université de TOURS

Développement objet C++

Dominique H. Li

dominique.li@univ-tours.fr

Licence Informatique - Blois

Université de Tours

université de TOURS

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World !" << std::endl;
    return 0;
}
```

6. Polymorphisme

Développement objet C++

Le polymorphisme

- Le *polymorphisme* est la capacité pour une entité de prendre plusieurs formes
- En programmation objet, le polymorphisme caractérise une entité qui fait référence au moment de l'exécution à des occurrences de différentes classes
- Deux types de polymorphisme
  - Polymorphisme par sous-typage (surcharge/redéfinition)
  - Polymorphisme paramétrique (modèle, ou dit template)

Surcharge des fonctions

- En C++, lorsque plusieurs fonctions effectuent un même traitement mais sur des paramètres de type différents ou sur un nombre différent de paramètres, il est possible de donner le même nom à ces fonctions
- Une utilisation multiple d'un nom est appelée *surcharge du nom de fonction*
- La surcharge peut simplifier l'accès aux membres données

Exemple 6.1 Surcharge du nom de fonction

```
int Date::jour() {
        return j_;
}

void Date::jour(int j) {
        j_ = j;
}
```

Surcharge des opérateurs

- C++ permet la *surcharge des opérateurs*, la possibilité de définir plusieurs fonctions différentes portant le même nom
- Plusieurs opérateurs C++ sont déjà surchargés
  - L'opérateur d'indirection * et l'opérateur d'adressage **&**
  - L'opérateur d'entrée et l'opérateur de sortie **<<** et **>>**
  - L'opérateur d'addition **+** (**++**)

Exemple 6.2 Surcharge des opérateurs

```
Date d(2012, 12, 21);          // d = "21/12/2012";
d.impr();                      // std::cout << d << std::endl;
d.incr();                      // ++d ou d++
d.decr();                      // --d ou d--
```

Exemple 6.3 Surcharge de l'opérateur = (.h)

```
class Date {
public:
        // Les membres publics
        Date &operator=(const char *);
private:
        // Les membres privés
};
```

Exemple 6.3 Surcharge de l'opérateur = (.cpp)

```
Date &Date::operator=(const char *s) {
        // "21/12/2012"
        // j_ <- 21
        // m_ <- 12
        // a_ <- 2012
        return *this;
}
```

Exemple 6.4 Surcharge de l'opérateur << (.h)

```
class Date {
        friend ostream &operator<<(ostream &, const Date &);
public:
        // Les membres publics
private:
        // Les membres privés
};
```

Exemple 6.4 Surcharge de l'opérateur << (.cpp)

```
ostream &operator<<(ostream &out, const Date &d) {
        out << d.j_ << "/" << d.m_ << "/" << d.a_;
        return out;
}
```

Exemple 6.5 Surcharge de l'opérateur >> (.h)

```
class Date {
        friend istream &operator>>(istream &, Date &);
public:
        // Les membres publics
private:
        // Les membres privés
};
```

Exemple 6.5 Surcharge de l'opérateur >> (.cpp)

```
istream &operator>>(istream &in, Date &d) {
        // TP
        return in;
}
```

Implémentation des opérateurs ++ et --

- Deux comportements différents
  - **++x** (ou **--x**) modifie l'objet avant de l'utiliser
  - **x++** (ou **x--**) modifie l'objet après de l'utiliser
- Implémentés par deux fonctions surchargées
  - **T &operator++()** (ou **--**) retourne une référence de l'objet modifié
  - **T operator++(int)** (ou **--**) crée une copie de l'objet, puis modifie l'objet, enfin retourne la copie de l'objet

Exemple 6.6 Surcharge des opérateurs ++ et --

```
class Date {
public:
        // Les membres publics
        Date &operator++();             // ++d
        Date operator++(int);           // d++
        Date &operator--();             // --d
        Date operator--(int);           // d—
private:
        // Les membres privés
};
```

Polymorphisme paramétrique

- La *généricité*, ou le *polymorphisme paramétrique*, consiste en la possibilité de paramétrer des entités de programme, en générale par des types
- Le *type (la classe) paramètre* est un type normal, sauf qu'il est paramétré par un ou plusieurs paramètres formels qui peuvent être utilisés comme des types dans sa définition
- Il est possible de paramétrer des types, objets ou fonctions

Exemple 6.7 Type paramétré

```
// std::vector<T> : le type paramétré
// T : le type formel
std::vector<int> v1;
std::vector<std::string> v2;
std::vector<Date> v3;
std::vector<Date *> v4;
```

Les C++ Templates

- Un *modèle de classe/fonction* ou *classe/fonction générique* est une classe/fonction qui sert de template, c'est-à-dire de modèle pour d'autres types
- Le mot réservé **template** commence la déclaration d'une classe générique ou d'une fonction générique
- Les méthodes de la classe générique doivent être définies en ligne ou en dehors de la classe, mais pas dans un fichier compilé séparément

Exemple 6.8 Tableau dynamique générique

```
template<class T>
class Array {
public:
        Array(size_t);
        ~Array() { delete[] data_; }
        T &operator[](size_t i) { return data_[i]; }
        const T &operator[](size_t i) const { return data_[i]; }
        size_t size() const { return size_; }
private:
        T *data_;
        size_t size_;
};

Array::Array(size_t n): size_(n) { data_ = new T[n]; }
```

Exemple 6.9 Tableaux dynamique de nombres

```
Array<int> nombres(10);
for (size_t i = 0; i < 10; i++) {
        nombres[i] = i + 1;
}
for (size_t i = 0; i < nombres.size(); i++) {
        std::cout << nombres[i] << std::endl;
}
```

Exemple 6.10 Tableau dynamique de dates

```
Array<Date *> dates(10);
for (size_t i = 0; i < dates.size(); i++) {
        Date *d = new Date();
        std::cin >> *d;
        dates[i] = d;
}
for (size_t i = 0; i < dates.size(); i++) {
        std::cout << *(dates[i]) << std::endl;
        delete dates[i];
}
```

Exemple 6.11 Un algorithme de tri sur les entiers

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

Exemple 6.11 Un algorithme de tri sur les entiers

```
void sort(int x[], size_t n) {
        for (size_t i = 0; i < n; i++)
                for (size_t j = i + 1; j < n; j++)
                        if (x[i] > x[j])
                                swap(x[i], x[j]);
}
```

Exemple 6.11 Un algorithme de tri sur les entiers

```
int main() {
        int y[] = {1, 2, 5, 4, 3};
        sort(y, 5);
        double z[] = {1.1, 2.2, 5.5, 4.4, 3.3};
        sort(z, 5);                    // Erreur : type incompatible
}
```

Exemple 6.12 Un algorithme de tri sur les réels

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

Exemple 6.12 Un algorithme de tri sur les réels

```
void sort(double x[], size_t n) {
        for (size_t i = 0; i < n; i++)
                for (size_t j = i + 1; j < n; j++)
                        if (x[i] > x[j])
                                swap(x[i], x[j]);
}
```

Exemple 6.12 Un algorithme de tri sur les réels

```
int main() {
        int y[] = {1, 2, 5, 4, 3};
        sort(y, 5);
        double z[] = {1.1, 2.2, 5.5, 4.4, 3.3};
        sort(z, 5);                    // OK
}
```

Exemple 6.13 Un algorithme de tri sur les dates

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

Exemple 6.13 Un algorithme de tri sur les dates

```
void sort(Date x[], size_t n) {
        for (size_t i = 0; i < n; i++)
                for (size_t j = i + 1; j < n; j++)
                        if (x[i] > x[j])
                                swap(x[i], x[j]);
}
```

Exemple 6.13 Un algorithme de tri sur les dates

```
int main() {
        Date d[5];
        for (size_t i = 0; i < 5; i++) {
                std::cin >> d[i];
        }
        sort(d, 5);
}
```

Exemple 6.14 Un algorithme de tri générique

```
template<class T>
void swap(T &a, T &b) {
        T tmp = a;
        a = b;
        b = tmp;
}
```

Exemple 6.14 Un algorithme de tri générique

```
template<class T>
void sort(T x[], size_t n) {
        for (size_t i = 0; i < n; i++)
                for (size_t j = i + 1; j < n; j++)
                        if (x[i] > x[j])
                                swap<T>(x[i], x[j]);
}
```

Exemple 6.14 Un algorithme de tri générique

```
int y[] = {1, 2, 5, 4, 3};
sort<int>(y, 5);

double z[] = {1.1, 2.2, 5.5, 4.4, 3.3};
sort<double>(z, 5);
```

Exemple 6.14 Un algorithme de tri générique

```
Date d[5];
for (size_t i = 0; i < 5; i++) {
        std::cin >> d[i];
}
sort<Date>(d, 5);
```

Exemple 6.14 Un algorithme de tri générique

```
std::string s[5];
for (size_t i = 0; i < 5; i++) {
        std::cin >> s[i];
}
sort<std::string>(s, 5);
```

Objets fonctions

- Un *objet fonction*, ou *foncteur*, est une entité qui peut être appelée comme une fonction
- Repose sur la surcharge de l'opérateur **()**
- Permet seulement l'appel de l'opérateur **()**
- Des objets fonctions jouent un rôle capital dans la bibliothèque standard de C++

Exemple 6.15 Un objet fonction

```
template<class T>
struct neg {
        T operator()(const T &x) const {
                return x * (-1);
        }
};
```

Exemple 6.15 Un objet fonction

```
neg<int> neg_int;
int x = neg_int(5);                    // x == -5

neg<double> neg_double;
double y = neg_double(5.5);            // y == -5.5
```

Exemple 6.15 Un objet fonction

```
neg<Date> neg_date;
Date d = nd(Date("21/12/2012"));          // d == 21/12/-2012
// L'opérateur * avec int doit ête implémentée
// Date Date::operator*(int) const {
        // TODO
// }
```

Exemple 6.16 Deux objets fonctions typiques

```
struct comparator {};

template<class T>
struct greater : public comparator {
        bool operator()(const T &a, const T &b) const {return a > b;}
};

template<class T>
struct less : public comparator {
        bool operator()(const T &a, const T &b) const {return a < b;}
};
```

Exemple 6.16 Deux objets fonctions typiques

```
int main() {
        greater<int> gt;
        if (gt(5, 3)) cout << "Oui, 5 > 3 !" << endl;
        less<int> lt;
        if (lt(5, 3)) cout << "Non, 5 > 3 ?" << endl;
}
```

Exemple 6.17 Un algorithme de tri paramétrique

```
template<class T>
void swap(T &a, T &b) {
        T tmp = a;
        a = b;
        b = tmp;
}
```

Exemple 6.17 Un algorithme de tri paramétrique

```
template<class T, class C>
void sort(T x[], size_t n, const C &comp) {
        for (size_t i = 0; i < n; i++)
                for (size_t j = i + 1; j < n; j++)
                        if (comp(x[i], x[j]))
                                swap<T>(x[i], x[j]);
}
```

Exemple 6.17 Un algorithme de tri paramétrique

```
int y[] = {1, 2, 5, 4, 3};
sort<int, comparator>(y, 5, greater<int>());
sort<int, comparator>(y, 5, less<int>());

double z[] = {1.1, 2.2, 5.5, 4.4, 3.3};
sort<double, comparator>(z, 5, greater<double>());
sort<double, comparator>(z, 5, less<double>());
```

Exemple 6.17 Un algorithme de tri paramétrique

```
Date d[5];
for (size_t i = 0; i < 5; i++) {
        std::cin >> d[i];
}
sort<Date, comparator>(d, 5, greater<Date>());
sort<Date, comparator>(d, 5, less<Date>());
```