Ch. 6 - Concepts de base en C++

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

11 octobre 2021

Table des matières

Introduction

Comparaison entre C et C++

- Au début, C++ était juste « C avec des classes »
- Des concepts classiques d'autres langages étaient aussi inclus
  - Surdéfinition
  - Exceptions
  - Manipulation facile de chaînes de caractères
- Au fil du temps, le langage s'est étoffé en fonction des besoins
  - Surcharge d'opérateurs
  - Templates
  - Conversions définies par l'utilisateur
- Une très grande partie des concepts C reste valide en C++

C++ : le programme minimal

- Fichier `hello-world.cpp`

```
1 #include <iostream>
2
3 int main() //variantes avec arguments en ligne de commande
4 {
5   std::cout << "Hello World!" << std::endl;
6 }
```

- `#include` permet d'importer les fichiers nécessaires
- La fonction `int main()` est l'unique point d'entrée du programme
- `std::cout` est un flux de sortie permettant d'écrire en console
  - On peut spécifier des « manipulateurs » pour formater l'affichage
  - `std::endl` est un manipulateur insérant une fin de ligne
- Syntaxe de commentaires comme en `Java`
- Compilation avec `g++ -o sortie monficher.cpp`

Concepts en vrac

Manipulation

- En C++, il existe une classe `string`
  - On peut affecter un littéral à un `string`
  - `string s = "Hello World!"`
- Taille : `size`, `capacity`, `reserve`
  - Allocations successives exponentiellement plus larges
  - Possibilité de spécifier un allocateur
- Accès : `[]`, `find`
- Modification : `+=`, `+`, `append`, `substr`
- Comparaison lexicographique : `==`, `!=`, `<`, `>`, `<=`, `>=`
- Parsing : `stoi`, `stoul`, `stof`, `stod`, `strtok`
  - L'interprétation stoppe sur la première espace
- Validation de caractères : `isalnum`, `isalpha`, `isdigit`, `islower`, `isupper`

Illustration

- Fichier `string.cpp`

```
1  int main()
2  {
3    string s1 = "Hello ";
4    string s2 = "World";
5
6    for(long i = 0; i < s1.length(); i++)
7      cout << s1[i] << endl;
8    for(long i = 0; i < s2.length(); i++)
9      cout << s2[i] << endl;
10   cout << endl;
11
12   string s3 = s1 + s2;
13   cout << s3 << endl;
14   s1 += s2;
15   cout << s1 << endl;
16
17   string s4;
18   for(int i = 0; i < 20; i++)
19   {
20     s4 += 'a';
21     cout << "Length = " << s4.length() << ", capacity = " << s4.capacity() << endl;
22   }
23 }
```

Peu de différences

- Comme en C, le standard fournit peu de garanties
- Pas de garantie de tailles
- Uniquement des garanties de bornes
- Les conversions implicites sur les types numériques, `char` et `bool` restent valides en C++
  - Hygiène : caster explicitement
  - Mécanisme dédié détaillé au chapitre ? ?

Bornes

- En C++ : fichier `bounds.cpp`
- On imprime avec `std::cout`

```cpp
#include <limits>
#include <iostream>

int main()
{
    std::cout << "type\tlowest\t\thighest\n";
    std::cout << "int\t"
              << std::numeric_limits<int>::lowest() << '\t'
              << std::numeric_limits<int>::max() << '\n';
    std::cout << "float\t"
              << std::numeric_limits<float>::lowest() << '\t'
              << std::numeric_limits<float>::max() << '\n';
    std::cout << "double\t"
              << std::numeric_limits<double>::lowest() << '\t'
              << std::numeric_limits<double>::max() << '\n';
}
```

Initialisation en C++

- Plusieurs types d'initialisation
  - Initialisation directe (on fournit des paramètres)
  - Initialisation par copie
  - Liste d'initialisation
  - Initialisation par référence
- Initialisation d'objets : cf. Ch. 8, 12, 14
- En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation
- Initialisation de types de base
  - avec `= ...` : affectation
  - avec `( ... )` : similaire à `=`
  - avec `{ ... }` et `={ ... }` : conversions dégradantes non autorisées

Exemple

- Fichier `init.cpp`

```
1  int main()
2  {
3      int i;      //indeterminate
4      int j = 2;
5      //int k {2}; //int k = {2};
6
7      //int k {2.1}; //int k = {2.1};
8
9      unsigned char * p; //undeterminate address
10     //*p = 3;
11     unsigned char * u = 2; //still up to no good
12     //*u = 3;
13 }
```

Introduction : C++

- Flux d'entrée / sortie standards
  - Sortie standard : `cout`
  - Sortie erreur standard : `cerr`
  - Fin de ligne : `endl`
- Lecture et écriture via '«'
  - Opérateurs d'injection de flux défini pour les types de base
  - Possibilité de surcharge pour le autres
- Chaînage possible
  - Associatif de gauche à droite
  - Un parenthésage peut être nécessaire
  - Résultat après injection : même flot que le premier opérande
- `iomanip.h` permet de modifier le formatage
  - `setw` : longueur de la prochaine injection
  - `setfill`, `left`, `right`, `internal` etc. : remplissage de la prochaine injection
  - `dec`, `oct`, `hex` : change la base entière
  - `setprecision` : précision de l'affichage

Exemple

- Fichier `cout.cpp`

```
1  const long double PI = atan(1) * 4;
2
3  int main()
4  {
5     cout << left << setw(16)
6          << "default pi" << " : " << PI << endl
7          << left << setw(16)
8          << "10-digits pi" << " : " << std::setprecision(10) << PI << endl
9          << left << setw(16)
10               << "max-digits pi" << " : "
11               << setprecision(numeric_limits<long double>::digits10 + 1)
12               << PI << endl;
13
14    int n = 42;
15    cout << "42 - base 8  : "<< oct << n << endl;
16    cout << "42 - base 10 : "<< dec << n << endl;
17    cout << "42 - base 16 : "<< hex << n << endl;
18 }
```

Lecture en C++ : std::cin

- Opérateur '»' défini pour les types de base
  - char : obtient le code du caractère (setlocale change le charset)
  - Entiers et flottants : acceptés quelle que soit la notation « autorisée », avec ou sans suffixe, décimale ou exponentielle, etc.
  - bool : 0 ou 1
- Possibilité de surcharge pour les autres
- Chaînage possible
  - Opérateur '»' associatif de gauche à droite

Exemple

```
int i; cin >> i;
```

- Même fonctionnement par buffer que scanf
  - Hygiène : lire une ligne ligne avec getline et parser

Exemple

- Fichier `cin.cpp`

```
1  int main()
2  {
3    cout << "Tapez un entier" << endl;
4    int i;
5    cin >> i;
6    cout << "Vous avez tapé " << i << endl;
7
8    cout << "Tapez un flottant" << endl;
9    double d;
10   cin >> d;
11   cout << "Vous avez tapé " << d << endl;
12
13   cout << "Tapez deux entier" << endl;
14   int j, k;
15   cin >> j >> k;
16   cout << "Vous avez tapé " << j << " et " << k << endl;
17
18   cout << "Tapez une chaîne de caractères" << endl;
19   string s;
20   cin >> s;
21   cout << "Vous avez tapé " << s << endl;
22 }
```

Idée

- Permet de déterminer automatiquement le type d'une variable *à la compilation*
- Mots-clés `auto` et `decltype`
- La déduction est effectuée à partir de l'initialisation
  - `auto i = 2; //ok : int`
  - `decltype(i + 2.3) f; //ok : double`
  - `auto j; //ko`
- On peut accompagner `auto` de `const`, `&`, etc.
  - Permet de raffiner le type si nécessaire
- Pratique pour
  - raccourcir des « noms de types trop longs » sans alias
  - nommer un type déterminable à la compilation (mais inconnu du programmeur)

Exemple

- Fichier `auto.cpp`

```
1  auto f(int i) {
2    switch(i) {
3      case 1 : return sqrt(i);
4      case 2 : return cos(i);
5      default : return i + 0.;
6    }
7  }
8
9  int main() {
10   auto a = 3 + 4;
11   cout << a << " of type " << typeid(a).name() << endl;
12
13   decltype(a) b;
14   b = 7.2;
15   cout << b << " of type " << typeid(b).name() << endl;
16
17   for(int i = 0; i <=3; i++)
18     cout << f(i) << " of type " << typeid(f(i)).name() << endl;
19
20   auto l = {1, 2};
21   cout << "Type of l : " << typeid(l).name() << endl;
22 }
```

- Lancer avec `./auto | c++filt -t`

Namespaces

- Permet d'organiser le code en modules aux fonctionnalités similaires
- Se rapproche de la notion de package en `Java`
- Non restreint à un répertoire
- Mention explicite du namespace à l'utilisation
  - Sauf si `using namespace`
- Utilisation de `::` (opérateur de résolution de portée)

Hygiène de programmation

- Pas de `using namespace std;` dans un code réutilisable

Exemple

- Fichiers `dist.h`, `dist.cpp` et `dist-main.cpp`

```
1  namespace math {
2    double dist(double x1, double y1, double x2, double y2);
3  }
```

```
1  #include "dist.h"
2  #include <cmath>
3
4  namespace math {
5    double dist(double x1, double y1, double x2, double y2) {
6      return std::sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
7    }
8  }
```

```
1  #include <iostream>
2  #include "dist.h"
3
4  using namespace std;
5  //using namespace math;
6
7  int main() {
8    //cout << dist(0,0,1,1) << endl;
9    cout << math::dist(0,0,1,1) << endl;
10 }
```

Les alias de type

- Nom référant un type défini précédemment
- Utile pour abréger certains noms
- En C, on utilisais `typedef`

Syntaxe en C++

- `using Alias = Type;`

- `using` est compatible avec les templates, `typedef` non

Exemple

```
1  struct point
2  {
3    double x, y;
4  };
5
6  typedef struct point point;
7
8  point p;
```

```
1  using two_points = std::pair<point,point>;
2
3  template<class T> using ptr = T*;
4  ptr<point> = &p;
```

Principe

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 12)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

Exemple

- Fichier `enum.cpp`

```
1  enum class Color { red, green = 20, blue };
2
3  enum class altitude : char
4  {
5       high='h',
6       low='l'
7  };
8
9  int main()
10 {
11     Color r = Color::blue;
12     switch(r)
13     {
14         case Color::red   : cout << "red" << endl;   break;
15         case Color::green: cout << "green" << endl; break;
16         case Color::blue  : cout << "blue" << endl;  break;
17     }
18
19     // int n = r;
20     int n = static_cast<int>(r); // OK, n = 21
21 }
```

Constantes et constexpr

Overview

- Le mot clé `const` a la même signification en C++ qu'en C
- Il peut être également utilisé sur des fonctions membres
- Le cas échéant, on annonce qu'appeler cette fonction ne modifie pas `this`
- Les fonctions membres non constantes ne peuvent pas être appelées sur des objets constants
  - Mais on peut appeler des fonctions membres constantes sur des objets non constants
- Un autre concept existe en C++ : les `constexpr`

Intuition

- constexpr = évaluable à la compilation

Illustration

- Fichier `const.cpp`

```
1  class A { //définition d'une classe
2       public:
3              void f() const {} //fonction membre constante
4              void g() {}
5  };
6
7  int main() {
8      int i = 2;
9      const int ci = i;
10
11     A a; //allocation par défaut de A (cf. Ch. 8)
12     const A ca;
13
14     i++; //ok
15     //ci++; //ko
16
17     a.f(); //ok
18     a.g(); //ok
19     ca.f(); //ok
20     //ca.g(); //ko
21 }
```

Expressions constexpr

- Variable, fonction ou constructeur *évaluable à la compilation*
- Implique `const`
- Utilisation du mot-clé `constexpr`
- Offre de grandes performances *à l'exécution*
  - Certains calculs sont effectués *une fois* à la compilation

Variable constexpr

Contraintes

1. Doit être un littéral
2. Doit être immédiatement assigné ou construit
   - Pas de déclaration sans assignation
   - Les paramètres doivent être des littéraux, des constantes ou fonctions constexpr
   - Le constructeur doit être constexpr

- Les contraintes ci-dessus offrent une possibilité d'évaluation et d'assignation de la variable à la compilation

Fonction constexpr

Contraintes

1. Ne peut pas être polymorphique
2. Son type de retour doit être un littéral
3. Les paramètres doivent être des littéraux, des constantes ou fonctions constexpr
4. Le corps ne peut pas contenir d'instruction non constexpr
5. Pas de try / catch
6. Pas de définition de variable non littérale
7. Etc.

Exemple

- Fichier `constexpr.cpp`
- Ne prêtez pas attention à la `struct constN`
  - Affichage en compile-time

```
1  constexpr double PI = atan(1) * 4;
2
3  constexpr int factorial(int n)//c++11 : recursion, one return statement
4  {
5      return n <= 1 ? 1 : n * factorial(n - 1);
6  }
7
8  constexpr long long int test(long long int n)//c++14
9  {
10   int i = n;
11   while(i >= 0)
12     i--;
13   return i;
14 }
```

- Un appel `test(9999999)` prend du temps à compiler

Références

Overview

- Dans les deux cas, les pointeurs et références sont utilisés comme des étiquettes pour désigner d'autres objets
- On les utilise
  - si l'on veut propager en écriture des effets de bords (modifier un paramètre de fonction)
  - si on ne veut pas que des copies implicites de données soient effectuées
- Les références sont un concept C++ uniquement
- Il y a des cas où l'on ne peut pas se passer de pointeurs / références
- Toutefois, ces concepts ne sont *pas* les mêmes
  - Syntaxe et manipulation différentes
  - On peut faire des écritures avec des pointeurs invalides avec des références

Concept de référence

- Alias vers une variable existante
  - Ne peut pas être « nul » (doit exister)
- Se manipule comme une variable
- Conversions implicites possibles à l'affectation

Hygiène de programmation

- Utilisez les références quand vous pouvez, les pointeurs quand vous devez

- Les références sont constantes
  - Réaffecter une référence réaffecte l'objet référencé, pas la référence

Exemple

- Fichier `ref.cpp`

```
1  int main()
2  {
3    int i = 2;
4    int & ri = i;
5    //int & rj;
6    cout << i << " " << ri << endl;
7
8    i++;
9    cout << i << " " << ri << endl;
10   ri = 5;
11   cout << i << " " << ri << endl;
12
13   int j = 8;
14   ri = j;
15
16   cout << i << " " << ri << endl;
17 }
```

- En pratique, ces références sont des références de *lvalue*
- Il existe d'autres types de référence (cf. Ch. 13)

Les lvalue

Intuition

lvalue

- « Valeur qui peut apparaître à gauche d'un opérateur d'affectation »

- Définition insuffisante

Remarque

- `const int a = 2; //ok`
- `a = 3; //ko`

Notion de lvalue

Intuition

```
double x, y, a, b;
y = a*x + b; //ok
a*x + b = y; //ko
(x + 1) = 4; //ko
```

- Le premier opérande *doit* référencer un emplacement mémoire non temporaire
  - Autres langages : variable, en C / C++, pas assez précis
- Contrainte nécessaire à plusieurs endroits dans le langage
- Propriété très importante pour les *expressions*
- Ce qui n'est pas une `lvalue` est une *rvalue* (immédiats, temporaires, anonymes, etc.)

Conversions entre lvalues et rvalues

- Souvent, les opérateurs (et fonctions) requièrent des arguments rvalues

Exemple

- `int i = 1;`
- `int j = 2;`
- `int k = i + j;`

1. `i` et `j` sont des lvalue
2. `+` requiert des rvalue
3. `i` et `j` sont convertis en rvalue
4. Une rvalue est retournée

Règles de conversions

Conversion lvalue vers rvalue

- Toutes les lvalues qui ne sont pas des tableaux, des fonctions et des types incomplets peuvent être convertis en rvalues

Conversion rvalue vers lvalue

- Impossible implicitement
- Le résultat d'un opérateur (rvalue) peut être explicitement affecté en une lvalue

- On peut produire des lvalue à partir de rvalue explicitement
  - Le déférencement prend une rvalue et produit une lvalue
  - L'opérateur `&` prend une lvalue et produit une rvalue

Exemple

- Fichier `rvalue-conv.cpp`

```
1  int main()
2  {
3    int a[] = {1, 2};
4    int* pt = &a[0];
5    *(pt + 1) = 10;   //OK : p + 1 is an rvalue, but *(p + 1) is an lvalue
6
7    //taking adress
8    int i = 10;
9    //int* pti = &(i + 1);    // KO : lvalue required
10   int* pti = &i;            // OK: i is an lvalue
11   //&i = 20;                // KO : lvalue required
12
13   //reference making
14   //std::string& sref = std::string(); //KO : non const-ref init from rvalue
15 }
```

Références de lvalue

- Fichier lvalue-ref.cpp

```
1  int n = 5;
2  int& truc = n;
3
4  int& brol() { return n; }
5
6  int main() {
7    cout << brol() << endl;
8    brol() = 10;
9    cout << brol() << endl;
10
11   truc = 15;
12   cout << brol() << endl;
13
14   //int & i = 2;
15 }
```

- n est une lvalue, brol() aussi
- Les références de lvalue sont des lvalue
  - Utile pour l'opérateur []
  - v[10] = 42;

La motivation des contraintes

- On ne veut pas pouvoir réaffecter un temporaire / immédiat

```
1  int a = 42;
2  int b = 43;
3
4   // a and b are both lvalues:
5  a = b; // ok
6  b = a; // ok
7  a = a * b; // ok
8
9  // a * b is an rvalue:
10 int c = a * b; // ok, rvalue on right hand side of assignment
11 a * b = 42; // error, rvalue on left hand side of assignment
12
13 //2 is a rvalue
14 int d = 2;
15 int & d = 2; //error
16 const int& e = 2; //ok : you are allowed to bind a const lvalue to a rvalue
```

- Il existe néanmoins des références de rvalue (cf. Ch. 13)
- Le fait qu'une expression soit une rvalue ou une lvalue est appelé la *value category*

Pointeurs intelligents

Hygiène de new

- En C, utiliser l'allocation dynamique est parfois indispensable

Je veux faire un new en C++

. NON !

- Il y a d'autres mécanismes
  - Pointeurs intelligents
  - Références de rvalue (cf. Ch. 13)

Pas de new en C++

- Je veux quand même faire un new en C++
  1. J'ai l'habitude en Java
     - En Java, on ne peut pas écrire sur la pile
     - En Java, il y a un garbage collector (compromis efficacité / sûreté)
  2. Je veux éviter une copie de paramètre de fonction
     - Utilise le passage par référence
  3. Je veux qu'une fonction modifie ses paramètres (effet de bord)
     - Utilise le passage par référence
  4. Je veux éviter une copie de retour de fonction
     - Utilise les références de rvalue
  5. Je veux avoir un attribut sans avoir à le recopier par le constructeur
     - Utilise une référence
  6. Je veux activer le polymorphisme
     - Utilise une référence
  7. Je veux allouer qqch de « gros »
     - Utilise les pointeurs intelligents

Moralité

Hygiène de programmation en C++

- Utilisez des références quand vous pouvez
- Utilisez des pointeurs quand vous devez

- Exemple d'obligation : résoudre une dépendance cyclique

Si on veut *vraiment* utiliser new

- Encapsulation dans des *pointeurs intelligents*

Nécessité en allocation dynamique

- Quand on sort d'un scope, il faut décider quoi faire de la mémoire allouée
- Utilisation du patron de classe, `unique_ptr`, `shared_ptr` et `weak_ptr`.
- Paramétré par le type de la variable dynamique à encapsuler
- Comportements « similaires » aux pointeurs / références, en classe automatique, implémenté via un patron de classe et la surcharge d'opérateurs (cf. Ch. 12).
- Chaque patron définit comment la mémoire doit être gérée à la destruction (automatique) du pointeur intelligent.
  - On détruit les données ?
  - On détruit les données si plus rien ne pointe dessus ?
  - On ne détruit rien ?
- Implémenté en comptant le nombre de références dans le constructeur à l'aide d'une variable statique.
- Inclure `memory.h`

Pointeurs intelligents

- Quand un pointeur possédant un objet est détruit, il faut définir comment libérer la mémoire
- Trois types de pointeurs intelligents « principaux »
  1. `unique_ptr` : pointeur intelligent qui n'autorise qu'une possession unique de l'objet.
     - Copier et affecter le pointeur provoque une erreur de compilation.
     - Quand le pointeur est détruit, la donnée est détruite.
  2. `shared_ptr` : pointeur intelligent qui autorise des possessions multiples d'un même objet.
     - Les données pointées sont détruites si plus rien ne pointe dessus.
     - Le dernier pointeur possédant les données est détruit
  3. `weak_ptr` : pointeur intelligent qui ne « possède pas » d'objet.
     - Doit être converti en `shared_ptr` pour accéder l'objet (`lock()`).
     - Pratique pour une possession temporaire, quand l'objet peut être détruit n'importe quand par un facteur extérieur.
- Instanciation « à la volée » avec `new` ou via `make_unique`, `make_shared`

Exemple unique_ptr

- Fichier unique.cpp

```
1  int main()
2  {
3    int i = 2;
4    int * pti = &i;
5    unique_ptr<int> u1(&i);
6    {
7      //unique_ptr<int> u2(&i); //bad idea
8      //unique_ptr<int> u2 = u1; //compile error
9      unique_ptr<int> u2 = move(u1); //u2 owns, u1 invalid
10   }
11   cout << *pti << endl;
12   u1.reset();//deletes memory (why ?!)
13   cout << i << endl; //seg fault
14 }
```

Exemple shared_ptr

- Fichier shared.cpp

```
1  int main()
2  {
3    shared_ptr<int> p1(new int(5));
4    weak_ptr<int> wp1 = p1; //p1 owns the memory.
5
6    {
7      shared_ptr<int> p2 = wp1.lock(); //Now p1 and p2 own the memory.
8      if(p2) //check if the memory still exists!
9      {
10         cout << "if p2" << endl;
11     }
12   } //p2 is destroyed. Memory is owned by p1.
13
14   p1.reset(); //Memory is deleted.
15
16   shared_ptr<int> p3 = wp1.lock(); //Memory is gone, so we get an empty shared_ptr.
17   if(p3)
18   {
19     cout << "if p3" << endl;
20   }
21 }
```

Exemple de fuite mémoire : initialisation (1/2)

```
1  int f(shared_ptr<int> i, int j);
2  int g();
3
4  f(shared_ptr<int> (new int (42)), g());
```

- Ordre d'appel
  1. Allocation dynamique de l'entier 42
  2. Création du `shared_ptr<int>`
  3. Appel de la fonction *g*
  4. Appel de la fonction *f*

Exemple de fuite mémoire : initialisation (2/2)

Ordre d'appel

- 3 peut avoir lieu *avant* 1 et 2, et peut en particulier être appelé entre 1 et 2

Problème potentiel : *g* lance une exception

- Le `shared_ptr` n'as pas encore eu le temps de posséder la mémoire
- Il ne peut pas la libérer
- Fuite mémoire

Solution

- Première idée

```
1 int f(shared_ptr<int> i, int j);
2 int g();
3
4 shared_ptr<int> si (new int (42));
5 f(si, g());
```

- Meilleure idée

```
1 int f(shared_ptr<int> i, int j);
2 int g();
3 f(make_shared<int>(42), g());
```

- Moralité : ne pas faire de `new`

Exemple de fuite mémoire : cycle

- Fichier `cycle.cpp`

```
1  class A
2  {
3    public:
4        shared_ptr<B> ptB;
5  };
6
7  class B
8  {
9      public:
10       shared_ptr<A> ptA;
11 };
12
13 int main()
14 {
15   shared_ptr<A> a(new A);
16   shared_ptr<B> b(new B);
17   cout << a.use_count() << ", " << b.use_count() << endl;
18   a->ptB = b;
19   cout << a.use_count() << ", " << b.use_count() << endl;
20   b->ptA = a;
21   cout << a.use_count() << ", " << b.use_count() << endl;
22   a.reset();
23   b.reset();
24   cout << a.use_count() << ", " << b.use_count() << endl;
25 }
```

Sournoiserie

- Affichage à la ligne 17 : `1 1`
- Affichage à la ligne 19 : `1 2`
  - `b` et `ptB` pointent vers l'objet de type `B`
- Affichage à la ligne 21 : `2 2`
- Affichage à la ligne 24 : `0 0`

Zombie

- `ptB` dans `A` fait survivre `B`
- `ptA` dans `B` fait survivre `A`

Solution

- Utiliser `weak_ptr`

Solution

- Fichier `cycle-sol.cpp`

```
1  class A
2  {
3    public:
4        shared_ptr<B> ptB;
5  };
6
7  class B
8  {
9      public:
10       weak_ptr<A> ptA;
11 };
12
13 int main()
14 {
15   shared_ptr<A> a(new A);
16   shared_ptr<B> b(new B);
17   cout << a.use_count() << ", " << b.use_count() << endl;
18   a->ptB = b;
19   cout << a.use_count() << ", " << b.use_count() << endl;
20   b->ptA = a;
21   cout << a.use_count() << ", " << b.use_count() << endl;
22   a.reset();
23   b.reset();
24   cout << a.use_count() << ", " << b.use_count() << endl;
25 }
```

Exemple complet

- Liste simplement chaînée
  1. Fichier `linkedlist-new.cpp`
  2. Fichier `linkedlist-smart.cpp`
- Classe interne de nœud
  - Une donnée, un élément suivant
  - Difficile d'utiliser des références
    - Initialement, la liste est vide (tête et queue)
    - La queue a toujours un successeur vide
- Remarque : aucune gestion explicite de la mémoire avec les pointeurs intelligents
  - Pas de `new`, `delete`
  - Pas de destructeur
- En `C`, on n'a pas le choix
  - `malloc` et `free` manuels nécessaires
  - Fichier `linkedlist-c.c`

Remarque

Hygiène de programmation C++

1. Ne faites pas de `new`
   1. Utilisez des références
   2. Utilisez des pointeurs intelligents
2. Utilisez `make_shared` et `make_unique`
   1. Évite de faire un `new` et de gérer la mémoire « à la main »
   2. Évite de créer des `shared_ptr` temporaires

- Seul cas justifiable
  - `new` et `delete` sont au sein du même bloc
  - Les instructions entre les deux sont `noexcept`
  - `new` est amené à allouer une grande quantité de mémoire
    - Par exemple, `T * array = new T[n * n];`