C++

Clément Royer

M1 Mathématiques et Applications - Parcours Mathématiques Appliquées

Version du 4 mai 2020

Lien vers la dernière version

Contenus thématiques

- Introduction et motivation;
- Premiers pas en C++ :
  - Exemple;
  - Variables, fonctions, opérateurs.
- Programmation impérative :
  - Pointeurs, gestion de la mémoire;
  - Références.
- Programmation orientée objet :
  - Les classes en C++;
  - L'héritage.
- Programmation générique :
  - Les patrons;
  - La surcharge des opérateurs.

Sommaire

A propos du langage C++

Historique

- Crée par Bjarne Stroustrup (AT& T labs) dans les années 1980, à partir du langage C. Nom originel : *C with classes*.
- Première norme en 1998, enrichissement majeur en 2011 avec C++ 11.
- Version courante (dialecte) : C++ 20, déployée en 2019.

Philosophie

- C++ ajoute des outils de programmation orientée objet à un langage procédural (C);
- Ce n'est pas un langage purement objet comme Java;
- C'est plus un langage objet que Fortran ou Basic.

Bibliographie

Ouvrages

- *The C++ programming language*, Bjarne Stroustrup (2013 pour la 4ième édition);
- *Programmer en C++*, Claude Delannoy (2019 pour la 10ième édition).

En ligne

- Wikibooks pour C++
- Tutoriel sur cplusplus.com
- https://stackoverflow.com/
  Attention à lire attentivement les discussions !

Pourquoi étudier le C++?

Palmarès IEEE 2019 :

- 4ème langage derrière Python, Java, et C (2e en 2018);
- 3ème pour les applications mobiles derrière Java et C;
- 3ème pour les applications embarqués derrière Python et C.

Logiciels basés sur C++ :

- Java Virtual Machine;
- Google Chrome, Firefox;
- Adobe Photoshop.

Pourquoi programmer en C++?

Du point de vue informatique

- Rapidité (compile en instructions natives);
- Fonctionnalités (classes, généricité);
- Particulièrement utilisé en systèmes embarqués.

En mathématiques appliquées

- Les prototypes de recherche sont souvent écrits dans des langages interprétés (Matlab, Python);
- Les codes les plus performants sont plutôt en Fortran ou C/C++, pour augmenter la rapidité et la puissance de calcul.

Exemples en optimisation numérique

IPOPT (`https://github.com/coin-or/Ipopt`)

- Développé par A. Wächter et L. Biegler depuis 2006;
- Algorithme de points intérieurs;
- L'un des meilleurs (le meilleur ?) solveur pour programmation non linéaire.

NOMAD (`https://www.gerad.ca/nomad/`)

- Développé à l'École Polytechnique de Montréal;
- Solveur pour l'optimisation sans dérivées, en C++;
- Utilisé dans l'industrie hydro-électrique.

Objectifs du cours

Comprendre les principes et spécificités du C++...

- Programmation procédurale, orientée objet, générique;
- Particularités du C++ par rapport à d'autres langages.

...et les mettre en pratique

- Séances sur machine;
- Restitution via le projet.

Sommaire

Écriture de programme et compilation

- Java/Python : langages interprétés, machine virtuelle;
- C++ : langage compilé, en lien avec la machine réelle;

Procédure basique (version Unix avec g++)

1. Écrire le programme dans un fichier (extension .cpp ou .cc);
2. Compilation du fichier en un exécutable :

```
g++ monfichier.cpp -o monfichier
```

   - -o résultat (*output*) de la compilation;
   - Ne produit rien en cas d'erreur.

3. Exécution de l'exécutable :

```
./monfichier
```

   - Peut produire des erreurs d'exécution.

Des erreurs classiques

Erreurs de compilation

- Parenthèse/Accolade manquante ou en trop;
- Erreur de syntaxe;
- Variable non déclarée.

Erreurs d'exécution

- Division par zéro;
- *Segmentation fault/Core dumped*.

Dans les deux cas, il faut corriger et recompiler !

Un premier programme ?

```
#include <iostream>
using namespace std;
int main(){cout << "Hello world";return 0;}
```

Un premier programme

```
/* Un premier programme en C++
   Version 0, 2020.01.20 */

#include <iostream>
using namespace std;

// Code principal
int main()
{
  cout << "Hello world";
  return 0;
}
```

Deux règles de base

1. Commenter (avec `/*...*/` ou `//...`);
2. Indenter.

Architecture d'un programme C++

Corps du programme

- Fonction `main` : Programme principal, qui sera exécuté;
- À l'intérieur :
  - Des blocs, délimités par des accolades {...};
  - Des instructions, terminées par des points-virgules.

En général

- D'autres fonctions au sein du programme;
- Ou ailleurs (modules).

Retour sur le programme

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

// Code principal
int main()
{
  cout << "Hello world";
  return 0;
}
```

Sommaire

Avant le main

Pré-traitement (*preprocessing*)

- `#include` : Importe des librairies d'autres fichiers;
- `using namespace` : S'affranchit d'un préfixe.

```
#include <iostream>

std::cout<<"Hello";
```

```
#include <iostream>
using namespace std

cout<<"Hello";
```

Entrées/Sorties

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

char c;

cout<<"Entrez un caractere :";

cin>>c;

cout<<"Le caractere est"<<c<<endl;
```

Ecriture

- `cout` : sortie standard, affiche des messages à l'écran;
- « est un opérateur;
- `endl` (ou `'\n'`) désigne la fin d'une ligne.

Entrées/Sorties

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

char c;

cout<<"Entrez un caractere :";

cin>>c;

cout<<"Le caractere est"<<c<<endl;
```

Lecture

- `cin` : entrée standard (lecture clavier);
- » opérateur;
- Pour stocker ce qui est lu, on utilise une variable.

Sommaire

Les types en C++

Types de base

- `int` Entier relatif;
- `float` Nombre réel (en écriture flottante);
- `double` Réel en double précision arithmétique;
- `char` Caractère ASCII (`'a'`, `'A'`,`'+'`,...);
- `bool` Valeur logique (`true`, `false`).

- C++ est fortement typé : toute variable devra avoir un type.
- La mémoire allouée à la variable dépendra de ce type.
- L'opérateur `sizeof` permet de connaître cette taille.
  Ex) `sizeof(int)` renvoie 2 (octets).

Déclarations

- Toute variable doit être déclarée avant d'être utilisée;
- Le type d'une variable est défini lors de la déclaration.

Exemples

```
// Declaration
int i;
// Declaration et initialisation
int j=0;
// Declaration d'une variable immuable
const int MAXITS=2000;
```

Déclarations (2)

Déclarations multiples

```
int i,j,k;
int i=j=1; // Affecte 1 a i et j
```

Variante d'initialisation

```
int n {1}; // Equivalent a int n=1
int n {};// Correspond a la valeur nulle

// Attention au typage
int n = 3.4; // Correct (n recoit 3)
int n {3.4}; // Rejet lors de la compilation
```

Sommaire

Opérateur d'affectation

- L'opérateur binaire = permet d'affecter;
- Ex) Initialisation d'une variable

```
int n=3;
```

- **lvalue** : tout ce qui peut être à gauche d'un =;
- Notion plus générale que celle de variable.

Opérations de base

Opérations arithmétiques : +,-,*,/,% (division euclidienne)

- **Le résultat et la validité de l'opération dépendent du type.**

```
int a=1/2; //a vaut 0
float b=1.0/2.0;
int c=a+b;
float c=a+b;
```

Opérations de base

Opérations arithmétiques : +,-,*,/,% (division euclidienne)

- **Le résultat et la validité de l'opération dépendent du type.**

```
int a=1/2; //a vaut 0
float b=1.0/2.0;
int c=a+b;
float c=a+b;
```

Affectation élargie

- Incrémentation, décrémentation;
- Evite de répéter la variable.

```
int a=1;
a=a+1;
```

```
int a=1;
a++;
```

```
int a=1;
a+=1;
```

Opérateurs de comparaison

Opérateurs booléens

==,!=,>,>=,<,<=

```
int a=3;
int b=3;
char c='t';
a==b;// Renvoie true
a!=b;//
bool cu = c=='u';// cu recoit false
```

Sommaire

Instruction de choix : if

```
int a = 3,b;

// Instruction conditionnelle
if (a>2){
   b=0;
}
else{
   b=2;
}
```

Un if en une ligne

```
int a=3;
int b= (a>2) ? 0 : 2;
```

Instruction de choix : `if` (2)

```
int a = 3;
if (a>2){
  if (a){
    // Effectue si a est non nul
  }
  else{
    // Effectue si a est nul
  }
}
```

- Importance de l'indentation;
- Pas besoin de `{...}` pour un `if/else` avec une seule instruction.

Instruction switch

```
int n;
cin >> n;

switch(n){
   case 0 : cout<<"=0";
   case 1 : cout<<"<=1"; break;
   case 2 :
   default : cout<<">=2";
}
```

- Gestion de possibilités multiples;
- `break` : instruction de branchement.

Boucle for

```
int i;
for(i=0;i<5;i++){
  cout<<"Iteration "<<i<<" de la boucle\n";
}
```

- Boucle infinie : `for( ; ; )` (on en sort via `break`);
- L'indice peut être initialisé dans la clause `for`:

```
for(int i=0;i<5;i++){
  cout<<"Iteration "<<i<<" de la boucle\n";
}
```

Boucles while and do...while

```
int i=0;

while (i<100){
  i++;
}
```

```
int i=0;
do{
  i++;
}
while (i<100);
```

- Deux constructions différentes;
- `while(true)` : boucle infinie (peut être combinée avec `break`).

Sommaire

Fonctions

Structure d'un programme

- Programme principal `main` : c'est une fonction;
- Il peut y avoir d'autres fonctions dans un même fichier;
- On peut utiliser des fonctions d'autres fichiers.

Paradigme de programmation procédurale

- Fonction : un bloc d'instructions que l'on peut utiliser plusieurs fois;
- L'exécution peut changer selon les paramètres de la fonction.

Syntaxe

```
type_retour nomfonction([type1 arg1,type2 arg2,...])
```

Exemple

```
float mafonction(int arg1, float arg2)
{
  //Code de la fonction
  return arg2;
}
```

- Deux arguments locaux : le type doit être déclaré;
- Type de retour + instruction `return`.

Fonction sans argument

```
void fonctionrien() { }
```

La fonction main

```
int main()
{
  // Code principal
  return 0;
}
```

- Pas de `return` $\Leftrightarrow$ `return 0` en dernière instruction.
- En pratique, différentes valeurs (convergence, budget dépassé, etc).

Fonctions dans un programme

```
// Declaration
float saxpy(float,float,float);

int main (){
  float a=1.0,x=2.0,y=4.0;
  float z=saxpy(a,x,y);
  return 0;
}

// Code de la fonction
float saxpy(float a, float b, float c){
  float val;
  val = a*b+c;
  return val;
}
```

Fonctions dans un programme (2)

```
// Declaration
float saxpy(float a, float b, float c);

int main (){
  float a=1.0,x=2.0,y=4.0;
  float z=saxpy(a,x,y);
  return 0;
}

// Code de la fonction
float saxpy(float a, float b, float c){
  return a*b+c;
}
```

Arguments et valeurs par défaut

```
int incrementer(int,int=1);

int incrementer(int n, int p){
  return n+p;
}

int main(){
  int n=2;
  n=incrementer(n,2);
  int p=incrementer(n);
}
```

- Les valeurs par défaut sont fixées lors de la **déclaration**;
- Les arguments fixés par défaut doivent être les **derniers de la liste des arguments**.

Variables locales

```
void incr(int a){ a++;}

int main()
{
   int a=3;
   incr(a);
}
```

- La variable `a` n'est pas modifiée !
- Passage par valeur : La fonction `incr` fait une copie locale de la variable et agit sur cette copie.

Variables globales

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

int i; // Une variable globale
void printi();

void printi()
{
  cout<<"Valeur de i : "<<i<<endl;
}

int main()
{
  for(i=1;i<=5;i++) printi();
}
```

Variables globales

Principe

- Variable déclarée en dehors du `main`;
- Connue de toutes les fonctions du programme, et modifiable.

Remarque

- On peut vouloir modifier certaines variables et pas d'autres au sein de la même fonction;
- On préfèrera donc le passage par référence aux variables globales.

Variable locale VS globale

```
int n; // Variable globale
void myfonc()
{
  int n=1; // Variable locale
  ::n = 2;
  n = 3;
}
```

- Le `::` permet de comprendre l'espace de noms (`namespace`) dans lequel on se place;
- Par défaut, on considère la variable locale.

Conclusion

A retenir

- Typage, déclaration : essentiels en C++ !
- Structure de fichiers (`main`+fonctions);
- Instructions de base.

A pratiquer (cf TP1)

- Lecture/écriture;
- Déclarations, initialisation;
- Boucles, conditionnelles;
- Syntaxe et appels de fonctions.

Sommaire

Résumé

Ce que nous avons vu

- Types et opérateurs de base;
- Fonctions.

Vers des aspects de programmation procédurale

- Tableaux, chaînes de caractères (version `C`);
- **Pointeurs**;
- Structures.

Sommaire

Les tableaux

Définition, déclaration

- Un tableau est un ensemble d'éléments de même type;
- Un tableau peut avoir plusieurs dimensions.

```
float tabflot[3]={1.0,2.0,3.0};
int matrice[10][15];
```

Accès aux éléments d'un tableau

- Les indices sont entiers, commencent à 0;
- Pas de contrôle d'indice dans les compilateurs !
- Pour les tableaux multi-dimensionnels :

```
matrice[0][0],matrice[0][1],...
```

Initialisation

```
float tab1[]={1,2,3};// Tableau de taille 3
float tab2[3]=4;
float mat1[2][3] = {{1,2,3},{4,5,6}};
float mat2[2][3] = {{1,2},{3,4,5}};
```

- On peut omettre des éléments;
- Affectation globale impossible.

Un tableau particulier : la chaîne de caractères

Chaînes en C++ :

- Utilisées directement : `"Hello world"`;
- Le type/la classe `string`;
- Comme un tableau de caractères (`C`).

Chaîne

```
// Une chaine de longueur 8
char chainehello[]="bonjour";
char cfin=chainehello[7];//cfin vaut '\0'
```

Tableaux et fonctions

- On peut passer un tableau en entrée d'une fonction;
- Les valeurs du tableau seront modifiées.

```
/*  Declaration de fonction avec
  tableau en argument */
void fct(char tab[10]);
void fct(char tab[]);
```

- Pas besoin de préciser la taille du tableau;
- Seule importe l'adresse du tableau.

Sommaire

Motivation

```
#include <iostream>
void carre(int n){ n=n*n;}

int main()
{
   int i=3;
   carre(i);
   std::cout<<i;
}
```

- Variable `i` non modifiée;
- Déclarer `i` comme variable globale ne conviendrait pas.

- Passer la valeur en paramètre;
- Passer la variable en paramètre.

Passage par référence en C++

```
#include <iostream>
void carre(int &n){ n=n*n;}

int main()
{
   int i=3;
   carre(i);
   std::cout<<i;
}
```

- L'appel n'a pas changé;
- On déclare la variable d'entrée comme modifiable $\Rightarrow$ passage par référence.

Mémoire

- Décomposition en blocs identifiés par des adresses;
- Un pointeur est une variable contenant une adresse;
- Une référence est une adresse correspondant à une partie de la mémoire.

Opérateurs

- Pointeur : opérateur *;
- Référence : opérateur &.

Manipulation de pointeurs

Allocation et comparaison

```
int n = 2;
int *p1 = &n;
int *p2 = &n;
bool b = (p1==p2); // b true
int *p3 = p2;//Pointeurs de meme type
```

Pointeur nul (C++11)

- Pointeur sans adresse;
- À l'origine, 0 ou une constante `NULL`.

```
int *p = nullptr;
int *q = 0;
```

```
int n;
int *an = &n;
```

- `&n` contient l'adresse de la variable `n`;
- `&n` ne peut pas être modifiée.

Pointeur et référence

- On peut incrémenter une adresse stockée dans un pointeur :

```
int n=3;
int *an = &n;
an++;
```

- Pas nécessairement cohérent avec le contenu mémoire !

Pointeurs comme argument d'entrée

```
void fun(int *p){
  *p = 4;
}
int main(){
  int a=3;
  fun(&a);
  // a== 4
}
```

Un exemple issu du TP 2 (1/3)

Passage par valeur

```
void ajouter(int a,int b){ b+=a;}
int main(){
  int a=3,b=0;
  ajouter(2*a,b);
}
```

- Avant `ajouter` : `a`=3, `b`=0;
- Après `ajouter` : `a`=3, `b`=0.

- Copie locale des variables à l'appel de la fonction;
- Destruction de ces variables après appel.

Un exemple issu du TP 2 (2/3)

Passage par référence (propre à C++)

```
void ajouter(int a,int & b){ b+=a;}
int main(){
   int a=3,b=0;
   ajouter(2*a,b);
}
```

- &b est l'adresse de la variable b en mémoire;
- Avant ajouter : a=3, b=0;
- Après ajouter : a=3, b=6.

Un exemple issu du TP 2 (3/3)

Passage par pointeur (dans le style de C)

```
void ajouter(int a,int * b){ *b +=a;}
int main(){
   int a=3,b=0;
   int *p=&b;
   ajouter(2*a,p);
}
```

- p est un **pointeur** sur la variable n;
- C'est une variable dont la valeur est l'adresse de n;
- Avant ajouter : a=3, b=0;
- Après ajouter : a=3, b=6.

Derrière la notion de tableau...

- Un **pointeur constant** vers le premier élément !
- Un système d'indexation.

```
int tab[10];
int *p1 = &tab[0];
p1++; // p1=&tab[1];
```

```
void fun(int *t,int n){
  for(int i=0;i<n;i++){
    *t = i;
    t += 1;
  }
}
```

Parcours de tableau via un pointeur

```
int t[10];
int *pt = &t[0];
// Parcours 1 du tableau
for(int i=0;i<10;i++){
   *pt=0;
   pt++;
}
// Parcours 2 du tableau
for(int *p=t;p<t+10;p++){
   *p=1;
}
```

- Les pointeurs vers des tableaux sont utiles pour faire des boucles;
- Leur incrémentation a du sens car les entrées du tableau sont contiguës en mémoire.

Gestion dynamique de la mémoire

- **Principe :** Réserver et libérer des emplacements mémoire durant l'exécution du programme;
- Différence avec la gestion statique vue jusqu'à présent.

Allocation

```
int *tab = new int[10];
float *pr = new float;
```

Libération

```
delete [] tab;
delete pr;
```

- S'applique à un élément dont la mémoire a été allouée avec `new` et n'a pas encore été libérée;
- Le comportement est non défini sinon.

Sommaire

Types avancés en C++

- On peut aller au-delà des types et structures de données basiques...
- ...mais on touche vite aux concepts de la programmation orientée objet.

Deux exemples

- Types énumération;
- Types structurés (`struct`).

Les énumérations

- Cas particulier de type entier;
- Nombre fini de valeurs (constantes) du type;
- Chaque constante correspond à une valeur entière.

```
enum departement {LSO,MIDO,MSO};
int n=LSO; // Equivaut a n=0
```

- Les constantes sont valides pour tout un bloc :

```
enum couleur1 {rouge,bleu,vert};
// rouge est deja defini, l'instruction
// ci-dessous genere une erreur
// a la compilation
enum couleur2 {rouge,jaune,orange};
```

Le constructeur struct

Un constructeur venant du C

- Permet d'aggréger plusieurs variables de types différents, appelées champs;
- À la déclaration, réserve un emplacement mémoire pour chaque variable contenue dans le `struct`.

```
struct etudiant{
  int numero_etu;
  float note_cpp;
};
```

Manipulation de structures

Accès aux champs, initialisation

```
// Declaration
etudiant etu1;
// Initialisations
etu1.numero_etu=122; //Partielle
etudiant etu2={12,19.5};//Totale
```

Aspects avancés

Une structure définit un type, qui peut apparaître dans

- Des tableaux;
- Des arguments d'entrée/de sortie d'une fonction;
- Des pointeurs.

Structures et pointeurs vers la structure

- Une structure **ne peut pas** avoir de champ du même type que la structure!
- Elle **peut** en revanche avoir un pointeur sur un élément de ce type.

```
struct etudiant{
  int numero_etu;
  float note_cpp;
  etudiant *parrain;
}
```

- La variable `parrain` est une adresse : on connaît sa taille en mémoire.

Structures et fonctions membres

```
struct etudiant{
  int numero_etu;
  float notes[20];
  //Declaration d'une fonction
  float calcul_moyenne();
};
```

- `calcul_moyenne` s'appelle une **fonction membre**;
- Le code de la fonction sera écrit ailleurs.

Écriture de la fonction membre

```
float etudiant::calcul_moyenne(){
  int s=0;
  for(float *p=notes;p<notes+20;p++){
    s+=*p;
  }
  s/=20;
  return s;
}
```

- Préfixe `etudiant::` essentiel;
- Permet la surcharge d'opérateurs : `calcul_moyenne` peut être définie pour d'autres structures.

Le type string

- Libraire dédiée (incluse dans iostream)

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

- En toute rigueur un type classe.

Déclaration, initialisation

```
string ch1;
cout<<ch1;//N'imprime rien
ch1="hello";
ch1={'h','e','l','l','o'};
string ch2 (10,'a');
```

Opérations et fonctions pour le type `string`

```
string ch;
unsigned int n=ch.size();
bool b=ch.empty();
```

- La taille d'une variable `string` varie au cours de l'exécution !

Opérations et fonctions pour le type string

Affectation, concaténation

```
string ch1="hello", ch2="world";
string ch=ch1+ch2;
ch=ch1+" !";

// Instruction invalide
ch="hello"+"world";
```

Accès aux éléments

```
char c = ch[0];
char c2 = ch[1]++;//Caractere ASCII suivant
```

Comme pour les tableaux :

- Caractères consécutifs en mémoire;
- Pas de contrôle d'indice.

Boucles de chaîne

```
// Boucle de parametre entier
for(unsigned int i=0;i<ch.size();i++){ }
```

Variante (C++11)

```
// Avec copie des caracteres
for(char c:ch){ }

// Sans copie des caracteres
for(char &c:ch){ }
```

Conclusions : Aspects procéduraux du C++

Points-clés

- Passage par valeur/référence/pointeurs;
- Pointeurs, tableaux.

Vers des aspects objet

- Les structures;
- Le type `string`.

Mise en pratique

- Exercices;
- TP 2 (Pointeurs) et TP 3 (Structures).

Sommaire

Sommaire

Les structures

```
struct etudiant{
  int promo;
  etudiant *parrain;

  void change_dpt(dptDauphine);
};
```

- Accès direct aux champs :

```
etudiant e;
int n=e.promo;
```

- Possible de définir des fonctions à l'intérieur des structures en C++ (mais pas en C !).

Qu'est-ce qu'une classe ?

- Un type particulier (généralement défini par l'utilisateur);
- Associé à des attributs (**membres données**) et des méthodes (**fonctions membres**);
- En programmation orientée objet "pure", les attributs sont **privés**, càd accessibles seulement au sein de l'objet $\Rightarrow$ **Principe d'encapsulation**

Classe VS Structure

- Tout est public dans une structure, en particulier les champs;
- Pas de respect du principe d'encapsulation dans une structure.

Déclaration d'une classe (version naïve)

```
class Point{
    // Membres donnees
    float x;
    float y;

    // Fonctions membres
    void init(float,float);
};
```

- Par défaut, les membres d'une classe sont en accès privé (`private`);
- **Principe d'encapsulation :** les membres *données* doivent être privés;
- En général, on veut que l'utilisateur ait un accès indirect aux membres données via les fonctions membres.

Déclaration d'une classe

```
class Point{
  private:
    float x;
    float y;
  public:
    void init(float,float);
};
```

- Ici les membres données sont privés, les fonctions membres sont publiques.
- On peut combiner plusieurs déclarations `private` et `public`.

Utilisation d'une variable de type classe

```
class Point{
  float x;
  float y;
  public:
    void init(float,float);
};

Point p;
p.init(3.0,3.0);
```

- `p` est une instance/un objet de la classe `Point`;
- Impossible de faire `p.x` car le membre donnée `x` est privé;
- On peut par contre appeler la fonction membre, et celle-ci peut accéder aux membres données !

Définition des fonctions membres

- Après la déclaration, écriture du code des fonctions membres à part;
- Utilisation d'un préfixe correspondant à la classe.

```
void Point::init(float abs,float ord){
  x=abs;
  y=ord;
}
```

- Erreur de compilation si la classe `Point` n'est pas connue (définie plus haut ou importée);
- Une fonction membre reçoit un argument implicite du type classe associé et a accès à tous ses membres (privés comme publics).

Classe et composant logiciel

Comment (ré)-utiliser le code de la classe `Point` *?*

- Créer un fichier en-tête `Point.h` contenant la déclaration de la classe, à importer dans le fichier principal :
- Compiler le fichier classe `Point.cpp` contenant les définitions de la classe en un fichier `Point.o` :

```
g++ -c Point.cpp
```

- Tout fichier peut alors utiliser `Point` en ayant **uniquement** les fichiers `.h` et `.o` :
  - Dans le fichier principal :

```
// Fichier testPoint.cpp
#include "Point.h"
```

  - Compilation en un exécutable :

```
g++ -c testPoint.cpp
g++ -o testPoint Point.o testPoint.o
```

Sommaire

Création et destruction d'un objet de type classe

- Approche objet : via une fonction membre;
- Peut poser des soucis en termes d'allocation mémoire;
- En Java, le "ramasse-miettes" aide à cela.

En C++

RAII: Resource Acquisition Is Initialization

- Système de constructeur et destructeur;
- Le constructeur acquiert les ressources;
- Le destructeur les libère.

Constructeurs

```
class Point{
  float x;
  float y;
  Point(float);
  Point(float,float);
};
```

- Porte le même nom que la classe;
- Fonction appelée immédiatement après la création d'un objet;
- Absence de constructeur = Constructeur par défaut;
- Le constructeur peut être **surdéfini** avec des arguments différents en type et en nombre.

Exemples de constructeurs pour la classe Point (1/2)

```
class Point{
  float x;
  float y;
  Point(float,float);
};

//Code basique du constructeur
Point::Point(float a,float b){
  x(a);
  y(b);
}
```

Utilisation

```
Point p (3.0,3.0);
```

Exemples de constructeurs pour la classe Point (2/2)

```
class Point{
  float x;
  float y;
  Point(float=0,float=0);
};

//Code basique du constructeur
Point::Point(float a,float b) : x(a),y(b) {}
```

- Valeurs par défaut dans le constructeur;
- Écriture simplifiée du constructeur pour les initialisations.

```
Point p;
Point q (2.0,1.0);
```

Constructeur par défaut (1/2)

- Lorsqu'aucun constructeur n'est défini;
- Peut suffire si la classe n'implique pas de gestion dynamique de mémoire.

```
class Point{
  float x,y;
  public:
    void init(float,float);
};

void Point::init(float a,float b){
  x=a;y=b;
}
```

Constructeur par défaut (2/2)

Utilisation

```
// Appel au constructeur par
// defaut
Point p;
// Initialisation via une
// fonction membre
p.init(2.0,3.0);
```

Destructeur

Propriétés

- Nom : $\sim$ + le nom de la classe;
- Appelé automatiquement avant la destruction de l'objet;
- Ne prend pas d'argument en entrée (et ne renvoie rien);
- Inutile en général si le constructeur se contente d'initialiser des objets de manière statique.

Exemple de classe avec allocation dynamique

```
class Tableau{
    int taille, *vec;
    public:
      Tableau(int);
      ~Tableau();
};

Tableau::Tableau(int t){
  taille = t;
  vec = new int[taille];
}

Tableau::~Tableau(){
  delete vec;
}
```

Constructeurs et passage par valeur

- Constructeur : appelé à chaque création d'objet;
- Passage par valeur d'un argument d'une fonction : crée une **copie** de l'argument;
- **Comment gérer la copie d'un objet ?**

Problématique

- Passage par valeur d'un argument de type classe
  $\Rightarrow$ Copie de la variable et de ses champs;
- Peut poser des soucis en cas d'allocation dynamique :
  - Seuls les pointeurs seront recopiés;
  - Risque de double suppression des données.

Constructeur de recopie (1/2)

Définition

- Syntaxe pour la classe `MaClasse` :

```
MaClasse(MaClasse &);
```

- Syntaxe recommandée (permet certaines affectations) :

```
MaClasse(const MaClasse &);
```

Constructeur de recopie par défaut

- Effectue une copie de chacun des champs;
- Problème en allocation dynamique : copie un pointeur mais pas ce vers quoi il pointe.

Constructeur de recopie (2/2)

```
class Tableau{
  int taille, *tab;
  public:
    // Constructeur
    Tableau (int);
    // Constructeur de recopie
    Tableau (const Tableau &);
};

Tableau::Tableau(int n):taille(n),tab(new int[n]){}

Tableau::Tableau(const Tableau &t){
  taille = t.taille;
  tab = new int[taille];
  for(int i=0;i<taille;i++){ tab[i]=t.tab[i];}
}
```

Sommaire

Le mot-clé this

- Les fonctions membres connaissent implicitement l'objet les appelant;
- `this` désigne un pointeur sur cet objet.

```
class Point{
  float x,y;
  public:
    bool memePt(Point *);
};

bool Point::memePt(Point *p){
  return this==p;
}
```

Membres données statiques

```
class Point{
  float x,y;
  static int nb_points;
}
```

- `static` : commun à toutes les instances de la classe;
- Initialisation hors déclaration et hors définition du constructeur :

```
int Point::nb_points=0;
```

- Attention : un membre statique est privé par défaut.

Fonctions membres statiques (1/2)

```
// Fichier .h
class Point{
  float x,y;
  static int nb_points;

  public:
    static int compte_points();
};
// Fichier .cpp
int Point::nb_points=0;
int Point compte_points(){
  return nb_points;
}
```

Fonctions membres statiques (2/2)

Utilisation d'une fonction `static`

```
int main(){
  int n=Point::compte_points();
}
```

- Peut être appelée même sans variable `Point` définie !

Sommaire

Passage d'une variable d'un type classe en argument

- Par valeur (défaut) :

```
void ptzero(Point p){p.init(0,0);}
```

- Par référence :

```
void ptzero2(Point &p){p.init(0,0);}
```

- On peut définir un pointeur sur une variable de type classe;
- Opérateur -> : remplace (*).

```
void ptzero3(Point *p){(*p).init(0,0);}
Point p;
Point *q=&p;
ptzero3(q);
q->init(0,0);
```

Mot-clé const et classes

Pour les arguments

```
void mafonction(const Point &);
```

- Garantit que l'argument ne sera pas modifié;
- Si fonction membre de la classe `Point`, l'objet appelant peut toujours être modifié.

Pour les fonctions membres

```
void Point::mafonction() const;
```

- Permet de l'appliquer aux objets constants;
- Pour les méthodes qui ne modifient pas les données membres.

Gestion dynamique des objets (1/2)

Sans constructeur

```
class Point{
  float x,y;
};

int main(){
  Point *adPoint = new Point;
  delete adPoint;
}
```

Gestion dynamique des objets (2/2)

Avec constructeur(s)

```
class Point{
  float x,y;
  public:
    Point (float,float);
};

int main(){
  Point *adPoint = new Point(1.0,2.0);
  delete adPoint;
}
```

Objets membres et concepts associés

Objet membre

- **Déf :** un membre donnée d'une classe du type d'une autre classe.
- Possède donc ses propres membres données et fonctions membres.

Problèmes d'objets membres

- Accès aux membres données;
- Constructeurs.

Fonctions amies

- Déclarées dans la classe via le mot-clé `friend`;
- Peuvent accéder aux membres privés d'une classe.
- Version indépendante :

```
class Tableau{
  int taille, *tab;
  public:
    friend int somme(Tableau &);
};

int somme(Tableau &t){
  int res=0;
  for(int i=0;i<t.taille;i++){
    res+=t.tab[i];
  }
  return res;
}
```

Fonctions amies dans une autre classe

```
class Point;
class TabPoint{
  int taille;
  Point *tab;
  public:
    Point bary();//Moyenne des points
    Point premier();//Renvoie tab[0]
};
class Point{
  float x,y;
  friend Point TabPoint::bary();
};
```

- Le compilateur doit reconnaître la classe `Point`;
- Mais la déclaration de `TabPoint` doit s'effectuer avant celle de la classe `Point` !

Classe amie

```
class Point;
class TabPoint{
  int taille;
  Point *tab;
  public:
    Point bary();//Moyenne des points
    Point premier();//Renvoie tab[0]
};
class Point{
  float x,y;
  friend class TabPoint;
};
```

- Toutes les fonctions membres d'une classe amie sont des fonctions amies.

Constructeur et objets membres (1/2)

Contexte : un objet membre possède un constructeur.

```
class Point{
  float x,y;
  Point(float,float);
};

class Cercle{
  float rayon;
  Point centre;
  Cercle(float,float,float);
};
```

Constructeur et objets membres (2/2)

Règle : le constructeur principal doit spécifier les arguments du constructeur de l'objet membre.

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

Cercle::Cercle(float a, float b, float c){
  rayon=a;
  centre = Point(b,c);
}
```

Tableaux dynamiques d'objets (1/2)

```
class TabPoint{
  Point *tab;
  int taille;

  public:
    TabPoint(int);
    ~TabPoint();
};
```

Tableaux dynamiques d'objets (2/2)

Construction

- Instruction d'allocation dynamique :

```
TabPoint::TabPoint(int t){
  taille=t;
  tab = new Point[t];
}
```

- La classe `Point` doit comporter un constructeur **sans argument** (ou aucun constructeur), sinon erreur de compilation !

Destruction

```
TabPoint::~TabPoint(){
  delete [] tab;
}
```

Sommaire

Surdéfinition

Principe

- C++ autorise les définitions multiples de fonctions en changeant le nombre/le type d'arguments;
- Dans une classe, possible pour toutes les fonctions sauf le destructeur.

```
class Point{
  float x,y;
  public:
    Point();
    Point(float);
    Point(float,float);
};
```

Surdéfinition des opérateurs

- **Idée :** Pouvoir utiliser les opérateurs classiques sur des types classe;
- Certains sont définis par défaut : `=`, `->`, `new`, `delete`.
- Exemples d'opérations "surdéfinissables" :
  - `+,-,*,/,++,--`
  - `=,+=,-=,==,&&`
  - `>>,<<`
  - `(),[],->`
  - `new`, `delete`.

Deux façons de surdéfinir

- Via une fonction indépendante (en général amie);
- Via une fonction membre.

Exemple de surcharge d'opérateur via une fonction indépendante

```
class Point{
  float x,y;
  public:
  Point(float,float);
  friend Point operator+(Point p1,Point p2);
};

Point operator+(Point p1,Point p2){
  Point p (p1.x+p2.x,p1.y+p2.y);
  return p;
}
```

- `p=p1+p2;` est interprétée comme

```
p = operator + (p1,p2);
```

Exemple de surcharge d'opérateur via une fonction membre

```
class Point{
  float x,y;
  public:
  Point(float,float);
  Point operator+(Point);
};

Point Point::operator+(Point p2){
  Point p (x+p2.x,y+p2.y);
  return p;
}
```

- `p=p1+p2;` est interprétée comme

```
p = p1.operator+(p2);
```

Surcharge et opérateur d'affectation

L'opérateur = pour les classes

- **Par défaut :** recopie les valeurs des champs de la seconde opérande dans ceux de la première;
- Difficulté : gestion de parties dynamiques des objets;
- Souhaitable : renvoi d'une valeur pour traiter les affectations multiples du type `a=b=c;`

Comment le redéfinir ?

- Nécessairement comme fonction membre;
- Distinguer le cas de deux opérandes identiques.

Exemple pour la classe Tableau (1/2)

```
class Tableau{
  int taille, *tab;

  public:
    // Constructeur
    Tableau(int);
    // Constructeur de recopie
    Tableau(const Tableau &);
    // Surcharge de l'operateur =
    Tableau & operator=(const Tableau &);
};
```

- `Tableau &` en type de retour : évite une recopie de la variable;
- Mot-clé `const` optionnel mais recommandé car permet certaines affectations souhaitables.

Exemple pour la classe Tableau (2/2)

```
Tableau & Tableau::operator=(const Tableau &t){
  if (this!=&t){
    delete tab;
    taille = t.taille;
    tab = new int[taille];
    for(int i=0;i<taille;i++)
      tab[i] = t.tab[i];
  }
  return *this;
}
```

Forme canonique d'une classe

- Classe avec pointeurs/parties dynamiques
  $\Rightarrow$ redéfinition des constructeurs, destructeur, =
- Décrit la **forme canonique** de la classe.

Exemple sur la classe Tableau

```
class Tableau{
  int taille,*tab;

  public:
    Tableau(int);
    Tableau(const Tableau &);
    ~Tableau();
    Tableau & operator=(const Tableau &);
};
```

Surcharge de l'opérateur []

Pour la classe Tableau

- Idée : accéder au i-ème élément du tableau sans passer par le membre donnée;
- Définition sous forme de fonction membre;
- Renvoie une référence pour pouvoir changer la valeur.

```
class Tableau{
  int taille, *tab;
  public:
  int & operator [](int);
};

int & Tableau::operator[](int i){
  return tab[i];
}
```

Surcharge de l'opérateur ()

- Principe : permettre aux objets d'être utilisés comme une fonction;
- A distinguer de l'appel à un constructeur.

Exemple : Modifiation d'un élément dans la classe `Tableau`

```
class Tableau{
  int taille,*tab;
  public:
    Tableau(int);
    float operator() (int,float);
};

Tableau t1(3);
t1(3,2.0);
```

Autres surcharges d'opérateurs

Surcharge de `new` et `delete`

- Impact uniquement sur les objets alloués dynamiquement;
- Fonctions membres statiques;
- Appels suivis du constructeur ou destructeur.

Remarques générales

- Indépendance des opérateurs : surcharger + et = ne surcharge pas +=;
- Ne pas surdéfinir à tout prix : penser à la facilité d'interprétation de l'opérateur.

Sommaire

Le concept d'héritage simple

- Définir une classe à partir d'une classe existante;
- La première est dite **classe dérivée**, la seconde **classe de base**.

```
class Point{
  float x,y;
  public:
  void affiche() const;
};

class PointNomme : public Point
{
  public:
    void affichenom() const;
  private:
    char nom;
};
```

Classe dérivée

```
class PointNomme : public Point { ... };
```

- `PointNomme` dérive de `Point`;
- Les membres `public` de la classe `Point` seront des membres `public` de la classe dérivée;
- Les membres privés de `Point` ne seront **pas** accessibles dans la classe `PointNomme` (encapsulation).
- Les déclarations d'amitié ne s'héritent pas.

Fonctions membres et héritage (1/2)

```
void Point::affiche(){ cout<<x<<y<<endl;}

void PointNomme::affichenom(){
  cout<<"Point "<<nom<<" :";
  affiche();
}
```

- Appel à `affiche` possible ici car `public`;
- L'utilisateur d'un objet `PointNomme` peut appeler `affiche` ou `affichenom`.

Fonctions membres et héritage (2/2)

```
void Point::affiche(){ cout<<x<<y<<endl;}

void PointNomme::affiche(){
  cout<<"Point "<<nom<<" :";
  Point::affiche();
}
```

- Redéfinition : l'utilisateur d'un objet `PointNomme` ne peut appeler que `PointNomme::affiche`.

Constructeur et héritage (1/2)

```
class A;
class B : public A;
```

- Si `A` dispose d'un constructeur, pas besoin de l'appeler dans le constructeur de `B`;
- Mais si besoin, on peut passer les arguments de constructeur à constructeur.

Constructeur et héritage (2/2)

```
class Point{
  public:
    Point(float,float);
};

class PointNomme{
  public:
    PointNomme(float,float,char);
};

PointNomme::PointNomme(float a,float b,char c)
: Point(a,b), nom(c){ }
```

Le statut protégé (protected)

Membres protected

```
class Point {
  protected:
    float x,y;
};
```

- Équivalent à `private` pour un utilisateur de la classe;
- Équivalent à `public` **pour les classes dérivées**.

Utilisation du statut protected

- Peut faciliter l'implémentation;
- Violation (contrôlée) du principe d'encapsulation.

Dérivations publique et privée

```
class B : public A;
```

- Dans la classe `B` : accès aux membres `public` et `protected` de `A`;
- Pour un utilisateur de `B` : accès aux membres `public` de `A`.

```
class B : private A;
```

- Dans la classe `B` : accès aux membres `public` de `A`;
- Pour un utilisateur de `B` : pas d'accès.

```
class B : protected A;
```

- Dans la classe `B` : accès aux membres `public` et `protected` de `A` mais tous prennent le statut `protected` dans `B`;
- Pour un utilisateur de `B` : pas d'accès.

Héritage et pointeurs

```
Point p (3,5);
PointNomme pn (6,3,'A');
Point *adp = &p;
PointNomme *adpn = &pn;

// Instructions valides
adp = adpn;
adp.affiche(); // Appelle Point::affiche
p=pn;//Slicing: perte d'une partie d'un objet
```

Constructeur de recopie et dérivation

Rappels

Constructeur de recopie appelé si :

- Initialisation d'un objet par objet de même type;
- Passage par valeur d'un objet en argument ou en retour de fonction.

- **Convention :** Si défini, le constructeur de recopie de la classe dérivée prend en charge l'intégralité de la recopie;
- On peut transmettre des arguments au constructeur de la classe de base;
- **Syntaxe :**

```
B (B & x) : A (x) {
  // ...
}
```

Dérivation publique et types

```
class B : public A;
```

Tout appel à une fonction membre de `A` par un objet de la classe `B` aura du sens :

- Toujours du type de retour défini dans `A`;
- Arguments du type imposé par la déclaration.

Si + est surchargé dans la classe `Point`:

- Interdit :

```
Point a,b;
PointNomme c=a+b;
```

- Autorisé :

```
PointNomme a,b;
Point c=a+b;
```

Sur l'héritage

- Dérivation/héritage en série possible;
- On obtient une arborescence en héritage simple;
- Avec l'héritage multiple, on obtiendra un graphe.

Héritage multiple

- N'existe pas en Java (remplacé par des interfaces);
- Pose plusieurs difficultés en termes de résolution de portée et de dérivations successives.

Héritage multiple

```
class Personne{
  string nom, prenom;
};
class Date{
  int mois,jour,annee;
};

class Naissance : public Personne, public Date
{
  //...
};
```

- L'ordre des classes de base est important;
- `public` peut être remplacé par `private` ou `protected`.

Ce qu'implique l'héritage multiple

Si la classe `C` dérive des classes `A` et `B` :

- Il peut y avoir des fonctions membres de même nom dans `A` et `B`!
  - Appels via `A::mafonction` et `B::mafonction`;
  - Redéfinition possible dans la classe `C` !
- Idem pour les membres données (on peut avoir `A::x` et `B::x`).

Constructeurs et héritage multiple

```
class A;
class B;
class C : public A, public B;
```

Si chaque classe dispose d'un constructeur,

- L'en-tête du constructeur devra respecter la déclaration :

```
C(int nA,int nB) : A(nA), B(nB)
```

- L'ordre d'appel des constructeurs sera alors : `A`, `B`, `C`.
  Remarque : les destructeurs éventuels sont appelés dans l'ordre inverse.

Classes virtuelles (1/2)

Contexte : héritages en cascade

```
class A;
class B : public A;
class C : public A;
class D : public B, public C;
```

- La classe `D` hérite "doublement" de la classe `A`;
- Si `x` est un membre donnée de `A`, on distinguera `B::x` et `C::x` dans `D`;
- Duplication des membres données de la classe `A` : deux objets de type `A` construits;
- Ordre d'appel des constructeurs à la construction d'un objet de classe `D` : `A,B,A,C,D`.

Classes virtuelles (2/2)

Mot-clé `virtual`

```
class A;
class B : public virtual A;
class C : public virtual A;
class D : public B, public C;
```

- Permet de n'incorporer qu'une seule fois les membres données de `A` dans la classe `D`;
- Ne change rien pour les classes `B` et `C`;
- Le constructeur de `D` précisera les informations destinées à `A`:

```
D(int iA,int iB,int iD) : B(int iB),A(int iA)
```

- Les constructeurs de `B` ou `C` ne devront pas préciser des informations pour `A`;
- `A` devra avoir un constructeur sans argument (ou par défaut).

En bref

Les classes en C++

- Membres données, fonctions membres;
- RAII : constructeurs et destructeurs;
- Première mise en pratique dans le TP 4.

Aspects avancés

- Fonctions/Classes amies, surcharge d'opérateurs (TP 5);
- Héritage (TP 6).

Sommaire

Ce que nous avons vu

- Des types de base : `int`,`char`,etc;
- Des types structurés;
- Des types classes.

- On peut surdéfinir des opérateurs/des fonctions pour chaque type;
- Ou on peut penser plus générique !

Sommaire

Patrons de fonctions

- **Principe** : écrire une fonction sans préciser le type de ses paramètres;
- Synonyme de fonction générique ou de modèle de fonction;
- Mots-clés utiles :
  - `template` : déclare cette fonction comme un **patron** de fonctions;
  - `typename` : identifie un paramètre de type (`class` est aussi valide).

```
template <typename T> T minimum(T a,T b){
  return a > b ? b : a;
}
```

- Peut être appelée pour tout type `T` tel que l'opérateur > est défini;
- `T` peut être un type standard ou un type classe.

Utilisation d'un patron de fonctions

Déclaration et définition

- La définition d'un patron de fonctions est vue comme une déclaration;
- Pour utiliser des déclarations de patrons au sens usuel, il faut employer le mot-clé `export` (permet une pré-compilation);
- **Principe essentiel :** À chaque appel avec un type donné, le compilateur **instancie** le patron.

```
int n=3,m=2;
float x=2.5,y=4.0;

n=minimum(n,m);// int minimum(int,int)

x=minimum(x,y);// float minimum(float,float)

minimum(&n,&m);// int* minimum(int *,int *)
```

Paramètres de type (1/2)

```
template <typename T> T minimum(T a,T b){...}
```

- Utilisables n'importe où dans la définition du patron, y compris pour de l'allocation dynamique;
- Il peut y avoir plusieurs paramètres de type dans un patron de fonctions.

Pour un appel correct

- `const` change le type;
- Un tableau sera converti en pointeur;
- On peut forcer le typage (pas forcément recommandé) :

```
int t[10],n, *p;
char c;
minimum(t,p); // int *
minimum<char>(n,c); // char (conversion de n)
```

Paramètres de type (2/2)

Initialisation de variables

```
template <typename T> monpat( )
{
  T x (3);
}
```

- Syntaxe générique;
- Doit être permise par le type instancié.

Valeurs par défaut

```
template <typename T=int> void (T x=0){
  // ...
}
```

Paramètres expressions

Les paramètres expressions

- S'ajoutent aux paramètres de type, ou à ceux impliquant des paramètres de type;
- Manipulés comme les paramètres classiques de fonctions

```
template <typename T> int eltstab(T* tab,int n){
  int nz=0;
  for(int i=0;i<n;i++) if (!tab[i]) nz++;
  return nz;
}
```

Surdéfinition des patrons de fonctions

- Possible si les paramètres d'appel sont différents;
- Plusieurs définitions = plusieurs patrons;
- Il faut éviter toute ambiguïté pour le compilateur.

```
template <typename T> T minimum(T a,T b){
  return a > b ? b : a;
}
// Surdefinition non ambigue
template <typename T> T minimum(T *a, T b){
  return *a > b ? b : *a;
}
// Surdefinition ambigue
template <typename T> T minimum(T *a, T *b){
  return *a > *b ? *b : *a;
}
```

Spécialisation des patrons de fonctions

- Diffère fondamentalement de la surdéfinition !
- On veut **spécifier** un comportement particulier du patron pour une certaine valeur du paramètre de type.

```
template <typename T> T fct(T a,T b){
  // ...
}
int fct(int a, int b){
  // ...
}
```

Règles lors d'un appel de patron de fonctions

Priorité à l'instance la plus spécialisée :

1. Correspondance exacte;
2. Spécialisation partielle;
3. Paramètres d'expression;
4. Paramètres de type;

+ Erreur si ambiguïté durant le processus.

```
void fct(int a,int b){ }
// prime sur
template <typename T> void fct(T a,int b) { }
//qui prime sur
template <typename T,typename U> void fct(T a, U b){ }
```

Exercice : patron de fonctions

On considère les déclarations suivantes

```
template <typename T,typename U> void fct(T a,U b){ }
template <typename T,typename U> void fct(T *a, U b){ }
void fct(int a,float b){ }

int n,p;
float x;
double z;
```

Que vont donner les appels suivants ?

```
fct(n,p);
fct(n,z);
fct(n,x);
fct(&n,p);
```

Exercice : patron de fonctions

On considère les déclarations suivantes

```
template <typename T,typename U> void fct(T a,U b){ }
template <typename T,typename U> void fct(T *a, U b){ }
void fct(int a,float b){ }

int n,p,q;
float x,y;
double z;
```

Que vont donner les appels suivants ?

```
fct(n,m);// Appel patron 1 T=U=int
fct(n,z);// Appel patron 1 T=int,U=double
fct(n,x);// Appel specification (int,float)
fct(&n,m);// Ambiguite entre patrons 1 et 2 : erreur
```

Sommaire

Les patrons de classes

Principe

- Classes définies en fonction d'un ou plusieurs types génériques (paramètre de type);
- Contraintes différentes sur les paramètres de type par rapport aux patrons de fonctions.

Patrons de classes

```
template <typename T> class PPoint{
  T x,y;
  public:
    PPoint(T,T);
    void affiche();
};
```

```
PPoint<int> p1(0,0);
PPoint<float> p2 (1.5,2);
```

Définitions de patrons de classes

- La définition d'un patron de classe (fonctions membres comprises) est en fait une **déclaration**;
- Une définition se place dans un fichier `.h` puis s'importe.

- L'**instanciation** du patron de classes se fait lorsqu'une instruction le requiert;
- Une instance de patron de classes s'appelle une classe patron.

Spécialisation d'un patron de classes

- **Important :** un patron de classes ne peut pas être surdéfini !
- Il peut en revanche être spécialisé :
  - Via ses fonctions membres;
  - Directement sous forme de classe.

```
// Classe generique
template <typename T, typename U> class A { ...};
// Specialisation totale
template <> class A <char,int> {...};
// Specialisation partielle
template <typename T> class A <T,T*> {...};
```

Fonctions membres de patrons de classes

- Peuvent être des patrons de fonctions !
- Syntaxe particulière pour définition en dehors de la classe :

```
template <typename T> class PPoint{
  T x,y;
  public:
    void afficher();
};

template <typename T> void PPoint<T>::afficher(){ }
```

Spécialisation des fonctions membres dans un patron de classes

```
template <typename T> class PPoint{
  T x,y;
  public:
    void afficher();
};

// Code generique
template <typename T> void PPoint<T>::afficher(){ }

// Code specialise
template <> void PPoint<char>::afficher(){ }
```

Paramètres de types et d'expressions

- Comme pour les patrons de fonctions, il peut y en avoir plusieurs;
- Ils peuvent avoir des valeurs par défaut.

```
template <typename T=float,int n=3> class TTableau{
  T tab[n];
  public:
    T operator[](int);
};

// Exemple d'instanciation
TTableau<int,4> v;
TTableau w;// Tableau de 3 flottants
```

Déclaration d'amitié

Cas 1 : Le paramètre de type n'est pas impliqué

```
template <typename T> class A{
  public:
    friend class B;// amie de toutes les instances
    friend void fct(); // amie de toutes les instances
};
```

Cas 2 : Le paramètre de type est impliqué

```
template <typename T> class A{
  public:
    friend class PPoint<T>;
    friend void fct(T);
};
```

Chaque fonction/classe amie le sera avec la classe patron correspondante.

Patrons et classes ordinaires

```
template <typename T> class A {...};

// Classe ordinaire derivant d'une classe patron
class B : public A<int> {...};
// Patron de classe derivant d'une classe ordinaire
template <typename T> class C : public B {...};
```

Patron de classes dérivé d'un patron de classes

- Même(s) paramètre(s) de types $\Rightarrow$ autant de classes dérivées que de classes de bases possibles;
- Plus de paramètres de types $\Rightarrow$ chaque classe de base engendre un patron de classes dérivées.

```
template <typename T> class D : public A<T> {...};
template <typename T,typename U> class E : public A<T> {...};
```

La bibliothèque standard

Ce que nous avons déjà vu

- La bibliothèque `iostream` : `cin`, `cout`, classe `string`, >>, <<;
- La bibliothèque `cmath` : `abs`, `pow`, `max`, `min`.

Plus généralement : les bibliothèques

- Ensembles de patrons de classes et de patrons de fonctions;
- Deux concepts-clés : les **conteneurs** et les **itérateurs**;
- Implémentation sous-jacente **efficace**.

Conteneurs

- Patrons de classes paramétrés par le type de leurs éléments;
- Représentent les structures de données les plus classiques : listes, ensembles, etc;
- Homogénéisation : fonctions membres communes à plusieurs conteneurs.

Conteneurs séquentiels

- Conteneurs séquentiels : éléments ordonnés suivant un ordre imposé par le programme;
- Conteneurs associatifs : valeur associé à une clé et non un emplacement (ordre non imposé par le programme).

Conteneurs séquentiels

- Représentent des concepts naturels de structures de données;
- Trois principaux : `vector`, `list`, `deque`;
- Homogénéisation : fonctions communes à ces types.

- Certaines opérations sont plus efficaces dans un conteneur que dans un autre;
  Ex) Accès à un élément plus rapide pour un vecteur que pour une liste.
- On peut passer d'un conteneur à un autre :
  - Fonction `assign` pour copier des éléments;
  - Fonction `swap` pour échanger les éléments de conteneurs.

Le conteneur vector (1/2)

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

// Vecteur d'entiers de taille 5
vector<int> v (5);
vector<float> w (4);
```

- Patron de classe;
- Même idée de base que les tableaux, mais la taille d'un vecteur peut varier !
- Accès efficace aux éléments;
- *NB : Pas la classe* `Vecteur` *du projet !*

Le conteneur vector (2/2)

Initialisation

```
vector<int> vi (5); // 5 elements initialises a 0
vector<string> vs (5,"aa"); // 5 elements valant "aa"
```

Manipulation des éléments du vecteur

```
vector<int> v (5);
v[4]=1;
int u=v[4];

//Ajout d'un element en queue de vecteur
v.push_back(u);
v.size();//Renvoie 6

//Suppression de l'element en queue
v.pop_back();
```

Le conteneur deque

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

deque<char> dc;
dc.push_front('b');
dc.push_front('a');
dc.push_back('c');
dc.pop_front();
```

- Comme `vector` : accès rapide aux éléments, ajout/suppression en fin;
- En plus : ajout/suppression rapide en début de “deck”.

Le conteneur `list` (1/2)

- Modélisation de listes;
- Possibilités multiples d'insertion et de suppression d'éléments;
- Fonctions de tri, suppression de doublons, etc.

- Ajout/insertion plus efficaces que pour `vector` et `deque`;
- Mais pas d'accès direct aux éléments.

Le conteneur list (2/2)

```
#include<list>

// Liste de flottants vide
std::list<float> lf;
// Liste de 4 booleens (initialises a false)
std::list<bool> lb (4);
```

```
int tab[4]={3,1,2,3};
std::list<int> li (tab,tab+4);
li.sort();// 1 2 3 3
li.unique();// 1 2 3
li.insert(3,2);
li.erase(2);
```

Conteneurs associatifs

- **Principe :** Retrouver un élément en fonction d'une partie de sa valeur, ou clé.
- Conteneurs principaux :
  - `map` : Un seul élément par clé;
  - `multimap` : Plusieurs éléments possibles par clé.
- Cas particulier : `set`/`multiset`, les éléments sont leur propre clé.

- **Outil :** Patron de classes `pair`;
- Regroupe deux valeurs dans un objet.

```
pair<int,char> p;
p = make_pair(3,'c');
p.first=5;
```

Le conteneur associatif map

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

map<char,int> mymap;

mymap.insert(pair<char,int>('A',1));
mymap.insert(pair<char,int>('Z',26));

map<char,int>::iterator it=mymap.find('Z');
```

Le conteneur associatif set

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

set<int> myset;

//Instructions redondantes
myset.insert(10);
myset.insert(10);

myset.insert(20);
myset.erase(20);
int t=myset.size();//t=2
myset.clear();
t=myset.size();//t=0
```

Itérateurs

- **Idée :** une action réalisable avec deux conteneurs différents doit se programmer de la même manière;
- Un itérateur généralise la notion de pointeur :
  - Sa valeur désigne un élément d'un conteneur;
  - Peut être incrémenté (++) (unidirectionnel) voire décrémenté (bidirectionnel);
- Chaque conteneur possède un itérateur (`iterator`) et un ordre sur ses éléments.

```
vector<int> v;
vector<int>::iterator i;
for(i=v.begin();i!=v.end();i++){}
for(int j=0;j<v.size();j++){}
```

Les algorithmes

La bibliothèque

```
#include<algorithm>
using namespace std
```

- Sous forme de patrons de fonctions;
- Utilisent des itérateurs.

Structure typique d'un algorithme

- Basé sur une séquence (intervalle d'itérateur);
- Différents types : transformation, recherche, tri, suppression, fusion...

Illustration : la fonction max_element

- max de cmath ne fonctionne que pour 2 éléments;
- Il existe un algorithme pour les tableaux.

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

int tab[7]={3,2,4,0,2,1,-1};
int *m=max_element(tab,tab+7);
```

Sommaire

Contexte

```
class Point2D{
  int x,y;
  public:
    bool sur_un_axe();
};
class Point3D : public Point2D{
  int z;
  public:
    bool sur_un_axe();
};
```

- Un pointeur sur un objet de type `Point2D` peut recevoir l'adresse d'un objet de type `Point3D`...
- ...mais l'appel à `sur_un_axe()` sera un appel à la fonction du type pointé (`Point2D`).

Polymorphisme : définition

Principe

- Typage ou *ligature* statique par défaut en C++ : le type est déterminé à la compilation;
- **But :** Faire prendre en compte le type de l'objet pointé;
- Typage **dynamique**, déterminé à l'exécution plutôt qu'à la compilation.

Le mot-clé `virtual`

- S'utilise (uniquement) pour des fonctions membres d'une classe;
- Indique que les appels devront utiliser un typage dynamique.

Exemples d'utilisation (1/2)

```
class Point2D{
  int x,y;
  public:
    virtual bool sur_un_axe();
};

class Point3D : public Point2D{
  int z;
  public:
    bool sur_un_axe();
};

Point2D p2; Point3D p3;
// ...
Point2D *pp = &p2;
pp->sur_un_axe();//Appel Point2D::
pp = &p3;
pp->sur_un_axe();//Appel Point3D::
```

Exemples d'utilisation (2/2)

```
class Point2D{
  public:
    virtual void affiche();
    void coordonnees();//Utilisee dans affiche
};

class Point3D : public Point2D{
  public:
    void coordonnees;
};

void Point2D::affiche{
  cout<<coordonnees()<<endl;
}
```

Parenthèse : la bibliothèque typeinfo

- Dans la bibliothèque standard de C++;
- Permet de connaître le véritable type d'un objet désigné par un pointeur ou par une référence lors de l'exécution.

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

class Point2D{...};
class Point3D : public Point2D{...};

Point3D p3; Point2D *adp=&p3;

type_info t = typeid(adp);
cout<<t.name();//Renvoie Point2D*
type_info t2 = typeid(*adp);
cout<<t.name();//Renvoie Point3D
```

Les fonctions virtuelles

- Une fonction virtuelle dans une classe reste virtuelle pour toute classe dérivée;
- Un destructeur peut être virtuel (recommandé en cas de polymorphisme);
- Un constructeur ne peut pas être virtuel.

Classes abstraites

- Une fonction virtuelle est dite pure si elle n'est pas implémentée dans sa classe;
- Une classe qui comporte au moins une fonction membre virtuelle pure s'appelle une **classe abstraite**;
- Définition implicite (explicite en Java via le mot-clé *abstract*).

Principe des classes abstraites

- Pas destinées à être instanciées en objets...
- ...mais à donner d'autres classes par héritage;
- Pour qu'une classe dérivée ne soit pas abstraite, elle doit redéfinir toutes les fonctions virtuelles pures.

Exemple

```
class Point{
  public:
    virtual void affiche();
};

class Point2D : public Point{
  float x,y;
  public:
    void affiche();
};

void Point2D::affiche(){
  cout<<x<<","<<y<<endl;
}
```

Sommaire

Les exceptions

- Mécanisme de détection d'une anomalie;
- Découple la détection et le traitement de l'erreur.

- Mot-clé `throw` : déclare une exception identifiée par un type/une classe;
- Structure de blocs `try`/`catch` : Exécution et possible détection d'erreur/Traitement de l'erreur.

Exemple avec la classe Tableau (1/2)

```
class Tableau{
  int taille, *vec;
  public:
  int & operator[] (int);
};
// Classe d'exception
class TableauLim{ };

int & Tableau::operator[] (int i){
  if (i<0 || i>taille){
    TableauLim t;
    throw (t);
  }
  return vec[i];
}
```

Exemple avec la classe Tableau (2/2)

```
#include <iostream>
#include <cstdlib>

int main(){
  try {
    Tableau t (10);
    t[11]=2;
  }
  catch(TableauLim t){
    cout<<"Exception limite";
    exit(-1);
  }
}
```

- `exit` (bibliothèque standard) sort du bloc `try`, autrement l'exécution reprend après l'instruction problématique;
- `abort` arrêterait l'exécution.

Parenthèse : la bibliothèque `cstlib`

Présentation :

- Fait partie des bibliothèques importées du `C`, comme `cmath`;
- Partage donc certaines caractéristiques d'efficacité.

Contenus :

- Conversion de `string` en `int` (`atoi`), `float` (`strtof`),etc;
- Générateur de nombres aléatoires `rand`;
- Recherche binaire et tri dans un tableau;
- Gestion de programmes : `abort`, `exit`, `system`, etc.

Gestion des exceptions

- On peut avoir plusieurs blocs `catch` à la suite, avec des types d'exceptions différents $\Rightarrow$ Gestionnaire d'exceptions;
- Les règles de choix de gestionnaires sont similaires à celles de la surdéfinition de fonctions;
- Si pas de gestionnaire approprié : appel d'une fonction `terminate` qui effectue l'instruction `abort`.

Les exceptions standards

Ex) bad_alloc, out_of_range, invalid_argument.

- Classes dérivées d'une classe de base `exception`;
- Déclaration dans le fichier en-tête `stdexcept`;
- Ces exceptions peuvent être déclenchées par des fonctions/opérateurs de la bibliothèque standard (ex : `bad_alloc` via `new`).

La classe `exception` et ses dérivées

- Fonction membre virtuelle `what` décrivant la nature de l'exception;
- Chaque classe possède un constructeur avec un argument de type chaîne;
- On peut créer ses propres classes dérivées.

Illustration : exceptions standards

```
#include <iostream>
#include <cstdlib>
#include <stdexcept>

class monexception : public std::exception {
  char * texte;
  public :
    monexception (char *t){ texte=t;};
    const char * what() const throw(){ return texte;}
}

try{// ...
  throw std::runtime_error("Pb runtime");
  // ...
  throw monexception("Mon pb");
}

catch(const std::exception & ex){
  std::cout<<ex.what();
}
```

Résumé : programmation générique (1/2)

Les patrons

- Patrons de fonctions : définitions, peuvent être surdéfinis ou spécialisés;
- Patrons de classes : définitions, instanciation en classe patron;
- Importance de la compilation et de la priorité des définitions;
- Mise en pratique dans le **TP 7**.

Le polymorphisme

- Une (autre) technique de programmation générique;
- Repose sur la notion de fonction virtuelle, permet la déclaration de classes abstraites;
- Utile pour les types pointés $\Rightarrow$ typage dynamique.

Résumé : programmation générique (2/2)

Bibliothèque standard

- Des conteneurs (patrons de classes) représentant des structures de données standard;
- Des itérateurs pour parcourir efficacement l'ensemble des éléments;
- Des algorithmes (patrons de fonctions) rapides de tri, comparaison, etc.

Les exceptions

- Séparent détection et traitement d'erreurs;
- Existent dans la bibliothèque standard;
- Possible de créer ses propres exceptions.