Introduction à C++

Plan

1. "Un meilleur C"
2. Arguments par référence
3. Surcharge de fonctions
4. Entrées-sorties
5. Allocation dynamique
6. Classes et objets
7. Constructeurs
8. Un exemple: un compte en banque
9. Héritage
10. Composition
11. Méthodes virtuelles
12. Encapsulation
13. Les patrons

Arguments par référence

En C Ansi:

```
void echange(int* a, int* b) {
  int t;
  t = *a;
  *a = *b;
  *b = t;
}
```

Appel

```
echange (&x, &y);
```

En C++

```
void echange(int& a, int& b) {
  int t;
  t = a;
  a = b;
  b = t;
}
```

Appel

```
echange (x, y);
```

Surcharge de fonctions

Un même identificateur peut désigner plusieurs fonctions, à conditions qu'elles diffèrent par la liste des types de leurs arguments.

```
float max (float a, float b) {
  return (a>b) ? a : b; }
float max (float a, float b, float c) {
  return max (max (a, b), c); }
float max (int, float []);
```

L'analyse de la signature détermine la fonction à utiliser.

```
void main() {
  float a = 1.86, b = 3.14, c =37.2;
  float T[] ={11.1, 22.2, 33.3, 44.4, 7.7, 8.8 };

  max (a, b);
  max (a, b, c);
  max (6, T);
}
```

On peut aussi surcharger les operateurs : définir par exemple l'addition de nombres complexes et représenter cette addition par le signe `+`.

Entrées-sorties

Les entrées et sorties sont gérées dans C++ à travers des objets particuliers appelées *streams* ou *flots*.
Deux *opérateurs* sont surchargées de manière appropriée pour les flots:

- l'opérateur d'*insertion* `<<` (écriture)
- l'opérateur d'*extraction* `>>` (lecture)

Les flots prédéfinis sont

- `cout` attaché à la sortie standard;
- `cerr` attaché à la sortie erreur standard;
- `cin` attaché à l'entrée standard.

```
#include <iostream.h>
main() {
  cout << "Bonjour, monde !\n";
}
```

```
#include <iostream.h>
int i
main() {
  cout << "Un entier : ";
  cin >> i;
  cout << "Le carre de " << i <<" est " << i*i << endl;
}
```

Les opérateurs d'insertion et extraction sont

- sont binaires avec arguments un stream et une expression;
- il retournent le flots du premier argument;
- ils sont assciatifs à gauche.

```
cout << "Carre de " << i <<" est " << i*i << endl;
```

équivaut à

```
((((cout << "Carre de ") << i) <<" est ") << i*i) << endl;
```

chacune des expressions

```
cout << "Carre de "
(cout << "Carre de ") << i
((cout << "Carre de ") << i) <<" est "
(((cout << "Carre de ") << i) <<" est ") <<
((((cout << "Carre de ") << i) <<" est ") << i*i) << endl
```

vaut `cout`.

Allocation dynamique

L'opérateur **new** alloue la mémoire requise par le type passé en argument et retourne l'adresse.

```
char t[] = "Hello, world !";

char* copy(char* t) {
  char* a;
  a = new(char[strlen(t)+1]);
  strcpy(a,t);
  return a;
}


typedef int vec[120];
...
vec* a = new(vec);
```

Classes et objets

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

Trois mots-clé définissent une classe : `struct`, `class`, et `union`.

Un classe regroupe

- un ensemble de données (champs nommés);
- un ensemble de procédures (méthodes).

```
struct date {
  int jour, mois, annee;
  void faire(int, int, int);
  void lendemain();
  void imprimer();
}
```

Un *objet* ou une *instance* d'une classe, est un objet dont le type est la classe. Il est obtenu par *instanciation*.

Emploi:

```
int j;
date d;
j = d.jour;
d.imprimer();
```

Appeler une méthode sur un objet revient à appeler une fonction dont un premier paramètre implicite est l'objet.

```
d.imprimer();
```

revient à

```
imprimer_(&d);
```

De même, le nom de l'instance est implicite dans une méthode:

```
void date::faire(int j, int m, int a) {
  jour = j; mois = m; annee = a;
}
```

On a un accès à l'objet, à l'intérieur d'une méthode, par le pointeur `this`.

```
void date::faire(int j, int m, int a) {
  this -> jour = j;
  this -> mois = m;
  this -> annee = a;
}
```

A noter:

- Les champs (variables membres) ont des valeurs qui peuvent différer d'une instance à l'autre
- Les méthodes sont les mêmes pour toutes les instances d'une classe.

Constructeurs

Chaque classe a par défaut un *constructeur* par défaut qui ne fait rien d'autre que réserver la place.

Ce constructeur est appelé lors de l'instanciation d'un abjet.

```
date d;               //constructeur par defaut
date *pd;
...
pd = new(date);       //constructeur par defaut
```

Un constructeur se définit comme une fonction, mais sans type de retour, et porte le nom de la classe.

```
struct date {
  int jour, mois, annee;
  ...
  date () { jour = mois = annee = 0;}
  date(int j, int m, int a) {
    jour = j; mois = m; annee = a;}
}
```

Emploi :

```
date d;               // champs nuls
date e(8,12,1997);    // champs affectes
```

Un exemple: un compte en banque

```
class Compte {
public:
  char nom[20];
  int debit;
  int credit;

  Compte(char *);
  void deposer(int);
  void retirer(int);
  int solde();
  virtual void etat();
};
```

Il y a

- trois *données*: `nom`, `debit`, `credit`;
- quatre *méthodes*: `deposer`, `retirer`, `solde` et `etat`, plus un *constructeur*.

Un champs d'une classe est par défaut `private` et inaccessible sauf aux méthodes de la classe.

Les autres spécificateurs de droits d'accès sont `protected` (accès autorisé par les méthodes de la classe et des classes dérivées) et `public` (accès sans restriction).

Une `struct` est une classes dont tous les champs sont `public`.

On déclare un objet comme en C, avec d'éventuels paramètres d'initialisation:

```
Compte pierre("Pierre"), paul("Paul");
```

La définition du constructeur est:

```
Compte::Compte(char * n) {
  strcpy(nom, n);
  credit = 0;
  debit = 0;
}
```

Appel des méthodes

```
  paul.deposer(500);
  paul.retirer(300);
  paul.etat();
```

`paul.deposer(500);` réalise l'appel de la fonction `deposer` *sur* l'objet paul. Voici l'implémentation de la méthode:

```
void Compte::deposer(int s) {
  credit += s;     // this -> credit
}
```

L'effet de

```
  paul.deposer(500);
```

est donc d'incrémenter le champ `credit` de paul.

De manière générale, la méthode s'applique à l'objet instance; elle effectue l'action sur l'objet appelant.

```
void Compte::retirer(int s) {
  debit += s;
}

int Compte::solde() {
  return credit - debit;
}
```

```
void Compte::etat() {
  cout <<nom<<"\t : debit "<<debit
     <<" : credit "<<credit<<" : solde "<<solde()<<endl;
}
```

La séquence de messages

```
  pierre.etat();
  paul.deposer(500);
  paul.retirer(300);
  paul.etat();
  pierre.retirer(500);
  pierre.etat();
```

donne

```
Pierre   : debit 0 : credit 0 : solde 0
Paul     : debit 300 : credit 500 : solde 200
Pierre   : debit 500 : credit 0 : solde -500
```

Héritage

L'héritage est un mécanisme de spécialisation, par

- adjonction de nouvelles propriétés et nouvelles méthodes;
- remplacement (masquage) de méthodes existantes.

La classe *dérivée* hérite des données et méthodes de la classe *de base*. La syntaxe est

```
classe CDer : public CBase {
..
}
```

Ici, `public` donne à la classe dérivée accès sans restriction à la classe de base.

```
class CompteEpargne : public Compte {
public:
  float taux;
  void interets();
  void etat();
  CompteEpargne(char* n, float t) : Compte(n) { taux = t;}
};
```

Nouvelle donnée:

```
  float taux;
```

Nouvelles méthodes:

```
  void interets();  // nouvelle
  void etat();      // redefinie
```

Constructeur propre:

```
  CompteEpargne(char* n, float t) : Compte(n) { taux = t;}
```

utilise le constructeur `Compte(n)` pour initialiser la partie `Compte` et ne précise que la partie spécifique.

Implémentation:

```
void CompteEpargne::interets() {
  credit += (int) ((credit - debit)*taux/100);
}

void CompteEpargne::etat() {
  interets();
  Compte::etat();
}
```

La méthode `etat()`, appelée sur un objet de la classe `CompteEpargne`, utilise d'abord la méthodes `interets()` puis la méthode `Compte::etat()` de la classe de base.

Sans la précision de la classe `Compte::`, on aurait obtenu une fonction récursive.

A noter

- Un objet de la classe *dérivée* possède les champs et méthodes de sa superclasse (ou classe de base).
- Les méthodes de la classe dérivée l'emportent sur celles de la classe de base.

Ici

```
CompteEpargne eve("Marie",4.5);
eve.deposer(1000);
eve.etat();
```

donne

```
Marie     : debit 0 : credit 1045 : solde 1045
```

car la méthodes `etat()` utilisée par `eve` est celle de la classe `CompteEpargne`.

Composition

Une donnée d'une classe peut être objet d'une autre classe. Voici l'exemple d'un "banque" rudimentaire, réduite à une liste de comptes, et manipulant des comptes.

```
class Banque {
public:
  int nclients;
  Compte* liste[10];

  Banque() { nclients = 0; }
  int ouvrircompte(char*) ;
  int ouvrircompte(char*, float);
  void etat();
  void deposer(int s, int c);
  void retirer(int s, int c);
  void virer(int s, int cdepart, int carrive);
};
```

Le *constructeur* initialise la banque.

La *surcharge* permet de donner le même nom à des fonctions, comme `ouvrircompte` qui diffèrent par la signature de leur liste d'arguments.

L'ouverture d'un compte retourne un *numéro* par lequel passent les transactions.

Les méthodes:

```
int Banque::ouvrircompte(char * n) {
  liste[nclients] = new Compte(n);
  return nclients++;
}

int Banque::ouvrircompte(char * n, float t) {
  liste[nclients] = new CompteEpargne(n,t);
  return nclients++;
}

void Banque::etat() {
  for (int c=0; c<nclients; c++)
    liste[c] ->etat();
}

void Banque::deposer(int s, int c) {
  liste[c] -> deposer(s);
}
void Banque::retirer(int s, int c) {
  liste[c] -> retirer(s);
}
void Banque::virer(int s, int cdepart, int carrive) {
  retirer(s, cdepart);
  deposer(s, carrive);
}
```

Le scénario

```
Banque b;
int p, e;
p = b.ouvrircompte("Pierre");
e = b.ouvrircompte("Eve",4.5);
b.deposer(500,p);
b.deposer(500,e);
b.virer(300,p,e);
b.retirer(100,p);
b.deposer(1300,e);
b.virer(100,e,p);

b.etat();
```

donne l'état:

```
Pierre    : debit 400 : credit 600 : solde 200
Eve       : debit 100 : credit 2190 : solde 2090
```

Méthodes virtuelles

A noter:

```
int Banque::ouvrircompte(char * n, float t) {
  liste[nclients] = new CompteEpargne(n,t);
  return nclients++;
}

void Banque::etat() {
  for (int c=0; c<nclients; c++)
    liste[c] ->etat();
}
```

La ligne

```
liste[nclients] = new CompteEpargne(n,t);
```

ne transforme pas l'*objet* de la classe `CompteEpargne` en un objet de la classe `Compte` mais seulement le pointeur associé.

La méthode

```
liste[c] ->etat();
```

- est-elle celle de la classe `Compte`, ce qui est raisonnable puisque `liste` est de type `Compte*` ?
- ou est-elle de la classe `CompteEpargne` pour `Eve`, puisque `Eve` *est* un compte d'épargne ?

Les deux réalisations sont possibles:

- une méthóde déclarée *virtuelle* dans la classe de base cède la place à la méthode de la classe dérivée si l'objet est, à la construction, de la classe dérivée.
- une méthode non virtuelle est appelée en fonction du type de la variable qui contient l'objet.

Encapsulation

L'*encapsulation* a pour but de

- rapprocher les données et leur traitement;
- masquer les détails d'implémentation.

Quand la représentation d'un type est cachée, un mécanisme d'initialisation de variables de ce type est fourni par un constructeur.

Un exemple: les complexes

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

Le constructeur par défaut est `Compl()`

```
  Compl s, t;
```

définit deux complexes `s`, `t` initialisés à zéro.

```
  Compl s(3);
```

définit un réel, et

```
  Compl s(0,5);
```

définit un imaginaire pur.

Variantes possibles:

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

Un autre exemple: les rationnels

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

Dans notre implémentation, un rationnel est toujours réduit, et le dénominateur est toujours positif. D'où la nécessité d'une méthode de réduction.

```
class rat {
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();
  }
};
```

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

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 `operator`*op*`(`*types des opérandes*`);`

Par exemple

`rat operator+(rat,rat);`.

L'expression

`a+b`

équivaut à

`operator+(a ,b)`.

Un opérateur surchargé peut se définir

- au niveau global (comme ici)
- comme membre de classe

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);
}
```

Le moins unaire se définit par:

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

Les opérateurs d'entrée et sortie se surchargent également:

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

Les opérateurs d'entrée et de sortie prennent et retournent la *référence* à un flot en argument, pour éviter une copie.

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 << "J'ai lu: " << a <<" "<< b;
   cout << endl;
   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;
}
```

Ceci donne:

```
1 - 1/2 = 1/2
Donner deux rationnels : 2 5 8 4
J'ai lu: 2/5 2/1
-2/5
3/5
1/10
1/2

Resultat 9/2
```

Les patrons

Une classe *modèle* ou *patron*, en anglais *template* est une classe paramétrée par un type ou une classe. La syntaxe est illustrée sur l'exemple

```
template <class T> class Pile {
  private:
    T p[10];
    int sp;
  public
    Pile() {sp = 0;}
    void push(T x) { p[sp++] = x; }
    T pop(void) {return p[--sp]; }
};
```

On remarque

- Un nouveau mot-clé `template`;
- Le paramètre est entre chevrons;
- Définition de classe concrète par spécification d'un type.

On définit une pile d'entiers par

```
Pile <int> S;
```

et une pile de caractères par

```
Pile <char> S;
```