Introduction à la programmation objet - C à C++

January 6, 2023

Outline

Table of Contents

Code source / Compilation

Sources

- les fichiers sources C++ ont généralement
  - l'extension `.cc` ou `.cpp`
  - l'extention `.h` ou `.hpp` pour les fichiers d'entête

Compilation

- la compilation se fait comme pour C avec un compilateur qu'on invoque avec `c++`, `g++`, `cpp`
- ce sont en général des alias vers un compilateur qui n'a pas forcément ce nom
- De nombreux compilateurs C++ existent : voir page wikipedia
- Deux principaux compilateur open-source : `gcc` (GNU) et `clang` (LLVM)
- D'autres plus ou moins à source ouvert : `icc` puis `icx` (Intel)

Invocation en ligne de commande

- Le vrai nom du compilateur

```
% g++ --version
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --with-gxx-include-dir=/usr/include/c+
Apple LLVM version 9.0.0 (clang-900.0.39.2)
Target: x86_64-apple-darwin17.3.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
```

- Un exemple simple : même principes de compilation qu'en C

```
% g++ main.cc utilitaires.cc biblio.cc -o biblio
% ./biblio
```

- Invocation d'un standard, e.g C++11

```
% g++ -std=c++11 main.cc utilitaires.cc biblio.cc -o biblio
```

Standardisation ISO

- 98 première normalisation
- 03 : corrections du C++98 + value initialization
- 11: beaucoup de nouveautés dont l'inférence de type (`auto`), boucle for range-based, constexpr, lambdas, override+final, initializer lists, modèle mémoire pour multi-threading.
- 14 : mise-à-jour de C++11
- 17 : nouveautés dont uniformisation accès containers (`std::size`, `std::data`, `std::empty`), type `std::any`. Inclusion lib. FileSystem. Parallelisation de certains algorithmes.
- 20 : nouveautés
- 23 : nouveautés

Table of Contents

Hello World

- Nouvelle biblothèque `iostream` en remplacement de `<stdio.h>`.
- fournit `cout`, `cin`, `endl`, ...

```
#include <iostream>

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

- Notation `::` pour la résolution de portée, vers un espace de noms ici `std`

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

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

Hello World

- `cout` et `cin` représentent resp. la sortie et l'entrée standard
- on peut envoyer vers resp. la sortie et entrée avec les opérateurs « et »
- « et » permettent d'enchainer l'affichage de types différents (surcharge).

```
#include <iostream> // en remplacement de <stdio.h>
#include <string> // different de <string.h>
using namespace std;

int main() {
   int age;
   string nom;
   cout << "Entrez votre nom:";
   cin >> nom;
   cout << "Entrez votre age:";
   cin >> age;
   cout << "Bonjour " << nom << ", vous avez " << age << " ans." << endl;
   return 0;
}
```

C++ Standard Libray (stdlib)

- Collection de classes et fonctions qui font partie du standard ISO.
- Fournie à travers différentes entêtes (liste) (e.g `<iostream>`).
- Tout ce qui est fourni est déclaré dans l'espace de nom `std`.

- Différentes implémentations existent : `libstdc++` (gcc), `libc++` (clang/llvm), ...
- Note : il y a une différence entre la stdlib (normalisée) et la STL (Standard Template Library), implémentation de containers et d'algorithmes qui a servi de base à la stdlib. La confusion reste répandue.

Table of Contents

Grandes idées

- C++, en tant que LOO, "renforce" la place des structures de données.
  - Favorise la déclaration de données intialisées au plus tôt
  - Certains types de données sont des classes (les types primitifs du C restant).
  - Les types de données peuvent être génériques avec les templates
- C++ offre des moyens de "sécuriser" la gestion de la mémoire (bien qu'elle reste à la charge du programmeur).
- C++ vient avec une bibliothèque normalisée stdlib plus riche en algorithmes courants.

... Bien sûr, la façon de concervoir une application change bien plus que ça avec les principes de la POO. Ce sera l'objet du cours à suivre ....

Les initialisations

- En C, des variables de type primitif peuvent être initialisées:

```
int a = 10;
double = 40.4;
```

- En C++, toujours possible, mais notation alternative davantage utilisée (rappelle l'utilisation d'un constructeur)

```
int a(10); // ou int a { 10 } depuis c++11
double d(40.4); // ou double d { 40.4 } depuis c++11
string s("hello"); // ou string s { "hello" } depuis c++11
const double SMIC_net(1188.);
```

- C'est le principe RAII qui transparait ici. On le retrouvera sur la déclaration des objets.

Table of Contents

Gestion dynamique de la mémoire

- Equivalents de `malloc()` et `free()` en C
- C++ propose les opérateurs `new` et `delete` en remplacement

```
int main() {
  int *p;
  int *tableau;

  p = new int; // allocation d'une zone memoire pour 1 entier
  tableau = new int[20]; // allocation d'un tableau de 20 entiers

  delete p;
  delete[] tableau; // liberation de memoire (tableau)
  return 0;
}
```

- Attention à être cohérent :

```
int main() {
int *t = new int[20];
delete t; // oops! ** error: pointer being freed was not allocated
}
```

Gestion dynamique de la mémoire

- Un pointeur non-initialisé doit prendre la valeur `nullptr` (oubliez `NULL` ou `0`)
- Les opérateurs `new` et `delete` doivent être considérés comme des opérations bas niveau que l'évolution de C++ tend à bannir. On essaye de les garder dans du code bien testé de bibliothèque, ou de les encapsuler pour éviter au programmeur de s'en occuper.
- Est favorisé l'utilisation de pointeurs intelligents (*smart pointers*) qui gèrent de manière autonome l'acquisition / relâchement de mémoire (voir plus tard).

Les références

- La notion de référence est une nouveauté du C++ même si elle étend celle du C

Les références

- La notion de référence est une nouveauté du C++ même si elle étend celle du C
- Une référence à une variable peut être interprétée comme un alias vers cette variable : c'est un autre nom pour désigner la même adresse mémoire

Les références

- La notion de référence est une nouveauté du C++ même si elle étend celle du C
- Une référence à une variable peut être interprétée comme un alias vers cette variable : c'est un autre nom pour désigner la même adresse mémoire
- Notation : & précédant le nom de la référence

```
int main() {
   int valeur = 10;
   int &refvaleur = valeur; // on cree la reference

   cout << "valeur=" << valeur << ", ref=" << refvaleur << endl;
   valeur=15;
   cout << "valeur=" << valeur << ", ref=" << refvaleur << endl;
   return 0;
}
```

Les références

- La notion de référence est une nouveauté du C++ même si elle étend celle du C
- Une référence à une variable peut être interprétée comme un alias vers cette variable : c'est un autre nom pour désigner la même adresse mémoire
- Notation : & précédant le nom de la référence

```
int main() {
  int valeur = 10;
  int &refvaleur = valeur; // on cree la reference

  cout << "valeur=" << valeur << ", ref=" << refvaleur << endl;
  valeur=15;
  cout << "valeur=" << valeur << ", ref=" << refvaleur << endl;
  return 0;
}
```

- Exécution:

```
% ./a.out
valeur=10, ref=10
valeur=15, ref=15
```

Les références : Attention !

Attention

- la notation & n'est une référence que si elle apparaît que dans une déclaration
- l'opérateur & existe toujours pour obtenir l'adresse d'une variable.

Règle

- une référence doit être initialisée dès sa déclaration : `int &refvaleur = valeur;`. Le compilateur le vérifie.

Les références : et les pointeurs ?

- Les pointeurs existent toujours
- Les références sont une alternative au pointeur, avec l'objectif de simplifier la manipulation des adresses

références

```
int main() {
    int v = 10;
    int &refv = v; // on cree la reference

    cout << "v=" << v << ",␣refv=" << refv << endl;
    v=15;
    cout << "v=" << v << ",␣refv=" << refv << endl;

    return 0;
}
```

pointeurs

```
int main() {
    int v = 10;
    int *refv = &v;
// faire pointer refvaleur vers l'adresse de valeur

    cout << "v=" << v << ",␣ref=" << *refv << endl;
    v=15;
    cout << "v=" << v << ",␣refv=" << *refv << endl;
    return 0;
}
```

Les références dans les paramètres de fonctions

- Les références sont beaucoup utilisées pour passer l'adresse d'une variable à une fonction (et donc permettre la modification du contenu) sans avoir de notation de manipulation d'adresse

```
struct coord {
    int x;
    int y;
};

void init_coord(coord &point) {
    // point une reference, s'utilise comme un nom de
    // variable normal, ne pas utiliser ->
    point.x = 0;
    point.y = 0;
};

int main() {
    coord point;
    init_coord(point); // Inutile d'utiliser & lors de l'appel

    return 0;
}
```

Les références dans les paramètres de fonctions

- En comparaison, on aurait écrit avec les pointeurs:

```
struct coord {
    int x;
    int y;
};

void init_coord(coord *point) {
    // point est un pointeur, utiliser ->
    point->x = 0;
    point->y = 0;
};

int main() {
    coord point;
    init_coord(&point);

    return 0;
}
```

Table of Contents

Les strings

- `char`, `int`, `bool`, `float`, `double` sont des types élémentaires de C++ : codage sur longueur fixe en mémoire
- En C, une chaine de caractères est un tableau dont on mémorise l'adresse de début
- C++ se dote d'un type `string` (`#include <string>`) qui est une classe. Les manipulations sont beaucoup plus simples.

```
int main() {
   string chaine1("Bonjour");
   string chaine2("Monde");
   string chaine3;

   chaine3 = chaine1 + "␣" + chaine2; // concatenation
   cout << chaine3 << endl; // "Bonjour Monde"
   chaine3 = "Hello"; // redefinition
   cout << "len(chaine3)=" << chaine3.size();
   cout << "car␣2-5␣=" << chaine3.substr(2,3) <<endl; // sous-chaine "llo"
   return 0;
}
```

La Standard Library

- En C, difficile d'implémenter des containers (vecteurs, listes, ensembles ordonnés) génériques, simple d'utilisation et efficaces (utilisation de `void *`).
- C++ vient avec la *stdlib* qui fournit un grand nombre de classes génériques
- Avant de développer un type abstrait de données de type conteneur, se demander si il n'est pas disponible dans la *stdlib*
- Voir Documentation de Référence

Containers: Quelques exemples

paire	std::pair<T1,T2>
liste	std::list<T,...>
vecteur	std::vector<T,...>
ensemble	std::set<T,...>
table de hachage	std::map<K,T,...>

Container: exemple vector

- Vecteur : conteneur de valeurs stockées de manière contigue représentant des tableaux, dont la taille peut changer dynamiquement.

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

int main() {
  std::vector<int> mon_vecteur;
  mon_vecteur.push_back(4);
  mon_vecteur.push_back(2);
  mon_vecteur.push_back(5);

  // Pour parcourir un vector on peut utiliser les iterators ou les index
  for(std::size_t i=0; i<mon_vecteur.size(); ++i) {
    std::cout << mon_vecteur[i] << ' ';
  }
  std::cout << std::endl;
  return 0;
}
```

Container: exemple (vector 2D)

- Vecteur multi-dimensionnel

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

vector< vector<int> > vec; // peut necessiter un espace entre 1er > et 2e >

for (int i = 0; i < 10; i++) {
    vector<int> ligne; // cree une ligne vide
    for (int j = 0; j < 20; j++) {
        ligne.push_back(i * j); // ajoute un element
    }
    vec.push_back(ligne); // ajoute la ligne complete au vecteur
}
```

Iterator

- L'iterateur est une classe créée pour parcourir un conteneur

```
vector<int> mon_vecteur;
vector<int>::iterator it; //on cree un iterateur su vecteur de int

mon_vecteur.push_back(2); // ajout a la fin
mon_vecteur.push_back(3);
mon_vecteur.push_back(4);
mon_vecteur.push_front(1); // 1 2 3 4
for(it = mon_vecteur.begin() ; it != mon_vecteur.end() ; ++it)
   cout << (*it);

//suppression du dernier element
mon_vecteur.pop_back();

// suppression du 2e element
it = mon_vecteur.begin()+2;
mon_vecteur.remove(it);
```

Attention : pour parcourir un container marqué `const`, utiliser `const_iterator`.

Container: exmple liste

- Liste: conteneurs pour des sequences de valeurs. Implémentée par une liste doublement chainée (pas une zone mémoire contigue).

```
#include <iostream>
#include <list>
#include <vector>

int main () {

  std::list<int> ma_liste;
  std::list<int>::iterator it;

  // ajout en fin de liste
  for (int i=1; i<=5; ++i)
    ma_liste.push_back(i); // 1 2 3 4 5

  it = ma_liste.begin();
  ++it; // it pointe sur le nombre 2

  // insertion a un endroit donne
  ma_list.insert (it,10); // 1 10 2 3 4 5
  // it pointe toujours sur 2 ^
  // insertion de 2 valeurs 20
  mylist.insert (it,2,20); // 1 10 20 20 2 3 4 5
  // it pointe toujours sur 2 ^
```

Note sur les containers et les références

- Comme nous manipulons fréquement des références en C++, il vient naturellement l'idée des les stocker dans des structures de type container (comme `vector`).
- Mais un vector ne peut contenir que des éléments assignables. Ce n'est pas le cas des références.
- utilisation d'une classe `reference_wrapper` (c++11)

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

int main() {
  vector<reference_wrapper<int> > vect_of_refs;
  int a=1;
  int b=2;
  vect_of_refs.push_back(a);
  vect_of_refs.push_back(b);
  for (int & i : vect_of_refs)
    i = i*2;
  for (int & i : vect_of_refs)
    cout << i << "␣";
  cout << endl;

  return 0;
}
```

Table of Contents

Exceptions

- Objectif: signaler une erreur sans recourir à un retour de méthode/fonction.
- Principe :
  - un bloc d'instruction est signalé comme potentiellement source d'erreur : `try {}`
  - si erreur, on le signale en lançant un objet contenant l'informaton sur l'erreur : `throw objet;`
  - à l'endroit où l'on souhaite gérer l'erreur on attrape l'objet et on décrit la gestion de l'erreur : `catch (param) { }`

Exceptions

Premier essai :

```
int minimum(vector<int> &valeurs) {
  try {
      if (valeurs.size() == 0)
        throw string("Pas de minimum pour un vecteur vide!");
        // sort de la fonction
      else {
       mini = valeurs[0];
       for (auto e : valeurs)
         if e < mini
           mini = e;
       return mini;
      }
   }
   catch(string const& chaine) {
      cerr << chaine << endl;
   }
}

int main() {
 vector<int> vals = { 4,5,3,9,-1,2 };
 cout << minimum(vals) << endl;
 return 0;
}
```

```
$ c++ -std=c++11 exc.cpp
exc.cpp:20:1: warning: control may reach end of non-void function
[-Wreturn-type]
```

Exceptions

Plutôt récupérer l'exception que peut générer la fonction lors de l'appel de la fonction.

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

int minimum(vector<int> &valeurs) {
  if (valeurs.size() == 0)
       throw string("Pas de minimum pour un vecteur vide!");
  else {
       auto mini = valeurs[0];
       for (auto e : valeurs)
              if (e < mini)
                mini = e;
       return mini;
  }
}

int main() {
   vector<int> vals = { 4,5,3,9,-1,2 };
   try {
     cout << minimum(vals) << endl;
    }
    catch(string const& chaine) {
       cerr << chaine << endl;
    }
    return 0;
}
```