Rappels de C++

Les bases

Vincent Lemaire
vincent.lemaire@upmc.fr

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector<double> v;
    double d;

    while (cin >> d) {
        if (d == 0) break;
        v.push_back(d);
    }
    cout << "Lecture de " << v.size() << " elements" << endl;

    reverse(v.begin(),v.end());

    cout << "En ordre inverse" << endl;
    for (int i = 0; i<v.size(); ++i) cout << v[i] << endl;

    return 0;
}
```

Types « fondamentaux » en C et C++

- Entiers : **char**, **short**, **long** (avec modificateur **unsigned**)
  int = **short** (16 bits) ou **long** (32 bits) en fonction de l'architecture
  En C, le résultat d'une expression booléenne (un test) est un entier :
  - non nul, si l'expression est vraie
  - nul sinon.

  En C++, le résultat d'une expression booléenne est de type **bool** :
  - **true** si l'expression est vraie
  - **false** sinon.

- Réels : **float**, **double**, **long double**
  Pour le calcul scientifique, toujours utiliser des **double** (64 bits).

- Types dérivés
  - Tableaux []
  - Fonctions ()
  - Pointeurs *
  - Structures **struct**
  - Unions **union**

Types plus évolués en C++

La libraire standard du C++, la STL (Standard Template Library), définit des nouveaux types :

- `string` : pour manipuler aisément les chaînes de caractères
- `vector<T>` : pour les structures vectorielles
- `list<T>` : pour les listes chaînées

Les types `vector<T>` et `list<T>` sont génériques, c'est à dire qu'ils sont construits pour un type `T` connu à la compilation. Par exemple pour déclarer un `vector` de `list` de **`double`** on utilise

```
vector< list<double> > v;
```

On détaillera l'utilisation de la STL dans le 3ème rappel.

Variables globales, locales, constantes, statiques, externes...

- Une variable globale est déclarée en dehors de toute fonction et est accessible/modifiable dans toute fonction. A ne pas utiliser !
- Une variable locale est déclarée dans un bloc (entre accolades `{}`) et n'est accessible que dans ce bloc.
- Les modificateurs **`const`**, **`static`**, **`extern`** peuvent modifiés le comportement/la portée des variables.

```
#include <iostream>
int n = 2;
const int p = 3;

bool f(int k) {
    int r = k % (n + p);
    return (r == 0);
};

int main() {
    if (f(5)) ++n;
    return 0;
}
```

Surcharge

Prototype d'une fonction défini par :

- type de sortie
- nom de la fonction
- le nombre et le type de ses arguments

Signature d'une fonction définie par :

- nom de la fonction
- le nombre et le type de ses arguments

En C++, la signature (et non le nom !) permet de distinguer 2 fonctions.
On peut donc assigner à deux fonctions différentes le même nom.

```
int puissance(int n, int m) {
    // code pour n * ... * n
};
double puissance(double x, int n) {
    // code pour x * ... * x
};
double puissance(double x, double a) {
    // code pour exp(a * log(x))
};
```

Passage d'arguments par copie

Il est important de savoir qu'une fonction ne modifie pas la valeur de ses arguments.

Par exemple si on considère le programme suivant

```
#include <iostream>

void echange(double x, double y) {
    double tmp = x;
    x = y;
    y = tmp;
};

int main() {
    double a = 3, b = 5;
    echange(a, b);
    std::cout << "a = " << a << std::endl;
    std::cout << "b = " << b << std::endl;
    return 0;
}
```

Valeurs par défaut

Les derniers arguments peuvent prendre des valeurs par défaut (préciser dans le prototype) *à la déclaration* de la fonction.

```
double puissance(double x, unsigned int n = 2, bool affiche = true) {
    double y = 1;
    while (n-- > 0) y *= x;
    if (affiche) cout << y;
    return y;
}
```

Les appels suivants sont tous corrects

```
puissance(3.14, 5, false);
puissance(3.14, 5);
puissance(3.14);
```

Inlining

Permet d'optimiser l'appel de fonctions : approprier pour de petites fonctions appelées très souvent, comme c'est le cas en POO.

Syntaxe : mot-clé **`inline`** devant le prototype de la fonction

Pointeurs : déclaration

Un pointeur permet de manipuler l'adresse mémoire d'une variable ou d'une fonction.
Très utile en C mais dangereux : accès direct à la mémoire sans aucune vérification.
En C++ il est préférable d'utiliser les références et des « smart pointer ».

Syntaxe :

- pointeur sur variable :

```
type * variable
```

- pointeur sur fonction :

```
type (*fct)(type1 arg1, ..., typeN argN)
```

Pointeurs particuliers (très important à connaître)

- pointeur générique : **`void *`**
- pointeur trivial : `NULL`

Pointeurs : initialisation

Deux façons d'intialiser un pointeur sur variable.

- en récupérant l'adresse d'une variable : on utilise l'opérateur `&`

```
double x = 3;
double * p = &x;
```

- en démandant la création d'une zone mémoire et en récupérant cette adresse : on utilise l'opérateur **`new`** (en C la fonction `malloc`)

```
double * a = new double;
int * b = new int[5];
```

après utilisation on doit *obligatoirement* libérer la mémoire allouée

```
delete a;
delete [] b;
```

Un pointeur sur fonction ne peut s'initialiser qu'en récupérant l'adresse d'une fonction existente.

Pointeurs : utilisation

Pour récupérer la valeur pointée on utilise l'opérateur de déréférencement `*`.

```
cout << *p << endl;
*a = 3.14;
*b = 1.2;
```

Il est possible d'effectuer les opérations arithmétiques suivantes sur les pointeurs :

- addition d'un pointeur et d'un entier
- soustraction entre 2 pointeurs

Ainsi pour accéder à la 2ème case mémoire réservée en `b`, on effectue `*(b+1)`.
Pour remplir (initialiser) la zone mémoire `b`

```
for (int * p = b; p != b+5; ++p)
    *p = ...
```

L'opérateur d'indexation `[]` permet une écriture plus aisée

`p[k]` est équivalent à `*(p+k)`

Pointeurs et tableaux

- pointeur sur valeur constante :

```
const type * nom_pointeur          (historique)
type const * nom_pointeur          (recommandée)
```

- pointeur constant :

```
type * const nom_pointeur
```

- pointeur constant sur valeur constante :

```
const type * const nom_pointeur          (historique)
type const * const nom_pointeur          (recommandée)
```

Dans l'écriture recommandée le **const** s'applique toujours sur ce qui vient directement à sa gauche.

Un tableau statique en C ou C++ se comporte exactement comme un pointeur constant.

Références : déclaration

La référence existe en C++ et pas en C.
C'est une autre façon de manipuler les adresses des objets (variables, fonctions) en mémoire.

Une variable de type référence est une adresse qui

- à l'initialisation : récupère l'adresse d'un **objet existant**
- durant sa vie : se comporte exactement comme l'objet référencé

Syntaxe :

- `type & variable`

Exemple :

```
int k = 2, j = 3;
int & r = k;    // r est une reference sur k
r = j;
```

Utilisation des références

Les références sont très utiles en tant que

- paramètres (constants ou non) de fonctions
  Exemple de la fonction `swap`

- type de sortie d'une fonction : une fonction renvoyant une référence peut se retrouver à gauche d'une affectation (*l-value*).

  Exemple : supposons que le tableau `prenoms` de `N string` soit déjà initialisé, et qu'on veuille initialiser un tableau de `N` entiers à l'aide d'une fonction `note` de la façon suivante :

```
string prenoms[N];
int notes[N];

note("Albert") = 15;
note("Maurice") = 18;
...
```

Retour sur le type `struct` en C

Le type **struct** permet

- de regrouper dans une même variable un ensemble de variables de types différents,
- de clarifier un programme (beaucoup plus lisible),
- d'implémenter les structures récursives : liste chaînée, arbres...

Exemple : écriture d'un programme qui manipule des entiers p-adiques

$$n = \sum_{k=0}^{r-1} a_k p^k, \quad 0 \leqslant a_k < p \quad \text{et} \quad a_{r-1} \neq 0, \quad \text{et} \quad a_r = 0.$$

Il est naturel de regrouper les entiers p, r, et $(a_k)_{0 \leqslant k \leqslant r-1}$ dans un nouveau type :

```
struct p_adic {
    unsigned p, r;
    unsigned * coeff;
};
```

Exemple sur les p-adiques (toujours en C)

Une fonction d'initialisation pour une variable de type **struct** p_adic (ou simplement p_adic depuis le C 99) s'écrit :

```
void Initialisation(unsigned p, unsigned r, unsigned a[], p_adic * obj) {
    int k;
    obj->p = p;
    obj->r = r;
    obj->coeff = malloc(r * sizeof(unsigned));
    for (k = 0; k < r; ++k) obj->coeff[k] = a[k];
};
```

Pour utiliser un p_adic, on doit alors le déclarer puis l'initialiser :

```
unsigned a[] = {1, 2, 14, 7};
p_adic n;
Initialisation(19, 4, a, &n);
```

de même, on pourrait définir une seconde fonction d'initialisation qui définirait r et $(a_k)_{0 \leqslant k \leqslant r-1}$ à partir de p et n

```
p_adic n;
Initialisation_bis(19, 45, &n);
```

Ecriture en C++, le constructeur

Le type **struct** est considérablement enrichi en C++ :

- il peut contenir des fonctions
- les variables/fonctions membres peuvent être privées ou publiques
- un type **struct** peut « hériter » d'un autre type **struct**

La fonction d'initialisation que l'on a codé en C fait maintenant partie la structure.
C'est un constructeur i.e. une fonction membre qui a le même nom que la structure et qui ne renvoie aucun argument.

```
struct p_adic {
    p_adic(unsigned p, unsigned r, unsigned a[]) {
        this->p = p;
        this->r = r;
        this->coeff = new unsigned[r];
        for (int k = 0; k < r; ++k) this->coeff[k] = a[k];
    };
    unsigned p, r;
    unsigned * coeff;
};
```

Ecriture en C++, le constructeur -2-

Une deuxième écriture possible est de déclarer la fonction membre dans la structure et de la définir en-dehors (en utilisant l'opérateur de résolution de portée ::)

```
struct p_adic {
    p_adic(unsigned p, unsigned r, unsigned a[]);
    unsigned p, r;
    unsigned * coeff;
};

p_adic::p_adic(unsigned p, unsigned r, unsigned a[]) {
    this->p = p;
    this->r = r;
    this->coeff = new unsigned[r];
    for (int k = 0; k < r; ++k) this->coeff[k] = a[k];
};
```

L'appel du constructeur peut se faire de différentes façons :

```
unsigned a[] = { 1, 2, 14, 7 };
p_adic n(19, 4, a);
p_adic m = p_adic(19, 4, a);
p_adic * p = new p_adic(19, 4, a);
```

Constructeurs, liste d'initialisation

La liste d'initialisation permet une écriture simplifiée du constructeur.
Elle se déclare entre le prototype et la définition de la fonction en utilisant le symbole « : ».

```
struct p_adic {
    p_adic(unsigned p, unsigned r, unsigned a[])
      : p(p), r(r), coeff(new unsigned[r]) {
        for (int k = 0; k < r; ++k) this->coeff[k] = a[k];
    };
    unsigned p, r;
    unsigned * coeff;
};
```

A suivre...