Formation C++ avancée
ou comment être les stars du C++

Raffi Enficiaud
INRIA

16-18 février 2009

INRIA - IMEDIA

Formation C++ avancée

Organisation générale

À qui s'adresse-t-elle ?

Public

À tout développeur C++ ayant idéalement quelques mois (années) d'expérience.

Prérequis

La personne doit savoir manipuler le C++, ou en tout cas le "C+-" (C++ façon C)

Thèmes de la formation

1. vous permettre d'être le plus autonome possible avec le C++
   - Mieux comprendre certaines "subtilités" du langage
   - Mieux comprendre ce que génère le compilateur à partir du code que vous écrivez
2. vous rendre plus productif avec les mécanismes du C++
   - Utiliser le compilateur pour faire des évolutions incrémentales
   - Travailler en équipe
   - Identifier plus précisément les sources potentielles de bug
   - Mettre en place des mécanismes de contrôle de fonctionnalité et d'évolution
   - S'intégrer au mieux à un "monde extérieur" à votre (partie de) programme

Déroulement

3 jours intenses de formation :)

Jour 1

- Café
- Café
- Rappel des bases du C++ (+ exercices)
- (Mini) étude de cas

Jour 2

- Les templates (+ exercices)
- La STL (+ exercices)
- La programmation générique (+ exercices)
- Les patrons de conception (+ exercices)

Jour 3

- Organisation de code dans les "grands" programmes
- Utilisation de Boost

Formation C++ avancée

J1 : C++ avancé

J1 : C++ avancé

Contenu

Quelques difficultés du langage

Subtilité ?

Exemples...

Syntaxe pas trop difficile
Fonctionnalités importantes
$\Rightarrow$ Langage difficile

Boucle for

```
1 int i;
2 for(int i = 0; i < 100; ++i) {
3 }
```

Réferences

Class A

```
1 class A {
2 std::string &s;
3 public:
4 A(std::string s_) : s(s_){}
5 const std::string& give_me_s() const {return
      s;}
6 };
```

Utilisation de A

```
1 A a("toto");
2 assert(a.give_me_s() == "toto"); //assert
      throws or bug
```

Rappels sur les instructions

- Types
  - struct
  - union
  - POD
  - typedef
  - extern
- Contrôle
- Pointeurs
- Référence
- Garantie de non modification : const
- Espace de nom
- Pointeurs sur fonction
- Références sur fonction
- volatile

Chaînes de caractère

Même là-dessus il faut revenir...

Plusieurs types de chaîne de caractère :
Déclaration :

1. Chaîne de type "classique" : char
2. Chaîne de type étendu "international" : wchar_t

Le type international est souvent utile pour ce qui concerne le système de fichier, les messages utilisateurs, etc...

```
1 char *s_constant = "blablabla";
2 wchar_t* s_wide = L"wide blablabla";
```

Préprocesseur :
Concatène automatiquement les chaînes de caractères qui se suivent :

```
1 char* toto = "One big string split " "among small strings"
2   " and among many lines"
3   "... ";
```

Mot clef "*enum*"

Definition ("enum")

Les enum définissent un ensemble (réduit) de valeurs.

Les enums sont des entiers (constant), mais les entiers ne sont pas des enums.

```
1  enum e_my_enum { // enum nommé e_my_enum
2    val1 ,  // par défaut = 0
3    val2 ,  // par défaut = 1
4    val4 = 4 // val4 forcé à 4
5  };
6
7  enum { // enum anonyme
8    u_1 ,
9    u_2 ,
10 } e1, e2;  // et dont e1 et e2 sont des instances
11
12 typedef /*enum*/ e_my_enum my_set_elements;
13
14 void func(enum e_my_enum e);
15 void func2(my_set_elements e);
```

Structures

Definition ("struct" (ou structure))

Une structure est une agrégation d'éléments

La structure est l'ancêtre de l'objet...

Structure nommée

```
1  struct A { // structure nommée A (facultatif)
2    int i;  // C
3    int j;
4  private:  // C++
5    int p;
6  };
7  A a;
8  a.i = 10; // ok
9  a.o = 20; // accès impossible
```

Structure anonyme

```
1  struct  { // structure anonyme
2    int i;
3    float f;
4  } b, c; // b et c en sont des instances
5  b.i = c.i = 10;
6  b.f = c.f = 20.f;
```

Il n'y a aucune différence fonctionnelle entre une structure et une classe.

Par compatibilité avec le C, l'accès par défaut est public (cf. partie sur les classes)

Union

Definition ("union")

Une union est une structure particulière, n'utilisant qu'un type à la fois.

Union

Déclaration

```
1  union u {
2    int a;
3    int j;
4
5  private:  // C++
6    int p;
7  };
```

Utilisation

```
1  struct A_union {
2    union unnamed_union {
3      std::string *s;
4      const char  *p;
5      double      d;
6      unnamed_union() : d(3.14) {}
7    } current_value;
8
9    enum enum_type {
10     e_string,
11     e_char,
12     e_float
13   } current_type;
14
15   A_union(enum_type e = e_float) : p(
         current_type), current_type(e),
         current_value() {}
16     ~A_union() {
17   }
18
19 };
```

Il n'y a aucune différence fonctionnelle entre une structure et une classe.

Types POD

Les types "natifs" ou agrégation de type natif

Definition (POD)

POD = Plain Old Data

Plus précisément, tout type ayant un équivalent direct avec le C, concernant l'initialisation, la taille, et l'adressage :

- toute valeur numérique (bool, int, double...)
- les enums et les unions
- les structures sans aucun constructeur, opérateur d'attribution, classe de base, méthode virtuelle, variable non-statique non publique, etc.

Mot clef "typedef"

Definition (typedef)

Crée un alias entre une désignation complète et un nom.

```
1 typedef struct s_my_struct toto;
2 typedef std::vector<int> int_vector;
```

Les alias ne prennent aucune "taille" au moment de l'exécution.

Généralement, les typedefs permettent d'avoir un code plus concis, et dans certains cas d'accélérer les temps de compilation (template).
Nous verrons quelques utilisations du mot clef "typedef" pendant la partie sur les templates.

Mot clef "extern"

Definition (extern)

Directive indiquant la définition d'un objet alloué "extérieurement" (ailleurs que la ligne indiquant sa définition).

- Les déclarations de fonction sont extern par défaut (par opposition à static)
- Il est possible d'avoir une déclaration extern suivie de la déclaration concrète

```
1 extern std::string s; // ambiguïté entre déclaration et stockage
2 std::string s;  // levée par le mot clef extern
3 void function1(); // déclaration par défaut extern
4 void function1(){} // déclaration du corps de la fonction
```

- extern désigne également externe au corps courant

```
1 void func(int i) {
2   extern A a1, a2, a3; // variables externes au corps de func
3   switch(i) {
4   case 1: a1++; break;
5   case 2: a2++; break;
6   default:a3++; break;
7 }}
8 A a1, a2, a3;
```

Contrôle

Boucles "for"

```
1 for(int i = 0; i < 100; i++)
```

La variable "i" est interne au bloc for. Les ";" définissent les champs : il est possible de ne pas les initialiser :

```
1 int i = 0;
2 for(; i < 100; i++)
```

Pointeurs

Définition

Definition (pointeur)

Un pointeur est une variable désignant un autre objet (l'objet pointé)

- Le type du pointeur contient le type de l'objet pointé
- Le fait d'accéder à l'objet pointé est le déférencement du pointeur.

La valeur "0" (zéro) est l'équivalent C++ de la macro C NULL. Elle permet de spécifier qu'un pointeur n'a pas été initialisé !

```
1 int *p; // pointeur pointant sur un entier (non initialisé, vaut n'importe quoi et donc éventuellement
      sur des données existantes)
2 int *u(0); // pareil, mais initialisé à 0 (ne pointe sur rien, même JAMAIS sur des données existantes)
3 int i = 10;
4 *p = 20;  // BUG HORRIBLE car ne se manifeste parfois pas
5 p = &i;   // p prend l'adresse de i
6 assert(*p == i); // déférencement de p
7 *p = *u;  // BUG PAS HORRIBLE car se manifeste TOUT LE TEMPS !
```

Référence

Définition

Definition (Référence)

Réfère un objet existant : alias sur l'objet référé (la référence est l'objet référé)

Attention !

- Une référence est une variable
- Mais la variable ne peut pas changer d'objet référé (contrairement à un pointeur).

Une variable référence doit toujours être initialisée lors de sa construction.

```
1 int a = 0, b = 1;
2 int &ref_a = a;
3 std::cout << "ref_a = " << ref_a << std::endl;
4
5 ref_a = b;
6 std::cout << "ref_a = " << ref_a << std::endl;
7 std::cout << "but the value of a is now = " << a << std::endl;
```

Mot clef "*const*"

Définition

Definition (const)

Mot clef indiquant que l'objet est constant (non mutable ie. non modifiable).

Un objet constant ne peut être pointé ou référé par une variable rompant la constance de l'objet pointé ou référé.

```
1  int const a = 0, b = 1;
2  //int &ref_a = a; // erreur, ref_a ne garantie pas que a est constant
3  int const &ref_a = a;
4  const int &ref_b = b;
5
6  //int *p_a = &a;  // erreur, p_a ne garantie pas que a est constant
7  int const *p_a = &a;
```

Nous verrons plus tard que, pour un objet, seules des méthodes assurant la constance de l'objet peuvent être appelées.

Mot clef "*const*"

Utilisation des pointeurs

- "const int" et "int const" sont sémantiquement équivalents.
- "const int *" et "int const *" sont donc deux pointeurs sur un entier constant (le const est avant l'*).

Ne pas confondre

"const int *", "int const *" et "int * const"

```
1  const int a;
2  int b, c;
3  const int *p_a = &a; // pointeur sur un entier constant
4  int* const p_b = &b; // pointeur *constant* sur un entier (la variable pointeur ne peut pas être changé)
5
6  p_a = &b; //ok
7  p_b = &c; // erreur, p_b (variable) est constant
```

Rappel : Une variable référence est toujours constante

int & const n'a pas tellement de sens (identique à int &).

Mot clef "*namespace*"

Définition & syntaxe

Definition ("namespace")

Permet de séparer un ensemble de définitions, types, fonctions... du "reste du monde" (inclusion de librairies/headers externes par exemple).

```
1  namespace ns1 {
2    int const a = 0, b = 1;
3    bool fonction1(int i, in j);
4  }
5
6  namespace ns2 {
7    int const a = 0, b = 1;
8    bool fonction1(int i, in j);
9    namespace ns22 {
10     void fonction1(float, float);
11   }
12 }
13 // Utilisation
14 ns2::fonction1(ns1::a, ns2::b);
```

Mot clef "*namespace*"

Usage

Alias de namespace

Il est possible d'avoir un alias sur un namespace

```
1 namespace NS = ns2::ns22;
2 NS::fonction1(0.32f, 0.31f);
```

Namespace implicite

Il est possible de dire au compilateur d'aller chercher dans un namespace de manière automatique :

```
1 using namespace ns2::ns22;
2 fonction1(0.32f, 0.31f);
```

Attention à cette dernière fonctionnalité : le côté implicite est souvent source d'erreurs.

Pointeurs sur fonction

Rappels

Definition (et usage)

Permet de stocker l'adresse d'une fonction dans une variable (l'attribution de cette variable pouvant être donné par une fonction).

La variable peut ensuite (bien-sûr) être utilisée pour appeler la fonction sur laquelle elle pointe.

```
1  bool func2(const double&, int, float&);
2  void func1(bool (*toto)(const double&, int, float&)) {
3    // Déclaration de la variable toto, comme étant un pointeur
4    // sur une fonction de type bool (const double&, int, float&)
5    float returned_f;
6
7    if(toto(0.39, 42, returned_f))
8      std::cout << "function returned true and the float is = to" << returned_f << std::endl;
9    else
10     std::cout << "function returned false" << std::endl;
11 }
12 func1(0); // erreur à l'exécution (lors de l'appel à toto): fonction nulle
13 func1(&func2); // attribution de toto à func2
```

Références sur fonction

Definition (et usage)

Permet de stocker la référence d'une fonction dans une variable (l'attribution de cette variable pouvant être donné par une fonction).

Contrairement au pointeur, la référence sur fonction est non-mutable (ré-assignation de la référence impossible).

```
1  bool func2(const double&, int, float&);
2  void func1(bool (&var)(const double&, int, float&)) {
3    // Déclaration de la variable toto, comme étant une référence
4    // sur une fonction de type bool (const double&, int, float&)
5    float returned_f;
6
7    if(var(0.39, 42, returned_f))
8      std::cout << "function returned true and the float is = to" << returned_f << std::endl;
9    else
10     std::cout << "function returned false" << std::endl;
11 }
12 func1(0); // erreur à la compilation: fonction nulle n'existe pas
13 func1(func2); // attribution de toto à func2
```

Mot clef "volatile"

Definition ("volatile")

" volatile " est un "modificateur"[a] indiquant au compilateur que la variable doit être lue en mémoire à chaque accès.

a. modifie le comportement par défaut d'un type

Ce modificateur a été initialement utilisé pour permettre des accès à des variables externes au processeur sur lequel le programme s'exécute (chip annexe par exemple). Il indique au compilateur, en outre, que la variable en question ne participe pas à des optimisations (inférences sur sa valeur).
Pour ceux/celles qui utilisent les threads, ce modificateur doit être utilisé pour les variables dont l'accès se fait via des threads concurrents.

Exercice

Fonction retournant un répertoire temporaire

On souhaite avoir une fonction, temporary_path, qui retourne une chaîne représentant le chemin complet d'un répertoire temporaire.
On souhaite en outre :

1. que le chemin temporaire soit lu à partir de la variable d'environnement "TEMP". Pour cela, on utilisera la fonction C/ANSI "std :: getenv" (dans "<cstdlib>") qui prend en paramètre le nom de la variable d'environnement et qui retourne "char *". Ce retour peut valoir "NULL" si la variable d'environnement n'existe pas.
2. on souhaite que cette variable ne soit lue qu'une seule fois, de manière à réduire l'accès "coûteux" à l'environnement.
3. si cette variable n'existe pas, on ne veut pas que notre programme plante. On initialisera la variable au répertoire contenant le fichier compilé (c'est super laid, mais c'est juste à titre d'exemple)[1]. Pour cela on se servira de la macro C/ANSI "__FILE__", de "std :: string" et de "std : string :: rfind" (on cherchera le '.' à partir de la fin).
4. on veut pouvoir modifier ce répertoire temporaire de la manière suivante :

```
1 | temporary_path() = "mon_nouveau_repertoire";
```

Indice : il s'agit d'une initialisation de type statique, à l'aide d'une fonction retournant la chaîne qu'on souhaite.

Classes

- Attributs membres
- Méthodes membres
- Attributs statiques
- Méthodes statiques
- Attributs mutables
- Champs membres
- this
- Pointeur sur membres
- Constructeurs de classe
- Destructeurs
- Surcharge d'opérateurs
- Opérateurs arithmétiques
- Opérateur d'attribution
- Opérateur de conversion
- Opérateur de fonction

Classe

Définition

Definition (Classe)

Une classe déclare des propriétés communes à un ensemble d'objets. Elle définie des attributs (variables membres) représentant l'état des objets, et des méthodes définissant leurs comportements.

Definition (Instance)

Une instance de classe est un objet suivant la définition de la classe, et dont les variables membres ont une valeur.

Classe

Définition (2)

La classe est l'élément central de la programmation orientée objet. L'objet est une instance de classe.

Rappel

Il n'y a pas de différence entre une classe et une structure.

```
1  // Déclaration d'une classe
2  class A {
3  public:
4    int i;
5    float j;
6
7    A() {}  // fabrique / constructeur (par défaut)
8
9    void evaluateMe() const; // méthode membre
10 };
11
12
13 A a;  // déclaration d'une instance de classe A
```

Classe

Déclaration & Syntaxe

- La *déclaration* est donné par la directive "class" ou "struct"
- Elle est suivie éventuellement du corps de la classe (accolades)
- Elle se termine par un ";"

Si le corps est défini, il est possible de déclarer des objets :

```
1 class A {} a, b, c;
2 struct B {/*...*/} d,e,f;
```

Si le corps n'est pas défini, la directive déclare une classe dont le corps est inconnu. Puisque le corps est inconnu, il est impossible de déclarer des objets de cette classe :

```
1 class A;
2 A a; // erreur: la taille de a est inconnue
3 A *a; // ok: la taille du pointeur et son type sont connus
```

Le corps de la classe définie ses membres. Une fois le corps terminé, il n'est pas possible d'ajouter des membres.

Classe

Attributs membres

Definition (attribut)

Les attributs définissent le contenu des classes.

Les attributs participent directement à la taille occupée en mémoire par la classe.

```
1 struct A {
2   bool value;
3   bool& getValue(); // déclaration de la méthode getValue
4   bool  getValue() const; // déclaration de la méthode getValue "const"
5 };
```

Classe

Méthodes

Definition (méthodes)

Les méthodes définissent le comportement des classes.

- Le corps de la classe doit contenir la déclaration ou la définition complète de la méthode membre.
- Les méthodes ne participent qu'exceptionnellement (cf. héritage) à la place occupée en mémoire par la classe
- Elle indiquent également si leur appel modifie la classe (ie. si la classe est "mutable" ou non).

```
1 struct A {
2   bool value;
3   bool& getValue(); // déclaration de la méthode getValue
4   bool  getValue() const; // déclaration de la méthode getValue "const"
5 };
```

Il est possible de définir les méthodes membres à l'intérieur du corps de la classe : il deviennent alors "inline" (déclarable dans un header par exemple).

Classe

Attributs static

Definition ("static")

Variable membre non liée à une instance particulière, et partagée par toutes les instances d'une classe.

- Il faut l'initialiser de la même manière qu'une variable statique normale, et en dehors du corps de la classe
- Sauf pour un entier constant, ou un enum, qui peut être initialisé à l'intérieur du corps de la classe
- La syntaxe impose la déclaration dans le corps de la classe, et l'attribution/initialisation à l'extérieur du corps de la classe. Attention aux accès concurrentiels.

Compteur d'instance

```
1 struct A {
2   static int nb_instance;
3   A () { nb_instance ++;}
4 };
5 int A::nb_instance = 0;
```

Classe

Méthodes statiques

Definition (Statique)

Méthode d'une classe non liée à une instance particulière, et partagée par toutes les instances.

1. Utilisable par toutes les instances d'une classe
   1. pas besoin d'une instance pour appeler la méthode
   2. impossible d'attribuer un modificateur "const" à la méthode
2. Accède à tous les autres membres statiques de la classe
3. Accède à tous les membres non statiques de le classe, si on lui fournit un pointeur sur l'instance courante (this)

"factory" (ie. fabrique) de classe

```
1 struct A {
2   static int nb_instance; // déclaration de l'attribut static
3   static int NbInstances() {return nb_instance;} // définition
4   static A create(); // déclaration seule
5 };
6 A A::create() {
7   nb_instance++; return A;
8 }
```

Classe

Attributs "mutable"

Definition ("mutable")

Variable membre non "static" et non "const", qui ne participe pas à la gestion du const de la classe.

La modification d'une variable mutable est autorisée à partir d'une méthode "const".

```
1 struct A {
2   mutable int value;
3   int  getValue() const { return value++; }
4 };
```

Les variables "mutable" permettent de cacher certains détails d'implémentation (ex. mécanisme de cache).

Classe

Contrôle d'accès

Dans le corps de la classe, on attribut à chaque membre une restriction sur l'accès, parmi :

- public : le membre est visible de "tout le monde"
- protégé (protected) : le membre n'est visible que d'une classe fille (cf. héritages)
- privé (private) : le membre n'est visible que d'une instance strictement du même type (qui peut être l'instance courante)

Par défaut, une "class" a une visibilité privée, alors qu'une "struct" a une visibilité publique (compatibilité avec le C).

```
1 class A {
2   int i_private; // privé
3   void method_private();
4 public:
5   int i_public;
6   void method();
7 } ;
```

Classe

Champs membres

Definition (champ)

La définition d'un type par l'utilisation de l'instruction "typedef".

```
1  struct A {
2    typedef A this_type;
3    typedef std::vector<int> storage_type;
4    typedef storage_type::value_type value_type;
5    /*...*/
6  };
```

Les champs de type sont valides à l'intérieur de l'espace de nom de la classe.
Rappel : les champs de type ne prennent aucune place en mémoire, et donc ne participent pas à la taille occupée en mémoire par la classe.

Classe

Mot clef "*this*"

Definition (this)

"this" pointe sur l'instance courante

```
1  struct A {
2    int i; float f;
3    void Addition(const A& r) {
4      this->i += r.i;
5      this->j += r.j;
6    }
7  } a1, a2;
```

Sauf...

Avant que la classe ne soit complètement construite : par exemple avant d'entrer dans le constructeur.

Nous verrons ce dernier point pendant les constructeurs.

Exercice

Décomposition en nombres premiers

Objectif

Avoir une classe, "decomposed_primes", décomposant un entier en puissances de nombres premiers.

1. un champ "return_type" précisera le type de stockage. Il sera du type "map"
2. la méthode "decompose" prendra un entier en entrée, et retournera "return_type"
3. une liste initiale de nombre premiers sera définie au lancement du programme (note : variable statique)
4. la classe aura une "mémoire" cachée, lui permettant d'ajouter des nombres premiers à sa liste initiale
5. pour plus de performances, la classe stockera la décomposition en nombre premiers des nombres inférieurs à 100.

On se servira du type "caché", "std :: map<int,int>" (tableau associatif), pour stocker une décomposition. On se servira du type "caché", "std :: set<int>" (ensemble ordonné), pour stocker les nombres premiers dans l'ordre croissant.
On s'aidera des fichiers "decomposed_primes.[h|cpp]".

Exercice

Décomposition en nombres premiers

Rappels sur map (liste associative)

```
1  #include <map>
2
3  // déclaration d'une liste dont les indices sont entiers,
4  // et dont les associations sont entières
5  std::map<int, int> map_numbers;
6
7  map_numbers.count(42); // retourne 1 si 42 est dans la liste des indices, 0 sinon
8  map_numbers[42] = 73; // met la valeur 73 à l'indice 42
9  int i = map_numbers[42]; // met la valeur de l'indice 42 dans i (seulement si 42 fait partie des indices
       de map_numbers, sinon erreur)
```

Rappels sur set (ensemble ordonné)

```
1  #include <set>
2
3  // déclaration d'un ensemble ordonné d'entiers
4  std::set<int> prime_numbers;
5
6  prime_numbers.count(42); // retourne 1 si 42 est dans la liste des indices, 0 sinon
7  prime_numbers.insert(42); // met la valeur 73 à l'indice 42
8  prime_numbers.insert(74);
9
10 // parcours de l'ensemble
11 for(std::set<int>::const_iterator it = prime_numbers.begin(), ite = prime_numbers.end();
12   it != ite;
13   ++it) {
14   std::cout << *it << " ";
15 }
```

Pointeurs sur membres

Syntaxe

Les pointeurs sur membre non static doivent indiquer qu'ils pointent à l'intérieur d'une classe. Il doivent donc spécifier le bon espace de nom.

```
1  struct A {
2    int i, j, k, l, m, n;
3    int methode(float);
4    int methode2(float);
5  };
6  int A::*; // pointeur sur un entier de A
7  int A::* p_i = &A::i; // pointeur "p_i" sur un entier dans A, initialisé sur l'adresse de i
8  int (A::*)(float); // pointeur sur une méthode de A (retournant int, et prenant un argument float)
9  int (A::*p_methode)(float) = &A::methode2; // pointeur p_methode initialisé sur A::methode2
```

1. S'ils pointent sur un membre static, ils sont absolus, n'ont pas besoin de spécifier l'espace de nom, et n'ont pas besoin d'une instance pour être déférencés.
2. S'ils pointent sur un membre non static, ils sont valident naturellement à travers une instance (nécessaire pour déférencer le pointeur).

Pointeurs sur membres

Syntaxe (2)

```
1  class A2 {
2  public:
3    static const int i = 0;
4    const int j = 1;
5    float myOperation1(const A2&) const;
6    float myOperation2(const A2&) const;
7    float myOperation3(const A2&) const;
8    static float myOperation4(const A2&);
9    static float myOperation5(const A2&);
10   static int i;
11   int j;
12 };
13 int A2::i = 0;
14
15 // ...
16
17 // definition du type "
18 // pointeur sur une fonction membre (-- cf retour (*)())
19 // dans le namespace A (cf. A::*)
20 // et laissant le namespace inchangé (const final)
21 // retournant float et prenant un référence const sur A
22 typedef float (A2::*operator_type)(const A2&) const;
23
24 // déclaration de variables
25 operator_type v1 = &A2::myOperation1, v2 = &A2::myOperation3;
26
27 // pointeur sur méthode static, identique à un pointeur sur fonction
28 float (*v4)(const A2&) = &A2::myOperation5;
```

Pointeurs sur membres

Syntaxe (3)

```
1  // appels
2  A2 a1;
3  A2 *a2 = &a1;
4  (a1.*v1)(a1);
5  a2->*v2(a1);  // méthode non static
6
7  v4(a1);      // méthode static
8
9  int A2::*  p_j = &A2::j;
10 int * p_i = &A2::i;
11 a1.*p_j = 10; // variable non static
12 *p_i = 20;    // variable static
```

Constructeurs

Définition

Definition (Constructeur)

Le constructeur initialise la classe, c'est-à-dire ses ressources et ses variables membres. Il est représenté par la méthode portant le nom de la classe. Il peut être surchargé.

Definition (liste d'initialisation)

Il s'agit d'une liste apparaissant juste après la parenthèse fermante du constructeur, et commençant par " :". Elle se termine avec l'accolade ouvrante du constructeur.

```
1 struct A {
2   int i,j,k
3   A() :   // début de la liste d'initialisation
4     i(0)
5     , j(1)
6     , k(2)
7   {     // fin de la liste d'initialisation
8   }
9 };
```

Constructeurs

Rappel

À ne pas confondre...

```
1 A a;       // déclaration d'un objet a de type A: le constructeur par défaut est utilisé
2 A a();       // déclaration d'une fonction a, sans paramètres et retournant un objet de type A
```

Constructeurs

Par défaut // par recopie

Definition (Constructeur par défaut)

Constructeur appelé sans arguments

Definition (Constructeur par recopie)

Constructeur avec comme argument une référence à une autre instance de la même classe

```
1  class A {
2    std::string *s;
3  public:
4    A() : s(0) {} // constructeur par défaut
5    A(const A& a_) : s(new std::string(*a_.s)) {} // constructeur par recopie
6  };
```

Attention ! !

Si non défini, le compilateur génère *automatiquement* des constructeurs par défaut (si possible) et/ou par recopie (si possible).

Constructeurs

Construction des variables membres

Les variables membres sont construites dans la liste d'initialisation.

```
1  struct A {
2    int i,j,k
3    A() :   // début de la liste d'initialisation
4      i(0)
5      , j(1)
6      , k(2)
7    {      // fin de la liste d'initialisation
8    }
9  };
```

Ordre de construction

L'ordre de construction des variables membres suit l'ordre de déclaration dans la classe.

Rappel : les méthodes ne sont pas des variables (et donc n'influencent pas la taille de la classe).

Constructeurs

Construction des variables membres (2)

Il existe bien des manières d'initialiser un objet...

Construction par défaut

Si le constructeur d'une variable membre n'est pas appelé explicitement dans la liste d'initialisation, et si la variable n'est pas de type POD, alors son constructeur par défaut est appelé.

```
1  class A {
2    std::string s;
3    int         i, j;
4  public:
5    A() : s(), i(0) {}
6    // j, non initialisé (vaut n'importe quoi) car int n'a pas de constructeur par défaut
7  };
```

Il est plus performant d'initialiser les variables membres directement dans la liste d'initialisation, plutôt que dans le corps du constructeur (bonne pratique : l'initialisation n'est faite qu'une fois).

Constructeurs

Contrôle d'accès

Les mêmes règles d'accès s'appliquent au constructeur :

Definition (constructeur public)

Constructeur appelable par "tout le monde"

Definition (constructeur privé)

Constructeur appelable uniquement par la classe elle-même

```
1  class A {
2    std::string *s;
3    A(const A&);     // interdit l'utilisation implicite du constructeur par recopie
4  public:
5    A() : s(0) {}      // défini et implémente le constructeur par défaut
6  };
```

Definition (constructeur protégé)

Constructeur appelable uniquement par une classe fille (cf. héritage)

Exercice

"Copy constructible"

Objectif

Faire en sorte que la classe suivante soit "copy constructible", c'est-à-dire qu'elle implémente le constructeur par recopie.

Classe

```
1  struct A_union {
2    union {
3      std::string *s;
4      const char  *p;
5      double      d;
6    } current_value;
7    enum {
8      e_string,
9      e_char,
10     e_float
11   } current_type;
12
13   const int some_int;
14
15   A_union() : some_int(0), current_type(
           e_float), current_value() {}
16   ~A_union() {if(current_type == e_string)
           delete current_value.s;}
17 };
```

Test

```
1  A_union a;
2  std::cout << a;
3
4  a.current_type = A_union::e_string;
5  a.current_value.s = new std::string("this is
       a string");
6  std::cout << a;
7
8  A_union b = a;
9  std::cout << b;
```

Exercice

"Copy constructible" (2)

Objectif

Faire en sorte que la classe suivante soit non "copy constructible".

Classe

```
1  struct A_union {
2    union {
3      std::string *s;
4      const char  *p;
5      double      d;
6    } current_value;
7    enum {
8      e_string,
9      e_char,
10     e_float
11   } current_type;
12
13   A_union() : some_int(0), current_type(
           e_float), current_value() {}
14 };
```

Test

```
1  A_union a;
2  std::cout << a;
3
4  a.current_type = A_union::e_string;
5  a.current_value.s = new std::string("this is
       a string");
6  std::cout << a;
7
8  A_union b = a;
9  std::cout << b;
```

Constructeurs

Variable membres *const*

Variables membres *const*

Ne peuvent être changée après instanciation de la classe

Donc...

Doivent être initialisée dans la liste du constructeur

```
1  class A {
2     const int  i, j;
3  public:
4    A() : i(0) {
5       j = 1; // erreur
6    }
7  };
```

Constructeurs

Variable membres *référence*

Les références doivent être initialisées dans la liste d'initialisation.

Attention

Les attributs références doivent référer soit :

- des objets dont la durée de vie dépasse celle de la classe !
- soit des objets temporaires "automatiques" (ce qui veut dire que la référence est "const")

Les "variables" références étant constantes, elles doivent être initialisées dans la liste constructeur également.

Trouver le bug

```
1  struct A {
2    std::string &s
3    A(std::string s_) : s(s_) {}
4  };
5
6  A a("value__");
7  if(a.s == "value__")
8    std::cout << "compilo buggé !" << std::endl;
```

Ce code fonctionne sous GCC 4.0.1 MacOSX et non sous Visual Studio 2008... mais ça reste un bug, même s'il fonctionne sur certaines architectures

Constructeurs

Restriction des conversions : mot clef "*explicit*"

Definition ("explicit")

Force l'appel du constructeur avec les types exacts passés en paramètres de celui-ci.

Aucun "trans-typage" (conversion implicite) n'est effectué pour les constructeurs explicites.

```
1 struct A {
2   explicit A(int, const float&);
3   explicit A(long, const double&);
4 };
```

Constructeurs

Mise en garde

Limite sur l'utilisation de *this*

La classe est totalement construite à la fin du bloc du constructeur (héritage de classe).

Donc ...

On ne peut pas utiliser *this* correctement

Mais ...

Les variables membres existent à l'entrée du bloc du constructeur

Destructeurs

Présentation

Objet

1. Détruit les objets qui appartiennent à la classe
2. Libère la mémoire allouée par la classe
3. Gère la durée de vie des pointeurs.

Il porte le même nom que la classe, et n'a jamais de paramètres en entrée. Il faut l'appeler avec l'opérateur "delete" et (pratiquement) jamais directement.

Syntaxe

```
1 struct A {
2   ~A() {}
3 };
```

Dans un programme stable, le destructeur est TOUT AUSSI important que le constructeur (ou n'importe quelle autre méthode).

Les objets sont détruits dans l'ordre inverse de leur déclaration lors de l'accolade fermante du destructeur.

Exercice

Classe de matrice

Enoncé

Ecrire une classe de matrice "matrice_d" de type double. Le constructeur de la classe aura pour paramètres deux entiers indiquant la taille (allocation dynamique).

Classes & surcharge d'opérateurs

Introduction

Definition (opérateurs)

Fonctions unaires ou binaires

Lorsqu'on défini l'opérateur d'une classe, on dit généralement qu'on surcharge cet opérateur. Ce sous-entend que si ces opérateurs ne sont pas définis, le compilateur essaie de les générer automatiquement.
Plusieurs opérateur d'intérêt, exemple :

```
1 operator= ; // attribution
2 operator== ; operator!= ; operator<; operator> ; ... // comparaison
3 operator++ ; operator-- ; // auto incrémentation et décrémentation (pré ou post)
4 operator+ ; operator+=; operator- ; operator-=; ... // arithmétique
5 operator* ; // déférencement ou multiplication
6 operator! ; // négation
```

Classes & surcharge d'opérateurs

Types d'opérateurs

Il existe deux types d'opérateurs :

1. les opérateurs unaires : ils n'impliquent qu'un seul objet, et donc ne prennent pas de paramètres

```
1 struct A {
2   A operator!();
3 };
```

2. les opérateurs binaires : ils impliquent deux objets : l'objet courant, implicite et à gauche de l'expression, et l'objet argument (explicite, en paramètre)

```
1 struct U {};
2 struct A {
3   A operator+(const U&);
4 };
5 A a, b;
6 U u;
7 b = a + u;  // a (de type A) est à gauche de u (de type U)
```

Si l'on souhaite définir un opérateur alors que l'opérande gauche est d'un autre type, il faut faire appel aux méthodes amies.

Dans une expression, le sens de l'évaluation dépend de la priorité de l'opérateur.

Exercice

Opérateurs arithmétiques

Enoncé

Pour une matrice :

- Ecrire les opérateurs d'addition, de multiplication avec un double
- Ecrire l'opérateur de multiplication avec une autre matrice

Enoncé

- Ecrire l'opérateur d'addition entre deux décompositions en nombre premier (classe "decomposed_primes").
- Ecrire l'opérateur d'addition entre une décomposition et un entier
- Ecrire l'opérateur de multiplication (deux "decomposed_primes" et un entier)
- Ecrire l'opérateur de division (deux "decomposed_primes" et un entier) : on se permet d'avoir des exposants négatifs

Opérateur d'attribution

Définition & syntaxe

Definition ("operator=")

Opérateur permettant de changer l'état de la classe grâce au signe =.

Syntaxe

```
1 struct A {
2   A& operator=(const A&);
3 };
```

Cet opérateur est utile lorsque la copie n'est pas triviale (ie. typiquement avec membres de type pointeurs et/ou références). Souvent, lorsque l'opérateur ne peut être généré automatiquement, le compilateur émet un avertissement (warning).

Opérateur d'attribution

Transitivité

Transitivité : $a = b = c = \ldots = z$

= est transitif

Il doit être possible d'écrire :

```
1 struct A {/*...*/} a1, a2, a3;
2 a1.init(42);  // initialisation
3 a2 = a3 = a1;   // transitivité
```

Donc

Le retour de l'opérateur d'attribution doit être une référence sur l'instance.
Exemple :

```
1 A& A::operator=(const A& r_) {
2   i = r_.i;
3   return *this;
4 }
```

Opérateur d'attribution

Remarque : auto-attribution (1)

Il faut gérer le cas $a = a$

C'est-à-dire ... il doit être possible d'écrire :

```
1 struct A {std::string *s; A& A::operator=(const A& r_);} a;
2 a = a;     // auto-attribution
```

sans danger pour la classe a ;

Exemple à ne pas faire...

```
1 A& A::operator=(const A& r_) {
2   delete s;
3   s = new std::string(r_.s);  // s n'existe plus si &r_ == this
4   return *this;
5 }
```

Opérateur d'attribution : remarques

Auto-attribution (2)

Donc...

1. gérer *explicitement* le cas $a = a$ par "this"

```
1  A& A::operator=(const A& r_)
2  {
3    if(this == &r_) return *this;
4    delete s;
5    s = new std::string(r_.s);
6    return *this;
7  }
```

2. changer l'ordre des opérations pour avoir une gestion *implicite*

```
1  A& A::operator=(const A& r_)
2  {
3    std::string s_tmp = new std::string(r_.s);
4    delete s;
5    s = s_tmp;
6    return *this;
7  }
```

Opérateur d'attribution

Code auto-généré

Le compilateur "sait" faire des choses :

Si les types contenus dans la structure/classe sont de type triviaux ou possèdent tous un constructeur par recopie (défini ou trivial), alors il n'est pas nécessaire de définir l'opérateur d'attribution (même remarque s'applique au constructeur par recopie).

Donc :

N'écrire l'opérateur d'attribution seulement quand vous le jugez nécessaire (membres pointeur/référence/const).

Opérateur d'attribution

Classe de matrice

Enoncé

- Ecrire l'opérateur d'attribution de la matrice avec une autre matrice : si les matrices ne sont pas de même taille, alors on fait la recopie.
- Ecrire l'opérateur d'attribution entre une matrice et une autre, entre une matrice et un entier.

Opérateur de conversion

Définition

Definition ("operator T")

Opérateur appelé lors d'un cast explicite vers un type particulier.

Cet opérateur peut être très pratique dans certains cas. Il ne définit pas de type de retour, puisque ce type est implicite par le cast.

Syntaxe

```
1 | struct A { operator int() { /* cast de A vers entier int */ } };
```

Exercice

Ecrire une classe qui affiche des informations sur son état lorsqu'on la transforme en chaîne de caractère. On utilisera par exemple la classe "decomposed_primes" et la classe "matrice_d" (exercices précédents - pour "decompose_primes" voir la fonction fournie dans les fichiers).

Classes : surcharge d'opérateur

Opérateur de fonction

Definition ("operator()")

Opérateur donnant à la classe l'interface d'une fonction.

Syntaxe

```
1  struct A {
2    bool operator()(double, float, int) const;
3    float operator()(int, float);
4  };
5
6  A a;
7  a(0.34, .32f, 42);
8  a(42, .32f);
```

Exercice

Ecrire l'opérateur de fonction de la matrice, afin d'accéder à ses éléments.
Question subsidiaire : pourquoi ne pas utiliser l'"operator []"?

Héritage

- Introduction
- Contrôle d'accès
- Visibilité & désambiguïsation
- Construction et destruction
- Conversions
- Héritage multiple
- Classes virtuelles
- Classes virtuelles pures
- Interfaces

Héritage

Définition

Que signifie héritage ?
Soit une classe A héritant d'une classe B :

1. A hérite partiellement du comportement de B
2. A "hérite" (sémantiquement parlant) de B par le biais des méthodes de B
3. A hérite également des attributs de B, s'ils sont marqués comme tel, c'est-à-dire si les attributs de B ont un accès public ou protégé.
4. A hérite des champs de type de B

Syntaxe :

```
1 class B {};
2 class A : [accès] B {}; // accès : public, protected ou private (ou rien)
```

Héritage

Contrôle d'accès

Le contrôle d'accès d'une classe mère s'applique aux classes filles. L'accès à la classe mère peut être davantage restreint.

```
1  class B {
2    std::string s; // attribut privé
3  protected:
4    int value; // attribut accéssible aux classes héritant de B
5  public:
6    int public_value;
7    void printMe() const;
8  };
9
10 class A : public B {
11   // A ne peut accéder à s mais peut accéder à value et public_value
12   // value est protected
13   // public_value est public
14 };
15
16 A a;
17 a.value = 10; // erreur, value est protected
18 a.printMe();  // a hérite de ce comportement, car l'héritage de B est public
```

Héritage

Contrôle d'accès (2)

Si l'accès n'est pas spécifié, par défaut il est :

- private pour les classes
- public pour les structures

Héritage de type

- public : visibilité inchangée (accès aux membres public et protected)
- private : la totalité des membres accessibles (ie. non privés) de la classe mère deviennent privés
- protected : public devient protected (private toujours inaccessible)

Héritage

Visibilité

Les attributs de la classe courante sont visibles prioritairement sur celles de la classe mère.

Les méthodes peuvent donc boucler sur elle-même !

Définitions

```
1  struct A {
2    typedef int value_type;
3    value_type doSthg() const;
4  };
5
6  struct B : public A {
7    typedef float value_type;
8    typedef A::value_type a_value_type;
9    value_type doSthg() const {
10     doSthg(); // appelle B::doSthg au lieu de
                 A::doSthg
11     // do some other things
12   }
13 };
```

Usage

```
1  B b;
2  B::value_type val = b.doSthg(); // val est
       float, boucle infiniment
```

Il est possible de se référer explicitement à une méthode ou un attribut de l'une des classes mères (si les contrôles d'accès le permettent), par la syntaxe suivante :

```
1  B::value_type doSthg() const {
2    A::doSthg(); // appelle A::doSthg au lieu de A::doSthg
3    // do some other things
4  }
```

Héritage

Constructeurs

Une classe fille contient la classe mère (c'en est une extension). Donc, si la classe fille est construite, il faut que la classe mère le soit également.

Constructeur : règle

- Le constructeur par défaut de la classe mère est appelé avant l'initialisation du premier (ie. de tous) membre de la classe fille.
- La classe fille peut appeler un constructeur particulier de la classe mère, dans sa liste d'initialisation

En fait, la classe mère peut être vue (en terme d'implémentation) comme une variable membre avant toutes les autres.

Les contrôles d'accès du constructeur de la classe mère est identique à n'importe quelle autre membre

Héritage

Constructeurs : exercice

Soit la classe mère suivante, ouvrant un fichier dans le répertoire courant :

```
1  class FileUtil {
2  public:
3    // initialise la classe avec le fichier non ouvert filename
4    FileUtil(const std::string &filename) {/*...*/}
5
6    /*...*/
7  };
```

Une fois le fichier ouvert, la classe propose des manipulations de ce fichier (or de propos ici).
Nous souhaitons une nouvelle classe héritant ces méthodes de manipulation, mais ouvrant les fichiers de manière différente.
Note : Nous considérerons que la classe mère stocke un *handle* de fichier dans la variable membre "FILE * m_file_handler". Elle offrirait à ses classes filles un constructeur particulier...

Héritage

Destructeurs

Destructeur : règle

Le destructeur de la classe mère est appelé après la destruction de tous les membre de la classe fille (lors de l'exécution de l'accolade fermante de la classe fille).

C'est à dire, dans l'ordre inverse de la création des éléments.

Héritage

Conversions

Il est possible de couper une classe, mais il n'est pas possible d'inventer des extensions de classe.

Une classe fille est une extension de sa classe mère. Il est donc possible de présenter la partie concernant sa classe mère.

```
1 // A fille de B
2 A a;
3 B *p_b = &a; // "du plus précis au plus général"
4 B& ref_b = a;
```

L'inverse n'est pas vrai, car il faudrait inventer des données manquantes :

```
1 // B fille de A
2 A a;
3 B *p_b = &a; // erreur de compilation: le compilateur ne sait pas
4 B& ref_b = a;// si ces instructions sont valides (a priori elles ne le sont pas).
```

Héritage

Conversions & dynamic_cast

Definition ("dynamic_cast")

Opérateur permettant de déterminer si une classe est transformable ("cast") en une autre en suivant le graphe de l'héritage de classe. Dans le cas de pointeurs en argument de l'opérateur, si le cast échoue, l'opérateur renvoie "0" (zéro).

```
1 struct A {};
2 struct B : public A {};
3 B b;
4 A *p_b = &b;
5 B *p_b2 = dynamic_cast<B*>(p_b);
```

Il est possible de tester à l'exécution(et parfois à la compilation), à partir d'un pointeur ou d'une référence sur une classe, si une classe dérive ou est une classe mère d'une autre.
Plus précisément, le compilateur sait si une classe est mère d'une autre à la compilation, mais l'inverse n'est pas vrai. Le test doit être fait en runtime.

Héritage multiple

Définition

Definition (Héritage multiple)

L'héritage multiple est lorsqu'une classe hérite de plusieurs autres classes.

```
1 struct A {};
2 struct B {};
3 struct C : public A, public B {}; // hérite de A et B
```

Les mêmes règles que l'héritage simple s'appliquent pour :

- Les contrôles d'accès
- La visibilité

Le graphe d'héritage étant plus complexe, le compilateur peut se plaindre parfois de ne pas pouvoir résoudre un appel (deux classes mères ayant une méthode de même nom/paramètres). Il faut alors faire la désambiguïsation de la même manière que pour l'héritage simple.

Héritage multiple

Constructeurs & destructeurs

De la même manière que pour l'héritage simple :

- Les classes mères sont toutes construites avant n'importe quel attribut membre de la classe.
- Si aucun constructeur de classe mère n'est appelé dans la liste d'initialisation, le constructeur par défaut est appelé.
- Les classes mères sont construites de gauche à droite dans la liste des classes de base (ex. précédent : A puis B).

et

- Les classes mères sont détruites après destruction du dernier membre de la classe fille
- Elles sont détruites dans l'ordre inverse de leur construction (de droite à gauche dans la liste des classes de base).

Classes virtuelles

Définition

Classe virtuelle - Définition

Une classe virtuelle est classe contenant au moins une méthode virtuelle

Méthode virtuelle - Définition

Méthode dont l'implémentation peut être (re)définie par une classe dérivée.

Le mot clef "virtual" indique si la méthode en question est virtuelle (ou pas).
Une méthode virtuelle participe à la taille occupée en mémoire par la classe.

Méthode virtuelle

La méthode appelée par défaut (ie. sans spécification d'espace de nom) est toujours la plus profonde dans la hiérarchie des classes.

Classe virtuelle

Exemple

Classes virtuelles

Classe A (mère)

```
1  class A {
2    std::string s;
3  public:
4    A(const std::string &s_ = "") : s(s_){}
5    ~A(){}
6    virtual void printMe() const {
7      std::cout << "I am A" << std::endl;
8    }
9  };
```

Classe implémentation A (fille) - invisible

```
1  // classe d'implémentation
2  class A_impl : public class A {
3    int *p;
4  public:
5    A_impl(const int size = 1000000000000) : A
         (), p(0) {
6      p = new int[size]; }
7    ~A_impl(){delete [] p;}
8
9    virtual void printMe() const {
10     std::cout << "I am the implementation of
           A" << std::endl;
11   }};
12 // factory
13 A* factoryA(){ return new A_impl();}
```

Appel

```
1  A* my_class = factoryA();
2  my_class->printMe();
```

Classes virtuelles

Utilisation

- Raffinement/spécialisation du comportement d'une classe
- Classe mère fournit des services minimaux
- Utilisation d'algorithmes existants avec de nouvelles implémentations

Classe implémentation A (fille)

```
1  class A_impl : public class A {
2    int *p;
3  public:
4    A_impl(const int size = 1000000000000) : A(), p(0) { p(new int[size]); }
5    ~A_impl(){delete [] p;}
6
7    virtual void printMe() const {
8      A::printMe();
9      std::cout << "I am the implementation of A" << std::endl;
10     std::cout << "I print some additional information" << std::endl;
11   }
12 };
```

Classes virtuelles pures

Définition

Definition (Classe virtuelle pure)

Une classe virtuelle pure est classe contenant au moins une méthode virtuelle pure.

Definition (Méthode virtuelle pure)

Méthode virtuelle sans implémentation (ie. $= 0$)

Syntaxe

```
1 struct A {
2   // Méthode virtuelle pure
3   virtual void printMe() const = 0;
4 };
```

Classes virtuelles pures

Propriétés

Une classe virtuelle pure ne peut être instanciée

```
1  struct A {
2    virtual void printMe() const = 0;
3  };
4  A* a = new A();          // erreur à la compilation : printMe non définit
```

Une classe mère peut donc "forcer" l'implémentation (dans les classes filles) des méthodes qu'elle déclare pures

Destructeurs

Héritage

Que se passe-t-il si delete est appelé à partir d'une la classe mère ?

Exemple

Classe mère : interface

```
1  struct A {
2    std::string s;
3    A(const std::string &s_) : s(s_){}
4    ~A(){}
5    virtual void doSomething() = 0;
6  };
```

Classe fille et "factory" : implémentation réelle

```
1  struct A_impl : public struct A {
2    A_impl(const int size = 1E5) : A(), p(0) {
           p(new int[size]); }
3    ~A_impl(){delete [] p;}
4    virtual void doSomething() {}
5
6  private: int *p;
7  };
8
9
10 A* factoryA(){
11   return new A_impl();
12 }
```

```
1  A* my_class = factoryA();
2  delete my_class;
```

Faire l'essai, et proposer une solution.

Interfaces

Définition

Definition (Interface)

Sert à définir un "couplage" entre un service (l'objet/composant) et ses clients.

Concept général indiquant les méthodes pour accéder à un objet (au moins en exploiter certaines fonctionnalités) : l'*interface* définie la signature pour l'utilisation de ces fonctionnalités. La particularité importante de l'interface est le découplage de l'accès d'un objet et de son implémentation. Cette particularité fait en sorte que :

1. Les clients ne connaissent pas les détails d'implémentation, et ne sont donc pas soumis aux variations de celle-ci (concept objet).
2. L'interface est supposée stable en termes de fonctionnalités et signatures, notamment par rapport à l'évolution d'un composant logiciel.
3. L'approche par interfaces est souvent moins performante, puisque les signatures ne sont pas optimales pour une implémentation particulière (et que l'accès aux variables membres se fait par le biais de méthodes d'accès).

En C++, une interface est une classe ne définissant aucune variable membre et dont toutes les méthodes sont publiques et virtuelles pures.

Interfaces

Exemple

Interface d'une objet permettant d'exploiter des vidéos

```
1  class IVideoService {
2  public:
3    virtual bool openVideo(const std::string& file_name) = 0;
4    virtual bool isOpen() const = 0;
5    virtual bool closeVideo()   = 0;
6
7    virtual bool seekFrame(long int frame_number) = 0;
8    virtual int  getFrame(unsigned char* buffer, const long int buffer_size, int &width, int &height) = 0;
9
10
11   // Destructeur virtuel
12   virtual ~IVideoService() {}
13 };
```

Rappel

Ne jamais oublier le destructeur virtuel

Même s'il s'agit d'un artefact du langage...

Exceptions structurées

- Définition & utilisation
- Détails
- Exceptions & constructeurs
- Exceptions & destructeurs
- Exceptions standards

Exceptions structurées

Définition

Definition (exception)

Une exception est un mécanisme spécial du C++, lancé par le mot clef "throw", et permettant d'arrêter l'exécution courante du programme jusqu'à une instruction de récupération d'exception (bloc "try" / "catch").

L'instruction "throw" peut être suivie de n'importe quelle classe, structure, entier.
L'instruction "catch" intercepte la même classe que celle du "throw".

Syntaxe

Déclarations

```
1 class Exception1 {};
2 class Exception2 : public Exception1 {};
3 struct Exception3 {
4   const char* const what;
5   Exception3(const char *w) : what(w){}
6 };
```

Fonctions levant des exceptions

```
1 // fonctions
2 void function1() {
3   std::cout << "function1 will throw soon" <<
            std::endl;
4   throw Exception1();
5 }
6 void function2() {
7   throw float(0.34f);
8 }
```

Exceptions structurées

Interceptions & relancement

Dans le bloc "catch", throw sans argument relance l'exception au bloc try/catch supérieur.

try/catch

Interceptions

```
1  try {
2    function1();
3  }
4  catch(Exception1 &e){
5    std::cout << "Caught Ex1" << std::endl;
6  }
7
8  try {
9    function2();
10 }
11 catch(float &e){
12   std::cout << "Caught float Ex " << e << std
         ::endl;
13 }
```

Relancement

```
1  void function3() {
2    try {
3      function1();
4    }
5    catch(Exception1& e) {
6      std::cout << "Caught in f3: rethrow" <<
           std::endl;
7      throw;
8    }
9  }
10
11 try {
12   function3();
13 }
14 catch(Exception1 &e){
15   std::cout << "Caught Ex1 " << std::endl;
16 }
```

Exceptions structurées

Interception générale

Dans le bloc "catch", avec pour argument l'ellipse (...) interceptes toute les exceptions.

Attention : puisque l'exception n'est pas nommée, il n'est pas possible d'avoir plus de précision sur l'exception.

```
1 try {
2   function3();
3 }
4 catch(...){
5   std::cout << "Caught some unknown exception " << std::endl;
6 }
```

Mais : Il est possible de relancer l'exception

Exceptions structurées

Déroulement des interceptions

Les intructions "catch" sont testées dans l'ordre.

```
1  try {
2    function3();
3  }
4  catch(Exception3 &e){
5    std::cout << "Caught Ex3" << std::endl;
6  }
7  catch(Exception1 &e){
8    std::cout << "Caught Ex1" << std::endl;
9  }
```

Attention :

1. à la gestion de l'héritage entre plusieurs exceptions
2. à l'ellipse, qui masque toute les autres interceptions

Exceptions structurées

Déroulement des interceptions : détails

Voici ce qu'il se passe lorsqu'une exception est levée :

1. On arrête l'exécution du programme au point du "throw"
2. Le programme teste s'il se trouve dans un bloc "try" (si ce n'est pas le cas, fin de programme)
3. On cherche un bloc "catch" compatible avec le type de l'exception levée (si aucun, fin de programme)
4. On crée une sorte de "tunnel/point de connexion" avec ce bloc "catch", et on se place dans un mode privilégié
5. Dans ce tunnel, on appelle les destructeurs de tous les objets crées sur la pile, dans l'ordre inverse de leur création
6. On retrouve l'exécution du programme au niveau du bloc "catch"

On appelle le processus de destruction des objets temporaires : le déroulement de pile (*stack unwinding*).

Exceptions

Utilisation

Deux philosophies

1. Retour de fonction (comme en Java)
2. Arrêt d'exécution sur faute "grave". Exemple : on appelle une fonction définie par certains critères sur les données qu'elle traite, et ces critères ne sont pas respectés.

Performances

Plus d'information sur la pile pour la gestion de l'exception, donc ralentissement des appels.

1. Possibilité de marquer le type d'exception possible levée dans une fonction
2. Rapprochement du catch de là où l'exception est potentiellement levée
3. Utilisation d'un mot clef pour indiquer qu'un appel ne lève pas d'exception ou ne lève une exception que d'un certain type (non supporté par tous les compilateurs).

```
1 void function1() throw();
2 void function2() throw(Exception3);
3 void function2() throw(...);
```

Exceptions

Cohabitation avec les constructeurs

Les exceptions cohabitent très bien avec les constructeurs !

En effet, les constructeurs n'ont pas de valeur de retour (résultat = objet construit). Lever une exception dans le constructeur en cas de problème permet :

1. Lorsqu'une exception est levée dans le constructeur, la classe n'est pas créée.
2. Les destructeurs des variables membres initialisés sont appelés
3. Le destructeur de la classe en cours de construction n'est pas appelé (la classe n'a pas été construite).
4. Les destructeurs des classes mères instanciées sont appelées (l'ordre d'instanciation joue un rôle important).

Exceptions

Cohabitation avec les destructeurs

Rappel : (quelques diapositives précédentes) le processus de *stack unwinding* est exécuté dans un mode particulier.
En fait, ce mode est particulièrement fragile, et ne supporte absolument pas un lancement d'exception.

Donc ...

Il NE FAUT PAS lever une exception dans le destructeur

Qu'avons-nous avec l'exemple ci-dessous ?

```
1  class A {
2    ~A() {
3      // exception levée dans le destructeur de A.
4      throw std::runtime_exception("some error");
5    }
6  };
7  try {
8    A a;
9    throw std::runtime_exception("juste pour embeter");
10 } catch(...) {
11   std::cout << "C'est bon, je recupere" << std::endl;
12 }
```

Exceptions

Standard "STL"

Le **standard** (fichier "<stdexcept>" de la STL) défini quelques exceptions "type". La classe de base est "std :: exception", qui implémente la méthode "virtual const char * std :: exception :: what() const", donnant des renseignements "utilisateurs" sur le message de l'exception.

```
1 std::runtime_error; // reporte les erreurs à l'exécution
2 std::logic_error; // reporte les erreurs de logique d'appel
3 std::invalid_argument; // argument invalide d'un appel
4 std::out_or_range;  // reporte les dépassements : utilisé dans certains appels STL (substring, vector
      ...)
5 std::bad_alloc;    // reporte les problèmes d'allocation mémoire (dans header <new>)
```

Il faut généralement se reporter à la documentation pour connaître les exceptions qu'on doit récupérer. Les exceptions font partie du comportement du service demandé : elles affectent le fonctionnement de VOTRE programme.

Exceptions

Exercice

But

Soit une classe "ContenuFichier" qui contient le contenu d'un fichier. À son instanciation, il ouvre le fichier, lit son contenu et le met dans un buffer, puis ferme le fichier.

- Il ouvre le fichier à l'aide de la fonction "fopen".
- Si le fichier n'existe pas, il lance une exception " file_not_found ". Cette exception est à définir. Elle doit contenir le nom du fichier recherché.
- Si le fichier existe, le constructeur alloue une zone mémoire de type "char $*$" d'une taille de 1024 octet (1Ko). Il lit ensuite le contenu du fichier à l'aide de la fonction C "fread" ("#include <cstdio>").
- tant qu'il n'a pas terminé la lecture du fichier, il place le bloc précédemment lu à la fin d'un tableau. Pour cela, on utilisera un vecteur STL "std :: vector" ("#include <vector>") et sa méthode "std :: vector :: push_back".
- à la fin de lecture, il doit allouer un gros bloc mémoire, dans lequel il va copier les blocs précédemment lus et placés en mémoire.

Exceptions

Exercice (suite)

Il doit gérer très proprement

- les problèmes de lecture
- les problèmes d'allocation mémoire ("std :: bad_alloc")
- la fermeture du fichier (toujours)
- la libération des ressources mémoires intermédiaires
- le lancement d'exceptions suffisamment informatives

Etude de cas

Etude de cas

Chaîne algorithmique

On veut un programme qui applique des algorithmes à la chaîne à une structure de donnée initiale. Chaque algorithme à une interface identique à tous les autres algorithmes. Il sera identifié dans la chaîne par son nom. Pour chaque chaîne algorithmique, chaque algorithme à un nom unique. Chaque algorithme "A" peut créer une nouvelle donnée, qui pourra être utilisée dans la chaîne par les algorithmes en aval de "A". La donnée sera par exemple identifiée par son type exact et le nom de l'algorithme qui l'a créé.
L'exercice est en deux phases :

1. Phase de réflexion/étude/design : à l'aide de mécanismes d'héritage et d'interfaces, définissez les intervenants informatiques. On fera un point sur le design.
2. Phase de développement : vous retroussez vos manches.

En 4 équipes.

1. Définition de l'interface d'un algorithme
2. Définition de la structure de donnée
3. Création de deux algorithmes

Formation C++ avancée

J2 : Programmation générique

J2 : Programmation générique

Contenu

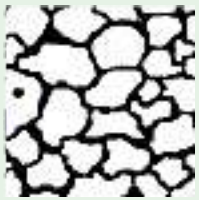
Les templates

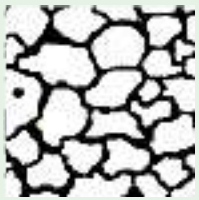
- Introduction
- Présentation
- Mécanismes
  - Déclaration & définition
  - Espace de nom
  - Inférence de type
  - Spécialisation
  - Programmation incrémentale
- Synthèse

Template

Introduction : problème de redondance algorithmique

Labélisation





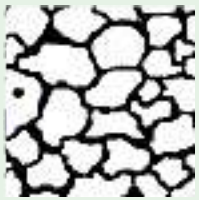
Mesures du nombre de composantes

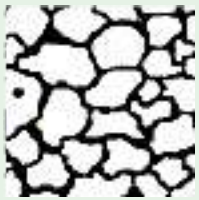
Développements trop spécifiques à une application !

Template

Introduction : problème de redondance algorithmique

Labélisation





Mesures du nombre de composantes

Développements trop spécifiques à une application !

Sans méta-programmation

Mesures du nombre de composantes

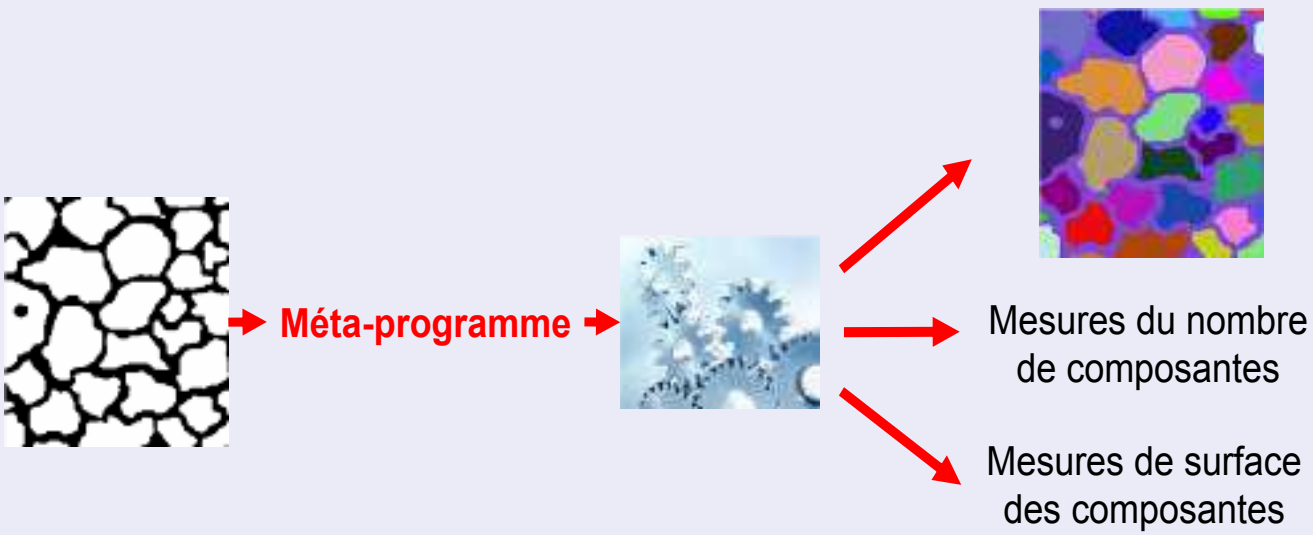
Mesures de surface des composantes

. . . . . .

Template

Introduction : problème de redondance algorithmique

Avec méta-programmation



Templates

Egalement appelés "patrons"

Definition (Template)

Les templates permettent de définir des familles de classes, structures ou fonctions.

- Mécanisme très puissant !
  1. permet de rapidement (à partir d'un code complètement typé) définir une famille de classe ou de fonction
  2. permet d'ajouter des mécanismes logiques dans le choix des structures/fonctions invoquées (méta-programmation)
  3. permet d'avoir un code facilement extensible à des nouveaux cas d'utilisation
- Difficile à prendre en main
- Certains compilateurs sont en retard dans le support des templates (il faut les bannir)

Templates

Présentation

Les familles en question sont paramétrées par des types :

- Ces types restent abstraits lors de l'écriture du template en question
- Il deviennent concrets lors de l'utilisation de ces templates (à la compilation)

Il est possible de définir des templates :

1. sur des structures ou des classes
2. sur des fonctions

```
1  // Structure template, avec 3 arguments:
2  // - deux classes I et U,
3  // - un booléen par défaut à true
4  template <typename I, class U, bool B = true>
5    struct s_structure {
6    /*...*/
7  };
8
9  // Fonction template, avec 3 arguments
10 // - un argument explicite booléen B
11 // - deux arguments implicites de type U et V (indéfinis)
12 template <bool B, class U, class V>
13   bool function_template(const U&, V) {
14   /*...*/
15 }
```

Templates

Exemple

Un exemple trivial de l'abstraction du type pour des fonctions...

Transformation

Algorithme typé

```
1 | int maximum(const int i1, const int i2) {
2 |   return i1 > i2 ? i1 : i2;
3 | }
```

Méta algorithme (int transformé en "T" et taggé comme argument template)

```
1 | template <class T>
2 |   T maximum(T i1, T i2)
3 | {
4 |   return i1 > i2 ? i1 : i2;
5 | }
```

Templates

Exemple & remarque

La fonction *maximum* sera alors appelable pour tout type "T", MAIS avec les conditions suivantes :

1. l'algorithme est valide si l'opérateur ">" existe pour le type "T"
2. la fonction prend des copies des objets et retourne une copie, ce qui n'est pas valide pour tout type "T" :
   1. l'opérateur par recopie est correct pour l'algorithme (il retient l'information permettant de faire la comparaison)
   2. l'opérateur par recopie est correct pour la partie appelante ("return" ne retourne pas une référence, mais une copie temporaire)

On commence à comprendre pourquoi c'est si séduisant, mais un peu délicat ?

Templates

Exemple (2)

Un exemple trivial de l'abstraction du type pour des structures...

Transformation

Structure typée

```
1  struct s_my_struct {
2    int value1[10];
3    typedef float return_type;
4
5    return_type operator()() {
6      int ret = 0;
7      for(int i = 0; i < 10; i ++) ret +=
               value1[i];
8      return ret / 10.f;
9    }
10 };
```

Méta structure

```
1  template <
2    class storage_type,
3    int size = 10,
4    class return_type_ = float>
5  struct s_my_struct_t {
6    storage_type value1[size];
7    typedef return_type_ return_type;
8
9    return_type operator()() {
10     storage_type ret = 0;
11     for(int i = 0; i < size; i ++)
12       ret += value1[i];
13     return ret / size;
14   }
15 };
```

Templates

Détails

Il existe plusieurs mécanismes autour de l'utilisation des templates :

1. la déclaration et la définition
2. la spécialisation totale ou partielle
3. la déduction (inférence) de type pour les fonctions templates

Templates

Déclaration & définition

La déclaration commence par le mot clef "template", suivi de :

1. "<" une liste de types ">"
2. "class" ou "struct" pour les classes et structures, ou un type de retour pour les fonctions templates
3. le nom de la fonction ou de la structure/classe
4. les arguments pour une fonction template (entre parenthèses)
5. optionnellement la définition du corps
6. et enfin ";" (point de terminaison)

Déclaration

```
1 template <class T> struct s_template_t; // structure template
2 template <class T> T func(int, T);      // fonction template
```

Les déclarations, classiquement, indiquent au compilateur que ces fonctions et ces structures templates existent.
Elles indiquent également le nombre d'arguments abstraits dans le template (utile pour la spécialisation ou pour les arguments templates).

Templates

Liste des types admissibles

Les types admissibles dans la liste des type sont :

- des types "abstraits" : "class" ou "typename" suivi de leur nom
- des valeurs constantes connues au moment de la compilation, et du type : entier, enum, pointeur, référence ou pointeur sur membre (le pointeur inclut son naturellement le type pointé)
- des classes templates

Il est possible d'avoir, pour les classes et structures templates, des arguments templates par défaut (et qui peuvent référer à des types précédents)

```
1  template <class T, class U = int, class V = T>
2    struct s_struct;
3
4  // même principe que pour la surcharge, V non optionnel ne peut suivre un argument optionnel
5  template <class T, class U = int, class V>
6    struct s_struct2;
7
8  template <class T, class U = int>
9    T func1(T, U);  // interdit ! pas de défaut sur les fonctions templates
```

Templates

Liste des types admissibles : valeurs

```
1  // rappel: "static const int" est définissable dans le corps de la classe
2  template <int I> struct s_template_t1 {
3    static const int i = I;
4    int j, k, l, m, o;
5  };
6
7  s_template_t1<20>   o1;      // ok
8
9  template <bool B> struct s_template_t2;
10 s_template_t2<true> o2;      // ok
11
12 bool b = false;
13 s_template_t2<b>  o3;      // erreur: b variable (inconnu au moment de la compilation)
14
15 const bool bc = false;
16 s_template_t2<bc>   o4;      // ok : bc constant
17
18 s_template_t1<bc>   o5;      // ok : bool "true" evalué à 1, false évalué à 0
19
20 enum e_val {val1, val2 = 2};
21 template <e_val E> struct s_template_t3;
22 s_template_t3<val2> o6;      // ok
23 s_template_t3<2>  o7;      // erreur: entier n'est pas enum !
```

Templates

Liste des types admissibles : valeurs (2)

Suite...

```
1
2  template <float F> struct s_template_t4; // erreur: float interdit
3
4  static char * const chaine = "blablabla";
5  template <chaine> struct s_template_t5;
6
7  // template avec argument un pointeur sur fonction (retournant
8  // void et sans paramètres)
9  template <void (*U)()> struct s_template_t6;
10
11 void func1();
12 s_template_t6<&func1> o8; // ok
13 s_template_t6<func1> o9; // erreur: il faudrait une référence
14
15 template <int I, int s_template_t1<I>::* J> struct s_template_t7 {
16   s_template_t7() : value_p(J) {}
17   int s_template_t1<I>::* value_p; // pointeur mutable
18   static const int s_template_t1<I>::* value_p_s;
19 };
20 template <int I, int s_template_t1<I>::* J>
21 const int s_template_t1<I>::* s_template_t7<I,J>::value_p_s = J;
22
23 s_template_t1<10> op;
24 op.i = op.j = 20;
25 op.k = op.l = 30;
26
27 typedef s_template_t7<10, &s_template_t1<10>::l> op2_t;
28 op2_t op2;
29 std::cout << op.*op2.value_p << std::endl;
30 std::cout << op.*op2.value_p_s << std::endl;
31 std::cout << op.*op2_t::value_p_s << std::endl;
```

Templates

Liste des types admissibles : template

Il est également possible de mettre parmi les arguments, un autre template. Il doit être déclaré précisément avec la même liste template que sa définition/déclaration.

```
1  // déclaration de s_template1, s_template2, s_template3
2  template <class A, int I>
3    struct s_template1 {/*...*/};
4
5  template <class A, int I>
6    struct s_template2;
7
8  template <class A, class B, class C>
9    struct s_template3 {/*...*/};
10
11 // déclaration de s_template4 acceptant un template en argument
12 // cet argument template (A) a deux arguments template : class et int
13 template <template <class A1, int I1> class A>
14   struct s_template4
15 {
16   A<int, 10> op_int;
17   A<float, 40> op_float;
18   //...
19 };
20
21 s_template4<s_template1> op1; // ok
22 s_template4<s_template2> op2; // non ok ! le corps de s_template2 n'est pas connu
23 s_template4<s_template3> op3; // erreur sur la mise en correspondance des arguments
```

Templates

Espace de noms & champs de type

Un peu le même principe que les namespace : utilisation de l'opérateur :: .

Definition ("types complets")

Dans un template, les types complets sont les types connus :

- soit parce que ce ne sont pas des types abstraits (constantes)
- soit parce qu'ils font partie de la liste d'arguments template [a]

a. les arguments templates sont également des types

Definition ("noms dépendants")

Un nom dépendant est un type qu'il n'est possible de connaître qu'une fois le type template complètement résolu.

Règle

Les noms dépendants doivent être accédés via l'opérateur "typename". Ce mot clef indique au compilateur que le nom dépendant est un type.
Il n'est valide qu'à l'intérieur des templates.

Templates

Espace de noms & champs de type (2)

Syntaxe

```
1  template <class T, class U>
2  struct A {
3    T t;
4    U u;
5    typedef typename T::return_type return_type; // champ membre de type T::return_type
6
7    A(const U&u_ = U()) : t(), u(u_) {}
8
9    return_type operator()() const {
10     typename U::const_iterator  // variables locales de type U::iterator_type
11       it = u.begin(),
12       ite = u.end();
13     return t(it, ite);       // T::operator()
14   }
15 };
```

Templates

Exercice 1

1. S'abstraire des opérations de recopies dans le template suivant :

```
1  template <class T>
2    T maximum(const T i1, const T i2)
3  {
4    return i1 > i2 ? i1 : i2;
5  }
```

2. En faire un functor, qui sache renseigner ses types d'entrée et de sortie.

Templates

Exercice 2

Ecrire une structure template *give_me_square*, présentant une interface de tableau indicé "operator []", et retournant le carré de l'opérateur de fonction template ("functor" - "operator()") suivant :

```
1  template <
2    class storage_type,
3    int size = 10,
4    class return_type_ = float>
5  struct s_my_struct_t {
6    storage_type value1[size];
7    typedef return_type_ return_type;
8
9    return_type operator()() {
10     storage_type ret = 0;
11     for(int i = 0; i < size; i ++)
12       ret += value1[i];
13     return ret / size;
14   }
15 };
```

Ce mini exercice doit montrer comment interfacer correctement deux templates.

Templates

Exercice 3

```
1  template <class T, class U>
2  struct A {
3    T t;
4    U u;
5    typedef typename T::return_type return_type; // champ membre de type T::return_type
6
7    A(const U&u_ = U()) : t(), u(u_) {}
8
9    return_type operator()() const {
10     typename U::const_iterator  // variables locales de type U::iterator_type
11       it = u.begin(),
12       ite = u.end();
13     return t(it, ite);       // T::operator()
14   }
15 };
```

Exercice

Ecrire des classes "T" et une classe "U" qui soient compatibles avec la structure template "A" ci-dessus. On pourra prendre par exemple pour "U" : une classe créée par vos propres soins (si vous avancez vite), ou "std :: vector" (facile) ou encore "std :: map" (un peu moins facile).

Exercice

Ecrire une fonction cliente de la structure "A" de test.

Templates

Intanciation/appel explicite

Definition (appel explicite)

Le fait de spécifier tous les types est nommé l'appel explicite

Exemple pour une classe

```
1 template <class T> struct A {/*...*/};
2 A<bool> op;
```

Exemple pour une fonction

```
1 template <class T>
2   bool function_t(const T& t) {/*...*/}
3
4 bool b = false;
5 b = function_t<bool>(b);
6
7 typedef std::vector<int> vect_int;
8 vect_int val;
9 b = function_t<vect_int>(val);
```

Templates

Inférence de types

Definition ("inférence de types")

La déduction/inférence des types est un mécanisme côté compilateur permettant de découvrir automatiquement les types (inférence) en fonction des arguments de l'appel d'une fonction template.

Exemple

```
1  template <class T, class U>
2  bool my_function(class T& t, class U& u) {
3
4    typename T::return_type return_value;     // variable de type "champ membre T::return_type"
5
6    typename U::const_iterator          // variables locales de type "U::iterator_type"
7        it = u.begin(),
8        ite = u.end();
9    for(; it != ite; ++it) {
10     return_value += t(*it);          // T::operator()
11   }
12
13   return return_value > 1000;          // exemple de transformation en booléen
14 }
```

Templates

Inférence de types (2)

Il est possible d'utiliser l'inférence de type uniquement sur les types passés en argument de l'appel. L'inférence échoue donc

- si certains types templates ne peuvent être déduits de l'appel
- si certains types templates concernent les types de retour des fonctions templates

Exemple 1 (qui ne fonctionne pas)

```
1 template <class T, class U>
2 bool my_function(const T& t, const T& u) {
3   // ...
4 }
```

Exemple 2 (qui ne fonctionne pas)

```
1 template <class T, class U>
2 U my_function(const T& t, const T& u) {
3   // ...
4 }
```

Templates

Inférence de types : exercice

Enoncé

Ecrire une fonction template qui écrit dans "std :: cout" le type passé en argument (la valeur ne nous intéresse pas ici). On se servira le mot clef C++ "typeid(T)" et la méthode "name" de l'objet retourné "typeid(T).name()".

Attention : "typeid"

La sortie de "typeid" est compilateur dépendant ! Il est utilisé pour tester l'égalité de types, mais non pour une sortie textuelle véritablement informative. "typeid" retourne un objet CONSTANT "type_info" (inclure le fichier #include <typeinfo>).

Templates

Spécialisation

Definition (Spécialisation)

La spécialisation permet d'indiquer que, pour un template donné, et pour une combinaison particulière de ses types, il existe une nouvelle version de ce template.

Syntaxe

```
1  template <class T, class U>
2  struct s_my_template        // déclaration de s_my_template
3  {
4    typedef T result_type;
5  };
6
7  template <>
8  struct s_my_template<int, int>    // spécialisation de s_my_template pour la combinaison int, int
9  {
10   typedef s_my_template<int, int> self_type; // le contenu de la spécialisation est complètement
         différent
11 };
12
13 s_my_template<float, bool> op;  // première version
14 s_my_template<int, int> op2;  // deuxième version
```

Templates

Spécialisation totale/partielle

Notion FONDAMENTALE !

La spécialisation est un mécanisme puissant qui relègue au compilateur les choix des structures lors de la découverte des paramètres templates.

Notion FONDAMENTALE ! (bis)

La spécialisation définit des NOUVEAUX TYPES. Une classe template et ses spécialisations n'ont RIEN DE COMMUN [a].

a. en termes de contenu

Cf. exemples précédents : les classes "s_my_template<U,V>" et " s_my_template<int, int>" sont des TYPES DIFFERENTS.

Definition (Spécialisation totale)

C'est un type de spécialisation où tous les types templates sont renseignés.

Cf. exemples précédents.

Templates

Spécialisation totale/partielle

Definition (Spécialisation partielle)

C'est un type de spécialisation où certains types restent à renseigner.

Syntaxe

```
1  template <class T, class U>
2  struct s_my_template          // déclaration de s_my_template
3  {
4    typedef T result_type;
5  };
6
7  template <typename U>
8  struct s_my_template<int, U>    // spécialisation de s_my_template pour la combinaison int, U
9  {
10   typedef s_my_template<int, U> self_type;
11 };
12
13 s_my_template<float, bool> op;  // première version
14 s_my_template<int, int> op2;  // deuxième version
```

Templates

Spécialisation totale/partielle : exercice

Enoncé

Ecrire une spécialisation de la structure de copie suivante, lorsqu'elle est appelée avec deux pointeurs constants sur entier (paramètre "T") et un pointeur sur double (paramètre "U").

Structure à spécialiser

```
1  template <class T, class U>
2  struct Copy {
3    void operator()(T it1, T it2, U it_out)
4    {
5      for(; it1 != it2; ++it1, ++it_out)
6      {
7        *it_out = *it1;
8      }
9    }
10 };
```

Enoncé

Ecrire une spécialisation de cette même structure, lorsqu'elle est appelée avec des pointeurs de même type (entier par exemple).

Templates

Spécialisation & programmation incrémentale

- Supposons que nous avons un functor "s_do_very_weird_things" qui travaille sur un ensemble de types "T_i"
- Supposons que maintenant, nous avons une version "améliorée" de ce functor, pour une combinaison C de ces types

Alors, il est possible d'ajouter cette version "améliorée" dans notre programme, sans "rupture de contrat" pour les appels existants. Le compilateur utilise alors la version améliorée ou générale automatiquement, selon la combinaison de type.

On [a] appelle ce mécanisme la *programmation incrémentale* : on commence par une version très générale, et ensuite on ajoute le particulier, sans retoucher le code client existant.

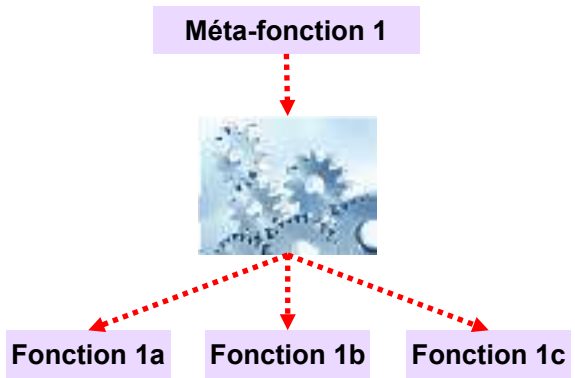
a. enfin je...

C'est donc le compilateur qui travaille plus que nous.

Méta-programmation ?

Résolution de nombreuses tâches par le compilateur

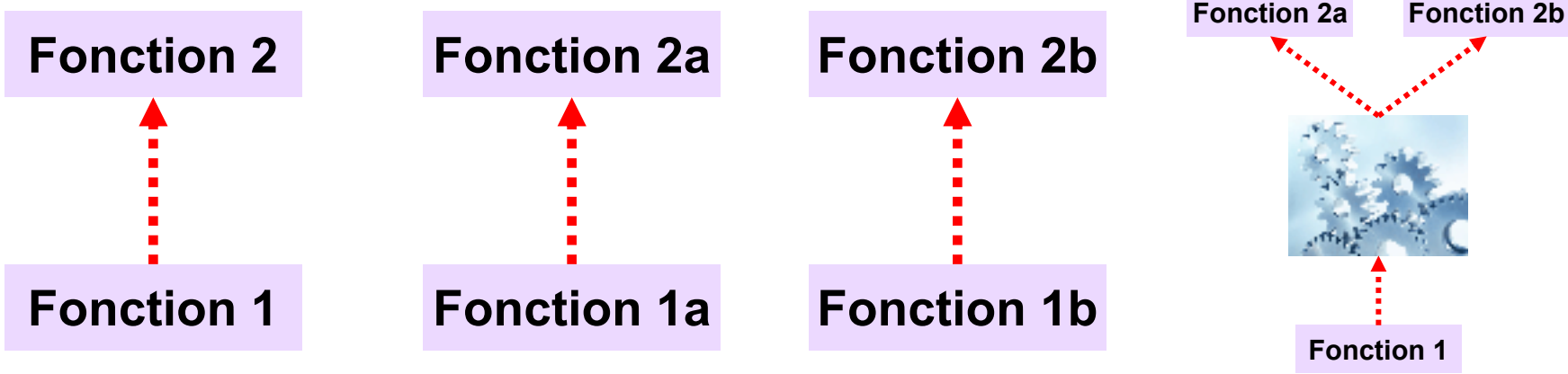
1. Résolution des types
2. Spécialisation



Méta-programmation ?

Résolution de nombreuses tâches par le compilateur

1. Résolution des types
2. Spécialisation



Méta-programmation

Programmation par templates : synthèse

Objectifs de la méta-programmation

1. Concentration des efforts sur le développement des algorithmes
2. Capitalisation
3. Réutilisation efficace de l'existant
4. Portage algorithmique

STL

- Functors - objets fonctionnels
  - <functional>
- STL & conteneurs
  - <vector>
  - <map>
- STL & algorithmes génériques
  - <limits>
  - <algorithm>

STL

Présentation

La STL (*Standard Template Library*) est une librairie **standard**[2], distribuée avec (pratiquement ?) tous les compilateurs C++.
Elle fournit des services minimaux, comme des structures template de "*container*" (éléments contenant), des algorithmes d'exploitation et des classes utilitaires de renseignement. Elle participe beaucoup à la création d'algorithmes templates.

Il faut très souvent faire appel à la STL
Utiliser la STL est bon pour votre productivité

2. ie. répondant à des spécifications fonctionnelles strictes

Algorithmes génériques

Functors & objets fonctionnels - rappel

Definition (functor)

Un *functor* est une structure proposant l'interface d'une fonction.

Il s'agit donc d'un objet, qui est par définition plus riche qu'une fonction, et qu'il est possible d'appeler comme une fonction.

```
1  struct s_functor_dumb_limited {
2    //! Surcharge de l'opérateur operator(), induisant le comportement désiré
3    bool operator()(const bool b) const throw() {return !b;}
4  };
5
6  s_functor_dumb_limited op;
7  assert(!op(true));
```

Plus généralement, un functor permet de faire des choses plus puissantes que des fonctions.

À ne pas confondre...

```
1  s_functor_dumb_limited op;    // déclaration d'un objet op de type s_functor_dumb_limited;
2  s_functor_dumb_limited op();  // déclaration d'une fonction op, sans paramètres et retournant un objet
       de type s_functor_dumb_limited
```

Algorithmes génériques

<functional>

Definition

"<functional>" Propose des functors utilitaires ainsi que des adapteurs pour créer "facilement" des functors.

Exemples :

```
1  // functor version de <, >, <=, >=
2  std::less, std::greater, std::less_equal, std::greater_equal;
3  std::plus, std::minus, std::multiplies, std::divides; // +, -, *, /
```

"std :: ptr_fun"

fonction template créant un functor à partir d'une fonction (binaire ou unaire)

```
1  // Déclaration d'une fonction binaire
2  void fonction_binaire(const int&, const float&);
3
4  //...
5  std::list<int> l(100);
6
7  // remplit la liste avec des nombres aléatoires
8  std::generate_n(l.begin(), 100, rand );
9
10 // trie la liste avec un prédicat : notre fonction
11 // transformée en functor
12 l.sort(std::ptr_fun(fonction_binaire));
```

Trouver les erreurs !

Algorithmes génériques

<functional> (2)

"std :: bind1st" / "std :: bind2nd" : fonctions mettant un des deux paramètres d'une fonction binaire à une constante
Le functor résultant devient unaire.

```
1  // Déclaration d'une fonction binaire
2  int fonction_binaire(const int&, const float&);
3
4  // generation de std::list<int> l
5
6  // trie la liste avec notre fonction
7  std::transform(
8    l.begin(), l.end(), l.begin(),
9    std::bind2nd(
10     std::ptr_fun(fonction_binaire),
11     0.34f)
12   );
```

<vector>

Tableau dynamique

Bon, tout le monde connaît ?

<map>

Tableau associatifs

Definition ("map")

Une "map" est un tableau associatif : à chaque clef, elle associe une valeur.

- Le type de la clef et celui de la valeur sont les deux paramètres templates **obligatoires**.
- Un troisième paramètre très utile donne la fonction d'ordre sur la clef. Il vaut par défaut le functor "std :: less" (inférant un ordre "<" sur l'espace des clefs).

L'ordre est donné par un functor, il est donc très facile d'en définir un sur une structure de clef complexe.

"map" garantie l'unicité des clefs, mais selon le modèle suivant :
Soit k_1 et k_2 deux clefs, P est le prédicat d'ordre.

$$\neg P(k_1, k_2) \wedge \neg P(k_2, k_1) \Rightarrow k_1 \text{équivalent à } k_2$$

Si la relation d'ordre n'est pas correctement construite, on peut avoir des mauvaises surprises.

<map>

Exercice

Enoncé

Soit la structure suivante :

```
1  struct s_my_key {
2    int coarse_order;
3    float fine_order;
4  };
```

1. La transformer pour qu'elle soit compatible avec "std :: map" et avec l'ordre par défaut. On souhaite associer "s_my_key" à des chaînes de caractères.
2. On souhaite avoir plusieurs ordres totaux. Proposez une approche avec des functors.

```
1  std::map<s_my_key, std::string> my_map;
2  s_my_key k[100];
3
4  for(int i = 0; i < 100; i++) {
5    k[i].coarse_order = 100 - i - 1;
6    k[i].fine_order = i / 10.f;
7    std::ostringstream o;
8    o << "my_string_" << i << "_" << k[i].coarse_order << "_" << k[i].fine_order;
9    my_map[k] = o.str();
10 }
11 assert(my_map.count() == 100);
```

<limits>

Renseignements numériques

<limits>

"<limits>" contient des informations relatives aux types numériques. Elle contient principalement une classe template "T", "<numeric_limits>" qui donne des informations sur le type numérique "T".

Soit un type "T", "std :: numeric_limit <T>" définie des méthodes statiques (accessible sans instance) permettant d'accéder à un ensemble de fonctions renseignant "T" :

```
1 std::numeric_limit<T>::max();      // fonction statique retournant le maximum possible pour le type T
2 std::numeric_limit<T>::min();      // le minimum possible pour le type T
3 std::numeric_limit<T>::is_integer(); // booléen indiquant que T est bien un type entier
4 std::numeric_limit<T>::epsilon();    // valeur indiquant la limite numérique d'indistinction entre deux
      T successifs
```

<limits>

Exemple d'utilisation

Initialisation correcte de l'algorithme en cas de liste vide :

Exemple sur l'initialisation

Déclaration

```
1  template <class T> typename T::value_type
       min_of_collection(T it, T ite) {
2    typedef typename T::value_type v_t;
3    v_t min_val = std::numeric_limits<v_t>::max
         ();
4    for(; it != ite; ++it)
5      if(min_val < *it)
6        min_val = *it;
7    return min_val;
8  }
```

Utilisation

```
1  std::vector<int> v1;
2  std::list<double> l1;
3  for(int i = 0; i < 100; i++) {
4    double val = rand();
5    v1.push_back(static_cast<int>(val * 1000 +
         0.5));
6    l1.push_back(val);
7  }
8  std::cout << min_of_collection(v1.begin(), v1
       .end()) << std::endl;
9  std::cout << min_of_collection(l1.begin(), l1
       .end()) << std::endl;
```

Algorithmes génériques

Utilisation de <algorithm>

La partie <algorithm> de la STL propose une suite d'algorithmes fréquemment utilisées : tri, partition, ordre lexicographique, etc.
Les algorithmes se déclinent généralement en plusieurs variantes qui se déclinent suivant les types en entrée.

Exemple

```
1  void do_some_stuff_on_list()
2  {
3    std::list<double> l(100);
4    std::generate_n(l.begin(), 100, rand); // "generate_n" : génération de conteneur par N appels à une
         fonction
5    std::list<double>::iterator it = std::find(l.begin(), l.end(), 20); // recherche
6    it = std::max_element(l.begin(), l.end()); // iterateur sur l'élément maximal de l'ensemble
7    std::stable_sort(l.begin(), l.end());     // ordre préservant les positions relatives
8
9    std::for_each(l.begin(), l.end(), std::bind1st(std::multiplies<double>(), 3));
10
11 }
```

Patrons de conception (Design patterns)

- Itérateurs
- Singletons

Patrons de conception

Définition

Definition (Patron de conception)

Les patrons de conception (en anglais *Design Patterns) sont des propositions de conception informatiques largement éprouvées dans la résolution de problèmes génie logiciel types.*

- De nombreux patrons sont définis (cf. Wikipedia), nous en verrons quelques uns
- La STL utilise abondamment la notion d'itérateur qui est un patron de conception
- Le C++ se prête bien à l'utilisation de patrons...

Améliore grandement le travail collaboratif (non intrusion de notions complémentaires, séparation fonctionnelle...)

Mais les patrons de conception restent des propositions (pouvant vous donner des idées) et suivre ces modèles n'est pas une fin en soi.

Patrons de conception

Itérateurs

Definition (Itérateur)

Un itérateur est un objet ayant une interface limitée, permettant de parcourir tout ou partie d'un objet contenant.

Avantage

Cet objet intermédiaire de parcours isole l'algorithme client de la structure interne du contenant : l'algorithme n'a pas besoin de connaître comment est constitué l'ensemble qu'il doit parcourir.

Algorithme de moyenne

Exemple d'utilisation... (trouver le bug)

```
1  template <class T> float make_my_mean(const T& container) {
2    float m = 0;
3    int i_nb = 0;
4    for(typename T::const_iterator it = container.begin(), ite = container.end();
5      it != ite;
6      ++it) {
7      m += *it;
8      i_nb++;
9    }
10   return m / i_nb;
11 }
```

Patrons de conception

Itérateurs

Itérateur : interface

1. test de fin d'itération : pour un itérateur en C++, un couple d'itérateur est nécessaire : le premier itère, le second marque la fin. Un test d'égalité permet alors de déterminer la fin de l'itération
2. incrémentation ou décrémentation : avance ou recule l'itérateur d'un (ou plusieurs) élément(s)

- le test de fin est implémenté en surchargeant l'"operator!=" de la classe de l'itérateur
- l'incrémentation est implémentée en surchargeant l'"operator++" de la classe de l'itérateur

Il existe deux versions de l'"operator++" : le préfixé ("++it") et le post-fixé ("it ++"). Le premier ne prend pas d'argument et retourne une référence sur lui-même ("return $*$this"). Le second prend un argument entier (non utilisé) et retourne une copie de lui-même avant incrémentation.

Patrons

Singletons

Definition (singleton)

Assure qu'une classe n'est instanciée qu'une seule et unique fois.

Utile dans certains contextes : par exemple une classe de synchronisation de ressources, une classe de mappage d'évènements, etc.

Exemple d'implémentation

Template de singleton

```
1  template <class T> class singleton {
2    static T singleton_object;
3  public:
4    static T& GetInstance() {
5      return singleton_object;
6    }
7  };
8
9  template <class T>
10   T singleton<T>::singleton_object = T();
```

Classe singleton

```
1  // Ecrire la classe ExempleSingleton
       utilisant le template précédent
2  class ExempleSingleton;
```

Ecrire la classe "ExempleSingleton" (utilisation minimale du singleton).

Formation C++ avancée

J3 : Librairies externes et organisation de code

J3 : Librairies externes et organisation de code

Contenu

Librairies externes

- Compilation et édition de liens
- Visibilité du code & interfaces

Librairies externes

Interfaçage avec le "C"

Le C++ permet l'interfaçage "natif" avec le C

Il faut pour cela que les types C soit correctement renseignés (au compilateur C++) comme étant C.

Lorsque l'on compile une unité de code en C++ :

1. la macro __cplusplus est toujours définie (tout compilateur confondu)
2. la directive extern "C" permet alors de renseigner le type (structure, fonction)

Librairies externes

Interfaçage avec le C : exemple

Exemple de déclaration

Déclaration d'un header compatible avec le C et le C++

```
1  // fichier toto.h : header C/C++
2  #ifdef __cplusplus
3  extern "C" {
4  #endif
5
6  #include "header_c.h" // inclusion de
       déclaration C
7
8  struct s_toujours_c {
9    int i;
10   int *tab;
11   /*...*/
12 };
13
14 void do_process(struct s_toujours_c *);
15
16 #ifdef __cplusplus
17 } // accolade fermate extern "C"
18 #endif
```

Utilisation

```
1  // fichier toto.cpp
2  #include "toto.h"
3
4  class toto : public s_toujours_c {
5    toto() : i(100) {}
6    void init() {
7      do_process(static_cast<s_toujours_c*>(
           this));
8    }
9  };
10
11 // fichier toto.c
12 #include "toto.h"
13
14 void do_process(struct s_toujours_c *s) {
15   s->tab = (int*)malloc(s.i * sizeof(int));
16   /*...*/
17 }
```

Visibilité du code & interface

Remarques sur les inclusions

Remarque 1

Il n'existe pas de mécanisme trivial pour cacher des détails d'implémentation.

"private/protected/public" ne sont que des directives destinées au compilateur, et indiquant approximativement l'usage des membres d'une classe.

Remarque 2

Le nombre de types utilisés pour les implémentations croît rapidement.

Pour qu'un utilisateur externe puisse utiliser une classe, il faut qu'il manipule tous les types des méthodes que cette classe appelle, ainsi que tous les types utilisés pour l'implémentation.

Ces deux points sont gênants, à la fois

- pour le fournisseur (divulgation)
- pour l'utilisateur (pollution)

Visibilité du code & interface

Rappel

Interfaces

Les interfaces permettent d'arriver à un résultat correct pour les deux parties, pour un surcoût négligeable.

- Elles laissent le soin au fournisseur de service de modifier l'implémentation, tant que l'interface du service reste la même
- Elles empêchent la divulgation de détails d'implémentation
- Elles évitent au client le fardeau de la gestion de type liés à l'implémentation du fournisseur
- Elles sont compatibles avec les templates et potentiellement la présence de beaucoup de types
- Elles restent souples à manipuler pour le client

Extension C++ avec Boost

- Présentation
- Modes d'utilisation & compilation
- filesystem
- Tests unitaires
- Thread

Boost

Présentation

Boost (http ://www.boost.org) est un Consortium de développeur C++, travaillant sur des extensions standards du C++.

Licence

La licence d'utilisation est d'exploitation de Boost est permissive et non intrusive (redistribution sans restriction dans des programmes à but commercial ou non, sans obligation de mention ni redistribution des sources).

Il faut utiliser Boost.
Utiliser Boost est une bonne chose, car Boost vous veut du bien.
Il faut convaincre tout le monde d'utiliser Boost.
Boost est le prochain standard.

Boost

Modes d'utilisation

Deux manière d'utiliser Boost :

1. version "headers" : à inclure dans l'unité de compilation concernée, et c'est tout (ex. : graph, mpl, proto, fusion, asio...)
2. version compilée : certaines (peu) parties de Boost nécessitent une compilation, et (donc) une liaison avec le programme client (ex. : filesystem, regex, mpi,...)

Il faut utiliser Boost, c'est simple à installer.

Boost

Compilation et installation

- Unises (Unix, Linux, Mac, Cygwin, ...)
  1. `$ cd path/to/boost_1_37_0`
  2. `$ ./configure --prefix=path/to/installation/prefix`
  3. `$ make install`
- Windows avec MS Visual Studio
  1. Démarrer la ligne de commande Visual (normalement dans le menu démarrer et dans le répertoire de Visual)
  2. `$ cd path/to/boost_1_37_0/tools/jam`
  3. `$ build.bat`
  4. `$ cd path/to/boost_1_37_0`
  5. `$ copy path/to/-boost_1_37_0/tools/jam/src/bin.Quelquechose/bjam .`
  6. `$ bjam --build-dir=D:\temporary_dir --toolset=msvc --build-type=complete stage`

Boost

<boost/filesystem>

Objectif

Propose une abstraction plate-forme pour la manipulation du système de fichiers.

Ex. : nom des fichiers, gestion de l'arborescence, gestion des droits d'accès, existence, etc.

Il faut préalablement compiler la partie correspondante de Boost ("with-filesystem" dans bjam), et lier (linker) avec "boost_filesystem" lors de la création du programme.

Concaténation de répertoires

```
1  #include <boost/filesystem.hpp>
2  #include <boost/filesystem/fstream.hpp>
3  namespace fs = boost::filesystem;
4
5  fs::path dir_(".");   // objet de type "path"
6
7  fs::path new_p = dir_ / "new_p" / "file_set" / "my_new_file"; // "operator/" overload
8  new_p.create_directories(new_p.parent_path());
9
10 std::ofstream my_file(new_p);
11 my_file << "toto" << std::endl;
12 my_file.close();
```

Boost

<boost/filesystem> : exemple

Détermination du type d'un chemin

```
1  #include <boost/filesystem.hpp>
2  namespace fs = boost::filesystem;
3  //...
4  std::string file_name = "my_file_or_file";
5  fs::path file_(file_name);  // objet de type "path"
6  if(!fs::exists(file_))
7    std::cout << "file \"" << file_name << "\" does not exists !";
8
9  fs::file_status stat_ = fs::status(file_);
10 std::cout << "This is a ";
11 switch(stat_.type()) {
12 case fs::regular_file: // affichage de la taille pour les fichiers
13   std::cout << "regular file of size " << fs::file_size(file_) << std::endl;
14   break;
15
16 case fs::directory_file: // liste des fichier et nombre de fichier pour un répertoire
17   {
18     std::cout << "directory containing " << std::endl;
19     int i = 0;
20     for(fs::directory_iterator itr(file_); itr != fs::directory_iterator(); ++itr)
21     {
22       i++;
23       std::cout << "\t" << itr->path().filename() << std::endl;
24     }
25     std::cout << std::endl << i << " files" << std::endl;
26   }
27   break;
28 default:
29   std::cout << "Unknown type";
30   break;
31 }
32 //...
```

Boost

<boost/test>

But

Proposer un ensemble d'outils pour écrire des tests unitaires et de régression robustes.

La librairie attend que nous définissions une fonction

```
1   boost::unit_test::test_suite* init_unit_test_suite(int argc, char* argv[]); // avec arguments ligne de
        commande
2   bool init_unit_test_suite(); // plus simple
```

Le corps de cette fonction va déclarer nos suites de tests, de la manière suivante :

```
1  void test_case1();
2  test_suite* init_unit_test_suite(){
3    test_suite* ts1 = BOOST_TEST_SUITE("test suite 1");
4    ts1->add(BOOST_TEST_CASE(&test_case1));
5  }
```

Les tests ne démarreront qu'au retour de cette fonction de de déclaration.

Boost

<boost/test> : exemple

Exemple de test

```
1  void test_case1() {
2    BOOST_CHECK(1 == 1);     // test 1
3    BOOST_CHECK_MESSAGE(1 == 2, "seems that 1 != 2"); // test échoue avec un message, mais il continue
4    BOOST_CHECK_THROW(throw std::runtime_exception("blablabla"), std::runtime_exception); // test de throw
5    BOOST_ERROR("message d'erreur"); // message d'erreur et erreur enregistrée
6
7    BOOST_CHECH_EQUAL(2+2, 4);  // test
8
9    BOOST_REQUIRE(true);     // doit être vrai pour continuer ce test.
10 }
```

Boost

<boost/thread.hpp>

Objectif

Propose une abstraction plate-forme pour la manipulation des threads.
Propose des manipulateurs haut niveau pour la gestion des threads.

Il faut préalablement compiler la partie correspondante de Boost ("with-thread dans bjam), et linker avec "boost_thread" lors de la création du programme.

Quelques objets de "boost : :thread"

```
1 boost::thread t;       // un thread
2 boost::thread_group g;    // groupe de threads
```

Boost

<boost/thread.hpp> : exemple de functor

Objet fonctionnel dont le corps sera exécuté dans un thread

```
1  #include <boost/thread/thread.hpp>
2  #include "boost/date_time/posix_time/posix_time_types.hpp"
3  using namespace boost::posix_time;      // pour les sleeps
4
5    struct my_functor {
6      int init_value1;
7      my_functor(int init_value1_) : init_value1(init_value1_) {}
8    // corps du thread
9      void operator()()
10     {
11       boost::this_thread::disable_interruption di;
12       {
13         std::cout << "Thread (functor) inited with " << init_value1  << std::endl;
14         std::cout << "Thread id is " << boost::this_thread::get_id() << std::endl; // identifiant
15       }
16
17       int count = 0;
18       while(!boost::this_thread::interruption_requested()) // test de demande d'interruption
19       {
20         // section sans interruption
21         boost::this_thread::disable_interruption di;
22         {
23           std::cout << "." << count++ << std::endl;
24           if(count % init_value1)
25             std::cout << std::endl;
26         }
27         boost::this_thread::sleep(time_duration(0,0,0,100)); // sleep de 2 secondes
28       }
29     }
30   };
```

Boost

<boost/thread.hpp> : exemple d'appel

Programme de test (sur un groupe de threads)

```
1  void test() {
2    boost::thread_group g;     // groupe de thread
3
4    for(int i = 1; i < 10; i++)
5    {
6      boost::thread *my_thread = new boost::thread(my_functor(i*300));
7      //boost::thread my_thread(my_functor(i*300));
8      g.add_thread(my_thread);
9    }
10   std::cout << "Init end" << std::endl;
11   char c = 0;
12   std::cout << "Press a key" << std::endl;
13   while(!(std::cin >> c)) {
14   }
15
16   g.interrupt_all();  // lance un signal d'interruption
17   g.join_all();      // attend la fin de tous les threads
18
19   std::cout << "end of all threads" << std::endl;
20 }
```

Optimisation

- Questions générales
- Conteneurs
- Optimisations logicielles

Optimisation

Questions générales autour du C++

"Premature optimization is the root of all evil" (Donald Knuth)

Que veut dire optimisation ?

On optimise toujours en fonction d'une ou plusieurs contraintes... Ex. : temps de calcul, occupation mémoire, algorithmique...
Les contraintes sont parfois contradictoires et/ou conflictuelles.

Definition (Coût algorithmique)

Coût théorique de l'algorithme exprimé par rapport à la taille N des données traitées et exprimé en $O(\dagger(N))$, pour $N \in \mathbb{N}^*$.

$\dagger$: 1 (temps constant), identité, polynôme, *log*...

Optimisation

Questions générales autour du C++

Rappels importants

1. Le coût est asymptotique, c'est-à-dire exprimé pour N grand (ce qui n'est pas systématiquement le cas).
2. $O(\dagger(N))$ est vrai à une constante près, qui peut être potentiellement importante !

Optimisation

Questions générales autour du C++

Questions classiques :

1. Le C++ est plus lent que le C : vrai ou faux ?
   Réponse mitigée. D'expérience : vrai pour des parties isolées de code, faux pour des projets d'envergure importante. Même débat entre le C et l'assembleur par exemple...
2. Où et quand optimiser ?
   Une fois qu'on a enfin un truc qui fonctionne, selon un design relativement correct.
   Remarque : les contraintes influencent nécessairement la phase de design, donc inutile de faire de l'optimisation logicielle trop tôt.
3. Comment optimiser ?
   Encore une fois, selon la contrainte.

Nous allons voir quelques éléments d'optimisation algorithmique et logicielle...

Optimisation

Choix des conteneurs

Alerte ! Alerte ! Alerte !

Le choix du conteneur est FONDAMENTAL !

Le choix est dirigé par l'utilisation qu'on compte en faire :

1. Si l'ordre d'insertion est important
2. Si on va parcourir l'ensemble contenu (coût et performance des itérateurs)
3. Si on va insérer des éléments relativement à d'autres (et non simplement en début ou fin)
4. ...

Les coûts d'accès des conteneurs standards sont également standards ! Donc garantie multi plate-forme.

Optimisation

Choix des conteneurs (1)

std : :vector

1. ajout d'élément en début ou fin en $O(1)$
2. accès aux éléments en $O(1)$
3. requête de taille en $O(1)$
4. parcours en $\alpha_v O(N)$, avec $\alpha_v \approx 1$, avec possibilité d'avoir $\alpha_v < 1$
5. insertion d'élément en $O(N)$ (recopie) ! dramatique, ne pas faire avec vecteur...

Optimisation

Choix des conteneurs (2)

std : :list

1. ajout d'élément en début ou fin en $O(1)$
2. accès aux éléments en $O(N)$: mauvais
3. requête de taille en $O(1)$ à vérifier
4. parcours en $\alpha_l O(N)$, avec $\alpha_l \approx 1$, mais $\alpha_l > \alpha_v$
5. insertion d'élément en $O(1)$! bon point !

Optimisation

Comprendre et gérer la bidouille de compilateurs

Idée

Faire en sorte que le compilateur aille plus loin dans la génération d'un code rapide et/ou compact.

Code rapide :

- Activer les flags d'optimisation
  - Choix du compilateur ! !
  - Options d'optimisation de plusieurs niveaux (GCC : O1, O2, 03, O45... active un ensemble d'optimisations)
  - Choix du processeur cible
  - Utilisation automatique de librairies externes (open MP, etc)
  - Profondeur des inlines
  - Utilisation des fonctions intrinsèques
  - Optimisations globales à l'édition de lien
  - ...

Avantage important : méthode d'optimisation non-intrusive

Optimisation

Comprendre et gérer la bidouille de compilateurs

Code rapide (suite) :

- ...
- Utilisation des inlines
  1. Méthode d'optimisation intrusive !
  2. Pas vraiment une optimisation : le choix de l'inlining reste le choix du compilateur
- Utilisation de portions de code assembleur
  1. Très performant (souvent, mais pas toujours)
  2. Très spécifique
  3. Dur
  4. Très dur même...
  5. Difficile à maintenir
  6. Plate-forme dépendant

Formation C++ avancée

Synthèse de la formation

Synthèse de la formation

Contenu

Un outil à maîtriser

Tenir compte des risques

Le code est meilleur et plus compact mais :

- risque d'*hubris*[a] (donc perte de focalisation)
- problèmes de lourdeur (design, dev, exécution)

---

a. Chez les Grecs, tout ce qui, dans la conduite de l'homme, est considéré par les dieux comme démesure, orgueil, et devant appeler leur vengeance.

Risque majeur :

Quand on est lancés, pourquoi s'arrêter à ce dont on a besoin ? !

Savoir-faire

Pour être performant, il faut :

- maîtriser le plus grand nombre de “trucs”
- en connaître les avantages/inconvénients
- spécifier dès le début (notamment les perfs)

Et pour ça

le mieux est d'en faire beaucoup et d'en jeter beaucoup. Itérer.

La généricité “light”

“design for change”

- Commencer simple (YAGNI) [a]
- Rendre le code generic-friendly
- Concevoir pour l'évolutif

a. "You Ain't Gonna Need It"

L'important

c'est de se fermer le moins d'options, tout en restant optimal sur ce qu'on fait *maintenant*.

Merci de votre attention

Release early, release often...