Programmation Système Répartie et Concurrente
Master 1 Informatique – 4I400

Cours 2 : C++ classes, opérateurs, gestion mémoire

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

Plan

On a vu au cours précédent

- La syntaxe de base, les variables, les références

Aujourd'hui :

- Les classes, attributs, fonctions membres (méthodes), opérateurs
- Gestion de la mémoire dynamique, new/delete
- Templates, auto

Références : cppreference.com , cplusplus.com

Cours de Denis Poitrenaud, IUT Paris Descartes

Page ue :

https://pages.lip6.fr/Yann.Thierry-Mieg/PR/

Surcharge, Polymorphisme, Résolution de fonctions

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);
        //  ou en one-liner c++
        //  initialise(std::to_string(i).c_str())
}
// NB : std::stringstream permet d'utiliser operator<< en mémoire
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'aie 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/par défaut

- **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.
- Un constructeur à un seul argument définit **implicitement** un opérateur de conversion
  - f(« 314 ») => recherche d'un constructeur de Nombre à partir de const char*

Promotion de types

- Toute constructeur a un seul argument => définit implicitement une opération de conversion
  - Constructeur : Nombre(const char *),
  - Fonction : void foo(const Nombre &n)
  - L'utilisation de : foo(« 123 ») => construction d'un Nombre
  - Le mot clé « explicit » à la déclaration du constructeur évite ce comportement
- La promotion n'intervient que si une signature plus adaptée n'est pas trouvée

Allocation dynamique

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 **nullptr**.
  - Dans le premier cas, `delete` libère la zone mémoire,
  - dans le second, il est sans effet.
- Ces appels système gèrent la mise en service de segments de mémoire adressables par le processus en lecture/écriture
  - La mémoire qui n'est pas libérée => fuite mémoire consommant des ressources du système
  - Double delete => faute vis-à-vis du contrat avec le système

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.
- L'ordre de l'initialisation suit celui de la **déclaration** des attributs
- On peut aussi invoquer :
  - Un constructeur de super classe (si héritage)
  - Un autre constructeur de la classe (déléguer)

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

```
namespace pr {

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:
  static const int DEFAULT_CAPACITY = 10, FACTOR = 2;
  int *stack;              // tableau dynamique
  int head;                // indice du sommet de la pile
  int capacity;            // capacité de la pile
};
} // fin namespace
```

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

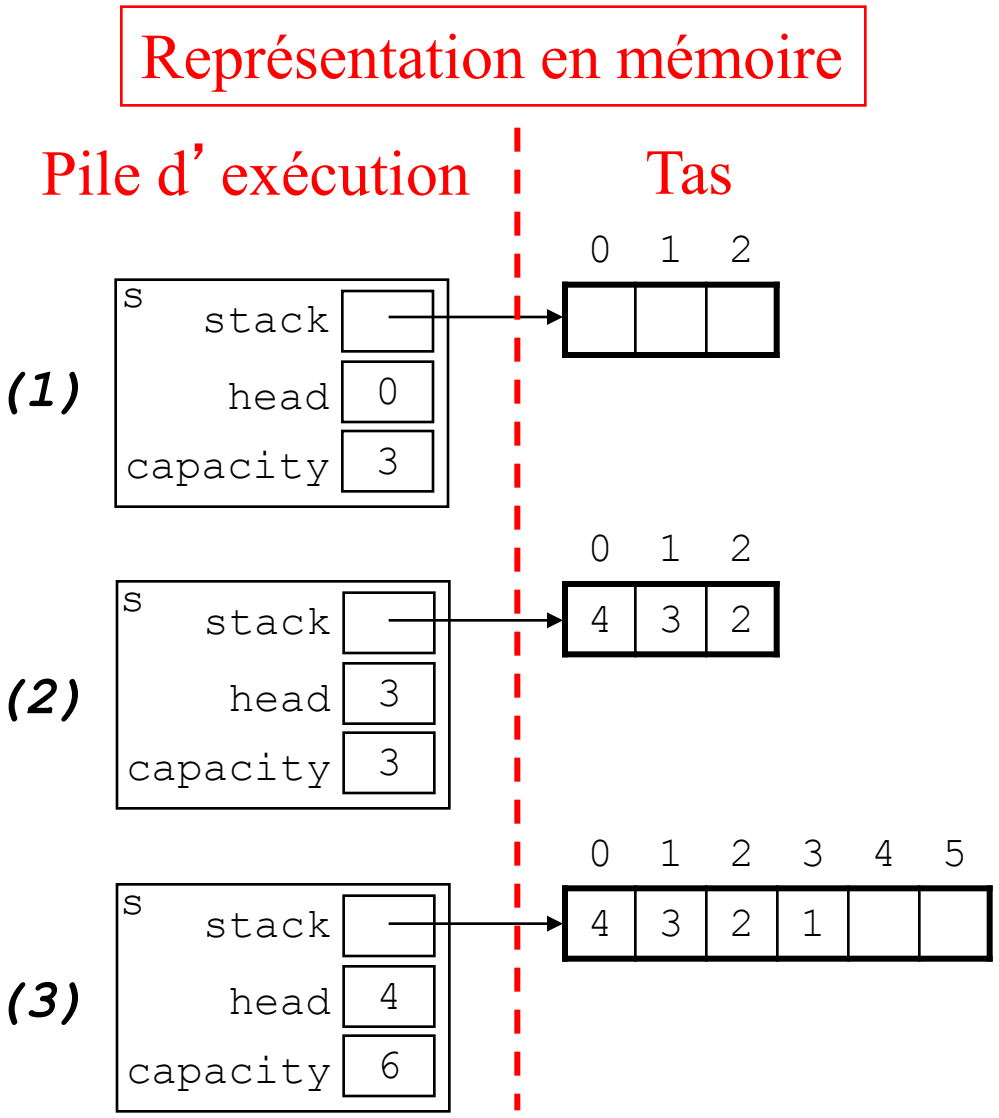
Lorsque c'est nécessaire, la capacité est doublée

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

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

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

void Stack::pop() {
  --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;
}
```



Copie, Sémantique Affectation

Affectation et construction par copie

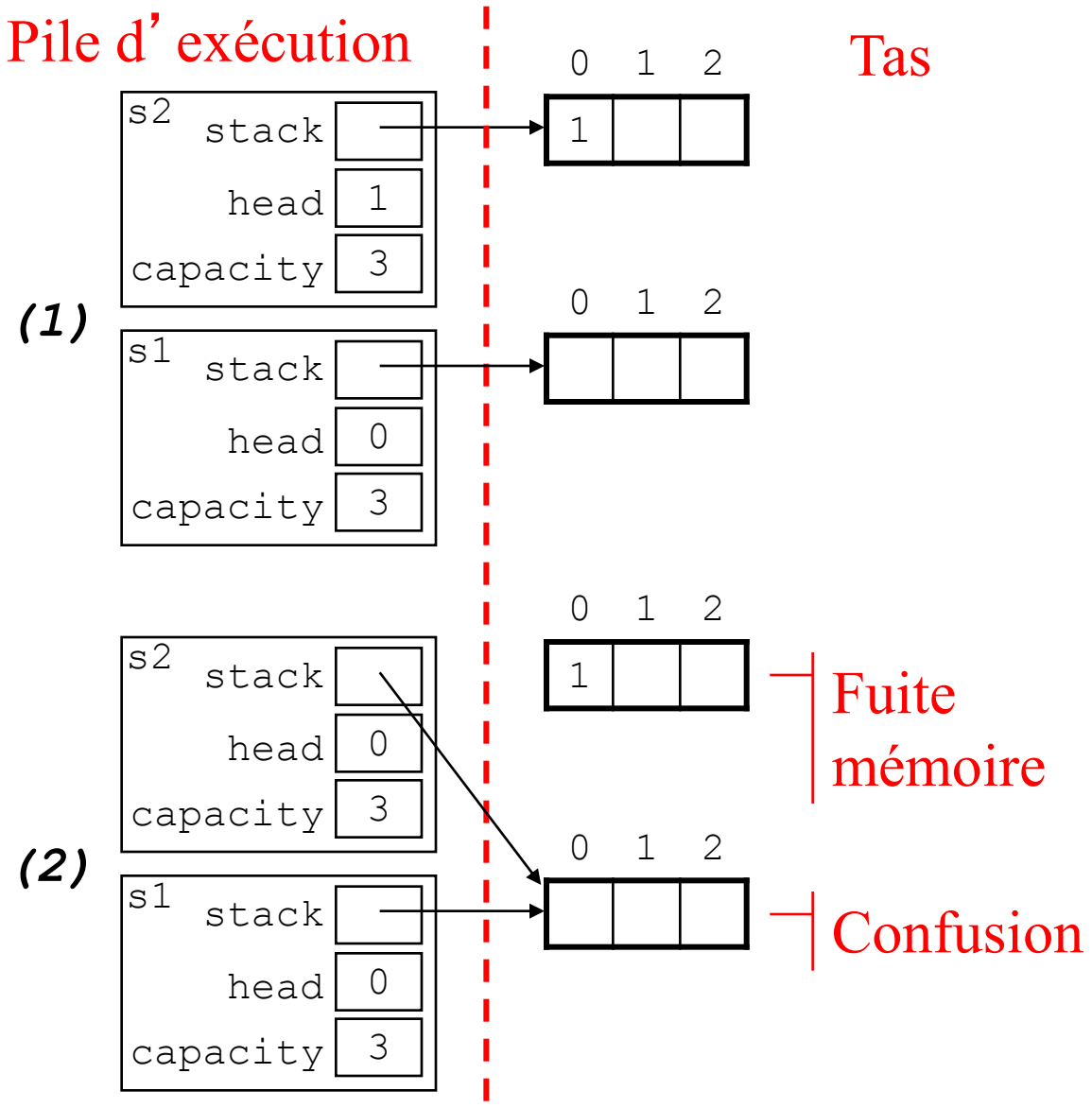
- Pour toute classe, l'*affectation* entre instances de même type et la *construction par copie* d'instance sont toujours possibles

```
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.

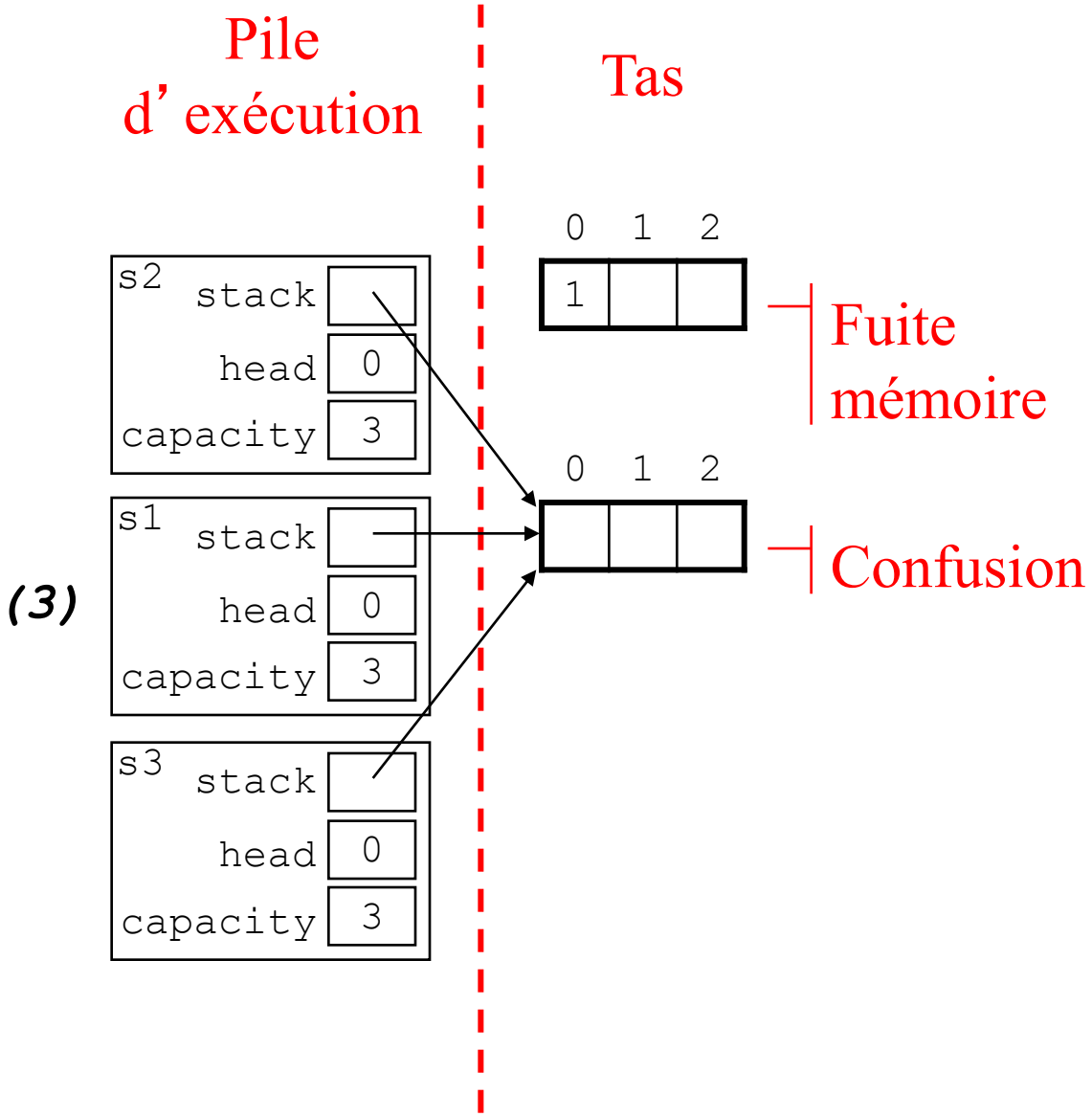
Problématique : affectation

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



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;
}
```

Est-ce une auto-affectation?

Désallocation du tableau

Duplication du tableau

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

Opérateurs sur classes

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` ?**

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

.h

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

Déclaration d'amitié : `afficher` est «amie» de la classe `Personne`

C'est une fonction et non pas une méthode

Déclaration de afficher

.cpp

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

Elle accède aux données privées de la classe

Exemple : surcharge de operator<<

```
#pragma once
#include <iosfwd>
class Point
{
        double x;
        double y;
public :
        Point(double x, double y);
        double distOri();
        friend std::ostream & operator<<(std::ostream &os, const
Point &p);
};
std::ostream & operator<<(std::ostream &os, const Point &p);
```

```
#include <iostream>

std::ostream & operator<<(std::ostream & os, const Point & p)
{
        os << p.x << "," << p.y << std::endl;
        return os;
}
```

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;
};

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;
}
```

Opérateur binaire:
- l'opérande gauche est l'instance courante
- l'opérande droite est le paramètre

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) -> déclaration d'amitié fonction
  friend istream& operator>>(istream& is, Vecteur3d& v);
private:
  int x, y, z;
};
// entrée (saisie) -> déclaration de fonction
istream& operator>>(istream& is, Vecteur3d& v);
// sortie (affichage) -> déclaration de 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

Généricité, Auto

Fonctions Templates

- Deux fonctions surchargées (même nom) qui diffèrent par le typage
  - Même corps/code
- Version générique qui évite la duplication ?

```
int sum (int a, int b)
{   return a+b; }

double sum (double a, double b)
{   return a+b; }

int main ()
{
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

Fonctions Templates (2)

```
template<typename T>
T sum (T a, T b)
{   return a+b;  }
```

- On préfixe la déclaration par la déclaration des paramètres génériques
  - Introduit ici que T est un type
- On peut utiliser T là ou un nom de type ou de classe est attendu (signatures…)
- Le code n'est pas compilé tel quel, il doit être visible du client (e.g. dans un .h)
- Le code est instancié à l'invocation
  - sum (10,20) **-> instantiation T = int**
  - sum (1.0,2.0) **-> instantiation T = float**
  - sum("a","b") **-> erreur + pour const char ***
  - sum<string> ("a","b") **-> T = string (forcé)**
- On substitue à T un type effectif et on compile le code => nécessite une variante de : T operator+(T a,T b)

Classes Template

- Classe dont le type des attributs et/ou la signature des méthodes est générique
- Le corps des déclarations doit être présent dans le .h
- Un ou plusieurs argument génériques possibles

```
template <typename T, typename U>
class pair {
  public:
    T first; U second;
    mypair (T first, U second)
                    : first(first),second(second) {}
          // ... operateurs, accesseurs
};
```

```
template <typename T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {       values[0]=first;
            values[1]=second;  }
};
```

Templates : concepts avancés

- Utilisation d'un type générique dans une signature de template

```
template < template <typename> class Cont, typename T >
const T & findFirst (const Cont<T> & c) { return c[0]; }
```

- Spécialisation partielle de template
  - vector<bool> => implémentation BitSet
- Paramètres génériques par défaut
  - E.g. Allocateur par défaut sur les conteneurs standard
- D'autres usages sont possibles en méta programmation
  - S'appuie sur l'instanciation de génériques pour générer du code
- Beaucoup de bibliothèques C++ sont très génériques
  - Cf Boost, presque entièrement header purs.
- Mon conseil personnel :
  - Beaucoup des arguments pour les codes génériques relèvent de l'optimisation. A EVITER DANS UN PREMIER TEMPS

auto : inférence de type

- C++11 apporte une notion nouvelle d'inférence de type
  - Mot clé : auto
- Permet de typer :
  - Les variables dont le type est déduit de l'initialisation
  - Les paramètres de fonction anonymes (lambdas)
  - Le type de retour d'une fonction (déduit du typage de return)
- Très confortable, surtout pour les types dérivés ou à rallonge, et dans le code des fonctions génériques
  - Des exemples dans la suite du cours
- Les règles régissant « auto » sont en évolution (e.g. plus d'inférences possibles en C++17)

auto : exemples

- Attention différences entre
  - auto et auto &
- L'inférence fonctionne aussi avec les classes
- Remplace des expressions classiques

```
auto d = 5.0; // double
auto i = 1 + 2; // int

int add(int x, int y)
{ return x + y; }

int main()
{   auto sum = add(5, 6); // int
    return 0;
}
```

```
std::vector<int>::const_iterator it = vec.end(); //AVANT
auto it = vec.end();  // C++11
```

Conteneurs associatifs : Maps, tables de hash

Les conteneurs associatifs

- Matérialisent une table associative <map> <unordered_map>
- Soit map<string, int> agenda. On peut voir agenda comme :
  - ✓ Un ensemble de paires clé/valeur :
    - { <toto;0601>, <tata;0602>}
  - ✓ Une fonction; i.e. une application de string dans int telle que :
    - agenda(toto)=0601 et agenda(tata)=0602
  - ✓ Un tableau indicé sur des string :
    - agenda[« toto »] = 0601, agenda[« tata »] = 0602
- Propriété de clé : à une clé donnée ne correspond qu'une valeur
- Bonnes complexités du test d'existence d'une clé, recherche et ajouts.

Table de Hash : principes

- On *hash* la clé => size_t, on cherche modulo le nombre de *buckets* dans la liste à cet indice
- Si le hash est rapide et « bon », O(1) pour trouver une entrée (paire)