un blanc quelconque alors que d'autres lui donnent un sens (tel que la fin d'instruction).

De plus, certains langages (dont `Python`) font une distinction entre plusieurs types de lignes. La **ligne physique** est une entité se finissant par `NL` alors que la **ligne logique** est une composition d'une ou plusieurs lignes physiques en une seule entité terminée par le lexème `NEWLINE`, propre à l'alphabet. L'**indentation** est, ici, le dernier niveau de séparation : il regroupe une ou plusieurs lignes logiques en **blocs**.

Les blancs également peuvent servir définir plusieurs familles de langages : certains langages donnent un sens aux blancs, d'autres les ignorent. Actuellement, les blancs servent surtout de **délimiteurs**. Un délimiteur est une séparation impérative entre deux lexèmes lorsqu'ils peuvent être considérés comme un seul (**int a ;**).

1.5.1 Les commentaires

Lorsqu'un programme est écrit, son développeur peut écrire des informations non destinées au binaire exécutable, donc des informations à ignorer lors de la compilation. Ces données sont appelées commentaires et doivent être marquées explicitement

comme tel à l'aide d'un lexème. (Attention, les docstring en `Python` sont plus proches des *meta-données* que des commentaires à proprement parler.)

1.5.2 Les séquences d'échappement

Lorsque l'alphabet d'un langage est limité, il y a moyen d'écrire certains caractères n'appartenant pas à l'alphabet ou certains caractères réservés en les *échappant*. Le principe est donc de placer le caractère `\` devant un autre caractère ou une séquence afin de l'échapper. Les plus connus sont le passage à la ligne `\n`, la tabulation `\t`, l'échappement hexadécimal `\x...`, etc.

1.6 Le vocabulaire d'un langage

Contrairement à l'alphabet qui est un ensemble fini totalement prédéfini, le vocabulaire est composé d'une base de mots mais est augmenté de mots introduits par le programmeur lors du développement (en suivant certaines syntaxes et conventions d'écriture). Ces ajouts sont les **identificateurs** et les **notations littérales**.

1.6.1 Le vocabulaire fixe

La sous-partie du vocabulaire qui est fixe est décomposé en les *symboles* et les *mots*. Les symboles sont soit des opérateurs (des méthodes ou fonctions particulières), soit des séparateurs (ponctuation). Les mots sont, quant à eux, composés de lettres (_ inclus) et de chiffres. Ils sont, eux aussi, réparties en deux catégories : les **mots-clefs** et les **mots prédéfinis**. Les mots-clefs ont un sens et un contexte d'utilisation très précis, ils ne peuvent être utilisés en dehors. Les mots prédéfinis, quant à eux, ont un sens précis dans un certain contexte, mais peuvent être utilisés de manière standard en dehors dudit contexte.

1.6.2 Identificateurs et notations littérales

Comme mentionné plus haut, le programmeur peut augmenter le vocabulaire du langage en déclarant lui-même des mots qui ont un sens dans le programme (nom de variable, de fonction, etc.). Ces mots supplémentaires sont appelés identificateurs. Une notation littérale par contre est un mot dont la syntaxe particulière décrit immédiatement la valeur, nécessairement constante, ainsi que le type.

1.6.3 bibliothèque standard

Les langages ont diminué leur nombre de mots-clefs en faisant passer des mots-clefs tels que `PRINT` dans une bibliothèque normalisée afin que l'utilisation des procédés décrits par ces mots-clefs puisse être faite avec un simple appel à fonction ou à méthode. La bibliothèque standard d'un langage est donc un ensemble (*kit*) de types, classes, fonctions disponibles automatique par défaut ou par référence (include, import, etc.) Ces fonctions servent à produire un comportement normalisé des opérations complexes mais fréquentes qui nécessitent un nombre important d'instructions élémentaires. (Le code de ces bibliothèques est lié dans l'environnement d'exécution.) Les mises à jour des langages concernent donc plus l'augmentation et l'optimisation de la bibliothèque standard plutôt que le changement de la syntaxe ou de la sémantique.

1.7 La syntaxe d'un langage

La **syntaxe** d'un langage définit la manière de regrouper les lexèmes en instructions ou directives (messages au compilateur). Ces instructions se combinent ensuite, donnant une structure d'**imbrications syntaxiques**, formalisée en une structure hiérarchique sous forme d'arbre (*arbre syntaxique*).

Les instructions sont subdivisées en deux catégories : celles induisant une sous-structure (que l'on appelle **instructions de contrôle**), et celles n'en produisant pas (que l'on appelle **instructions simples**).

1.7.1 Bloc et ordre d'évaluation

Pour des soucis de logique et de simplicité, il est plus agréable de ne permettre à une instruction de contrôle de n'avoir qu'une seule sous-entité. Cette entité s'appelle **bloc**. Un bloc est donc une séquence d'instructions (simples et de contrôle). Ces instructions sont exécutées dans l'ordre de déclaration.

Une **expression** est instruction particulière qui peut être évaluée. Lors d'une telle évaluation, il existe certaines règles de priorité des opérations (pas uniquement les opérations arithmétiques mais également les opérations logiques, les conversions, etc.) qui sont définies de manière univoque par la syntaxe du langage. Attention cependant aux opérateurs évalués de gauche à droite et ceux évalués de droite à gauche.

1.8 La sémantique d'un langage

La sémantique du langage est la couche au-dessus du sens immédiat que l'on peut associer à une instruction. Une fois que le compilateur a fait son arbre syntaxique, il lui faut encore associer les noms aux fonctions, variables, etc ; correspondantes, il lui faut déterminer le type des opérations, etc. Cette analyse peut être décrite par une grammaire rigoureuse (plusieurs normes existent pour cela), mais la partie sémantique est habituellement écrite, dans les normes, en *anglais jargonnant* donc en anglais avec des terminologies précises pour certains mots (*must, shall, etc.*), contrairement donc au vocabulaire et à la syntaxe qui sont définis de manière formelle. Cela laisse place à une grande part d'interprétation dans le sens d'un morceau de code.

Chapitre 2

Typage et types élémentaires

2.1 Un type

Un **type** de donnée est une notion importante pour la réalisation d'algorithmes et donc de programmes. Un type est défini par un ensemble fini de valeurs que peut prendre une donnée, un nombre d'opérations finies que l'on peut lui appliquer, et un norme de codage sur un certain nombre de bits. Le nombre de bits nécessaires pour coder la donnée peut être fixé pour le type, variable au sein même du programme, ou variable en fonction de l'implémentation.

2.2 Typage statique ou dynamique

Une caractéristique très importante d'un langage est son typage : statique ou dynamique, et fort ou faible (ci-dessous). Un langage est typé **statiquement** si la vérification de la faisabilité des opérations et les conversions implicites sont faites au moment de la compilation (et sont donc figés), et il est typé **dynamiquement** si ces mêmes vérifications sont faites pendant l'exécution.

Pour faire ces vérifications durant l'exécution, il faut garder une information de type dans l'objet en question. Un langage typé statiquement doit pouvoir déterminer de manière non-ambiguë le type d'un objet sur seule base du texte du programme. On utilise pour cela des **déclarations** qui sont des instructions ayant pour objectif d'associer une entité avec son type, donc l'ensemble des valeurs qui lui sont accessibles (attention, il existe des langages typés statiquement mais n'ayant pas de déclarations : le type est défini par la première utilisation de la donnée).

La plupart des langages interprétatifs sont typés dynamiquement pour des raisons évidentes alors que la plupart des langages compilés sont typés statiquement pour des raisons d'optimisation

2.3 Typage fort ou faible

On appelle *langage à typage fort* un langage qui propose une robustesse et une sécurité importante sur les manipulations de type : un typage strict avec une vérifications riche pour la cohérence des opérations appliquées sur un type. *A contrario*, les langages à typage faible n'associent à un type que certaines opérations de base et surtout une taille. Ils laissent, de plus, bien souvent un accès libre à tout octet de la mémoire.

Les langages `C` et `C++` sont proches des langages de bas niveau (d'assemblage) et c'est voulu. Dans ces derniers langages, les données ne sont qu'un nombre de bits, il n'y a pas de réelle notion de type (les données sont interprétées différemment et se voient appliquer des opérations différentes, mais tout n'est qu'implicite). `C` offre donc la possibilité de convertir explicitement (*cast*) presque tous les types de données entre eux et de manipuler de l'arithmétique de pointeurs très libre et peu contraignante.

2.4 Polymorphisme et métaphore du canard

Le polymorphisme est la caractéristique de beaucoup de langages *orientés objets* qui leur permet d'avoir des informations sur le typage gardées pendant le *runtime* afin de lever les ambiguïtés sur les méthodes à appeler mais donc d'appliquer la même fonction sur plusieurs types de données différents. Ce polymorphisme amène à deux types de langages : les langages à typage **de classe** car un type est lié à une classe *statiquement définie* et les langages à typage **de canard** car le type est propre à chaque objet et est évolutif (*Quand je vois un oiseau qui marche comme un canard, nage comme un canard, caquette comme un canard, j'appelle cet oiseau un canard*).

2.5 Déclaration, définition et initialisation

Un vocable assez précis a été mis au point concernant la *création* d'*entités* [1]. Lorsque l'on associe un identificateur à un type ayant un ensemble de propriétés précises, on parle de **déclaration**. Lorsque l'on fait une déclaration qui crée une entité de ce type, on parle de **définition**. Et lorsque qu'une définition associe à l'identificateur une valeur, on parle d'**initialisation**.

Dans les langages tels que `C` et `C++` , ils est impératif de déclarer tout objet avant son utilisation et de ne le déclarer qu'**une et une seule fois**, on parle de la loi d'*ODR* (One Definition Rule). Il faut cependant savoir que pour toute variable ou tout objet, une déclaration est une définition (mais la déclaration d'un type n'est pas une définition). Lors d'une définition sans initialisation, la variable ou l'objet prend la valeur par défaut (donc la case mémoire inchangée pour les types primitifs et le constructeur par défaut pour les objets complexes).

2.6 L-value, R-value, X-value, GL-value et PR-value

Lors des évaluations, les opérations se font dans le processeur, et une partie des résultats n'en sortent jamais (vers la mémoire) et donc restent dans les registres, sans adresse mémoire, sans identifiant. Les compilateurs créent alors des **objets temporaires** (souvent sur la pile) pour pouvoir y accéder. Ces objets sont gérés de manière *explicite* dans le code binaire, mais sont instaurés de manière *implicite* par le compilateur.

Afin de minimiser le nombre de ces *valeurs fantômes* sur la pile, les compilateurs font un travail de prédiction de pré-évaluation tant que possible afin de remplacer toutes les opérations qui donneront le même résultat, peu importe l'exécution. Ces évaluation deviennent alors des **valeurs immédiates** et sont codées directement dans le binaire dans l'opération associée. Les langages font cependant une distinction entre les zones mémoires créées par le programmeur pour y stocker ses informations (qu'il peut manipuler comme bon lui semble), et les zones créées implicitement par le compilateur qui contiennent un résultat temporaire. La différence entre les deux est propre au langage (défini dans la norme) et est entièrement sémantique.

On parle de **L-value** lorsque l'objet persiste au-delà de l'expression dans laquelle il est utilisé. Une adresse mémoire peut donc lui être associé, et il pourrait *a priori* [2] être une opérande gauche d'une assignation (le *L* vient de ***L**eft-side value*). Lorsque, au contraire, l'objet ne persiste pas au-delà de l'expression dans laquelle elle naît, la valeur est temporaire et pourrait être une opérande droite d'une assignation. On parle alors de **R-value** (le *R* vient de ***R**ight-side value*).

Seules ces deux distinctions étaient définies dans la norme `C++` avant la version `C++` 11. Mais depuis `C++` 11, trois nouvelles distinctions se sont faites : une **X-value** est une valeur (plus précisément une R-value) en fin de vie (*scope*) qui peut être réutilisée anticipativement (le *X* vient de *e**X**piring value*. Une **PR-value** est l'opposé d'une X-value : c'est une R-value qui n'est pas encore en état temporaire de fin de vie (avant de disparaitre) (le *PR* vient de ***P**ure **R**ight value*). Il reste donc une **GL-value**, qui est soit une L-value, soit une X-value. Une GL-value généralise donc toute expression dans laquelle on peut placer une valeur (Le *GL* vient de ***G**eneralized **L**eft value*).

1. Termes vagues intentionnellement choisis.
2. Sauf cas de `const` par exemple.

Chapitre 3

Les types composés

Par opposition aux types **élémentaires** [3], il existe des types dits **composés** qui sont de nouveaux types définis par les programmeurs sur base de combinaisons de types élémentaires ou d'autres types composés. Les types composant les types composés sont appelés ses **types de base**. On regroupe les types composés en cinq grandes familles :

1. les **pointeurs** et **références** : accès *indirect* à un type de base ;
2. les **énumérations** : sous-type d'entier constitué de noms de *constantes* ;
3. les **tableau** : collection **homogène** et **contiguë** en mémoire indicée ;
4. les **fonctions** : une valeur d'un certain type à produire sur base de certains paramètres de type(s) défini(s) ;
5. les **classes** : collection **hétérogène** identifiées par un identificateur. On distingue plusieurs sortes de classes :
   (a) les **enregistrements** : collection **hétérogène** et **contiguë** en mémoire ;
   (b) les **unions** : une zone mémoire définie interprétée comme plusieurs types différents ;
   (c) les **bit-fields** : rassemblement compact de quelques bits ;
   (d) les **n-uplets** : collection **hétérogène** indicée ;
   (e) les **classes à méthodes** : type composé de données (**attributs**) et également de fonctions (**méthodes**) agissant sur ledit type.

En C++ , les types élémentaires, les pointeurs, références et les énumérations sont appelés **types scalaires** car ils peuvent tenir dans un ou plusieurs registres et être traités comme des types élémentaires. De plus, on appelle **POD** (pour ***P**lain **O**ld **D**ata*) les types composés composés d'une collection contiguë en mémoire de dimension statique définie lors de la compilation. Les types scalaires en C++ sont donc des POD.

Pour faire une classe POD en C++ , il lui faut les propriétés suivantes : être **struct** ou **union**, être **triviale** [4], et disposer d'une **représentation standard** [5]. Les POD peuvent donc permettre d'importantes améliorations et optimisations par leur nature (par opposition aux types éclatés, ou *dynamiques*).

3.1 Pointeurs et références

Les pointeurs et les références permettent un accès indirect vers une donnée stockée en mémoire. Leur fonctionnement précis dépend du modèle de gestion de la mémoire du langage. Dans la plupart des langages, ce modèle est celui d'une architecture de von Neumann : la **mémoire** est un ensemble d'octets identifiés par une **adresse** de manière **bijective** et telle que lorsqu'un objet nécessite plusieurs octets, son adresse mémoire soit celle de l'octet d'*adresse minimale*.

Remarque Dans les langages de haut niveau, les accès directs en mémoire sont limités à certaines opérations sur certains types particuliers (pointeurs, références, etc.) alors qu'en langage d'assemblage, la gestion de la mémoire est au centre du code à produire.

3. À savoir les types primitifs proposés par un langage sur lesquels les opérations disponibles sont celles propres au processeur.
4. Construite et initialisée statiquement dès sa définition et détruite ensuite.
5. Peut être considérée comme une zone mémoire de dimension finie copiable de manière *brutale*.

3.1.1 Gestion de la mémoire

Les architectures physiques actuelles sont plus efficaces pour transférer des données de plusieurs octets lorsque leur adresse est un multiple d'une certaine valeur (puissance de deux). On appelle cela la granularité de l'adressage. L'**alignement** est le fait de placer les données à des multiples d'une puissance de deux (4 par exemple) pour des raisons d'optimisation, quitte à rajouter des octets de **padding** entre les données. Cette gestion de l'alignement est faite par le compilateur, à moins que le programmeur n'explicite le positionnement de ses données (avec des opérateurs tels que `alignas` en `C++`).

Toutes les données ont une **taille** (habituellement exprimée en octet car c'est l'unité atomique de l'adressage mémoire). Lorsque des données ont une taille réelle inférieure à 1 (bit-fields par exemple), elles sont regroupées et paddées afin d'être adressées correctement par le compilateur. La taille effective d'un objet en `C` /`C++` peut être obtenue avec l'opérateur `sizeof`.

Il existe cependant un problème lors de l'adressage d'une donnée stockée sur plusieurs octets : l'octet donné par l'adresse est-il l'octet contenant le bit de poids fort ou celui de poids faible ? On parle alors de **boutisme** qui est l'ordre de placement des octets dans une donnée en nécessitant plusieurs. On parle de **gros-boutiste** (big-endian) pour une architecture plaçant l'octet contenant le bit de poids fort à l'adresse la plus petite, et on parle de **petit-boutiste** (little-endian) pour une architecture plaçant l'octet contenant le bit de poids faible à l'adresse de l'objet. En big-endian, les octets sont donc stockés par poids décroissant alors qu'ils le sont par poids croissant en little-endian.

3.1.2 Pointeurs

Un **pointeur** est un type composé d'objet permettant d'accéder à un objet *pointé* depuis la valeur du pointeur. L'accès à la donnée pointée est appelée **déréférencement**, et la localisation d'une information en mémoire est appelée **prise d'adresse**. Dans le cas d'un pointeur, ces opérations son **explicites**. Un pointeur est considéré comme typé s'il ne peut pointer qu'un type de données bien défini.

Un **point nul** est un pointeur (dit également **nul**) qui ne pointe vers aucune donnée : sa valeur est une constante propre au langage définissant un pointeur nul (`nullptr` en `C++`). Un pointeur nul ne peut être déréférencé sans lancer une erreur. Et peu importe le type de base du pointeur, la taille du pointeur est constante sur un architecture.

3.1.2.1 Arithmétique sur pointeurs

Les pointeurs étant des variables comme les autres (bien qu'elles n'aient pas le même sens, elles ont la même représentation), il est possible d'y appliquer des opérations arithmétiques de base ayant un sens particulier [6]. En `C++` un incrément ou un décrément sur un pointeur ajoute (ou enlève) **la taille du type pointé** à la valeur du pointeur. Ces opérations permettent donc de se déplacer dans un tableau. L'addition ou la soustraction d'une valeur positive correspond à autant d'incréments ou décréments successifs. La différence entre deux pointeurs est de type `std::ptrdiff_t` (défini dans `cstddef`) et correspond au **nombre (entier) d'éléments qui sépareraient ces deux pointeurs dans un tableau**. Les opérations arithmétiques disponibles sur les pointeurs s'arrêtent là. Il n'y a pas moyen de multiplier des pointeurs, de les multiplier, ou autre car cela n'a pas de sens direct si ce n'est jouer **dangereusement** avec les adresses mémoires et risquer un *SEGFAULT* [7]. Des pointeurs peuvent cependant (tant qu'ils sont de même type) être comparés à l'aide des opérateurs d'égalités et d'inégalités. Le sens de `p < q` est le même que `0 < q-p` ou encore *est-ce que le nombre de cases entre* `q` *et* `p` *est positif ?*. Si oui, c'est que l'adresse contenue dans `q` est plus élevée que celle contenue dans `p`.

Les langages comme `C++` n'offrent aucune vérification de l'intégrité du pointeur, c'est entièrement au programmeur d'y faire attention.

3.1.2.2 Pointeurs sur fonctions

Un pointeur n'étant autre qu'une variable entière contenant une adresse, il peut contenir absolument n'importe quelle adresse. Même l'adresse d'une fonction [8]. L'adresse contenue dans le pointeur est l'adresse de la première instruction de la fonction. Ces pointeurs sont manipulés comme des pointeurs habituels (avec *récupération d'adresse* et *déréférencement*) sauf que leur type est **pointeur sur fonction**. De plus, il faut préciser le type de retour et le type des paramètres dans le type du pointeur. Pour Voici un exemple de déclaration et d'utilisation de pointeurs sur fonctions en `C++` :

```
#include <iostream>

void f1(int t) {std::cout << t << std::endl;}
void (*p1)(int) = f1;
int *f2(int *t, int i) {return &t[i];}
int *(*p2)(int *, int) = f2;
```

6. La priorité des opérations reste identique, même lorsque le type manipulé est un pointeur.
7. Ou assimilé sur les machines non segmentées.
8. Ou méthode ou procédure.

```
int main(void)
{
    int tab[3] = {10, 20, 30};
    (*p1)(5);
    std::cout << *(*p2)(tab, 1) << std::endl;
    return 0;
}
```

3.1.2.3 Allocation dynamique

Les pointeurs ont une autre utilité (des plus fréquentes) : la manipulation d'adresses mémoires **alloués dynamiquement**, ou encore non-alloués statiquement. Ce sont donc des zones mémoires requises par le programme après le début de son exécution. Cette allocation nécessite une syntaxe particulière (`new/delete`, `malloc()/free()`, ...) et dépendante des langages.

En fonction des langages, la gestion de désallocation (ou de libération) de la mémoire ne se fait pas du tout de la même manière. Certains langages implémentent un **garbage collector** dans le runtime, mais d'autres imposent au programmeur de gérer seul la mémoire qu'il désire (dés)allouer sur le tas. Par exemple, en `C` , l'allocation se fait par la fonction `std::malloc` et la désallocation par la fonction `free` et en `C++` , elle se fait par les opérateurs `new` et `delete`. Comme aucune récupération automatique de la mémoire allouée n'est gérée dans ces deux langages, il y a une **pseudo-règle** qui dit que *toute allocation de mémoire doit avoir une désallocation associée*, que ce soit un `free()` en `C` ou un `delete` en `C++` [9].

D'autres langages, tels que `Python` ou `Java` gèrent un **garbage collecteur** (ou ramasse-miettes) qui est composé d'un algorithme très complexe qui est appelé soit explicitement soit à intervalle régulier soit lorsque le runtime opère un appel explicite automatique lié à un manque de mémoire effective disponible. Un tel outil permet de se charger des désallocation lorsque c'est possible afin que le programmeur n'ait pas à en tenir compte.

3.1.3 Références

Les références sont comparables aux pointeurs car ce sont des types qui donnent un accès indirect à une variable, cependant les références sont différentes de ces derniers. Quand on fait usage d'une référence, il n'y a pas moyen d'accéder à sa valeur propre car l'usage d'une référence implique un **déréférencement automatique**. Une référence est donc utilisée comme une *simple variable*. De plus, une référence est **liée** à une **entité particulière** dès sa déclaration [10] et n'est donc pas réassignable. Une référence est *assimilable* à un **pointeur constant**. Cependant, contrairement à un pointeur (constant ou pas), la référence n'existe que dans le code source du programme, et pas dans l'exécutable produit. Une référence est un alias. Donc lors de la phase de compilation (pour les langages compilés), les références sont changées par l'objet référencé. Cette propriété fait qu'il n'est donc pas possible de créer un objet persistant (pointeur/tableau) [11] sur une référence car dans l'exécutable, ce type ne correspondrait à rien. Une référence vers une fonction est toutefois sémantiquement correct.

3.1.3.1 Référence vers [L/R]-value

Afin d'être totalement rigoureux, il faut préciser qu'un tel type de référence (en `C++`) est une référence sur une L-value car elle permet de modifier la valeur référencée. Il peut cependant arriver que le programmeur désire faire une référence sur une R-value en quel cas ce type de référence ne peut être utilisé car il permettrait de modifier une R-value, ce qui est sémantiquement incorrect de par la définition d'une R-value. Il faut alors définir la référence avec le déclarateur `&&` et plus uniquement `&`. Cela peut être utile dans le cas d'un cast d'une L-value (une référence simple par exemple) en une R-value. Il existe cependant la fonction standard `std::move` en `C++` qui permet de faire cette même chose et qui est conseillée dans ce cas.

3.1.4 Copies et transferts

Lors de manipulations de références et de pointeurs, une copie peut avoir deux sens : une **copie superficielle** (ou **shallow-copy**) ou une copie **profonde** (ou **deep-copy**). Le premier est une copie uniquement de l'adresse ou de la référence. Il y a donc présence de deux objets pointant ou référençant le même objet. La modification de l'objet pointé (ou référencé) par l'un entraine donc une

9. Attention, la désallocation agit sur la **valeur** du pointeur, donc sur l'adresse qu'il pointe. Mais sa valeur à lui n'est pas modifiée. Il est donc **fortement recommandé** au programmeur d'assigner une valeur nulle à un pointeur lors de sa libération afin de savoir quels pointeurs sont effectifs et lesquels sont obsolètes.

10. Certains langages permettent des références nulles.

11. **Note** : lors d'une initialisation, il est tout à fait autorisé d'utiliser une référence pour la valeur d'un pointeur car `int ref = x, *pointeur = ref;` est équivalent à `int ref = x, *pointeur; *pointeur = ref;` où donc le pointeur est d'abord déréférencé. Nous n'avons pas affaire à un pointeur sur référence.

modification de l'objet pointé (ou référencé) par le second étant donné que c'est le même objet. Le second est une copie complète de l'objet, donc implique la création (dynamique donc sur le tas) d'un nouvel objet de même type qui est initialisé à la même valeur que l'objet copié [12]. Dans ce second cas, la modification de l'un n'implique nullement une modification de l'autre.

3.2 Les types hétérogènes

3.2.1 Les enregistrements

Les **enregistrements** sont des **agrégats** (donc contigus en mémoire) de types différents. On les appelle également **structures** ou **compound data**. En `C++` , pour déclarer une structure, il faut utiliser le mot-clef `struct` suivi d'accolades contenant les déclarations des variables (type + identificateur). Les variables sont placées de manière contiguë en mémoire dans l'ordre de déclaration [13]. Les variables de la structure sont appelées **attributs** et sont accessibles via l'opérateur `.` [14] et sont des variables *simples* (L-values) donc utilisées comme telles. Il est possible de copier toute une structure dans une autre (shallow-copy) à l'aide de l'opérateur d'affectation [15].

L'initialisation d'une telle structure peut se faire soit par défaut (dans la déclaration du type), soit lors de la déclaration soit indépendamment pour chaque variable après la déclaration. Si une initialisation n'est pas complète, les attributs non précisés sont mis à la valeur nulle [16] par défaut. Si une telle structure contient elle-même une structure, l'initialisation peut se faire à l'aide d'accolades imbriquées :

```
struct {int i; struct {char c1, c2;}; double d;} s = {15, {'R', 'P'}, .5};
```

3.2.1.1 Pointeurs sur structures

Un pointeur étant une simple adresse mémoire peut très bien pointer sur une structure (qui est un type comme un autre). Pour accéder à un attribut, il faut donc d'abord déréférencer le pointeur, puis accéder au membre puis y faire l'opération souhaitée (soient une structure et un pointeur sur cette structure déclarés comme suit : `strct {...} s1 = {...}, *p = &s;`, l'accès à un paramètre `v` de `s` par `p` se fait comme suit : `(*p).v`). Il y a cependant une autre syntaxe qui déréférence automatiquement le pointeur. C'est l'opérateur `->` qui est utilisé à la place de l'opérateur **dot**. L'accès à un attribut `v` se fait donc de la sorte : `p->v`.

Remarque Avoir un attribut qui est une structure ou un pointeur sur structure n'est pas du tout pareil. Dans le premier cas, on parle de **relation de composition** [17] alors que dans le second, on parle de **relation d'agrégation** [18].

3.2.1.2 Structure nommée

Afin de simplifier le code et d'améliorer sa lisibilité, il y a moyen de déclarer la structure comme un type à part entière et de déclarer des variables de ce type partout dans le code. On parle alors de **structure nommée** par opposition aux **structures non-nommées** vues précédemment. La déclaration se fait exactement de la même manière si ce n'est que l'identificateur qui sert de nom au type doit être placé entre le mot-clef `struct` et l'accolade ouvrante.

Remarque Si on veut donner à une variable le même identificateur qu'à la structure, il faut faire précéder le nom de la structure de `struct`. Cette pratique est peu recommandée car elle est une exception à l'*ODR*. De plus, elle n'aide pas à la compréhensible facile du code.

3.2.1.3 Pointeur sur membre

Il est possible d'avoir un pointeur sur un certain attribut d'une structure. Il faut, pour le déclarer, préciser le type pointé mais également le contexte dans lequel il est utilisé (à savoir la structure dans laquelle il agit) :

12. *La même* valeur est un singulier, mais on peut considérer l'ensemble des valeurs d'un type composé comme une seule valeur qui est celle de l'objet de ce type.
13. Attention tout de même au padding (§3.1.1.) qui peut les placer de manière contiguë mais en laissant des bytes **fantômes**.
14. L'opérateur **dot** (ou *point*).
15. La copie n'est possible en comportement par défaut géré par le compilateur que de par la propriété de contiguïté des attributs en mémoire.
16. 0 pour les types élémentaires ou constructeurs par défaut pour les classes.
17. Ou **relation-*owns-a***.
18. Ou **relation-*has-a***.

```
struct struct_t {int i;};
int main(void)
{
    int struct_t::*pi = &struct_t::i;
    struct_t s = {42}; //s.*pi == 42
    struct_t t = {-5}; //t.*pi == -5
    return 0;
}
```

Dans le code ci-dessus, une structure est déclarée, et un pointeur sur son attribut est déclaré. Le principe d'un tel pointeur est qu'il **ne pointe pas sur une variable précise**, mais il pointe sur un **offset** par rapport au début de la structure afin qu'il puisse être utilisé sur n'importe quelle instance de cette structure. Les opérateurs `.` et `->` ne peuvent être employés sur un tel pointeur. À la place, il faut utiliser les opérateurs `.*` et `->*` qui sont utilisés exactement dans le même contexte que leurs homologues.

3.2.2 Les unions

Contrairement à un enregistrement qui est une suite contiguë de variables en mémoire, une **union** est une seule zone mémoire qui peut être interprétée selon plusieurs types différents. Pour déclarer une unions en `C` /`C++` , il faut utiliser le mot-clef `union` et mettre entre accolades les différents types pouvant interpréter la variable. Par exemple, voici un exemple de déclaration d'union :

```
union U {double d; char c;}
```

Une itération de cette union ne contient qu'**une seule valeur** mais pouvant être interprétée soit comme un double, soit comme un char. La taille d'un `double` étant (habituellement) 8 octets et celle d'un `char` étant 1, il faut bien sûr définir l'union telle que `sizeof(u) == sizeof(double)` où `u` est une instance de cette union. Effectivement, pour pouvoir interpréter la valeur comme tout type de l'union, il faut que le type nécessitant le plus de place puisse y être stocké.

Une **union anonyme** est une déclaration d'union non-étiquetée. Dans ce cas, les *variables* sont déclarées directement dans le *scope* de l'union, et sont donc accessibles mais à la même adresse. Cela correspond à une instance d'union déclarée directement et ne devant pas être référencée.

Remarque Attention : les unions ne peuvent contenir que des types primitifs [19].

3.2.2.1 Les bitfields

Les **champs de bits** (*bitfields*) sont une forme particulière de structures ou d'unions dans lesquels il est possible de choisir la taille allouée à chaque attribut. La notation est de faire suivre l'identificateur de la variable de `:` puis du nombre de bits (et pas d'octets !) qui doivent lui être alloués. Si un des éléments n'est pas une variable avec identificateur (`type:taille;`), il s'agit uniquement de *padding*. Il est également de prévoir cette syntaxe du padding en inscrivant une taille nulle. L'effet est d'aligner le prochain champ à un début d'octet (s'assurer qu'il n'y a pas de début sur un bit n'étant pas le bit 0 d'un octet).

3.3 Les énumérations

Une **énumération** est un type composé de noms de constantes représentant un *subrange* d'un type de base. En `C++` , une énumération se déclare comme une structure ou comme une union [20]. Par défaut, la taille d'un élément de ce type est la même que la taille d'un entier, mais il est possible de spécifier le type représentant ce *subrange* à l'aide de la notation `enum Nom:Type {...}`.

3.3.1 Énumération à région déclarative

Dans une énumération *classique*, les identificateurs (qui sont des noms de constante) ont un impact sur tout le *scope* de leur déclaration. Il est donc impossible de nommer une variable comme une des constantes, ce qui créerait une erreur lors de la compilation. Pour palier à cela, il est possible de créer des **énumérations avec région déclarative** de la sorte : `enum struct Nom:Type {C1, C2, C3, ...}`. Afin d'accéder à ces constantes, il faut utiliser l'**opérateur de résolution de portée** (`::`).

19. Ou des structures ou d'autres unions, mais pas une classe nécessitant un constructeur, destructeur, etc.
20. Et peut également être nommée ou non.

3.4 Tableaux

Un **tableau** est une zone mémoire allouée pour un nombre **fini** d'éléments **contigus** et **homogènes**. La référence à un de ces éléments se fait par indiçage. L'accès mémoire de cet élément est un calcul d'adresse de base du tableau décalé d'un *offset* calculé sur base de l'indice. La taille du tableau (tout comme la taille d'une variable) est le nombre d'indices accessibles (**et non le nombre d'octets utilisés !**).

Le type du tableau est le type des éléments qu'il contient. Ce type peut donc être un des types fondamentaux, une union, une structure, etc. mais peut également être un tableau. Un tableau de tableaux est un **tableau multidimensionnel**. Dans le cas d'un **tableau unidimensionnel**, on parle de **vecteur**. Certaines architectures comportent un **processeur vectoriel** qui peut exécuter des opérations sur tout un vecteur en une seule instruction. Certains langages s'en sont fortement enrichis et rendu dépendant.

3.4.1 Tableaux statiques

En `C++` , pour déclarer un vecteur (ou en réalité un tableau quel que soit son type), il faut utiliser la syntaxe suivante : `Type Nom[Taille];`. Cependant, un tableau étant une zone contiguë en mémoire, il est très simple de réaliser une initialisation d'un tableau : `Type Nom[Taille] = {N1, N2, N3, ...};` où la taille peut être omise en quel cas elle sera interprétée par le compilateur comme le nombre d'éléments fournis entre les accolades. Cependant, si `Taille` est fourni et vaut n alors que seuls m valeurs sont données (avec $m < n$), les $n - m$ valeurs suivantes sont initialisées à la valeur nulle.

Remarque Lorsqu'un tableau est passé en paramètre à une fonction, il est impossible de connaître sa taille (sauf si elle est défini comme une constante globale disponible dans ce scope du code source ou si une valeur particulière est utilisée pour codifier la fin de tableau). Il faut donc la préciser en paramètre si elle veut être utilisée. Ce paramètre **doit impérativement** être de type `size_t` [21] afin d'éviter des problèmes de compilation lors de surcharges de certaines fonctions.

3.4.2 Tableaux dynamiques

Un tableau peut également être *créé* lors de l'exécution du programme. Les opérateurs `new` et `delete` ont été vus plus haut pour allouer un espace mémoire, il existe un homologue pour allouer des tableaux. Ce sont les opérateurs `new[]` et `delete[]`. Ils s'utilisent de la même manière si ce n'est que pour l'allocation, le nombre d'éléments doit être précisé entre crochets (ex : `int *tab = new int[15];`). De plus, `new[]` renvoie un pointeur sur le type du tableau. Mais il y a conversion implicite entre vecteurs et pointeurs. Concernant l'opérateur de désallocation, tout est exactement similaire à son homologue.

Remarque L'indiçage (opérateur `[]`) est totalement **commutatif** ! Donc soient `idx`, un entier et `tab`, un tableau, il faut savoir que `tab[idx] == idx[tab] == *((tab)+(idx))`. La règle de commutativité découle de celle de l'addition en laquelle est transformée tout indiçage. Cependant, ces notations ont beau être sémantiquement correctes, il est **plus que recommandé** d'utiliser la notation `tab[idx]`.

Remarque (bis) Malgré cela, un vecteur **n'est pas** un pointeur. Il y a conversion, certes implicite, mais conversion tout de même ! Cette conversion est normalisée par le langage et n'est pas toujours possible. De plus, *tableau* est un type à part entière, donc il est possible de faire un *pointeur sur tableau* avec son sens arithmétique logique, à savoir que `int (*ptr_sur_tab)[5] = &tab;` crée un pointeur sur tableau et que `ptr_sur_tab[1]` est équivalent à `*((ptr_sur_tab)+(1))` mais où cet incrément est de `sizeof(tab)` [22]. Donc `ptr_sur_tab[1]` pointe vers une zone mémoire possiblement *illégale* (représente le tableau de 5 entiers suivant `tab` en mémoire).

3.4.3 Slicing

Bien qu'en `C++` , un slicing comme en `Python` n'est pas possible, un équivalent peut être fait par pointeur (qui donc prend comme adresse de base le nième élément d'un tableau ce qui permet de commencer un indiçage à cet endroit là) voire en plus propre par une référence et un cast explicite : `int (&ref_slicing)[5] = *reinterpret_cast<int (*)[5]>(&tab);` où `tab` est un tableau de 5 entiers.

21. Qui est un entier non signé.
22. Voir §3.1.2.1.

3.4.4 Tableaux multidimensionnels

Autant un **vecteur** est un tableau unidimensionnel, autant un tableau bidimensionnel est appelé **matrice**. La notion de *dimension* est cependant totalement humaine et n'existe plus du tout dans l'exécutable binaire. Lorsqu'un tableau multidimensionnel est déclaré, il occupe **un et un seul** bloc mémoire contigu. Il existe toutefois deux conventions pour l'écriture en mémoire : la convention **colonne par colonne** et la convention **ligne par ligne**. Dans le premier cas, l'élément $M_{i,j}$ donc i^{e} ligne et j^{e} colonne est stockée à l'adresse $M + i + nb\_lignes \cdot j$. Le second quant à lui place ce même élément $M_{i,j}$ à l'adresse $M + i \cdot nb\_colonnes + j$. Dans les deux cas, M est l'adresse de début du tableau, i et j commencent à 0 et nb_lignes et $nb\_colonnes$ sont respectivement le nombre de lignes de de colonnes de la matrice. La première convention est moins utilisée de nos jours (mais l'est toujours pour certains langages tels que `FORTRAN`) alors que la seconde est la plus utilisée (vient du `C` /`C++`).

3.4.4.1 Conception récursive

La convention *ligne par ligne* en `C` /`C++` vient de la conception d'un tableau de dimension n comme un simple vecteur où chaque élément est un tableau de dimension $n - 1$. Cette conception peut amener à des allocation dynamiques faciles : un pointeur sur pointeur sur `type` peut se voir allouer une zone mémoire pour y stocker les adresses de chaque tableau. Par exemple :

```
int **p;
p = new int *[N];
for(int i = 0; i < N; ++i)
    p[i] = new int[M]
```

Ce code permet de créer dynamiquement une matrice de dimensions $N \times M$. Pour désallouer cette-même matrice, il faut procéder de la même manière, avec une boucle pour d'abord libérer `p[i]` et puis il faut libérer `p`.

Remarque Il faut cependant faire attention avec une conception récursive et une allocation dynamique de la sorte car **rien ne garantit au programmeur que tous les tableaux de dimension** $n - 1$ **sont de même taille** !

Remarque (bis) Lorsque nous avons affaire à un tableau de dimension supérieure à 1, il faut bien comprendre le sens de l'opérateur `[]`. Il est opéré de gauche à droite pour chaque indiçage.
Donc `tab[i][j][k] == (*(tab+i))[j][k] == (*(*(tab+i)+j))[k] == *(*(*(tab+i)+j)+k)`.

3.4.5 Les chaines de caractères

Un vecteur ayant pour type de base le type caractère (`char` en `C` /`C++`) est une **chaine de caractères** ou encore un(e) `string`. Ce n'est en réalité qu'un vecteur comme tant d'autres, mais beaucoup de langages implémentent soit une syntaxe particulière soit directement un type particulier pour simplifier la vie au développeurs. Il existe trois *familles* de strings : les **fixed-length strings** qui sont, comme leur nom l'indique de longueur définie et fixée (un vecteur de taille n), les **length-encoded strings** ou **length-prefixed strings** qui ont une taille définie aussi mais variable ; cette longueur est codée sur le premier octet de la chaine (ce qui limite les chaines à 255 caractères), et enfin les **null-terminated strings** ou **ASCIIZ-strings** qui sont des suites de caractères [23] terminées par un caractère nul (de code ASCII ou unicode 0) ce qui permet de savoir où la chaine s'arrête.

En `C` , c'est la dernière famille qui représente des chaines de caractères, en `C++` , des chaines standards sont également définies comme cela, mais la bibliothèque standard implémente un type plus générique (`std::string`) qui est basé sur la seconde famille.

3.4.5.1 chaines littérales préfixées en `C++`

Une chaine définie entre guillemets directement dans le code source est appelée **chaine littérale**. Il existe cependant plusieurs formats pour stocker ces chaines dans le binaire final. Si rien n'est précisé, la chaine sera stockée comme elle l'est dans le fichier source. Il est cependant possible de préciser au compilateur le format désiré. Pour cela, il faut *préfixer* la chaine littérale par : soit **un "u" minuscule** pour forcer le type `const char16_t[]`, soit un **un "U" majuscule** pour forcer le type `const char32_t[]`, soit **un "L" majuscule** pour forcer le type `const wchar_t[]` [24]. Ces trois préfixes forcent le type représentant à chaque fois un caractère (un élément de la chaine) mais la valeur mise à l'intérieur dépend toujours de la codification du fichier source. Il existe un quatrième préfixe, **un "u" minuscule suivi du chiffre "8"** qui, quant à lui, ne force pas le type (donc qui reste `char`) mais force la codification en UTF-8. La manière de représenter la chaine dans le binaire ne dépend donc plus de la codification du fichier de code source.

23. Attention tout de même car les chaines peuvent être soit **homogènes** donc avec tous les caractères de taille équivalente (ASCII), ou elles peuvent être **hétérogènes** donc avec des caractères de taille variable (UTF-8).
24. Défini comme *un caractère de taille suffisante pour pouvoir représenter n'importe quel caractère sur la machine de compilation*.

Il est rare d'avoir recours à cela : c'est utile lors de manipulation de chaines précises et définies. Il est cependant important de savoir qu'une chaine de caractère (en C++) homogène peut avoir des caractères de longueur 1, 2, ou 4 selon le contexte et qu'il existe des chaines hétérogènes également.