Programmation Objet en C++

ISTIC

Mise à Niveau en Master 2

Thierry Duval (ISTIC / IRISA VR4I)

d'après Serge Morvan et Jacques Tisseau (ENIB / CERV)

Plan du cours

- Introduction 3
- Du C au C++ : le premier + ... 22
- Les classes en C++ 56
- Surdéfinition des opérateurs 158
- Espaces de nommage et STL 242
- L'héritage (et le polymorphisme, la liaison dynamique, . . .) 260
- Les exceptions 338
- Etude de cas : des piles 350
- Bibliographie 401
- Sommaire 404

Introduction

- **Introduction**
  - **1. Origines du C++ 4**
  - **2. Le langage C++ 5**
  - **3. Premier programme en C++ 6**
  - **4. Première classe en C++ : une** *Pile* **..... 7**
- Du C au C++ : le premier +
- Les classes en C++
- Surdéfinition des opérateurs
- Espaces de nommage et STL
- L'héritage
- Les exceptions
- Etude de cas : des piles
- Bibliographie
- Sommaire

Origines du C++

– Concepteur : Bjarne Stroustrup

  – http://www2.research.att.com/~bs/

– Date de naissance : Début des années 1980

– Lieu de naissance : Laboratoires Bell (ATT)

– But initial : Programmation par objets en C

Le langage C++

– Le premier +
  – Compatibilité avec le langage C
  – Typage fort
  – Références
  – Surdéfinition des fonctions
  – Généricité

– Le deuxième +
  – Classes et objets
  – Héritages simple et multiple
  – Surdéfinition des opérateurs
  – Polymorphisme et liaison dynamique

Mon premier programme C++

```
// inclusion des entrees/sorties

#include <iostream>

/* programme qui dit bonjour ... */

int main (void) {
  std::cout << "hello world" << std::endl ; // utilisation effective du flot
  return 0 ;
}
```


Un premier exemple complet : une classe *Pile*

– Définition de la classe : son interface : *pile0.h* 8
– Implémentation d'une classe : *pile0.C* 10
– Un petit programme principal : *main0.C* 14
– La phase de compilation et d'édition des liens 15
– L'exécution du programme principal *pile* 16
– Un second programme principal : *hanoi.C* 17
– Le *makefile* associé 19
– Compilation et édition des liens avec le *makefile* 20
– L'exécution du programme principal *hanoi* 21

pile0.h (1/2)

```
#ifndef PILE_H
#define PILE_H

class Pile {

  public :

    Pile (void) ;
    ~Pile (void) ;

    void empiler (double valeur) ;
    void depiler (void) ;
    double sommet (void) ;
    bool vide (void) ;
```

pile0.h (2/2)

```
    void afficher (void) ;

  protected :

    struct _Element {
      double valeur ;
      _Element * suivant ;
    } ;

    _Element * _tete ;

} ;

#endif
```


pile0.C (1/4)

```
#include "pile0.h"
#include <cassert>
#include <iostream>

Pile::Pile (void) {
  _tete = 0 ;
}

Pile::~Pile (void) {
  while (! vide()) {
    depiler () ;
  }
}
```


pile0.C (2/4)

```
void Pile::empiler (double valeur) {
  _Element * nouveau = new _Element ;
  nouveau->valeur = valeur ;
  nouveau->suivant = _tete ;
  _tete = nouveau ;
}

void Pile::depiler (void) {
  assert (! vide ()) ;
  _Element * ancien = _tete ;
  _tete = _tete->suivant ;
  delete ancien ;
}
```


pile0.C (3/4)

```
double Pile::sommet (void) {
  assert (! vide ()) ;
  return (_tete->valeur) ;
}

bool Pile::vide (void) {
  return (_tete == 0) ;
}
```


pile0.C (4/4)

```
void Pile::afficher (void) {
  _Element * courant = _tete ;
  std::cout << "Pile : " ;
  while (courant != 0) {
    std::cout << courant->valeur << " " ;
    courant = courant->suivant ;
  }
  std::cout << std::endl ;
}
```


main0.C

```
#include <iostream>
#include "pile0.h"
int main (void) {
  Pile p ;
  double x, y, z ;
  std::cout << "entrez x, y et z : " ;
  std::cin >> x >> y >> z ;
  p.empiler (x) ;
  p.empiler (y) ;
  p.empiler (z) ;
  p.afficher () ;
  while (! p.vide ()) {
    std::cout << p.sommet () << std::endl ;
    p.depiler () ;
  }
  return 0 ;
}
```


Compilation et édition des liens

```
<tduval@cigale> ls
hanoi.C   main0.C   makefile  pile0.C   pile0.h
<tduval@cigale> g++ -c pile0.C
<tduval@cigale> ls
hanoi.C   main0.C   makefile  pile0.C   pile0.h   pile0.o
<tduval@cigale> g++ -c main0.C
<tduval@cigale> ls
hanoi.C   main0.C   main0.o   makefile  pile0.C   pile0.h   pile0.o
<tduval@cigale> g++ -o pile main0.o pile0.o
<tduval@cigale> ls
hanoi.C   main0.C   main0.o   makefile  pile     pile0.C   pile0.h   pile0.o
<tduval@cigale>
```


Exécution de *pile*

```
<tduval@cigale> ./pile
entrez x, y et z : 1.2 3.4 5.6
Pile : 5.6     3.4     1.2
5.6
3.4
1.2
<tduval@cigale>
```

Les tours de Hanoï : *hanoi.C* (1/2)

```
#include <iostream>
#include "pile0.h"

void hanoi (int n, Pile & d, Pile & i, Pile & a) {
  if (n > 0) {
    hanoi (n - 1, d, a, i) ;
    double x ;
    x = d.sommet () ;
    d.depiler () ;
    a.empiler (x) ;
    std::cout << "deplacer " << n << " de " << &d << " vers " << &a << std::endl ;
    hanoi (n - 1, i, d, a) ;
  }
}
```


Les tours de Hanoï : *hanoi.C* (2/2)

```
int main (void) {
  Pile d, i, a ;
  int n ;
  std::cout << "nombre d'anneaux : " ;
  std::cin >> n ;
  for (int nb = n ; nb >= 1 ; nb --) {
    d.empiler (nb) ;
  }
  d.afficher () ;
  i.afficher () ;
  a.afficher () ;
  hanoi (n, d, i, a) ;
  d.afficher () ;
  i.afficher () ;
  a.afficher () ;
  return 0 ;
}
```


Le *makefile* associé

```
hanoi : hanoi.o pile0.o
	g++ -o hanoi hanoi.o pile0.o

hanoi.o : hanoi.C pile0.h
	g++ -c hanoi.C

pile0.o : pile0.h pile0.C
	g++ -c pile0.C

clean :
	rm -f hanoi *.o a.out core
```


Compilation et édition des liens avec le *makefile*

```
<tduval@cigale> clean
<tduval@cigale> ls
hanoi.C   main0.C   makefile  pile      pile0.C   pile0.h
<tduval@cigale> make
        g++ -c hanoi.C
        g++ -c pile0.C
        g++ -o hanoi hanoi.o pile0.o
<tduval@cigale> ls
hanoi     hanoi.o   makefile  pile0.C   pile0.o
hanoi.C   main0.C   pile      pile0.h
<tduval@cigale>
```


Exécution des tours de Hanoï

```
<tduval@cigale> ./hanoi
nombre d'anneaux : 3
Pile : 1     2     3
Pile :
Pile :
deplacer 1 de 0x7fff2efc vers 0x7fff2ef4
deplacer 2 de 0x7fff2efc vers 0x7fff2ef8
deplacer 1 de 0x7fff2ef4 vers 0x7fff2ef8
deplacer 3 de 0x7fff2efc vers 0x7fff2ef4
deplacer 1 de 0x7fff2ef8 vers 0x7fff2efc
deplacer 2 de 0x7fff2ef8 vers 0x7fff2ef4
deplacer 1 de 0x7fff2efc vers 0x7fff2ef4
Pile :
Pile :
Pile : 1     2     3
<tduval@cigale>
```


C++ : le premier +

- Introduction
- **Du C au C++ : le premier +**
  - **1. C et C++** **23**
  - **2. Les variables en C++** **29**
  - **3. Les fonctions en C++** **36**
- Les classes en C++
- Surdéfinition des opérateurs
- Espaces de nommage et STL
- L'héritage
- Les exceptions
- Etude de cas : des piles
- Bibliographie
- Sommaire

C et C++

- **Du C au C++ : le premier +**
  1. **C et C++**
     - **(a) Compatibilité C/C++** **24**
     - **(b) Un typage + fort** **25**
     - **(c) + de mots réservés** **26**
     - **(d) Des entrées/sortie + simples** **27**
  2. Les variables en C++
  3. Les fonctions en C++

Compatibilité C/C++

– Un compilateur C++ peut compiler du code C écrit selon la norme ANSI C

Un typage + fort

– Le langage C est un langage *faiblement typé*, très permissif, laissant beaucoup de liberté au programmeur

– Cette liberté est source de nombreux effets de bord souvent pernicieux

– En particulier, l'usage immodéré du préprocesseur peut avoir des conséquences catastrophiques sur la lisibilité et la fiabilité des programmes

– Le langage C++ introduit un *typage fort* beaucoup plus strict, et permet de limiter l'usage du préprocesseur à la compilation conditionnelle

+ de mots réservés

asm	*double*	**new**	*switch*
auto	*else*	**operator**	**template**
break	*enum*	**private**	**this**
case	*extern*	**protected**	**throw**
catch	*float*	**public**	**try**
char	*for*	*register*	*typedef*
class	**friend**	*return*	*union*
const	*goto*	*short*	*unsigned*
continue	*if*	*signed*	**virtual**
default	**inline**	*sizeof*	*void*
delete	*int*	*static*	*volatile*
do	*long*	*struct*	*while*

Remarque : les mots écrits en **gras** sont spécifiques au C++

Des entrées/sorties + simples qu'en C

```
#include <stdio.h>
int main (void) {
   int numero ;
   char nom [128] ;
   char action [128] ;
   double somme ;
   printf ("entrez le nom, le numero, l'action et la valeur : \n") ;
   scanf ("%s%d%s%lf", nom, &numero, action, &somme) ;
   printf ("%s %d %s %lf\n", nom, numero, action, somme) ;
   return 0 ;
}

<tduval@cigale> ./scanprint
entrez le nom, le numero, l'action et la valeur :
clown 12 gagne 23.78
clown 12 gagne 23.780000
<tduval@cigale>
```


Flots d'entrées/sorties du C++

```
#include <iostream>
int main (void) {
   int numero ;
   char nom [128] ;
   char action [128] ;
   double somme ;
   std::cout << "entrez le nom, le numero, l'action et la valeur : " << std::endl ;
   std::cin >> nom >> numero >> action >> somme ;
   std::cout << nom << " " << numero << " " << action << " " << somme << std::endl ;
   return 0 ;
}

<tduval@cigale> ./cincout
entrez le nom, le numero, l'action et la valeur :
clown 12 gagne 23.78
clown 12 gagne 23.78
<tduval@cigale>
```


Les variables en C++

■ **Du C au C++ : le premier +**

1. C et C++
2. **Les variables en C++**
   - **(a) Définition au + près** 30
   - **(b) Définition des constantes** 31
   - **(c) Les références en +** 32
   - **(d) Résolution de portée** 33
   - **(e) Allocation dynamique de mémoire + simple** 34
3. Les fonctions en C++

Définition au + près

– En C, définition au loin :

```
int i ;
...
for (i = 0 ; i < n ; i = i + 1) {
   ...
}
```

– En C++, définition au plus près :

```
for (int i = 0 ; i < n ; i = i + 1) {
   ...
}
```


Définition des constantes

– En C, portée : le fichier :

```
#define PI 3.14

static const int i = 12 ;
```

– En C++, on évite d'utiliser les macro–définitions :

```
const double PI = 3.14 ;

const int i = 12 ;
```


Les références en +

– On définit un type *référence* en faisant suivre le spécificateur de type par l'opérateur d'adressage : *type &*

– Une *référence*, de même qu'une constante, doit être initialisée

– Les *références* sont des *alias* : elles sont utilisées comme alternative au nom de l'objet avec lequel elles ont été initialisées

```
int i ;
int & ref = i ;
int * p = &i ;
(*p == ref) && (p == &ref) ;
```

– Les *références* sont essentiellement utilisées comme arguments ou types de fonctions

Résolution de portée

```
#include <iostream>

// uniquement en C++, operateur ::

int i = 0 ; // ::i

int main (void) {
  int i = 1 ;
  std::cout << "i = " << i << std::endl ; // 1
  std::cout << "::i = " << ::i << std::endl ; // 0
  {
    int i = 2 ;
    std::cout << "i = " << i << std::endl ; // 2
    std::cout << "::i = " << ::i << std::endl ; // 0
  }
  return 0 ;
}
```


Allocation dynamique de mémoire + simple qu'en C

```
#include <stdio.h>

int main (void) {
  int * i ;
  char * chaine ;
  i = (int *)malloc (sizeof (int)) ;
  chaine = (char *)malloc (256 * sizeof (char)) ;
  printf ("entrer un entier puis une chaine : ") ;
  scanf ("%d%s", i, chaine) ;
  printf ("i = %d ; chaine = %s\n", *i, chaine) ;
  free (i) ;
  free (chaine) ;
  return 0 ;
}
```


Allocation dynamique de mémoire en C++

```
#include <iostream>

int main (void) {
  int * i ;
  char * chaine ;
  i = new int ;
  chaine = new char [256] ;
  std::cout << "entrer un entier puis une chaine : " ;
  std::cin >> *i >> chaine ;
  std::cout << "i = " << *i << " ; chaine = " << chaine << std::endl ;
  delete i ;
  delete [] chaine ;
  return 0 ;
}
```


Les fonctions en C++

- **Du C au C++ : le premier +**
  1. C et C++
  2. Les variables en C++
  3. **Les fonctions en C++**
     - **(a) Prototypes de fonctions 37**
     - **(b) Fonctions en ligne 39**
     - **(c) Arguments par défaut 41**
     - **(d) Surdéfinition des fonctions 43**
     - **(e) Fonctions génériques 45**
     - **(f) Passage des arguments 49**
       - **i. Passage par valeurs 50**
       - **ii. Passage par pointeurs 52**
       - **iii. Passage par références 54**

Prototypes de fonctions en C

```
#include <stdio.h>

/* absence de declaration autorisee car f1 renvoie un entier ... */
/* int f1 () ; */
/* int f1 (char *, double) ; */
/* int f1 (char * s, double d) ; */

int main (void) {
  int un = f1 ("bonjour", 3.14159) ;
  return 0 ;
}

int f1 (char * chaine, double d) {
  printf ("chaine = %s ; d = %lf\n", chaine, d) ;
  return (1) ;
}
```


Prototypes de fonctions en C++

```
#include <iostream>

// int f1 (const char *, const double) ;
int f1 (const char * s, const double d) ;

int main (void) {
  int un = f1 ("bonjour", 3.14159) ;
  return 0 ;
}

int f1 (const char * chaine, const double d) {
  std::cout << "chaine = " << chaine << " ; d = " << d << std::endl ;
  return (1) ;
}
```


Fonctions en ligne en C : macro–définitions

```
#include <stdio.h>
#include <math.h>

/* attention, pas d'espace entre ABS et (x) !! */

#define ABS(x) ((x) > 0 ? (x) : - (x))

int main (void) {
  printf ("sqrt (abs (-3.14159)) = %lf\n", sqrt (ABS (-3.14159))) ;
  return 0 ;
}
```


Fonctions en ligne en C++

```
#include <iostream>
#include <cmath>

inline double ABS (const double x) {
  return ((x) > 0 ? (x) : - (x)) ;
}

int main (void) {
  std::cout << "sqrt (abs (-3.14159)) = " << sqrt (ABS (-3.14159)) << std::endl ;
  return 0 ;
}
```

Arguments par défaut en C++ (1/2)

– On fixe les valeurs par défaut dans le prototype de la fonction et non dans sa définition

– Les arguments concernés doivent être les derniers de la liste d'arguments

Arguments par défaut en C++ (2/2)

```
#include <iostream>

// uniquement en C++ et dans le prototype

void f (const char *, const int = 12, const float = 0.0) ;

int main (void) {
  f ("bonjour", 5, 3.14) ;
  f ("bonjour", 5) ;      // f ("bonjour",  5, 0.0)
  f ("bonjour") ;         // f ("bonjour", 12, 0.0)
  return 0 ;
}

void f (const char * chaine, const int i, const float f) {
  std::cout << "chaine = " << chaine << " ; i = " << i << " ; f = " << f << std::endl ;
}
```


Surdéfinition des fonctions en C++ (1/2)

```
#include <iostream>

int max (const int, const int) ;
int max (const int, const int, const int) ;
int max (const int [], const int) ;
// double max (const int, const int) ; // est une erreur

int main (void) {
  const int nbmax = 10 ;
  int tab [nbmax] ;
  for (int i = 0 ; i < nbmax ; i = i + 1) {
    tab [i] = nbmax - i ;
  }
  std::cout << max (5, 3) << " " << max (6, 5, 7)  << " " << max (tab, nbmax) << std::endl ;
  return 0 ;
}
```


Surdéfinition des fonctions en C++ (2/2)

```
int max (const int a, const int b) {
  int res = a ;
  if (b > res) { res = b ; }
  return (res) ;
}

int max (const int a, const int b, const int c) {
  return (max (a, max (b, c))) ;
}

int max (const int t [], const int n) {
  int res = t [0] ;
  for (int i = 1 ; i < n ; i = i + 1) {
    res = max (res, t [i]) ;
  }
  return (res) ;
}
```


Fonctions génériques en C++ : *ABS*

```
#include <iostream>
#include <cmath>

template <class Type>
inline Type ABS (const Type a) {
  Type res = a ;
  if (a < 0) {
    res = -a ;
  }
  return (res) ;
}

int main (void) {
  std::cout << "sqrt (abs (-3.14159)) = " << sqrt (ABS (-3.14159)) << std::endl ;
  return 0 ;
}
```


Fonctions génériques en C++ : *max* (1/3)

```
#include <iostream>

template <class Type>
Type max (const Type a, const Type b) {
  Type res = a ;
  if (b > res) {
    res = b ;
  }
  return (res) ;
}

template <class Type>
Type max (const Type a, const Type b, const Type c) {
  return (max (a, max (b, c))) ;
}
```


Fonctions génériques en C++ : *max* (2/3)

```
template <class Type>
Type max (const Type t [], const int n) {
  Type res = t [0] ;
  for (int i = 1 ; i < n ; i = i + 1) {
    res = max (res, t [i]) ;
  }
  return (res) ;
}
```


Fonctions génériques en C++ : *max* (3/3)

```
int main (void) {
  const int nbmax = 10 ;
  int tab [nbmax] ;
  for (int i = 0 ; i < nbmax ; i = i + 1) {
    tab [i] = nbmax - i ;
  }
  std::cout << max (5.3, 3.6) << " " << max (6, 5, 7)  << " " << max (tab, nbmax) << std::endl ;
  return 0 ;
}
```

Passage des arguments

Passage par valeur :

→ la fonction n'a jamais accès aux arguments effectifs de l'appel : les valeurs que la fonction manipule sont des copies locales

→ Le contenu des arguments effectifs de l'appel n'est pas modifié

Problèmes :

1. quand un objet volumineux est passé à l'appel
2. quand le contenu des arguments effectifs doit être modifié

Solutions :

→ en C : passer des pointeurs !

→ en C++ : passer des pointeurs ou des références ! . . . et des références de préférence !

Passage par valeurs (1/2)

```
#include <iostream>

void swap (int x, int y) {
  int tmp = x ;
  x = y ;
  y = tmp ;
}

int main (void) {
  int a = 10, b = 20 ;
  std::cout << "a = " << a << " ; b = " << b << std::endl ;
  swap (a, b) ;
  std::cout << "a = " << a << " ; b = " << b << std::endl ;
  return 0 ;
}
```


Passage par valeurs (2/2)

```
<tduval@cigale> g++ swapvaleur.C -o swapvaleur
<tduval@cigale> ./swapvaleur
a = 10 ; b = 20
a = 10 ; b = 20
<tduval@cigale>
```


Passage par pointeurs (1/2)

```
#include <iostream>

void swap (int * px, int * py) {
  int tmp = *px ;
  *px = *py ;
  *py = tmp ;
}

int main (void) {
  int a = 10, b = 20 ;
  std::cout << "a = " << a << " ; b = " << b << std::endl ;
  swap (&a, &b) ;
  std::cout << "a = " << a << " ; b = " << b << std::endl ;
  return 0 ;
}
```


Passage par pointeurs (2/2)

```
<tduval@cigale> g++ swappointeur.C -o swappointeur
<tduval@cigale> ./swappointeur
a = 10 ; b = 20
a = 20 ; b = 10
<tduval@cigale>
```


Passage par références (1/2)

```
#include <iostream>

void swap (int & x, int & y) {
  int tmp = x ;
  x = y ;
  y = tmp ;
}

int main (void) {
  int a = 10, b = 20 ;
  std::cout << "a = " << a << " ; b = " << b << std::endl ;
  swap (a, b) ;
  std::cout << "a = " << a << " ; b = " << b << std::endl ;
  return 0 ;
}
```


Passage par références (2/2)

```
<tduval@cigale> g++ swapreference.C -o swapreference
<tduval@cigale> ./swapreference
a = 10 ; b = 20
a = 20 ; b = 10
<tduval@cigale>
```


Les classes en C++

- Introduction
- Du C au C++ : le premier +
- **Les classes en C++**
  - **1. C++ : le deuxième +** **57**
  - **2. Les classes** **60**
  - **3. Les objets** **74**
  - **4. Les fonctions membres** **100**
  - **5. Les fonctions amies** **129**
  - **6. Les classes génériques** **141**
- Surdéfinition des opérateurs
- Espaces de nommage et STL
- L'héritage
- Les exceptions
- Etude de cas : des piles
- Bibliographie
- Sommaire

C++ : le deuxième +

■ **Les classes en C++**
1. **C++ : le deuxième +**
   (a) C++ et programmation par objets 58
   (b) Classification 59
2. Les classes
3. Les objets
4. Les fonctions membres
5. Les fonctions amies
6. Les classes génériques

C++ et programmation par objets

– Le langage C++ introduit les notions de *classe* et d'*instance* de la programmation par objets

– Un mécanisme de protection très élaboré des attributs et des méthodes permet une *encapsulation* efficace

– En C++, l'*héritage* est simple ou multiple

– Le typage est statique

– Les fonctions virtuelles (associées aux pointeurs) permettent le *polymorphisme* et la *liaison dynamique*

Classification

– La **classification** regroupe des objets ayant la même structure (*les attributs*) et le même comportement (*les méthodes*) au sein d'une même classe

– Une classe est une **abstraction** qui décrit un ensemble d'objets individuels

– Chaque objet est dit **instance** de sa classe

Les classes

■ **Les classes en C++**
1. C++ : le deuxième +
2. **Les classes**
   (a) Implémentation d'un TAD 61
   (b) Masquage de l'information 63
   (c) Classe ≡ composant logiciel 66
   (d) Compilation séparée en C++ 68
   (e) Classe ***Point*** **........ 69**
3. Les objets
4. Les fonctions membres
5. Les fonctions amies
6. Les classes génériques

Implémentation d'un TAD

– Une classe est une implémentation d'un type abstrait de données (TAD)

– Les classes possèdent :
  – des données membres : *les attributs*
  – des fonctions membres : *les méthodes*

– Le nom de la classe est le *spécificateur* de type

– Les objets de type X sont les *instances* de la classe X

Exemple sommaire de TAD : un *Entier*

```
#include <iostream>

class Entier {          // Entier : nom de la classe
  public :
    int i ;             // i : attribut
    int fcarre (void) { // fcarre : methode
      return (i * i) ;
    }
} ;

int main (void) {
  Entier e ;            // e : instance d'Entier
  e.i = 12 ;
  std::cout << e.fcarre () << std::endl ;
  return 0 ;
}
```

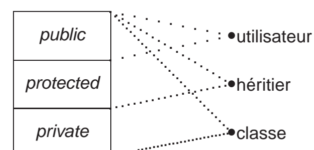

Masquage de l'information (1/3)

– Le masquage de l'information (ou **encapsulation**) consiste à séparer les aspects externes d'un objet, accessibles par les autres objets, des détails de son implémentation interne, rendus invisibles aux autres objets

– L'implémentation d'un objet peut ainsi être modifiée sans affecter les applications qui emploient cet objet

Masquage de l'information (2/3)

– Il existe 3 niveaux de masquage de l'information :
  – *private*
  – *protected*
  – *public*



Masquage de l'information (3/3)

– Par défaut, les attributs et les méthodes sont *private*

– La granularité de la protection est la classe et non l'objet :
  → une fonction membre pourra accéder aux données privées de n'importe quel autre objet de la même classe

Classe ≡ composant logiciel

– Une classe constitue une unité de compilation indépendante

classe ≡ type ≡ module

– Le fichier en–tête (*classe.h*) contient la déclaration de la classe : il joue le rôle d'interface de la classe (*spécification*)

– Le fichier source (*classe.C*) contient la définition des fonctions membres (*implémentation*)

– Une fonction définie à l'extérieur de la classe, doit être qualifiée par le nom de cette classe, à l'aide de l'opérateur de résolution de portée **: :**

type classe : :nom(signature)

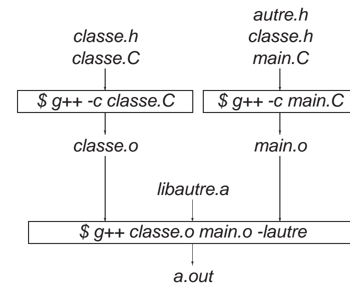
Remarques sur les classes

– La définition des fonctions peut en fait se trouver soit dans le fichier en–tête (*.h*), soit dans le fichier source (*.C* ou *.cpp* ou *.cxx* ou *.c++*)

– Les fonctions définies dans l'interface sont *inline* par défaut

– Les fonctions *inline* définies hors de l'interface doivent l'être dans le fichier en–tête

Compilation séparée en C++



Exemple : une classe *Point*

– Deux attributs de type entier : abscisse et ordonnée

– Premiers besoins :
  – initialiser un *Point* avec deux valeurs entières
  – déplacer un *Point* de *dx* et *dy*
  – afficher un *Point*, c'est-à-dire ses coordonnées

Interface de la classe *Point* v1 : *Point1.h*

```
#ifndef POINT_H
#define POINT_H

class Point {

  public :
    void initialise (short x, short y) ;
    void deplace (short dx, short dy) ;
    void affiche (void) ;

  protected :
    short x, y ;

} ;

#endif
```

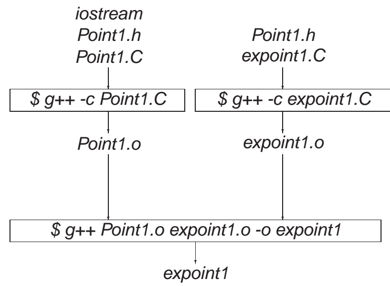

Implémentation de la classe *Point* v1 : *Point1.C*

```
#include "Point1.h"
#include <iostream>

void Point::initialise (short abs, short ord) {
  x = abs ;
  y = ord ;
}

void Point::deplace (short dx, short dy) {
  x = x + dx ;
  y = y + dy ;
}

void Point::affiche (void) {
  std::cout << "abscisse : " << x << " ; ordonnee : " << y << std::endl ;
}
```


Utilisation de la classe *Point* v1 : *expoint1.C*

```
#include "Point1.h"

int main (void) {
  Point a, b ;
  a.initialise (5, 2) ; a.affiche () ;
  a.deplace (-2, 4) ; a.affiche () ;
  b.initialise (2, 9) ; b.affiche () ;
  return 0 ;
}

<tduval@cigale> ./expoint1
abscisse : 5 ; ordonnee : 2
abscisse : 3 ; ordonnee : 6
abscisse : 2 ; ordonnee : 9
<tduval@cigale>
```

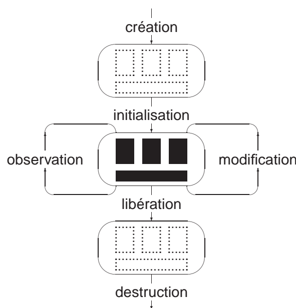
Classe *Point* v1 : compilation



Les objets

■ **Les classes en C++**
1. C++ : le deuxième +
2. Les classes
3. **Les objets**
   - (a) La vie d'un objet — Création et initialisation 75
   - (b) Affectation 77
   - (c) Opérations par défaut 78
   - (d) Dynamique des objets 91
   - (e) Auto–référence : le pointeur *this* 93
   - (f) Variables de classe 95
4. Les fonctions membres
5. Les fonctions amies
6. Les classes génériques

La vie d'un objet



Création et initialisation

– Création :

- – C++ fournit un constructeur par défaut
- – ce constructeur permet la création de nouveaux objets
- – on pourra définir ses propres constructeurs

– Initialisation :

- – l'initialisation doit être prévue par le concepteur de la classe

– Adressage :

- – C++ permet l'accès aux objets par l'intermédiaire de l'opérateur &
- – on pourra surdéfinir l'opérateur d'adressage &

Affectation

– Par défaut, l'affectation de deux objets de même type correspond à une simple recopie des valeurs des données membres à membres, publiques, protégées et privées

– Les objets comportant une partie d'allocation dynamique n'obtiendront qu'une simple recopie des pointeurs sur les zones dynamiques

– Pour une affectation complète, une surdéfinition de l'opérateur d'affectation (**=**) est nécessaire

– Pour une affectation partielle, une surdéfinition de l'opérateur d'affectation (**=**) est également nécessaire

Opérations par défaut

```
class C {          // déclaration de la classe C

} ;

int main (void) {
   C obj1, obj2 ;  // instanciations de obj1 et obj2
   C * p ;         // déclaration d'un pointeur sur une instance de C
   p = &obj1 ;     // adressage
   C & ref = obj2 ; // référence
   obj1 = obj2 ;   // affectation
   p = new C ;     // allocation dynamique
   delete p ;      // désallocation dynamique
   return 0 ;
}
```


Exemple d'affectation : *PointNomme.h* (1/2)

```
#ifndef POINTNOMME_H_
#define POINTNOMME_H_

#include <string>

class PointNomme {

  protected :

     int x, y ;
     std::string nom ;
```


Exemple d'affectation : *PointNomme.h* (2/2)

```
  public :

     PointNomme (std::string nom) ;
     void initialise (int, int) ;
     void deplace (int, int) ;
     void affiche (void) ;
     int getX (void) ;
     int getY (void) ;
     std::string getNom (void) ;

} ;

#endif
```

Exemple d'affectation : *PointNomme.c++* (1/3)

```
#include "PointNomme.h"

#include <iostream>

using namespace std ;

PointNomme::PointNomme (string nom = "") {
  this->nom = nom ;
}
```


Exemple d'affectation : *PointNomme.c++* (2/3)

```
void PointNomme::initialise (int abs, int ord) {
  x = abs ;
  y = ord ;
}

void PointNomme::deplace (int dx, int dy) {
  x = x + dx ;
  y = y + dy ;
}

void PointNomme::affiche (void) {
  cout << nom << " : x : " << x << " ; y : " << y << endl ;
}
```


Exemple d'affectation : *PointNomme.c++* (3/3)

```
int PointNomme::getX (void) {
  return x ;
}

int PointNomme::getY (void) {
  return y ;
}

string PointNomme::getNom (void) {
  return nom ;
}
```


Exemple d'affectation : *MainPointNomme.c++*

```
#include "PointNomme.h"
#include <iostream>

int main (void) {
  PointNommeInitial p1 ("P1") ;
  p1.initialise (1, 2) ;
  p1.affiche () ;
  PointNommeInitial p2 ("P2") ;
  p2.initialise (3, 4) ;
  p2.affiche () ;
  p2 = p1 ;
  p2.affiche () ;
}
```


Exemple d'affectation : exécution du test

```
P1 : x : 1 ; y : 2
P2 : x : 3 ; y : 4
P1 : x : 1 ; y : 2
```


Exemple d'initialisation de références : *Segment.h*

```
#ifndef SEGMENT_H_
#define SEGMENT_H_
#include "PointNomme.h"

class Segment {
  protected :
    PointNomme & pointDepart ;
    PointNomme & pointArrivee ;
  public :
    Segment (PointNomme & pointDepart, PointNomme & pointArrivee) ;
    void affiche (void) ;
    double getLongueur (void) ;
    virtual ~Segment () ;
} ;

#endif /* SEGMENT_H_ */
```


Exemple d'initialisation de références : *Segment.cpp* (1/2)

```
#include "Segment.h"
#include <iostream>
#include <cmath>

Segment::Segment (PointNomme & p1, PointNomme & p2)
         : pointDepart (p1), pointArrivee (p2) {
}

Segment::~Segment (void) {

}
```


Exemple d'initialisation de références : *Segment.cpp* (1/2)

```
void Segment::affiche (void) {
  std::cout << "Segment reliant " << pointDepart.getNom ()
            << " à " << pointArrivee.getNom ()
            << " de longueur " << getLongueur () << std::endl ;
}

double Segment::getLongueur (void) {
  int dx = pointArrivee.getX () - pointDepart.getX () ;
  int dy = pointArrivee.getY () - pointDepart.getY () ;
  return sqrt (dx * dx + dy * dy) ;
}
```

Exemple d'initialisation de références : *MainSegment.cpp*

```
#include "PointNomme.h"
#include "Segment.h"
#include <iostream>

int main (void) {
   PointNomme p1 ("P1") ;
   p1.initialise (1, 2) ;
   p1.affiche () ;
   PointNomme p2 ("P2") ;
   p2.initialise (3, 4) ;
   p2.affiche () ;
   Segment s (p1, p2) ;
   s.affiche () ;
   p2 = p1 ;
   p2.affiche () ;
   s.affiche () ;
}
```


Exemple d'initialisation de références : exécution

```
P1 : x : 1 ; y : 2
P2 : x : 3 ; y : 4
Segment reliant P1 à P2 de longueur 2.82843
P1 : x : 1 ; y : 2
Segment reliant P1 à P1 de longueur 0
```


Dynamique des objets

- **Objets statiques :**
  → variables globales, ou objets déclarés *static*, définis à la compilation
- **Objets dynamiques :**
  → définis par un pointeur sur une zone mémoire allouée par l'opérateur *new*
  → ces objets peuvent être détruits par l'opérateur *delete*
- **Objets automatiques :**
  → variables locales, leur durée de vie correspond à l'activation du bloc dans lequel elles sont déclarées
- **Objets temporaires :**
  → ils sont créés par le compilateur pour effectuer des calculs intermédiaires
  → pour contrôler la création de ces objets temporaires, il sera nécessaire de définir un constructeur par recopie

Exemples d'objets

```
#include "Point1.h"
Point a ;                                   // objet global donc statique

void action (void) {
   Point b ;                                // objet automatique
   Point * c ;                              // pointeur pour objet dynamique
   a.initialise (5, 2) ; a.affiche () ;
   b = a ; b.affiche () ;
   c = new Point ;                          // objet dynamique
   c->initialise (1, 5) ; c->affiche () ;
   delete c ;                               // récupération de l'objet dynamique
}

int main (void) {
   action () ;
   return 0 ;
}
```


Auto–référence : le pointeur *this*

- Le mot réservé *this* désigne un pointeur, implicitement déclaré, sur l'objet lui–même (auto–référence)
- Il peut être utilisé dans n'importe quelle fonction membre et constitue un alias de l'objet
- Implicitement, chaque fonction membre possède comme premier argument le pointeur *this*, adresse de l'objet . . .

Utilisation de *this* : *Point2.C*

```
#include "Point1.h"
#include <iostream>

void Point::initialise (short x, short y) {
   this->x = x ; // usage nécessaire car les attributs x et y
   this->y = y ; // de l'objet sont masqués par les arguments
}

void Point::deplace (short dx, short dy) {
   this->x = this->x + dx ; // usage bidon ...
   y = y + dy ;
}

void Point::affiche (void) {
   std::cout << "abscisse : " << this->x                // usage bidon ...
          << " ; ordonnee : " << this->y << std::endl ; // usage bidon ...
}
```


Variables de classe

- Les membres déclarés *static* sont des membres partagés par tous les objets de la classe et stockés en un seul endroit
- Par défaut, les variables *static* sont initialisées à 0, et ne peuvent pas être initialisées à la définition
- Une donnée membre *static* peut apparaître comme argument par défaut d'une fonction membre
- Une donnée membre *static* peut prendre pour valeur un objet de la classe

Variable de classe : interface : *Point3.h*

```
#ifndef POINT_H
#define POINT_H

class Point {

   public :
      void initialise (short x, short y) ;
      void deplace (short dx, short dy) ;
      void affiche (void) ;
      static int nbrPoints ;

   protected :
      short x, y ;

} ;

#endif
```

Variable de classe : implémentation : *Point3.C*

```
#include "Point3.h"
#include <iostream>

int Point::nbrPoints = 0 ;

void Point::initialise (short abs, short ord) {
   x = abs ; y = ord ;
   nbrPoints = nbrPoints + 1 ;
}

void Point::deplace (short dx, short dy) {
   x = x + dx ; y = y + dy ;
}

void Point::affiche (void) {
   std::cout << "abscisse : " << x << " ; ordonnee : " << y << std::endl ;
}
```


Variable de classe : utilisation : *expoint3.C*

```
#include "Point3.h"
#include <iostream>

int main (void) {
   Point a, b, * c ;
   a.initialise (5, 2) ; a.affiche () ;
   std::cout << "nombre de points initialisés : " << Point::nbrPoints << std::endl ;
   b = a ; b.affiche () ;
   std::cout << "nombre de points : " << Point::nbrPoints << std::endl ;
   c = new Point ; c->initialise (1, 5) ; c->affiche () ;
   std::cout << "nombre de points initialisés : " << Point::nbrPoints << std::endl ;
   return 0 ;
}
```


Variable de classe : résultat de l'utilisation

```
<tduval@cigale> ./expoint3
abscisse : 5 ; ordonnee : 2
nombre de points : 1
abscisse : 5 ; ordonnee : 2
nombre de points : 1
abscisse : 1 ; ordonnee : 5
nombre de points : 2
<tduval@cigale>
```


Les fonctions membres

■ **Les classes en C++**
1. C++ : le deuxième +
2. Les classes
3. Les objets
4. **Les fonctions membres**
   - **(a) Prototype d'une fonction membre** **101**
   - **(b) Fonctions membres** *const* **102**
   - **(c) Constructeurs** **104**
   - **(d) Destructeur** **108**
   - **(e) Fonctions membres** *static* **118**
   - **(f) Pointeurs sur fonctions membres** **126**
5. Les fonctions amies
6. Les classes génériques

Prototype d'une fonction membre

– Une fonction membre est caractérisée par :
  – son nom (identificateur)
  – sa signature (liste des arguments)
  – son type
  – sa classe

type classe : :nom (signature) ;

Fonctions membres *const*

– Une fonction membre peut être autorisée à consulter mais pas à modifier l'objet sur lequel elle est appelée

– La garantie que l'objet invoqué (**this*) ne sera pas modifié est donnée par le mot réservé *const* suffixé à la liste des arguments

type classe : :nom (signature) const ;

– Une fonction membre *const* peut être invoquée sur un objet *const*

– Une fonction membre ordinaire ne peut pas être invoquée sur un objet *const*

Fonctions membres *const* : exemple

```
class MaClasse {
   public :
      void modifie (void) {}
      void neModifiePas (void) const {}
} ;

int main (void) {
   const MaClasse objetNonModifiable ;
   // objetNonModifiable.modifie () ; // erreur de compilation
   objetNonModifiable.neModifiePas () ;
   MaClasse objetModifiable ;
   objetModifiable.modifie () ;
   objetModifiable.neModifiePas () ;
   return 0 ;
}
```


Les constructeurs : définition

– Un constructeur est une fonction membre dont le nom est le même que celui de la classe

– Un constructeur n'a pas de type

– Une classe comporte généralement plusieurs constructeurs (surdéfinition de fonctions) offrant à l'utilisateur un choix pour l'instanciation de nouveaux objets

– En fonction du nombre et du type des arguments passés à l'appel, le constructeur adapté sera choisi

– Si au moins un constructeur existe, l'utilisation du constructeur par défaut fourni par C++ n'est plus autorisée

Les constructeurs : exemples

```
class MaClasse {

  public :

    MaClasse (void) ;            // constructeur par defaut
    MaClasse (const int) ;       // constructeur avec initialisation
    MaClasse (const MaClasse &) ; // constructeur par recopie : clonage

  protected :

    int x ;

} ;
```


Les constructeurs : suite

– **Rôle** : *instanciation* des objets, c'est-à-dire initialisation et allocation éventuelle de mémoire.

– **Appel** : le constructeur est appelé *après* la création de l'objet

– **Initialisation des objets** : il est possible de transmettre une liste d'arguments au constructeur, cette liste apparait dans l'en–tête de la fonction constructeur, précédée de l'opérateur **:**

– **Constructeur par recopie** : afin de régler le problème de la recopie des zones dynamiques d'un objet, ce constructeur reçoit la référence d'un objet en argument (*clônage*)

Les constructeurs : exemple d'initialisation

```
class MaClasse {

  public :

    MaClasse (const int y) : x (y) {}

  protected :

    int x ;

} ;
```


Le destructeur

– **Définition** : un destructeur est une fonction membre dont le nom est le même que le nom de la classe précédé d'un tilda (~)

– Un destructeur n'a pas de type, ni d'arguments

– **Rôle** : *libération mémoire* pour des objets, automatiques, statiques, dynamiques ou temporaires

– **Appel** : le destructeur est appelé *avant* la destruction de l'objet

Le destructeur : exemple de déclaration

```
class MaClasse {

  public :

    MaClasse (void) ;            // constructeur par defaut
    MaClasse (const int) ;       // constructeur avec initialisation
    MaClasse (const MaClasse &) ; // constructeur par recopie : clonage

    ~MaClasse () ;               // destructeur

  protected :

    int x ;

} ;
```


Constructeurs et destructeurs : *Point4.h* (1/2)

```
#ifndef POINT_H
#define POINT_H

class Point {

  public :

    Point (void) ;
    Point (const short) ;
    Point (const short, const short) ;
    Point (const Point &) ;

    ~Point (void) ;
```


Constructeurs et destructeurs : *Point4.h* (2/2)

```
    void initialise (const short, const short) ;
    void deplace (const short, const short) ;
    void affiche (void) const ;

  protected :

    short x, y ;

} ;

#endif
```


Constructeurs et destructeurs : *Point4.C* (1/3)

```
#include "Point4.h"
#include <iostream>

Point::Point (void) {
  x = 0 ;
  y = 0 ;
}

Point::Point (const short x) {
  this->x = x ;
  y = x ;
  std::cout << "appel constructeur à un argument" << std::endl ;
}
```

Constructeurs et destructeurs : *Point4.C* (2/3)

```
Point::Point (const short x, const short y) {
   this->x = x ;
   this->y = y ;
   std::cout << "appel constructeur à deux arguments" << std::endl ;
}

Point::Point (const Point & p) {
   x = p.x ;
   y = p.y ;
   std::cout << "appel constructeur par recopie" << std::endl ;
}

Point::~Point (void) {
   std::cout << "appel destructeur" << std::endl ;
}
```


Constructeurs et destructeurs : *Point4.C* (3/3)

```
void Point::initialise (const short abs, const short ord) {
   x = abs ;
   y = ord ;
}

void Point::deplace (const short dx, const short dy) {
   x = x + dx ;
   y = y + dy ;
}

void Point::affiche (void) const {
   std::cout << "abscisse : " << x << " ; ordonnee : " << y << std::endl ;
}
```


Constructeurs et destructeurs : *expoint4.C*

```
#include "Point4.h"

Point a ;

Point creePoint (void) { Point p (7,8) ; return (p) ; }

int main (void) {
   Point b (5), c (5,2), e (b), f = c ;
   Point * d ;
   a.affiche () ; b.affiche () ; c.affiche () ;
   d = new Point (6) ; d->affiche () ; delete d ;
   e.affiche () ; f.affiche () ;
   f = creePoint () ; f.affiche () ;
   return 0 ;
}
```


Constructeurs et destructeurs : exécution (1/2)

```
<tduval@cigale> expoint4
appel constructeur à un argument
appel constructeur à deux arguments
appel constructeur par recopie
appel constructeur par recopie
abscisse : 0 ; ordonnee : 0
abscisse : 5 ; ordonnee : 5
abscisse : 5 ; ordonnee : 2
appel constructeur à un argument
abscisse : 6 ; ordonnee : 6
appel destructeur
abscisse : 5 ; ordonnee : 5
abscisse : 5 ; ordonnee : 2
```


Constructeurs et destructeurs : exécution (2/2)

```
appel constructeur à deux arguments
appel constructeur par recopie
appel destructeur
abscisse : 7 ; ordonnee : 8
appel destructeur
appel destructeur
appel destructeur
appel destructeur
appel destructeur
appel destructeur
<tduval@cigale>
```


Fonctions membres *static*

- Une fonction membre manipulant des données statiques, c'est-à-dire indépendantes de tout objet, peut être déclarée *static*
- L'appel d'une telle fonction ne nécessite que la référence à la classe
- Il est néanmoins autorisé de faire référence à un objet instance de la classe
- Le premier argument d'une fonction *static* n'est plus implicitement le pointeur *this*

Fonctions membres *static* : *Point5.h* (1/2)

```
#ifndef POINT_H
#define POINT_H

class Point {

   public :

      Point (void) ;
      Point (const short) ;
      Point (const short, const short) ;
      Point (const Point &) ;

      ~Point (void) ;
```


Fonctions membres *static* : *Point5.h* (2/2)

```
      void initialise (const short, const short) ;
      void deplace (const short, const short) ;
      void affiche (void) const ;

      static void afficheCompteur (void) ;

   protected :

      short x, y ;
      static int nbrPoints ;

} ;

#endif
```

Fonctions membres *static* : *Point5.C* (1/3)

```
#include "Point5.h"
#include <iostream>

int Point::nbrPoints = 0 ;

Point::Point (void) {
  x = 0 ; y = 0 ;
  nbrPoints = nbrPoints + 1 ;
  std::cout << "appel constructeur par defaut" << std::endl ;
}

Point::Point (const short x) {
  this->x = x ; y = x ;
  nbrPoints = nbrPoints + 1 ;
  std::cout << "appel constructeur à un argument" << std::endl ;
}
```


Fonctions membres *static* : *Point5.C* (2/3)

```
Point::Point (const short x, const short y) {
  this->x = x ; this->y = y ;
  nbrPoints = nbrPoints + 1 ;
  std::cout << "appel constructeur à deux arguments" << std::endl ;
}

Point::Point (const Point & p) {
  x = p.x ; y = p.y ;
  nbrPoints = nbrPoints + 1 ;
  std::cout << "appel constructeur par recopie" << std::endl ;
}

Point::~Point (void) {
  nbrPoints = nbrPoints - 1 ;
  std::cout << "appel destructeur" << std::endl ;
}
```


Fonctions membres *static* : *Point5.C* (3/3)

```
void Point::initialise (const short abs, const short ord) {
  x = abs ;
  y = ord ;
}

void Point::deplace (const short dx, const short dy) {
  x = x + dx ;
  y = y + dy ;
}

void Point::affiche (void) const {
  std::cout << "abscisse : " << x << " ; ordonnee : " << y << std::endl ;
}

void Point::afficheCompteur (void) {
  std::cout << "nombre de points : " << nbrPoints << std::endl ;
}
```


Fonctions membres *static* : *expoint5.C*

```
#include "Point5.h"

int main (void) {
  Point a (5, 2), b, * c ;
  Point::afficheCompteur () ; // appel avec référence à la classe
  Point d = a ;
  Point::afficheCompteur () ; // appel avec référence à la classe
  c = new Point (1, 5) ;
  a.afficheCompteur () ; // appel avec référence à un objet : toléré
  delete c ;
  Point::afficheCompteur () ; // appel avec référence à la classe
  return 0 ;
}
```


Fonctions membres *static* : exécution

```
<tduval@cigale> expoint5
appel constructeur à deux arguments
appel constructeur par defaut
nombre de points : 2
appel constructeur par recopie
nombre de points : 3
appel constructeur à deux arguments
nombre de points : 4
appel destructeur
nombre de points : 3
appel destructeur
appel destructeur
appel destructeur
<tduval@cigale>
```


Pointeur sur des fonctions membres

- L'identificateur d'une fonction est un pointeur . . .
- A l'appel, on qualifie la fonction par le spécificateur de sa classe

Pointeur sur des fonctions membres : *expoint5bis.C*

```
#include "Point5.h"

int main (void) {
  Point a, b, * c = new Point ;
  void (Point::* ptr1) (void) const ;
  void (Point::* ptr2) (const short, const short) ;
  ptr1 = &Point::affiche ;
  ptr2 = &Point::initialise ;
  (a.*ptr2) (4, 6) ; (a.*ptr1) () ;
  (c->*ptr2) (1, 1) ; (c->*ptr1) () ;
  ptr2 = &Point::deplace ;
  (a.*ptr2) (3, 4) ; (a.*ptr1) () ;
  (c->*ptr2) (3, 4) ; (c->*ptr1) () ;
  delete c ;
  return 0 ;
}
```


Pointeur sur des fonctions membres : exécution

```
<tduval@cigale> expoint5bis
appel constructeur par defaut
appel constructeur par defaut
appel constructeur par defaut
abscisse : 4 ; ordonnee : 6
abscisse : 1 ; ordonnee : 1
abscisse : 7 ; ordonnee : 10
abscisse : 4 ; ordonnee : 5
appel destructeur
appel destructeur
appel destructeur
<tduval@cigale>
```

Les fonctions amies

- **Les classes en C++**
  1. C++ : le deuxième +
  2. Les classes
  3. Les objets
  4. Les fonctions membres
  5. **Les fonctions amies**

(a) Les *friend* 130
(b) Les fonctions *friend* indépendantes 131
(c) Les fonctions *friend* membres d'une autre classe 133
(d) Les fonctions *friend* amies de plusieurs classes 136
(e) Les classes *friend* amies d'autres classes 139

  6. Les classes génériques

Les *friend*

– Normalement, seuls les membres d'une classe ont accès à ses parties *private*

– Le mécanisme des "amies" permet à une fonction non membre, voire même à une autre classe, de rompre le mécanisme d'encapsulation

– Ces fonctions (ou ces classes) devront être déclarées *friend* dans la classe

– Les *friend* peuvent être :
  – des fonctions indépendantes
  – des fonctions membres d'une autre classe
  – des fonctions amies de plusieurs classes
  – des classes amies d'autres classes

Fonctions *friend* indépendantes (1/2)

```
#include <iostream>

class MaClasse {

  public :

    friend void init (MaClasse &, int) ;
    friend void affiche (const MaClasse &) ;

  private :

    int i ;

} ;
```


Fonctions *friend* indépendantes (2/2)

```
void init (MaClasse & mc, int i) {
  mc.i = i ;
}

void affiche (const MaClasse & mc) {
  std::cout << mc.i << std::endl ;
}

int main (void) {
  MaClasse monInstance ;
  init (monInstance, 12) ;
  affiche (monInstance) ;
  return 0 ;
}
```


Fonctions *friend* membres d'une autre classe (1/3)

```
#include <iostream>

class MaClasse ;

class MonAutreClasse {

  public :

    void affiche (const MaClasse &) ;

} ;
```


Fonctions *friend* membres d'une autre classe (2/3)

```
class MaClasse {

  public :

    MaClasse (int i = 12) { this->i = i ; }
    friend void MonAutreClasse::affiche (const MaClasse &) ;

  private :

    int i ;

} ;
```


Fonctions *friend* membres d'une autre classe (3/3)

```
void MonAutreClasse::affiche (const MaClasse & mc) {
  std::cout << mc.i << std::endl ;
}

int main (void) {
  MaClasse monInstance (15) ;
  MonAutreClasse monAutreInstance ;
  monAutreInstance.affiche (monInstance) ;
  return 0 ;
}
```


Fonctions *friend* amies de plusieurs classes (1/3)

```
#include <iostream>

class MonAutreClasse ;

class MaClasse {

  public :

    MaClasse (int i = 12) { this->i = i ; }
    friend void affiche (const MaClasse &, const MonAutreClasse &) ;

  private :

    int i ;

} ;
```

Fonctions *friend* amies de plusieurs classes (2/3)

```
class MonAutreClasse {

  public :

    MonAutreClasse (float x = 3.14) { this->x = x ; }
    friend void affiche (const MaClasse &, const MonAutreClasse &) ;

  private :

    float x ;

} ;
```


Fonctions *friend* amies de plusieurs classes (3/3)

```
void affiche (const MaClasse & mc, const MonAutreClasse & mac) {
  std::cout << mc.i << " " << mac.x << std::endl ;
}

int main (void) {
  MaClasse monInstance (15) ;
  MonAutreClasse monAutreInstance ;
  affiche (monInstance, monAutreInstance) ;
  return 0 ;
}
```


Classes *friend* amies d'autres classes (1/2)

```
#include <iostream>

class MonAutreClasse ;

class MaClasse {

  public :

    MaClasse (int i = 12) { this->i = i ; }
    friend class MonAutreClasse ;

  private :

    int i ;

} ;
```


Classes *friend* amies d'autres classes (2/2)

```
class MonAutreClasse {

  public :

    void affiche (const MaClasse &mc) {
      std::cout << mc.i << std::endl ;
    }

} ;

int main (void) {
  MaClasse monInstance (15) ;
  MonAutreClasse monAutreInstance ;
  monAutreInstance.affiche (monInstance) ;
  return 0 ;
}
```


Les classes génériques

- **Les classes en C++**
  1. C++ : le deuxième +
  2. Les classes
  3. Les objets
  4. Les fonctions membres
  5. Les fonctions amies
  6. **Les classes génériques**
     - **(a) Les types génériques 142**
     - **(b) Exemple simple de classe générique 143**
     - **(c) Exemple complet de classe générique 146**

Les types génériques

- Classes destinées à manipuler des "objets" de type inconnu *a priori*
- En C++, la généricité n'est pas contrainte : on n'a aucune garantie sur le type générique (est-il comparable, additionnable, ... ?)
- Pour éviter les problèmes, les méthodes doivent être définies *inline* dans le fichier *.h*, avec la déclaration de la classe
- La détermination de type se fait à l'instanciation
- Des problèmes peuvent rester cachés lors de la compilation d'une classe générique (ce n'est pas une compilation "réelle" dans le cas du C++)
- Ces problèmes peuvent apparaître lors de la compilation du code qui réalise l'instanciation pour des types concrets

Exemple simple de classe générique (1/3)

```
template <class Type>

class MaClasse {

  public :

    MaClasse (const Type &) ;
    MaClasse (const MaClasse<Type> &) ;

    Type valeur (void) ;

  protected :

    Type v ;

} ;
```


Exemple simple de classe générique (2/3)

```
template <class Type>
MaClasse<Type>::MaClasse (const Type & v) {
  this->v = v ;
}

template <class Type>
MaClasse<Type>::MaClasse (const MaClasse<Type> & mc) {
  v = mc.v ;
}

template <class Type>
Type MaClasse<Type>::valeur (void) {
  return (v) ;
}
```

Exemple simple de classe générique (3/3)

```
#include <iostream>

int main (void) {
  MaClasse <int> mci (12) ;
  MaClasse <float> mcf (3.14) ;
  std::cout << mci.valeur () << " " << mcf.valeur () << std::endl ;
  return 0 ;
}
```


Exemple complet de classe générique : une classe *Pile*

- Modification de la classe *Pile* précédente
- Cette classe va maintenant pouvoir manipuler des objets de type inconnu *a priori*
- Les fonctions membres sont définies dans le fichier *.h* pour assurer la portabilité de la classe
- Elles sont toutes définies *inline* pour éviter les problèmes
- Le fichier *.C* associé ne sert qu'à permettre quelques vérifications lors de la compilation

pile1.h (1/7)

```
#ifndef PILE_H
#define PILE_H

template <class Type>

class Pile {

  public :

    Pile (void) ;
    Pile (const Pile <Type> &) ;
    ~Pile (void) ;

    void empiler (Type) ;
    void depiler (void) ;
    Type sommet (void) const ;
    bool vide (void) const ;
```


pile1.h (2/7)

```
    void afficher (void) const ;

  protected :

    struct _Element {
      Type valeur ;
      _Element * suivant ;
    } ;

    _Element * _tete ;

} ;
```


pile1.h (3/7)

```
#include <cassert>
#include <iostream>

template <class Type>
inline Pile<Type>::Pile (void) {
  _tete = 0 ;
}
```


pile1.h (4/7)

```
template <class Type>
inline Pile<Type>::Pile (const Pile <Type> & p) {
  _tete = 0 ;
  if (p._tete != 0) {
    _tete = new _Element ;
    _tete->valeur = p._tete->valeur ;
    _Element * parcoursCourant = _tete ;
    _Element * parcoursAutre = p._tete ;
    while (parcoursAutre->suivant != 0) {
      parcoursAutre = parcoursAutre->suivant ;
      parcoursCourant->suivant = new _Element ;
      parcoursCourant = parcoursCourant->suivant ;
      parcoursCourant->valeur = parcoursAutre->valeur ;
    }
    parcoursCourant->suivant = 0 ;
  }
}
```


pile1.h (5/7)

```
template <class Type>
inline Pile<Type>::~Pile (void) {
  while (! vide()) {
    depiler () ;
  }
}

template <class Type>
inline void Pile<Type>::empiler (Type valeur) {
  _Element * nouveau = new _Element ;
  nouveau->valeur = valeur ;
  nouveau->suivant = _tete ;
  _tete = nouveau ;
}
```


pile1.h (6/7)

```
template <class Type>
inline void Pile<Type>::depiler (void) {
  assert (! vide ()) ;
  _Element * ancien = _tete ;
  _tete = _tete->suivant ;
  delete ancien ;
}

template <class Type>
inline Type Pile<Type>::sommet (void) const {
  assert (! vide ()) ;
  return (_tete->valeur) ;
}
```

pile1.h (7/7)

```
template <class Type>
inline bool Pile<Type>::vide (void) const {
   return (_tete == 0) ;
}

template <class Type>
inline void Pile<Type>::afficher (void) const {
   _Element * courant = _tete ;
   std::cout << "Pile : " ;
   while (courant != 0) {
      std::cout << courant->valeur << " " ;
      courant = courant->suivant ;
   }
   std::cout << std::endl ;
}

#endif
```


pile1.C

```
#include "pile1.h"
```


main1.C

```
#include "pile1.h"

int main (void) {
   Pile<int> d ;
   Pile<float> f ;
   Pile<const char *> c ;
   d.empiler (1) ; d.empiler (2) ; d.empiler (3) ; d.afficher () ;
   f.empiler (1.1) ; f.empiler (2.2) ; f.empiler (3.3) ; f.afficher () ;
   c.empiler ("monde") ; c.empiler ("le") ; c.empiler ("bonjour") ;
   c.afficher () ;
   Pile<float> ff (f) ;
   ff.empiler (4.4) ; ff.afficher () ;
   f.afficher () ;
   return 0 ;
}
```


Le makefile

```
pile : main1.o pile1.o
	g++ -o pile main1.o

main1.o : main1.C pile1.h
	g++ -c main1.C

pile1.o : pile1.h pile1.C
	g++ -c pile1.C

clean :
	rm -f pile *.o a.out core
```


Exécution

```
<tduval@cigale> pile
Pile : 3      2      1
Pile : 3.3    2.2    1.1
Pile : bonjour  le    monde
Pile : 4.4    3.3    2.2    1.1
Pile : 3.3    2.2    1.1
<tduval@cigale>
```


Surdéfinition des opérateurs

- Introduction
- Du C au C++ : le premier +
- Les classes en C++
- **Surdéfinition des opérateurs**
  1. **Les opérateurs en C++ 159**
  2. **Modes de surdéfinition : comment surdéfinir ? 169**
  3. **Exemples de surdéfinitions 187**
- Espaces de nommage et STL
- L'héritage
- Les exceptions
- Etude de cas : des piles
- Bibliographie
- Sommaire

Les opérateurs en C++

- **Surdéfinition des opérateurs**
  1. **Les opérateurs en C++**
     - **(a) Opérateurs prédéfinis 160**
     - **(b) Opérateur ≡ Fonction 166**
     - **(c) Opérateurs surchargeables 167**
  2. Modes de surdéfinition
  3. Exemples de surdéfinitions

Opérateurs prédéfinis (1/5)

: :	résolution de portée	*classe : :membre*
: :	global	*: :nom*
.	sélection de membre	*objet.membre*
->	sélection de membre	*pointeur->membre*
[]	indexation	*pointeur [expr]*
()	appel de fonction	*expr (liste expr)*
()	constructeur	*type (liste expr)*
sizeof	taille d'un objet	*sizeof expr*
sizeof	taille d'un type	*sizeof (type)*

Opérateurs prédéfinis (2/5)

++	postincrémentation	*lvalue++*
++	préincrémentation	*++lvalue*
–	postdécrémentation	*lvalue–*
–	prédécrémentation	*–lvalue*
~	complément	*~expr*
!	négation	*!expr*
-	moins unaire	*-expr*
+	plus unaire	*+expr*
&	adresse de	*&lvalue*
*	déréférence	**expr*
new	allocation mémoire	*new type*
delete	désallocation mémoire	*delete pointeur*
delete []	désallocation tableau	*delete [] pointeur*
()	conversion de type	*(type)expr*

Opérateurs prédéfinis (3/5)

.*	sélection de membre	*objet.*pointeur*
->*	sélection de membre	*pointeur->*pointeur*
*	multiplication	*expr * expr*
/	division	*expr / expr*
%	modulo	*expr % expr*
+	addition	*expr + expr*
-	soustraction	*expr - expr*
«	décalage à gauche	*expr « expr*
»	décalage à droite	*expr » expr*

Opérateurs prédéfinis (4/5)

<	plus petit	*expr < expr*
<=	plus petit ou égal	*expr <= expr*
>	plus grand	*expr > expr*
>=	plus grand ou égal	*expr >= expr*
==	égalité	*expr == expr*
!=	non égalité	*expr != expr*
&	ET bit–à–bit	*expr & expr*
^	OU exclusif bit–à–bit	*expr ^ expr*
|	OU inclusif bit–à–bit	*expr | expr*
&&	ET logique	*expr && expr*
||	OU logique	*expr || expr*
? :	expression conditionnelle	*expr ? expr : expr*

Opérateurs prédéfinis (5/5)

=	affectation	*lvalue = expr*
*=	multiplication et affectation	*lvalue *= expr*
/=	division et affectation	*lvalue /= expr*
%=	modulo et affectation	*lvalue %= expr*
+=	addition et affectation	*lvalue += expr*
-=	soustraction et affectation	*lvalue -= expr*
«=	décalage et affectation	*lvalue «= expr*
»=	décalage et affectation	*lvalue »= expr*
&=	ET et affectation	*lvalue &= expr*
|=	OU inclusif et affectation	*lvalue |= expr*
^=	OU exclusif et affectation	*lvalue ^= expr*
,	séquence (virgule)	*expr , expr*

Associativité et priorité des opérateurs prédéfinis

– Les opérateurs unaires et les opérateurs d'affectation sont associatifs à droite
– tous les autres sont associatifs à gauche

**p++* ≡ **(p++)*

a = b = c ≡ *a = (b = c)*

a + b + c ≡ *(a + b) + c*

– Dans les tableaux des pages précédentes :
  – chaque encadré contient les opérateurs de même précédence
  – un opérateur a une précédence plus élevée que ceux des encadrés inférieurs

Opérateur ≡ Fonction

– Les opérateurs prédéfinis du langage C++ sont des fonctions que l'on peut donc surdéfinir pour les types définis par l'utilisateur

$$x = y + z$$
$$\equiv$$
operator=$(x,$ *operator+*$(y, z))$

– La priorité, l'associativité et l'arité des opérateurs prédéfinis ne peuvent pas être modifiées
– Il n'est pas possible de créer de nouveaux opérateurs
– Les opérateurs d'affectation (=) et d'adressage (&) sont implicitement surdéfinis par le compilateur
– Il est impossible de modifier la définition des opérateurs des types prédéfinis

Opérateurs surchargeables

+	-	*	/	%
^	&	|	~	!
=	<	>	+=	-=
*=	/=	%=	^=	&=
|=	«	»	»=	«=
==	!=	<=	>=	&&
||	++	–	,	->*
->	()	[]	*new*	*delete*

Opérateurs non surchargeables

: : *.** *.* *? :*

Modes de surdéfinitions

■ **Surdéfinition des opérateurs**

1. Les opérateurs en C++
2. **Modes de surdéfinition : comment surdéfinir ?**

(a) Membre ou non membre ? 170
(b) Opérateur membre 171
(c) Opérateur non membre 176
(d) Critères de choix 181
(e) Bonnes habitudes : cohérence des opérateurs 182

3. Exemples de surdéfinitions

Membre ou non membre ?

– Une fonction membre possède implicitement comme premier opérande le pointeur d'autoréférence *this*

type1 Classe : :membre(/*Classe* this,*/*type2,type3)*

→ si un opérateur requiert un membre gauche d'une autre classe, il ne peut être une fonction membre

– Un opérateur non membre ayant besoin d'accéder aux membres non publics d'une classe doit être une fonction *friend*

– Les opérateurs *(), =, ->* et *[]* doivent être des fonctions membres non statiques pour garantir que leur premier opérande soit une "lvalue"

– Les opérateurs commutatifs sont usuellement réalisés par des fonctions non membres

Opérateur membre : *Point6.h*

```
#ifndef POINT_H
#define POINT_H

class Point {

  public :
    Point (const short = 0, const short = 0) ;
    Point (const Point &) ;
    void affiche (void) const ;
    Point operator + (const Point &) const ;

  protected :
    short x, y ;

} ;

#endif
```


Opérateur membre : *Point6.C* (1/2)

```
#include "Point6.h"
#include <iostream>

Point::Point (const short x, const short y) {
  this->x = x ;
  this->y = y ;
}

Point::Point (const Point & p) {
  x = p.x ;
  y = p.y ;
}

void Point::affiche (void) const {
  std::cout << "abscisse : " << x << " ; ordonnee : " << y << std::endl ;
}
```


Opérateur membre : *Point6.C* (2/2)

```
Point Point::operator + (const Point & p) const {
  Point somme ;
  somme.x = x + p.x ;
  somme.y = y + p.y ;
  return somme ;
}
```


Opérateur membre : *expoint6.C*

```
<tduval@cigale> cat expoint6.C
#include "Point6.h"

int main (void) {
  Point a (1, 3), b (2, 5) ;
  Point c ;
  c = a + b + b ;
  c.affiche () ;
  c = c + 2 ;
  c.affiche () ;
  return 0 ;
}
<tduval@cigale> expoint6
abscisse : 5 ; ordonnee : 13
abscisse : 7 ; ordonnee : 13
<tduval@cigale>
```


Opérateur membre : remarques et commentaires

– La fonction membre reçoit implicitement un objet *Point* comme premier argument

– Pour le compilateur, on a :
p1 + p2 ≡ *p1.operator + (p2)*
p1 + p2 + p3 ≡ *(p1.operator+ (p2)).operator+ (p3)*
p1 + 2 ≡ *p1.operator + (2)*
2 + p1 ≡ *2.operator + (p1) // erreur*

Opérateur non membre : *Point7.h*

```
#ifndef POINT_H
#define POINT_H

class Point {

  public :
    Point (const short = 0, const short = 0) ;
    Point (const Point &) ;
    void affiche (void) const ;
    friend Point operator + (const Point &, const Point &) ;

  protected :
    short x, y ;

} ;

#endif
```

Opérateur non membre : *Point7.C* (1/2)

```
#include "Point7.h"
#include <iostream>

Point::Point (const short x, const short y) {
  this->x = x ;
  this->y = y ;
}

Point::Point (const Point & p) {
  x = p.x ;
  y = p.y ;
}

void Point::affiche (void) const {
  std::cout << "abscisse : " << x << " ; ordonnee : " << y << std::endl ;
}
```


Opérateur non membre : *Point7.C* (2/2)

```
Point operator + (const Point & p1, const Point & p2) {
  Point somme ;
  somme.x = p1.x + p2.x ;
  somme.y = p1.y + p2.y ;
  return somme ;
}
```


Opérateur non membre : *expoint7.C*

```
#include "Point7.h"

int main (void) {
  Point a (1, 3), b (2, 5) ;
  Point c ;
  c = a + b + b ; c.affiche () ;
  c = c + 2 ; c.affiche () ;
  c = 4 + c ; c.affiche () ;
  return 0 ;
}

<tduval@cigale> expoint7
abscisse : 5 ; ordonnee : 13
abscisse : 7 ; ordonnee : 13
abscisse : 11 ; ordonnee : 13
<tduval@cigale>
```


Opérateur non membre : remarques et commentaires

– La fonction **friend** reçoit explicitement tous les arguments

– Pour le compilateur, on a :
p1 + p2 $\equiv$ *operator+ (p1, p2)*
p1 + p2 + p3 $\equiv$ *operator+ (operator+ (p1, p2) ,p3)*
p1 + 2 $\equiv$ *operator + (p1, 2)*
2 + p1 $\equiv$ *operator + (2, p1)*

Critères de choix

– Le choix d'une fonction membre ou non membre affecte :
1. le code qui réalise l'opérateur
   (une fonction membre peut se référer à *this*, pas une fonction non membre)
2. le comportement de l'opérateur tel qu'il est perçu par l'utilisateur
   (si l'opérateur est une fonction membre, les conversions implicites ne pourront pas être appliquées au membre de gauche)

=, *()*, *[]*, ->	membres !
Opérateurs unaires	membres
Opérateurs d'affectation	membres
Opérateurs binaires	non membres

Bonnes habitudes

– Redéfinir l'opérateur d'affectation

– Redéfinir les opérateurs d'égalité et de différence

– Redéfinir l'opérateur d'injection dans un flot de sortie

Cohérence des opérateurs

– Respecter la sémantique habituelle des opérateurs
$+$: addition ...
$=$: affectation
$<<$: injection
...

– Respecter les relations entre opérateurs
$==$ et $!=$
$<$, $<=$, $>=$ et $>$
...

Cohérence des opérateurs $==$ et $!=$

```
bool Classe::operator == (const Classe &) const ; // a definir

bool Classe::operator != (const Classe & c) const {
  return (! operator == (c)) ; // ou (! ((*this) == c))
}
```

Cohérence des opérateurs $<$, $<=$, $>=$ et $>$

```
bool Classe::operator < (const Classe &) const ; // a definir

bool Classe::operator <= (const Classe & c) const {
  return (! (c < (*this))) ;
}

bool Classe::operator >= (const Classe & c) const {
  return (! ((*this) < c)) ; // ou (! operator < (c))
}

bool Classe::operator > (const Classe & c) const {
  return (c < (*this)) ;
}
```


Cohérence des opérateurs $++$ et $++$

```
// preincrementation
Classe & Classe::operator ++ (void) ; // a definir

// postincrementation
Classe & Classe::operator ++ (int) {
  Classe c (*this) ;
  ++ (*this) ;
  return (c) ;
}
```


Exemples de surdéfinitions

- **Surdéfinition des opérateurs**
  1. Les opérateurs en C++
  2. Modes de surdéfinition : comment surdéfinir ?
  3. **Exemples de surdéfinitions**
     - (a) **Une classe** *PointNomme* **188**
     - (b) **Une classe** *Pile* **202**
     - (c) **Une classe** *Pile* **générique** **221**

Exemple d'opérateurs : *PointNomme.h* (1/2)

```
#ifndef POINTNOMME_H_
#define POINTNOMME_H_

#include <string>
#include <iostream>

class PointNomme {
  public :
    PointNomme (const std::string & nom) ;
    PointNomme (const PointNomme & p) ;
    void initialise (const short x, const short y) ;
    void deplace (const short dx, const short dy) ;
    void affiche (std::ostream & out = std::cout) const ;
    int getX (void) const ;
    int getY (void) const ;
    std::string getNom (void) const ;
```


Exemple d'opérateurs : *PointNomme.h* (2/2)

```
    PointNomme & operator= (const PointNomme &) ;
    friend PointNomme operator+ (const PointNomme & p1, const PointNomme & p2) ;
    friend bool operator== (const PointNomme & p1, const PointNomme & p2) ;

  protected :

    int x, y ;
    std::string nom ;

} ;

std::ostream & operator<< (std::ostream & out, const PointNomme & p) ;

#endif
```


Exemple d'opérateurs : *PointNomme.c++* (1/4)

```
#include "PointNomme.h"
using namespace std ;

PointNomme::PointNomme (const string & nom) {
  this->nom = nom ;
}

PointNomme::PointNomme (const PointNomme & p) {
  x = p.x ;
  y = p.y ;
}

int PointNomme::getY (void) const { return y ; }

int PointNomme::getX (void) const { return x ; }
```


Exemple d'opérateurs : *PointNomme.c++* (2/4)

```
void PointNomme::initialise (const short x, const short y) {
  this->x = x ;
  this->y = y ;
}

void PointNomme::deplace (const short dx, const short dy) {
  x = x + dx ;
  y = y + dy ;
}

void PointNomme::affiche (ostream & out) const {
  out << nom << " : x : " << x << " ; y : " << y ;
}

std::string PointNomme::getNom (void) const { return nom ; }
```


Exemple d'opérateurs : *PointNomme.c++* (3/4)

```
PointNomme & PointNomme::operator= (const PointNomme & p) {
  if (this != &p) {
    x = p.x ;
    y = p.y ;
  }
  return (*this) ;
}

PointNomme operator+ (const PointNomme & p1, const PointNomme & p2) {
  PointNomme p ("") ;
  p.x = p1.x + p2.x ;
  p.y = p1.y + p2.y ;
  return p ;
}
```

Exemple d'opérateurs : *PointNomme.c++* (4/4)

```
ostream & operator<< (ostream & out, const PointNomme & p) {
  p.affiche (out) ;
  return out ;
}

bool operator== (const PointNomme & p1, const PointNomme & p2) {
  return ((p1.x == p2.x) && (p1.y == p2.y)) ;
}
```


Exemple d'opérateurs : *MainPointNomme.c++*

```
int main (void) {
  PointNommeEvolue p1 ("P1") ;
  p1.initialise (1, 2) ;
  cout << p1 << endl ;
  PointNommeEvolue p2 ("P2") ;
  p2.initialise (3, 4) ;
  cout << p2 << endl ;
  cout << "p1 == p2 ? " << (p1 == p2) << endl ;
  p2 = p1 ;
  cout << p2 << endl ;
  cout << "p1 == p2 ? " << (p1 == p2) << endl ;
  PointNommeEvolue p3 ("P3") ;
  p3 = p1 + p2 ;
  cout << p3 << endl ;
}
```


Exemple d'opérateurs : exécution du test

```
P1 : x : 1 ; y : 2
P2 : x : 3 ; y : 4
p1 == p2 ? 0
P2 : x : 1 ; y : 2
p1 == p2 ? 1
P3 : x : 2 ; y : 4
```


Exemple d'initialisation de références : *Segment.h* (1/2)

```
#ifndef SEGMENT_H_
#define SEGMENT_H_

#include <iostream>
#include "PointNomme.h"

class Segment {

  protected :

    const PointNomme & pointDepart ;
    const PointNomme & pointArrivee ;
```


Exemple d'initialisation de références : *Segment.h* (2/2)

```
  public :

    Segment (const PointNomme & pointDepart, const PointNomme & pointArrivee) ;
    void affiche (std::ostream & out = std::cout) const ;
    double getLongueur (void) const ;
    virtual ~Segment () ;

} ;

std::ostream & operator<< (std::ostream & out, const Segment & s) ;

#endif /* SEGMENT_H_ */
```


Exemple d'initialisation de références : *Segment.cpp* (1/2)

```
#include "Segment.h"
#include <iostream>
#include <cmath>

using namespace std ;

Segment::Segment (const PointNomme & p1, const PointNomme & p2)
          : pointDepart (p1), pointArrivee (p2) {

}

Segment::~Segment (void) {

}
```


Exemple d'initialisation de références : *Segment.cpp* (1/2)

```
void Segment::affiche (ostream & out) const {
  out << "Segment reliant " << pointDepart.getNom ()
      << " a " << pointArrivee.getNom () << " de longueur " << getLongueur () ;
}

double Segment::getLongueur (void) const {
  int dx = pointArrivee.getX () - pointDepart.getX () ;
  int dy = pointArrivee.getY () - pointDepart.getY () ;
  return sqrt (dx * dx + dy * dy) ;
}

std::ostream & operator<< (std::ostream & out, const Segment & s) {
  s.affiche (out) ;
  return out ;
}
```


Exemple d'initialisation de références : *MainSegment.cpp*

```
#include "PointNomme.h"
#include "Segment.h"
using namespace std ;

int main (void) {
  PointNomme p1 ("P1") ;
  p1.initialise (1, 2) ;
  cout << p1 << endl ;
  PointNomme p2 ("P2") ;
  p2.initialise (3, 4) ;
  cout << p2 << endl ;
  Segment s (p1, p2) ;
  cout << s << endl ;
  p2 = p1 ;
  cout << p2 << endl ;
  cout << s << endl ;
}
```

Exemple d'initialisation de références : exécution

```
P1 : x : 1 ; y : 2
P2 : x : 3 ; y : 4
Segment reliant P1 a P2 de longueur 2.82843
P2 : x : 1 ; y : 2
Segment reliant P1 a P2 de longueur 0
```


Surdéfinition des opérateurs pour une classe *Pile*

- Opérateurs membres :
  - affectation
  - injection d'un élément dans une pile
  - extraction d'un élément à partir d'une pile
- Opérateurs non membres :
  - égalité
  - différence
  - injection d'une pile dans un flot de sortie

pile1.5.h (1/3)

```
#ifndef PILE_H
#define PILE_H

#include <iostream>

class Pile {

  public :

    Pile (void) ;
    Pile (const Pile &) ;
    virtual ~Pile (void) ;
```


pile1.5.h (2/3)

```
    virtual void empiler (const double &) ;
    virtual void depiler (void) ;
    virtual double & sommet (void) const ;
    virtual bool vide (void) const ;

    virtual void afficher (std::ostream & = std::cout) const ;

    Pile & operator = (const Pile &) ;
    virtual Pile & operator << (const double &) ;
    virtual Pile & operator >> (double &) ;

    friend bool operator == (const Pile &, const Pile &) ;
    friend bool operator != (const Pile &, const Pile &) ;
    friend std::ostream & operator << (std::ostream &, const Pile &) ;
```


pile1.5.h (3/3)

```
  protected :

    struct _Element {
      double valeur ;
      _Element * suivant ;
    } ;

    _Element * _tete ;
    void _copier (const Pile &) ;
    void _vider (void) ;

} ;

#endif
```


pile1.5.C (1/7)

```
#include "pile1.5.h"
#include <cassert>

Pile::Pile (void) {
  _tete = 0 ;
}

Pile::Pile (const Pile & p) {
  _copier (p) ;
}

Pile::~Pile (void) {
  _vider () ;
}
```


pile1.5.C (2/7)

```
void Pile::empiler (const double & valeur) {
  _Element * nouveau = new _Element ;
  nouveau->valeur = valeur ;
  nouveau->suivant = _tete ;
  _tete = nouveau ;
}

void Pile::depiler (void) {
  assert (! vide ()) ;
  _Element * ancien = _tete ;
  _tete = _tete->suivant ;
  delete ancien ;
}
```


pile1.5.C (3/7)

```
double & Pile::sommet (void) const {
  assert (! vide ()) ;
  return (_tete->valeur) ;
}

bool Pile::vide (void) const {
  return (_tete == 0) ;
}

void Pile::_vider (void) {
  while (! vide()) {
    depiler () ;
  }
}
```

pile1.5.C (4/7)

```
Pile & Pile::operator = (const Pile & p) {
   if (this != &p) {
      _vider () ;
      _copier (p) ;
   }
   return (*this) ;
}

Pile & Pile::operator << (const double & ele) {
   empiler (ele) ;
   return (*this) ;
}

Pile & Pile::operator >> (double & ele) {
   ele = sommet () ; depiler () ;
   return (*this) ;
}
```


pile1.5.C (5/7)

```
void Pile::_copier (const Pile & p) {
   _tete = 0 ;
   if (p._tete != 0) {
      _tete = new _Element ;
      _tete->valeur = p._tete->valeur ;
      const _Element * ancien = p._tete ;
      _Element * nouveau = _tete ;
      while (ancien->suivant != 0) {
         ancien = ancien->suivant ;
         nouveau->suivant = new _Element ;
         nouveau = nouveau->suivant ;
         nouveau->valeur = ancien->valeur ;
      }
      nouveau->suivant = 0 ;
   }
}
```


pile1.5.C (6/7)

```
bool operator == (const Pile & p1, const Pile & p2) {
   const Pile::_Element * parcours1 = p1._tete ;
   const Pile::_Element * parcours2 = p2._tete ;
   bool identique = true ;
   while (identique && (parcours1 != 0) && (parcours2 != 0)) {
      identique = (parcours1->valeur == parcours2->valeur) ;
      parcours1 = parcours1->suivant ;
      parcours2 = parcours2->suivant ;
   }
   return (identique && (parcours1 == 0) && (parcours2 == 0)) ;
}

bool operator != (const Pile & p1, const Pile & p2) {
   return (! (p1 == p2)) ;
}
```


pile1.5.C (7/7)

```
void Pile::afficher (std::ostream & out) const {
   const _Element * courant = _tete ;
   while (courant != 0) {
      out << courant->valeur << " " ;
      courant = courant->suivant ;
   }
   out << std::endl ;
}

std::ostream & operator << (std::ostream & out, const Pile & p) {
   p.afficher (out) ;
   return (out) ;
}
```


main1.5.C (1/4)

```
#include "pile1.5.h"
#include <iostream>

void hanoi (int n, Pile & d, Pile & i, Pile & a) {
   if (n > 0) {
      hanoi (n - 1, d, a, i) ;
      double x ;
      d >> x ;
      std::cout << "deplacer : " << x << " de " << &d << " vers " << &a << std::endl ;
      a << x ;
      hanoi (n - 1, i, d, a) ;
   }
}
```


main1.5.C (2/4)

```
int main (void) {
   Pile p1 ;
   p1.empiler (0.1) ;
   p1.empiler (2.3) ;
   p1.empiler (4.5) ;
   Pile p2 = p1 ;
   Pile p3 (p1) ;
   Pile p4 ;
   std::cout << "p1 == p2 : " << (p1 == p2) << std::endl ;
   std::cout << "p1 != p2 : " << (p1 != p2) << std::endl ;
   std::cout << "p1 == p4 : " << (p1 == p4) << std::endl ;
   std::cout << "p1 != p4 : " << (p1 != p4) << std::endl ;
   p4 = p1 ;
   std::cout << p1<< p2 << p3 << p4 ;
```


main1.5.C (3/4)

```
   p1.depiler () ;
   std::cout << "p1 == p2 : " << (p1 == p2) << std::endl ;
   std::cout << "p1 != p2 : " << (p1 != p2) << std::endl ;
   p2.empiler (6.7) ;
   p3.depiler () ;
   p3.depiler () ;
   p3.empiler (8.9) ;
   std::cout << p1 << p2 << p3 << p4 ;
   std::cout << p3.sommet () << std::endl ;
   float x, y ;
   p2 >> x >> y ;
   std::cout << x << " " << y << std::endl ;
   std::cout << p2 ;
```


main1.5.C (4/4)

```
   Pile d, i, a ;
   int n ;
   std::cout << "nombre d'anneaux : " ;
   std::cin >> n ;
   for (int nb = n ; nb >= 1 ; nb --) {
      d << nb ;
   }
   std::cout << d << i << a ;
   hanoi (n, d, i, a) ;
   std::cout << d << i << a ;
   return 0 ;
}
```

makefile

```
pile : main1.5.o pile1.5.o
	g++ -o pile main1.5.o pile1.5.o

main1.5.o : main1.5.C pile1.5.h
	g++ -c main1.5.C

pile1.5.o : pile1.5.h pile1.5.C
	g++ -c pile1.5.C

clean :
	rm -f pile *.o a.out core
```


Exécution (1/3)

```
<tduval@cigale> pile
p1 == p2 : 1
p1 != p2 : 0
p1 == p4 : 0
p1 != p4 : 1
4.5   2.3   0.1
4.5   2.3   0.1
4.5   2.3   0.1
4.5   2.3   0.1
```


Exécution (2/3)

```
p1 == p2 : 0
p1 != p2 : 1
2.3   0.1
6.7   4.5   2.3   0.1
8.9   0.1
4.5   2.3   0.1
8.9
6.7   4.5
2.3   0.1
```


Exécution (3/3)

```
nombre d'anneaux : 3
1     2     3


deplacer : 1 de 0x7fff2ecc vers 0x7fff2ebc
deplacer : 2 de 0x7fff2ecc vers 0x7fff2ec4
deplacer : 1 de 0x7fff2ebc vers 0x7fff2ec4
deplacer : 3 de 0x7fff2ecc vers 0x7fff2ebc
deplacer : 1 de 0x7fff2ec4 vers 0x7fff2ecc
deplacer : 2 de 0x7fff2ec4 vers 0x7fff2ebc
deplacer : 1 de 0x7fff2ecc vers 0x7fff2ebc


1     2     3
<tduval@cigale>
```


Surdéfinition des opérateurs pour une classe *Pile* générique

– C'est presque la même chose que pour la classe non générique

– Le programme de test sera un peu plus complet . . .

– Opérateurs membres :
  – affectation
  – injection d'un élément dans une pile
  – extraction d'un élément à partir d'une pile

– Opérateurs non membres :
  – égalité
  – différence
  – injection d'une pile dans un flot de sortie

pile2.h (1/11)

```
#ifndef PILE_H
#define PILE_H

#include <iostream>

template <class Type>

class Pile {

  public :

    Pile (void) ;
    Pile (const Pile<Type> &) ;
    virtual ~Pile (void) ;
```


pile2.h (2/11)

```
    virtual void empiler (const Type &) ;
    virtual void depiler (void) ;
    virtual Type & sommet (void) const ;
    virtual bool vide (void) const ;

    virtual void afficher (std::ostream & = std::cout) const ;

    Pile<Type> & operator = (const Pile<Type> &) ;
    virtual Pile<Type> & operator << (const Type &) ;
    virtual Pile<Type> & operator >> (Type &) ;

    template <class T2> friend bool operator == (const Pile<T2> &, const Pile<T2> &) ;
    template <class T2> friend bool operator != (const Pile<T2> &, const Pile<T2> &) ;
    template <class T2>
              friend std::ostream & operator << (std::ostream &, const Pile<T2> &) ;
```


pile2.h (3/11)

```
  protected :

    struct _Element {
      Type valeur ;
      _Element * suivant ;
    } ;

    _Element * _tete ;
    void _copier (const Pile<Type> &) ;
    void _vider (void) ;

} ;
```

pile2.h (4/11)

```
#include <cassert>

template <class Type>
inline Pile<Type>::Pile (void) {
   _tete = 0 ;
}

template <class Type>
inline Pile<Type>::Pile (const Pile<Type> & p) {
   _copier (p) ;
}

template <class Type>
inline Pile<Type>::~Pile (void) {
   _vider () ;
}
```


pile2.h (5/11)

```
template <class Type>
inline void Pile<Type>::empiler (const Type & valeur) {
   _Element * nouveau = new _Element ;
   nouveau->valeur = valeur ;
   nouveau->suivant = _tete ;
   _tete = nouveau ;
}

template <class Type>
inline void Pile<Type>::depiler (void) {
   assert (! vide ()) ;
   _Element * ancien = _tete ;
   _tete = _tete->suivant ;
   delete ancien ;
}
```


pile2.h (6/11)

```
template <class Type>
inline Type & Pile<Type>::sommet (void) const {
   assert (! vide ()) ;
   return (_tete->valeur) ;
}

template <class Type>
inline bool Pile<Type>::vide (void) const {
   return (_tete == 0) ;
}

template <class Type>
inline void Pile<Type>::_vider (void) {
   while (! vide()) {
      depiler () ;
   }
}
```


pile2.h (7/11)

```
template <class Type>
inline Pile<Type> & Pile<Type>::operator = (const Pile<Type> & p) {
   if (this != &p) {
      _vider () ;
      _copier (p) ;
   }
   return (*this) ;
}
```


pile2.h (8/11)

```
template <class Type>
inline Pile<Type> & Pile<Type>::operator << (const Type & ele) {
   empiler (ele) ;
   return (*this) ;
}

template <class Type>
inline Pile<Type> & Pile<Type>::operator >> (Type & ele) {
   ele = sommet () ;
   depiler () ;
   return (*this) ;
}
```


pile2.h (9/11)

```
template <class Type>
inline bool operator == (const Pile<Type> & p1, const Pile<Type> & p2) {
   const Pile<Type>::_Element * parcours1 = p1._tete ;
   const Pile<Type>::_Element * parcours2 = p2._tete ;
   bool identique = true ;
   while (identique && (parcours1 != 0) && (parcours2 != 0)) {
      identique = (parcours1->valeur == parcours2->valeur) ;
      parcours1 = parcours1->suivant ;
      parcours2 = parcours2->suivant ;
   }
   return (identique && (parcours1 == 0) && (parcours2 == 0)) ;
}

template <class Type>
inline bool operator != (const Pile<Type> & p1, const Pile<Type> & p2) {
   return (! (p1 == p2)) ;
}
```


pile2.h (10/11)

```
template <class Type>
inline void Pile<Type>::_copier (const Pile<Type> & p) {
   _tete = 0 ;
   if (p._tete != 0) {
      _tete = new _Element ;
      _tete->valeur = p._tete->valeur ;
      const _Element * ancien = p._tete ;
      _Element * nouveau = _tete ;
      while (ancien->suivant != 0) {
         ancien = ancien->suivant ;
         nouveau->suivant = new _Element ;
         nouveau = nouveau->suivant ;
         nouveau->valeur = ancien->valeur ;
      }
      nouveau->suivant = 0 ;
   }
}
```


pile2.h (11/11)

```
template <class Type>
inline void Pile<Type>::afficher (std::ostream & out) const {
   const _Element * courant = _tete ;
   while (courant != 0) {
      out << courant->valeur << " " ;
      courant = courant->suivant ;
   }
   out << std::endl ;
}

template <class Type>
inline std::ostream & operator << (std::ostream & out, const Pile<Type> & p) {
   p.afficher (out) ;
   return (out) ;
}

#endif
```

pile2.C

```
#include "pile2.h"
```


main2.C (1/4)

```
#include "pile2.h"

void hanoi (int n, Pile <int> & d, Pile <int> & i, Pile <int> & a) {
  if (n > 0) {
    hanoi (n - 1, d, a, i) ;
    int x ;
    d >> x ;
    std::cout << "deplacer : " << x << " de " << &d << " vers " << &a << std::endl ;
    a << x ;
    hanoi (n - 1, i, d, a) ;
  }
}
```


main2.C (2/4)

```
int main (void) {
  Pile <double> p1 ;
  p1.empiler (0.1) ;
  p1.empiler (2.3) ;
  p1.empiler (4.5) ;
  Pile <double> p2 = p1 ;
  Pile <double> p3 (p1) ;
  Pile <double> p4 ;
  std::cout << "p1 == p2 : " << (p1 == p2) << std::endl ;
  std::cout << "p1 != p2 : " << (p1 != p2) << std::endl ;
  std::cout << "p1 == p4 : " << (p1 == p4) << std::endl ;
  std::cout << "p1 != p4 : " << (p1 != p4) << std::endl ;
  p4 = p1 ;
  std::cout << p1<< p2 << p3 << p4 ;
```


main2.C (3/4)

```
  p1.depiler () ;
  std::cout << "p1 == p2 : " << (p1 == p2) << std::endl ;
  std::cout << "p1 != p2 : " << (p1 != p2) << std::endl ;
  p2.empiler (6.7) ;
  p3.depiler () ;
  p3.depiler () ;
  p3.empiler (8.9) ;
  std::cout << p1 << p2 << p3 << p4 ;
  std::cout << p3.sommet () << std::endl ;
  Pile<int> p5 ;
  Pile<int> p6 (p5) ;
  p5 << 1 << 2 << 3 << 4 ;
  p6 << 5 ;
  std::cout << p5 << p6 ;
```


main2.C (4/4)

```
  int x, y ;
  p5 >> x >> y ;
  std::cout << x << " " << y << std::endl ;
  std::cout << p5 << p6 ;
  Pile<int> d, i, a ;
  int n ;
  std::cout << "nombre d'anneaux : " ;
  std::cin >> n ;
  for (int nb = n ; nb >= 1 ; nb --) {
    d << nb ;
  }
  std::cout << d << i << a ;
  hanoi (n, d, i, a) ;
  std::cout << d << i << a ;
  return 0 ;
}
```


makefile

```
pile : main2.o pile2.o
	g++ -o pile main2.o

main2.o : main2.C pile2.h
	g++ -c main2.C

pile2.o : pile2.h pile2.C
	g++ -c pile2.C

clean :
	rm -f pile *.o a.out core
```


Exécution (1/3)

```
<tduval@cigale> pile
p1 == p2 : 1
p1 != p2 : 0
p1 == p4 : 0
p1 != p4 : 1
4.5    2.3    0.1
4.5    2.3    0.1
4.5    2.3    0.1
4.5    2.3    0.1
p1 == p2 : 0
p1 != p2 : 1
```


Exécution (2/3)

```
2.3    0.1
6.7    4.5    2.3    0.1
8.9    0.1
4.5    2.3    0.1
8.9
4      3      2      1
5
4      3
2      1
5
```

Exécution (3/3)

```
nombre d'anneaux : 3
1    2    3


deplacer : 1 de 0x7fff2e94 vers 0x7fff2e84
deplacer : 2 de 0x7fff2e94 vers 0x7fff2e8c
deplacer : 1 de 0x7fff2e84 vers 0x7fff2e8c
deplacer : 3 de 0x7fff2e94 vers 0x7fff2e84
deplacer : 1 de 0x7fff2e8c vers 0x7fff2e94
deplacer : 2 de 0x7fff2e8c vers 0x7fff2e84
deplacer : 1 de 0x7fff2e94 vers 0x7fff2e84


1    2    3
<tduval@cigale>
```


Espaces de nommage et STL

- Introduction
- Du C au C++ : le premier +
- Les classes en C++
- Surdéfinition des opérateurs
- **Espaces de nommage et STL**
  1. **Espaces de nommage** **243**
  2. **STL** **249**
  3. **STL et Itérateurs** **250**
- L'héritage
- Les exceptions
- Etude de cas : des piles
- Bibliographie
- Sommaire

Espaces de nommage

– Pour permettre l'accès aux ressources d'un package
– Nécessite d'avoir inclus le package !
– Exemple : utilisation des flots du package std :

  – Inclusion du package :
    #include <iostream>

  – Utilisation classique :
    std::cout << "Geometrie3D" << std::endl ;

  – Déclaration d'utilisation :
    using namespace std ;

  – Utilisation après déclaration :
    cout << "Geometrie3D" << endl ;

Espaces de nommage : *Point3D.h* (1/2)

```
#ifndef POINT3D_H_
#define POINT3D_H_

#include <iostream>

namespace Geometrie3D {

  class Point3D {

    public :

      Point3D (const double x, const double y, const double z) ;
      virtual void affiche (std::ostream & out = std::cout) const ;
      virtual ~Point3D (void) ;
```


Espaces de nommage : *Point3D.h* (2/2)

```
    protected :
      double x ;
      double y ;
      double z ;

  } ;

} ;

std::ostream & operator<< (std::ostream & out, const Geometrie3D::Point3D & p) ;

#endif /* POINT3D_H_ */
```


Espaces de nommage : *Point3D.c++* (1/2)

```
#include "Point3D.h"

Geometrie3D::Point3D::Point3D (const double x, const double y, const double z) {
  this->x = x ;
  this->y = y ;
  this->z = z ;
}

std::ostream & operator<< (std::ostream & out, const Geometrie3D::Point3D & p) {
  p.affiche (out) ;
  return out ;
}
```


Espaces de nommage : *Point3D.c++* (2/2)

```
namespace Geometrie3D {

  void Point3D::affiche (std::ostream & out) const {
    std::cout << x << " " << y << " " << z ;
  }

  Point3D::~Point3D () {

  }

}
```


Espaces de nommage : *MainPoint3D.c++*

```
using namespace Geometrie3D ;

int main (void) {
  Point3D p (1, 2, 3) ;
  std::cout << p << std::endl ;
  return 0 ;
}
```

Standard Template Library

- Ensemble de classes et d'algorithmes qui implémentent les types de données standards :
  - vecteurs : *vector*
  - listes : *list*
  - ensembles : *set*
  - piles : *stack*
  - files : *queue*
  - files à priorités : *priority_queue*
  - ensembles de bits : *bitset*
  - tables : *map*
  - tables multiples : *multimap*

STL et Itérateurs

- Moyens de parcourir facilement les structures STL
  - dans l'ordre normal : *iterator*
  - dans l'ordre inverse : *reverse_iterator*
  - sans modifier els éléments : *const_iterator* et *const_reverse_iterator*
- Accès aux bornes des structures :
  - dans l'ordre normal : *begin* et *end*
  - dans l'ordre inverse : *rbegin* et *rend*
- Si *it* est un iterateur :
  - *it* ++ permet d'aller à l'élement suivant
  - *it* - - permet d'aller à l'élement précédent
  - **it* permet d'accéder à l'élement courant

Itérateurs STL : code C++ (1/5)

```
#include "PointNomme.h"
#include <string>
#include <list>
#include <map>

int main (void) {
  PointNomme p1 ("P1") ;
  p1.initialise (1, 2) ;
  PointNomme p2 ("P2") ;
  p2.initialise (3, 4) ;
  PointNomme p3 ("P3") ;
  p3 = p1 + p2 ;
```


Itérateurs STL : code C++ (2/5)

```
std::list<PointNomme> listPoints ;
listPoints.push_back (p1) ;
listPoints.push_back (p2) ;
listPoints.push_back (p3) ;
std::cout << "parcours de la \"list\" dans l'ordre normal des éléments : " << std::endl ;
for (std::list<PointNomme>::const_iterator it = listPoints.begin () ;
    it != listPoints.end () ; it ++) {
  std::cout << (*it) << std::endl;
}
std::cout << "parcours de la \"list\" dans l'ordre inverse des éléments : " << std::endl ;
for (std::list<PointNomme>::const_reverse_iterator it = listPoints.rbegin () ;
    it != listPoints.rend () ; it ++) {
  std::cout << (*it) << std::endl;
}
```


Itérateurs STL : exécution (1/4)

```
parcours de la "list" dans l'ordre normal des éléments :
 : x : 1 ; y : 2
 : x : 3 ; y : 4
 : x : 4 ; y : 6
parcours de la "list" dans l'ordre inverse des éléments :
 : x : 4 ; y : 6
 : x : 3 ; y : 4
 : x : 1 ; y : 2
```


Itérateurs STL : code C++ (3/5)

```
std::list<PointNomme *> listPointeursPoints ;
listPointeursPoints.push_back (&p1) ;
listPointeursPoints.push_back (&p1) ;
listPointeursPoints.push_back (&p2) ;
listPointeursPoints.push_back (&p1) ;
listPointeursPoints.push_back (&p3) ;
listPointeursPoints.push_back (&p1) ;
std::cout << "parcours de la \"list\" dans l'ordre normal des éléments : " << std::endl ;
for (std::list<PointNomme *>::const_iterator it = listPointeursPoints.begin () ;
    it != listPointeursPoints.end () ; it ++) {
  std::cout << *(*it) << std::endl;
}
```


Itérateurs STL : exécution (2/4)

```
parcours de la "list" dans l'ordre normal des éléments :
P1 : x : 1 ; y : 2
P1 : x : 1 ; y : 2
P2 : x : 3 ; y : 4
P1 : x : 1 ; y : 2
P3 : x : 4 ; y : 6
P1 : x : 1 ; y : 2
```


Itérateurs STL : code C++ (4/5)

```
std::cout << "parcours de la \"list\" avec suppression d'un élément : " << std::endl ;
std::list<PointNomme *>::iterator it = listPointeursPoints.begin () ;
while (it != listPointeursPoints.end ()) {
  std::cout << *(*it) << std::endl;
  if ((*it)->getNom () == "P1") {
    it = listPointeursPoints.erase (it) ;
  } else {
    it ++ ;
  }
}
std::cout << "parcours de la \"list\" pour vérification de la suppression : " << std::endl ;
for (std::list<PointNomme *>::const_iterator it = listPointeursPoints.begin () ;
    it != listPointeursPoints.end () ; it ++) {
  std::cout << *(*it) << std::endl;
}
```

Itérateurs STL : exécution (3/4)

```
parcours de la "list" avec suppression d'un élément :
P1 : x : 1 ; y : 2
P1 : x : 1 ; y : 2
P2 : x : 3 ; y : 4
P1 : x : 1 ; y : 2
P3 : x : 4 ; y : 6
P1 : x : 1 ; y : 2
parcours de la "list" pour vérification de la suppression :
P2 : x : 3 ; y : 4
P3 : x : 4 ; y : 6
```


Itérateurs STL : code C++ (5/5)

```
std::map<std::string, PointNomme &> mapPoints ;
mapPoints.insert (std::pair <std::string, PointNomme &>(p1.getNom (), p1)) ;
mapPoints.insert (std::pair <std::string, PointNomme &>(p2.getNom (), p2)) ;
mapPoints.insert (std::pair <std::string, PointNomme &>(p3.getNom (), p3)) ;
std::cout << "parcours de la \"map\" dans l'ordre normal des clés : " << std::endl ;
for (std::map<std::string, PointNomme &>::const_iterator
      it = mapPoints.begin () ; it != mapPoints.end () ; it ++) {
  std::cout << (*it).first << " => " << (*it).second << std::endl;
}
std::cout << "parcours de la \"map\" dans l'ordre inverse des clés : " << std::endl ;
for (std::map<std::string, PointNomme &>::const_reverse_iterator
      it = mapPoints.rbegin () ; it != mapPoints.rend () ; it ++) {
  std::cout << (*it).first << " => " << (*it).second << std::endl;
}

}
```


Itérateurs STL : exécution (4/4)

```
parcours de la "map" dans l'ordre normal des clés (ordre croissant) :
P1 => P1 : x : 1 ; y : 2
P2 => P2 : x : 3 ; y : 4
P3 => P3 : x : 4 ; y : 6
parcours de la "map" dans l'ordre inverse des clés :
P3 => P3 : x : 4 ; y : 6
P2 => P2 : x : 3 ; y : 4
P1 => P1 : x : 1 ; y : 2
```


L'héritage

- Introduction
- Du C au C++ : le premier +
- Les classes en C++
- Surdéfinition des opérateurs
- Espaces de nommage et STL
- **L'héritage**
  1. **Mécanismes de base** . . . 261
  2. **Masquage de l'information : accès aux membres** . . . 279
  3. **Modes de dérivation** . . . 286
  4. **Envois de messages** . . . 297
  5. **Classes abstraites** . . . 320
- Les exceptions
- Etude de cas : des piles
- Bibliographie
- Sommaire

Mécanismes de base

- **L'héritage**
  1. **Mécanismes de base**
     - **(a) Principes** . . . 262
     - **(b) Remarques** . . . 263
     - **(c) Spécialisation** . . . 264
     - **(d) Enrichissement** . . . 265
     - **(e) Héritage simple** . . . 266
     - **(f) Héritage multiple** . . . 272
  2. Masquage de l'information : accès aux membres
  3. Modes de dérivation
  4. Envois de messages
  5. Classes abstraites

Principes

– Une classe *B* peut hériter d'une classe *A*

→ *A* est la classe de base

→ *B* est la classe dérivée

```
class A {} ;

class B : public A {} ;    // B hérite de A
```

– Certains attributs et méthodes de la classe de base *A* pourront être utilisées dans la classe dérivée *B* sans réécriture de code

– L'interface (parties *public* et *protected*) de la classe *A* est ajoutée à l'interface de la classe *B*

Remarques

– Une classe *B* dérivée d'une classe *A* n'hérite pas :

→ des constructeurs

→ du destructeur

→ de l'affectation

– Pour une classe *B* dérivée d'une classe *A* :

→ lors de la construction d'une instance de *B*, le constructeur de la classe *B* est appelé **après** celui de la classe *A*

→ lors de la destruction d'une instance de *B*, le destructeur de la classe *B* est appelé **avant** celui de la classe *A*

Spécialisation

– Une redéfinition des méthodes de la classe de base est possible dans la classe dérivée

```
class A {
  public :
    int f (void) ;
    int g (void) ;
  protected :
    int _i ;
} ;

class B : public A {    // B hérite de A
  public :
    int f (void) ;
} ;
```

Enrichissement

– De nouveaux membres peuvent être définis dans la classe dérivée

```
class A {
  public :
    int f (void) ;
    int g (void) ;
  protected :
    int _i ;
} ;

class B : public A {    // B hérite de A
  public :
    int h (void) ;
  protected :
    int _j ;
} ;
```


Héritage simple : des points colorés

– Utilisation d'une classe *Point* existante

– Dérivation de cette classe pour y ajouter la notion de couleur

– Dans la nouvelle classe, il faut :
– ajouter un nouvel attribut
– définir les nouveaux constructeurs
– redéfinir une méthode

Point6.h (déjà vu . . .)

```
#ifndef POINT_H
#define POINT_H

class Point {

  public :
    Point (const short = 0, const short = 0) ;
    Point (const Point &) ;
    void affiche (void) const ;
    Point operator + (const Point &) const ;

  protected :
    short x, y ;

} ;

#endif
```


PointColore1.h

```
#ifndef POINTCOLORE_H
#define POINTCOLORE_H
#include "Point6.h"

class PointColore : public Point {

  public :
    PointColore (const short x = 0, const short y = 0, const short c = 0) ;
    PointColore (const PointColore &) ;
    void affiche (void) const ;

  protected :
    short couleur ;

} ;

#endif
```


PointColore1.C

```
#include "PointColore1.h"
#include <iostream>

PointColore::PointColore (const short x, const short y, const short c)
        : Point (x, y) {
  couleur = c ;
}

PointColore::PointColore (const PointColore & p) : Point (p) {
  couleur = p.couleur ;
}

void PointColore::affiche (void) const {
  std::cout << "couleur : " << couleur << " ; " ;
  Point::affiche () ;
}
```


expointcolore1.C

```
#include "PointColore1.h"

int main (void) {
  PointColore p1 (2, 4), p2 (1, 2, 9) ;
  Point p3 (p2), p4 (3, 5) ;
  p1.affiche () ;
  p2.affiche () ;
  p2.Point::affiche () ;
  p3.affiche () ;
  p4.affiche () ;
  p4 = p4 + p1 ;
  p4.affiche () ;
  return 0 ;
}
```


Exécution

```
<tduval@cigale> expointcolore1
couleur : 0 ; abscisse : 2 ; ordonnee : 4
couleur : 9 ; abscisse : 1 ; ordonnee : 2
abscisse : 1 ; ordonnee : 2
abscisse : 1 ; ordonnee : 2
abscisse : 3 ; ordonnee : 5
abscisse : 5 ; ordonnee : 9
<tduval@cigale>
```


Héritage multiple : encore des points colorés !

– Utilisation d'une classe *Point* existante

– Utilisation d'une classe *Couleur* existante

– Création d'une nouvelle classe par dérivation de ces deux classes

– Dans la nouvelle classe, il faut :
– définir les nouveaux constructeurs
– redéfinir une méthode

Couleur1.h

```
#ifndef COULEUR_H
#define COULEUR_H

class Couleur {

  public :
    Couleur (const short x = 0) ;
    Couleur (const Couleur &) ;
    void affiche (void) const ;

  protected :
    short couleur ;

} ;

#endif
```


Couleur1.C

```
#include "Couleur1.h"
#include <iostream>

Couleur::Couleur (const short c) {
  couleur = c ;
}

Couleur::Couleur (const Couleur & c) {
  couleur = c.couleur ;
}

void Couleur::affiche (void) const {
  std::cout << "couleur : " << couleur << " ; " ;
}
```


PointColore2.h

```
#ifndef POINTCOLORE_H
#define POINTCOLORE_H

#include "Point6.h"
#include "Couleur1.h"

class PointColore : public Point, public Couleur {

  public :

    PointColore (const short x = 0, const short y = 0, const short c = 0) ;
    PointColore (const PointColore &) ;
    void affiche (void) const ;

} ;

#endif
```


PointColore2.C

```
#include "PointColore2.h"

PointColore::PointColore (const short x, const short y, const short c)
        : Point (x, y), Couleur (c) {
}

PointColore::PointColore (const PointColore & p)
        : Point (p), Couleur (p) {
}

void PointColore::affiche (void) const {
  Couleur::affiche () ;
  Point::affiche () ;
}
```


expointcolore2.C

```
#include "PointColore2.h"

int main (void) {
  PointColore p1 (2, 4), p2 (1, 2, 9) ;
  Point p3 (p2), p4 (3, 5) ;
  p1.affiche () ;
  p2.affiche () ;
  p2.Couleur::affiche () ;
  p2.Point::affiche () ;
  p3.affiche () ;
  p4.affiche () ;
  p4 = p4 + p1 ;
  p4.affiche () ;
  return 0 ;
}
```


Exécution

```
<tduval@cigale> expointcolore2
couleur : 0 ; abscisse : 2 ; ordonnee : 4
couleur : 9 ; abscisse : 1 ; ordonnee : 2
couleur : 9 ; abscisse : 1 ; ordonnee : 2
abscisse : 1 ; ordonnee : 2
abscisse : 3 ; ordonnee : 5
abscisse : 5 ; ordonnee : 9
<tduval@cigale>
```


Masquage de l'information : accès aux membres

- **L'héritage**
  1. Mécanismes de base
  2. **Masquage de l'information : accès aux membres**
     - **(a) Autorisation d'accès aux attributs et méthodes 280**
     - **(b) Fonctions non membres de la classe de base 281**
     - **(c) Fonctions membres de la classe de base 283**
     - **(d) Fonctions d'une classe dérivée 284**
  3. Modes de dérivation
  4. Envois de messages
  5. Classes abstraites

Autorisation d'accès aux attributs et méthodes

– Les **fonctions non membres** d'une classe n'ont accès qu'aux membres (attributs et méthodes) *public* de cette classe

– Les **membres** d'une classe ont accès aux membres *public*, *protected* et *private* de la classe

– Un **membre d'une classe dérivée** a accès aux membres *public* et *protected* de la classe de base

Fonctions non membres de la classe de base X (1/2)

```
class X {
    private :
        int priv ;
    protected :
        int prot ;
    public :
        int publ ;
} ;
```


Fonctions non membres de la classe de base X (2/2)

```
class Y {
    public :
        void g (X & x) {
            // x.priv = 1 ; // erreur
            // x.prot = 2 ; // erreur
            x.publ = 3 ;
        }
} ;

void h (X & x) {
  // x.priv = 1 ; // erreur
  // x.prot = 2 ; // erreur
  x.publ = 3 ;
}
```


Fonctions membres de la classe de base X

```
class X {
    private :
        int priv ;
    protected :
        int prot ;
    public :
        int publ ;
        void f (void) ;
} ;

void X::f (void) {
    priv = 1 ;
    prot = 2 ;
    publ = 3 ;
}
```


Fonctions d'une classe dérivée de la classe X (1/2)

```
class X {
    private :
        int priv ;
    protected :
        int prot ;
    public :
        int publ ;
} ;
```


Fonctions d'une classe dérivée de la classe X (2/2)

```
class Y : public X {
    public :
        void f () ;
} ;

void Y::f () {
    // priv = 1 ; // erreur
    prot = 2 ;
    publ = 3 ;
}
```


Modes de dérivation

- **L'héritage**
  1. Mécanismes de base
  2. Masquage de l'information : accès aux membres
  3. **Modes de dérivation**
     - **(a) Accès aux classes de base 287**
     - **(b) Dérivation publique 288**
     - **(c) Dérivation protégée 291**
     - **(d) Dérivation privée 294**
  4. Envois de messages
  5. Classes abstraites

Accès aux classes de base

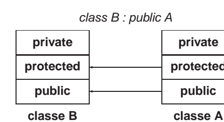
– Il existe trois façons d'hériter d'une classe :
  – héritage public (*public*)
  – héritage protégé (*protected*)
  – héritage privé (*private*)

– Ces modes d'héritage n'influent pas sur la perception de la classe ancêtre par la classe héritière :

→ dans les trois cas, les méthodes membres de la classe héritière n'auront accès qu'aux membres publics et protégés de la classe ancêtre

– Par contre ils vont influer sur la perception de cet héritage qu'auront :
  – des fonctions extérieures à ces classes
  – des classes dérivées de la classe héritière

Dérivation publique (1/3)



– L'héritage est public, c'est-à-dire officiel : tout le monde sait que la classe *B* hérite de la classe *A*

– Les fonctions extérieures à la classe *B* peuvent accéder à ses membres publics hérités de la classe *A*

– Les fonctions membres des classes dérivées de la classe *B* peuvent accéder à ses membres publics et protégés

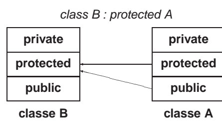
Dérivation publique (2/3)

```
class X {
    public :
        int a ;
    protected :
        int b ;
    private :
        int c ;
} ;

class Y1 : public X {
} ;

class Z1 : private Y1 {
    public :
        void f (void) ;
} ;
```


Dérivation publique (3/3)

```
void Z1::f (void) {
    Y1 y1 ;
    X x = y1 ;
    int a = this->a ;
    int b = this->b ;
    // int c = this->c ; // erreur
}

void g (void) {
    Y1 y1 ;
    X x = y1 ;
    int a = y1.a ;
    // int b = y1.b ; // erreur
    // int c = y1.c ; // erreur
}
```


Dérivation protégée (1/3)



– L'héritage est protégé, c'est-à-dire limité à la famille : seuls les héritiers de la classe *B* savent qu'elle hérite de la classe *A*

– Les fonctions extérieures à la classe *B* ne peuvent pas accéder à ses membres hérités de la classe *A*

– Les fonctions membres des classes dérivées de la classe *B* peuvent accéder à ses membres publics et protégés

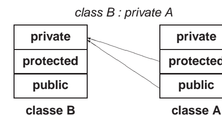
Dérivation protégée (2/3)

```
class X {
    public :
        int a ;
    protected :
        int b ;
    private :
        int c ;
} ;

class Y2 : protected X {
} ;

class Z2 : private Y2 {
    public :
        void f (void) ;
} ;
```


Dérivation protégée (3/3)

```
void Z2::f (void) {
    Y2 y2 ;
    X x = y2 ;
    int a = this->a ;
    int b = this->b ;
    // int c = this->c ; // erreur
}

void g (void) {
    Y2 y2 ;
    // int a = y2.a ; // erreur
    // int b = y2.b ; // erreur
    // int c = y2.c ; // erreur
    // X x = y2 ; // erreur
}
```


Dérivation privée (1/3)



– L'héritage est privé, c'est-à-dire limité à la seule classe dérivée : seule la classe *B* sait qu'elle hérite de la classe *A*

– Les fonctions extérieures à la classe *B* ne peuvent pas accéder à ses membres hérités de la classe *A*

– Les fonctions membres des classes dérivées de la classe *B* peuvent accéder à ses membres publics et protégés

Dérivation privée (2/3)

```
class X {
    public :
        int a ;
    protected :
        int b ;
    private :
        int c ;
} ;

class Y3 : private X {
} ;

class Z3 : private Y3 {
    public :
        void f (void) ;
} ;
```


Dérivation privée (3/3)

```
void Z3::f (void) {
    Y3 y3 ;
    // X x = y3 ; // erreur
    // int a = this->a ; // erreur
    // int b = this->b ; // erreur
    // int c = this->c ; // erreur
}

void g (void) {
    Y3 y3 ;
    // int a = y3.a ; // erreur
    // int b = y3.b ; // erreur
    // int c = y3.c ; // erreur
    // X x = y3 ; // erreur
}
```

Envois de messages

- **L'héritage**
  1. Mécanismes de base
  2. Masquage de l'information : accès aux membres
  3. Modes de dérivation
  4. **Envois de messages**

(a) Liaison statique **298**
(b) Liaison dynamique **304**
(c) Polymorphisme **318**

  5. Classes abstraites

Liaison statique

– L'héritage permet la conversion implicite d'un objet d'une classe dérivée à un objet d'une classe de base :
→ un *PointColore* **est une sorte de** *Point*
→ un *PointColore* **est une sorte de** *Couleur (!)*

– Un pointeur sur un objet d'une classe de base peut être redirigé sur une instance d'une classe dérivée

– Toutefois, le type de ce pointeur ayant été défini à la compilation, les méthodes choisies à l'exécution seront celles définies à la compilation

Encore des points colorés !

– Reprise des classes des points colorés précédents :
  – *Point6.h* et *Point6.C*
  – *Couleur1.h* et *Couleur1.C*
  – *PointColore2.h* et *PointColore2.C*

– Nouveau programme de manipulation mettant en évidence la liaison statique :
  – *expointcolore2.5.C*

expointcolore2.5.C

```
#include "PointColore2.h"

int main (void) {
  PointColore p1 (2, 4, 9) ;
  Point p2 ;
  Point * pp3 ;
  Couleur * c1 ;
  p1.affiche () ;
  p1.Couleur::affiche () ;
  p1.Point::affiche () ;
  p2 = p1 ;
  p2.affiche () ;
  pp3 = &p2 ; pp3->affiche () ;
  pp3 = &p1 ; pp3->affiche () ;
  c1 = &p1 ; c1->affiche () ;
  return 0 ;
}
```


Exécution

```
<tduval@cigale> expointcolore2.5
couleur : 9 ; abscisse : 2 ; ordonnee : 4
couleur : 9 ; abscisse : 2 ; ordonnee : 4
abscisse : 2 ; ordonnee : 4
abscisse : 2 ; ordonnee : 4
abscisse : 2 ; ordonnee : 4
couleur : 9 ; <tduval@cigale>
```


Liaison dynamique

– Les fonctions **virtual** permettent de retarder le choix de la méthode la plus adaptée

– Un pointeur sur un objet défini dans une classe de base peut, pendant le déroulement du programme, être redirigé sur n'importe quelle autre instance des classes dérivées

– Le choix de la méthode à appliquer sera fait à l'exécution si la méthode dans la classe de base est déclarée **virtual**

Liaison dynamique, constructeurs et destructeur

– La liaison dynamique ne fonctionne pas dans les constructeurs :
→ **c'est trop tôt !**

– Un constructeur ne peut pas être déclaré *virtual*

– La liaison dynamique ne fonctionne pas dans le destructeur :
→ **c'est trop tard !**

– Un destructeur peut (et devrait toujours) être déclaré *virtual*

Toujours des points colorés !

– Utilisation d'une classe *Point* existante

– Utilisation d'une classe *Couleur* existante

– Toutes les méthodes de ces deux classes sont *virtual* si possible

– Création d'une nouvelle classe par dérivation de ces deux classes ...

Point8.h

```
#ifndef POINT_H
#define POINT_H

class Point {

  public :
    Point (const short = 0, const short = 0) ;
    Point (const Point &) ;
    virtual ~Point (void) ;
    virtual void affiche (void) const ;
    virtual void dynamique (void) const ;

  protected :
    short x, y ;
} ;

#endif
```


Point8.C (1/2)

```
#include "Point8.h"
#include <iostream>

Point::Point (const short x, const short y) {
  this->x = x ;
  this->y = y ;
  dynamique () ;
  std::cout << "constructeur Point à 2 arguments" << std::endl ;
}

Point::Point (const Point & p) {
  x = p.x ;
  y = p.y ;
  dynamique () ;
  std::cout << "constructeur Point par recopie" << std::endl ;
}
```


Point8.C (2/2)

```
Point::~Point (void) {
  dynamique () ;
  std::cout << "destructeur Point" << std::endl ;
}

void Point::affiche (void) const {
  std::cout << "abscisse : " << x << " ; ordonnee : " << y << std::endl ;
}

void Point::dynamique (void) const {
  std::cout << "Point::dynamique " ;
}
```


Couleur2.h

```
#ifndef COULEUR_H
#define COULEUR_H

class Couleur {

  public :
    Couleur (const short x = 0) ;
    Couleur (const Couleur &) ;
    virtual ~Couleur (void) ;
    virtual void affiche (void) const ;
    virtual void dynamique (void) const ;

  protected :
    short couleur ;
} ;

#endif
```


Couleur2.C (1/2)

```
#include "Couleur2.h"
#include <iostream>

Couleur::Couleur (const short c) {
  couleur = c ;
  dynamique () ;
  std::cout << "constructeur Couleur a 1 argument" << std::endl ;
}

Couleur::Couleur (const Couleur & c) {
  couleur = c.couleur ;
  dynamique () ;
  std::cout << "constructeur Couleur par recopie" << std::endl ;
}
```


Couleur2.C (2/2)

```
Couleur::~Couleur (void) {
  dynamique () ;
  std::cout << "destructeur Couleur" << std::endl ;
}

void Couleur::affiche (void) const {
  std::cout << "couleur : " << couleur << " ; " ;
}

void Couleur::dynamique (void) const {
  std::cout << "Couleur::dynamique " ;
}
```


PointColore3.h

```
#ifndef POINTCOLORE_H
#define POINTCOLORE_H
#include "Point8.h"
#include "Couleur2.h"

class PointColore : public Point, public Couleur {

  public :
    PointColore (const short x = 0, const short y = 0, const short c = 0) ;
    PointColore (const PointColore &) ;
    virtual ~PointColore (void) ;
    virtual void affiche (void) const ;
    virtual void dynamique (void) const ;

} ;

#endif
```


PointColore3.C (1/2)

```
#include "PointColore3.h"
#include <iostream>

PointColore::PointColore (const short x, const short y, const short c)
        : Point (x, y), Couleur (c) {
  dynamique () ;
  std::cout << "constructeur PointColore a 3 arguments" << std::endl ;
}

PointColore::PointColore (const PointColore & p)
        : Point (p), Couleur (p) {
  dynamique () ;
  std::cout << "constructeur PointColore par recopie" << std::endl ;
}
```

PointColore3.C (2/2)

```
PointColore::~PointColore (void) {
  dynamique () ;
  std::cout << "destructeur PointColore" << std::endl ;
}

void PointColore::affiche (void) const {
  Couleur::affiche () ;
  Point::affiche () ;
}

void PointColore::dynamique (void) const {
  std::cout << "PointColore::dynamique " ;
}
```


expointcolore3.C

```
#include "PointColore3.h"
int main (void) {
  PointColore pc (2, 4, 9), * ppc ;
  Point p, * pp ;
  Couleur * c ;
  pc.affiche () ; pc.Couleur::affiche () ; pc.Point::affiche () ;
  pp = new PointColore (3, 4, 8) ; pp->affiche () ;
  delete pp ;
  c = new PointColore (1, 7, 5) ; c->affiche () ;
  delete c ;
  p = pc ; p.affiche () ;
  pp = &p ; pp->affiche () ;
  pp = &pc ; pp->affiche () ;
  ppc = (PointColore *)pp ; ppc->affiche () ;
  c = &pc ; c->affiche () ;
  return 0 ;
}
```


Exécution (1/2)

```
<tduval@cigale> expointcolore3
Point::dynamique constructeur Point a 2 arguments
Couleur::dynamique constructeur Couleur a 1 argument
PointColore::dynamique constructeur PointColore a 3 arguments
Point::dynamique constructeur Point a 2 arguments
couleur : 9 ; abscisse : 2 ; ordonnee : 4
couleur : 9 ; abscisse : 2 ; ordonnee : 4
Point::dynamique constructeur Point a 2 arguments
Couleur::dynamique constructeur Couleur a 1 argument
PointColore::dynamique constructeur PointColore a 3 arguments
couleur : 8 ; abscisse : 3 ; ordonnee : 4
PointColore::dynamique destructeur PointColore
Couleur::dynamique destructeur Couleur
Point::dynamique destructeur Point
```


Exécution (2/2)

```
Point::dynamique constructeur Point a 2 arguments
Couleur::dynamique constructeur Couleur a 1 argument
PointColore::dynamique constructeur PointColore a 3 arguments
couleur : 5 ; abscisse : 1 ; ordonnee : 7
PointColore::dynamique destructeur PointColore
Couleur::dynamique destructeur Couleur
Point::dynamique destructeur Point
abscisse : 2 ; ordonnee : 4
abscisse : 2 ; ordonnee : 4
couleur : 9 ; abscisse : 2 ; ordonnee : 4
couleur : 9 ; abscisse : 2 ; ordonnee : 4
couleur : 9 ; abscisse : 2 ; ordonnee : 4
Point::dynamique destructeur Point
PointColore::dynamique destructeur PointColore
Couleur::dynamique destructeur Couleur
Point::dynamique destructeur Point
<tduval@cigale>
```


Vérification de type après transtypage

```
#include "PointColore3.h"
int main (void) {
  PointColore * ppc ;
  Point * pp ;
  pp = new PointColore (3, 4, 8) ;
  ppc = dynamic_cast<PointColore *> (pp) ;
  if (ppc != 0) {
    ppc->affiche () ;
  }
  return 0 ;
}
```


Polymorphisme

– Le polymorphisme, c'est la possibilité de faire référence à une classe dérivée à partir d'une variable instance d'une classe de base
– On parle en général d'*affectation polymorphique*
– En C++, cela passe par l'usage de pointeurs . . .
– Inversement, lorsque l'on sait qu'une variable désigne une instance d'une classe dérivée, on peut l'affecter à une autre variable du type dérivé : c'est l'*affectation polymorphique inverse*
– Cette *affectation polymorphique inverse* peut être dangereuse !
– Quelque part, cela signifie que la puissance du concept de polymorphisme et de liaison dynamique est limitée . . .
– En C++ on parle de *transtypage* ou de *cast*

Polymorphisme en C++

```
PointColore pc (2, 4, 9), * ppc ;
Point * pp ;
pp = &pc ; // affectation polymorphique
pp = ppc ; // idem
ppc = (PointColore *)pp ; // cast, transtypage, affectation polymorphique inverse
ppc = dynamic_cast<PointColore *> (pp) ; // idem
```


Classes abstraites

■ **L'héritage**

1. Mécanismes de base
2. Masquage de l'information : accès aux membres
3. Modes de dérivation
4. Envois de messages
5. **Classes abstraites**
   (a) Méthodes virtuelles pures 321
   (b) Classes concrètes dérivées 322
   (c) Pourquoi des classes abstraites ? 323
   (d) Exemple de classe abstraite 324

Méthodes virtuelles pures

– Une classe possédant au moins une fonction membre **virtual** à valeur nulle (une telle fonction est dite *virtuelle pure*) est dite *abstraite* :
 → on ne peut pas déclarer d'instances de cette classe
 → on ne peut déclarer que des pointeurs sur des objets de cette classe
 → ces pointeurs pointeront en fait sur une instance d'une classe concrète dérivée de la classe abstraite

Classes concrètes dérivées

– Les concepteurs d'une classe dérivée d'une classe abstraite sont obligés :
 → soit de définir toutes ces fonctions dans la classe dérivée, cette classe dérivée est alors concrète : on peut en déclarer des instances
 → soit de laisser la classe dérivée abstraite elle aussi (elle hérite des méthodes virtuelles pures non redéfinies)

– La définition des méthodes virtuelles pures peut se faire à plusieurs niveaux de dérivation

Pourquoi des classes abstraites ?

– Pour factoriser des connaissances sur un type d'objet :
 → on peut expliciter tout ce qui est nécessaire pour manipuler correctement des objets d'un certain type (factorisation d'un TAD)
 → on ne sait pas forcement comment le faire (cela peut dépendre de choix d'implémentations à faire dans des classes dérivées)
 → on peut déjà savoir faire un certain nombre de choses : c'est autant de gagné pour les classes dérivées
 → on a la garantie que les classes concrètes dérivées auront bien défini les méthodes nécessaires

Exemple de classe abstraite

– Une classe abstraite *ObjetGeometrique*

– Et tout ce qui va autour :
 → une classe *Point*
 → une classe *Rectangle*
 → une classe *Cercle*
 → un programme de test

La classe *Point* (1/2)

```
#include <iostream>

const float pi = 3.14159 ;

class Point {

  public :
    Point (const float, const float) ;
    float x, y ;
    Point operator / (const float) ;
    friend Point operator + (const Point &, const Point &) ;

} ;
```


La classe *Point* (2/2)

```
Point::Point (const float x, const float y) {
  this->x = x ;
  this->y = y ;
}

Point Point::operator / (const float d) {
  Point resultat (x / d, y / d) ;
  return (resultat) ;
}

Point operator + (const Point & p1, const Point & p2) {
  Point somme (p1.x + p2.x, p1.y + p2.y) ;
  return (somme) ;
}
```


La classe *ObjetGeometrique* (1/3)

```
class ObjetGeometrique {

  public :
    ObjetGeometrique (const Point &) ;
    virtual ~ObjetGeometrique (void) ;
    virtual void deplacer (const Point &) ;
    virtual Point centre (void) const ;
    virtual float perimetre (void) const = 0 ;
    virtual float surface (void) const = 0 ;
    virtual void afficher (void) const ;

  protected :
    Point _centre ;
} ;
```


La classe *ObjetGeometrique* (2/3)

```
ObjetGeometrique::ObjetGeometrique (const Point & p)
                          : _centre (p) {
  std::cout << "constructeur ObjetGeometrique" << std::endl ;
}

ObjetGeometrique::~ObjetGeometrique (void) {
  std::cout << "destructeur ObjetGeometrique" << std::endl ;
}
```

La classe *ObjetGeometrique* (3/3)

```
void ObjetGeometrique::deplacer (const Point & p) {
  _centre = _centre + p ;
}

Point ObjetGeometrique::centre (void) const {
  return (_centre) ;
}

void ObjetGeometrique::afficher (void) const {
  std::cout << centre ().x << " " << centre ().y << " "
        << perimetre () << " " << surface () << " " ;
}
```


La classe *Rectangle* (1/3)

```
class Rectangle : public ObjetGeometrique {

  public :
    Rectangle (const Point & chg, const Point & cbd) ;
    virtual ~Rectangle (void) ;
    virtual float largeur (void) const ;
    virtual float hauteur (void) const ;
    virtual float perimetre (void) const ;
    virtual float surface (void) const ;
    virtual void afficher (void) const ;

  protected :
    float laHauteur, laLargeur ;

} ;
```


La classe *Rectangle* (2/3)

```
Rectangle::Rectangle (const Point & chg, const Point & cbd)
          : ObjetGeometrique ((chg + cbd) / 2) {
  laHauteur = cbd.y - chg.y ;
  laLargeur = cbd.x - chg.x ;
  std::cout << "constructeur Rectangle" << std::endl ;
}

Rectangle::~Rectangle (void) {
  std::cout << "destructeur Rectangle" << std::endl ;
}

void Rectangle::afficher (void) const {
  ObjetGeometrique::afficher () ;
  std::cout << largeur () << " " << hauteur () << std::endl ;
}
```


La classe *Rectangle* (3/3)

```
float Rectangle::largeur (void) const {
  return (laLargeur) ;
}

float Rectangle::hauteur (void) const {
  return (laHauteur) ;
}

float Rectangle::perimetre (void) const {
  return (2 * (laLargeur + laHauteur)) ;
}

float Rectangle::surface (void) const {
  return (laLargeur * laHauteur) ;
}
```


La classe *Cercle* (1/3)

```
class Cercle : public ObjetGeometrique {

  public :
    Cercle (const Point & c, const float d) ;
    virtual ~Cercle (void) ;
    virtual float rayon (void) const ;
    virtual float diametre (void) const ;
    virtual float perimetre (void) const ;
    virtual float surface (void) const ;
    virtual void afficher (void) const ;

  protected :
    float leRayon ;

} ;
```


La classe *Cercle* (2/3)

```
Cercle::Cercle (const Point & c, const float d)
       : ObjetGeometrique (c) {
  std::cout << "constructeur Cercle" << std::endl ;
  leRayon = d / 2 ;
}

Cercle::~Cercle (void) {
  std::cout << "destructeur Cercle" << std::endl ;
}

void Cercle::afficher (void) const {
  ObjetGeometrique::afficher () ;
  std::cout << rayon () << " " << diametre () << std::endl ;
}
```


La classe *Cercle* (3/3)

```
float Cercle::rayon (void) const {
  return (leRayon) ;
}

float Cercle::diametre (void) const {
  return (leRayon * 2) ;
}

float Cercle::perimetre (void) const {
  return (2 * pi * leRayon) ;
}

float Cercle::surface (void) const {
  return (pi * leRayon * leRayon) ;
}
```


Le programme de test

```
int main (void) {
  Point centre (2, 3), delta (6, 5) ;
  ObjetGeometrique * unObjet ;
  unObjet = new Rectangle (centre, delta) ;
  unObjet->afficher () ;
  unObjet->deplacer (delta) ;
  unObjet->afficher () ;
  delete unObjet ;
  unObjet = new Cercle (delta, 6) ;
  unObjet->afficher () ;
  unObjet->deplacer (delta) ;
  unObjet->afficher () ;
  delete unObjet ;
  return 0 ;
}
```

L'exécution du test

```
<tduval@cigale> testgeo
constructeur ObjetGeometrique
constructeur Rectangle
4 4 12 8 4 2
10 9 12 8 4 2
destructeur Rectangle
destructeur ObjetGeometrique
constructeur ObjetGeometrique
constructeur Cercle
6 5 18.8495 28.2743 3 6
12 10 18.8495 28.2743 3 6
destructeur Cercle
destructeur ObjetGeometrique
<tduval@cigale>
```


Les exceptions en C++

- Introduction
- Du C au C++ : le premier +
- Les classes en C++
- Surdéfinition des opérateurs
- Espaces de nommage et STL
- L'héritage
- **Les exceptions**
  1. **La gestion des exceptions** **339**
  2. **Les classes d'exception standard** **340**
  3. *try / catch / throw* **341**
  4. **Exemples** **342**
- Etude de cas : des piles
- Bibliographie
- Sommaire

La gestion des exceptions

– La bibliothèque standard C++ fournit une classe de base pour déclarer des objets destinés à être lancés comme exceptions

– C'est la classe *exception* du package *std* qu'on inclut à l'aide de l'entête *<exception>*

– Cette classe propose une mt́hode *const char * what (void)* destiné à informer sur la nature de l'exception

Exceptions standards

– Toutes les classes de la bibliothèque standard C++ lancent des exceptions dérivées de la classe *exception* :

– *bad_alloc* : lancée par un échec d'allocation mémoire

– *bad_cast* : lancée par un échec de *dynamic_cast*

– *bad_exception* : lancée quand une exception ne peut pas être attrapée

– *bad_typeid* : lancée par un échec de typage (?)

– *ios_base : :failure* : lancée par des fonctions d'entrées/sorties

try / catch / throw

– Une exception peut être lancé à l'aide de la clause *throw*

– Les fonctions ou les méthodes susceptibles de lancer des exceptions doivent être invoquées dans un bloc *try / catch*

– Il est possible de limiter les exceptions pouvant être lancés par une fonction ou une méthode en précisant à la fin de sa déclaration :
  – *throw (MyException)* : seulement des exceptions du type précisé
  – *throw ()* : aucune exception autorisée

Classe d'exception

```
#include <iostream>
#include <exception>

using namespace std;

class NullPointerException : public exception {

  virtual const char * what () const throw () {
    return "NullPointerException happened" ;
  }

} ;
```


Fonction lançant l'exception

```
// pas de limitation de lancement d'exception
void affichageEntierPointe (int * pObjet) {
  if (pObjet != NULL) {
    cout << *pObjet << endl ;
  } else {
    throw NullPointerException () ;
  }
}
```


Traitement de l'exception

```
int main (void) {
  int * myEntier = NULL ;
  try {
    affichageEntierPointe (myEntier) ;
  } catch (exception & e) {
    cout << "Standard exception : " << e.what () << endl;
  } catch (...) {
    cout << "Other exception..." << endl ;
  }
  return 0 ;
}
```

Résultat

```
Standard exception : NullPointerException happened
```


Fonction lançant l'exception - bis

```
// lancement d'exceptions NullPointerException autorise
void affichageEntierPointe (int * pObjet) throw (NullPointerException) {
  if (pObjet != NULL) {
    cout << *pObjet << endl ;
  } else {
    throw NullPointerException () ;
  }
}
```


Résultat - bis

```
Standard exception : NullPointerException happened
```


Fonction lançant l'exception - ter

```
// pas de lancement d'exception autorise
void affichageEntierPointe (int * pObjet) throw () {
  if (pObjet != NULL) {
    cout << *pObjet << endl ;
  } else {
    throw NullPointerException () ;
  }
}
```


Résultat - ter

```
terminate called after throwing an instance of 'NullPointerException'
  what():  NullPointerException happened
```


Etude de cas : des piles

■ Introduction
■ Du C au C++ : le premier +
■ Les classes en C++
■ Surdéfinition des opérateurs
■ Espaces de nommage et STL
■ L'héritage
■ Les exceptions
■ **Etude de cas : des piles**
1. **Problématique** 351
2. **La classe** *Liste* **à réutiliser** 352
3. **La classe** *PileAbstraite* 369
4. **La classe** *PileComposee* 376
5. **La classe** *PileDerivee* 384
6. **Le test** 392
■ Bibliographie
■ Sommaire

Problématique

– Créer une classe *Pile* concrète en :
  – héritant d'une classe *Pile* abstraite
  – réutilisant une classe *Liste* fournie

– Possibilités :
1. hériter publiquement de la classe pile abstraite et utiliser une instance de liste (protégée ou privée)
2. hériter publiquement de la classe pile abstraite et hériter de façon protégée ou privée de la classe liste

La classe *Liste* à réutiliser

– Elle est générique

– Elle est doublement chaînée

– Elle n'est pas forcément idéale mais elle a le mérite d'exister

– Elle va nous permettre de développer rapidement nos piles . . .

Liste.h (1/15)

```
#ifndef LISTE_H
#define LISTE_H

#include <iostream>

template <class Type>

class Liste {

  public :

    Liste (void) ;
    Liste (const Liste<Type> &) ;
    virtual ~Liste (void) ;

    Liste<Type> & operator = (const Liste<Type> &) ;
```


Liste.h (2/15)

```
    virtual void ajouterEnTete (const Type &) ;
    virtual void enleverEnTete (void) ;
    virtual void ajouterEnQueue (const Type &) ;
    virtual void enleverEnQueue (void) ;
    virtual Type & tete (void) const ;
    virtual Type & queue (void) const ;
    virtual bool vide (void) const ;

  template <class T> friend bool operator == (const Liste<T> &, const Liste<T> &) ;
  template <class T> friend bool operator != (const Liste<T> &, const Liste<T> &) ;
  template <class T> friend std::ostream & operator << (std::ostream &, const Liste<T> &) ;
```


Liste.h (3/15)

```
  protected :

    struct _Element {
      Type valeur ;
      _Element * suivant ;
      _Element * precedent ;
    } ;

    _Element * _tete ;
    _Element * _queue ;

    virtual void _copier (const Liste<Type> &) ;
    virtual void _vider (void) ;

} ;
```


Liste.h (4/15)

```
#include <cassert>

template <class Type>
inline Liste<Type>::Liste (void) {
  _tete = 0 ;
  _queue = 0 ;
}

template <class Type>
inline Liste<Type>::Liste (const Liste<Type> & l) {
  _copier (l) ;
}
```


Liste.h (5/15)

```
template <class Type>
inline Liste<Type>::~Liste (void) {
  _vider () ;
}

template <class Type>
inline Liste<Type> & Liste<Type>::operator = (const Liste<Type> & l) {
  if (this != &l) {
    _vider () ;
    _copier (l) ;
  }
  return (*this) ;
}
```


Liste.h (6/15)

```
template <class Type>
inline void Liste<Type>::ajouterEnTete (const Type & x) {
  _Element * nouveau = new _Element ;
  nouveau->valeur = x ;
  nouveau->suivant = _tete ;
  nouveau->precedent = 0 ;
  if (_queue == 0) {
    _queue = nouveau ;
  } else {
    _tete->precedent = nouveau ;
  }
  _tete = nouveau ;
}
```


Liste.h (7/15)

```
template <class Type>
inline void Liste<Type>::enleverEnTete (void) {
  assert (! vide ()) ;
  _Element * ancien ;
  ancien = _tete ;
  _tete = _tete->suivant ;
  if (_tete == 0) {
    _queue = 0 ;
  } else {
    _tete->precedent = 0 ;
  }
  delete ancien ;
}
```


Liste.h (8/15)

```
template <class Type>
inline void Liste<Type>::ajouterEnQueue (const Type & x) {
  _Element * nouveau = new _Element ;
  nouveau->valeur = x ;
  nouveau->suivant = 0 ;
  nouveau->precedent = _queue ;
  if (_queue == 0) {
    _tete = nouveau ;
  } else {
    _queue->suivant = nouveau ;
  }
  _queue = nouveau ;
}
```

Liste.h (9/15)

```
template <class Type>
inline void Liste<Type>::enleverEnQueue (void) {
  assert (! vide ()) ;
  _Element * ancien ;
  ancien = _queue ;
  _queue = _queue->precedent ;
  if (_queue == 0) {
    _tete = 0 ;
  } else {
    _queue->suivant = 0 ;
  }
  delete ancien ;
}
```


Liste.h (10/15)

```
template <class Type>
inline Type & Liste<Type>::tete (void) const {
  assert (! vide ()) ;
  return (_tete->valeur) ;
}

template <class Type>
inline Type & Liste<Type>::queue (void) const {
  assert (! vide ()) ;
  return (_queue->valeur) ;
}

template <class Type>
inline bool Liste<Type>::vide (void) const {
  return (_tete == 0) ;
}
```


Liste.h (11/15)

```
template <class Type>
inline void Liste<Type>::_copier (const Liste<Type> & l) {
  _tete = 0 ;
  _queue = 0 ;
  if (! l.vide ()) {
    _Element * prec ;
    _Element * ancien = l._tete ;
    _Element * nouveau = new _Element ;
    nouveau->valeur = (*ancien).valeur ;
    nouveau->precedent = 0 ;
    _tete = nouveau ;
```


Liste.h (12/15)

```
    while (ancien->suivant != 0) {
      ancien = ancien->suivant ;
      nouveau->suivant = new _Element ;
      prec = nouveau ;
      nouveau = nouveau->suivant ;
      nouveau->precedent = prec ;
      nouveau->valeur = ancien->valeur ;
    }
    nouveau->suivant = 0 ;
    _queue = nouveau ;
  }
}
```


Liste.h (13/15)

```
template <class Type>
inline void Liste<Type>::_vider (void) {
  while (! vide ()) {
    enleverEnTete () ;
  }
}

template <class Type>
inline std::ostream & operator << (std::ostream & out, const Liste<Type> & p) {
  struct Liste<Type>::_Element * courant = p._tete ;
  while (courant != 0) {
    out << courant->valeur << ' ' ;
    courant = courant->suivant ;
  }
  out << " " ;
  return (out) ;
}
```


Liste.h (14/15)

```
template <class Type>
inline bool operator == (const Liste<Type> & l1, const Liste<Type> &l2) {
  struct Liste<Type>::_Element * e1 = l1._tete ;
  struct Liste<Type>::_Element * e2 = l2._tete ;
  bool identique = true ;
  while ((e1 != 0) && (e2 != 0) && (identique)) {
    identique = (e1->valeur == e2->valeur) ;
    e1 = e1->suivant ;
    e2 = e2->suivant ;
  }
  return (identique && (e1 == 0) && (e2 == 0)) ;
}
```


Liste.h (15/15)

```
template <class Type>
inline bool operator != (const Liste<Type> & l1, const Liste<Type> &l2) {
  return (! (l1 == l2)) ;
}

#endif
```


Liste.C

```
#include "Liste.h"
```

La classe *PileAbstraite* à réutiliser

– Elle décrit le TAD pile

– Elle y ajoute une notion de nom

– Elle implémente quelques opérateurs utilitaires :
  – injection d'un élément dans une pile
  – extraction d'un élément à partir d'une pile
  – injection d'une pile dans un flot de sortie

PileAbstraite.h (1/5)

```
#ifndef PILEABSTRAITE_H
#define PILEABSTRAITE_H

#include <iostream>

template <class Type>

class PileAbstraite {

  public :

    PileAbstraite (char * = "PileAbstraite") ;
    virtual ~PileAbstraite (void) ;

    virtual char * nom (void) const ;
    virtual void nommer (char *) ;
```


PileAbstraite.h (2/5)

```
    virtual void empiler (const Type &) = 0 ;
    virtual void depiler (void) = 0 ;
    virtual Type & sommet (void) const = 0 ;
    virtual bool vide (void) const = 0 ;

    virtual PileAbstraite<Type> & operator << (const Type &) ;
    virtual PileAbstraite<Type> & operator >> (Type &) ;
    template <class T>
         friend std::ostream & operator << (std::ostream &, const PileAbstraite<T> &) ;

  protected :

    char * _nom ;
    virtual void afficher (ostream & out = std::cout) const = 0 ;

} ;
```


PileAbstraite.h (3/5)

```
#include <string.h>

template <class Type>
inline PileAbstraite<Type>::PileAbstraite (char * nom) {
  _nom = strdup (nom) ;
}

template <class Type>
inline PileAbstraite<Type>::~PileAbstraite (void) {
   delete [] _nom ;
}
```


PileAbstraite.h (4/5)

```
template <class Type>
inline char * PileAbstraite<Type>::nom (void) const {
  return (_nom) ;
}

template <class Type>
inline void PileAbstraite<Type>::nommer (char * nom) {
  delete [] _nom ;
  _nom = strdup (nom) ;
}

template <class Type>
inline std::ostream & operator << (std::ostream & out, const PileAbstraite<Type> & p) {
  p.afficher (out) ;
  return (out) ;
}
```


PileAbstraite.h (5/5)

```
template <class Type>
inline PileAbstraite<Type> &
     PileAbstraite<Type>::operator << (const Type & e) {
  empiler (e) ;
  return (*this) ;
}

template <class Type>
inline PileAbstraite<Type> & PileAbstraite<Type>::operator >> (Type & e) {
  e = sommet () ;
  depiler () ;
  return (*this) ;
}

#endif
```


PileAbstraite.C

```
#include "PileAbstraite.h"
```


La classe *PileComposee*

– Elle dérive publiquement de la classe *PileAbstraite*

– Elle utilise une *Liste*

PileComposee.h (1/6)

```
#ifndef PILECOMPOSEE_H
#define PILECOMPOSEE_H
#include <iostream>
#include "PileAbstraite.h"
#include "Liste.h"

template <class Type>

class PileComposee : public PileAbstraite<Type> {

  public :

    PileComposee (char * = "PileComposee") ;
    PileComposee (const PileComposee<Type> &) ;
    virtual ~PileComposee (void) ;

    PileComposee<Type> & operator = (const PileComposee<Type> &) ;
```


PileComposee.h (2/6)

```
    virtual void empiler (const Type &) ;
    virtual void depiler (void) ;
    virtual Type & sommet (void) const ;
    virtual bool vide (void) const ;

    template <class T> friend bool operator == (const PileComposee<T> &,
                                   const PileComposee<T> &) ;
    template <class T> friend bool operator != (const PileComposee<T> &,
                                   const PileComposee<T> &) ;

  protected :

    Liste<Type> _liste ;
    virtual void afficher (std::ostream & out = std::cout) const ;

} ;
```


PileComposee.h (3/6)

```
template <class Type>
inline PileComposee<Type>::PileComposee (char * nom)
            : PileAbstraite<Type> (nom) {
}

template <class Type>
inline PileComposee<Type>::PileComposee (const PileComposee<Type> & p)
            : PileAbstraite<Type> (p._nom), _liste (p._liste) {
}

template <class Type>
inline PileComposee<Type>::~PileComposee (void) {
}
```


PileComposee.h (4/6)

```
template <class Type>
inline PileComposee<Type> &
     PileComposee<Type>::operator = (const PileComposee<Type> & p) {
  _liste = p._liste ;
  return (*this) ;
}

template <class Type>
inline void PileComposee<Type>::empiler (const Type & valeur) {
  _liste.ajouterEnTete (valeur) ;
}

template <class Type>
inline void PileComposee<Type>::depiler (void) {
  _liste.enleverEnTete () ;
}
```


PileComposee.h (5/6)

```
template <class Type>
inline Type & PileComposee<Type>::sommet (void) const {
  return (_liste.tete ()) ;
}

template <class Type>
inline bool PileComposee<Type>::vide (void) const {
  return (_liste.vide ()) ;
}

template <class Type>
inline void PileComposee<Type>::afficher (std::ostream & out) const {
  out << PileAbstraite<Type>::_nom << " " << _liste ;
}
```


PileComposee.h (6/6)

```
template <class Type>
inline bool operator == (const PileComposee<Type> & p1,
               const PileComposee<Type> & p2) {
  return (p1._liste == p2._liste) ;
}

template <class Type>
inline bool operator != (const PileComposee<Type> & p1,
               const PileComposee<Type> & p2) {
  return (! (p1 == p2)) ;
}

#endif
```


PileComposee.C

```
#include "PileComposee.h"
```


La classe *PileDerivee*

– Elle dérive publiquement de la classe *PileAbstraite*

– Elle dérive en mode protégé de la classe *Liste*

PileDerivee.h (1/6)

```
#ifndef PILEDERIVEE_H
#define PILEDERIVEE_H
#include <iostream>
#include "PileAbstraite.h"
#include "Liste.h"

template <class Type>

class PileDerivee : public PileAbstraite<Type>, protected Liste<Type> {

  public :

    PileDerivee (char * = "PileDerivee") ;
    PileDerivee (const PileDerivee<Type> &) ;
    virtual ~PileDerivee (void) ;

    PileDerivee<Type> & operator = (const PileDerivee<Type> &) ;
```


PileDerivee.h (2/6)

```
    virtual void empiler (const Type &) ;
    virtual void depiler (void) ;
    virtual Type & sommet (void) const ;
    virtual bool vide (void) const ;

    template <class T> friend bool operator == (const PileDerivee<T> &,
                                 const PileDerivee<T> &) ;
    template <class T> friend bool operator != (const PileDerivee<T> &,
                                 const PileDerivee<T> &) ;

  protected :

    virtual void afficher (std::ostream & out = std::cout) const ;

} ;
```


PileDerivee.h (3/6)

```
template <class Type>
inline PileDerivee<Type>::PileDerivee (char * nom)
        : PileAbstraite<Type> (nom), Liste<Type> () {
}

template <class Type>
inline PileDerivee<Type>::PileDerivee (const PileDerivee<Type> & p)
        : PileAbstraite<Type> (p._nom), Liste<Type> (p) {
}

template <class Type>
inline PileDerivee<Type>::~PileDerivee (void) {
}
```


PileDerivee.h (4/6)

```
template <class Type>
inline PileDerivee<Type> &
    PileDerivee<Type>::operator = (const PileDerivee<Type> & p) {
  Liste<Type>::operator = (p) ;
  return (*this) ;
}

template <class Type>
inline void PileDerivee<Type>::empiler (const Type & valeur) {
  ajouterEnTete (valeur) ;
}

template <class Type>
inline void PileDerivee<Type>::depiler (void) {
  enleverEnTete () ;
}
```


PileDerivee.h (5/6)

```
template <class Type>
inline Type & PileDerivee<Type>::sommet (void) const {
  return (tete ()) ;
}

template <class Type>
inline bool PileDerivee<Type>::vide (void) const {
  return (Liste<Type>::vide ()) ;
}

template <class Type>
inline void PileDerivee<Type>::afficher (std::ostream & out) const {
  out << PileDerivee<Type>::_nom << " " << (Liste<Type>)(*this) ;
}
```


PileDerivee.h (6/6)

```
template <class Type>
inline bool operator == (const PileDerivee<Type> & p1,
               const PileDerivee<Type> & p2) {
  return ((Liste<Type>)p1 == (Liste<Type>)p2) ;
}

template <class Type>
inline bool operator != (const PileDerivee<Type> & p1,
               const PileDerivee<Type> & p2) {
  return (! (p1 == p2)) ;
}

#endif
```


PileDerivee.C

```
#include "PileDerivee.h"
```


Le test

– Mélange d'instances de *PileComposee* et de *PileDerivee* au travers d'instances de *PileAbstraite*

MainPile.C (1/5)

```
#include "PileAbstraite.h"
#include "PileDerivee.h"
#include "PileComposee.h"

void hanoi (int n, PileAbstraite <int> * d, PileAbstraite <int> * i, PileAbstraite <int> * a) {
  if (n > 0) {
    hanoi (n - 1, d, a, i) ;
    int x ;
    *d >> x ;
    std::cout << "deplacer " << x << " de " << d->nom ()
                    << " vers " << a->nom () << std::endl ;
    *a << x ;
    std::cout << "      " << *d << *i << *a << std::endl ;
    hanoi (n - 1, i, d, a) ;
  }
}
```


MainPile.C (2/5)

```
int main (void) {
  PileDerivee <double> p1 ;
  p1 << 0.1 << 2.3 << 4.5 ;
  PileDerivee <double> p2 = p1 ;
  PileDerivee <double> p3 ;
  p3 = p1 ;
  std::cout << *(PileAbstraite<double> *)&p1 << std::endl
      << *(PileAbstraite<double> *)&p2 << std::endl
      << *(PileAbstraite<double> *)&p3 << std::endl ;
  if (p1 == p3) {
    std::cout << "p1 = p3" << std::endl ;
  } else {
    std::cout << "p1 != p3" << std::endl ;
  }
  p1.nommer ("p1") ;
  p2.nommer ("p2") ;
  p3.nommer ("p3") ;
```


MainPile.C (3/5)

```
  p2.empiler (6.7) ;
  p3.depiler () ; p3.empiler (8.9) ;
  std::cout << *(PileAbstraite<double> *)&p1 << std::endl
      << *(PileAbstraite<double> *)&p2 << std::endl
      << *(PileAbstraite<double> *)&p3 << std::endl ;
  if (p1 != p3) {
    std::cout << "p1 != p3" << std::endl ;
  } else {
    std::cout << "p1 = p3" << std::endl ;
  }
  std::cout << p3.sommet () << std::endl ;
  PileComposee<const char *> p4 ("p4") ;
  p4 << "un" << "deux" << "trois" ;
  PileComposee<const char *> p5 (p4) ;
  PileComposee<const char *> p6 ("p6") ;
  p6 = p4 ;
  std::cout << p4 << std::endl << p5 << std::endl << p6 << std::endl ;
```


MainPile.C (4/5)

```
  if (p6 == p5) {
    std::cout << "p6 = p5" << std::endl ;
  } else {
    std::cout << "p6 != p5" << std::endl ;
  }
  p5.nommer ("p5") ;
  p5.depiler () ;
  p5.empiler ("quatre") ;
  std::cout << p4 << std::endl << p5 << std::endl << p6 << std::endl ;
  if (p6 != p5) {
    std::cout << "p6 != p5" << std::endl ;
  } else {
    std::cout << "p6 = p5" << std::endl ;
  }
```


MainPile.C (5/5)

```
  PileComposee<int> d ("depart") ;
  PileAbstraite<int> * i = new PileComposee<int> ("intermediaire") ;
  PileAbstraite<int> * a = new PileDerivee<int> ("arrivee") ;
  int n ;
  std::cout << "nombre d'anneaux : " ;
  std::cin >> n ;
  for (int nb = n ; nb >= 1 ; nb --) {
    d << nb ;
  }
  std::cout << "debut : " << d << *i << *a << std::endl ;
  hanoi (n, &d, i, a) ;
  std::cout << "fin : " << d << *i << *a << std::endl ;
  return 0 ;
}
```


makefile

```
pile : PileAbstraite.o PileDerivee.o PileComposee.o Liste.o mainPile.o
        g++ -o pile mainPile.o
mainPile.o : mainPile.C PileAbstraite.h PileComposee.h PileDerivee.h
        g++ -c mainPile.C
Liste.o : Liste.h Liste.C
        g++ -c Liste.C
PileAbstraite.o : PileAbstraite.h PileAbstraite.C
        g++ -c PileAbstraite.C
PileDerivee.o : PileAbstraite.h PileDerivee.h PileDerivee.C
        g++ -c PileDerivee.C
PileComposee.o : PileAbstraite.h PileComposee.h PileComposee.C
        g++ -c PileComposee.C
clean :
        rm -f pile *.o a.out core
```


Exécution du test (1/2)

```
PileDerivee 4.5 2.3 0.1
PileDerivee 4.5 2.3 0.1
PileDerivee 4.5 2.3 0.1
p1 = p3
p1 4.5 2.3 0.1
p2 6.7 4.5 2.3 0.1
p3 8.9 2.3 0.1
p1 != p3
8.9
p4 trois deux un
p4 trois deux un
p6 trois deux un
p6 = p5
```


Exécution du test (2/2)

```
p4 trois deux un
p5 quatre deux un
p6 trois deux un
p6 != p5
nombre d'anneaux : 2
debut : depart 1 2  intermediaire  arrivee
deplacer 1 de depart vers intermediaire
        depart 2  arrivee  intermediaire 1
deplacer 2 de depart vers arrivee
        depart  intermediaire 1  arrivee 2
deplacer 1 de intermediaire vers arrivee
        intermediaire  depart  arrivee 1 2
fin : depart  intermediaire  arrivee 1 2
<tduval@cigale>
```

Bibliographie

- Introduction
- Du C au C++ : le premier +
- Les classes en C++
- Surdéfinition des opérateurs
- La généricité
- L'héritage
- Les exceptions
- Etude de cas : des piles
- **Bibliographie**
- Sommaire

Références (1/2)

– J.O. Coplien : *Programmation avancée en C++ : styles et idiomes*, Addison–Wesley, 1992
– M.J. Ellis, B. Stroustrup : *The annoted C++ Reference Manual*, Addison–Wesley, 1990
– S.B. Lippman : *L'essentiel du C++*, Addison–Wesley, 1992
– **S. Morvan, J. Tisseau : *Programmation par objets — Le langage C++*, Notes de cours ENIB, 1994/1996**
– R.B. Murray : *Stratégies et tactiques en C++*, Addison–Wesley, 1994
– S. Oualline : *Programmation C++ par la pratique*, Editions O'Reilly, 1997
– I. Pohl : *C++ for C programmers*, Benjamin/Cummings, 1989
– P. Prados : *La qualité en C++*, Eyrolles, 1996
– B. Stroustrup : *Le langage C++*, International Thomson Publishing France, 1996

Références (2/2)

– http ://www.cppreference.com/

– http ://www.cplusplus.com/ref/

– http ://www.dofactory.com/Patterns/Patterns.aspx

– http ://www.mindspring.com/ mgrand/ pattern_synopses.htm

– http ://www2.research.att.com/ bs/C++.html

Sommaire

- Introduction 3
- C++ : le premier + ... 22
- Les classes en C++ 56
- Surdéfinition des opérateurs 158
- Espaces de nommage et STL 242
- L'héritage 260
- Les exceptions 338
- Etude de cas : des piles 350
- Bibliographie 401
- **Sommaire 404**

- Introduction 3
  1. Origines du C++ 4
  2. Le langage C++ 5
  3. Premier programme en C++ 6
  4. Première classe en C++ : une *Pile* 7
- Du C au C++ : le premier + ... 22
  1. C et C++ 23
     (a) Compatibilité C/C++ 24
     (b) Un typage + fort 25
     (c) + de mots réservés 26
     (d) Des entrées/sortie + simples 27
  2. Les variables en C++ 29
     (a) Définition au + près 30
     (b) Définition des constantes 31
     (c) Les références en + ... 32
     (d) Résolution de portée 33
     (e) Allocation dynamique de mémoire + simple 34
  3. Les fonctions en C++ 36
     (a) Prototypes de fonctions 37
     (b) Fonctions en ligne 39
     (c) Arguments par défaut 41
     (d) Surdéfinition des fonctions 43
     (e) Fonctions génériques 45
     (f) Passage des arguments 49
        i. Passage par valeurs 50
        ii. Passage par pointeurs 52
        iii. Passage par références 54
- Les classes en C++ 56
  1. C++ : le deuxième + ... 57
     (a) C++ et programmation par objets 58
     (b) Classification 59
  2. Les classes 60
     (a) Implémentation d'un TAD 61
     (b) Masquage de l'information 63
     (c) Classe ≡ composant logiciel 66
     (d) Compilation séparée en C++ 68
     (e) Classe *Point* 69
  3. Les objets 74
     (a) La vie d'un objet 75
     (b) Création et initialisation 76
     (c) Affectation 77
     (d) Opérations par défaut 78
     (e) Dynamique des objets 91
     (f) Auto–référence : le pointeur *this* 93
     (g) Variables de classe 95
  4. Les fonctions membres 100
     (a) Prototype d'une fonction membre 101
     (b) Fonctions membres *const* 102
     (c) Constructeurs 104
     (d) Destructeur 108
     (e) Fonctions membres *static* 118
     (f) Pointeurs sur fonctions membres 126
  5. Les fonctions amies 129
     (a) Les *friend* 130
     (b) Les fonctions *friend* indépendantes 131
     (c) Les fonctions *friend* membres d'une autre classe 133
     (d) Les fonctions *friend* amies de plusieurs classes 136
     (e) Les classes *friend* amies d'autres classes 139
  6. Les classes génériques 141
     (a) Les types génériques 142
     (b) Exemple simple de classe générique 143
     (c) Exemple complet de classe générique 146
- Surdéfinition des opérateurs 158
  1. Les opérateurs en C++ 159
     (a) Opérateurs prédéfinis 160
     (b) Opérateur ≡ Fonction 166
     (c) Opérateurs surchargeables 167
  2. Modes de surdéfinition : comment surdéfinir ? 169
     (a) Membre ou non membre ? 170
     (b) Opérateur membre 171
     (c) Opérateur non membre 176
     (d) Critères de choix 181
     (e) Bonnes habitudes : cohérence des opérateurs 182
  3. Exemples de surdéfinitions 187
     (a) Une classe *PointNommeEvolue* 188
     (b) Une classe *Pile* 202
     (c) Une classe *Pile* générique 221
- Espaces de nommage et STL 242
  1. Espaces de nommage 243
  2. STL 249

3. STL et Itérateurs 250

■ L'héritage 260

1. Mécanismes de base
(a) Principes 262
(b) Remarques 263
(c) Spécialisation 264
(d) Enrichissement 265
(e) Héritage simple 266
(f) Héritage multiple 272

2. Masquage de l'information : accès aux membres
(a) Autorisation d'accès aux attributs et méthodes 280
(b) Fonctions non membres de la classe de base 281
(c) Fonctions membres de la classe de base 283
(d) Fonctions d'une classe dérivée 284

3. Modes de dérivation
(a) Accès aux classes de base 287
(b) Dérivation publique 288
(c) Dérivation protégée 291
(d) Dérivation privée 294

4. Envois de messages
(a) Liaison statique 298
(b) Liaison dynamique 304
(c) Polymorphisme 318

5. Classes abstraites
(a) Méthodes virtuelles pures 321
(b) Classes concrètes dérivées 322
(c) Pourquoi des classes abstraites ? 323
(d) Exemple de classe abstraite 324

■ Les exceptions 338

1. La gestion des exceptions 339
2. Les classes d'exception standard 340
3. *try / catch / throw* 341
4. Exemples 342

■ Etude de cas : des piles 350

1. Problématique 351
2. La classe *Liste* à réutiliser 352
3. La classe *PileAbstraite* 369
4. La classe *PileComposee* 376
5. La classe *PileDerivee* 384
6. Le test 392

■ Bibliographie 401

■ Sommaire 404