Architecture des Ordinateurs

Chapitre 1 : Programmation en C/C++

Jacques Supcik, Daniel Gachet, Luca Haab

2022-10-12

Table des matières

1 Programmation en C/C++ (pour programmeur Java)

Les systèmes d'exploitation et les systèmes embarqués sont encore beaucoup programmés en C ou en C++. Une première raison est historique ; les premiers systèmes d'exploitation UNIX qui ont été créés dans les années 70 et le langage *C* était, à cette période, le langage le plus adapté à la programmation de ces systèmes. Une autre raison est que le langage C est d'assez bas niveau et donne facilement accès aux interfaces proches du microprocesseur.

En 1985, Bjarne Stroustrup ajoute au langage *C* des mécanismes pour permettre la programmation orientée objet et sort le langage *C++*. Ce langage est très populaire, car il est compatible avec C et peut donc utiliser les nombreuses bibliothèques écrites pour *C* et sa syntaxe de base reste proche du *C* que les programmeurs de l'époque connaissent bien. Tout comme *C*, *C++* est un langage compilé qui offre d'excellentes performances et c'est aussi pour cette raison qu'il reste beaucoup utilisé dans les systèmes embarqués. L'environnement Arduino, très populaire pour les systèmes embarqués grand public utilise C++ comme langage de base.

Le langage Java que la plupart d'entre vous connaissent est sorti en 1995 et la syntaxe de Java reste proche de celle de C++.

Le C++ est un langage beaucoup plus complexe que Java et l'étude complète du langage sort du cadre de ce cours. Ce chapitre présente les principales différences entre Java et C++ pour vous permettre de rapidement écrire vos premiers programmes en C++ pour les travaux pratiques.

Le C++ a évolué depuis sa création et il existe plusieurs "dialectes" de C++. Ces derniers sont identifiés par un nombre qui indique l'année de publication du standard. Les standards officiels sont les suivantes :

— C++98 (1998)
— C++03 (2003)
— C++11 / C++0x (2011)
— **C++14** / C++1y (2014)
— C++17 / C++1z (2017)
— C++20 / C++2a (2020)

À partir de la version 6.5 du compilateur *gcc*, la version par défaut est *C++14* et c'est celle que nous utiliserons pour les travaux pratiques.

1.1 L'utilisation de bibliothèques

Les langages C et C++ n'offrent que les fonctionnalités de base et la plupart de programme que vous écrirez utiliseront des bibliothèques. Il a des bibliothèques standards disponibles pour la quasi-totalité des systèmes et il y a des bibliothèques spécialisées dans un domaine. Pour utiliser une bibliothèque dans un programme, il faut tout d'abord en importer l'interface avec la pseudo-instruction "`#include`". Nous avons écrit *pseudo*-instruction, car ce n'est pas vraiment une instruction du langage, mais plutôt du *préprocesseur*. Ce *préprocesseur* était nécessaire au temps où les compilateurs n'avaient que peu de mémoire à disposition et nous en avons encore besoin aujourd'hui pour les bibliothèques.

La version 20 de C++ permet de remplacer les "`#include`" par un concept de modules plus fiable et on pourrait bientôt pouvoir se passer de ce reliquat du passé. Mais pour l'instant, nous devons encore l'utiliser.

Si la bibliothèque est une bibliothèque que vous avez écrite vous-même pour votre projet, vous l'inclurez avec la syntaxe suivante :

```
1 #include "LIBRARY_FILE"
```

en remplaçant `LIBRARY_FILE` par le nom du fichier que vous voulez inclure.

Si par contre la bibliothèque est une bibliothèque système, vous utilisez la syntaxe suivante :

```
1 #include <LIBRARY_FILE>
```

L'utilisation de `""` ou de <> change la manière dont le processeur recherche les fichiers à inclure.

1.2 Les entrées sorties

Dans la plupart des ouvrages sur C++, vous verrez que pour afficher "Hello World !" sur la console, vous pouvez écrire :

```
1 std::cout << "Hello, World!" << std::endl;
```

Cette technique utilise le concept de flux (streams) et c'est, en effet, la manière standard de faire en C++. Cependant, avec des petits systèmes comme ceux que nous utilisons pour les travaux pratiques, cette technique n'est pas recommandée. Nous préférons utiliser la fonction "`printf`" de la librairie "`stdio.h`". Cette fonction est semblable à la méthode" `printf`" pour les instances de la classe "`PrintStream`" de Java.

Pour les entrées (input), vous pouvez également utiliser les flux de C++ avec l'objet "`cin`" ou alors la fonction "`scanf`".

1.3 La compilation séparée

Nous venons de voir comment inclure une bibliothèque avec la pseudo-instruction "`#include`" et le fichier que nous importons ainsi est en général un fichier avec l'extension "`.h`" ou "`.hpp`" (spécifique à C++).

Contrairement à Java où nous n'avons qu'un fichier "`.java`" et où nous utilisons les modificateurs "**`public`**", "**`protected`**" ou "**`private`**" pour indiquer ce qui est privé ou public, une bibliothèque C/C++ a besoin de 2 fichiers. Un fichier "`.c`" ou "`.cpp`" avec l'implémentation et un autre fichier "`.h`" ou "`.hpp`" avec la déclaration des éléments publics.

Les extensions "`.c`" et "`.h`" s'utilisent pour un fichier source en C et les extensions "`.cpp`" et "`.hpp`" pour un fichier en C++.

Certaines bibliothèques en C++ utilisent aussi l'extension "`.h`" et parfois même aucune extension, mais dans le contexte de ce cours, nous utiliserons l'extension "`.hpp`" pour les fichiers en C++.

Illustrons ceci avec un exemple. Supposons que nous souhaitons mettre à disposition une bibliothèque avec une fonction qui calcule le maximum de deux nombres. La fonction ressemblerait à ça :

```
1 int max(int a, int b) {
2     if (a > b) {
3         return a;
4     } else {
5         return b;
6     }
7 }
```

Pour pouvoir importer cette fonction dans un programme, nous écrivons un fichier "`.hpp`" (par exemple "`max.hpp`") avec le contenu suivant :

```
#ifndef max_h
#define max_h
int max(int a, int b);
#endif
```

On appelle ces fichiers des “header files” (d’où l’extension “.h”) et ils ne contiennent que les *entêtes* (ou les interfaces) des procédures.

Notez les trois autres pseudo-instructions qui commencent par un “#”. Ces instructions s’appellent des gardes (guards) et empêchent une interface d’être importée plusieurs fois. Le nom du symbole (“max_h”) peut être choisi librement, mais chaque fichier “.h” doit avoir un symbole différent. Une convention souvent utilisée consiste à reprendre le nom du fichier et à remplacer les . par des _. Certains ajoutent encore un “_” à la fin (“max_h_”) pour éviter de potentiels conflits.

On implémente ensuite la fonction dans un fichier “.cpp” (par exemple “max.cpp”) avec le contenu suivant.

```
#include "max.hpp"

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

Notez que l’implémentation importe sa propre interface (le fichier “max.cpp” importe “max.hpp”). Ce n’est pas toujours obligatoire, mais c’est une pratique très courante qui permet au compilateur de vérifier que l’implémentation correspond bien à l’interface.

Une application se compose souvent de plusieurs fichiers “.c”, “.cpp” / “.h” “.hpp” pour les bibliothèques locales ainsi qu’un fichier dans lequel est implémentée la méthode “int main()”. Vous pouvez appeler ce fichier comme vous voulez, mais on l’appelle souvent “main.c” ou “main.cpp”. La méthode main sera appelée lorsque vous exécuterez votre programme. Notez que ce fichier ne s’accompagne pas d’un fichier “.h” ou “.hpp”, car c’est le programme principal et il n’est pas importé par d’autres fichiers.

Le C et le C++ sont compatibles jusqu’à un certain point. Une fonction écrite en C peut être utilisée par un programme en C++ si on lui ajoute le préfixe “extern "C"”. Si vous voulez écrire une bibliothèque utilisable en C comme en C++, la bonne pratique consiste à implémenter le fichier “.h” de la manière suivante :

```
#ifndef GUARD_H
#define GUARD_H
```

```
3  #ifdef __cplusplus
4  extern "C"
5  {
6  #endif
7
8  // Declare C functions here
9
10 #ifdef __cplusplus
11 }
12 #endif
13 #endif
```

Le préprocesseur ajoute ce qu'il faut si on importe ce fichier depuis C++ et ignore la déclaration "`extern "C"`" supplémentaire si on importe depuis C.

1.4 Les types de données

En C/C++, nous retrouvons les types simples de Java tels que "**`int`**", "**`float`**" ou "**`char`**", mais contrairement à Java, la taille de ces types n'est pas définie et dépend de la machine pour laquelle est compilé le code. Un "**`int`**" sera sur 2 octets sur une machine 16 bits ou sur 4 octets sur une machine 32 ou 64 bits.

Une autre différence est que le C/C++ propose des types entiers non signés.

On peut souvent se contenter d'entiers dont la taille correspond à celle de la machine, mais nous avons aussi parfois de spécifier une taille précise. Nous pouvons le faire un utilisant la bibliothèque "`stdint`".

En C, nous importons la bibliothèque avec "`#include <stdint.h>`" et nous avons maintenant accès (entre autres) aux types suivants :

- `uint8_t`
- `int8_t`
- `uint16_t`
- `int16_t`
- `uint32_t`
- `int32_t`
- `uint64_t`
- `int64_t`

Un "`u`" en préfixe signifie *unsigned*.

En C++ nous pouvons aussi utiliser "`#include <stdint.h>`", mais pour éviter de "polluer" le "namespace" de C++, il serait préférable d'utiliser "`#include <cstdint>`" (avec un "`c`"

au début et pas d'extension). On devrait alors préfixer chaque type avec "`std::`" (par exemple "`std::int32_t`"), mais ça devient long à écrire et la plupart des implémentations de "`cstdlib`" mettent aussi les types dans le "namespace" global et nous pouvons les utiliser sans le préfixe "`std::`". Aucune solution n'est vraiment élégante et si vous voulez éviter le préfixe "`std::`" alors nous vous recommandons d'utiliser "`#include <stdint.h>`" comme en C.

Le type "**`char`**" en Java est sur 16 bits et encode les caractères en UTF-16. En C/C++, le type "**`char`**" est en général sur 8 bits et est soit signé, soit non signé.

En C il n'y a pas de type "**`boolean`**" et on utilise un "**`int`**" à la place. Il n'y a pas non plus de "**`true`**" ou de "**`false`**" mais un "`0`" représente *faux* et tout ce qui est différent de zéro représente *vrai*. En C++ le type `bool`, ainsi que les constantes "**`true`**" et "**`false`**" ont été ajoutés.

En C, une chaîne de caractère n'est autre qu'un tableau de caractères ("**`char`**`[]`"). Le caractère avec le code ASCII 0 marque la fin de la chaîne. Il faut toujours prendre en compte ce caractère de fin, et si on a besoin d'une chaîne de "`n`" caractères, il nous faut un tableau de taille "`n+1`". En C++, la bibliothèque standard offre un type "`string`" avec des caractéristiques semblables à celui de Java sauf que les "`String`" en Java sont immuables alors que le contenu des "`string`" de C++ peut être modifié.

Pour comparer des chaînes de caractères représentés par un tableau de caractères, vous pouvez utiliser la fonction "`strcmp`" ou `strncmp` de la bibliothèque "`string`" (à importer avec "`#include <string.h>`"). Pour les chaînes de caractères représentés par le type "`string`" de la bibliothèque standard, vous pouvez simplement les comparer avec l'opérateur "`==`". Contrairement à Java, C++ permet de redéfinir les opérateurs et ça permet de relier l'opérateur "`==`" à une méthode qui compare les deux objets.

1.5 Les variables

La déclaration de variables locales en C/C++ est la même qu'en Java. En plus des variables locales, C/C++ permet de déclarer des variables globales. Il suffit de déclarer ces variables en dehors des procédures. Une variable globale est accessible par toutes les procédures de tous les fichiers "`.c`" ou "`.cpp`" de votre application.

Vous pouvez limiter la portée des variables globales en utilisant le modificateur "**`static`**" (par exemple "**`static int`** `counter`"). Dans ce cas, la variable n'est globale qu'au fichier dans lequel elle est déclarée et non au programme tout entier. Les variables globales doivent être évitées le plus possible, et si vous en avez vraiment besoin, limitez alors leur portée avec le modificateur "**`static`**".

Les variables locales peuvent aussi être déclarées avec le modificateur "**`static`**". Dans ce cas, la variable est en fait globale, mais elle n'est visible que dans la procédure dans laquelle elle est déclarée.

Considérons par exemple la fonction "NextId" ci-dessous :

```
1 int NextId() {
2     static int id = 0;
3     return id++;
4 }
```

Cette fonction retourne "0" au premier appel, puis "1" au deuxième, "2" au troisième … La variable est globale et conserve sa valeur entre les appels.

Une variable en C/C++ peut aussi avoir le modificateur "**volatile**" et signifie au compilateur qu'il doit déclencher certaines optimisations, car cette variable peut être modifiée par d'autres parties. Nous verrons dans les TPs l'importance de ce modificateur dans le contexte des systèmes embarqués.

Si en C on doit déclarer le type de chaque variable explicitement, C++ est capable de faire de l'inférence de type. Utilisez pour cela le type "auto" :

```
1 auto myVar = 42;
```

Le compilateur choisira un type compatible avec l'assignation. Ce mécanisme peut être pratique, mais un type explicite, avec une taille précise, est souvent une meilleure option.

1.6 Les constantes

Historiquement, les constantes en *C* étaient déclarées avec la pseudo-instruction "#define" du préprocesseur. Par exemple "#define MAX_INT 65535". Le préprocesseur fera une substitution textuelle et remplacera simplement toutes les occurrences de "MAX_INT" par "65535". Le principal défaut de cette construction, c'est que la constante n'a pas de type et que la substitution textuelle peut être la source d'erreur difficile à trouver. Considérez l'exemple suivant :

```
1 #define MY_CONSTANT 12 + 4
2 int x = MY_CONSTANT * 2;
```

On pourrait croire que "MY_CONSTANT" vaut 16 et que "MY_CONSTANT * 2" vaut donc 32, mais comme le préprocesseur fait une substitution textuelle, l'assignation à la variable "x" devient "**int** x = 12 + 4 * 2;" et "x" vaut donc 20 et non 32 !

Si vous devez vraiment utiliser le préprocesseur et la pseudo-instruction "#define", assurez-vous alors d'entourer les expressions de parenthèses (par exemple "#define MY_CONSTANT (12 + 4)". Avec le C++, nous pouvons déclarer des constantes avec les modificateurs "**const**" ou "constexpr" :

```
1 constexpr int32_t kMyConstant = 12 + 4;
```

le modificateur "**const**" est l'équivalent du "**final**" de Java et indique que la valeur assignée ne changera pas. Cette valeur peut éventuellement n'être connue que lors de l'exécution du programme ("run-time"). Le modificateur "constexpr" indique que la valeur est calculée lors de la compilation.

La constante "kMyConstant" a un type précis et on obtient le résultat attendu si on la multiplie par une autre valeur. Le préfixe "k" dans le nom de la variable est une pratique tirée du "Google C++ Style Guide".

1.7 Les instructions de contrôle de flux

Les instructions de contrôle de flux telles que "**if**", "**while**", "**for**" ou "**switch**" sont pratiquement identiques en Java et en C/C++. Tout comme en Java, assurez-vous d'ajouter les instructions "**break**" dans les "**switch**/**case**".

1.8 Les classes

En Java, les classes permettent de regrouper des attributs et des méthodes dans un nouveau type de données. En C/C++, si on n'a que des attributs et pas de méthode, on peut utiliser le type "struct". Ce type permet de regrouper des attributs et est aussi disponible en C :

```
1 struct point {
2     int x;
3     int y;
4 }
```

On peut ensuite déclarer une variable de ce type :

```
1 point myPoint;
```

```
1 point myPoint = {12, 42};
```

ou en spécifiant les attributs :

```
1 point myPoint = {.x = 12, .y = 42};
```

On peut aussi combiner la déclaration du type et de la variable :

```
1 struct point {
```

```
    int x;
    int y;
} myPoint = {.x = 12, .y = 42};
```

Si on souhaite ajouter des *méthodes* à un type, nous pouvons le faire en C avec des pointeurs de fonctions, mais c'est une technique assez lourde et sujette aux erreurs. On préfère passer à C++ qui propose un vrai concept de classes et d'objets.

Nous pouvons déclarer une classe (par exemple, dans un fichier "`point.hpp`") de la manière suivante :

```
class Point {
public:
  Point();
  Point(int x, int y);
  void Move(int dx, int dy);
private:
  int x_;
  int y_;
};
```

La convention d'ajouter un underscore à la fin d'un attribut vient du "Google C++ Style Guide".

Nous implémentons ensuite les méthodes dans un fichier "`point.cpp`" :

```
#include "point.hpp"

Point::Point() {
    x_ = 0;
    y_ = 0;
}

Point::Point(int x, int y) {
    x_ = x;
    y_ = y;
}

void Point::Move(int dx, int dy) {
    x_ += dx;
    y_ += dy;
}
```

Notez que les constructeurs ("`Point::Point`") peuvent initialiser les attributs avec une syntaxe simplifiée :

```
Point::Point() : x_(0), y_(0) {};
Point::Point(int x, int y) : x_(x), y_(y) {};
```

Tout comme en Java, il est possible de définir des membres de classe statiques. Il suffit d'utiliser le

mot clé "`static`" [1].

Si un constructeur alloue des ressources (des fichiers, de la mémoire, ...), il faut aussi retourner ces ressources au système. Cette opération se fait en général avec un destructeur. C'est comme en Java, sauf qu'en Java le destructeur s'appelle "`finalize`" et en C++, est le nom de la classe précédé du symbole "~". C'est une bonne pratique de toujours ajouter un destructeur *virtuel*, même s'il est vide [2] :

```
1  class Point {
2  public:
3    Point();
4    virtual ~Point() {}; // destructor
5    Point(int x, int y);
6    void Move(int dx, int dy);
7  private:
8    int x_;
9    int y_;
10 };
```

1.9 Les objets

En Java, tous les objets sont des références. En C++ on a le choix. Par défaut, un objet n'est pas une référence, mais bien est une variable qui contient tous les attributs et les références vers les méthodes.

```
1  Point p(12,42);
```

On peut aussi déclarer un objet avec le constructeur par défaut :

```
1  Point p;
```

En Java, une déclaration similaire déclare une référence non initialisée (*null*) vers un point. En *C++* cette déclaration crée l'objet et l'initialise.

Si on copie un objet, on copie effectivement tous les attributs :

```
1  Point q = p;
```

En Java, l'instruction ci-dessus n'aurait fait qu'une nouvelle référence vers l'objet p, mais C++ crée réellement un nouvel objet. Après avoir créé l'objet, C++ appelle également un "copy-constructeur". Vous pouvez définir votre propre constructeur, mais si nous n'en définissez aucun, C++ définira un "trivial copy-constructeur" pour vous. Dans la plupart des cas, le "trivial copy-constructeur" est suffisant, car il copie récursivement tous les attributs de l'objet, mais si l'objet contient des pointeurs, il les copiera également tels quels, c'est-à-dire que les deux objets pointeront vers le même élément. Pour plus de

1. https ://en.cppreference.com/w/cpp/language/static
2. Voir : https ://en.cppreference.com/w/cpp/language/destructor

détails concernant les “copy-constructeurs” nous vous conseillons de lire la page copy_constructor [3] sur cppreference.com.

C++ n’est pas obligé de copier chaque objet et supporte également le concept de référence en faisant appel aux *pointeurs*. Nous verrons ça plus tard dans ce chapitre.

1.10 Les fonctions

En Java, toutes les méthodes sont liées à une classe. En C++ (de même qu’en C), les *méthodes* peuvent exister indépendamment de toute classe. Notez qu’en C nous parlerons plutôt de *fonctions* ou de *procédures* [4] à la place de *méthodes* ; les *méthodes* sont liées aux classes de C++.

```
1 int Max(int x, int y) {
2     return x > y ? y : x;
3 }
```

En Java les arguments de types simples (“`int`”, “`float`”, …) sont toujours passés par valeur (copie) et les arguments de type “`string`”, les tableaux et les objets sont toujours passés par référence. En C++ les tableaux sont passés par référence, mais pour les autres types on a le choix. Par défaut, les arguments sont passés par valeur, mais on peut les passer par référence si on ajoute un “&” à la fin du type. Par exemple :

```
1 void Swap(int& a, int& b) {
2     int temp = a;
3     a = b;
4     b = temp;
5 }
```

Lorsque nous appelons “`Swap(x,y)`”, nous passons des références vers “`x`” et “`y`” et la procédure “`Swap`” peut modifier les valeurs de “`x`” et de “`y`”. Notez que si vous appelez avec des constantes (“`Swap(1,2)`”), la compilation ne sera pas possible et le compilateur indiquera une erreur.

On peut aussi choisir de passer des arguments par référence pour des questions de performance sans forcément vouloir laisser la fonction modifier le contenu de l’argument. On peut le faire en C++ en ajoutant le préfixe “`const`” :

```
1 void Find(const BigObject& o, int x) {
2     ...
3 }
```

3. https ://en.cppreference.com/w/cpp/language/copy_constructor
4. La différence entre une *fonction* et une *procédure* c’est qu’une fonction retourne une valeur alors qu’une procédure ne retourne rien (“`void`”).

Tout comme Java, le C++ permet de surcharger des fonctions du moment que les arguments sont de types différents :

```
int Max(int a, int b) {
    return a > b ? a : b;
}

float Max(float a, float b) {
    return a > b ? a : b;
}
```

La surcharge de fonction n'est pas possible en C.

Une autre caractéristique de C++ qui n'est pas possible en C ou en Java est et de permettre des valeurs par défaut pour les arguments :

```
void Sort(Array& array, bool descending=false) {
    ...
}
```

On peut appeler cette méthode avec un ou deux arguments :

```
Sort(myArray);        // descending = false (default)
Sort(myArray, false); // OK, but redundant
Sort(myArray, true);  // descending = true
}
```

Cette construction permet bien souvent de s'épargner la surcharge de fonction et elle simplifie aussi le code.

1.11 Les tableaux et les vecteurs

En C, un tableau est juste une séquence d'éléments en mémoire et il est représenté par l'adresse du premier élément. Si un "`int`" fait 4 octets, un tableau de 10 "`int`" fait 40 octets. Le C n'a aucun moyen de connaître la taille d'un tableau et ne peut donc pas vérifier que l'index soit dans le domaine autorisé.

Un tableau de 10 entiers est déclaré comme ça :

```
int a[10];
```

Si on passe ce tableau à une procédure, un passe simplement l'adresse du premier élément. Le tableau est passé par référence à la procédure. Comme C ne connaît pas la taille d'un tableau passé en argument, il est commun de passer un argument supplémentaire avec la taille.

La fonction suivante est un exemple d'une telle fonction qui détecte si un nombre est présent dans un tableau :

```
bool IsPresent(int x, int array[], int length) {
    for (int i = 0; i < length; i++) {
        if (array[i] == x) return true;
    }
    return false;
}
```

Ce traitement très "bas niveau" des tableaux en C est la source de beaucoup d'erreurs en C et le C++ offre une structure similaire aux tableaux de Java avec le type "`vector`" de la bibliothèque standard. Avec le type "`vector`", la fonction "`IsPresent`" devient :

```
bool IsPresent(int x, std::vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        if (v[i] == x) return true;
    }
    return false;
}
```

Notez qu'en C++, comme en Java, on peut utiliser l'instruction "**`for`**" pour itérer sur une plage (Range-based for) :

```
bool IsPresent(int x, std::vector<int>& v) {
    for (int i : v) {
        if (i == x) return true;
    }
    return false;
}
```

C'est également possible avec des tableaux dont la taille est connue à la compilation :

```
int a[] = {1,2,3,4,5};
int sum = 0;
for (int i : a) {
    sum += i;
}
```

Mais ça ne fonctionne pas quand le tableau est passé en paramètre :

```
bool IsPresent(int x, int v[]) {
    for (int i : v) { // !!! ERROR
        if (i == x) return true;
    }
    return false;
}
```

En C++, l'opérateur "`[]`" ne vérifie pas que l'index soit valide et on peut très bien lire ou écrire au-delà des limites du tableau. Si on souhaite une vérification au "run time", on doit utiliser la méthode "`at`" au lieu de "`[]`".

Les tableaux simples sont plus efficaces et prennent moins de place en mémoire et les vecteurs sont plus flexibles.

1.12 Les pointeurs

Nous avons vu plus haut qu'en C++, les objets ne sont pas forcément des références. Si on déclare une variable "`Point p(12,42);`", alors "`p`" est l'objet lui-même et non une référence. Nous avons aussi vu que nous pouvions passer un objet "par référence" à une procédure en ajoutant le suffixe "&" au type. Mais, en C++, nous pouvons aussi obtenir une *référence* vers un objet en utilisant "&" comme opérateur et non comme suffixe à un type. Considérons l'exemple suivant :

```
1 Point p(12,42);
2 Point* p2 = &p;
```

Nous voyons deux choses. Tout d'abord le type de "`p2`" est "`Point*`" avec une "`*`" et on lit "`p2` est un pointeur vers un `Point`". Nous voyons aussi l'opérateur "&" qui signifie "l'adresse de". La deuxième ligne se lit donc "`p2` est un pointeur vers un Point et il pointe actuellement vers l'objet `p`". On pourrait aussi dire que "la valeur de `p2` est l'adresse de `p`".

Comme en Java, on peut aussi assigner une valeur nulle à un pointeur :

```
1 Point* p2 = NULL;
```

Dans le chapitre sur les fonctions, on a vu quelque chose qui ressemblait aux pointeurs avec les *références*. Nous avons vu ce concept dans le passage d'arguments, mais on peut aussi l'utiliser en dehors de ce contexte. On peut par exemple écrire :

```
1 Point p(12,42);
2 Point& p2 = p;
```

Dans ce cas "`p2`" est une *référence* sur le point "`p`". Contrairement aux pointeurs, les références sont des **constantes** et par conséquent ne peuvent pas être "**`null`**".

Nous pouvons aussi créer un objet dynamiquement avec l'opérateur "`new`" qui appelle aussi le constructeur :

```
1 Point* p2 = new Point(12, 42);
```

Jusqu'ici ça ressemble assez à Java, mais C++ a une syntaxe différente pour accéder aux attributs d'un objet représenté par un pointeur. En effet, "p2" dans les exemples ci-dessus est un pointeur et si on veut l'objet lui-même, il faut tout d'abord "déréférencer" le pointeur avec l'opérateur "*" :

```
1 Point* p2 = new Point(12, 42);
2 (*p2).move(1,2);
```

Cette deuxième ligne peut se lire : "suis le pointeur"p2" pour obtenir un objet de type "Point" et déplace de point selon les arguments". Notez que les parenthèses sont nécessaires, car l'opération "." a une précédence plus élevée que "*".

La combinaison d'opération "déréférence" + "accès à un attribut" est très fréquente eu C++ et le langage nous offre un raccourci. Nous pouvons écrire cette deuxième ligne ainsi :

```
1 p2->move(1,2);
```

Vous pouvez vous souvenir que l'opérateur "." s'utilise pour les objets et "->"s'utilise pour les pointeurs.

On peut aussi utiliser l'opérateur "*" pour copier un objet référencé :

```
1 Point p3 = *p2;
```

Ici "p3" n'est plus une référence, mais un nouvel objet initialisé avec les membres de "p2".

Ce qui est déroutant au début avec C++ c'est que les symboles "&" et "*" peuvent être utilisés comme opérateur ou comme suffixe à un type.

- "&" comme opérateur précède une variable et retourne l'adresse de cette variable : &p
- "&" comme suffixe à un type est utilisé dans les arguments des procédures et signifie que l'argument est passé par référence : "(Point& p)"
- "*" comme opérateur précède une variable de type pointeur et retourne l'objet pointé par la variable : "*p2"
- "*" comme suffixe à un type indique que le type est pointeur vers ce type : "Point* p2"

Sachez que C++ ne vérifie pas si un pointeur est une référence valide et l'utilisation de pointeurs non initialisés peut provoquer de graves erreurs difficiles à trouver.

Une autre différence de taille est que C++ n'a pas de "*garbage collector*" comme Java. Lorsque vous créez une référence avec "**new**", il est de votre responsabilité de rendre la mémoire avec l'opérateur "delete" :

```
1 Point* p2 = new Point(12, 42);
2 ...
3 delete p2;
```

Si vous oubliez de rendre la mémoire, personne ne le fera à votre place et votre programme souffrira d'un bug connu sous le nom de "memory leak". Si le programme tourne suffisamment longtemps, il risque d'utiliser toute la mémoire disponible et provoquera l'arrêt brutal du programme.

En C++, les pointeurs ne sont pas réservés aux objets, et on peut très bien déclarer un pointeur vers un entier :

```
1 int i = 42;
2 int* j = &i;
3 *j = 43;
```

Dans l'exemple ci-dessus, "`j`" est un pointeur vers l'entier "`i`" et si on souhaite obtenir ou modifier sa valeur, vous devons le déréférencer avec l'opérateur "`*`".

l'opérateur "**`new`**" peut aussi être utilisé en C++ pour créer des tableaux dynamiques. L'exemple suivant crée un tableau "`a`" de 10 entiers :

```
1 int* a = new int[10];
```

Le tableau est créé au "*run time*" et la variable "`a`" est donc un pointeur. Ca explique la notation du type avec l'astérisque ("**`int`**`*`"). Mais où est donc passé l'information que c'est un tableau ? Pourquoi n'avons-nous pas écrit "**`int`**`[]* a`" pour indiquer que "`a`" est un pointeur vers un tableau ? Comment savoir si "`a`" pointe vers un entier ou vers un tableau d'entiers ?

Pour C/C++, l'adresse vers un tableau est équivalente à l'adresse vers le premier élément du tableau. Quand nous voyons "**`int`**`* a`" nous savons que "`a`" pointe vers un entier, mais il peut aussi pointer sur le premier élément d'un tableau d'entiers.

Tout comme pour les objets, le programmeur C++ est responsable de rendre la mémoire allouée pour un tableau avec l'opérateur "`delete[]`".

```
1 delete[] a;
```

Nous retrouvons souvent ce concept avec les chaînes de caractères. En C/C++, une chaîne de caractères est un tableau de caractères et si une fonction attend une chaîne de caractère comme paramètre, elle le spécifie souvent comme un pointeur. Nous pouvons le voir par exemple dans la déclaration de la fonction "`printf`" :

```
1 int printf (const char* format, ...);
```

Le "**`const`**" indique que la méthode ne modifie pas la variable "`format`" et permet au compilateur de mieux optimiser le code.

Les opérateurs "**`new`**" et "`delete`" sont propre à C++. En C, nous pouvons aussi allouer et retourner de la mémoire avec les fonctions "`malloc`/`calloc`" et "`free`" de la bibliothèque `"stdlib.h`".

En C/C++, en plus de pointer vers des variables, un pointeur peut aussi pointer vers une fonction. Les pointeurs de fonctions sont utilisés en C pour émuler les méthodes et pour permettre une pseudo-programmation orientée objet. On les utilise aussi un peu comme des *interfaces fonctionnelles* ou des *lambdas* [5] de Java comme par exemple la fonction "`bsearch`" de la bibliothèque "`stdlib.h`" qui implémente une recherche binaire :

```
#include <stdlib.h>
void *bsearch(const void* key, const void* base,
    size_t nmemb, size_t size,
    int (*compar)(const void*, const void*));
```

Dans cette déclaration, nous voyons des pointeurs vers "**`void`**"qui sont des pointeurs "universels" et correspondent aux "`Object`" de Java et nous voyons aussi l'argument "`compar`" qui est un pointeur de fonction. Pour être compatible avec l'argument, le pointeur doit pointer vers une méthode qui prend deux arguments de type ("**`const void`**`*`") et retourne un "**`int`**".

À l'intérieur des méthodes, tout comme en Java, le mot clé "`this`" est une référence vers l'objet pour lequel la méthode a été appelée. En C++, "**`this`**" est un pointeur, et pour accéder aux membres, on doit tout d'abord le déréférencer ou utiliser la notation "`->`" (par exemple "**`this`**`->Print();`").

Selon une étude de Microsoft [6], 70% des bugs de sécurité sont liés à la gestion de la mémoire. Ce n'est pas facile de bien gérer la mémoire et l'absence de "*garbage collector*" dans C++ ne simplifie pas la tâche. Afin d'adresser cette problématique, la version C++11 propose le concept de "*Smart Pointer*". Ces "*Smart Pointers*" sont des sortes de "*wrappers*" qui s'occupent de libérer la mémoire allouée pour un pointeur de manière sûre. Ces "*Smart Pointers*" sont disponibles en important la bibliothèque "`<memory>`".

On distingue 3 types de "*Smart Pointer*" :

- Les "`unique_ptr`"
- Les "`shared_ptr`"
- Les "`weak_ptr`"

Un "`unique_ptr`", comme son nom l'indique, est unique et ne peut pas être partagé. On ne peut pas le copier vers un autre "`unique_ptr`" et on ne peut pas le passer "par valeur" à une fonction. En d'autres termes, il ne peut y avoir qu'un seul *propriétaire* de ce pointeur.

```
void UseSmartPointer()
{
    unique_ptr<Point> p(new Point(12, 42));
    // ...
} // p is automatically deleted here.
```

5. Le C++ connaît aussi le concept de *lambda* comme proposé par Java, mais nous ne l'étudierons pas dans le cadre de ce cours.

6. https://youtu.be/PjbGojjnBZQ?t=835

Dès que le propriétaire disparaît, son destructeur libère la mémoire. On ne risque donc plus d'oublier le "`delete`".

Si on souhaite avoir plusieurs propriétaires pour un pointeur, il faut utiliser des "`shared_ptr`" avec éventuellement des "`weak_ptr`". Ces concepts sortent du cadre de cours d'introduction et le lecteur intéressé trouvera beaucoup d'information en ligne ou dans la littérature sur le sujet.

1.13 L'héritage

La syntaxe de l'héritage en C++ est semblable à celle de Java. En C++, nous utilisons "`: public`" au lieu de "`extends`" pour indiquer l'héritage.

Si nous partons du principe qu'un *pixel* est un point avec de la couleur, nous pouvons alors définir la classe "`Pixel`" ainsi :

```
1 class Pixel : public Point {
2 public:
3     Pixel(int x, int y, uint16_t color);
4     void Paint();
5 private:
6     uint16_t color_;
7 };
```

En Java, on appelle le constructeur de la classe parente avec le mot clé "`super`". En C++, on appelle simplement le constructeur de la classe parente directement avec son nom :

```
1 Pixel::Pixel(int x, int y, uint16_t color) : Point(x, y) {
2     color_ = color;
3 }
```

Dans l'exemple ci-dessus, l'appel du constructeur de la classe parente est à l'extérieur du "body" et l'attribut "`color_`" est initialisé à l'intérieur.

On peut aussi initialiser l'attribut à l'extérieur :

```
1 Pixel::Pixel(intx, inty, uint16_tcolor)
2     : Point(x, y)
3     , color_(color)
4 {
5 }
```

Il n'est cependant pas possible d'appeler le constructeur de la classe parente à l'intérieur du *body*; l'appel doit être à l'extérieur,

Une autre caractéristique de C++ est que par défaut, les méthodes ne peuvent pas être redéfinies dans

une classe dérivée. C'est l'équivalent d'une méthode "**final**" de Java. Pour permettre une redéfinition d'une méthode, il faut préfixer la déclaration avec le mot clé "virtual".

```
class Pixel : public Point {
public:
    Pixel(int x, int y, uint16_t color);
    virtual void Paint();
private:
    uint16_t color_;
};
```

On peut maintenant définir un "SuperPixel" qui hérite de "Pixel" et qui redéfinit Paint :

```
class SuperPixel : public Pixel {
public:
    SuperPixel(int x, int y, uint16_t color) : Pixel(x, y, color) {};
    void Paint();
};

void SuperPixel::Paint() {
    ...
}
```

Pour bien montrer qu'une méthode est redéfinie, on peut ajouter le mot clé "override" après la déclaration :

```
class SuperPixel : public Pixel {
public:
    SuperPixel(int x, int y, uint16_t color) : Pixel(x, y, color) {};
    void Paint() override;
};
```

Comme en Java, les attributs privés ne sont pas accessibles dans les classes dérivées. On peut les déclarer public, mais si on ne souhaite pas ouvrir autant et n'autoriser l'accès qu'aux classes dérivées, on peut utiliser le modificateur "**protected**".

```
class Point {
public:
  Point();
  Point(int x, int y);
  void Move(int dx, int dy);
protected:
  int x_;
  int y_;
};
```

Le langage C++ est très complexe et cette courte introduction n'en a effleuré que la surface, mais ça devrait vous permettre de réaliser les TPs. Si vous souhaitez vous perfectionner en C++, vous trouverez beaucoup de références en lignes ainsi que de nombreux ouvrages spécialisés.

Notez que le C++ supporte l'héritage multiple. C'est-à-dire qu'une classe peut avoir plusieurs parents et contrairement à Java, il n'y a pas de syntaxe spécifique pour une *interface*. Par contre, vous pouvez déclarer une méthode abstraite (l'équivalent de "**abstract**" en Java) en mettant "= 0" après la signature de la méthode. Par exemple :

```
1 virtual int f() = 0;
```

En plus des méthodes, le C++ permet aussi de redéfinir les opérateurs tels que "+" ou "=".

Ces concepts sont intéressants, mais leurs détails dépassent le cadre de ce chapitre.

1.14 Exercice 1 : Opérations binaires

Pour cet exercice, vous pouvez installer un environnement de développement C/C++ sur votre machine, ou simplement utiliser un compilateur dans le "cloud" tel que sur : tutorialspoint.com [7].

1.14.1 CountBits

Écrivez une fonction en C++ qui compte le nombre de bits à "1" dans un entier positif La signature de cette méthode est :

```
1 int CountBits(uint32_t n);
```

Testez votre fonction avec un programme principal.

1.14.2 MostSignificantBit

Écrivez une fonction en C++ qui détermine le bit le plus signifiant dans un entier positif La signature de cette méthode est :

```
1 int MostSignificantBit(uint32_t n);
```

Testez votre fonction avec un programme principal.

1.14.3 ReverseBits

Écrivez une fonction en C++ qui renverse les bits d'un entier positif. Le bit 31 (poids fort) est échangé avec le bit 0, le bit 30 est échangé avec le bit 1, etc.

7. https ://www.tutorialspoint.com/compile_cpp_online.php

La signature de cette méthode est :

```
uint32_t ReverseBits(uint32_t n);
```

Testez votre fonction avec un programme principal.

1.14.4 Autres opérations binaires

Implémentez les fonctions ci-dessous conformément aux commentaires :

```
uint32_t SetBit(uint32_t n, int i);    // returns n with bit i set
uint32_t ClearBit(uint32_t n, int i);  // returns n with bit i cleared
uint32_t ToggleBit(uint32_t n, int i); // returns n with bit i inverted
bool TestBit(uint32_t n, int i);       // tests if bit i of n is set
```

Testez vos fonctions avec un programme principal.

1.15 Exercice 2 : La fonction mystère

Implémentez l'algorithme suivant dans une fonction C++ :

1. soit n un nombre tel que $n \leq 9999$ et qui ne soit pas composé de 4 fois le même chiffre (c'est-à-dire pas 0000, 1111, 2222, …)
2. calculer max comme le plus grand nombre de 4 digits qu'on peut obtenir avec les digits de n
3. calculer min comme le plus petit nombre de 4 digits qu'on peut obtenir avec les digits de n
4. calculer $d = max - min$
5. si $d = n$ retourner d et l'algorithme est terminé
6. sinon, assigner d à n et reprendre le point 2.

Pour cet exercice, nous vous recommandons d'utiliser des vecteurs et la fonction "`std::sort`" proposée par la bibliothèque "`algorithm`". Votre programme pourrait importer les bibliothèques suivantes :

```
#include <iostream> // std::cout
#include <algorithm> // std::sort
#include <vector> // std::vector
```

Voici quelques opérations que vous pouvez faire avec un vecteur d'entiers :

```
std::vector<int> a;
a.insert(a.begin(), x); // insert x at the beginning of the vector
a.push_back(x); // append x at the end of the vector
std::sort(a.begin(), a.end()); // sort the vector a (ascending)
```

```
5 std::sort(a.begin(), a.end(), std::greater<int>()); // sort descending
```

Si vous voulez savoir pourquoi votre méthode retourne toujours 6174, lisez l'article suivant sur Wikipedia :
https ://en.wikipedia.org/wiki/6174_(number)

1.16 Exercice 3 : Affichage 7-segments

Écrivez un programme en C++ qui simule un affichage à sept segments.

L'interface publique de la classe `Segments` est donnée :

```
1 class Segments {
2 public:
3     Segments(int n = 1);
4     void Set(int v);
5     void Draw(int w = 3, int h = 2, int dx = 1);
6 private:
7     // TODO
8 };
```

L'argument du constructeur indique le nombre de "digits" qui compose l'affichage. La méthode `Set` définit le nombre à afficher. La méthode `Draw` simule l'affichage avec des segments horizontaux de `w` caractères, des segments verticaux de `h` caractères et un espace de `dx` entre deux segments.

Testez avec le programme principal suivant :

```
1 int main()
2 {
3     Segments s(4) ;
4     s.Set(639);
5     s.Draw();
6 }
```

Le résultat devrait ressembler à ça :

```
1  ---   ---   ---   ---
2 |   | |         | |   |
3 |   | |         | |   |
4          ---   ---   ---
5 |   | |   |     |     |
6 |   | |   |     |     |
7  ---   ---   ---   ---
```

Pour l'affichage, vous pouvez utiliser la classe `Canvas` que nous vous donnons ci-dessous, mais vous pouvez aussi faire autrement si vous voulez :

```
#include <iostream>
#include <vector>
#include <stdint.h>

class Canvas {
public:
    Canvas (int w, int h);
    virtual ~Canvas() {};
    void Clear (char c = '.');
    void HLine(int x, int y, int len, char c = '*');
    void VLine(int x, int y, int len, char c = '*');
    void Print();
private:
    std::vector<std::vector<char>> canvas_;
};

Canvas::Canvas (int w, int h) {
    canvas_.resize(h);
    for (int y = 0; y < h; y++) {
        canvas_[y].resize(w, ' ');
    }
}

void Canvas::Clear (char c) {
    for (int y = 0; y < canvas_.size(); y++) {
        for (int x = 0; x < canvas_[y].size(); x++) {
            canvas_[y][x] = c;
        }
    }
}

void Canvas::HLine(int x, int y, int len, char c) {
    for (int i = 0; i < len; i++) {
        canvas_[y][x+i] = c;
    }
}

void Canvas::VLine(int x, int y, int len, char c) {
    for (int i = 0; i < len; i++) {
        canvas_[y+i][x] = c;
    }
}

void Canvas::Print() {
    for (auto row: canvas_) {
        for (auto c: row) {
            std::cout << c;
        }
```

```
49          std::cout << std::endl;
50      }
51  }
```