SCIENCES SORBONNE UNIVERSITÉ

Programmation Répartie
Master 1 Informatique – 4I400

Cours 1 : Introduction au C++

Yann Thierry-Mieg
Yann.Thierry-Mieg@lip6.fr

Organisation

- 2h cours, 2h TD, 2h TME par semaine
- Examen réparti 1 (novembre) sur machines
- Examen réparti 2 (janvier) sur feuille
- Répartition : 10% note de TME, 30% Exam 1, 60% Exam 2.

- Attention : refonte des supports, nouvelle version de l'UE
- Passage au C++ !

C++ vs C

- C++ est un sur-ensemble du C98 standard
  - Types de données : char, int, long, float, double …+ unsigned
  - Tableaux, struct, pointeurs
  - Structures de contrôle : if/then/else, for (i=0 ; i<N; i++), while, do/while, switch/case
  - Fonctions, paramètres typés, sémantique mémoire stack/heap
- En plus on trouve :
  - Namespaces, visibilités
  - Type référence, type const
  - Classes, instances, orientation objet (héritage)
  - Polymorphisme très riche
  - Redéfinition d’opérateurs : +, *, &&, =, …
  - Généricité via templates
  - Librairie standard riche, librairies C++ efficaces
  - Lambda, inférence de type

C++ vs Java

- C++ partage avec Java
  - Les concepts d'orienté objet : classe, instance, héritage
  - Beaucoup de la syntaxe
- C++ offre en plus/moins
  - Pas de garbage collector, pas de classe parente Object
  - Accès fin à la mémoire / Gestion mémoire plus difficile
  - Généricité via la substitution vs « erased types »
- C++ offre en plus
  - Interaction immédiate avec les API noyau, devices, matériel
  - Efficacité mémoire accrue (~x10), performances
- Java offre en plus
  - Réflexion, introspection, dynamicité
  - Modèle sémantique uniforme et simple
  - Lib standard et non standard très étendue

Le Langage C++

- Sur-ensemble du C : gestion fine de la mémoire, du matériel
- Langage compilé : très efficace, pas de runtime
- Langage orienté objet : structuration des applications, variabilité
- Langage moderne en rapide évolution : C++11, 14, 17 et bientôt 20
- Fort support industriel : intel, microsoft, jeux…
- S'interface bien avec des langages front-end come Python
- Langage très versatile et puissant

Mais

- Langage relativement complexe, beaucoup de concepts

- Inutile de maîtriser tout le langage pour l'UE, deux séances pour apprendre la syntaxe et l'environnement + concepts plus avancés abordés au fil des séances

Plan des Séances

Objectif : principes de programmation d'applications concurrentes et réparties

1. Introduction C++, modèle mémoire, modèle objet
2. Conteneurs, Itérateurs, lib standard
3. Programmation concurrente : threads, mutex
4. Synchronisations : conditions, partage mémoire
5. Multi-processus : fork, exec, signal
6. Communications interprocessus (IPC) : shm, sem, pipe, …
7. Communications distantes : Sockets
8. Protocoles de communication : Sérialisation, protobuf
9. Parallélisme grain fin : lock-free, work steal
10. Ouverture : introduction à MPI, CUDA, API Cloud

Références

- Hypothèse / pré-requis :
  - Niveau intermédiaire en C,
  - Niveau confirmé en Java (POBJ, PRC en L3 ?)
- Références web :
  - http://www.cplusplus.com/ (tutos, docs de références)
  - https://fr.cppreference.com/ (des parties traduites, des parties en français)
  - Une version en ligne du man : https://man.cx/
  - StackOverflow : https://stackoverflow.com/
- Références biblio utiles :
  - Stroustrup, « The C++ Programming Language », 4th Ed (C++11)
  - Gamma, Helm, Vlissides, Johnson, « Design Patterns »
  - Meyers, « Effective XXX » XXX=C++, STL, modern C++
- Cours en partie basé sur les supports de
  - Denis Poitrenaud (P5), et Souheib Baarir (P10) sur le c++
  - L. Arantes, P.Sens (P6) sur Posix

Compilation : sources

- Source divisés en :
  - Header (.h, .hh, .hpp) : déclarations de types, de variables et de fonctions
  - Source (.cpp, .cc) : corps des fonctions, définition des variables statiques
- On utilise #include pour inclure un source
  - Gestion faite par le préprocesseur cpp
  - Headers standards : <string>, <vector>, <iostream>
  - Headers personnels : « MyClass.h », « util/Utility.h»
- Attention aux double include

```
#ifndef MYCLASS_H
#define MYCLASS_H
// includes, déclarations
#endif
```

- Sources plateformes indépendantes ?
  - Lib standard OK, « stdunix.h » « windows.h »… NOK
- Le préprocesseur traite aussi les macros, dont on déconseille l'usage dans l'UE.

Compilation : unité de compilation .o

- Chaque fichier source est compilé séparément
  - Un fichier .cpp -> .o, fichier binaire plateforme dépendant
  - Contient : le code des fonctions du cpp, de l'espace pour les static déclarés et les littéraux (constantes, chaînes) du programme
  - Référence indirectement les divers .h dans lequel il a trouvé les références aux fonctions invoquées dans le code
  - La compilation détecte un certain nombre de problèmes de syntaxe et de typage
  - La compilation cherche la déclaration adaptée, réalise les instanciations de paramètres et vérifie le typage des invocations
- Concrètement on passe au compilateur
  - un MyClass.cpp source
  - -c pour arrêter la compilation avant le link
  - -Wall pour activer les warnings
  - -std=c++1y (selon compilo) pour le langage
  - -g pour activer les symboles de debug : le .o garde des liens vers les sources

Compilation : librairie, exécutable

- Une librairie est un ensemble de .o agglomérés
  - Plateforme dépendant
  - Consistent (pas de double déclarations entre les .o)
  - On peut distribuer la librairie (pour une plateforme donnée, e.g. linux_x64) avec ses fichiers .h constituant son API
- Librairie statique ou dynamique
  - Change la façon dont l'exécutable se lie à la librairie
  - Lib dynamique : .so/.dylib/.dll, l'application invoque la lib (mise à jour possible de la lib sans recompiler l'application)
  - Lib statique : .a, .la l'application embarque le code utile de la lib
- Un exécutable est une application
  - Construit à partir d'un ensemble de .o et de librairies
  - Un seul des .o doit contenir une fonction **main**.
  - Embarque sélectivement le code des .o/.a fournis, lie sur les .so
  - Compilateurs modernes link optimisé : -fwhole-program

Hello world !

```
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
}
```

- Opérateur de résolution de namespace ::
- Headers standard, sans .h, entre <>
- Flux standard : cout, cerr, cin
- Opérateur << pour « pousser » dans le flux
- Fin de ligne + flush : endl

Entrée/sorties

- $<$ *iostream* $>$ offre une interface O.O. plus sécuritaire que $<$ *stdio.h* $>$.
- Les flots d'entrées/sorties prédéfinis sont :
  - *cout* associé à **la sortie standard** ($\Leftrightarrow$ *stdout* en C),
  - *cerr* associé à **la sortie erreur standard** ($\Leftrightarrow$ *stderr* en C),
  - *cin* associé à **l'entrée standard** ($\Leftrightarrow$ *stdin* en C).
- La fonction de sortie *printf*(...) est remplacé par l'opérateur d'insertion $<<$.
- La fonction de sortie *scanf*(...) est remplacé par l'opérateur d'extraction $>>$.

```
std::cout << "bonjour" << std::endl;
// printf("%s \n","bonjour");
int s; std::cin >> s ; // scanf("%d",s);
```

Variables, structures de contrôle

```
#include <iostream>
using namespace std;

int main ()
{
  int a=5;              // initial value: 5
  int b(3);             // initial value: 3
  int result;           //  no initial value

  a = a + b;
  result = a – 2 * b;
  cout << result;

  return 0;
}
```

- Clause « using namespace »
  - Les éléments de std deviennent visible dans ::
- Initialisation des variables
  - Syntaxe affectation = ou fonctionelle ()
  - Pas de valeur par défaut /!\
- Opérations arithmétique et priorités classiques (?)
- std::ostream polymorphique, accepte divers types
  - Cas particulier pour char * interprété comme une string du C

Types de base

Group	Type names*	Notes on size / precision
Character types	**char**	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than char. At least 16 bits.
	char32_t	Not smaller than char16_t. At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	**signed char**	Same size as char. At least 8 bits.
	signed **short** *int*	Not smaller than char. At least 16 bits.
	signed **int**	Not smaller than short. At least 16 bits.
	signed **long** *int*	Not smaller than int. At least 32 bits.
	signed **long long** *int*	Not smaller than long. At least 64 bits.
Integer types (unsigned)	**unsigned char**	(same size as their signed counterparts)
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	
Floating-point types	**float**	
	double	Precision not less than float
	long double	Precision not less than double
Boolean type	**bool**	
Void type	**void**	no storage
Null pointer	**decltype(nullptr)**	

<table>
<tr><th>Level</th><th>Precedence group</th><th>Operator</th><th>Description</th><th>Grouping</th></tr>
<tr><td>1</td><td>Scope</td><td>::</td><td>scope qualifier</td><td>Left-to-right</td></tr>
<tr><td rowspan="4">2</td><td rowspan="4">Postfix (unary)</td><td>++ --</td><td>postfix increment / decrement</td><td rowspan="4">Left-to-right</td></tr>
<tr><td>()</td><td>functional forms</td></tr>
<tr><td>[]</td><td>subscript</td></tr>
<tr><td>. -></td><td>member access</td></tr>
<tr><td rowspan="7">3</td><td rowspan="7">Prefix (unary)</td><td>++ --</td><td>prefix increment / decrement</td><td rowspan="7">Right-to-left</td></tr>
<tr><td>~ !</td><td>bitwise NOT / logical NOT</td></tr>
<tr><td>+ -</td><td>unary prefix</td></tr>
<tr><td>& *</td><td>reference / dereference</td></tr>
<tr><td>new delete</td><td>allocation / deallocation</td></tr>
<tr><td>sizeof</td><td>parameter pack</td></tr>
<tr><td>(type)</td><td>C-style type-casting</td></tr>
<tr><td>4</td><td>Pointer-to-member</td><td>.* ->*</td><td>access pointer</td><td>Left-to-right</td></tr>
<tr><td>5</td><td>Arithmetic: scaling</td><td>* / %</td><td>multiply, divide, modulo</td><td>Left-to-right</td></tr>
<tr><td>6</td><td>Arithmetic: addition</td><td>+ -</td><td>addition, subtraction</td><td>Left-to-right</td></tr>
<tr><td>7</td><td>Bitwise shift</td><td><< >></td><td>shift left, shift right</td><td>Left-to-right</td></tr>
<tr><td>8</td><td>Relational</td><td>< > <= >=</td><td>comparison operators</td><td>Left-to-right</td></tr>
<tr><td>9</td><td>Equality</td><td>== !=</td><td>equality / inequality</td><td>Left-to-right</td></tr>
<tr><td>10</td><td>And</td><td>&</td><td>bitwise AND</td><td>Left-to-right</td></tr>
<tr><td>11</td><td>Exclusive or</td><td>^</td><td>bitwise XOR</td><td>Left-to-right</td></tr>
<tr><td>12</td><td>Inclusive or</td><td>|</td><td>bitwise OR</td><td>Left-to-right</td></tr>
<tr><td>13</td><td>Conjunction</td><td>&&</td><td>logical AND</td><td>Left-to-right</td></tr>
<tr><td>14</td><td>Disjunction</td><td>||</td><td>logical OR</td><td>Left-to-right</td></tr>
<tr><td rowspan="2">15</td><td rowspan="2">Assignment-level expressions</td><td>= *= /= %= += -=
>>= <<= &= ^= |=</td><td>assignment / compound assignment</td><td rowspan="2">Right-to-left</td></tr>
<tr><td>?:</td><td>conditional operator</td></tr>
<tr><td>16</td><td>Sequencing</td><td>,</td><td>comma separator</td><td>Left-to-right</td></tr>
</table>

Opérateurs

- C++ possède un type bool
  - Constantes true et false
  - && and ; || or ; ! not
  - Toute expression != 0 est vraie par promotion
- Les opérateurs sont nombreux et leur règles de priorité complexe
  - Ne pas hésiter à sur-parenthèser un peu les expressions complexes
- Opérateurs new et delete ainsi que sizeof pour la gestion mémoire
- Opérateur de (cast) opère des conversions numériques
- Un sous ensemble de ces opérateurs peut être redéfini pour vos propres types (classes)

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

std::string

La chaîne standard du c++, vient avec ses opérateurs.

```
#include <iostream>
#include <string>

int main() {
  std::string s1;
  std::cin >> s1;
  std::cout << "s1 = " <<  s1
            << std::endl;

  std::string s2 = "abcd";
  std::cout << "s2 = " << s2
            << std::endl;

  if (s1 == s2)
    std::cout << "s1 == s2";
  else if (s1 < s2)
    std::cout << "s1 < s2";
  else
    std::cout << "s1 > s2";
  std::cout << std::endl;
```

```
  for (int i = 0; i < s1.length(); ++i)
    std::cout << s1[i] << s1[i];
  std::cout << std::endl;

  for (int i = 0; i < s1.length(); ++i)
    s1[i] = 'a';
  std::cout << "s1 = " <<  s1
            << std::endl;

  s1 = s2;
  s1 = s1 + s2;
  std::cout << "s1 = " <<  s1
            << std::endl;

  char s[10];
  strcpy(s, s1.c_str());
}
```

Constantes

- Les constantes sont déclarées via le mot clé **`const`**.

Syntaxe

`const` *<id type> <id constante>* = *<expr. constante>*

Exemple

```
const float pi = 3.1416;
```

- Cela indique que l'identificateur doit garder une valeur constante.

```
pi = 3.14; //erreur : modif. interdite
```

- L'initialisation est **obligatoire**.

```
const float pi; //erreur : cste non initialisée
```

- C'est le compilateur qui réalise ces vérifications.
- L'avantage par rapport au macro est essentiellement le contrôle de type.

Constantes et Pointeurs

- Un pointeur vers une variable ne peut pointer vers une constante

```
const float pi = 3.1416;
float* p1 = &pi;          // Erreur : on peut modifier pi via p1
```

- Déclaration de pointeur vers une constante

```
const float* p2 = &pi;      // Initialisation optionnelle
float e = 2.7; p2 = &e;     // OK
*p2 = 2.72;                 // Erreur : p2 pointe sur une constante
```

- Déclaration de pointeur constant

```
float* const p3 = &e;       // Initialisation obligatoire
*p3 = 2.72;                 // OK
p3 = &pi;                   // Erreur : p3 est une constante
```

- Déclaration de pointeur constant vers une constante

```
const float* const p4 = &pi; // Initialisation obligatoire
*p4 = 3.14159;               // Erreur : p4 pointe sur une constante
p4 = &pi;                    // Erreur : p4 est une constante
```

Constantes

Paramètres constants

```
float puissance(const float, const float);
```

- Le compilateur vérifie que les paramètres ne sont pas modifiés dans le corps de la fonction.
- `const` est peu employé dans ce cas car la fonction n'a pas d'effet de bord (les paramètres sont passés par valeur). On évite juste certains bogues.

```
void afficher (const Personne*);
void afficher (const Personne&);
```

- Le compilateur vérifie que la personne pointée (référencée) n'est pas modifiée.
- On a la vitesse du passage par adresse (par référence) et la sécurité du passage par valeur. En prime, on évite certains bogues !

Références

- En C, lors d'un appel de fonction, les paramètres effectifs sont toujours passés **par valeur**.

```
void f(int i) {
  int a = 10;
  i = a;
}
void g(int* i) {
  int a = 20;
  *i = a;
}
```

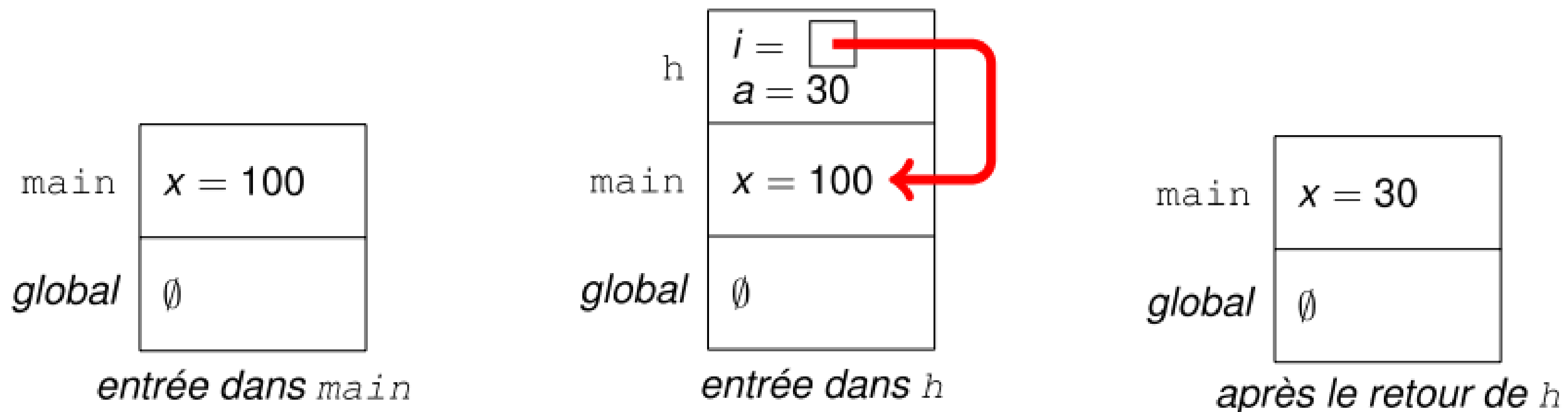
```
int main() {
  int x = 100;
  f(x);   // la valeur de x
  cout << x;
  g(&x); // la valeur de &x
  cout << x;
}
```

- Qu'affiche ce programme ?

- En C++, un nouveau type de passage de paramètres existe : le passage **par référence**.
- La modification de la valeur d'un paramètre passé par référence **est répercutée** au niveau de l'appelant.
- Exemple :

```
void h(int& i) {
  int a = 30;
  i = a;
}
```

```
int main() {
  int x = 100;
  h(x); // une référence sur x
  cout << x;
}
```



- Lors de l'appel `h(x)`, la variable locale `i` ***référence*** la variable `x`.
- Techniquement, c'est l'adresse de la variable fournie en paramètre qui est transmise à la fonction.

Références

Utilisations classiques – Paramètres par référence

```
void swap(int& a, int& b) {
   int tmp = a;
   a = b; b = tmp;
}
```

- La fonction `swap` prend en paramètre deux références vers des entiers.
- Les paramètres effectifs d'un appel ne peuvent être que des variables.

```
int main() {
   int x = 10, y = 20;
   swap(x, y);            // x = 20 et y = 10
   swap(2, x+y);          // error
   // cannot convert param. 1 from 'const int' to 'int &'
   // cannot convert param. 2 from 'int' to 'int &'
}
```

```
struct BigStruct {
  int tab[10000];
  ...
};

void print(const BigStruct& b) {
  std::cout << b.tab[0] << ... << std::endl;
}
```

- Il n'y a aucune contrainte sur les paramètres effectifs.
- Le compilateur contrôle que le contenu de la variable n'est pas modifié.
- On cumule les avantages du passage de paramètre par adresse, la facilité d'écriture et les contrôles du compilateur.

```
const int MAX = 10;
struct Personne {
  char nom[20];
  int age;
};

int& acces(Personne P[], const char nom[]) {
  for (int i = 0; i < MAX; ++i)
    if (strcmp(P[i].nom, nom) == 0)
      return P[i].age;
  // problème si le nom est introuvable
}
```

```
int main(void) {
  Personne Tab[MAX];
  ...
  acces(Tab, "toto") = 2;
}
```

- Une référence permet de créer un nom alternatif pour une variable.

```
int i = 10; int& ri = i; // i et ri désignent la même variable
int j = ri;              // j = 10
ri = 20;                 // i = 20
```

- Une référence doit être initialisée au moment de sa déclaration et référencera toujours la même variable.

```
int& ri;          // erreur, référence non initialisée
```

- Une référence peut être assimilée à un pointeur constant.

```
ri++;     // ri est inchangée, i est incrémentée
```

- Une référence peut désigner une constante uniquement si elle est déclarée comme étant une référence vers une constante.

```
char& r = 'a';              // illégal
const char& r = 'a';        // légal
char c;
const char& rc = c;         // légal
rc = 'a';                   // illégal
```

- Ne pas confondre l'opérateur & permettant d'obtenir l'adresse d'une variable avec le signe & employé pour déclarer des références.
- De manière générale, à partir de n'importe quel type `T`, on peut construire les nouveaux types suivants.
  - `T*` : pointeur vers une donnée de type `T` (`int *pi`).
  - `T[]` : tableau de données de type `T` (`char s[256]`).
  - `T&` : référence vers une donnée de type `T` (`int &ri`).

Une classe : déclaration

```
class <nom de la classe> {
public:
   <interface> => mode d'emploi de la classe
private:
   <partie déclarative de l'implémentation>
};
```

- L'interface et la partie déclarative de la classe comprennent :
  - ✓ des *prototypes de méthodes* (appelées aussi fonctions membres)
  - ✓ des *déclarations de champs* (appelées aussi données membres)

- Habituellement, l'interface **ne contient que** des prototypes de méthodes

Exemple : Pile en C++, utilisation de Classe

```
                class Pile {
                private:
Données            enum {TAILLE = 10};
privées            double tab[TAILLE];
                   unsigned int cpt;                   // nombre d'éléments empilés

                public:
                   Pile ();                            // constructeur vide
                   Pile (double);                      // construit avec 1 élément
                   bool   estPleine () const;
Interface          bool   estVide () const;
publique           void   empiler  (double);           /// @pre !estPleine()
                   double       depiler ();            /// @pre !estVide()
                   double       getSommet () const;    /// @pre !estVide()
                };
```

Une classe : objet et fonctions membres

- Dès qu'une classe est déclarée, on peut ***instancier des objets*** (i.e. déclarer des variables):

```
Pile p;
```

- Dès qu'un objet est instancié, on peut invoquer ces fonctions membres publiques :

```
p. empiler (1.);
Pile * pp = &p;
pp->getSommet();  // via un pointeur
```

- **Attention : l'appel d'une fonction membre est toujours associé à un objet de la classe.**

```
p. empiler (1.); // OK, p est l'objet invoqué
getSommet();  // erreur, pas d'objet destinataire
              // Sauf si instance courante (this)
```

Une classe : implémentation (1/3)

- Pour qu'une classe soit complètement définie, il faut préciser le code de toutes les méthodes apparaissant dans sa déclaration

```
#include "pile.h"
#include <iostream>
#include <cassert>
using namespace std;

// les positions occupées sont ceux d'indice 0 à cpt (non compris)
// c'est à dire ceux dont l'indice i vérifie
// i >= 0 && i < cpt

// le sommet est à l'indice cpt -1
// il y en a cpt positions occupés

// invariant de classe
// cpt >=0 && cpt <= TAILLE
```

Une classe : implémentation (2/3)

```
Pile::Pile () {                              // constructeur vide
        cpt = 0;                             // this->cpt = 0;
        assert (estVide());                  // post-condition
}

Pile::Pile (double x) {                      // autre constructeur
        empiler(x);                          // this->empiler(x);
}

void Pile::empiler (double x) {
        assert ( !estPleine());              // pré-condition
        tab[cpt++] =x;
        assert (getSommet() == x);           // post-condition
}
```

Une classe : implémentation (3/3)

```
double Pile::depiler (){
        assert (! estVide());                    // pré-condition
        return tab[--cpt];
}
bool Pile::estPleine () const {
        return (cpt == TAILLE);
}
bool Pile::estVide () const {
        return (cpt == 0);
}
double Pile::getSommet() const {
        assert (! estVide());                    // pré-condition
        return tab[cpt -1];
}
```

Une classe : programme utilisateur

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

int main() {
        Pile p;
        p.empiler(4.5);
        std::cout << p.getSommet();
}
```

Surcharge - Principe et motivation

- Lorsque plusieurs fonctions effectuent la même tâche sur des objets de types différents, il est souhaitable de leur donner le même nom.

- C'est déjà le cas pour les opérateurs arithmétiques sur les entiers et les réels.

Exemple:

```
void print(int);            // affiche un entier
void print(const char *);   // affiche une chaîne
```

Surcharge : résolution des conflits de noms (1/2)

- Lorsqu'une fonction `f` est appelée, le compilateur cherche à déterminer quelle fonction de nom `f` est invoquée...Cette détermination est réalisée en comparant le **nombre** et le **type** des *paramètres effectifs* avec le nombre et le type des *paramètres formels*.

- Le principe est d'invoquer le fonction ayant la « meilleure » correspondance d'arguments…des conversions implicites peuvent intervenir !

- Exemples :

```
void print(long);
void print(float);

print(1L);          // invoque print(long)
print(1.0);         // invoque print(float)
print(1);           // erreur => 2 candidats
```

Surcharge : résolution des conflits de noms (2/2)

- Deux fonctions ne peuvent pas être distinguées uniquement par le type de la valeur retournée.

- La surcharge de fonction peut être employée pour réaliser l'effet d'un argument par défaut.

  - Exemple

```
void print(int value, int base){
...
}

void print(int value) {
      print(value, 10);
}
```

Valeurs par défaut

- Soit le prototype :

```
void f(int i, char c = 'a', int n = 10);
```

- Tous les appels suivants sont valides :

```
f(1, 'c', 2);
f(2, 'z');          // <=> f(2, 'z', 10);
f(3);               // <=> f(3, 'a', 10);
```

Surcharge : les méthodes des classes (1/2)

- Les méthodes d'une classe peuvent être surchargées.

- Exemple

```
class Nombre {
public:
      void initialise(const char *);
      void initialise(int i=0);
      ...
private:
      int chiffre(int) const;
      int& chiffre(int);
      int tab[100];
};
```

Surcharge : les méthodes des classes (2/2)

```
void Nombre::initialise(const char *s) {
    int l = strlen(s), i;
    for (i=0; i<l; ++i) tab[i] = s[l-i-1]-'0';
    for(; i<100; ++i) tab[i] = 0;
}
void Nombre::initialise(int i) {
    char buff[100];
    sprintf(buff, "%d", i);
    initialise(buff);
}

int Nombre::chiffre(int i) const {
    return tab[100-i];
}

int& Nombre::chiffre(int i)   {
    return tab[100-i];
}
```

Constructeur : motivation (1/2)

- Il est courant que l'utilisation d'un objet n'a de sens que si ce dernier est préalablement initialisé

```
// fichier nombre.h
class Nombre {
public:
     void initialise(const char *);
     void affiche() const;
     ...
private:
     int tab[100];
};

int main() {
     Nombre n;
     n.affiche();  // affiche n'importe quoi
}
```

Constructeur : motivation (2/2)

- Une bonne utilisation de la classe `Nombre` impose que tout objet de la classe soit **initialisé une et une seule fois** avant son utilisation.
- **Une solution : les constructeurs**

```
class Nombre {
public:
     Nombre(const char *); // cette méthode est un constructeur
     void affiche() const;
     ...
private:
     enum {MAX=100};
     int tab[MAX];
};
```

- Un constructeur est une fonction particulière invoquée ***implicitement*** et ***automatiquement*** lors de l' apparition en mémoire d' une instance de la classe

Constructeur : Définition

- Un constructeur est une méthode **portant le même nom** que la classe.
- Elle ne renvoie **pas de résultat.**
- Elle peut prendre des paramètres.
- Elle peut être surchargée.
- Elle **ne peut pas être invoquée explicitement.**
- Elle est **implicitement et automatiquement invoquée** lors de la déclaration de variables, de l'allocation dynamique de variable, etc…

Constructeur : exemple

```
class Nombre {
public:
     Nombre(const char* s);
     void affiche() const;
     ...
private:
     enum {MAX=100};
     int tab[MAX];
};
Nombre::Nombre(const char* s){
     int l = strlen(s), i;
     assert(l<MAX);
     for (i=0; i<l; ++i) {
          tab[i] = s[l-i-1]-'0';
          assert(tab[i]>=0 && tab[i]<=9);
     }
     for(; i<MAX; ++i)
          tab[i] = 0;
}
```

```
int main() {
     Nombre n1("12345678");
     Nombre n2;// erreur
     ...
}
```

Constructeur : le constructeur par copie

- **Constructeur par copie** (un seul paramètre de même type que l'instance courante).
  - Exemple :

```
Nombre::Nombre(const Nombre& n) {
    for(int i=0; i<MAX; ++i)
        tab[i] = n.tab[i];
}
```

- Par défaut, toute classe dispose d'un constructeur par copie.
- S'il n'est pas surchargé, le constructeur par copie réalise une copie champ à champ.
- Ce constructeur est invoqué pour l'initialisation des paramètres passés par valeur et des variables temporaires employées pour l'évaluation d'une expression

Constructeur par copie : exemple

```
void f(Nombre n) {
    ...
}
```

```
Nombre g() {
    return Nombre("0");
}
```

```
int main() {
  Nombre n1("12345678");
  Nombre n2(n1);     // construction par copie
  Nombre n3 = n1;   // construction par copie <=> Nombre n3(n1);

  f(n1);             // la paramètre n de la fonction f est
                     // initialisée par copie de n1

  n2 = g();          // une variable temporaire est construite
                     // pour stocker le résultat. Elle est
                     // initialisée par copie de la valeur
                     // retournée
  return 0;
}
```

Constructeur : le constructeur vide

- **Constructeur vide** (sans paramètre) :
  - ✓ Exemple

```
Nombre::Nombre() {
        for(int i=0; i<MAX; ++i)
                tab[i] = 0;
}
```

- Lorsqu'une classe n'a aucun constructeur, c'est le seul mode de construction autorisée (avec la construction par copie).

- On ne peut déclarer un tableau d'objet que si la classe dispose d'un constructeur vide. Il est appliqué à chaque objet du tableau.

Construction et allocation dynamique : rappels

- Les fonctions d'allocation mémoire.

  - L'allocation dynamique et la désallocation sont réalisées respectivement par les opérateurs **`new`** et **`delete`**.

  - L'opérateur `new` prend en paramètre un *nom de type* et rend un pointeur vers la zone mémoire allouée (renvoie `0` en cas d'échec).

  - L'opérateur `delete` prend en paramètre un pointeur retourné par `new` ou un pointeur nul.
    - Dans le premier cas, `delete` libère la zone mémoire,
    - dans le second, il est sans effet.

Construction et allocation dynamique : exemples

```
int *pi = new int;
*pi = 12;
delete pi;
```

- `new` et `delete` peuvent aussi être employés pour créer et détruire des tableaux:

```
char *s = new char[strlen(nom)+1];
strcpy(s, nom);
delete [] s; // [] indique que c'est un tableau
```

- **Attention** : il n'est pas conseillé d'utiliser les opérateurs `new` et `delete` et les fonctions `malloc` et `free` au sein d'un même programme.

Construction et allocation dynamique : les objets

- Un objet alloué dynamiquement peut être initialisé par un appel à un constructeur

```
int main() {
  Nombre* p;

  p = new Nombre("12345678");// allocation +
                             // initialisation
  p->affiche();  // appel de méthode via p
  delete p;
  return 0;
}
```

Initialisation objet membre d'une classe (1/2)

- Comment initialiser (invoquer le constructeur) d'un objet membre d'une classe (i.e. un objet emboîté) ?

```
class Compte{
public:
      Compte(     const string& n,
                  const Nombre& m,
                  const Nombre& d);
      ...
private:
      string nom;
      Nombre mont;
      Nombre dec;
};
```

Initialisation objet membre d'une classe (2/2)

- On l'indique dans le constructeur de la classe

```
Compte::Compte(const string& n, const Nombre& m,
               const Nombre& d): nom(n), mont(m), dec(d)
{
}
```

- Les constructeurs des objets membres sont invoqués avant le constructeur de l'objet.

Exemple : une pile d'entiers (1/4)

```
class Stack {
public:
   Stack(int cap);          // cap = capacité initiale
   void push(int i);        // empiler i
   bool empty() const;      // pile vide?
   int  top() const;        // sommet de la pile?
   void pop();              // dépiler

private:
   enum {DEFAULT_CAPACITY = 10, FACTOR = 2};
   int *stack;              // tableau dynamique
   int head;                // indice du sommet de la pile
   int capacity;            // capacité de la pile
};
```

Exemple : une pile d'entiers (2/4)

```
Stack::Stack(int cap) {
  assert(cap > 0);
  capacity = cap;
  stack = new int[capacity];
  head = 0;
}

void Stack::push(int i) {
  if (head == capacity) {
    capacity *= FACTOR;
    int *tmp = new int[capacity];
    for (int j = 0; j < head; ++j)
      tmp[j] = stack[j];
    delete [] stack;
    stack = tmp;
  }
  stack[head++] = i;
}
```

Allocation initiale

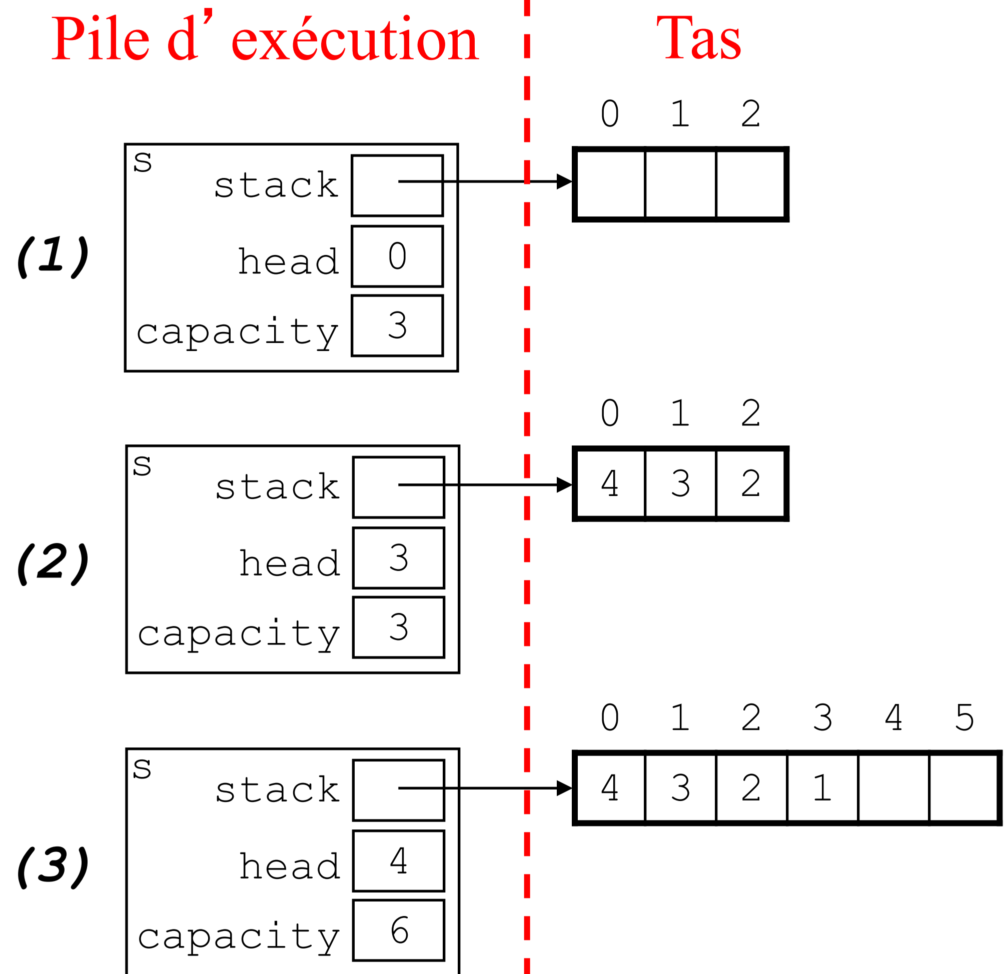
Lorsque c'est nécessaire, la capacité est doublée

Exemple : une pile d'entiers (3/4)

```
bool Stack::empty() const {
  return head == 0;
}

int Stack::top() const {
  assert(!empty());
  return stack[head-1];
}

void Stack::pop() {
  assert(!empty());
  --head;
}
```

Exemple : une pile d'entiers (4/4)

```
int main() {
  Stack s(3);
  // (1)

  for (int i=4; i>1; --i)
    s.push(i);
  // (2)

  s.push(1);
  // (3)
  return 0;
}
```



Affectation et construction par copie

- Pour toute classe, l'*affectation* entre instances de même type et la *construction par copie* d'instance sont toujours possibles

```
int main() {
  Stack s1(3);
  Stack s2(3);
  s2.push(1);

  s2 = s1;         // affectation

  Stack s3(s1);    // construction par copie
  Stack s4 = s1;   // construction par copie
  return 0;
}
```

- Par défaut, *la valeur de chaque champ est recopiée* d'une instance vers l'autre.

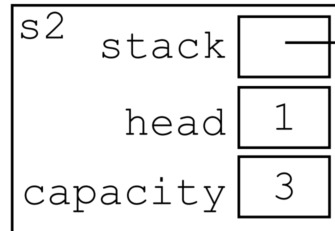
Problématique : affectation

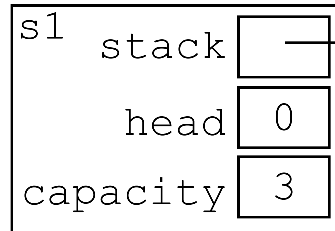
```
int main() {
  Stack s1(3);
  Stack s2(3);
  s2.push(1);
  // (1)
  s2 = s1;
  // (2)
  Stack s3(s1);
  // (page suiv.)
  return 0;
}
```

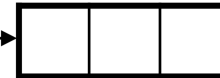
Pile d’exécution

Tas

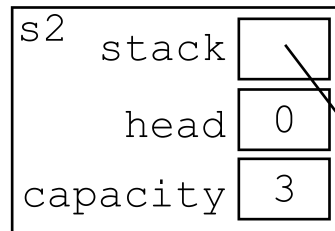
(1)

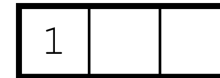




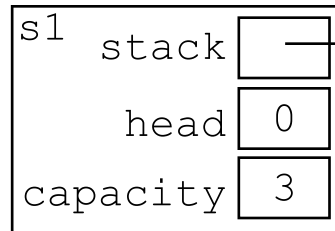


(2)





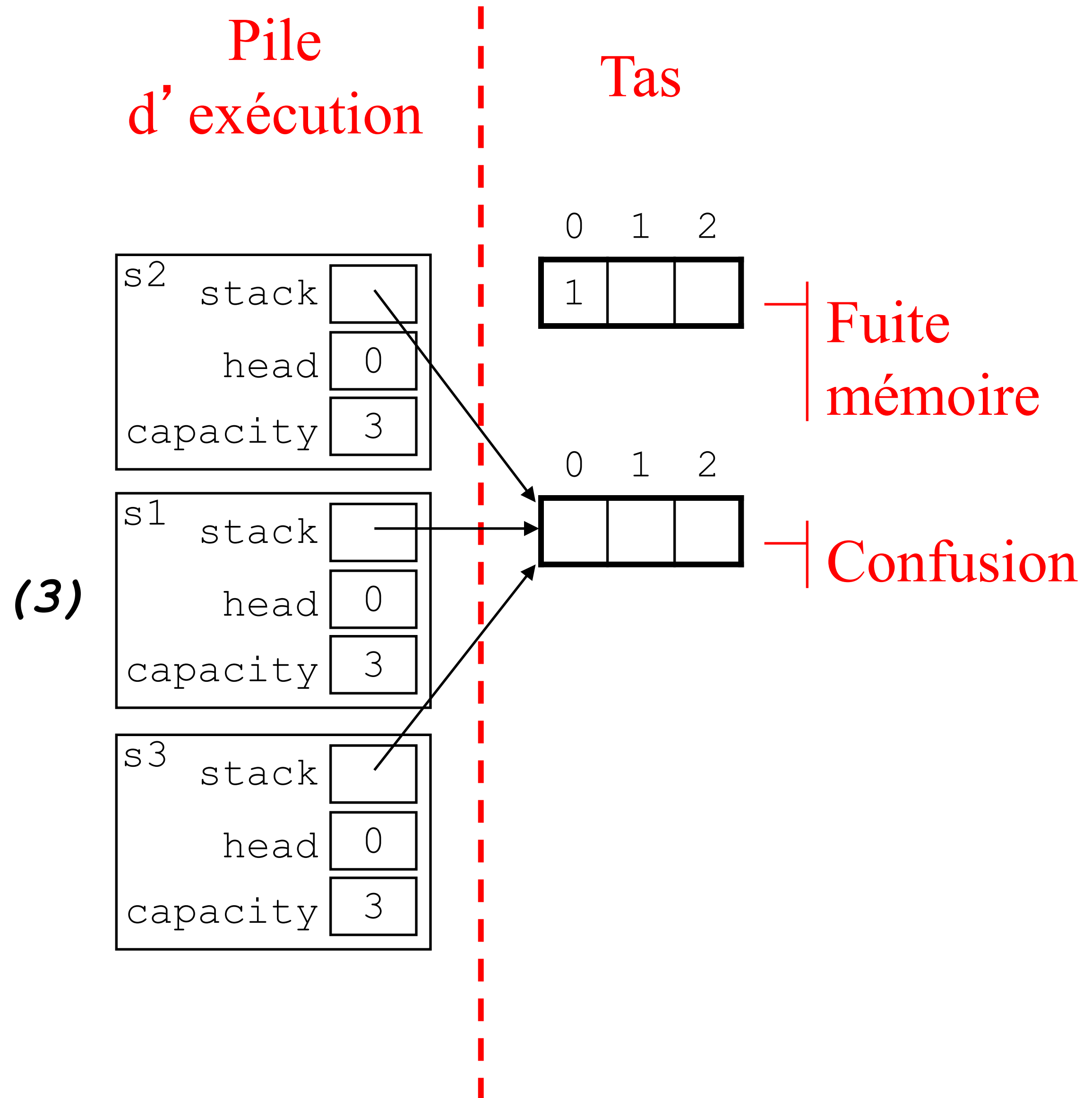
Fuite mémoire



Confusion

Problématique : construction par copie

```
int main() {
  Stack s1(3);
  Stack s2(3);
  s2.push(1);
  // (page préc.)
  s2 = s1;
  // (page préc.)
  Stack s3(s1);
  // (3)
  return 0;
}
```



Solution à la confusion

- Le constructeur par copie et l'opérateur d'affectation peuvent être définis de façon à corriger le comportement par défaut.

- Leurs prototypes sont les suivants:

```
class Stack {
public:
   // ...
   // construction par copie
   Stack(const Stack& s);
   // opérateur d'affectation
   Stack& operator=(const Stack& s);
private:
   // ...
};
```

Construction par copie

```
Stack(const Stack& s);
```

- C'est une construction.
- En conséquence, nous sommes assuré que l'objet qui doit être initialisé est une nouvelle variable du programme.
- Il est donc suffisant d'initialiser les champ de ce nouvel objet *en dupliquant les données dynamiquement allouées*

```
Stack::Stack(const Stack& s) {
  capacity = s.capacity;
  // duplication du tableau
  stack = new int[capacity];
  for (head = 0; head < s.head; ++head)
    stack[head] = s.stack[head];
}
```

Duplication du tableau

Opération d' affectation

```
Stack& operator=(const Stack& s);
```

- C' est une instruction.
- En conséquence, nous sommes assuré que l' objet qui doit être affecté est une variable du programme qui a été préalablement construite.
- Il est donc nécessaire de *désallouer les données créées dynamiquement* puis de ré-initialiser les champ *en dupliquant les données dynamiquement allouées*.
- Autres points à prendre en compte:
  - ✓ L'auto-affectation

    ```
    Stack s(3);
    s = s; // le détecter et ne rien faire
    ```

  - ✓ L'enchaînement d'affectation

    ```
    Stack s1(3), s2(3), s3(3);
    s1 = s2 = s3; // impose de renvoyer un résultat
    ```

Opération d’affectation (suite)

```
Stack& Stack::operator=(const Stack& s) {
if (this != &s) {
    delete [] stack;
    capacity = s.capacity;
    stack = new int[capacity];
    for (head = 0; head < s.head; ++head)
      stack[head] = s.stack[head];
  }
  return *this;
}
```

- `(this != &s)` : Est-ce une auto-affectation?
- `delete [] stack;` : Désallocation du tableau
- `stack = new int[capacity];` … `for` : Duplication du tableau
- `return *this;` : Permet les enchaînements d’affectations

Solution aux fuites mémoire

- Chaque fois qu'une instance est supprimée, il faut désallouer ce qui a été créé dynamiquement.
- C'est le rôle des **destructeurs**.
- Il sont appelés implicitement dès qu'une instance est supprimée, c'est-à-dire :
  - ✓ Lorsqu'une fonction se termine, les variables locales sont supprimées
  - ✓ Lorsqu'une instance créée dynamiquement est désallouée.

- Son prototype est le suivant:

```
class Stack {
public:
  // ...
  // destructeur
  ~Stack();
private:
  // ...
};
```

Destruction

```
~Stack();
```

Il est suffisant de *désallouer tout ce qui a été créé dynamiquement*

```
Stack::~Stack() {
    delete [] stack;
}
```

Désallocation du tableau

Forme canonique d'une classe

- Toute classe faisant de l'allocation dynamique de mémoire « doit » comporter :
  - ✓ Un constructeur par copie
  - ✓ Un opérateur d'affectation
  - ✓ Un destructeur

- Toute classe doit comporter : un constructeur vide

- Le constructeur vide est employé pour initialiser les tableaux d'objets.

Une classe respectant ces règles est dites "sous forme canonique"

Gestion des objets

- Les opérateurs de « move » permettent de transférer le contenu d'un temporaire (rvalue). Le temporaire va mourir tout de suite, on lui prend sa mémoire et on le vide.

Member function	typical form for class C:
Default constructor	`C::C();`
Destructor	`C::~C();`
Copy constructor	`C::C (const C&);`
Copy assignment	`C& operator= (const C&);`
Move constructor	`C::C (C&&);`
Move assignment	`C& operator= (C&&);`

Member function	implicitly defined:	default definition:
Default constructor	if no other constructors	does nothing
Destructor	if no destructor	does nothing
Copy constructor	if no move constructor and no move assignment	copies all members
Copy assignment	if no move constructor and no move assignment	copies all members
Move constructor	if no destructor, no copy constructor and no copy nor move assignment	moves all members
Move assignment	if no destructor, no copy constructor and no copy nor move assignment	moves all members

L’amitié

Toute fonction peut être déclarée «amie» d’une (ou plusieurs) classe(s)

Une fonction « amie » d’une classe peut accéder directement (sans passer par des méthodes) aux éléments privés de la classe

Exemple

```
class Personne {
public:
   ...
   friend void afficher(const Personne& p);
private:
     char nom[20];
     int age;
};

void afficher(const Personne& p) {
     cout << p.nom << p.age << endl;
}
```

La fonction `afficher` est «amie» de la classe `Personne`

C’est une fonction et non pas une méthode

Elle accède aux données privées de la classe

Une classe peut aussi être déclarée «amie» d'une autre classe

Dans ce cas, toutes les méthodes de la classe « amie » peuvent accéder directement aux éléments privés de la classe

Exemple

```
class Personne {
public:
    ...
  friend class Collect;
private:
  char nom[20];
  int age;
};
```

```
class Collect {
public:
    void AfficherNoms() const;
    ...
private:
    Personne tab[10];
    int nb;
};

void Collect::AfficherNoms() const {
    for (int i=0; i<nb; i++)
        cout << tab[i].nom << endl;
}
```

Il ne faut pas abuser de l'usage de `friend` ⇒ une fonction ou une classe ne doit être déclarée amie *<u>que dans des cas extrêmes</u>*

Exemple:

```
class Personne {
public:
    ...
  friend class Element;
  friend class Liste;
private:
  char nom[20];
  int age;
};

class Element {
  friend class Liste;
private:
  Personne p;
  Element *suivant;
};
```

```
class Liste {
public:
  Liste();
  void Ajouter(const Personne& p);
  ...
private:
  Element *debut;
};
```

Aucun membre de la classe `Element` n'est public excepté pour la classe `Liste` ⇒ seule cette classe peut employer des objets de la classe `Element`

Les opérateurs

Le C++ autorise la surcharge des opérateurs

L'objectif est de permettre au programmeur de fournir une notation plus conventionnelle et pratique que la notation fonctionnelle de base (par exemple, pour la manipulation d'objets arithmétiques complexes)

Il existe 2 manières de surcharger un opérateur

- ✓ Un opérateur peut être surchargé *par une fonction*. Dans ce cas au moins une opérande doit être de type «classe».
- ✓ Un opérateur peut être surchargé *par une méthode* d'une classe. Dans ce cas, la première opérande est l'objet pour laquelle la méthode est invoquée.

Surcharge par une fonction

```
class Vecteur3d {
public:
   Vecteur3d(int x=0, int y=0, int z=0);
   int get(int i) const;
private:
   int x, y, z;
};

Vecteur3d operator+(const Vecteur3d& v1,
                                        const Vecteur3d& v2);
```

```
// constructeur
Vecteur3d::Vecteur3d(int a, int b, int c) {
    x = a; y = b; z = c;
}

// récupération d'une valeur
int Vecteur3d::get(int i) const {
    assert((i>=1) && (i<=3));

    switch (i) {
    case 1: return x;
    case 2: return y;
    case 3: return z;
    }
}
```

```
// Addition de 2 Vecteurs
Vecteur3d operator+(const Vecteur3d& v1,
                    const Vecteur3d& v2) {
   int i, j, k;

   i = v1.get(1) + v2.get(1);
   j = v1.get(2) + v2.get(2);
   k = v1.get(3) + v2.get(3);
   return Vecteur3d(i,j,k);
}
```

- ♦ **Question : que faire pour que la fonction puisse accéder directement aux données de la classe `Vecteur3d` ?**

La fonction (l'opérateur) peut être invoquée de 2 façons

```
// utilisation de la classe Vecteur
int main() {
   Vecteur3d a(1,2,3), b(3,2,1), c, d;

   c = operator+(a, b);     // notation fonctionnelle
   d = a + b;               // notation usuelle

   for (int i=1; i<=3; i++) cout << c.get(i) << " " ;
   cout << endl;
   for (int i=1; i<=3; i++) cout << d.get(i) << " " ;
   cout << endl;
   return 0;
}
```

Surcharge par une méthode

```
class Vecteur3d {
public:
    Vecteur3d(int x=0, int y=0, int z=0);
    int get(int i) const;
    int operator*(const Vecteur3d& v) const;
private:
    int x, y, z;
};
```

Opérateur binaire:

- l'opérande gauche est l'instance courante
- l'opérande droite est le paramètre

```
Vecteur3d operator+(const Vecteur3d& v1, const Vecteur3d& v2);

// multiplication de 2 Vecteurs
int Vecteur3d::operator*(const Vecteur3d& v) const {
    int res;

    res = (this->x * v.x) + (y * v.y) + (z * v.z);

    return res;
}
```

La méthode (l'opérateur) peut être invoquée de 2 façons

```
// utilisation de la classe Vecteur
int main() {
    Vecteur3d a(1,2,3), b(3,2,1);
    int c, d;

    c = a.operator*(b);     // notation fonctionnelle
    d = a * b;              // notation usuelle

    cout << c << endl;
    cout << d << endl;
    return 0;
}
```

Généralités

De nombreux opérateurs peuvent être surchargés

On doit conserver leur « pluralité » (i.e. nombre d’opérandes).

Les opérateurs redéfinis gardent leur priorité et leur associativité (ordre d’évaluation)

Aucune hypothèse n’est faites sur la signification a priori d’un opérateur. Par exemple, la signification de += pour une classe ne peut être déduite automatiquement de la signification de + et de = pour cette même classe.

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

Opérateurs arithmétiques et relationnels

```
class Vecteur3d {
public:
   Vecteur3d(int x=0, int y=0, int z=0);
   int get(int i) const;
   Vecteur3d operator+(const Vecteur3d& v) const;
   int operator*(const Vecteur3d& v) const;

   bool operator==(const Vecteur3d& v) const; // comparaison
   Vecteur3d operator*(int i) const; // produit
private:
   int x, y, z;
};

Vecteur3d operator*(int i, const Vecteur3d& v);
```

Attention : cette fonction ne peut pas être transformée en une méthode

l 'opérande gauche n'est pas une instance de la classe

```
// comparaison (c'est une méthode)
bool Vecteur3d::operator==(const Vecteur3d& v) const {
   return ((x == v.x) && (y == v.y) && (z == v.z));
}

// produit vecteur-entier (c'est une méthode)
Vecteur3d Vecteur3d::operator*(int i) const {
   Vecteur3d v(x*i, y*i, z*i);
   return v;
}

// produit entier-vecteur (c'est une fonction)
Vecteur3d operator*(int i, const Vecteur3d& v) {
   return v * i;
}
```

Opérateurs d'entrée/sortie

On veut pouvoir utiliser les opérateurs << et >> pour afficher et saisir des vecteurs

Leurs prototypes sont

```
ostream& operator<<(ostream& os, const Vecteur3d& v);
istream& operator>>(istream& is, Vecteur3d& v);
```

`ostream` est le type de `cout` et `istream` celui de `cin`

Ces opérateurs ne peuvent être définis comme étant des méthodes de la classe `Vecteur3d`. En effet, l'opérande gauche appartient à la classe `ostream` et `istream`. Il faudrait donc les définir comme étant méthodes de ces classes (ce qui est impossible).

On doit donc les définir comme étant des fonctions (amies ou non) de la classe `Vecteur3d`

```
class Vecteur {
public:
  Vecteur3d(int x=0, int y=0, int y=0);
  int get(int i) const;
  Vecteur3d operator+(const Vecteur3d& v) const;
  int operator*(const Vecteur3d& v) const;
  int operator==(const Vecteur3d& v) const;
  Vecteur3d operator*(int i) const;
  friend Vecteur3d operator*(int i, const Vecteur3d& v);

  // entrée (saisie) -> fonction amie
  friend istream& operator>>(istream& is, Vecteur3d& v);
private:
  int x, y, z;
};

// sortie (affichage) -> fonction
ostream& operator<<(ostream& os, const Vecteur3d& v);
```

```
// saisie
istream& operator>>(istream& is, Vecteur3d& v)
{
  is >> v.x >> v.y >> v.z;
  return is;
}

// affichage
ostream& operator<<(ostream& os, const Vecteur3d& v)
{
  for (int i=1; i<=3; i++)
      os << v.get(i) << ' ';
  return os;
}
```

// Utilisation de la classe Vecteur3d

```
void main() {
   Vecteur3d a(5, 10, 15);
   Vecteur3d b;

   cin >> a;                    // saisie

   b = a*2;               // produit et affectation

   if (b==(2*a))          // produit et comparaison
      cout << "c'est correct" << endl;
   else  {                // affichage
      cout << "probleme" << endl;
      cout << a*2 << endl << 2*a << endl;
   }
}
```

Opérateur d'incrémentation et de décrémentation

- Les opérateurs d'incrémentation (`++`) et de décrémentation (`--`) peuvent être utilisés de manière préfixée (`++x`) ou postfixée (`x++`)
- La forme postfixée est distinguée de la forme préfixée par l'emploi d'un argument (de type `int`) supplémentaire non-utilisé

```
class Vecteur3d {
public:
    ...
    Vecteur3d operator++();             // préfixée
    Vecteur3d operator++(int);          // postfixée
private:
    ...
};
```

```
// incrémentation préfixée
Vecteur3d Vecteur3d::operator++() {
   x++;
   y++;
   z++;
   return *this; // renvoi de la valeur après incrément.
}
```

ne pas donner de nom au paramètre pour éviter un warning

```
// incrémentation postfixée
Vecteur3d Vecteur3d::operator++(int) {
  Vecteur3d res(*this); // construction par copie
   ++(*this); // incrémentation de l'instance
   return res; // renvoi de la valeur avant incrément.
}
```

Autres Opérateurs

Beaucoup d'opérateurs peuvent être surchargés

A titre d'exemple, voici un opérateur d'indexation pour la classe `Vecteur3d`

```
class Vecteur3d {
public:
        ...
        int& operator[](int);
        int operator[](int) const;
private:
        ...
};
```

```
// adressage indexe
int& Vecteur3d::operator[](int i) {
   assert((i>=1) && (i<=3));
   return (i==1 ? x : (i==2 ? y : z));
}
int Vecteur3d::operator[](int i) const {
   assert((i>=1) && (i<=3));
   return (i==1 ? x : (i==2 ? y : z));
}
// Utilisation
int main() {
   Vecteur3d a;

   for (int i=1; i<=3; i++) a[i] = i; // écriture
   for (int i=1; i<=3; i++)
        cout << a[i] << endl; // lecture
   return 0;
}
```

Conclusion: un vecteur peut être vu comme un tableau