TD 1 : Découverte de C++

Yann Régis-Gianas (yrg@pps.jussieu.fr)
POCA - Master II Pro - Université Paris 7

7 octobre 2008

Dans ce TD, nous allons découvrir C++ par la pratique, c'est-à-dire en programmant. Dans l'esprit du cours, nous partons des traits objets classiques o□erts par C++ par le biais de son système de types statique basé sur les classes. Ensuite, nous rentrerons dans les spécificités du langage, sans pour autant nous perdre dans les détails. Nous approfondirons en cours les quelques points techniques importants à connaître pour programmer correctement en C++.

Table des matières

1 Préliminaires

1.1 Références

Bibliographie Le langage C++ étant normalisé par l'ISO, l'unique document qui fait référence est la spécification ISO/IEC 14882 :2003. Il s'agit d'un imposant volume rassemblant, de façon informelle, l'ensemble des règles qui permettent de déterminer le comportement de tout programme C++.

Bjarne Stroustrup, le concepteur de C++, a écrit un livre plus accessible que la norme ISO :

The C++ Programming Language (Third Edition and Special Edition)
Bjarne Stroustrup
Addison-Wesley, ISBN 0-201-88954-4 et 0-201-70073-5.

On pourra le compléter par le livre suivant :

Effective C++ : 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)
Scott Meyers
Addison-Wesley Professional Computing Series, ISBN 0321334876 et 978-0321334879

Ressources en ligne

- ➲ Pour des explications générales concernant le langage :
  - ✓ http://www.cs.rit.edu/~cs4/ISO˙DRAFT/
  - ✓ http://fr.wikibooks.org/wiki/Programmation˙C%2B%2B
  - ✓ http://www.research.att.com/~bs/C++.html
  - ✓ http://en.wikipedia.org/wiki/C%2B%2B
  - ✓ http://www.csci.csusb.edu/dick/c++std/cd2/gram.html
- ➲ Pour une description de la bibliothèque standard :
  - ✓ http://www.cplusplus.com/reference
  - ✓ http://parallel.vub.ac.be/documentation/programming/stl-reference.ps.gz
  - ✓ http://www.proba.jussieu.fr/~lemaire/docs/M1CPP/stl-ref-card.pdf

1.2 Infrastructure

Compilateurs Il existe de nombreux compilateurs pour C++, certains respectent la norme ISO (ou au moins s'y attachent), d'autres n'y accordent pas une importance primordiale. Voici les principaux compilateurs C++ :

- GNU C++, http://gcc.gnu.org/.
- ICC, http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/277618.htm
- Microsoft Visual C++, http://www.microsoft.com/express/vc/Default.aspx

Dans ce cours, nous utiliserons le compilateur libre développé par GNU. Le nom de l'exécutable est g++. Une version moderne de cet outil a une version majeure égale à 4 (g++ –version pour vérifier).

Makefile Vous trouverez un MAKEFILE générique sur la page web du cours. Il est conseillé de travailler dans un répertoire différent pour chacunes des questions de ce sujet.

Dans le répertoire de travail, on copie donc le MAKEFILE. On créé un répertoire pour la `question-1` contenant un nouveau MAKEFILE de la forme :

```
SOURCES=myfile.cc
TARGET=myexe
-include ../Makefile
```

On a alors accès aux commandes habituelles :

- `make` : construire l'exécutable.
- `make clean` : supprimer tous les fichiers générés.

On rappelle que sous EMACS, la commande `M-x compile` permet de lancer une compilation dans un *buffer*. En cas d'erreur de compilation, ce dernier est analysé et le raccourci "`C-x ‘`" déplace sur l'erreur suivante.

2 C++, un langage à objets, statiquement typé, avec classes

2.1 Programmation orientée objet

Dans cette partie, vous pouvez travailler dans un unique fichier `.cc` du nom de votre choix (en C++, contrairement à Java, il n'y a aucune contrainte sur les noms des fichiers sources). On ne s'occupe pas pour le moment de définir des fichiers d'en-tête.

2.1.1 Encapsulation, classes et instances

Le fragment de code suivant illustre la façon dont on définit une classe, ses constructeurs, ses attributs privés et publics, ainsi que ses méthodes.

```
class my_class {
  // Liste des membres (par bloc)
  // Membres publics :
  public:
    my_class () { ... } // Constructeur (par défaut).
    my_class (const my_class& c) { ... } // Constructeur (par copie).
    my_class (int i) :
      _i(i),
      _j(i + 1)
    { ... } // Constructeur (à partir d'un entier i, on initialise les attributs _i et _j).
    int j; // Attribut (modifiable).
  // Membres privés :
  private:
    int _k; // Attribut privé (par convention, commence par un '_')
};
```

Pour créer une instance de cette classe, nous pouvons allouer un objet sur le tas ou sur la pile (nous reviendrons sur ces deux notions dans la section 3.1 :

```
// Objets sur la pile.
my_class o1; // Le constructeur par défaut est utilisé.
my_class o2 (o1); // Ici, c'est le constructeur par copie.
my_class o3 (1); // Et cette fois-ci, le constructeur à partir d'un entier.
// Objets alloués dans le tas.
my_class* o4 = new my_class (42); // Cet objet pourrait vivre en dehors du scope courant...
delete o4; // ... mais il est désalloué immédiatement.
```

Exercice 1

❶ *Définissez une classe File possédant uniquement un attribut privé stockant un descripteur de fichier UNIX.*

❷ *Instanciez cette classe. Est-ce possible ? Comment doit-on interpréter une définition de classe ne contenant pas de constructeurs ?*

❸ *Nous voulons nous assurer que toute instance de la File est associée :*
– *à un nom de fichier et/ou un descripteur de fichier ouvert.*
– *à un indicateur signifiant l'état ouvert (en mode lecture seule) ou fermé du descripteur de fichier.*
Donnez la nouvelle définition de la classe File tenant compte de cet invariant. On rajoutera les attributs nécessaires ainsi qu'un nouveau constructeur. Un navigateur web ouvert sur la page `http://www.cplusplus.com/reference/` *pourra aussi être utile. Peut-on encore construire une instance de la classe sans spécifier d'arguments d'initialisation ?*

❹ *La syntaxe de la définition de méthodes suit la syntaxe de C :*

```
class my_class {
  virtual int my_method (int argc) { /* ... */ }
};
```

Ajoutez les méthodes suivantes à votre classe File :

- *close, qui ferme le fichier si il est ouvert.*
- *contents, qui renvoie le contenu du fichier sous la forme d'une chaîne de caractère. Cette méthode exige que le fichier soit ouvert.*

□

2.1.2 Héritage

La relation d'héritage entre deux classes s'expriment en C++de la façon suivante :

```
class my_class :public my_parent {
  my_class (int x) :my_parent (x) {} // Appel explicite du constructeur du parent.
};
```

Nous verrons, plus tard, qu'il existe plusieurs catégories d'héritage (en remplaçant le mot-clé public par un protected par exemple. Pour le moment, retenez que la syntaxe précédente exprime le mécanisme d'héritage que vous connaissez déjà en JAVA.

Exercice 2

❶ *Définissez une classe FileSystemNode, représentant un nœud dans un système de fichier. Un nœud sait faire tout ce que notre classe précédente sait faire, excepté la méthode contents. Utilisez une hiérarchie de classes pour factoriser le code de vos deux classes. Est-il toujours possible d'avoir des attributs privés ? Quelle solution proposez-vous ?*

❷ *Définissez une classe Directory héritant de FileSystemNode et dénotant un répertoire d'un système de fichier. Cette classe a une méthode members renvoyant la liste des fichiers stockés dans le répertoire. On utilisera le type des listes contenant des pointeurs sur des objets de type FileSystemNode, écrit std::list⟨FileSystemNode*⟩ (il s'agit d'une classe générique fournie par la bibliothèque standard).*

❸ *Pour tester votre classe Directory, affichez le contenu d'un répertoire dont le nom est pris en argument du programme. Voici comment itérer sur une liste :*

```
for (std::list⟨FileSystemNode*⟩::iterator i = mylist.begin (); i != mylist.end(); ++i) {
  ...
}
```

Nous expliquerons en détails cette construction dans la suite du cours. Pour le moment, retenons que le type std::list⟨FileSystemNode⟩::ite est le type de l'itérateur* [1]

❹ *Voici un programme :*

```
class A {
protected:int _x;
};
class B :public A {
public:int get_x () { return _x; }
};
int main (int argc, char** argv) {
  A a;
  int y = a._x; // Ne compile pas !
  B b;
  int z = b.get_x (); // Compile !
}
```

[1] Notion que vous avez du voir en JAVA et qui correspond à un objet servant à itérer sur les éléments de la liste.

Qu'en pensez-vous ?

□

2.1.3 Subsomption, classes abstraites et méthodes virtuelles pures

Exercice 3

❶ *En tant que nœud dans un arbre, une instance de la classe FileSystemNode est soit une feuille, soit un sous-arbre du système de fichier. Dans les classes FileSystemNode, File et Directory, rajoutez les méthodes :*

- *is_leaf () renvoyant un booléen valant true si le nœud est une feuille.*
- *children () renvoyant la liste des sous-arbres du nœud.*

Dans la classe FileSystemNode, cela n'a pas de sens de donner une implémentation de ces méthodes. On les définit donc comme des méthodes virtuelles pures :

```
class my_class {
  virtual void my_pure_meth (int x) = 0;
};
```

❷ *Peut-on encore instancier une valeur de type FileSystemNode ? Que vous répond le compilateur lorsque vous essayez de le faire ? Peut-on accepter un argument de type FileSystemNode ? Pouvez-vous expliquer cette restriction ?*

❸ *Écrivez une fonction affichant encore le contenu du répertoire dont le nom est pris en argument du programme, mais de façon récursive (en affichant aussi le contenu des sous-répertoires). On décalera les sous-répertoires de 2 espaces pour représenter la profondeur.*

□

2.2 Traits exotiques

2.2.1 Mot-clé protected et ses amis

Exercice 4

❶ *Pour des raisons d'encapsulation, on ne veut pas que le descripteur de fichier ne soit pas accessible à un client de notre hiérarchie de classes. Pour cela, C++propose un niveau de protection à mi-chemin entre private et public, noté protected. Il donne accès à un membre de la classe depuis les sous-classes exclusivement. En d'autres termes, un membre protected est privé pour les composants externes à la hiérarchie de classe mais public pour toute classe dérivée. Modifiez la classe FileSystemNode en conséquence.*

❷ *Certaines classes ou fonctions peuvent avoir un statut particulier en C++qui leur permet de passer outre l'encapsulation. On les appelle les **amis** de la classe. Déclarer une classe amie et une fonction amie se fait ainsi :*

```
class my_class {
  friend class my_friend_class;
  friend void foo ();
};
```

Définissez une fonction print_fd qui affiche le descripteur de fichier de n'importe quel nœud.

□

2.2.2 Mot-clé static

Exercice 5

❶ *En C++, il est possible de partager des méthodes et des attributs entre toutes les instances d'une certaine classe. Pour cela, on préfixe la déclaration du membre par le mot-clé static.*

Lorsqu'il s'agit d'un attribut, on fournit une valeur d'initialisation. Cette valeur n'est pas modifiable (mais elle peut pointer vers une valeur modifiable). L'initialisation peut se faire à l'aide d'une constante ou bien à l'aide d'un appel de fonction.

Lorsqu'il s'agit d'une fonction, elle n'a évidemment accès qu'aux attributs statiques (et non les attributs d'instance particulière).

Partagez un entier représentant le nombre de descripteurs de fichier ouverts entre toutes les instances de la classe FileSystemNode. Attention, débrouillez-vous pour que cet entier soit modifiable. Modifiez votre fonction d'affichage récursive pour qu'en début de chaque ligne apparaisse le nombre de descripteur de fichiers ouverts.

□

2.2.3 Liaison statique

En C++, on peut définir des méthodes à liaison statique. Lorsqu'on appelle une telle méthode, ce n'est pas le code de la méthode définie dans la classe définissant le type initial (le plus précis) de l'objet qui est utilisée mais celle du type statique connu pour l'objet au point de l'appel.

Exercice 6

❶ *Supprimez le mot-clé virtual dans l'ensemble de vos classes. Remplacez les définitions de méthodes virtuelles pures par une fonction contenant un corps quelconque. Que se passe-t-il ?*

❷ *À votre avis, à quoi servent les méthodes à liaison statique ? Faites un programme effectuant 10^{15} appels à une méthode virtuelle et un autre effectuant le même nombre d'appels à une méthode statique. Utilisez la fonction UNIX `time` pour comparer les performances des deux types de liaison. Qu'en concluez-vous ?* □

2.2.4 Surcharge, opérateur et argument par défaut

Le langage C++ autorise, dans une même classe, l'utilisation d'un même nom pour plusieurs méthodes si ces dernières ont des arguments formels de types di□érents. Ces noms peuvent correspondre à des opérateurs du langage (comme +,-,/,++,\ldots).

Exercice 7

❶ *Rajoutez deux méthodes name dans la classe FileSystemNode :*
- *la première n'attend pas d'argument et renvoie la valeur de l'attribut de nom ;*
- *la seconde attend un argument de type std::string et renvoie la valeur de l'attribut de nom préfixée par cette chaîne.*

❷ *On peut surcharger de nombreuses constructions syntaxiques comme, par exemple, l'opérateur correspondant aux crochets. Pour cela, on écrit :*

```
class my_class {
  unsigned int operator[](unsigned int j) {
    return (j / 2);
  }
};
my_class a;
a[16] == 8;
```

Surchargez l'opérateur des parenthèses dans la classe Directory pour que, si d est de type Directory, alors d("name") retourne le pointeur valant NULL si il n'existe pas de sous-répertoire ou de fichier nommé "name" dans le répertoire représenté par d et un pointeur de type FileSystemNode pointant vers ce nœud dans le cas contraire.*

❸ *On peut omettre un argument si une valeur par défaut a été spécifiée. Par exemple :*

```
class my_class {
  void my_meth (int x = 0) {}
};
```

Fusionnez les deux méthodes name à l'aide de ce mécanisme du langage.

□

2.2.5 *Downcast*

À toute instance de classe est associée une **information dynamique de type**. C'est cette information qui permet de déterminer quelle méthode appelée lors de la liaison dynamique. Elle permet aussi d'e□ectuer des coercions de type (*cast*) de manière sûres. Pour cela, l'opérateur dynamic_cast⟨Type*⟩(expression) fournit un équivalent de instanceof de JAVA. Cet opérateur renvoie une pointeur nul en cas d'échec.

Exercice 8

❶ *Modifiez votre fonction d'affichage récursive pour qu'elle affiche une étoile au début des lignes concernant les fichiers.* □

2.2.6 Mot-clé const

Le mot-clé const sert à indiquer qu'un objet n'est pas censé être modifié. Le système de type de C++ essaie de maintenir cette propriété. Comme nous l'avons vu en cours, dans un contexte où on peut partager des pointeurs, cette volonté reste un vœux pieux si on ne se tient pas à une discipline stricte de programmation.

La présence du mot-clé const implique l'existence de deux modes d'utilisation d'un objet. L'un lorsque le type de l'objet est préfixé par const et l'autre lorsqu'il ne l'est pas. Voici un exemple :

```
#include ⟨cstdio⟩
class A {
public:
  void foo () const { printf ("I am in mode:const\n") ; }
  void foo () { printf ("I am in mode:non-const\n") ; }
  void bar () const { printf ("a non-const can do what const can do.\n") ; }
  void baz () { printf ("but the converse is false !\n") ; }
} ;
int main (int argc, char **argv) {
  const A a_rd = A () ;
  A a ;
  a_rd.foo () ;
  a.foo () ;
  a_rd.bar () ;
  a.bar () ;
  a_rd.baz () ; // Ne compile pas !
  a.baz () ;
}
```

L'avant dernière ligne ne compile pas puisque la méthode baz suppose que l'objet est modifiable. Or, un objet de type const est censé ne pas l'être donc le compilateur rejette l'appel.

Exercice 9

❶ *Quelles méthodes peuvent passer en mode const dans vos classes ?*

❷ *Voici un nouveau programme :*

```
#include ⟨cstdio⟩

class A {
private:int* _i ;
public:
  A (int* j) :_i (j) {}
  void show () const { printf ("%d\n", *_i) ; }
} ;

int main (int argc, char **argv) {
  int k = 42 ;
  const A a = A (&k) ;
```

```
  a.show () ;
  k = 31 ;
  a.show () ;
}
```

❷ *Qu'en concluez-vous ?*

□

2.2.7 Espaces de nommage

Pour éviter de mettre toutes les déclarations dans un même espace de nom (ce qui force à terme à préfixer tous les identificateurs par un nom unique), C++ intègre une notion d'espace de nom (*namespace*). Ils sont introduits par le mot-clé namespace et aussi par les déclarations de classes.

Les espaces de nom forment une hiérarchie dont les nœuds les plus externes sont des déclarations de classes, unions, structures, constantes, fonctions, énumérations et types :

```
namespace my_namespace_1 {
   namespace my_subnamespace_1 {
      class A {
         class B {
            typedef int t ;
         } ;
         enum C { A, B } ;
      } ;
      namespace my_subsubnamespace_3 {
         ...
      }
   }
   namespace my_subnamespace_2 {
   }
   ...
}
```

On repère un nœud dans cette hiérarchie soit de façon relative, soit de façon absolue, de façon analogue au chemin d'accès dans un système de fichiers UNIX. Le séparateur de chemin est ici « :: ». Ainsi, my_namespace_1::my_subnamespace_2::A::B::t est un type valide.

On peut rendre implicite certains chemins. C'est ainsi que le chemin std est implicitement consideré quand un identificateur n'est pas trouvé dans l'espace de noms courant. Pour cela, on écrit :

```
using namespace std ;
```

Exercice 10

❶ *Englobez vos définitions dans un namespace nommé fs.* □

2.3 Application : retour sur le *design pattern* « Visiteur »

Pour permettre l'extension en fonctionnalité de notre hiérarchie, nous voulons intégrer le *design pattern* de visiteur.

Exercice 11

❶ *Rappelez le problème de l'extension en fonctionnalité.*

❷ *Écrivez une classe abstraite Visitor contenant une méthode concrète appliquant le visiteur sur un objet de type FileSystemNode et deux méthodes virtuelles correspondant à ce que doit faire le visiteur dans chacun des deux cas de notre hiérarchie. N'oubliez pas d'utiliser le mécanisme de surcharge de méthodes de C++ pour condenser l'écriture.*

❸ *Implémentez un visiteur servant à calculer la somme des tailles de tous les fichiers d'un répertoire et de ses sous-répertoires.*

□

2.4 Application : *design pattern* « Usine »

Le patron de conception d'usine sert à cacher à un client l'existence de sous-classes. Pour cela, on ne publie que l'interface de la classe abstraite et une classe, souvent nommée Factory, qui produit des instances en fonction de critêre d'entrée de forme arbitraire.

Exercice 12

❶ *À votre avis, quand est-il utile de cacher l'existence de sous-classe à un client ?*

❷ *Quels pourraient être les critêres d'entrée utiles pour un client lui permettant de construire des instances de la classe FileSystemNode ?*

❸ *À partir de l'explication informelle précédente, implémentez une classe Factory.* □

3 C++, héritier du C

Dans cette section, nous allons introduire quelques traits de C++, hérités directement du langage C et qui induisent des di□cultés très spécifiques à ce langage.

3.1 Allocation et désallocation explicite

3.1.1 Rappel sur la pile et le tas

Tout comme C, C++ fait une di□érence explicite pour le programmeur entre ce qui est alloué sur le tas et sur la pile. À l'entrée dans une fonction, les variables locales (aussi appelée variables lexicales, définies dans l'espace de nom de la fonction) sont stockées sur la pile. Elles sont désallouées au moment où on quitte le code de la fonction.

Ainsi, le programme C suivant est invalide :

```
int* ouch_my_foot () { int x; return &x; }
int main (int argc, char** argv) { int * oh_no = ouch_my_foot (); *oh_no = 42; }
```

Il en est de même pour le programme C++ suivant :

```
class A { public:int x; };
A* ouch_my_leg () { A x; return &x; }
int main (int argc, char** argv) { A *oh_no = ouch_my_leg (); oh_no→x = 42; }
```

Par contre, notons bien que le programme suivant est valide :

```
class A { public:int x; };
A ouch_my_leg () { A x; return x; }
int main (int argc, char** argv) { A oh_no = ouch_my_leg (); oh_no.x = 42; }
```

. . .car l'opérateur de construction par copie est appelé dans ce cas.

Si on veut qu'une donnée, allouée dynamiquement, persiste à travers les appels de fonctions, il faut l'allouer sur le tas (on utilise malloc en C habituellement dans ce but). Comme il n'y a pas de glaneur de cellules (*garbage collector*) en C et en C++, on doit désallouer les objets présents sur le tas explicitement dès qu'ils n'ont plus d'utilité (pour éviter les problèmes de **fuite de mémoire**). Par contre, si on se méprend sur l'utilité d'une donnée (on désalloue une partie de la mémoire encore utilisée par certains composants), alors on s'expose à un arrêt brutal du programme à la volonté du système d'exploitation. Une des di□cultés de cette gestion manuelle de la mémoire est donc de savoir exactement quand désallouer. Nous allons voir que C++ améliore sensiblement les outils pour répondre à cette question (encore une fois, si on se fixe une discipline de programmation stricte).

3.1.2 new/delete, le couple infernal

En C++, pour allouer un objet sur la pile, on utilise la syntaxe « Type ClassName (ConstructorArgument) ; ». Si on veut allouer un objet sur le tas, on utilise l'opérateur new qui nous fournit un pointeur sur l'objet alloué. On utilise ensuite l'opérateur delete sur ce pointeur pour désalloué l'objet.

Pourquoi ne pas utiliser malloc et free ? Encore une fois, il s'agit d'encapsuler les comportements des objets à leur création et à leur destruction à l'intérieur des classes. En e□et, en plus de faire un appel à malloc pour allouer l'espace nécessaire pour stocker les attributs de l'objet, new appelle le code du constructeur de la classe. De même, avant d'appeler free, la commande delete appelle un **destructeur** de la classe.

3.1.3 Destructeur

La syntaxe de définition des destructeurs est :

```
class my_class :my_parent {
  ~my_class () {
     // Le destructeur de la classe my_parent est appelé automatiquement
     // après l'éxecution de celui-ci.
  }
};
```

Exercice 13

❶ *Installez un destructeur dans la classe FileSystemNode qui ferme le descripteur de fichier et décrémente le nombre de fichier ouvert.*

❷ *Utilisez l'opérateur delete pour supprimer les fuites de mémoire.*

❸ *Installez un destructeur dans la classe Directory, qui se contente d'afficher un message lorsqu'il est appelé. Vérifiez qu'il est bien appelé lors de la destruction des répertoires. Que remarquez-vous ?*

□

3.1.4 Destructeur virtuel

Une erreur très souvent commise en C++est l'ommission du mot-clé virtual devant les destructeurs. La question précédente montre que si on l'oublie, c'est le destructeur du type connu de l'objet au point d'appel qui est appelé, ce qui peut être incorrect.

Exercice 14

❶ *Vérifiez que l'ajout du mot-clé virtual devant la déclaration du destructeur corrige le problème précédent.* □

3.1.5 Référence

Tout programmeur C le sait : les pointeurs peuvent jouer de mauvais tour (si ils sont nuls ou bien si ils sont partagés par exemple). Les **références** tentent de répondre à une partie de ce problème en évitant au programmeur de manipuler des pointeurs pouvant être nuls et en lui donnant la possibilité de contrôler le partage de pointeurs (dans une certaine mesure).

Le type des références sur une valeur de type T s'écrit T&. Une référence se comportent comme un pointeur : la valeur est une adresse et modifier la valeur pointée par cette adresse, la change pour toutes les occurrences de cette adresse dans le programme (notion de partage). Cependant, une référence « est censée ne pas être nulle ». Encore une fois, il s'agit d'une indication, d'une convention, car il est possible de construire des références nulles.

En particulier, le type référence est utilisé pour définir le constructeur par copie :

```
class my_class {
  // La référence sur l'objet other est en lecture seule.
  // Le constructeur par copie doit choisir si il veut partager le
  // les références que tient l'objet other ou recopier ses attributs
  // en profondeur.
  my_class (const my_class& other) { ... }
};
```

Exercice 15

❶ *Modifiez vos définitions de classes pour y insérer des références là où vous aviez des pointeurs. Intéressez-vous précisément à la méthode members de la classe Directory.* □

3.2 Schéma de compilation

3.2.1 Compilation d'un programme

Unités de compilation Comme en C, les unités de compilations sont, par convention, structurées autour d'un fichier d'en-tête, contenant les déclarations (spécifications) et un fichier d'implémentation contenant les définitions (implémentations).

Voici un exemple d'unité de compilation formée d'un fichier d'en-tête `my˙class.hh` :

```
#ifndef MY_CLASS_HH
# define MY_CLASS_HH
class my_class {
  public:
    my_class ();
    my_class (int z);
    void my_meth1 ();
    bool my_meth2 (int x);
};
#endif
```

...et voici son fichier d'implémentation associée :

```
#include "my_class.hh"
my_class::my_class () { ... }
my_class::my_class (int z) { ... }
void my_class::my_meth1 () { ... }
void my_class::my_meth2 (int x) { ... }
```

Makefile Il est essentiel d'indiquer les dépendances entre les di□érents fichiers de votre projet dans votre Makefile. Pour cela, vous pouvez les maintenir manuellement ou bien utiliser le programme `makedepend` sur l'ensemble de vos sources pour produire un fichier inclu dans votre MAKEFILE.

Exercice 16

❶ *Restructurez votre hiérarchie en affectant une classe par unité de compilation.*

❷ *Mettez à jour votre* MAKEFILE.

□

3.2.2 Compilation de bibliothèques

La compilation et la liaison de bibliothèques se déroulent exactement comme en C. On utilise les outils `ar` et `ranlib` (pour les bibliothèques statiques).

Exercice 17

❶ *Transformez votre projet en bibliothèque. Quel fichier d'en-tête voulez-vous publier ?* □

3.2.3 Conséquences du mot-clé inline

Dans de nombreux cas, les méthodes des objets contiennent très peu de code, on aimerait donc qu'au moment de la compilation, ce code soit mis en ligne (*inline*) dans le code client pour des raisons d'e□cacité.

Le mot-clé inline a été créée dans ce but. Cependant, en C++, on est alors alors forçé de plublier le code de la fonction dans le fichier d'en-tête ! (ce qui casse l'abstraction entre implémentation et spécification.) Voici l'illustration de ce phénomène :

```
/tmp/z % zed foo.hh
#ifndef FOO_HH
# define FOO_HH
class A {
 public:inline int foo ();
};
#endif
/tmp/z % zed foo.cc
#include "foo.hh"
inline int A::foo () { return 0; }
/tmp/z % g++ -c foo.cc
/tmp/z % zed bar.cc
#include "foo.hh"
int main () {
  A a;
  a.foo();
}
/tmp/z % g++ -c bar.cc
foo.hh4:warning:inline function 'int A::foo()' used but never defined
```

On doit donc transformer le fichier "foo.hh" en :

```
#ifndef FOO_HH
# define FOO_HH
class A {
 public:inline int foo () { return 0; }
};
#endif
```

4 C++, un langage de *méta-programmation* objet

Dans cette dernière section, nous allons introduire la **programmation générique**, c'est-à-dire une programmation de composants réutilisables par le biais d'une paramétrisation du code par des types (polymorphisme paramétrique). En C++, cette programmation s'appuie sur des mécanismes de **méta-programmation** fournis par langage (assez rudimentaire) de **patrons** (*templates*).

Les trois mécanismes généraux sont :

- la définition de patrons de composants logiciels dans lequel un (ou plusieurs) type(s) sont des inconnues (une sorte de composant à trous) ;
- l'instanciation de ces patrons pour une valeur donné des paramètres de type (une sorte de copier/coller du code à trous dans lequel on a remplacé les paramètres par leurs valeurs réelles).
- la spécialisation du patron, un cas particulier du patron pour un jeu particulier de valeur de paramètres pour lequel la définition change.

4.1 Patrons de classe

La syntaxe des classes templates suit le schéma suivant :

```
template ⟨class T, class U, ...⟩
class A {
  // On peut utiliser les types T, U, ...comme des types classiques.
};

A⟨int, std::string, ...⟩ a; // Le patron A est instancié pour T = int, U = std : :string, ...
```

Voici un exemple plus concret. Il s'agit d'un patron représentant la famille des classes représentant une paire de deux attributs.

```
template ⟨class T, class U⟩
class pair {
  private:
    T _first;
    U _second;

  public:
  pair (const T& first, const U& second):
    _first (first),
    _second (second)
  {}
  const T& first () const { return _first; }
  const U& second () const { return _second; }
};

pair⟨int, int⟩ my_pair;
```

Exercice 18

❶ *Que se passe-t-il lorsqu'on instancie le patron pair avec la classe suivante ? :*

```
class A {
private:
  A (const A&) {}
public:
  A () {}
};
```

Qu'en conclure sur les vérifications effectuées par C++ *sur vos déclarations ?*

□

4.2 Fonctions paramétrées par des types

Si les classes peuvent être paramétrées par des types, les fonctions aussi. Voici un patron de fonction e□ectuant la transposition des deux composantes d'une paire :

```
template ⟨class T⟩
pair⟨T, T⟩ transpose (const pair⟨T, T⟩& p) {
  return (pair (p.second (), p.first ()));
}
```

Exercice 19

❶ *Implémentez un patron de fonction paramétré par un type T, qui attend un pointeur sur T et qui suppose que la classe T fournit :*

- *une méthode next () renvoyant un pointeur sur T.*
- *une méthode at_end () qui vaut false alors la méthode next () va renvoyer un pointeur valide et qui vaut false si elle va renvoyer un pointeur nul.*

❷ *Comment comparez-vous les deux fonctions précédentes ? Peut-on toutes les deux les instancier sur des types quelconques ?*

□

4.3 Spécialisation

On peut donner une définition spécialisée d'un patron pour un jeu donné de types. Par exemple, on peut implémenter une version spécialisée du patron pair pour deux composantes de type char :

```
template ⟨⟩
class pair⟨char, char⟩ {
  private:int _data;
  public:
    pair (char first, char second):
      _data (((int)first) ≪ 8 + second)
    { ... }
    char first() const { return _data ≫ 8; }
    char second() const { return _data & 0x□; }
};
```

La spécialisation peut être partielle. Voici par exemple le patron spécialisé pour toutes les paires de valeurs dont la première composante est de type void :

```
template ⟨class T⟩
class pair⟨void, T⟩ {
  private:T _first;
  // ...
};
```

Exercice 20

❶ *Implémentez une version spécialisée du patron de classes pair pour deux booléens.*

❷ *Comme nous l'avons mentionné plus tôt, une classe se comporte comme un espace de nom (namespace) donc il est possible de définir en son sein des types, des énumérations, d'autres classes, . . .*

Cette propriété est encore vraie pour les classes paramétrées. Combinez avec la spécialisation, il devient possible de définir des fonctions qui « calculent des types » à partir des types, d'où le terme de « méta-programmation » utilisé dans le titre de cette section.

```
template ⟨class T⟩
class set {
// Dans le cas général, on implémente un ensemble de T par une liste.
private:std::list⟨T⟩& _elements;\
public:
// Pour énumérer les éléments de la liste, on utilise un itérateur.
// Un itérateur fournit les méthodes begin(), end(), * et ++.
  typedef typename std::list⟨T⟩::iterator iterator;
// ...
};

template ⟨⟩
class set⟨bool⟩ {
  // Un ensemble de booléens est l'un des quatres cas possibles.
  enum set_case { Empty, JustTrue, JustFalse, TrueAndFalse };
  private:set_case _elements;
  public:
  class iterator {
    private:
    enum iterator_case { OnFalse, OnTrue, AtEnd };
    set_case _which_case;
```

```
    iterator_case _iterator_state;
    public:iterator& operator++ () {
      // ...
    }
  }
};
```

Complétez cette spécialisation de classe.

□

4.4 La bibliothèque standard du C++

La bibliothèque standard du C++est composée d'une couche de bas-niveau héritée de C, d'un module de gestion des chaînes de caractères, et d'un ensemble de patron de classes fournissant les structures de données usuelles dans une implémentation générique (cette partie se nomme la *Standard Template Library* ou, STL, en plus court).

Exercice 21

❶ *Observez le module* `list` *de la STL. Vous devez avoir tous les éléments nécessaires à la compréhension de son interface.*

❷ *Lancez votre programme d'affichage de répertoire dans un répertoire contenant beaucoup de fichiers. Qu'observez-vous ? Proposez une solution pour résoudre ce problème technique.*

□