

# Le Langage de Programmation

## C++

# Plan

**1- Les caractéristiques de base du langage**

**2- Les classes**

**3- Hiérarchie de classes**

# De C à C++ : Typage Fort

- Permet une plus grande sécurité
- Conserve les conversions implicites, "naturelles"
  - Exemple : entier -> réel
- Impose la notion de prototype de fonctions (ou signature)
  - En C, seul le type de la valeur de retour compte : **double** cos();
  - En C++, le type des arguments doit être spécifié : **double** cos(**double**);
  - Le concept de prototypage a été repris par ANSI C
- Permet la surcharge des opérateurs et des fonctions
  - double** sqrt (**double**);
  - int** sqrt(**int**);

# De C à C++ : Modularité et approche Objet

## ➤ Fonctions & Procédures

- structuration du flot de contrôle
- données locales

## ➤ Modules

- regroupement de données et de procédures
- notion de contrôle d'accès (information hiding)

## ➤ Types Abstrait, paquets

- le module devient un schéma de type
- on déclare des variables(des instances) de ce type

## ➤ Programmation Orientée Objets

- on introduit une hiérarchie entre les types (héritage)
- on permet le typage dynamique

# Les + de C++ comparé à C

## ◆ Langage de base

- Typage fort
- Surcharge des opérateurs et des fonctions
- Classe de mémoire constante (**const**)
- Nouveau type **reference** comme alternative (partielle) aux pointeurs

## ◆ Types abstraits

- Notion de classe (**class** : extension de la notion de *struct*)
- Contrôle d'accès aux membres d'une classe
- Contrôle des opérations de création, destruction, initialisation et affectation

## ◆ Approche Objet

- Héritage simple et multiple

# Implémentations et Versions de C++

- C++ est né au début des années 80
- Conçu et développé par Bjarne Stroustrup (AT&T Bell Labs)
- Stroustrup désirait ajouter au langage C les classes de Simula

## ➤ **Compilateurs disponibles sur le marché**

- **CC, cfront** : AT&T, version 3.0 (*Norme*)
- **g++** : GNU C++ (Free Software Foundation)
- Oregon C++, GlockenSpiel C++, Zortech C++, Turbo C++, ...

# Les compilateurs C++

## ◆ Les traducteurs de la famille "cfront"

Code C++      →      Code C      →      Code machine

- Facilement portables sur toute machine ayant un compilateur C
- Mise au point difficile avec les débogueurs symboliques classiques
- Compilation relativement lente et faibles optimisations

## ◆ Les compilateurs natifs (Ex: g++)


Code C++      →      Code machine

- Pas toujours disponibles sur une machine quelconque
- Bonnes performances, capables d'optimisation intéressantes
- Débogueur plus adapté (gdb, ...)

# C++ est un langage Hybride

On peut réaliser des applications

- Purement orientées objet,
- purement procédurales,
- ou combiner les deux approches à sa guise.

Apprendre C++  Apprendre un nouveau langage et surtout une nouvelle façon de penser



## Les Caractéristiques de base (1)

# Conventions Lexicales

## ➤ Six (6) types de tokens

- Identificateurs, mots clés
- **constantes**
- chaînes de caractères
- **Opérateurs**
- séparateurs

## ➤ Commentaires

- `/* ceci est un commentaire */`      `// cela également`

## ➤ Identificateurs

- une lettre ou un souligné, suivi par une lettre, souligné, chiffre
- **majuscules et minuscules significatives**
- longueur arbitraire

# Emplacement et portée des déclarations

## ◆ Emplacements

Les déclarations peuvent apparaître presque n'importe où, et non pas uniquement en début de bloc, comme en C.

## ◆ Portée

- **Locale** : une entité déclarée dans un bloc est locale au bloc
- **Fichier** : un nom déclaré en dehors de tout bloc, fonction, ou classe, est connu dans tout le fichier à partir de sa déclaration, et est considéré comme un identificateur global
- **Fonctions** : une étiquette (label) est connue dans l'ensemble de la fonction où elle apparaît.

# Tableaux et pointeurs

## ➤ Tableaux

- **int** Tab[12];                      **char** Buf[80][132];
- **float** ff[4] = { 1.0, 2.3, 0.5, 4.9};

## ➤ Pointeurs

- **int\*** pi = &a; **char\*** pc = Buf[12];      **char\*** str = "chaine";
- **Void\*** : pointeur générique, permet de stocker des pointeurs vers n'importe quel type d'objet
- **0** : pointeur nul générique

**void\*** pv = &str;  
**char\*** st = (**char\***) pv;  
st = **0**;

# Structures

## ◆ Structures

```
struct Nom {  
    char * prenom;  
    char * nomFamille;  
};
```

**Nom** est un nouveau type (pas besoin de typedef)

### Exemples :

```
Nom n;  
n. prenom = "Robert"  
n.nomFamille = "De Niro";  
Nom n2 = {"Meryl", "Streep"}
```

# Les constantes

**const** permet de déclarer et de définir des constantes typées

- **const** int BufSize = 1024;
- **const** char\* Version = '1.2';

**Une constante non externe doit être initialisée**

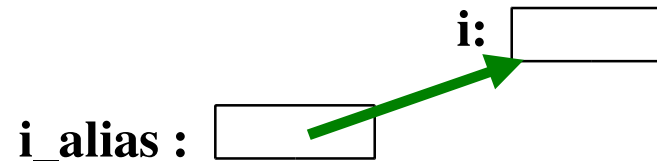
- **const** int a; // incorrect, manque l'initialisation
- **extern const** char BackSpace; // correct, constante définie ailleurs

## ➡ Contrôle d'accès

- Version[2] = '3'; // interdit
- int\* ptr = &BufSize; // interdit sinon on pourrait modifier BufSize
- \*ptr = 21; // via ptr, comme ceci

# Les références

- Une référence est un pointeur caché sur un objet, qui doit **toujours être initialisé**, et s'utilise **sans indirection**



- Une référence est une façon d'utiliser plusieurs noms pour un même objet. Elle doit obligatoirement être initialisée.

```
Int i;   int& i_alias = i;    // i_alias et i désigne la même variable
i = 2;   i_alias = 3;        // équivalent à i=3
int& j;   // erreur : initialisation absente
```

## Interdictions

référence à une référence, tableau de références, pointeur sur une référence

# Utilisation des références (1)

## ◆ Nommage d'éléments de tableau

```
int register[16];  
int& SP = register[14];  
int& PC = register[15];
```

## ◆ Tableau de constantes

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

### Programme principal

```
main () {  
    int i, j;  
    i = 6;  
    j = 8;  
    swap (i, j);  
}
```



## Utilisation des références (2)

### ◆ Type de retour d'une fonction

- Permet, par exemple, de manipuler un objet dont l'adresse est calculée par la fonction comme une variable classique

```
int& Tab(int i, int j) {  
    static int* tab;  
    .....  
    return tab[i * LineSize + j];  
}
```

#### Programme principal

```
main () {  
    ....  
    Tab(3, 4) = 12  
    ....  
    int a = Tab(8, 3);  
    ....  
}
```

# Programme et Entrées/Sorties

## ◆ Programme minimal : afficher "hello world"

```
#include <stream.h>
main()
{
    cout << "Hello, Worl! \n";
}
```

## ◆ Entrées/sorties standards

- cout <<
- cin >>
- cerr <<

# Allocation dynamique de mémoire

## ➤ new

- l'opérateur **new** permet de créer dynamiquement des objets
- **new(T)**, où T est un type de données, crée un objet de type T et retourne un pointeur vers cet objet

```
int* pi;
```

```
pi = new int;           // allocation d'un entier
```

```
pi = new int[10]        // allocation d'un tableau de 10 entiers
```

## ➤ delete

- l'opérateur **delete** libère un objet créé par **new**

```
int* i = new int;
```

```
i = new int[10];
```

```
delete i;
```

```
delete[] i;           // détruit le tableau de 10
```

# Mécanisme de surcharge (overloading)

## ➤ Surcharge

Un même nom de fonction peut désigner plusieurs fonctions différentes, si chacune a une liste de paramètres distincte de celles des autres.

```
int max (int, int);
```

```
double max(double, double);
```

```
int max(int, int, int);
```

```
double max(int, int);           // non autorisé
```

## ➤ Invocation

En cas d'ambiguïté, un algorithme de détermination de la meilleure solution est appliqué. Il met en jeu les conversions standards et définies entre types.

# Utilisation des pointeurs sur des constantes

- Les pointeurs sur des constantes sont souvent utilisés comme **arguments de fonctions dont on veut s'assurer qu'elles ne modifient pas les valeurs pointées.**

```
int strcpy(char* s1, const char* s2);
```

```
const char* version = "1.2";
```

```
char buf [BufSize];
```

```
strcpy(version, buf);           // Interdit
```

```
strcpy (buf, version);          // Correct
```

## ◆ Tableau de constantes

- **const int** Premiers[] = {2, 3, 5, 7, 9, 11, 13, 17, 19};

Premiers est un tableau de constantes entières

# Pointeurs sur constantes, pointeurs constants

## ◆ Pointeur sur constante

Un pointeur sur une constante ne doit pas forcément pointer sur une constante, mais sur un objet qui est considéré constant via ce pointeur.

```
const int* pci;           // pci est un pointeur sur une constante entière
pci = &BufSize;
int a;
pci = &a;                 // on ne peut pas modifier a via pci
```

## ◆ Pointeur constant

C'est un pointeur dont on ne peut modifier la valeur. Il pointe sur l'objet choisi à son apparition. Par contre, on peut modifier l'objet pointé.

```
int* const cpi = &a       // cpi est un pointeur constant sur a
*cpi = 6;                 // OK
const int* const cpBuf = &BufSize; // Combinaison des deux
```

# Les Classes (2)

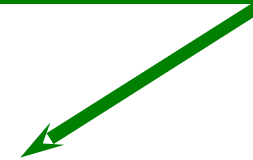
# Notion de fonctions membres (1)

## Exemple Rectangle (syntaxe C)

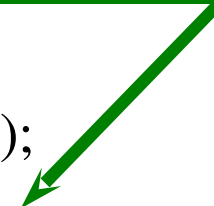
```
struct Rectangle {  
  short x,y ;  
  unsigned short largeur, hauteur;  
};
```

```
void initialiser_rectangle(Rectangle*, int, int, int, int);  
void deplacer_rectangle(Rectangle*, int, int);  
void afficher_rectangle(Rectangle*)
```

Structure de l'objet Rectangle



Comportement de l'objet Rectangle



⇒ il n'existe aucun lien entre le code et les données



## Notion de fonctions Membres (2)

```
struct Rectangle {  
short x,y ;  
unsigned short largeur, hauteur;
```

```
void initialiser(int, int, int, int);  
void deplacer(int, int);  
void afficher();};
```

Signature des fonctions



⇒ Les fonctions membres ne sont connues que par les objets de type Rectangle

### Exemple :

```
Rectangle r;  
r.initialiser(10,20,30,40);  
r.x = 10;    // pas de protection, besoin d'encapsulation
```

# Définition d'une classe

## ◆ Contrôle d'accès aux champs et fonctions membres d'un objet

- Un type est défini par sa représentation et les fonctions qui le manipulent
- Extension de la notion de structure C

**class** <Nom>

{

**private:**

déclaration des champs privés

**public:**

déclaration des champs publics

};

⇒ **Les champs sont des données ou des fonctions (*fonctions membres*)**

# Exemple de Classes

## Fichier Rectangle.h :

```
class Rectangle {  
  private:  
    short x,y ;  
    unsigned short largeur, hauteur;  
  public:  
    void init( int, int, int, int);  
    coord Bas_Gauche();  
    coord Haut_Droit();  
    void deplacer(int, int);  
    void afficher();  
    boolean contient_point(int xx, int yy); };
```

# Encapsulation (1)

*Par défaut, tous les membres d'une classe sont privés. Il est toujours préférable de toujours indiquer explicitement les droits d'accès aux champs*

## ◆ Portée

Les noms des membres (champs et méthodes) d'une classe sont locaux à cette classe.

## ◆ Visibilité

Les noms des membres privés (champs et méthodes) ne peuvent être utilisés que par les fonctions membres de la classe.

*Les noms des membres publics (champs et méthodes) constituent l'interface des objets de la classe*

## Encapsulation (2)

**C++ adopte une encapsulation de classe**

**C++ autorise la définition des variables de classes**

"static member variable"

Déclarer un membre **"static"** restreint sa visibilité et le rend indépendant des objets individuels de la classe

```
class Rectangle {  
    static int _NbRectangle = 0;  
    ...  
}
```

# Création d'objet

## Rappels

Un objet est un représentant d'une classe

*création d'un objet = instantiation d'une classe*

- Chaque objet contient un exemplaire des champs donnés d'une classe, avec des valeurs qui lui sont propres
- Une classe contient presque toujours une (ou plusieurs) méthode(s) particulière(s) destinée(s) à créer les objets dans un état cohérent.

# Les constructeurs

- Un constructeur est une fonction membre (en général publique) de même nom que la classe
- Cette fonction membre est généralement surchargée

```
class Rectangle{  
    ...  
    public:  
        Rectangle();           // correspond au new  
        Rectangle(int, int, int = 10, int = 10);  
        ...  
};
```

## Exemple :

```
Rectangle r1;  
Rectangle r2(0,0);  
Rectangle* ptr = new Rectangle(0,0,50,50);
```

# Le destructeur

- Un destructeur est une fonction membre (en général publique) de nom ~classe
- Cette fonction membre ne peut pas être surchargée
- L'opérateur **delete**, d'un objet de la classe X, fait automatiquement appel au destructeur

```
class Rectangle
{
...
public:
    ~Rectangle(); // destructeur
    ...
};
```

## Exemple:

```
Rectangle* r = new
Rectangle(0,0);
...
delete r;
```



# Envoi de messages aux objets

*L'invocation d'une méthode correspond à l'envoi d'un message à un objet.*

- Le type du message est le nom de la méthode, ses paramètres sont ceux de la méthode
- On utilise la syntaxe habituelle de l'accession aux membres d'une structure, sauf qu'ici il s'agit d'un appel de fonction

Rectangle r;

Rectangle\* ptrec = &r;

r.init(10,20,110,150);

ptrec->afficher();

# Identité d'objet

Dans toutes les méthodes d'une classe, le mot clé **this** désigne le pointeur vers l'objet recevant le message

Pour une classe donnée X, **this** est de type X\*

```
void Rectangle::PrinThis()
{
    cout << hex(this);    // équivaut à imprimer l'adresse de r
}
```

# Utilisation de this

- **\*this** désigne l'objet lui-même
- L'accès aux membres d'une classe se fait implicitement via **this**  
**this->x**,            **this->largueur**,            **this->afficher()**;

A l'intérieur d'une fonction membre l'utilisation de **this** est inutile, puisqu'implicite.

Cependant, il faut bien se souvenir de cet état de fait, qui est une caractéristique importante des objets et qui permet de détecter certaines erreurs.

(Utiliser **this** pour supprimer certaines ambiguïtés sur les variables locales)

## Exemple1 : la classe Rectangle

**Fichier Rectangle.h :**

```
class Rectangle{  
private:  
    short x,y ;  
    unsigned short largeur, hauteur;  
public:  
    Rectangle(int, int, int = 10, int = 10);  
    point Bas_Gauche();  
    point Haut_Droit();  
    void deplacer(int, int);  
    void afficher();  
    boolean contient(int xx, int yy);  
    ~Rectangle();};
```

## Exemple de méthode

Fichier Rectangle.cc :

```
boolean Rectangle::contient(int xx, int yy)
{
    if (xx >= x && xx < x+largeur && yy >=y && yy < y+hauteur)
        return True;
    else
        return False;
};
```

# Programme principal

## Fichier Main.cc

```
#include <stream.h>
#include "Rectangle.h"

main ()
{
    Rectangle r(12,34,110,25);
    Rectangle NullRect(0,0,0,0);
    if (r.contient(38,45))
        cout << "C'est ok\n";
    else
        cout << "Le point n'est pas dans le rectangle\n"; }
```

## Exemple2 : la classe STRING

```
class String
{
    char* p;
    int size;
    public:
        String(int sz)    { p = new char[size = sz+1];    };
        String(char* s ) {
                                p = new char[size = strlen(s) +1];
                                strcpy(p,s);
                                };
        ~String() { delete[] p; };
        ...
};
```

# La classe Fenêtre

**Exemple :**



Bienvenue

```
class Fenetre {  
    String _name;  
    Rectangle _shape;  
    public:  
        Fenetre(String s, Rectangle r);  
    ...  
};
```



# Initialisation des objets membres

## ◆ Initialisation des membres d'une classe

- Par défaut, le constructeur vide de ces objets est utilisé pour leur initialisation.
- On peut spécifier des arguments de constructions à ces objets si le constructeur vide ne convient pas.

## ◆ Ordre de construction/destruction d'une classe

- 1) construction des membres
- 2) construction de la classe elle-même

**La destruction se fait en sens inverse.**

## Exemple

```
class Fenetre {  
    String _name;  
    Rectangle _shape;  
    public:  
        Fenetre(String s, Rectangle r) : _name(s), _shape(r);  
        Fenetre(char* s, int x, int y, int z, int t) : _name(s),  
                                                         _shape(x,y,z,t);  
        ...  
};
```

**Attention** : l'initialisation d'un objet dans le corps du constructeur est possible, mais entraîne des constructions et des destructions inutiles d'objets.

```
Fenetre::Fenetre(String s, Rectangle r) {  
    _name = new String(s); };
```

## Membre statique

Compter le nombre d'instances d'une classe à un instant donné

```
class Fenetre {  
    ...  
    static int nb_fenetre;  
    public:  
        Fenetre(const String& s, const Rectangle& r) : _name(s), _shape(r);  
        ~Fenetre();  
};  
  
Fenetre::Fenetre(...){ nb_fenetre++;}  
Fenetre::~Fenetre(){ nb_fenetre--;}
```

# La Désencapsulation

Elle a pour but de supprimer l'"overhead" introduit par l'accès aux champs et aux fonctions membres d'une classe (appels de fonctions trop importants)

## ■ Champs **public**

```
class X{
```

```
...
```

```
public:
```

```
    Type _something;
```

```
};
```

=> **X::\_something** = ...;

## ■ Les fonctions **friend**

Ce mécanisme permet, à une fonction arbitraire, l'accès aux membres privés d'une classe

# Les fonctions friend

## Exemple

```
class X{  
    ...  
    friend void foo(char*, int);    // donne à la fonction foo l'accès aux champs  
                                     // de la classe X  
    friend int Y::bar();              // donne à la méthode bar de la classe Y  
                                     // l'accès aux champs de X  
  
    friend class Z;                  // donne à toutes les méthodes de la classe Z  
                                     // l'accès aux champs de la classe X  
}
```

## Exemple (1)

```
class Matrice {  
    float v[][];    int size;  
    public:  
        Matrice(int sz);  
        int length(){ return size}  
        float& elem(int i, int j) { return v[i][j];} }
```

```
class Vecteur {  
    float v[];    int size;  
    public:  
        Vecteur (int sz);  
        int length(){ return size}  
        float& elem(int i) {return v[i];} }
```

## Exemple(2)

```
Vecteur multiplier(Matrice& m, Vecteur& v) {  
    Vecteur res(v.length());  
    for (int i=0; i<m.length(); i++) {  
        res.elem(i) =0;  
        for (int j = 0; j <m.length(); j++)  
            res.elem(i) += m.elem(i, j) * v.elem(j); }  
    return res;  
}
```

**Bien que très naturelle, cette méthode risque de se révéler assez inefficace, surtout si on contrôle les bornes dans elem() et si elle n'est pas inline.**

Si  $v.size = 4$  et  $m.size = 4$ , il y a  $4*(1+4*3)$  appels de la méthode *elem*

## Exemple (3)

```
class Matrice { ...  
    friend Vecteur multiplier(Matrice&, Vecteur&);  
}  
  
class Vecteur { ...  
    friend Vecteur multiplier(Matrice&, Vecteur&);  
}  
  
Vecteur multiplier (Matrice& m, Vecteur& v) {  
    Vecteur res(v.size);  
    for (int i=0; i<m.size; i++) {   res.v[i] =0;  
                                     for (int j = 0; j <m.size; j++)  
                                         res.v[i] += m.v[i][j] * v.v[j]; }  
    return res; }
```



# Danger de la désencapsulation

- La désencapsulation permet à une fonction l'accès arbitraire aux membres privées d'une classe
- Cela brise donc le principe d'encapsulation des données, voir de modularité
- Il faut donc essayer de *minimiser* leur emploi

## DANGER :

- Si la représentation interne change, les fonctions **friend** nécessiteront souvent une réécriture complète
- De même si l'encapsulation des champs d'une classe n'est pas respectée (public), toutes les fonctions utilisant ces champs nécessiteront une réécriture. Dans le pire des cas, utiliser plutôt les fonctions **friend**.

# La redéfinition des opérateurs

*Les opérateurs sont des méthodes qui permettent de redéfinir des opérateurs C++, afin de pouvoir les appliquer aux objets d'une classe, et les manipuler aussi uniformément que les types de base du langage.*

```
class complex
{
    double re, im;
public:
    complex(double r, double i) {re=r; im=i}
    ...
    friend complex operator+(complex, complex);
    complex& operator+=(double);
    friend complex operator*(complex, complex)
    ...
};
```

## Exemples : Surcharge d'opérateur

```
complex operator+(complex a, complex b) {      // fonction friend
    return complex(a.re+b.re, a.im+b.im);
};
```

```
complex& complex::operator+=(complex a) {
    re += a.re;
    im +=a.im
    return *this
};
```

```
complex operator*(complex a, complex b) {      // fonction friend
    complex prod = a;
    prod *=b;
    return prod; };
```

# La surcharge des opérateurs

## ◆ Contraintes

- pas d'introduction de nouveaux opérateurs
- pas de modification de précedence
- pas de redéfinition des opérateurs de types de base
- pas de modification du nombre d'arguments

## ◆ Opérateurs surdéfinissables

- + - \* / % ^ & ~! , = < > <= >=
- ++ -- << >> == &= && || += -= /= %= ^= &= |= <<= >>= []  
( ) -> ->\* new delete

## ◆ Opérateurs non surdéfinissables

- :: . .\* ?: sizeof

# La classe String

```
Main() {  
    ...  
    String s1, s2 ("Hello World");  
    String s3;  
    s3 = "Bitmap";  
    String s4 = s2 + s3;  
    s4 += " et fin";  
    s4[1] = 'e';  
    if (s4 == "Hello") ...  
    if (s4 == s2) ...  
    if (s4.length() >= 5) ...  
    ...  
};
```

## La classe String (version 0)

```
Class String {  
    char* p;  
    int size;  
    public:  
        String (int sz=1);           // String x;  
        String (char* s );           // String x = "abc";  
        ~String() { delete p; };  
        int length() { return size; }  
        String& operator=(const char*);  
        char operator[](int i);  
        friend String operator+(const String&, const String&);  
        String& operator+=(const String&);  
        friend String operator+(const String&, const char*);  
        String& operator+=(const char*); }  
}
```

# Constructeurs / Destructeurs

```
String::String(int sz) {  
    p = new char[size = sz];  
    p[0] = 0;  
};
```

```
String::String(char* s) {  
    p = new char[size = strlen(s)+1];  
    strcpy(p,s);  
};
```

```
String::~~String() {  
    delete[] p;  
};
```

# Surcharge des opérateurs

```
Char String::operator[](int i) {  
    if (i<0 || strlen(p)<i){  
        cerr <<"index out of range";  
        exit(1);  
    };  
    else  
        return p[i];  };
```

```
String& String::operator=(const char* s) {  
    p = new char[size = strlen(s) +1];  
    strcpy(p,s);  
    return *this;  };
```



# Hiérarchie de classes (3)

# La conception orientée-objet

## ◆ Vocabulaire

- **La généralisation :** (lien **IS\_A** : sort of, kind of)
  - » Notion de super-classe, notion de classe abstraite
  
- **La spécialisation :** (lien **IS\_A** : sort of, kind of )
  - » Notion de sous-classe, notion de classe dérivée, ajout de nouvelles propriétés, redéfinition (surcharge) des opérateurs et méthodes
  
- **L'aggrégation :** (lien **IS\_PART\_OF**)
  - » Notion d'objets composants et composés

# Les classes dérivées

*C++ permet de construire des hiérarchies de classes (is-a), par le mécanisme des classes dérivées*

- Une classe dérivée hérite des membres de sa classe

Personne



Enfant

**Classe de base ou super-classe**

**Classe dérivée ou sous-classe**

- La classe dérivée peut ensuite être à son tour utilisée comme classe de base pour former ainsi une hiérarchie

## Définition d'une classe dérivée

```
class Personne {  
    string nom;  
    string prenom;  
    public:  
        char* comment_tu_t_appelles(); // retourne le nom et le prénom  
        char* get_prenom();           // retourne le prénom  
        void afficher(); };
```

```
class Enfant : Personne {  
    public:  
        Personne* pere, mere;  
        char* comment_tu_t_appelles(); // retourne le prénom uniquement  
        char* Nom_famille();           // retourne le nom de famille du père  
        void afficher(); };
```

## Classe dérivée ...

Une classe dérivée hérite de la structure et du comportement de sa super-classe

### Personne

nom:
prenom:

### Enfant

nom:
prenom:
Pere:
Mere:

**Personne** pa, ma;

...

**Enfant** bebe ("Eric", "Legrand");

bebe.pere = &pa;

bebe.mere = &ma;

**cout** << bebe.comment\_tu\_t\_appelles();

**cout** << pa.comment\_tu\_t\_appelles();

**cout** << bebe.mere->comment\_tu\_t\_appelles();

bebe

Nom: Legrand
Prenom: Eric
pere: @1
Mere: @2

# Héritage : les exceptions

**Une classe dérivée hérite de tous les membres de sa classe, à l'exception des suivants :**

- **le(s) constructeurs**
- **le destructeur**
- **l'opérateur =**

# Constructeurs / Destructeurs

**L'ordre de construction des objets de classes dérivées se fait dans l'ordre hiérarchique des classes :**

## ➤ **Création d'un objet de la classe C**

- appel du constructeur de A
- appel du constructeur de B
- appel du constructeur de C



**La destruction se fait dans l'ordre inverse de la construction**

# Initialisation des objets membres

- Initialisation des membres d'une classe, par défaut le constructeur vide de ces objets est utilisé pour leur initialisation
- On peut spécifier des arguments de construction à ces objets si le constructeur vide ne convient pas

## Exemple:

```
Enfant::Enfant(const char* n, const char* p) : personne(n,p) { };
```



# Encapsulation et Héritage

**En C++, il existe deux modes de dérivation des classes :**

- **Dérivation privée :** encapsulation forte des classes de base
  - » Tous les membres (publics et privés) de la classe de base sont privés dans la classe dérivée.
- **Dérivation publique :** encapsulation faible des classes de base
  - » Les membres de la classe de base gardent leurs propriétés d'accès dans la classe dérivée.
- **Les membres protégés : (protected)**
  - » Ils permettent de rendre des membres d'une classe accessibles uniquement aux classes qui en dérivent.

# Les parties protégées

*Les parties protégées permettent de rendre les membres d'une classe accessibles aux classes qui en dérivent*

```
class A {  
    protected:  
        int n;  
    ... };
```

```
class B : A {  
    ... };
```

- Les membres **protected** de **A** ne sont accessibles qu'aux classes qui en dérivent, ils restent privés pour les autres utilisateurs.

```
int B::foo() { return ++n; }    // ok
```

```
A a;  B b;
```

```
a.n = 3;                        // erreur : accès non autorisé
```

```
b.n = 4 ;                       // erreur : accès non autorisé
```

## Dérivation privée(1)

**Par défaut, la dérivation se fait de façon privée.**

- Cela signifie que les utilisateurs d'une classe dérivée n'ont pas accès aux membres publiques de sa classe de base. Ces membres sont privés dans la classe dérivée.
- D'éventuelles sous-classes de la classe dérivée n'y auront pas accès.

```
class A {...};           class B : private A {...};    // dérivation privée
```

Dans ce cas, la conversion standard d'un **B** vers un **A** n'est pas valable.

```
B b;
```

```
A* a = &b;           // interdit
```

En effet, une telle conversion permettrait aux objets de la classe **B** d'accéder aux membres de la classe **A** qui leur sont interdits.

## Dérivation privée (2) : exemple

```
class A {  
    private:  
        int a;  
    public:  
        void foo();  
        void toto();  
};
```

```
class B : private A {  
    public:  
        void bar();  
};
```

### Exemple :

```
B b;
```

```
b.foo();
```

// erreur : foo() est un membre privé

```
A a;
```

```
a.foo();
```

// ok : foo() est un membre publique de A

## Dérivation privée(3)

### Hériter des membres malgré une dérivation privée

- On peut spécifier l'accès à certains membres de la classe de base grâce à la convention suivante :

```
class B : private A {  
    public:
```

```
        A::toto();    // toto() sera aussi une fonction publique de B
```

- Cette notation ne permet pas de transgresser les règles d'accès :
  - » rendre **public** dans la classe un membre privé de sa classe de base
  - » rendre **private** dans la classe dérivée un membre public de sa classe de base

```
class B : private A {  
    private:
```

```
        A::a; }
```

```
        // erreur : a est spécifié private dans A
```

# Dérivation publique

- Les membres de la classe de base gardent leurs propriétés d'accès dans la classe dérivée.

```
class A {  
    private:  
        int n;  
    public:  
        void foo();  
        void bar();  
};
```

```
class B : public A {  
    // foo() et bar() sont accessibles  
    // pour un utilisateur de la classe B  
};
```

Les membres privés de **A** ne sont pas accessibles depuis **B**

```
A a;          B b;
```

```
a.n = 3;
```

```
b.n = 4 ;
```

```
// erreur : accès non autorisé
```

```
// erreur : accès non autorisé
```

## Classe dérivée

```
class Personne {  
    string nom;  
    string prenom;  
    public:  
        char* comment_tu_t_appelles();    // retourne le nom et le prénom  
        char* get_prenom();                // retourne le prénom  
        void afficher();  
};  
  
class Enfant : public Personne {  
    Personne* pere, mere;  
    public:  
        char* comment_tu_t_appelles();    // retourne le prénom uniquement  
        char* nom_famille();               // retourne le nom de famille du père  
        void afficher(); };
```

## Invocation des méthodes des classes de base

```
void Personne::afficher()
```

```
{
```

```
    cout << " le nom est " << nom << "\n";
```

```
    cout << " le prénom est " << prenom << "\n";
```

```
};
```

```
void Enfant::afficher()
```

```
{
```

```
    this->Personne::afficher();
```

```
    cout << " Voici le nom de mes parents \n";
```

```
    pere->afficher();
```

```
    mere->afficher();
```

```
    nom = "Test encapsulation";
```

```
    // erreur : accès non autorisé
```

```
};
```



# Liens d'héritage, conversion automatique

## ◆ Famille d'objets

- Les objets d'une classe dérivée peuvent être logiquement considérés comme des objets de leur classe de base.
- Il y a conversion automatique et implicite d'un objet d'une classe dérivée vers sa (ou ses) classe(s) de base.

**Enfant** bebe;

**Personne\*** ptr = &baby; // ok : si dérivation publique

**class A** { };

**class B : public A** { };

**class C : public B** { };

Un **C** est un **B**, un **B** est un **A**, un **C** est un **A** par transitivité

**C** c; **B\*** b = &c; **A\*** a = &c; b = a; // erreur : un **A** n'est pas un **B**

# Le Polymorphisme(1)

- **Langage monomorphe** : les valeurs du langage ont un type unique qui peut être déterminé à la compilation du programme. C'est le cas des langages traditionnels tels que Pascal.
- **Les langages polymorphes** : une valeur du langage, et notamment une fonction, peut appartenir à plusieurs types à la fois.

**Exemples de redéfinition d'une fonction membre :**

```
char* Personne::comment_tu_t_appelles();
```

```
void Personne::afficher();
```

```
char* Enfant::comment_tu_t_appelles();
```

```
void Enfant::afficher();
```

## Polymorphisme(2)

**Personne** moi;

**Enfant** bebe;

**Personne**\* ptr = &bebe;

**cout** << bebe.comment\_tu\_t\_appelles();

**cout** << moi.comment\_tu\_t\_appelles();

**cout** << ptr->comment\_tu\_t\_appelles();

**Le polymorphisme introduit la notion de *typage statique* et *typage dynamique*.**

- *bebe* est statiquement et dynamiquement un **Enfant**
- *moi* est statiquement et dynamiquement une **Personne**
- *ptr* est statiquement une **Personne**, *mais dynamiquement* un **Enfant** =>  
*Quelle méthode faut-il invoquer?*

# L'invocation de méthodes

- A cause du typage fort de C++, le type de l'objet est perdu après sa conversion vis à vis des messages.
- La détermination de la méthode à appeler est entièrement statique.

**Enfant** bebe;

**Personne**\* ptr = &bebe;

**cout** << bebe.comment\_tu\_t\_appelles(); // **Enfant**::comment\_tu\_t\_appelles()

**cout** << ptr->comment\_tu\_t\_appelles(); // **Personne**::comment\_tu\_t\_appelles()

- Toute méthode doit être une méthode connue au niveau du type statique du receveur dans la hierarchie.

**cout** << ptr->Nom\_famille();

// erreur : fonction non définie au  
// niveau de **Personne**

# Polymorphisme dans un langage classique

- En C, ou dans d'autres langages plus classiques, un tel comportement peut être décrit par l'utilisation de switch.

```
Switch (obj->type) {  
  case Personne : comment_tu_t_appelles_personne(obj); break;  
  case Enfant   : comment_tu_t_appelles_enfant(obj);   break;  
  ...}
```

- Le type n'est pas dynamique, l'ajout de nouveaux types d'objets impossible sans modifier le code, code plus important et moins performant
- On peut également utiliser une structure d'objets contenant des pointeurs sur les fonctions adéquates, mais le mécanisme reste lourd à réaliser, et on réinvente ce que C++ propose d'emblée.

# Les fonctions virtuelles(1)

- En raison des problèmes énoncés précédemment, C++ propose le mécanisme de **fonction virtuelle** (hérité de Simula) pour supporter la liaison tardive de méthodes.
- Les fonctions virtuelles vont permettre de conserver l'information de type même après la conversion d'objets dérivés en objets de leurs classes de base.

```
class Personne {  
    string nom;  
    string prenom;  
    public:  
        virtual char* comment_tu_t_appelles(); // retourne nom et prénom  
        char* get_prenom(); // retourne le prénom  
        virtual void afficher(); };
```

## Les fonctions virtuelles(2)

- La classe de base (super-classe) doit donner une définition des fonctions virtuelles. Si une classe dérivée ne redefinit pas une fonction virtuelle, la fonction appelée sera celle de son ancêtre le plus proche.

### Attention :

Ne pas qualifier “**inline**” les fonctions virtuelles, car de façon interne tous les compilateurs ont besoin de l'adresse de ces fonctions.

## Exemples(1)

```
class Forme
```

```
{ ...
```

```
public:
```

```
    virtual void deplacer(int x, int y);
```

```
    virtual void dessiner() = 0;
```

```
    virtual void pivoter(double angle) = 0 ;
```

```
};
```

```
void Forme::deplacer(int, int) {
```

```
    printf(" I don't know how to move! ( this:%x)\n", this);
```

```
};
```

```
Forme* s;
```

```
s->deplacer(2,3);
```

```
//ok
```



## Exemples(2)

```
class Polygone : public Forme {      ...  
    public:  
    void deplacer(int, int) { ... };  
    void dessiner() ;  
    void pivoter(double angle) ;  
};
```

```
class Cercle : public Forme {      ...  
    public:  
    void deplacer(int, int) { ... };  
    void dessiner();  
    void pivoter(double angle) ;  
};
```

## Exemples(3)

- Supposons définie une classe **Liste\_de\_forme**, avec une méthode ajouter (forme\*) pour ajouter une forme dans une liste.

```
Liste_de_forme elist;  
elist.ajouter(new Polygone(...));  
elist.ajouter(new Cercle(...));
```

Faire pivoter toutes les formes de la liste d'un angle *alpha* :

```
for (Forme* p = elist.first(); p != 0 ; p = elist.next())  
p->pivoter(alpha);
```

# Limitations des fonctions virtuelles

- Pas de spécialisation des signatures de méthodes

**Exemple :**

```
class A {  
    virtual void foo(A);  
};
```

```
class B : public A {  
    void foo(B);  
};
```

**La classe B répond aux invocations de signatures : foo(A) et foo(B)**

- La solution C++ est d'utiliser le **downward cast**

# Le downward cast

- *Le downward-cast est une technique non-sûre*

## Exemple :

```
Personne* pt_pers = new Personne;
```

```
Personne* pt_enf = new Enfant;
```

```
Enfant* bebe;
```

```
bebe = (Enfant*) pt_pers;    // Non sûre !
```

```
bebe = (Enfant*) pt_enf    // Ok, pt_enf est dynamiquement  
                           // un Enfant
```

## Solution C++

```
class A {  
    virtual void foo(A);  
};
```

```
class B : public A {  
    void foo(A a);  
};
```

```
void B::foo(A a) {  
    B* b; b=(B*)a;  
    ...};
```

- La classe **B** répond à l'invocation de la signature `foo(A)`; mais `foo` est redéfinie pour faire un downward cast du paramètre d'un **A** vers un **B**
- **Attention :** Cela fonctionne, mais ce n'est pas toujours sûr.

## Solution C++...

### ◆ **a1->foo(a2);**

- **Cas 1 :** a1 est statiquement et dynamiquement un **A**, alors invocation de **A::foo(A)**
- **Cas 2 :** a1 est statiquement un **A** et dynamiquement un **B**,  
a2 est statiquement un **A** et dynamiquement un **B**, alors invocation **B::foo(A)** => liaison tardive + spécialisation par **downward cast**
- **Cas 3 :** a1 est statiquement un **A** et dynamiquement un **B**,  
a2 est statiquement et dynamiquement un **A**, alors invocation de **B::foo(A)** => **Attention, aux risques d'erreurs**

# L'héritage multiple

- Une classe dérivée peut avoir plusieurs classes de base. On parle alors d'héritage multiple

```
class A {...};
```

```
class B {...};
```

```
class C : public A, public B  
    {...};
```

- L'ordre de citation des classes de base n'est pas significatif, sauf éventuellement pour l'appel des constructeurs, destructeurs.
- Une même classe ne peut apparaître plusieurs fois comme classe de base *directe* d'une classe dérivée. Par contre, elle peut être classe de base *indirecte* plusieurs fois.

```
class C : A, A {...};
```

// illégal, ambiguïtés insolubles

# Les ambiguïtés de nommage

```
class B : public A {...int x;... int f();...};
```

```
class C : public A {...int x; ...int f();...};
```

```
class D : public B, public C {...};    // légal
```

- Les ambiguïtés d'accès aux membres des classes qui portent le même nom, sont résolues en spécifiant le chemin d'accès complet :

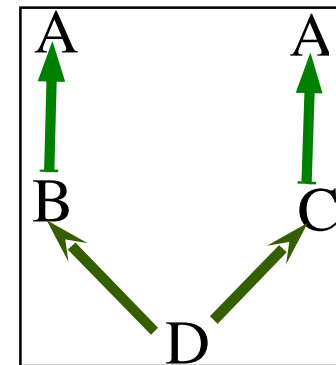
```
D compliq;           int r = compliq.B::f();
```

```
compliq.B::x = 3;     int s = compliq.C::f();
```

```
compliq.C::x = 4;
```

```
compliq.x;            // problème, ambiguïté. Quel x?
```

```
compliq.f();          //problème, ambiguïté. Quelle fonction f()?
```



- Il est conseillé de redéfinir localement toutes les méthodes pour lesquelles il y a une ambiguïté.



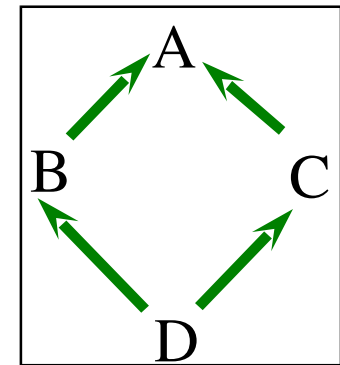
## Sous-classes communes

- Lorsqu'une classe de base apparaît plusieurs fois dans une dérivation, on peut spécifier qu'une seule instance de cette classe sera créée lors de la construction d'un objet dérivé.

```
class A {...};
```

```
class B : virtual public A {...};
```

```
class C : virtual public A {...};
```



```
class D : virtual public A, public B, public C, {...}; // légal
```

# Les classes Abstraites

- Une classe abstraite est une classe dont il ne peut exister aucune instance.
- Cela permet de décrire un modèle abstrait, regroupant un certain nombre de caractéristiques communes, qui va servir de base à la création de nouvelles classes.
- Une classe abstraite se distingue par l'existence d'au moins une fonction virtuelle pure.

# Exemples

```
class Forme {  
  
    char* couleur;  
    point centre;  
    public:  
  
        char* Couleur() { return couleur;};  
  
        void deplacer(point p ) { centre=p; dessiner();};  
  
        virtual void dessiner()= 0;                //méthode virtuelle pure  
  
        virtual void pivoter(double angle) = 0 ;    //méthode virtuelle pure  
  
};
```

## Exemples...

- On ne peut pas créer d'instance d'une classe abstraite; on peut juste manipuler des pointeurs ou des références sur des objets qui sont en réalité des objets d'une classe dérivée.

**Forme** s; // erreur : variable d'une classe abstraite

**Forme\*** fc = **new** Cercle(...);

**Forme\*** fp = **new** Polygone(...);

**Polygone** p;

**Forme&** ref\_p = p;

# La Généricité (1)

- ◆ **Les Templates** : Introduction de la généricité à partir de la version 3.0 de C++ ATT&T.

```
template< class T>
class Pile {
    T* v;   T* p;
    int sz;

public :
    Pile (int s) { v = p = new T[sz=s]; }
    ~Pile() { delete[] v; }
    void push(T a) { *p++ = a; }
    T pop() { return *--p;}
    int size() { return p-v; } };

```

## La Généricité (2)

```
Pile<char> sc(100); // pile de 100 caractères  
sc. push('a');
```



```
class Pile_char {  
    char* v;    char* p;  
    int sz;  
public :  
    Pile_char (int s) { v = p = new char[sz=s]; }  
    ~Pile_char() { delete[] v; }  
    void push(char a) { *p++ = a; }  
    char pop() { return *--p;}  
    int size() { return p-v; }    };
```

# La gestion des exceptions (1)

## ◆ Mise en place des exceptions : définition et invocation

```
class Vecteur {  
    int* p;  
    int sz;  
    public :  
        class Range { };    // classe d'exception de Vecteur  
    ...  
    int operator[] (int i);  
};  
  
int Vecteur::operator[] (int i) {  
    if (0<=i && i <sz) return p[i];  
    throw Range(); }  

```

## La gestion des exceptions(2)

### ◆ Récupération des messages d'erreur dans un programme

```
void foo(Vecteur v) {  
    // ...  
    try {  
        v[v.size()+10]; // instruction déclenchant une erreur  
    }  
    catch (Vecteur::Range) {  
        // récupération du message d'exception 'the handler'  
        do_something_else();  
    }  
}
```



# Extensions

# Références croisées

- Quand deux classes se référencent mutuellement, il faut explicitement indiquer que l'une existe avant de pouvoir l'utiliser.

```
class A;  
class B {  
    A* _a;  
}  
class A {  
    B* _b;  
}
```

Ou directement :

```
class B {  
    class A* _a; }
```

Il est souvent utile de déclarer automatiquement toutes les classes définies dans un fichier en-tête au début de ce fichier

# Conversions de types (user-defined)

Les classes fournissent un mécanisme de conversion entre les objets d'une classe et un type presque quelconque.

## ◆ Par constructeur (conversion d'un type donné vers une classe)

```
class X {  
    X(int);                // spécifie une conversion de int vers X;  
    X( const char*, int =0); // const char* -> X  
}
```

## ◆ Par opérateur (conversion d'une classe vers un type donné)

```
class X { ...  
    operator T();          // définit une conversion d'un objet de  
};                          // la classe X vers un objet de type T
```

## Exemple : Conversions par constructeurs

```
class X {  
    X(int);           // spécifie une conversion de int vers X  
    X( const char*, int =0); // const char* -> X  
};
```

```
f (X arg) {  
    X a = 1;           // <=> X (a(1))  
    X b = "toto";      // X b("toto", 0)  
    a = 2;             // a = X(2);  
    f(3);              // f (X(3))  
};
```

**Limitation :** un seul niveau de conversion, si aucun constructeur ne correspond, il n'y a pas d'application récursive du procédé.

## Exemple : Conversions par opérateurs

```
class X {  
    operator int();  
  
    {  
        X a;  
        int i = int(a);  
        i = (int) a;  
        i = a;  
        i = (a) ? 1+a : 0;  
    };  
};
```

- Dans les 4 cas, l'opérateur de conversion est utilisé.

# Conflit entre constructeur et opérateur de conversion

- Un opérateur de conversion et un constructeur définissant une conversion équivalente, ne peuvent coexister.

```
class A {  
    A(const B&);  
    ...  
};
```

```
class B {  
    operator A();    // erreur, une conversion d'un B vers un A  
                    // existe déjà.  
    ...  
};
```

# Résolution de la surdéfinition

## ➤ Problème de la recherche de l'opérateur adéquat

### ■ Conformance exacte

La signature de la fonction est exactement identique à l'appel.

### ■ Conversions standards

Conversions implicites ( [unsigned] int -> [unsigned] long, int -> real, float -> double, pointeur -> void\*, ...)

### ■ Application d'une conversion utilisateur

La résolution de la surcharge ne distingue pas un argument et une référence sur un objet.

Rectangle foo(Rectangle&) ; Rectangle foo(Rectangle); //ambiguïté

**Remarque :** les conversions peuvent rester privées à la classe, ce qui limite les problèmes.

# Friends, Membres et Conversions

```
class X
{
    ...
    X(int);

    int m1();
    int m2() const;

    friend int f1(X&);
    friend int f2(const X&);
    friend int f3(X);
    ...
};
```



## Friends, Membres et Conversions (2)

**void g()**

{

1.m1();     // erreur: **X**(1).m1() non appliqué

1.m2();     // erreur: **X**(1).m2() non appliqué

};

**void h()**

{

f1(1);     // f1(**X**(1))

f2(1);     // f2(**X**(1));

f3(1);     // f3(**X**(1));

};

# Les performances

## ◆ La désencapsulation ?

- "Overhead" introduit par l'accès aux champs d'une classe (appels de fonctions trop importants)

## ◆ La copie d'objets ?

- "Overhead" introduit par la construction de petits ou gros objets (passage par référence)
- "Garbage collector" : récupérer la place allouée pour les objets dynamiques
- Contrôler l'allocation et la désallocation des objets (attention aux copies cachées)

# Création & copie d'objets

La création d'un objet peut être :

- automatique (dans la pile) : **Rectangle r1;**
- dynamique (alloué sur le tas) : **Rectangle\* ptrec = new Rectangle;**
- automatique, temporaire (dans la pile) : **void f(Rectangle r) { ... }**

# Copies d'objets

Un objet d'une classe peut être copié de deux manières différentes :

- **par affectation**

```
Obj o1, o2;  
o1 = o2;
```

- **par initialisation : à la création**

```
Obj o1;  
Obj o2 = o1;    // ces deux écritures sont équivalentes  
Obj o3(o2);
```

## Copies d'objets (2)

- Lors des passages d'arguments

```
int f(Obj o);  
Obj o1;  
f(o1);
```

- En valeur de retour

```
Obj f() {  
    Obj o;  
    ...  
    return o; }
```

**Pour ces deux types d'initialisation d'objet, on utilise respectivement l'opérateur = et le constructeur prenant une référence à un objet de la classe.**

# Méthodes engendrées automatiquement

En cas de besoin, si elles n'existent pas, ces deux méthodes sont créées automatiquement et leur sémantique est :

- copié champ à champ pour l'opérateur =
- initialisation champ à champ pour le constructeur
  - » copie pour les champs simples
  - » initialisation pour les champs objets

**Cependant, attention aux problèmes de ramasse-miettes.....**

# Affectation

## Exemple :

```
void f()
{
    string s1("San Francisco");
    string s2("Paris");
    s1 = s2;
} // appel du destructeur pour s1 et s2
// destruction double sur la chaine "Paris", résultats étranges!...
```

## Affectation(2)

### ➤ Surcharge de l'opérateur affectation

```
string& string::operator=(const string& s);  
{  
    if (this != &s) { // attention au cas s = s  
        delete[] p;  
        p = new char[size = s.size+1];  
        strcpy(p,s.p);  
    }  
    return *this;  
};
```



# Initialisation

## Exemple :

```
void f()
{
    String s1(10);

    String s2 = s1;           // initialisation, et non pas affectation
}
```

## Initialisation (2)

### ◆ Surcharge du constructeur

```
String::String(const String& s);  
{  
    p = new char[size = s.size+1];  
  
    strcpy(p, s.p);  
};
```

# Résumé

```
class X {  
    ...  
    public :  
  
        X(something);           // constructeur  
  
        X(const X&);           // initialisation : copy constructor  
  
        X& operator=(const X&); // affectation : cleanup and copy  
  
        ~X();                  // destructeur : cleanup  
  
    ...  
};
```

# La classe STRING (version finale)

```
class String
{
    char* p;
    int size;

    public:
        String(int sz=1);
        String(const char* s );           // constructeur : type conversion
        String(const string& );           // constructeur : initialisation
        ~String() { delete p;};
        int length() {return size;}
        String& operator=(const String&); // affectation
}
```

## La classe STRING (version finale)

**char** operator[](int i);

**Friend** String operator+(**const** String&, **const** String&);

String& operator+=(**const** String&);

**Friend int** operator==(const String&, const String& );

**Friend int** operator!=(const String&, const String& );

**Friend int** operator<(const String&, const char\*);

**Friend int** operator>=(const String&, const char\*);

... };

# Opérateurs Friends

```
int operator==(const String& a, const String& b){  
    return strcmp(a.p, b.p) == 0;    }
```

```
int operator!=(const String& a, const String& b){  
    return strcmp(a.p, b.p) != 0;    }
```

```
int operator==(const String& a, const char*& b){  
    return strcmp(a.p, b) == 0;    }
```

```
int operator!=(const String& a, const char* b){  
    return strcmp(a.p, b) != 0;    }
```

**Plus précisément . . .**

## Static : Variables & Fonctions

- Un objet est créé lors de sa définition et détruit lorsque son nom est en dehors de son champs de définition.
- Un objet global est créé et initialisé une seule fois et “vit” jusqu'à la fin du programme. C'est le cas des objets déclarés **static**.

```
void f() {  
    int b=1;                // initialisé à chaque appel de f()  
    static int c=1;         // initialisé une seule fois  
  
    cout << " b= " << b++;  
    cout << " c= " << c++ << "\n"; }  
}
```

- Une variable **static** non explicitement initialisée est implicitement initialisée à 0.



## Static : Variables & Fonctions

- Un nom peut être fait local à un fichier en le déclarant **static**.

//fichier1.c

```
static int a = 6;  
static int f() { ... }
```

//fichier2.c

```
static int a = 7;  
static int f() { ... }
```

- Lorsque des variables et des fonctions sont explicitement déclarées **static**, un fragment de programme est plus facile à comprendre (on n'a pas besoin de regarder ailleurs).
- L'utilisation de **static** pour les fonctions a un impacte positif sur l'overhead dû aux appels de fonctions.

## Static : Membres de classe

- **Une classe est un type**, pas un objet de données. Chaque objet de la classe a une copie des membres données de la classe. **Mais**, certains types seront plus élégamment implémentés si **tous** les objets de la classe partagent certaines données.

```
Class task {  
    // ...  
    task* next;  
    static task* task_chain;  
    void schedule (int);  
    // ...  
};
```

- Il y aura **une seule copie** de task\_chain, pas une copie par objet.

## Static : Membres de classe

- A moins d'être déclaré **public**, task\_chain n'est pas accessible.

task :: task\_chain

- Dans une fonction membre, on peut référencer task\_chain directement par son nom (pas besoin de spécifier la classe).
- L'utilisation de membres **static** peut réduire considérablement le besoin de variables globales.

# Fonctions Inline

- A chaque appel d'une fonction déclarée **inline**, le compilateur génèrera le code de la fonction à l'endroit où la fonction est appelée.
- Une fonction membre définie (non seulement déclarée) dans la classe est implicitement **inline**.
- Une fonction membre peut aussi être déclarée **inline** hors de la classe.

## Fonctions Inline

```
class char_stack {  
    int size;  
    char* top;  
    char* s;  
  
    public:  
        char pop ( );  
        // . . .  
};  
inline char char_stack :: pop( )  
{  
    return *--top;  
};
```

# Fonctions Inline

```
struct x {  
    int f ( ) { return b; }  
    int b;  
};
```

est équivalent a :

```
struct x {  
    int f ( );  
    int b;  
};  
inline x: :f( ) { return b; }
```

# Surcharge de fonctions

- Pour protéger le programmeur contre une réutilisation accidentelle d'un nom, avant de surcharger une fonction, il doit d'abord la déclarer en utilisant le mot clé **overload**.

**overload** print ;

**void** print (**int**) ;

**void** print (**char\***) ;

- Le compilateur distinguera entre les fonctions grâce au type du paramètre lors de l'invocation de la fonction.

# Surcharge de fonctions

- Dans certains cas, une conversion explicite des types est nécessaire.

**overload**    print (**double**),    print (**long**);

```
void f (int a)
{
    print (a);
}
```

- L'ambiguïté peut être levée en utilisant une conversion explicite du type :

print (**long**(a));    ou    print (**double**(a));



# Friends & membres

- Quand doit-on utiliser des membres et quand utiliser des **friend** pour accéder à des parties privées d'un type (classe) utilisateur ?
- Certaines opérations doivent être des membres :
  - constructeurs, destructeurs,
  - fonctions virtuelles
- En général, on a le choix.

# Friends & membres

## Exemple :

```
class X {  
    // ...  
    X(int);  
    int m ( );  
    friend int f(X&);  
}
```

- A priori, il n'y a aucune raison de choisir un **friend** `f(X&)` au lieu d'un membre `X::m( )` et vice-versa pour implémenter une opération sur un objet de la classe `X`, **mais ...**

## Friends & membres

- Un membre ne peut être invoqué que pour un “objet réel” alors qu'un **friend** peut être appelé pour un objet crée par conversion implicite de type.

```
void g( ) {  
    1.m( );           // erreur  
    f (1);            // f ( X(1) );  
}
```

- Une opération qui modifie l'état d'un objet d'une classe doit donc être un membre, et **non pas** un **friend**.
- Les opérateurs qui nécessitent des opérandes *lvalue* (expression référant un objet telle que **\*p[a+10]**) (**=**, **\*=**, **++**, ...) sont en général naturellement définies comme des membres pour les types users.

# Registres

- Les (petits) objets sont accédés nettement plus rapidement lorsqu'ils sont placés dans un registre.
- Idéalement, le compilateur déterminera la stratégie optimale pour utiliser tout registre disponible sur la machine (tâche non triviale).
- Il est possible d'aider le compilateur en déclarant un objet **register**.

```
register int i;  
register point cursor;  
register char* p;
```

# Registres

## Attention:

- Les déclarations de registres ne doivent être utilisées que lorsque l'efficacité est **vraiment importante**.
- Déclarer toutes les variables de type **register** va :
  - encombrer le texte du programme,
  - augmenter la taille du code (c'est une possibilité),
  - augmenter le temps d'exécution (plusieurs instructions sont nécessaires pour mettre un objet dans un registre et l'en retirer).

# Registres

- Il est impossible de :
  - récupérer l'adresse d'un nom déclaré comme **register**,
  - Déclarer un nom de type **register** comme global

# Conclusion

Le concept de types abstraits de données facilite la définition d'un ensemble d'opérations (opérateurs et fonctions) pour un type de données particulier.

Il offre :

- une uniformité dans la manipulation (cacher l'implémentation physique),
- une certaine modularité,
- un contrôle d'accès aux membres d'une classe,
- un contrôle des opérations de création, destruction, initialisation et affectation,
- facilité la maintenance du code, et sa réutilisation.

## Bilan : Avantages de C++

### ◆ Sur-ensemble de C

- Accepte la plupart des programmes C standards (Norme ANSI)

### ◆ Bonnes performances

- Similaires à C et très supérieures à Smalltalk, Eiffel,...

### ◆ Popularité - Standard de facto

- Utilisateur de C et de l'approche objet
- Multi-vendeurs (AT&T, Apple, Appolo, Sun, ...)
- De plus en plus d'environnements de développement et de bibliothèques C++



## Bilan : Inconvénients de C++

### ◆ Langage orienté objet hybride

- Permet de dériver vers une programmation classique
- Possibilité de briser l'encapsulation

### ◆ Ramasse-miettes manuel

- Surcharge des programmes