Classes

1. Objectif
2. Déclaration
3. Définition
4. Encapsulation
5. Constructeurs
6. Une classe de complexes
7. Une classe de rationnels
8. Surcharge d'opérateurs
9. Liste d'initialisation
10. Membres statiques
11. Méthodes constantes
12. La classe string

Objectifs

Une classe est la description d'une famille d'objets ayant même structure et même comportement.

Une classe regroupe un ensemble d'attributs ou membres, répartis en

* un ensemble de données
* un ensemble de fonctions, appelées méthodes.

Avantages !!!

* simplifie l'utilisation des objets,
* rapproche les données et leur traitement: c'est l'objet qui sait le mieux comment gérer une demande,
* renforce la robustesse du programme et la structure
* simplifie la maintenance et la construction
* permet l'encapsulation : possibilité de ne montrer de l'objet que ce qui est nécessaire à son utilisation.
* (permet de) masquer l'implémentation

Classes (2)

L'utilisateur dispose
* de mécanismes de construction (et de destruction) d'objets;
* de méthodes d'accès et de modification des données encapsulées.

Un objet est un élément de la classe. C'est une instance de la classe. Il est obtenu par instanciation.

La classe permet de produire autant d'exemplaires d'objets que nécessaire.
* Les valeurs des données membres peuvent différer d'une instance à l'autre (sauf pour des données statiques, de classe).
* Les méthodes sont les mêmes pour toutes les instances d'une classe. On distingue entre méthodes d'objet (d'instance) et méthodes de classe.

La déclaration d'une classe donne la nature des membres (type, signature), et les droits d'accès: public, protected, private (défaut).

La définition d'une classe fournit la définition des méthodes.

L'encapsulation peut se pratiquer en donnant à l'utilisateur
* un fichier en-tête contenant la déclaration de la classe;
* un module objet contenant la version compilée du fichier contenant la définition de la classe.

Déclaration

La syntaxe est celle des structures. Une `struct` est une classe dont tous les attributs sont publics.

```
class Point {
  int x, y;
 public:
  void setPoint (int, int);
  void deplace (int, int);
  void affiche ();
};
```

* La classe `Point` a deux membres données **privés** `x` et `y`.
* Elle a trois méthodes **publiques** `setPoint, deplace` et `affiche`.

Définition

L' opérateur de portée :: indique la classe à laquelle appartient la méthode.

```
void Point::setPoint(int a, int b) {
  x = a; y = b ;
}

void Point::deplace (int dx , int dy) {
  x += dx; y += dy;
}
void Point::affiche() {
  cout << "x = " << x <<" , y = " << y << endl;
}
```

Les champs x, y invoqués dans les méthodes sont ceux de l'objet qui appelle la méthode. Cet objet est explicitement accessible par le pointeur this. On peut écrire

```
void Point::setPoint(int a, int b) {
  this -> x = a; this -> y = b ;
}
```

Utilisation

```
void main() {
  Point a, b;
  a.setPoint(1, 2);
  b.setPoint(3,-5);
  a.affiche(); // x = 1, y = 2
  b.affiche(); // x = 3, y = -5
  a.deplace(1,1);
  a.affiche(); // x = 2, y = 3
}
```

Encapsulation

Principe de programmation :
Il s'agit de **ne** montrer à l'utilisateur que ce qui lui est destiné.
Dans la version plus élaborée:

* seules des fonction d'accès et de modification sont rendues publiques;
* toute donnée est privée ou protégée;
* même la documentation ne fait état que des données et méthodes publiques.

Dans une version simplifiée, l'implémentation n'est disponible que par un module objet.

Exemple. Le projet est composé de trois fichiers

* Un fichier point.h de déclaration de la classe Point
* Un fichier point.c d'implémentation des méthodes de la classe
* Un fichier main.c d'utilisation.

Le fichier d'implémentation est à terme remplacé par un module objet point.o ou une bibliothèque.

```
// Fichier point.h
#ifndef POINT_H
#define POINT_H
class Point {
  int x, y;
 public:
  Point (int, int);
  void deplace (int, int);
  void affiche ();
};
#endif
```

Remarquer les directives du préprocesseur qui permettent d'inclure plusieurs fois le fichier point.h sans inconvénients.

```
// Fichier point.c
#include <iostream.h>
#include "point.h"
Point::Point(int a, int b) {
  x = a; y = b ;
}

void Point::deplace (int dx , int dy) {
  x += dx; y += dy;
}
void Point::affiche() {
  cout << "x = " << x <<" , y = "
       << y << endl;
}
```

```
// Fichier main.c
#include "point.h"
void main()
{
  Point a(1,2), b(3,-5);
  a.affiche();
  b.affiche();
  a.deplace(1,1);
  a.affiche();
}
```

Constructeurs

Les constructeurs permettent de définir les initialisations possible des instances d'une classe.

* En C++, un constructeur a le nom de la classe, et pas de type de retour.
* Une classe peut avoir plusieurs constructeurs.
* Un constructeur sans arguments est appelé constructeur par défaut.
    - ce constructeur existe implicitement, s'il est le seul constructeur.
    - la définition d'un deuxième constructeur exige que l'on définisse explicitement le constructeur par défaut si l'on veut s'en servir.

Le constructeur par défaut est utilisé

* lors de la définition d'un objet, par

```
X x;
```

* lors d'une allocation, par

```
px = new X;
```

Sauf écriture explicite, le constructeur par défaut n'effectue pas d'action.

Destructeur

Le destructeur est noté `~X()` pour une classe `X`. Il est utilisé

* lorsque le programme quitte le bloc où l'objet est déclaré
* pour la destruction explicite par

```
delete px;
```

Le destructeur permet de spécifier ce qui doit être fait quand l'objet est désalloué. Ceci est important quand un constructeur fait de l'allocation explicite.

En plus, il existe un constructeur de copie, un opérateur d'affectation, et des constructeurs de conversion qui seront vus plus tard.

Exemple constructeur

Remplacement de la méthode `setPoint` par un constructeur :

```
class Point {
  int x, y;
 public:
  Point (int, int);
  void deplace (int, int);
  void affiche ();
};
```

```
Point::Point(int a, int b) {
  x = a; y = b ;
}
...
void main()
{
  Point a(1.5,2.5), b(3,-5);
  ...
}
```

Le constructeur `Point(int,int)` rend **inopérant** le *constructeur par défaut* s'il n'est pas redéfini.

Le constructeur est employé en passant les arguments en paramètre.
C'est une abréviation de

```
Point a = Point(1.5,2.5), b = Point(3,-5);
```

Variations

* Un constructeur simple peut être défini en ligne, par

```
class Point {
  int x, y;
 public:
  Point (int a, int b) {x = a; y = b;}
  ...
};
```

* Au lieu de construire l'objet puis d'affecter des valeurs aux champs, on peut initialiser les champs avec les valeurs, par

```
Point (int a, int b) : x(a), y(b) {}
```

Tableaux d'Objets

Tableaux d'objets attention le constructeur par défaut est appelé sur chaque élément du tableau.

```
main () {
   const int taille = 7;
   Point* a = new Point[taille];  // par defaut
   for (int i=0; i < taille ; i++)
     cout << i << a[i].x << a[i].y << endl;
   delete [] a;
}
```

La libération d'un tableau d'objets par `delete []`

* parcourt le tableau et appelle le destructeur sur chaque élément du tableau;
* puis libère le tableau.

Construction&Destruction

La mention explicite du constructeur par défaut et du destructeur signifie leur redéfinition.

```
class Id {
  string nom;
public:
  Id() { cout << "Entrez votre nom : ";cin >> nom;}
  ~Id(){ cout <<"Mon nom est "<< nom <<endl; }
};
```

Le programme se réduit à:

```
void main()
{
  Id pierre, paul;
}
```

Ou:

```
Id pierre,paul;
void main(){} // :)
```

Voici une trace d'exécution:
Entrez votre nom : Pierre
Entrez votre nom : Paul
Mon nom est Paul
Mon nom est Pierre

Une classe de Complexes

```
class Complexe {
  double re, im;
public:
  Complexe() { re = im = 0;}
}
----
```

Le constructeur par défaut Complexe() est explicité pour fixer des valeurs de départ. Ainsi

```
Complexe s, t;
```

définit deux complexes s, t initialisés à zéro.

Un constructeur (de conversion) est utile pour convertir un couple de double en complexes:

```
Complexe::Complexe(double x, double y) {re = x; im = y;}
```

La conversion d'un réel en complexe est faite en mettant im à zéro:

```
Complexe::Complexe(double x) {re = x; im = 0;}
```

Si les fonctions sont courtes, on les définit en ligne. Ensemble:

```
class Complexe {
  double re, im;
  Complexe() { re = im = 0;}
  Complexe(double x, double y) {re = x; im = y;}
  Complexe(double x) {re = x; im = 0;}
}
```

En utilisant les arguments par défaut, les trois constructeurs se réduisent en un seul.
Ici, on remplace l'affectation par l'initialisation, et on ajoute une méthode:

```
class Complexe {
  double re, im;
  Complexe(double r = 0, double i = 0) : re(r), im(i) {}
  double getModule() { return sqrt(re*re + im*im); }
}
```

On s'en sert

* à l'initialisation, par exemple

```
Complexe h, a(3,5), b = Complexe(2,4);
Complexe c(5), d = Complexe(6), e = 7;
```

Par conversion de type, 7 est transformé en Complexe(7) puis copié. [ZZZ: LOURD!]

* pour la réalisation d'une conversion dans une expression:

```
void f(Complexe z) {...}
  ...
f(3.14)
```

* par appel explicite dans une instruction

```
r = Complexe(3,4).getModule();
```

Une interface plus complète

La déclaration est:

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

const double pi = 3.14159265358979323846;
class Complexe {
public :
  Complexe(double re = 0, double im = 0);
  Complexe(double rho, double theta, int polaire);
  double getRe();
  double getIm();
  double getModule();
  double getArg();
  void setRe(double);
  void setIm(double);
  void setModule(double);
  void setArg(double);
  void showXY() { cout << getRe() << " " << getIm() <<" "; }
  void showRT() { cout << getModule() << " " << getArg() <<" "; }
private:
  double re, im; // Premiere implementation
  static double getArgFromCart(double x, double y);// helper
};
```

```
Complexe::Complexe(double re, double im): re(re), im(im){}
Complexe::Complexe(double rho, double theta, int polaire) {
  re = rho*cos(theta);
  im = rho*sin(theta);
}
double Complexe::getRe() { return re; }
double Complexe::getIm() { return im; }
double Complexe::getModule() { return sqrt(re*re+im*im); }
double Complexe::getArg() { return getArgFromCart(re,im); }
void Complexe::setRe(double x) { re = x; }
void Complexe::setIm(double y) { im = y; }
void Complexe::setModule(double r) {
  double m = getModule();
  if(!m) { re = r; }
  else { re = re*r/m; im = im*r/m; }
}
void Complexe::setArg(double arg) {
  double r = getModule();
  re = r* cos(arg);
  im = r* sin(arg);
}

double Complexe::getArgFromCart(double x, double y) {
  if (!x) return atan(y/x);
  if (y >0) return pi/2;
  if (y < 0) return -pi/2;
  return 0;
}
```

ImplComp1

Deuxième Implémentation (le rôle des deux double change) pour simplfier nous avons changer leurs noms en rho et theta.

```
Complexe::Complexe(double re, double im) {
  rho = sqrt(re*re+im*im);
  theta = getArgFromCart(re,im);
}
Complexe::Complexe(double rho, double theta, int polaire):
  rho(rho), theta(theta) {}

double Complexe::getRe() { return rho*cos(theta); }
double Complexe::getIm() { return rho*sin(theta); }
double Complexe::getModule() { return rho; }
double Complexe::getArg() {return theta; }
void Complexe::setRe(double re) {
  double im = getIm(); rho = sqrt(re*re+im*im);
  theta = getArgFromCart(re,im);
}
void Complexe::setIm(double im) {
  double re = getRe(); rho = sqrt(re*re+im*im);
  theta = getArgFromCart(re,im);
}
void Complexe::setModule(double r) { rho = r; }
void Complexe::setArg(double arg) { theta = arg; }

double Complexe::getArgFromCart(double x, double y) {
  if (!x) return atan(y/x);
  if (y >0) return pi/2;
  if (y < 0) return -pi/2;
  return 0;
}
```

ImplCompl 2

Utilisation des Deux implémentations

Le module client (ici le main) ne change pas quand l'implémentation change, l'encapsulation est bien réalisée.

```
void main()
{
  Complexe a = Complexe(3,pi/4,0);
  cout << a.getModule() << endl;
  cout << a.getArg() << endl;
  a.setModule(10);
  a.showXY();a.showRT();
  a.setArg(pi/2);
  a.showXY();a.showRT();
  Complexe b(5);
  b.setArg(pi/4);
  b.showXY();b.showRT();
}
```

Un attribut n'a pas d'existence en dehors d'un objet. L'implantation n'assure pas l'existence physique d'un attribut.

Par exemple, le module n'est accessible que par getModule() et setModule(), mais il n'est pas assuré qu'il existe une données membre qui maintient la valeur courante.

On appelle attribut calculé un attribut qui n'est pas physiquement présent dans une donnée membre. Si l'implantation cache les données (data hiding), un utilisateur ne peut pas voir la différence entre un attribut présent et un attibut calculé.

Les rationnels

L'objectif est de montrer l'utilisation de surcharges, cette fois-ci sur les opérateurs arithmétiques et d'entrée-sortie.

½ -3547/977777 0/1 etc

Un nombre rationnel est toujours réduit, et le dénominateur est toujours strictement positif. D'où la nécessité d'une méthode de réduction.

[réduit: numérateur et dénominateur sont des entiers premiers entre eux, c-a-d les plus petits possibles.]

Rat

```
class Rat {
  static int pgcd(int,int);
public:
  int num, den;
  void red();  // reduit la fraction
  Rat(int n = 0) : num(n), den(1) {}
  Rat(int n, int d) : num(n), den(d)
     { red();}
};
```

Il y a deux constructeurs;
seul le deuxième fait usage du réducteur.

```
Rat a;       // a = 0
Rat b(3);    // b = 3
Rat c(3,2);  // c = 3/2
Rat d = 7;   // d = 7
```

Par conversion de type, 7 est transformé en Rat(7) puis d est initialisé à cette valeur
(ZZZ: appel des **deux** constructeurs).

La réduction fait appel à une fonction de calcul de pgcd:

void Rat::red() { int p = pgcd((num > 0) ? num : -num, den); num /= p; den /= p; }

int Rat::pgcd(int a, int b) { return (!b) ? a : pgcd(b, a%b); }

Une méthode déclarée static (*Méthode de classe*)
*** peut être appelée sans référence à une instance**
*** peut être appelée par référence à sa classe**
*** n'a pas de pointeur this associé.**

Une donnée déclarée static (*Donnée de classe*) est commune à toutes les instances, une seule allocation et donc une seule valeur.

Exemple de Surcharge d'opérateurs

Comme toute fonction, un opérateur peut également être surchargé. Pour surcharger un opérateur *op* on définit une nouvelle fonction de nom `operator`*op*

Par exemple `operator=` ou `operator+`.
La syntaxe est
type operatorop(types des opérandes) { /* corp de l'opérateur */ return valeur; }

Par exemple

```
Rat operator+(Rat,Rat);
```

L'expression

```
a+b
```

est implicitement traduite par le compilateur en

```
operator+(a, b)
```

Un opérateur surchargé peut se définir
* au niveau global,
* comme membre d'une classe.

Apport de la surcharge: le calcul s'écrit comme pour un type élémentaire !

Nous définissons les quatre opérations arithmétiques par:

```
Rat operator+(Rat a, Rat b) {
  return Rat(a.num*b.den + a.den*b.num, a.den*b.den);
}
Rat operator-(Rat a, Rat b) {
  return Rat(a.num*b.den - a.den*b.num, a.den*b.den);
}
Rat operator/(Rat a, Rat b) {
  return Rat(a.num*b.den, a.den*b.num);
}
Rat operator*(Rat a, Rat b) {
  return Rat(a.num*b.num, a.den*b.den);
}
```

[zzz: appel du constructeur sur la valeur de retour.]
Le moins unaire se définit par:

```
Rat operator-(Rat a) {
  return Rat(-a.num, a.den);
}
```

Les rationnels passé en argument ne sont pas modifiés, on peut donc les spécifier `const Rat&`.

```
Rat operator+(const Rat& a, const Rat& b) {...}
Rat operator-(const Rat& a, const Rat& b) {...}
...
```

L'opérateur d'incrémentation += se surcharge également:

```
Rat& operator+=(Rat& a, const Rat& b) {
  a.num = a.num*b.den + a.den*b.num;
  a.den = a.den*b.den; a.red();
  return a;
}
```

L'un des appels

```
   a += b; //ou
   operator+=(a,b);
```

* construit dans a passé par référence, la valeur incrémentée,
* et en transmet la référence en sortie.

Ici, une définition en méthode est plus logique (voir plus loin).

Les opérateurs d'entrée et de sortie se surchargent également:

```
ostream& operator<<(ostream& out, Rat r) {
  return out << r.num << "/" << r.den;
}
istream& operator>>(istream& in, Rat& r) {
  in >> r.num >> r.den; r.red();
  return in;
}
```

Les opérateurs d'entrée et de sortie prennent et retournent la référence au flot en argument, pour éviter une copie.
Noter la référence dans la lecture ! Et voici un test:

```
void main()
{
   Rat a,b,c;
   cout << "1 - 1/2 = "<<1 - Rat(1)/2<<endl;
   cout << "Donner deux rationnels : ";
   cin >> a >> b;
   cout <<  -a<< endl;
   cout <<  -a +1<< endl;
   cout <<  -a +1 - (Rat) 1/2<< endl;
   cout <<  -a  +1 - (Rat) 1/2 +a << endl;
   c = -a +1 - (Rat) 1/2 +a +b*2;
   cout << "\nResultat " << c << endl;
}
```

```
1 - 1/2 = 1/2
Donner deux rationnels : 2 5 8 4
-2/5
3/5
1/10
1/2

Resultat 9/2
```

Liste d'initialisation

Un constructeur peut recevoir une liste d'initialisations qui sont effectuées à la construction de l'objet. Ceci est important quand un ou des champs de l'objet sont eux-mêmes des objets.

La construction d'un objet par un constructeur sans liste d'initialisation se fait en trois étapes:
* réservation de la place pour l'objet;
* construction des champs par appel du constructeur par défaut de chacun des objets;
* exécution du corps de la fonction.

En présence d'une liste d'initialisation, les constructeurs par défaut sont remplacés par les appels spécifiés dans la liste.

Exemple : une classe de points et une classe de segments, formées de deux points.

```
class Point {
    int x, y;
  public:
   Point(int abs=0,int ord=0);
};
```

```
class Segment {
  Point d,f;
public:
Segment(int dx,int dy, int fx, int fy);
};
```

La Première implémentation du constructeur

```
Segment::Segment(int dx,int dy, int fx, int fy) {
  d = Point(dx,dy); f = Point(fx,fy);
}
```

Avec la construction

```
  Segment k(1,2,3,4);
```

provoque:

* deux appels au constructeur par défaut Point() pour l'initialisation de d et de f, et ceci avant d'entrer dans le corps du constructeur de segments;
* les appels de constructeurs Point(1,2) et Point(3,4) pour construire des objets temporaires;
* deux emplois de l'opérateur d'affectation (copie des Points);
* deux appels au destructeur pour détruire les objets temporaires.

Deuxième implémentation

```
Segment::Segment(int dx,int dy,int fx,int fy) : d(dx,dy), f(fx,fy)
  {} // Oui rien ! C'est bien
```

La construction ne nécessite que deux appels de constructeurs, pour construire d à partir de (dx,dy) et de f à partir de (fx,fy).

Utiliser le plus souvent possible les listes d'initialisation.

Membres statiques (le retour)

Parmi les attributs, on distingue
* les membres de classe, qualifiés de statiques;
* les membres d'objet.

Une donnée membre *statique* est spécifiée par `static`.
* Elle n'est instanciée qu'une seule fois;
* Elle est commune à toutes les instances de la classe;
* Si un objet la modifie, elle est modifiée pour tous les objets.
* Elle est initialisée avant utilisation.

Une donnée *statique* et *constante* est une donnée immuable.

Un méthode statique est spécifiée par `static`.
* Elle ne fait pas référence à un objet (pas de `this`);
* Elle est appelée comme une fonction globale;
* S'il faut préciser sa classe, son nom est préfixé à l'aide de l'opérateur de portée : `Math::pgcd()`.

Exemple de donnée statique

```
#include <iostream.h>

class XY {
 public:
   static int app;
   XY() { cout << "+ : " << ++app << endl; }
   ~XY() { cout << "- : "<< app-- << endl; }
};

int XY::app = 0; // initialisation globale

void main() {
  XY a, b, c;
  {
    XY a, b;
  }
  XY d;
}
```

Résultat:

```
+ : 1
+ : 2
+ : 3
+ : 4
+ : 5
- : 5
- : 4
+ : 4
- : 4
- : 3
- : 2
- : 1
```

Méthodes constantes

Les méthodes constantes ont deux utilisations:
* une formelle les objets constant ne sont manipulables que par des méthodes constantes, c'est validé par le compilateur.
* une pratique les méthodes constantes ne change pas l'état de l'instance.

Une méthode qui ne modifie pas l'objet appelant peut recevoir le qualificatif const en suffixe. Les ``getter functions'' sont de ce type.

Une des disciplines de programmation objet suggère:
* les données membres sont privées ou protégées;
* on peut obtenir la valeur d'un membre donnée par une fonction d'accès (*getter* function);
* on autorise la modification d'une donnée membre en fournissant une fonction de modification (*setter* function);

Cette discipline s'étend au nom des fonctions. Plusieurs variantes de cette discipline existent.

Toujours se conformer à la culture de l'entreprise.

[ZZZ; Utilisez les mêmes conventions que Julien Allali]

```
class Rat { public:
  ...
  int getnum() const { return num_;} // méthode constante
 ...
};
```

La classe string

La classe string de C++ facilite la manipulation de chaînes de caractères.

Pour employer la classe string, inclure le fichier en-tête string.

```
#include <string>
```

L'implémentation de la classe string est complexe, et basée sur plusieurs autres classes. Nous décrivons ses possibilités comme si elle était implémentée directement.

Constructeurs

`string()` contructeur par défaut, initialise une chaîne vide.

`string(const char* c)` construit une chaîne en l'initialisant avec la chaîne style C.

`string(const string& a)` contructeur par recopie, initialisé avec une copie de a.

```
string c;           // vide
string a = "Lotus";// string("Lotus")
string b("bleu");  // string("bleu")
string d(b);        // copie
for(int i=0;i<a.size();i++)
 cout << a[i] << ' '; // affiche L o t u s.
```

Noter que

* size() retourne la taille.
* a[i] rend le caractère correspondant à l'indice, par surcharge de l'opérateur d'indexation.
* at() joue le même rôle: a.at(i) équivaut à a[i], mais en plus il y a contrôle de la validité de l'indice.

Opérateurs sur string

Opérateurs

* "+" pour la concaténation.

```
string a = "Lotus", b("bleu");
string c = a + " " + b;
cout << c << endl;
```

donne Lotus bleu.

* "+=" de concaténation incrémentale:

```
string a = "Lotus"; a += " bleu";
```

donne à nouveau Lotus bleu.

* les 6 opérateurs de comparaison comme opérateurs globaux, et notammement == pour la comparaison de chaînes.

```
String a = "Lotus bleu";
cout ("bleu" > a) << endl;
```

donne 1.

Aussi,

```
string x;        //Chaine vide
string y = "";   //Chaine contenant mot vide
cout<< x.size() << y.size() << (x==y) << endl;
```

donne (heureusement) 001.

Méthodes de string

Autres méthodes

replace(int d, int n, string b)

remplace dans la chaîne appelante n caractères à partir de la position d par la chaîne b.

```
string a = "Lotus bleu ciel", b = "jaune";
a.replace(6,4, b);
```

donne "Lotus jaune ciel".

compare(const string& b)

retourne un entier négatif, nul, positif si la chaîne appelante est inférieure, égale, supérieure à b.

```
string x = "adam", y = "eve", z = "evelyne";
cout << x.compare(y) <<' '<< y.compare(z) <<' '
  << z.compare(z) <<' '<< z.compare(x)<< endl;
```

donne -4 -4 0 4.

Conversion en chaînes "C"
const char* c_str()
    copie les caractères dans une chaîne C constante, terminée par le caractère nul.

```
string s = "coup de main";   s[4] = 0; // caractere nul
cout << s << endl; const char* c = s.c_str();
cout << c << endl;
```

    donne

```
coupde main
coup
```

int copy(char *t, int n, int d = 0)

    copie dans la chaîne existante t au plus n caractères à partir de la position d.

```
string s = "coup de main";
char txt[20];
s.copy(txt, 4, 8);
txt[4] = 0;
cout << txt << endl;
```

    donne main.