Plan

Introduction syntaxique

la classe

Héritage

templates, exceptions et RTTI

Notes sur C++

Philippe Laroque

UCP/ETIS/CNRS

Janvier 2007

Plan I

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction

Plan II

- Surcharge d'opérateurs

3. Héritage en C++
   - Héritage simple
   - Liaison dynamique
   - Héritage multiple

4. Templates, exceptions et RTTI
   - Les templates
   - Les exceptions
   - Run-Time Type Identification

Introduction Syntaxique C / C++

C++ est un sur-ensemble de C
(Presque) tout programme C est compilable en C++: problème des nouveaux mots réservés
Principales différences (hormis le paradigme Objet):

- Une vérification de type beaucoup plus forte
- La notion de référence
- La surcharge d'opérateurs et de fonctions
- La gestion des entrées-sorties
- La gestion de l'allocation dynamique de mémoire
- ...

Forme d'un Programme

ZONE DE DECLARATIONS
// inclusion de fichiers en-tête pour les
// déclarations d'entités externes
// déclarations d'entités utilisables dans le
// module (static), ou exportables
// déclarations des fonctions utilisées
ZONE DE DEFINITION
// définition des fonctions du module
// éventuellement, fonction `main()`
Toute entité doit avoir été déclarée préalablement à son utilisation

Exemple simple

```
#include <iostream>
using namespace std;
int main() {
  cout << "Premier programme C++\n";
  return 0;
}
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Types Simples

Les types simples sont de deux sortes: prédéfinis ou énumérés

Types prédéfinis discrets:

- char (1 octet): 'e', '\n', '\0'
- short int (2 octets)
- int (1 mot)
- long int (4 octets)

Tous compatibles entre eux. Peuvent être non signés (unsigned)

Types prédéfinis flottants:

- float (4 octets)
- double (8 octets)

Types Enumérés:

```
enum Couleur { rouge, vert, bleu };
```

Compatibles avec les scalaires discrets. Prennent des valeurs croissantes à partir de 0:

rouge == 0, vert == 1, bleu == 2

Déclarations de types

- Introduite par le mot réservé `typedef`
- Ce n'est qu'une déclaration de synonymie
- On donne d'abord le nom du type existant, puis celui du nouveau

```
typedef unsigned char Carac;
typedef int Tableau[10];
```

Déclarations de Variables

- Toute variable doit avoir été déclarée lors de sa première utilisation
- La déclaration consiste à mentionner le type de la variable, puis son nom

```
int i;
Couleur c;
Tableau tab;
```

- On peut déclarer d'un coup plusieurs variables, n'importe où dans le code

```
int i, j, tab[3];
```

- On peut initialiser une variable que l'on déclare

```
unsigned short us = 3;
char c = 65; // 'A'
```

- Les constantes se déclarent en plaçant le mot `const` avant le type:

Statut des Variables

- `auto`: statut par défaut, stocké dans la pile
  - durée de vie: exécution du bloc englobant la déclaration
  - portée: bloc englobant la déclaration, sinon le programme
- `static`: stocké dans le tas
  - durée de vie: tout le programme
  - portée: bloc englobant la déclaration, sinon le fichier
- `extern`: correspond à une variable automatique définie dans un autre module
- `register`: stocké dans un registre du processeur
  - durée de vie: exécution du bloc englobant la déclaration
  - portée: bloc englobant la déclaration. NB: ON NE PEUT PAS EN PRENDRE L'ADRESSE !

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Déclarations de Fonctions

- Deux statuts: globale ou limitée au fichier (`static`)
- On mentionne le prototype de la fonction:

1. le type de la valeur de retour
2. le nom de la fonction
3. la liste des types des arguments entre parenthèses

```
double moyenne(double, double);
int nIeme(Tableau, int);
```

Opérateurs

Grande richesse du langage: ils sont surchargeables
Très nombreux, dans principalement trois domaines:

1. Les opérateurs arithmétiques
2. Les opérateurs logiques
3. Les opérateurs travaillant sur des tableaux de bits

Opérateurs Arithmétiques

- Les opérateurs arithmétiques classiques (+, -, *, /), utilisables en conjonction avec '='

```
long l = 1;
l += 3;
```

- L'opérateur / donne le résultat d'une division entière si les deux opérandes sont entiers; le reste s'obtient avec %:

```
int i = 10; nt j = 3;
int k = i / j; // 3
int l = i % j; // 1
```

- Les opérateurs d'incrémentation (décrémentation), ++ et –

```
int i = 2;
int j = i++; /* j == 2, i == 3 */
i = ++j; /* j == 3, i == 3 */
```

Opérateurs Logiques

- Pas de booléens en C++: toute valeur non nulle est considérée comme vraie
- Les opérateurs classiques <, <=, >, >= renvoient 1 ou 0
- Egalité et différence: ==, != renvoient aussi 1 ou 0
- La négation logique s'exprime par '!'
- Connecteurs logiques binaires: && (et), || (ou)

Opérateurs au Niveau "bit"

- décalage à gauche '<<', à droite '>>'
- connecteurs binaires et '&', ou '|', xor '^'
- complément à 1: '~'

```
i = 3;       /* 011 */
j = 4;       /* 100 */
k = i && j; /* 1 */
k = i & j;  /* 0 */
k = i || j; /* 1 */
k = i | j;  /* 111: 7 */
k = i ^ j;  /* 111 */
k = ~i;     /* 111..100 */
```

Opérateurs d'Accès

- L'adresse d'un objet s'obtient par l'opérateur UNAIRE '&'
- L'objet adressé par un pointeur s'obtient par l'opérateur UNAIRE '*'

```
int i; /* i = 12345'67, &i = 100 */
int *p; /* p = 2345678, &p = 104 */
/* "l'objet adressé par p est un entier" */
p = &i; /* p = 100 */
i = 3; /* i = *p = 3 */
*p = 2; /* i = *p = 2 */
```

Opérateur Ternaire

- L'opérateur `?:` est le seul opérateur ternaire du langage
- Sa valeur est celle du second opérande si le premier est non nul, celle du troisième opérande sinon ("if")

```
int i = 0;
int j = 2;
int k = (j - i) < 10 ? j+1: 2*i; /* 3 */
```

Allocation de Mémoire

- Les traditionnelles fonctions `malloc()` et `free()` sont remplacées par deux opérateurs:
- Syntaxe:

```
ptr = new <type>; // retourne 0 si échec
delete ptr;
```

- Il est possible d'allouer de la mémoire pour un vecteur d'Objets:

```
tab = new <type> [<nb>];
delete [] tab;
```

- `<type>` peut être prédéfini ou défini par l'utilisateur

set_new_handler() I

- La fonction `set_new_handler()` sert à spécifier une fonction à appeler en cas d'échec de l'allocateur `new`

N.B.

Pourra être avantageusement remplacée par une gestion d'exception

- Exemple d'utilisation:

```
#include <stdio.h>
#include <stdlib.h>
#include <new.h>
void noRoom () {
  cerr << "** Plus de place\n";
  exit(1);
}
```

set_new_handler() II

```
int main() {
  set_new_handler(noRoom);
  int *i = 0;
  int nb = 0;
  while (i = new int[10000])
    cout << nb++ << endl;
  cout << "Ceci ne sera jamais affiche\n";
  return 0;
}
```

Opérateur de Résolution de Portée

L'opérateur '::' (*scope qualifier operator*) permet l'accès à une entité globale même si elle est masquée par une entité locale de même nom

```
... // portée globale
char * s = "Global String";
...
f() {
...
char * s = "local string";
printf("%s\n%s\n",::s, s);
...
}
```

1. '::' est aussi (et surtout) utilisé lors de la manipulation des classes d'Objets
2. L'opérateur ne permet que l'accès à l'entité globale et non aux éventuels intermédiaires

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Structures de Contrôle

Essentiellement 4:

1. if: structure conditionnelle classique
2. while et do...while
3. for: plus puissante que dans les autres langages ou elle est définie
4. switch: permet de choisir une branche d'une alternative multiple

Des mots réservés permettent de contrôler le déroulement des structures itératives:

- `continue`: le contrôle passe à l'itération suivante
- `break`: le contrôle passe à l'instruction qui suit la boucle
- `return`: le contrôle passe au sous-programme appelant
- `exit`: le programme se termine

La Structure if

- Syntaxe: `if (<cond>) <sioui> [ else <sinon>]`

```
if (x == 2) {
  ...
} else
  x = 2;
```

- `<cond>` est une expression quelconque renvoyant un scalaire discret; 0 est la seule valeur "fausse"
- `<sioui>` et `<sinon>` sont des instructions simples (suivies de ";") ou composées (plusieurs instructions entre "{}"):

```
<instr>:== <instrSimple> | <instrComposée>
<instrSimple>:== <affectation>';' |
<appelFonction>';'
<instrComposée>:== '{'<instr>+ '}'
```

La Boucle while

- Syntaxe: `while (<cond>) <instr>`

```
while (! estVide(maPile)) {
  elt = pop(maPile);
  affiche(elt);
}
```

La Boucle for

- Syntaxe: `for (<condInit>;<testFin>;<incr>) <instr>`

```
for (ptr = first(maListe); ptr; ptr = next(ptr)) {
  nbElts++;
  push(maPile,val(ptr));
}
```

La structure switch

- Syntaxe:

```
switch (<expr>) {
[ case <val>:  <instr>*]+
[default:  <instr>*]
}
```

```
switch (caracLu) {
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
  val = caracLu - '0';
  break;
case '\n':
  val = EOF;
  break;
default:
  val = -2;
}
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Le Pré-processeur

- Intervient AVANT la compilation proprement dite
- Reconnaît une instruction si elle commence par un '#' en colonne 1
- 3 sortes de tâches:
  1. Inclusion de fichiers source avant compilation
  2. Définition de macros-instructions et de constantes
  3. Gestion de compilation conditionnelle (écriture de code portable, ...)

La Directive #include

- Permet d'éviter la réécriture de toutes les déclarations de fonctions externes manipulées dans le module
- On inclut en général des fichiers dits "en-tête", d'extension ".h"
- Syntaxe:

```
#include <iostream.h>
#include <sys/types.h>
#include "mesOutils.h"
```

Remarques

Les fichiers listés entre '<>' sont cherchés dans le répertoire `/usr/include`
Les fichiers entre "" sont d'abord recherchés dans le répertoire courant

Que Met-on dans un Fichier "en-tête"?

1. Des types définis par l'utilisateurs (des classes). Un type de donnée ne peut pas être défini plusieurs fois lors d'une même compilation
2. Des prototypes de fonctions utiles
3. Des déclarations de variables externes, définies dans un autre module
4. Eventuellement, des appels à inclure d'autres fichiers "en-tête"

La Directive #define

- Permet de définir des macros, dont le corps remplace l'appel avant compilation
- Une macro sans argument est une constante
- Les macros sont peu utilisées en C++, où l'on dispose de constantes au niveau du compilateur et de fonctions en-ligne

```
#define BUFSIZ 512
#define max(a,b) ((a)<(b))?(b):(a)
```

Les Directives #if...

- Permettent de sélectionner une portion de code à compiler en fonction de la valeur d'une constante du pré-processeur

```
...
#ifdef __STDC__
#define CONST const
#else
#define CONST
#endif
```

- Permet aussi de s'assurer qu'un même fichier en-tête n'est pas inclus plusieurs fois lors d'une même compilation:

```
#ifndef __MONFICH_H
#define __MONFICH_H
... /* contenu du fichier */
#endif
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Pointeurs, Tableaux et Références

- Les pointeurs sont des éléments fondamentaux du langage
- La notion de tableau n'existe pas en tant que telle (pas d'implantation d'un "type abstrait" Tableau)
- On manipule les tableaux à l'aide (explicite la plupart du temps) de pointeurs
- La notion de référence permet de s'affranchir de la nécessité de manipuler explicitement des pointeurs

Pointeurs et Tableaux

Un tableau se déclare de deux façons:

1. soit en en fixant la taille définitivement, lors de la création
2. soit en allouant dynamiquement la mémoire nécessaire au stockage de ses éléments

```
int tab[10];
int tab2[] = { 1, 2, 1, 0 };
int * tab3 = new int[tab2[1]];
```

Le mode de déclaration n'influe pas sur la méthode d'accès aux éléments: par définition, `tab[i] == *(tab + i)`
Modification possible du tableau UNIQUEMENT si déclaré comme pointeur:

```
tab3 = tab2; /* OK */
tab2 = tab; /* interdit */
```

Les Références

- Notion très importante en c++
- Une référence sur un Objet de type <T> est déclarée et définie en même temps
- L'Objet référencé doit avoir été créé au préalable
- Une référence ne peut changer d'Objet référencé:

```
int i = 1;
int * p = &i;
int & r = i; // référence sur int
*p = 2; // i vaut maintenant 2
r = 3; // i vaut maintenant 3
p = ... // modification de p
&r = ... // modification interdite
```

Remarques

1. On ne peut pas créer de tableau de références
2. Les références servent beaucoup dans la définition de fonctions

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Traitement des Fonctions

- Un même identificateur peut servir à dénoter plusieurs fonctions différentes: on parle de surcharge
- On peut accéder à l'adresse d'une fonction. Ceci permet de passer des fonctions comme arguments d'autres fonctions: `typedef int (*fComp)(void*,void*);`
- Les paramètres des fonctions peuvent se voir affecter une valeur par défaut
- Il est possible de définir des fonctions à nombre variable d'arguments
- Les fonctions courtes peuvent être étendues "en-ligne"

Surcharge de Fonctions

- Le même identificateur peut désigner un nombre quelconque de fonctions différentes
- Seule condition: chaque signature (prototype sans type de retour) doit être unique, le type de retour n'est pas significatif pour la résolution des éventuels conflits par le compilateur:

```
int max(int,int);
double max(double,double);
MyClass max(MyClass, MyClass); // user-defined
unsigned max(int,int); // error
```

Passage de Paramètres par Référence

- En C, le seul mode de passage de paramètres est le passage par valeur: chaque paramètre passé dans l'appelant est dupliqué en une variable locale à l'appelé (qui disparaît donc lors du retour)
- En C++, passage d'une référence possible: permet de masquer l'utilisation de pointeurs

```
void incremente (int& i) {
  i += 1;
}
...
int k = 3;
incremente(k);
// k vaut maintenant 4
```

Peut être utilisé conjointement avec `const` pour éviter la duplication de structures de grande taille: les modifications du paramètre dans la fonction sont alors interdites par le

Retour d'une Référence

- En C, le résultat d'une fonction est une r-value
- En C++, une fonction peut retourner une référence sur un Objet de type quelconque. L'appel de la fonction se comporte donc comme une l-value:

```
extern MyObj& nth(MyTab, int);
...
nth(monTab,3) = monObj;
```

Le principe de fonctionnement du retour d'une fonction est similaire à celui du passage de paramètres:

- par valeur, duplication de l'Objet retourné
- par référence, pas de duplication

Valeurs par Défaut de Paramètres

- A spécifier dans le prototype de la fonction
- Pour éviter les ambiguïtés, seul(s) le(s) dernier(s) paramètre(s) peuvent recevoir une valeur par défaut

```
int copy(void *, void*, int = 0);
...
copy (src,dest); /* shallow copy */
copy (src2,dest2,deep); /* deep copy */
```

Fonctions à Nombre Variable de Paramètres

Inclure le fichier `stdarg.h`
placer des "..." pour désactiver le contrôle de type lors de l'appel

```
#include <stdarg.h>
int plus (int first, ...) {
  va_list ptr;
  int res = first;
  int current;
  va_start(ptr,first);
  while (current = va_arg(ptr,int))
    res += current;
  va_end(ptr);
  return res;
}
int main() {
  printf("%d\n", plus(1,2,3,4,5,0));
```

Fonctions inline

- Une fonction déclarée `inline` se comporte presque comme une macro (substitution du corps à l'appel de la fonction)
- Sémantique du passage de paramètres "fonctionnel" préservée: beaucoup plus sûre qu'une macro

```
inline int goodMax(int i, int j) {
  return (i < j)? j: i;
}
#define badMax(i,j) (i) < (j)? (j): (i)
int a = 1, b = 2;
int c = goodMax(a++, b++);
// ici, a == 2, b == 3, c == 2
c = badMax(a--, b--);
// ici, a == 1, b == 1, c == 2!!!
```

Le spécificateur `inline` n'est traité par le compilateur que comme une indication: aucun message si échec.
Le corps doit être visible lors de chaque appel: les fonctions `inline` sont dans les fichiers en-tête.

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

La Librairie des Flots (iostream)

- Les entrées - sorties en C++ peuvent se faire via les fonctions standard du C
- Inconvénient: non extensibles aux types définis par l'utilisateur
- Des entrées-sorties polymorphes peuvent être réalisées grâce aux classes `istream` (flot d'entrée) et `ostream` (flot de sortie), et à plusieurs fonctions et opérateurs définis sur ces classes (voir `iostream.h`)
- Les deux opérateurs les plus fréquemment utilisés: $<<$ (pour l'écriture) et $>>$ (pour la lecture):

```
ostream& operator << (ostream&, <type>);
istream& operator >> (istream&, <type>);
```

Trois flots prédéfinis:

1. `cin` (id. stdin), pour l' entrée standard
2. `cout` (id. stdout), pour la sortie standard
3. `cerr` (id. stderr), pour la sortie diagnostic

Les opérateurs d'entrée-sortie

Inclusion de `iostream.h` nécessaire

Opérateurs polymorphes: fonctionnent sur tous les types prédéfinis

```
#include <iostream>
using namespace std;
main() {
  int v1, v2;
  cout << "** Enter two int values: ";
  cin >> v1 >> v2;
  cout << "average of " << v1 << " and "
       << v2 << " is " << (v1 + v2) / 2.0
       << '\n';
  return 0;
}
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

La Cohabitation C/C++

- La possibilité de récupérer (presque) sans modification des librairies de fonctions C dans un programme C++ est une des raisons du succès de ce langage
- Tout compilateur C++ définit la constante __cplusplus: `#define __cplusplus`
- Le problème de l'utilisation de fonctions C en C++ se résumera, pour l'utilisateur, à un problème de prototypes
- On peut spécifier dans un programme C++ qu'une fonction est disponible dans une librairie compilée en C en mentionnant la fonction avec le mot réservé `extern "C"`

Utilisation de Fonctions C en C++

Supposons qu'un fonction moyenne soit disponible dans une bibliothèque C:

```
double moyenne (x, y)
double x, y; {
  return (x + y) / 2.0;
}
```

Pour l'utiliser dans un module C++:

```
#include <iostream>
using namespace std;
extern "C" double moyenne(double, double);
int main() {
  double a,b;
  cin >> a >> b;
  cout << "Moyenne: " << moyenne(a,b)
       << '\n';
  return 0;
}
```

Fichiers en-tête pour C et C++

- On peut placer plusieurs déclarations dans un `extern "C"`
- En testant la constante `__cplusplus`, on peut écrire des ".h" utilisables en C et en C++:

```
#if __STDC__ | __cplusplus
#ifdef __cplusplus
extern "C" {
#endif
/* ANSI-C / C++ style declarations */
double moyenne(double, double);
...
#ifdef __cplusplus
}
#endif
#else
/* old C style declarations */
double moyenne();
...
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

La Classe en C++

Une classe n'est pas un Objet: seules les instances de cette classe en sont

On peut voir la notion de classe comme l'extension de celle de type

Lors de la déclaration de la classe, on définit:

1. La (les) classe(s) dont elle dérive, s'il y a lieu
2. Sa structure interne: champs, variables d'instances, données membres
3. L'ensemble des opérateurs qui agissent sur la classe: méthodes, routines, fonctions membres

Dans les deux catégories ci-dessus, les éléments visibles (interface de la classe) et cachés

Conventions:

1. Une classe se déclare en général dans un fichier en-tête séparé portant le nom de la classe
2. Les noms de classe commencent en général par une majuscule, les noms de fonctions membres par une

Structure

Une classe est divisée en une à trois parties. Dans chaque partie, on peut trouver des données et / ou des fonctions

- La partie privée (introduite par `private` ou rien) contient les membres cachés au monde extérieur à la classe
- La partie protégée (introduite par `protected`) contient les membres accessibles aux classes dérivées mais cachés aux clients ordinaires de la classe
- La partie publique (introduite par `public`) contient l'interface de la classe: accès libre

Chaque partie est optionnelle
Une `struct` est une classe avec par défaut uniquement une partie publique. Le mot réservé existe encore.

Exemple de Classe

```
// FirstExample.h
class FirstExample {
  private:   // optionnel
    int _priv; // invisible hors de la
               // portée de la classe
  protected:
    int _prot; // accessible depuis les
               // classes dérivées
  public:
    int _publ; // pas de restrictions
};
```

Programme de test

```
// tFirstExample.C
#include "FirstExample.h"
int main() {
  FirstExample myFirstEx;  // déclaration: sans "c
  cout << myFirstEx._publ  // Ok.
       << myFirstEx._prot  // compiler error
       << myFirstEx._priv; // idem
  return 0;
}
```

Protection et Portée

La visibilité du contenu des parties privée et protégée d'une classe dépend uniquement de l'endroit où s'effectue la tentative d'accès:

- Si on est dans la portée de la classe (lors de sa déclaration, ou à l'intérieur d'une fonction membre), tout est accessible
- Si on est dans la portée d'un classe dérivée, les parties publique et protégée sont accessibles
- Sinon, seule la partie publique est accessible

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Membres d'une Classe

Deux catégories de membres:

1. Les données membres, ou variables d'instances, ou champs
2. Les fonctions membres, ou routines, ou méthodes

Un membre peut être constant (attribut `const`). La signification de cet attribut est différente selon qu'il s'agit d'une donnée membre ou d'une fonction membre
Un membre peut être statique (attribut `static`). Là encore, signification différente

Fonctions Membres

- Une fonction membre se déclare en plaçant son prototype dans la partie souhaitée de la classe
- Le corps de la fonction est en général rejeté dans un fichier ".C" (ou ".cpp"). Exception: les fonctions membres `inline`
- L'accès à la fonction membre se fait, comme pour les données, par notation pointée

Exemple de traitement de fonctions membres

```
// WithFunctionMembers.h
class WithFunctionMembers {
  public:
    int exampleFun();
};
// WithFunctionMembers.C
#include "WithFunctionMembers.h"
int WithFunctionMembers:: exampleFun () {
  ...
}
// tWithFunctionMembers.C
#include "WithFunctionMembers.h"
int main() {
  WithFunctionMembers wf;
  ...
  wf.exampleFun();
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Aperçu sur les Constructeurs

- Un *constructeur* est une fonction membre spéciale appelée implicitement lors de la création d'une instance
- Le nom du constructeur est celui de la classe elle-même
- Le constructeur est la seule fonction membre qui n'a aucun type de retour (même pas `void`)
- Il peut y avoir un nombre quelconque de constructeurs dans une classe. Ils doivent différer, soit par le nombre de paramètres, soit par le type de l'un d'entre eux
- Une zone spéciale, appelée *zone d'initialisation*, permet d'initialiser les parties complexes d'un Objet:

```
<nomClasse> (<listePars>): <zoneInit> {
  <corps>
}
```

On peut y placer plusieurs initialisations, séparées alors les unes des autres par des virgules, comme pour des déclarations multiples (voir plus loin)

Fonctions Membres inline

- Une fonction membre `inline` doit être définie dans le ".h"
- Elle peut être définie dans la déclaration de la classe: il n'est alors pas nécessaire de mentionner le mot réservé inline
- Elle peut être définie après la déclaration de la classe: la définition doit alors commencer par `inline`

```
class WithInlines {
  ...
  int inl1() { return 3; }
  int inl2();
  ...
};
inline int WithInlines:: inl2() { return 3; }
```

Pour des raisons de clarté, la seconde solution est souvent préférable

La pseudo-variable this

- Une fonction membre (méthode) sert à répondre à un message qu'un Objet a reçu
- Une fonction membre fait implicitement référence à une instance particulière. Lorsque cette référence doit être explicitée, on fait appel à une pseudo-variable, `this`, dont la valeur est l'adresse du receveur du message:

```
Compteur& Compteur :: incr() {
  _val ++;
  return *this;
}
...
Compteur c;
c.incr().incr().incr();
```

Membres Constants

- Une donnée membre constante reçoit une valeur initiale lors de sa création. Cette valeur ne peut être modifiée durant toute la durée de vie de l'instance
- La déclaration d'une donnée membre constante est précédée du mot réservé `const`
- Une fonction membre constante garantit qu'elle n'effectue aucune modification du receveur: sécurité pour l'utilisateur. Conséquence: seule une fonction membre constante peut être appliquée à un Objet constant

```
class WithConstMembers {
  ...
  const int _ci;
  int nonConstFun();
  int constFun() const;
  ...
};
```

Programme de test

```
int main() {
  WithConstMembers nCst;
  const WithConstMembers cst = nCst;
  nCst.constFun();     // Ok.
  nCst.nonConstFun(); // Ok.
  cst.constFun();      // Ok.
  cst.nonConstFun();  // compiler error (warning)
  return 0;
}
```

Initialisation des données membres constantes

Les données membres constantes sont initialisées obligatoirement dans la zone d'initialisation du constructeur, pas dans le corps:

```
// WithConstMembers.h
class WithConstMembers {
  ...
  WithConstMembers(int = 0);
  ...
};
// WithConstMembers.C
#include "WithConstMembers.h"
WithConstMembers:: WithConstMembers (int def)
: _ci (def) // zone d'initialisation
{
  // corps: initialisation membres ordinaires
  ...
```

Membres Statiques

- Un membre statique se déclare avec le mot réservé `static`

donnée membre == variable d'instance
donnée membre statique == variable de classe

- Contrairement aux données membres ordinaires, une donnée membre statique n'est pas dupliquée à chaque création d'instance: il n'en existe qu'un exemplaire pour la classe. Chaque instance y accède de la même façon qu'aux données membres classiques

Fonctions membres statiques

- Une fonction membre statique n'est pas non plus liée à une instance particulière (pas de `this`). On ne peut donc y faire référence qu'à des variables de classe
- Une fonction membre statique peut être appelée, soit comme un membre ordinaire, soit comme une fonction globale
- Un membre statique (donnée comme fonction) peut être déclaré dans n'importe quelle partie de la classe.
- On peut voir un membre statique comme une entité globale ayant un accès à toutes les informations d'une classe et dont la portée peut être limitée à cette classe

Manipulation des membres statiques

```
// WithStaticMembers.h
class WithStaticMembers {
  ...
  static int StaticInt;
  static int getStaticInt();
  ...
};
// WithStaticMembers.C
#include "WithStaticMembers.h"
int WithStaticMembers:: StaticInt = 100;
int WithStaticMembers:: getStaticInt() {
  return StaticInt;
}
// appels possibles (equivalents):
WithStaticMembers wsm;
wsm.getStaticInt();
```

Plan

1 Introduction Syntaxique C / C++
- Types et variables
- Fonctions et opérateurs
- Structures de contrôle
- Le pré-processeur
- Pointeurs, tableaux et références
- Retour sur les fonctions
- Entrées-sorties en C++
- C et C++

2 La classe en C++
- Définition de classes C++
- Membres d'une classe
- Construction (initialisation)
- Les friends
- Construction (2) et destruction
- Retour sur la Construction
- Surcharge d'opérateurs

3 Héritage en C++

Les Friends

- Une fonction extérieure à une classe ne peut accéder qu'aux membres publiques de la classe
- Ceci peut s'avérer gênant dans certains cas, notamment quand la fonction doit accéder aux données internes de deux (plusieurs) classes différentes
- On peut alors spécifier, dans chaque classe accédée, la fonction comme amie (`friend`)

Exemple simple

```
// Vecteur.h
class Matrice;
class Vecteur {
  ...
  friend Vecteur * mult (Matrice*, Vecteur*);
};
// Matrice.h
class Vecteur;
class Matrice {
  ...
  friend Vecteur * mult (Matrice*, Vecteur*);
};
// ailleurs ...
Vecteur * mult (Matrice * m, Vecteur* v) {
  // Bien que globale, la fonction peut
  // accéder aux données privées de m et v
  ...
```

Classes amies

- Une fonction membre d'une classe peut être amie d'une autre
- Quand toutes les fonctions membres d'une classe A doivent être amies d'une classe B, B peut mentionner A elle-même comme amie:

```
class B {
  friend class A;
  ...
};
class A {
  ... // tous les membres accèdent
      // aux données privées de B
};
```

Les classes "outils" peuvent ne comporter qu'une partie privée et déclarer les classes qui les utilisent comme amies

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Construction / Destruction

On peut déclarer plusieurs constructeurs dans une classe:

```
class WithConstructors {
  private:
    int _i;
    float _f;
  public:
    WithConstructors();
    WithConstructors(int, float);
    ...
};
...
WithConstructors wc1; // constr # 1
WithConstructors wc2(1, 3.14159); // constr # 2
```

En revanche, un seul destructeur

Invocations

- Le constructeur est appelé *implicitement* lors de la création de l'Objet, le destructeur lors de sa disparition
- Le destructeur *peut* être appelé explicitement (à éviter)

Il faut *toujours* avoir un constructeur sans argument dans une classe (initialisation de tableaux d'objets)

Plan

1 Introduction Syntaxique C / C++
- Types et variables
- Fonctions et opérateurs
- Structures de contrôle
- Le pré-processeur
- Pointeurs, tableaux et références
- Retour sur les fonctions
- Entrées-sorties en C++
- C et C++

2 La classe en C++
- Définition de classes C++
- Membres d'une classe
- Construction (initialisation)
- Les friends
- Construction (2) et destruction
- Retour sur la Construction
- Surcharge d'opérateurs

3 Héritage en C++

Construction et Conversion

Tout constructeur admettant un paramètre unique est appelé implicitement en tant qu'opérateur de casting:

```
class Int {
  ...
public:
  Int(int = 0);
  ...
};
...
void traiteInt(Int);
Int j;
traiteInt(3); // Ok.
```

Dans l'exemple ci-dessus, il y a d'abord promotion de `int` en `Int` via le constructeur, puis exécution du corps de `traiteInt()`

Construction d'Objets Complexes

- Un Objet peut avoir d'autres Objets comme données membres
- Ces membres doivent eux-mêmes être initialisés par le constructeur de la classe à laquelle ils appartiennent, s'il existe
- Pour cela, on place l'initialisation de ces membres dans la zone d'initialisation du constructeur, en compagnie des membres constants

Exemple simple

```
class Moteur { ... Moteur(int); ... };
class Automobile {
private:
  ...
  Moteur _m;
public:
  Automobile(int);
  ...
};
inline Automobile:: Automobile(int i): _m(i) // ap
{
  ... // reste de la construction
}
```

Le "Copy-Constructor"

Le *copy-constructor* permet d'initialiser une instance à partir d'une autre instance, déjà initialisée

```
<NomClasse> (const <NomClasse> &);
```

Il est appelé (implicitement) dans trois cas de figure:

1. Lors de la définition d'une instance à partir d'une autre
2. Lors du passage (par valeur) d'une instance dans une fonction
3. Lors du retour d'une (valeur d'une) instance depuis une fonction

Exemple simple

```
class X { ... X(const X&); ... };
int fv(X), fr(X&);
X gv(); X& gr();
int main() {
  X anX;
  X another = anX; // premier cas
  int i = fv(anX); // second cas
  int j = fr(anX); // pas d'appel
  another = gv();  // troisième cas
  anX = gr();      // pas d'appel
  return 0;
}
```

Construction par Défaut

- Un copy-constructor "par défaut" est systématiquement créé par le compilateur si le besoin s'en fait sentir. Il effectue une simple copie membre à membre
- Dans certains cas (notamment dans les cas où le constructeur alloue de la mémoire), ce mécanisme peut se révéler insuffisant
- Il est possible d'interdire au compilateur de générer ce constructeur: il suffit d'en définir un dans la zone privée ou protégée de la classe (utile quand on ne veut pas que le compilateur autorise les copies d'instances)
- De la même façon, un opérateur d'affectation "=" et un opérateur de test d'égalité "==" sont fournis, qui effectuent également une affectation (comparaison) membre à membre. Là encore, en surchargeant ces opérateurs, il est possible de contrôler le mécanisme d'affectation / comparaison

Le Destructeur

- Le rôle du destructeur est dual de celui du copy-constructor
- Un seul destructeur par classe (au maximum)
- Le destructeur est appelé implicitement lorsque l'Objet sort de portée; il est néanmoins possible d'appeler explicitement un destructeur (déconseillé) en donnant son nom complet (avec l'opérateur "::")
- Le destructeur porte le nom de la classe, précédé de "~". Il est non typé, comme les constructeurs

Définition d'un destructeur

```
// WithDestructor.h
class WithDestructor {
  ...
  WithDestructor(int, float);
  ~WithDestructor();
};
// WithDestructor.C
#include "WithDestructor.h"
WithDestructor::~WithDestructor() {
  ...
}
```

Quand on définit un copy-constructor, il est souvent indispensable de définir un destructeur et de surcharger "=" et "=="

new et les Constructeurs I

- Avantage crucial de `new` et `delete` par rapport à `malloc()` et `free()`: appel implicite au constructeur (destructeur) s'il existe
- Le constructeur appelé dépendra des arguments éventuellement spécifiés derrière l'appel à `new`:

```
class X {
  ...
public:
  X();
  X(int, int);
  ~X();
  ...
};
...
X * ptr1 = new X;        // appel de X::X()
```

new et les Constructeurs II

```
X * ptr2 = new X(1,2); // appel de X::X(int,int)
...
delete ptr1;           // appel de X::~X();
delete ptr2;           // appel de X::~X();
```

Conséquence: utilisation simultanée des deux types de gestion de mémoire fortement déconseillée !!

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Surcharge d'Opérateurs

- Un opérateur peut être surchargé de la même façon qu'une fonction
- Le nom de l'opérateur est en fait "`operator<symboleDeLOperateur>`"
- La syntaxe d'une surcharge doit être la même que celle de la version prédéfinie de l'opérateur (même arité, même position des paramètres)
- Tous les opérateurs C++ sont surchargeables sauf "." et "?:"

Exemple simple

```
class Complex {
  double re, im;
public:
  Complex(double = 0, double = 0);
  Complex operator+(const Complex&) const;
  ...
};
...
Complex Complex:: operator + (const Complex& z) co
  return Complex(re+z.re, im+z.im);
}
z3 = z1 + 1; // <=> z3 = z1.operator+(Complex(1));
```

Opérateurs Membres / Amis

- La plupart du temps, la surcharge d'un opérateur peut se faire de deux façons différentes:
- soit en tant que membre de la classe (exemple précédent)
- soit en tant qu'ami
- L'avantage de la définition en tant qu'ami est que le premier opérande n'est pas nécessairement une instance de la classe qui redéfinit l'opérateur:

```
z3 = 1 + z2; // interdit pour l'instant
```

Certains opérateurs (typiquement, << et >>) DOIVENT être surchargés en tant qu'amis
Les opérateurs = [] () -> DOIVENT être surchargés en tant que membres

Surcharge et Conversion

- Dans le but de fondre aussi complètement que possible les types définis par l'utilisateur dans le langage, C++ dispose d'opérateurs de conversion
- Ces opérateurs permettent de convertir une instance d'une classe donnée en une variable d'un autre type (prédéfini ou non)
- Ne pas abuser de ces opérateurs: risque de mauvaise utilisation de la classe

Exemple simple

```
class String {
  ...
  operator char* () { return _val; }
};
...
String s = "taratata";
char tab[20];
strcpy(tab, s); // Ok: conversion en char*
```

Surcharge de new et delete

L'intérêt d'utiliser `new` et `delete` plutôt que `malloc()` et `free()` est double:

1. `new` appelle implicitement le constructeur, s'il existe
2. `new` et `delete` sont surchargeables dans toute classe qui désire prendre le contrôle de l'allocation mémoire

Prototypes:

```
void * operator new (size_t);
void operator delete (void*, size_t);
```

La surcharge est particulièrement intéressante dans le cas de classes qui font appel de manière intensive à l'allocation / libération de structures de même taille

Contrôler l'Allocation Mémoire

```
// Node.h
class Node {
  friend class List;
  void * _val;
  Node * _next;
  Node (void* = 0, Node* = 0);
  // Available pointe sur une liste de
  // Node disponibles
  static Node * Available;
  void* operator new (size_t);
  void operator delete (void*, size_t);
};
```

Le paramètre de type `size_t` est entièrement géré par le compilateur, sa valeur représente la taille (en octets) d'une instance de la classe

Quand on n'utilise pas un paramètre, il est préférable de ne pas le nommer dans la liste des paramètres formels

Exemple

```
// Node.C
#include "Node.h"
Node * Node:: Available = 0;
Node * Node:: operator new (size_t) {
  Node * tmp;
  if (Available) {
    tmp = Available;
    Available = Available->next;
  }
  else
    tmp =::new Node;
  return tmp;
}
void Node:: operator delete (void* ptr, size_t) {
  ((Node*)ptr) -> _next = Available;
  Available = ptr;
}
```

Visibilité inter-classes

- Deux classes en relation entre elles doivent pouvoir accéder à leur définition respective.
- Cependant, une même classe ne peut être définie plus d'une fois dans une unité de compilation. L'inclusion mutuelle est donc interdite
- Solution: ne déclarer que l'existence de la classe associée dans le fichier en-tête (".h") de chaque classe, la définition complète n'étant incluse que dans le fichier d'implémentation (".C"). Ceci interdit l'utilisation de la classe associée dans le ".h"

Exemple simple I

```
// Classe1.h
#ifndef __CLASSE1_H
#define __CLASSE1_H
class Classe2;
class Classe1 { ... };
#endif

// Classe2.h
#ifndef __CLASSE2_H
#define __CLASSE2_H
class Classe1;
class Classe2 { ... };
#endif

// Classe1.C
```

Exemple simple II

```
#include "Classe1.h”
#include "Classe2.h”
...

// Classe2.C
#include "Classe2.h”
#include "Classe1.h”
...
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Héritage Simple

- Une classe dérivée hérite de tous les membres de sa classe de base:

Structure(Derivee) == Structure(Base) + Membres Propres

- Problème d'accès aux membres hérités: dans quelle zone sont-ils placés? ça dépend!
- Deux types de dérivation en C++: dérivation publique et dérivation privée (la dérivation protégée existe; elle offre peu d'intérêt)
- Différence de conception autant que de fonctionnalités: seule la dérivation publique autorise la compatibilité entre type de base et type dérivé (-> liaison dynamique)

Exemple

```
class Base {
private:
  int _bPriv;
protected:
  int _bProt;
public:
  int _bPubl;
  Base(int, int, int);
};
```

Dérivation Publique

- Correspond à la notion d'héritage telle qu'elle est implémentée dans les autres langages
- On spécifie la classe de base lors de la définition de la classe dérivée:

```
class PublicDerived: public Base {
protected:
  int _enPlus;
public:
  PublicDerived(int, int, int, int);
};
```

Structure (Accessiblité des membres)

mêmes parties `public` et `protected` que la classe `Base`, mais partie `private` *inaccessible*!

Dérivation Privée

```
class PrivateDerived: private Base {
protected:
  int _enPlus;
public:
  PrivateDerived(int, int, int, int);
};
```

Structure (Accessiblité des membres)

Toute la partie Base passe dans la zone privée de `PrivateDerived`

1. Représente un très faible pourcentage des classes dérivées
2. Utile lorsqu'on souhaite masquer une partie de l'interface.

Exemple

la classe Pile, dérivée privée de Liste:

```
class Pile: private Liste {
public:
  void * top() const { return nieme(1); }
  void * pop() { return retirer(1); }
  void push (void* e) { ajouter(1, e); }
  int isEmpty() const { return! longueur(); }
};
```

Dérivation et Sous-Typage

- Seule la dérivation publique permet au compilateur de considérer une instance de classe dérivée comme (un cas particulier d') instance de la classe de base: on parle de sous-type
- On peut donc placer une instance d'un sous-type partout où le compilateur attend une instance du type de base, sans casting

```
Base * TasDeBases [10];
Base b(0,1,2);
PublicDerived pubD(1,2,3,4);
PrivateDerived privD(2,3,4,5);
...
TasDeBases[0] = &b;      // Ok, évidemment
TasDeBases[1] = &pubD;   // Ok, types compatibles
TasDeBases[2] = &privD; // compiler error
void f(Base& b) {
  ...
```

Dérivation et Construction I

- Les membres privés de la classe de base sont TOTALEMENT inaccessibles de manière directe depuis la classe dérivée
- Seul moyen d'initialisation: le constructeur de la classe de base. On l'appelle dans la zone d'initialisation de la classe dérivée

```
// PublicDerived.C
#include "PublicDerived.h"
PublicDerived:: PublicDerived(int a,int b,int c,in
{
  _enPlus = d;
}
```

Construction (fin)

On trouve donc, dans la zone d'initialisation d'un constructeur, trois sortes de choses:

1. l'initialisation des données membres constants
2. l'initialisation des données membres "Objets"
3. l'appel au(x) constructeur(s) de base

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Fonctions Virtuelles

- Dans un langage typé statiquement, la liaison entre le message envoyé (l'appel de fonction membre) et la méthode déclenchée (la fonction membre effectivement exécutée) est faite à la compilation
- Problème: à cause des classes dérivées et du sous-typage, le compilateur ne connaît pas toujours le type exact du receveur: celui-ci peut n'être connu qu'à l'exécution
- En C++ (et dans d'autres LO) il est possible de retarder cette liaison au moment de l'exécution du programme: il suffit de définir une fonction membre VIRTUELLE (mot réservé virtual)
- Quand une fonction membre est déclarée virtuelle dans une classe, elle sera implicitement virtuelle dans toutes ses classes dérivées

Exemple simple

```
class Base {
  void stat()        { cout << "Base\n"; }
  virtual void dyn() { cout << "Base\n"; }
};
class Derived: public Base {
  void stat() { cout << "Derived\n"; }
  void dyn()  { cout << "Derived\n"; }
};
Base* pBase = new Derived; // Ok, types compatible
pBase->stat();             // affiche "Base"
pBase->dyn();              // affiche "Derived"
```

Amis et Liaison Dynamique

- La liaison dynamique ne peut être demandée que sur les fonctions membres, pas sur les fonctions amies de la classe
- Dans le cas où l'opérateur ne peut être surchargé que comme ami (par exemple, <<), on peut recourir à une "indirection"

```
class Base {
...
friend ostream& operator<<(ostream&, const Base&);
virtual ostream& printOn(ostream&) const;
...
};
inline ostream& operator<<(ostream& os,const Base&
{
  return b.printOn(os);
}
```

Classes Abstraites I

- L'héritage et la liaison dynamique permettent de définir des classes dont le but est de factoriser des propriétés communes à plusieurs autres, et non d'être instanciées (par exemple, la classe FormeGraphique peut déclarer une méthode dessiner())
- Une telle classe sera dite abstraite, et on pourra interdire son instanciation en déclarant (au moins) une fonction membre virtuelle pure
- Une fonction virtuelle pure n'est pas définie, elle est placée à l'adresse 0
- Elle doit être définie dans toute classe dérivée qui se veut instanciable, sinon la classe dérivée est elle-même abstraite

Classes Abstraites II

```
class FormeGeometrique {
  virtual void dessine() = 0;
  ...
};
```

On peut admettre une deuxième sorte de classe abstraite: la description sous forme d'une classe d'une machine à mémoire (par exemple, un générateur de nombres semi-aléatoires), ce qui ne nécessite que des variables de classe

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Héritage Multiple

- C++ autorise depuis sa version 2.0 l'héritage multiple
- En théorie, la puissance d'expression n'est pas supérieure à celle d'un langage ne possédant que l'héritage simple
- L'héritage multiple conduit en général à des bibliothèques de classes comprenant de plus petits éléments, mais en plus grand nombre -> réutilisabilité accrue
- Compromis à évaluer entre plus de confort et de réutilisabilité, et problème de gestion des conflits
- En C++, l'héritage multiple introduit un surcoût au niveau de l'accès aux données membres héritées, mais surtout au niveau de l'utilisation des fonctions virtuelles

Déclaration de Plusieurs Sur-classes

Lors de la définition de la classe dérivée, il suffit de mentionner les sur-classes directes:

```
class Derived: public Base1, private Base2 {
   ...
};
```

Le constructeur de la classe dérivée doit faire appel aux constructeurs des classes de base (constructeurs appelés dans l'ordre de spécification; destructeurs dans l'ordre inverse)

```
Derived:: Derived(...) : Base2(...), Base1(...)
{
   ...
}
```

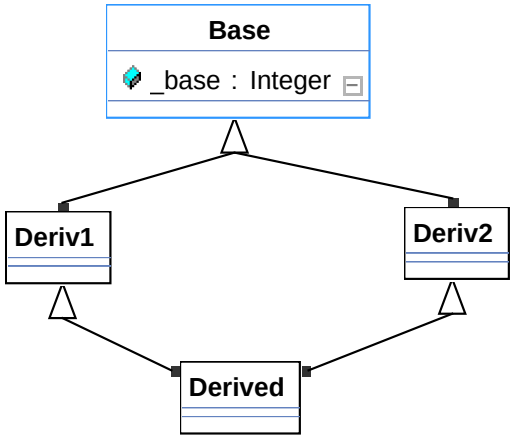
Conflits

Un conflit apparaît lorsque deux surclasses définissent un membre de même nom, qui n'est pas masqué dans la classe dérivée:

```
class Base1 {
  void f();
  ...
};
class Base2 {
  int f();
  ...
};
Derived d; // Compiler error
d.f();     // idem
```

Inclusion Multiple d'une Sur-classe

Second problème, aussi important que le premier: la gestion des inclusions multiples d'une classe de base: quelle est la structure d'une instance de `Derived`? ça dépend!



- Si `Deriv1` et `Deriv2` sont dérivée "ordinaires" de `Base`, il y deux occurences de `Base::_base` dans `Derived`
- Si elles sont dérivées "virtuelles", il n'y en aura qu'une

Problème

la structure de `Derived` dépend de la déclaration de `Deriv1` et `Deriv2`, qui a été faite alors que `Derived` n'existait pas!

Dérivation Ordinaire

- C'est le mode de dérivation par défaut
- Les parties de l'Objet représentant une instance d'une classe de base sont connexes à l'intérieur de l'Objet

Dérivation Virtuelle

- Si la duplication des membres de `Base` dans `Derived` n'est pas souhaitée, il faut déclarer `Deriv1` et `Deriv2` comme dérivées *virtuelles* de `Base`
- Ceci se fait en ajoutant le mot `virtual` lors de la déclaration

```
class Deriv1: virtual public Base {
  ...
};
// ordre des qualificatifs indifférents
class Deriv2: public virtual Base {
  ...
};
```

La structure de `Base` est alors partagée

Dérivation Virtuelle et Construction

- Les constructeurs des classes de base virtuelles sont appelés avant ceux des classes de base ordinaires
- Que se passe-t-il si la partie `Base` unique dans `Derived` est initialisée différemment par le constructeur de `Deriv1` et par celui de `Deriv2`?
- Solution retenue: on peut dans ce cas faire directement appel au constructeur de `Base` dans celui de `Derived`. Les modifications (concernant la partie `Base`) faites par les constructeurs de `Deriv1` et `Deriv2` seront ignorées

```
Derived:: Derived (int b, int d1, int d2, int e)
: Base(b), Deriv1(0,d1), Deriv2(-345, d2)
{
  _enPlus = e;
}
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Templates

- Les templates permettent de définir des types et des fonctions génériques, grâce aux types paramétrés
- Le but est de fournir au programmeur un outil permettant d'écrire du code plus général (et plus réutilisable) sans perte d'efficacité
- Cette notion est déjà présente dans de nombreux langages (ada, eiffel...) mais n'apparaît qu'en C++ version 3.0
- Deux sortes de templates: les templates de classes (classes génériques) et les templates de fonctions (fonctions génériques)
- On peut voir une template comme un moule permettant de créer plusieurs versions d'une même entité, ces versions ne différant entre elles que par la valeur d'un (de) paramètre(s) de la généricité

Classes Génériques

- Avec un langage typé dynamiquement, la généricité est "naturelle": la notion de "Pile" est représentable sans que l'on ait à spécifier la nature de ses éléments
- Dans un langage typé statiquement, il est nécessaire de définir une "Pile de QuelqueChose"
- Tout est à réécrire si l'on veut ensuite une "Pile de AutreChose"
- Une template de classe permet de définir une "Pile de *Elt*", où *Elt* est le paramètre de la template
- Lorsqu'on veut une Pile d'entiers, il suffit alors d'instancier la template avec `int` comme valeur du paramètre

Définition d'une Template de Classe

```
template<class Param> class Stack {
  Param * _body;
  Param * _top;
public:
  Stack(int=10);
  ~Stack();
  Param& top() const;
  Param& pop();
  void push(Param&);
  int isEmpty() const;
};
```

Définition d'une fonction membre

```
template<class Param> Param& Stack<Param>::top() c
  return *_top;
}
// Instanciation:
Stack<int> iStack(10); // Pile de 10 entiers
...
cout << iStack.top() << '\n';
```

Fonctions Génériques

De la même façon, la généricité peut être utilisée dans la définition de fonctions:

```
template<class T> int operator < (const T&, const
   ...
}
```

- Si une version non générique de l'opérateur existe (par exemple pour `int`), elle est utilisée; sinon, une version est automatiquement générée à partir de la template

Attention

Ceci suppose que le code de la template soit disponible AU MOMENT DE LA COMPILATION

Arguments de la Généricité

Les arguments de la généricité ne sont pas seulement des classes; ils peuvent être:

1. des chaînes de caractères
2. des noms de fonctions
3. des expressions constantes

```
template<class Param, int sz> class Stack {
Param _body[sz];
  ...
};
...
Stack<int,100> maPile;
```

- Des valeurs différentes des paramètres de la généricité créent des types incompatibles entre eux

Implémentation

- Lorsque le compilateur rencontre un appel à une fonction membre d'une template de classe, il est nécessaire qu'il dispose du corps de cette fonction
- L'implémentation des fonctions membres doit donc se trouver, soit dans le ".h" (comme les fonctions inline), soit dans un fichier inclus par ce dernier
- C'est le travail de l'éditeur de lien que d'éliminer les versions redondantes dans l'exécutable final

Exemple de programme I

La définition de la classe:

```
// Vector.h
template<class T> class Vector {
  int _size;
public:
  Vector();
  ~Vector();
  T& operator[] (int);
};
// eh oui! (ou corps des methodes ici)
#include "Vector.tmpl"
```

Le programme de test:

Exemple de programme II

```
// tVector.C
#include <iostream>
using namespace std;
#include "Vector.h"
typedef Vector<int> iVec;
extern void affiche(iVec&);
int main() {
  iVec vi;
  Vector<iVec> vvi;
  vi[3] = 3;
  vvi[3] = vi;
  cout << vi[3] << '\n';
  affiche(vvi[3]);
  return 0;
}
```

Reste à définir `affiche()`:

Exemple de programme III

```
// affVector.C
#include <iostream.h>
#include "Vector.h"
void affiche(Vector<int>& v) {
  cout << v[3] << '\n';
}
```

Construction et lancement du programme:

```
$ CC -o tVector tVector.C affVector.C
$ tVector
3
3
$
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Exceptions

A quoi sert une exception?

1. Le concepteur d'une classe sait *détecter* les erreurs à l'exécution mais ne peut les *traiter* sans restreindre la généralité de son code
2. L'utilisateur de la classe sait *ce qu'il veut faire* dans tel cas d'erreur à l'exécution, mais ne peut pas toujours les *détecter*

Avec un mécanisme de gestion d'exceptions, le concepteur peut prévenir qu'une "situation exceptionnelle" s'est produite (`throw`)
Libre alors à l'utilisateur d'ignorer ou de gérer (`catch`) comme il l'entend cette situation

Un Exemple Simple

Supposons que l'on veuille traiter les indices erronés dans une classe `Vector` (non générique!)

```
class Vector {
  int _size, * _body;
public:
  int& operator [] (int);
  ... // le reste
};
```

Traditionnellement, 4 techniques:

1. terminer le programme ("valeur par défaut"!)
2. retourner une valeur d'erreur (pas toujours faisable)
3. retourner une valeur légale (fonctionnement erratique de la suite du programme)
4. appeler une fonction d'erreur (qui, en général, fait 1, 2 ou 3!)

Les Classes Exceptions

Une exception sera souvent déclarée localement à une classe comme une "classe interne":

```
class Vector {
  int _size, * _body;
public:
  class Range {}; // classe exception
  int& operator [] (int)"throw(Range);
  ...
};
```

Le lancement d'une exception se fait par le mot réservé `throw`:

```
int& Vector:: operator [] (int i) throw(Range) {
  if (i>=0 && i<_size) return _body[i];
  throw Range();
}
```

Détection et Gestion d'une Exception

- La gestion des exceptions n'est possible que pour les instructions se trouvant dans un bloc `try`
- L'utilisateur mentionne explicitement l'(les) exception(s) qu'il souhaite gérer au moyen d'un (de) `catch`

```
void f (Vector& v) {
  ... // exceptions non gérées
  try { // armement gestion exc.
    ... // appel éventuel à []
  }
  catch (Vector::Range) {// gestion explicite
    ... // ce qu'on fait
  }
}
```

Remontée des exceptions

- Si l'exception n'est pas gérée à un niveau, elle est repassée au niveau appelant -> `main()`, puis la fonction terminate est appelée (`abort` par défaut). Cette fonction peut être définie par l'utilisateur (`set_terminate`, cf `set_new_handler`).
- Il peut y avoir un nombre quelconque de `catch` derrière un bloc `try`. *L'ordre importe*
- L'exception "..." est générique.

ATTENTION

l'envoi d'une exception dans un constructeur empêche l'appel implicite du destructeur: risque de mauvaise gestion de la mémoire

Paramètres des Exceptions

Comme une exception est souvent un Objet, elle peut contenir des informations sous forme de données membres:

```
class Vector {
  ...
  class Range {
  public:
    int index;
    Range(int i) { index = i; }
  };
};
...
void f(Vector& v) {
  ...
  try {
    ...
  }
  catch (Vector::Range r) { // exception nommée
```

Plan

1. Introduction Syntaxique C / C++
   - Types et variables
   - Fonctions et opérateurs
   - Structures de contrôle
   - Le pré-processeur
   - Pointeurs, tableaux et références
   - Retour sur les fonctions
   - Entrées-sorties en C++
   - C et C++
2. La classe en C++
   - Définition de classes C++
   - Membres d'une classe
   - Construction (initialisation)
   - Les friends
   - Construction (2) et destruction
   - Retour sur la Construction
   - Surcharge d'opérateurs
3. Héritage en C++

Identification de Type à l'Exécution

Son absence était l'un des reproches les plus fréquents au langage
Présent jusqu'alors sous N formes différentes (une par développeur): NORMALISATION
Essentiellement, deux outils:

1. un opérateur de conversion descendante "sûre" entre type de base et type dérivé;
2. une fonction donnant le type d'un objet passé en argument.

L'opérateur dynamic_cast

Permet de faire de la conversion descendante avec vérification de la validité (*type-safe downcast*):

```
#include <typeinfo.h>
class B { public: int b; };
class D: public B { public: int d; };
B * ptrBD, * ptrBB;
ptrBD = new D;       // OK
ptrBB = new B;
ptrBD->d = 3;        // erreur à la compilation
D * ptrDD = dynamic_cast<D*>(ptrBD); // ptrDD==ptr
ptrDD = ptrBD;       // erreur
D * ptrDB = dynamic_cast<D*>(ptrBB); // ptrDB==0
ptrDB = (D*) ptrBB; // OK mais pb!!
```

Conversions possibles

- Tout pointeur sur un Objet peut être converti ainsi. Un `void*` peut également être converti.
- Dans le cas des références, la valeur 0 n'a pas de signification particulière: l'exception `Bad_cast` est lancée
- Si le pointeur argument est nul, l'exception `Bad_typeid` est lancée

La fonction typeid

Permet de vérifier à l'exécution le type d'un Objet.
Utilisable avec une expression ou un type, prédéfini ou non (comme `sizeof`)
Le type `Type_info` - prédéfini - constitue le retour (en `const`) de la fonction. C'est une classe avec, pour principales méthodes:

- Les opérateurs == et `!=`
- La fonction membre `name()`, qui retourne une chaîne donnant le nom de la classe d'appartenance
- La fonction membre `before()`, qui teste l'ordre lexicographique des noms de classe (intérêt réduit!)

```
int controle (Person & p) {
  return typeid(p) == typeid(Invite);
}
```

Bibliographie I

- "Le langage C++" B. Stroustrup, 2nd Edition, InterEditions, 1993
  La bible, maintenant traduite en français. Complet, mais pas des plus pédagogiques ...
- "C++ primer" S.B. Lippman, 2nd edition, Addison-Wesley, 1993
  Très bon ouvrage de présentation du langage, à préférer - à mon sens - à celui de Stroustrup.
- "Programmation avancée en C++" J.O. Coplien, Addison-Wesley, 1992
  Uniquement après une solide première expérience de développement ... Bons conseils de design, excellent niveau technique.

Bibliographie II

- "Conception et programmation orientées objet en C++" K. Gorlen & al, Dunod, 1991
  Excellente présentation des techniques de base de conception de bibliothèques C++, par les concepteurs de NIH, importante bibliothèque du domaine public.
- "Design Patterns" E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1995.
  Un ouvrage qui ouvre les portes de la réutilisation au niveau de la conception, et pas seulement du codage. Les exemples -nombreux - sont en C++
- "Pattern Languages of Program Design" J. Coplein, D. Schmidt, Addison-Wesley, 1995
  Même créneau que le Gamma, mais plus axé sur C++, dont Coplien est un grand spécialiste

Bibliographie III

- "Le langage C" B. Kernighan, D.M. Ritchie, 2nd Edition, Masson, 1990
  La bible du C. Que dire de plus ?
- "C: Langage, bibliothèque, applications" H. Garreta, InterEditions, 1992
  Bon ouvrage d'introduction au langage, mettant l'accent sur des aspects laissés dans l'ombre par la bible (fonctionnement des entrées-sorties notamment)
- "Data structures: an advanced approach using C" J. Esakov & T. Weiss, Prentice-Hall, 1989
  Ouvrage de très bon niveau, montrant comment faire de l'encapsulation sans C++

Bibliographie IV

"Algorithmes en langage C" R. Sedgewick, InterEditions, 1991
Nombreuses études de cas programmées en C, couvrant un champ très étendu de domaines d'application.

"Les langages à objets" G. Masini et al, InterEditions, 1989
Vaste panorama très rigoureux des langages existants. A le mérite de commencer par une présentation des concepts et de la terminologie associée. Une référence dans ce domaine.

"Object Success" B. Meyer, Prentice Hall, 1995
Très court et très lisible. Bonne description de l'impact de la technologie Objet sur l'Entreprise.

"Eiffel: the Language" B. Meyer, Prentice Hall, 1992
Beucoup plus technique et moins "stratégique" que le précédent.

Bibliographie V

- "Succeeding with Objects" A. Goldberg, K. Rubin, Addison-Wesley, 1995
  Excellent guide de gestion de projet "objet"
- "Inside Smalltalk (tomes I et II)", LaLonde W., Pugh J., Prentice-Hall, 1991.
  Probablement les meilleurs sur Smalltalk, bien que datant un peu par rapport aux dernières releases des environnements.