

++C pour les ZZ

Tout ce
qu'il
faudrait
savoir !!!

Cours original OBC
Modifications par D. Hill
et C. Duhamel



AVERTISSEMENT



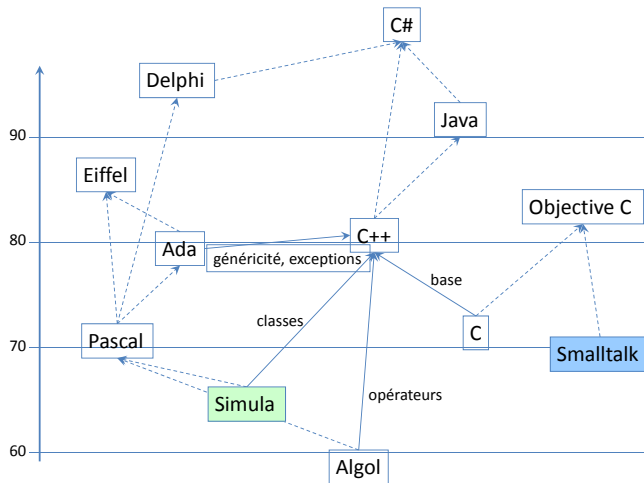
THESE SLIDES ARE RATED ZZ

- Le support original est présenté dans un cours
 - Certains transparents comportent **volontairement** des erreurs. Certaines peuvent aussi être involontaires
 - Il est fortement recommandé d'essayer les exemples proposés.
- La page de garde est honteusement copiée des ouvrages des éditions First.
 - Ce cours est libre et réutilisable pour tout à chacun sous la licence EMAILCOUCOUWARE ;-)



Caractéristiques générales

- Langage orienté objet
 - Classes au sens du langage SIMULA
 - Variables non objet
 - Fonctions
- Support de la généricité
 - Fonctions
 - Classes
- Traitement d'exceptions
 - Mécanisme évolué de gestion des erreurs



C / C++ / ++C ?

Ajouts par rapport au C

- ✓ Nouveau type de commentaires
- ✓ Surcharge des fonctions
- ✓ Valeur par défaut de paramètres
- ✓ Paramètres muets
- ✓ Constantes revues
- ✓ Type référence
- ✓ Variables de boucles
- ✓ Nouvelle gestion de la mémoire
- ✗ Opérateurs de transtypage efficaces
- ✗ Gestion des types à l'exécution en ZZ3

Commentaires

- Monoligne

```
// Fin de la ligne en commentaire
```

- Attention à ne pas les utiliser en C (sauf C99) !

- Plusieurs lignes à la manière du C

```
/* Début de section en commentaire
   // commentaire monoligne imbriqué
*/ Fin du commentaire
```

- Non imbriquables

```
/// pour les indications sur votre code
/* */ pour « virer » des portions de code
```

Surcharge

- Forme faible du polymorphisme

⇒ Fonctions de mêmes noms mais avec signatures différentes

- Exemple

- Système de gestion d'interface utilisateur
- Fenêtres repérées par :
 - Identificateur numérique
 - Chaîne de caractères
- Fonctions récupérant un pointeur sur fenêtre

```
Window* GetWindow(int identNumerique);
Window* GetWindow(const char *nom);
```

Paramètres avec valeur par défaut

- Motivation

« simplifier les appels en omettant les paramètres lorsque la valeur de ceux-ci est la plus courante »

- Syntaxe

- Préciser la valeur par défaut dans la déclaration
- Ne pas la rappeler dans la définition

Exemple : système de fenêtrage

- Affichage de fenêtre avec 2 paramètres
 - Pointeur vers la fenêtre (sans cesse différent ☺)
 - Mode d'affichage
 - Plein écran (peu courant)
 - Iconifiée (peu courant)
 - Taille spécifiée dans les ressources (le plus courant !)

• Déclaration

```
void ShowWindow(Window *toShow, int mode=DEFAULT_SIZE);
```

• Définition

```
void ShowWindow(Window *toShow, int mode) {  
    // Code d'implémentation  
}
```

Paramètres anonymes

- Motivation : avoir des paramètres muets
 - Compatibilité d'appel d'une version sur l'autre
 - Eviter les warnings de compilation
 - Respect de conventions (`main`, par exemple)

• Syntaxe

```
fonction(type param1, typemuet , type param2)
```

• Exemple :

- En C++, `main` prend au moins 2 paramètres
- Si vous ne les utilisez pas ⇨

```
int main(int, char **)
```

Constantes

- Utilisation de `const` limitée en C
- Etendue en C++
 - Véritables constantes numériques

```
const int TAILLE=5;  
int tableau[TAILLE];
```

• Conséquence

Plus jamais de `#define` pour définir des constantes en C++ !!!

- Autres applications avec les objets !

VARIABLES DE BOUCLE

- Déclarer un compteur de boucle dans le `for`
- Variable limitée à la boucle
- Impossible de tester la valeur en sortie de boucle

```
int i=5;  
int j=2;  
for (int i=0;i<10;i++) {  
    j += i; // i de la boucle  
}  
cout << i << endl;  
// i=5 !!! Retour au i global
```

Référence (1)

- Motivation
 - Uniquement le passage par valeur en C
 - Que faire pour les paramètres en mode out ou in/out ?
⇨ Passer un pointeur !
 - Conséquences
 - Code peu lisible
 - Source d'erreurs
- Référence
 - Pointeur masqué
 - Simule le passage par référence
 - Utilisation d'une référence identique à une variable

Référence (2)

```
void swap(int *a, int *b) {  
    int c=*b;  
    *b=*a;  
    *a=c;  
}
```

```
int main(int, char **) {  
    int i=5;  
    int j=6;  
  
    swap (&i, &j);  
  
    return 0;  
}
```

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

```
int main(int, char **) {  
    int i=5;  
    int j=6;  
  
    swap (i, j);  
  
    return 0;  
}
```

Référence (3)

- Avantages
 - ↳ Efficacité ⇨ passage d'adresse plutôt que copie
 - ↳ Code très lisible
 - ↳ Appel facilité
 - ↳ Moins d'erreurs
- Inconvénients
 - ↳ Syntaxe ambiguë du fait de l'utilisation de `&`
 - ↳ Notion parfois difficile à appréhender
- Différences majeures avec un pointeur :
 - Une référence est toujours liée à une variable
 - La référence nulle n'existe pas !

PROGRAMMATION ORIENTEE OBJET & C++

POO en C++

- Transposition des classes
- Relations entre classes
 - Agrégation
 - Héritage
 - Association ?
- Généricité
 - Fonctions
 - Classes
- Exceptions
- Éléments avancés ???

Transposition des classes

- Syntaxe générale
- Déclaration
 - Attributs et prototypes des méthodes
- Définition/implémentation des méthodes
- Raffinements successifs
 - Méthodes inline
 - Structuration du code source
- Cycle de vie des objets
 - Construction
 - Appel de méthodes
 - Destruction

Déclaration de la classe

- Débute par `class` NomDeLaClasse
- Attributs
- Prototypes des méthodes
- Modificateurs d'accès :
 - `public` : membre accessible universellement
 - ⇒ réservé exclusivement aux méthodes de l'interface
 - `private` : visible des méthodes de la classe
 - ⇒ tous les attributs
 - ⇒ méthodes non destinées à l'utilisateur

Classes en C++

Point
-x : entier
-y : entier
-compteur : entier
+getX() : entier
+getY() : entier
+deplacerDe(dx : entier, dy : entier)
+deplacerVers(px : entier, py : entier)
+getCompteur() : entier

```
class Point
{
public:
    Point(int px, int py);
    int getX() const;
    int getY() const;
    void deplacerDe(int dx, int dy);
    void deplacerVers(int px, int py);
    static int getCompteur();
private:
    int x;
    int y;
    static int compteur;
};
```

Mot clef `static`

Membre de classe !

TRES IMPORTANT !

```
// Définition des méthodes
Point::Point(int px, int py)
{
    x=px;
    y=py;
    ++compteur;
};
int Point::getX() const
{
    return x;
}
void Point::deplacerDe(int dx, int dy)
{
    x+=dx;
    y+=dy;
}
static int Point::getCompteur ()
{
    return compteur;
}
static int Point::compteur=0; // Attribut de classe
```

Définition des méthodes
et variables de classe

Manipulation basique

```
int main(int, char **)
{
    Point p(10, 10);

    std::cout << p.getX() << " ";
    std::cout << p.getY() << std::endl;

    p.deplacerVers(2, 4);

    return 0;
}
```

1^{ère} amélioration : méthodes inline

- Motivation
 - Dommage d'utiliser un appel de méthode pour
 - Traitement simple
 - Récupérer la valeur d'un attribut
 - Hors de question de mettre un attribut en public
- Solution
 - Méthode inline : développée comme une macro
- Avantage : Rapidité d'exécution
- Inconvénients
 - Augmentation taille exécutable ⇒ méthodes courtes
 - Implémentation dans la déclaration



Implémentation inline

- Dans la déclaration

```
int getX() const
{
    return x;
}
```

inline implicite

- Utilisation du mot clef `inline`

```
inline int getY() const ; // Déclaration
inline int Point::getY(void) const // Définition
{
    return y;
}
```

Structure du code source

- Fichier entête
 - Déclaration de classe
 - Méthodes inline

```
#ifndef __NomClasse_HXX__
#define __NomClasse_HXX__

// inclusions si necessaires
// declarations externes

class NomClasse
{
    // Proto methodes
    // Methodes inline
    // Attributs
};
#endif
```



- Fichier d'implémentation
 - Définition variables de classe
 - Définition méthodes

```
#include « NomDeLaClasse.hxx »

// Definition des variables
// de classe

// Definition des methodes
// externalisees
```



Cycle de vie des objets

- ① Construction
 - Allocation de mémoire
 - Appel d'un constructeur
- ② Vie
 - Appel des méthodes
- ③ Destruction
 - Appel du destructeur
 - Désallocation : rendu mémoire



Constructeur(s)

- Rôle : initialiser les objets
- Syntaxe
 - Même nom que la classe
 - Pas de type de retour
 - Surchargeable à volonté
 - Un élément particulier : la liste des initialisations

```
Point::Point(int px,int py) : x(px), y(py)
{
// ...
}
// Liste d'initialisations
```

Liste d'initialisations

[attribut(expression){, attribut(expression)}ⁿ]

- Expression pas limitée à une constante ou une valeur immédiate
 - Expressions arithmétiques
 - Fonctions
- Attributs initialisés dans l'ordre de déclaration
 - Sauf option g++
- Possibilité de mixer liste et code
- Liste d'initialisation traitée avant le code

Ordre des initialisations

Initialisation "complexe"

```
class Rationnel
{
public:
    Rationnel(int n=0, int d=1):
        deno_(d),
        nume_(n)
    {}
private:
    int nume;
    int deno;
};
```

Les initialisations doivent respecter l'ordre des déclarations !

```
Rationnel::Rationnel(int n=0,
                    int d=1):
    nume_(n), deno_(d)
{}
```

```
Point::Point(int px,int py):
    x(px),
    y(py),
{
    distance=sqrt(x*x+y*y);
}
```

Avec code

Ajout de distance

```
Point::Point(int px,int py):
    x(px),y(py),
    distance(sqrt(px*px+py*py))
{}
// Tout dans la liste
```

Création d'objets

- Trois classes d'allocation

Destruction automatique

1. Statique
 - variables globales
 - variables locales déclarées statiques
2. Automatique : variables locales sur la pile
3. Dynamiques : variables allouées sur le tas
 - Allocation avec **new** + constructeur
 - Destruction avec **delete**

Exemples d'instanciation

```
Point p1(120, 13); // p1 : objet statique
const int TAILLE=5; // constante entière

int main(int, char **)
{
    Point p2(12, 15); // p2 : objet point automatique
    Rationnel r1; // r1 : objet rationnel automatique
                  // initialisé avec valeurs / défaut
    Rationnel r5[TAILLE]; // Tableau de TAILLE Rationnel
                          // Tous initialisés avec défauts
    Rationnel r2(); // prototype de la fonction r2
                  // qui ne prend pas de paramètre et
                  // renvoie un Rationnel

    Rationnel *r;
    r = new Rationnel (10,25); // Créé un rationnel sur le tas;
    delete r; // Détruit un rationnel

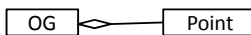
    return 0;
};
```

Méthodes constantes

- Déclarées avec le mot clef **const**
- Sur une méthode qui ne modifie pas les attributs de son objet
- Limité aux méthodes d'instance
- Avantages
 - callable sur un objet **const**
 - garantit que la méthode ne pourra modifier les attributs

Agrégation

- Un ou plusieurs objets dans un autre
- Trois cas :
 - Objet direct
 - Référence
 - Pointeur
- Provenance ou construction de l'objet
 - Objet construit par son agrégateur
 - Objet en provenance de l'extérieur
 - Recopié
 - Prise de pointeur ou de référence



La classe Point

```
#ifndef __POINT_HXX__
#define __POINT_HXX__
class Point
{
public:
    Point(int px=0, int py=0) :
        x(px), y(py) { };
    int getX(void) const
    {
        return x;
    }
    int getY(void) const
    {
        return y;
    }
    void deplacerVers(int versX,
                    int versY);
    void deplacerDe(int surX,
                    int surY);
private:
    int x;
    int y;
};
```

```
#include "Point.hxx"

void Point::deplacerVers(
    int versX, int versY)
{
    x = versX;
    y = versY;
}

void Point::deplacerDe(
    int surX, int surY)
{
    deplacerVers(x + surX,
                y + surY);
}
```

```
#ifndef __Objet_Graphique_HXX__
#define __Objet_Graphique_HXX__

#include "Point.hxx"

class ObjetGraphique
{
public:
    ObjetGraphique(int x, int y, int couleur=0, int epais=0) :
        pointDeBase_(x,y), couleur_(couleur), epaisseur_(epais)
    {}
    virtual void afficher(void) const=0 ;

protected:
    Point pointDeBase_;
    int couleur_;
    int epaisseur_;

public:
    enum { COULEURFOND=0 };
};

#endif
// Classe ObjetGraphique Entête
```

```

const Point & ObjetGraphique::pointDeBase(void) const
{
    return pointDeBase_;
}

int ObjetGraphique::couleur(void) const
{
    return couleur_;
}

int ObjetGraphique::epaisseur(void) const
{
    return epaisseur_;
}

virtual void ObjetGraphique::effacer(void)
{
    int sauveCouleur=couleur_;
    couleur_=COULEURFOND;
    afficher();
    couleur_=sauveCouleur;
}

void ObjetGraphique::deplacerVers(int versX, int versY)
{
    effacer();
    pointDeBase_.deplacerVers(versX, versY);
    afficher();
}

```

Classe ObjetGraphique
Définition

Héritage

- Dériver une classe d'une autre
- Modificateur conditionne la visibilité des attributs et méthodes de la classe mère

```
class derivee : modif base {, modif base}^n
```

	Fille	
Mère	Public	Private
Public	Public	Private
Protected	Protected	Private
Private	Non visible	Non visible

Protected ou private ?

- Modificateur d'accès **protected**
 - Visible de classe fille mais non de l'extérieur
- Vision classique de l'héritage
 - Attributs **protected** + héritage **public**
- Passer tous les attributs **private** en **protected** ?
 - ↳ Les attributs resteront accessibles directement en cas de dérivation
 - ↳ Dans le cas d'attributs nécessitant des mises à jour en cascade ou critiques, préférable d'utiliser des méthodes de la classe mère

L'héritage en **private** ?

- Une relation spéciale
 - « Est implémenté sous forme de »
- Avantages
 - Accès aux attributs de la classe d'implémentation
 - Un seul objet
- Inconvénients
 - Utilisation bizarre de l'héritage
 - Possibilité de réaliser cette relation avec de l'agrégation ... mais avec 2 objets !
- Conclusion

A mon avis à éviter

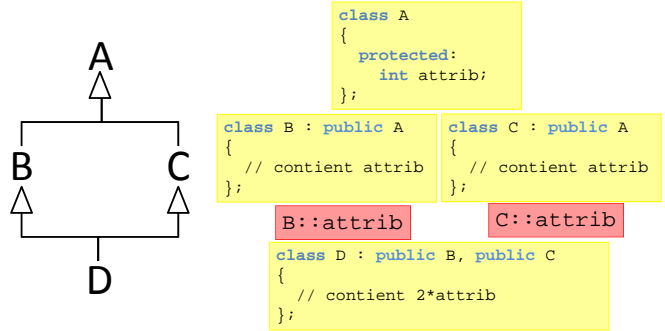
Polymorphisme

- Rendre une méthode polymorphe : **virtual**
 - Attention ! Virtuelle un jour, virtuelle toujours !
 - A redéfinir à chaque étape !
- Rendre une méthode abstraite
 - Utiliser conjointement **virtual** et **=0**
- Hériter d'implémentation de classe mère

```
virtual retour methode(signature) = 0;
```

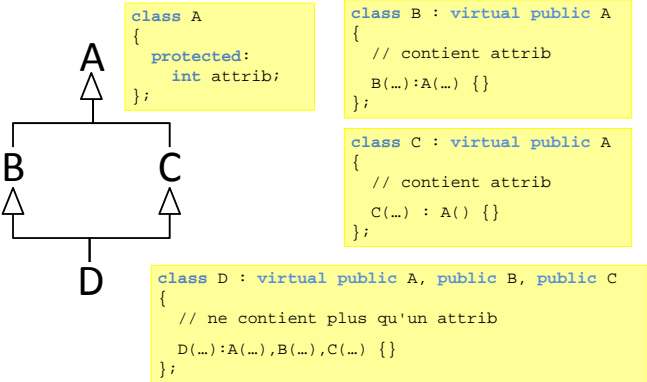
```
mère::methode(paramètres)
```

Héritage à répétition



Si attrib ne doit pas être dupliqué ?
Appeler le constructeur de A dans D ?

Héritage virtuel



```

const Point & ObjetGraphique::pointDeBase(void) const
{
    return pointDeBase_;
}

int ObjetGraphique::couleur(void) const
{
    return couleur_;
}

int ObjetGraphique::epaisseur(void) const
{
    return epaisseur_;
}

virtual void ObjetGraphique::effacer(void)
{
    int sauveCouleur=couleur_;
    couleur_=COULEURFOND;
    afficher();
    couleur_=sauveCouleur;
}

void deplacerVers(int versX, int versY)
{
    effacer();
    pointDeBase_.deplacerVers(versX, versY);
    afficher();
}

```

Classe ObjetGraphique
(Suite)

Classe Cercle

```

#ifndef __Cercle_HXX__
#define __Cercle_HXX__

#include "ObjetGraphique.hxx"

class Cercle : public ObjetGraphique
{
private:
    int rayon_;
public:
    Cercle(int x, int y, int rayon, int
couleur=0,
    int epaisseur=0) :
    ObjetGraphique(x,y,epaisseur,couleur),
    rayon_(rayon)
    {};
    void afficher(void) const;
};
#endif

```

Classe Ligne

```
#ifndef Ligne_HXX
#define Ligne_HXX

#include "ObjetGraphique.hxx"

class Ligne : public ObjetGraphique
{
protected:
    int longueur_;
    double angle_;
public:
    Ligne(int x, int y, int longueur, double angle,
         int couleur=0, int epaisseur=0) :
        ObjetGraphique(x,y,couleur,epaisseur),
        longueur_(longueur),
        angle_(angle)
    {}
    void afficher(void) const ;
};
#endif
```

Classe Chaîne

- Gérer dynamiquement la longueur de la chaîne de caractères

Chaîne
- taille : entier
- tab : tableau de char
+ Chaîne()
+ Chaîne(taille : int)
+ Chaîne(s : char *)
+ Chaîne(s : Chaîne)
+~Chaîne()
+ opérateur d'affectation
+ opérateur d'extraction de caractère
+ opérateur d'affichage

```
class Chaîne {
private:
    int taille;
    char* tab;
public:
    // la suite
}
```

```
Chaîne::Chaîne (int pTaille=0) :
    taille(pTaille ? pTaille+1 : 0) {
    if (taille)
        tab=new char [taille];
    else tab=0;
}
```

Constructeur par défaut
Taille d'allocation

```
Chaîne::Chaîne (const char *cstr) {
    if (cstr) {
        taille=strlen(cstr)+1;
        tab=new char [taille];
        strcpy(tab,cstr);
    } else {
        taille=0;
        tab =0;
    }
}
```

Autre constructeur
"chaîne C" => Chaîne

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

Destructeur

```
void Chaîne::afficher() {
    std::cout << tab << std::endl;
}
```



```
void Chaîne::remplacer(char * c) {
    strcpy(tab, c, taille-1);
    // un peu à l'arrache
}
```

```
int main(int, char **) {
    Chaîne s1;
    Chaîne s2("essai");
    Chaîne s3(s2);
    Chaîne s4 = s2;
    s1 = s2;
    s2.remplacer("oups");
    s1.afficher();
    // afficher les autres chaînes
    return 0;
}
```

Copie des objets par défaut
Affectation des objets par défaut

```
Chaîne (const Chaîne &uC) {
    taille=uc.taille;
    if (taille) {
        tab=new char [taille];
        strcpy(tab, uc.tab);
    }
}
```

Constructeur par recopie

```
Chaîne &operator=(const Chaîne &uC) {
    if (&uC != this) {
        delete [] tab;
        taille=uc.taille;
        if (taille) {
            tab=new char [taille];
            strcpy(tab, uc.tab);
        }
    }
    return *this;
}
```

Code commun à factoriser

Opérateur d'affectation

Renvoie une référence sur l'objet courant pour chaînage

Synthèse (1)

- Constructeur par défaut
 - Construire un objet non initialisé : un tableau, par ex
- Constructeur par recopie
 - Construction d'un objet par clonage
 - Passage par valeur
 - Renvoi d'un objet
- Opérateur d'affectation
 - Recopie d'un objet dans un autre au cours de sa vie
- Destructeur
 - Restauration des ressources prises par l'objet

Recopie et affectation par défaut

- En l'absence de définition explicite
- Copie binaire et brutale
 - Parfait pour classes simples
 - Inadapté dans tous les autres cas
 - En particulier pointeurs
 - Recopie des pointeurs et non des valeurs pointées
 - Accès en écriture concurrents sur même zones
 - Plusieurs libérations du même bloc

Initialisation ou affectation ? (1)

A partir du moment où l'on est dans une définition, c'est nécessairement une initialisation

T t1;	Constructeur par défaut
U u1(params);	Constructeur avec paramètres
T t2(t1);	Constructeur de recopie
T t3=t1;	Constructeur de recopie
T t4();	Fonction nommée t4 renvoyant un objet de type T sans paramètre
t1=t3;	Affectation

Initialisation ou affectation ? (2)

T t1;	T T(void);
U u1(params);	U U(liste de paramètres);
T t2(t1);	T T(const T&);
T t3=t1;	T T(const T&);
T t4();	T t4();
t1=t3;	T& T::operator=(const T&);

Du bon usage des références sur les objets

- Toujours passer les paramètres objets par référence
 - Évite un appel au constructeur par recopie
 - Garantit les appels polymorphiques
- Référence constante pour objets non modifiables
- Ne pas renvoyer de référence lorsque l'on doit renvoyer un objet !

Règles générales sur les opérateurs

- Deux grandes catégories
 - Méthodes
 - Fonctions externes
- Règle de bon sens
Un opérateur est mis en méthode si `this` a un rôle prépondérant par rapport à l'autre argument
- Conclusion
 - Opérateurs binaires en fonctions externes
 - Opérateurs unaires et toutes affectations en méthodes

Exemples d'opérateur

- Sortie sur flux <<
- Opérateur []
- Opérateur =
- Concaténation

Sortie sur flux : opérateur <<

- Surcharge de l'opérateur de sortie sur flux <<
`std::cout << chaîne;`
- Syntaxe générale

```
ostream & operator<<(ostream &, const Classe &)
```

- Remarques
 - Nécessaire de renvoyer ostream & pour chaînage
 - Passage de tous les paramètres en référence pour efficacité

Code pour operator <<

```
// Version sans friend
ostream & operator<<(ostream &o, const Chaîne &a)
{
    o << a.toCstr();
    return o;
}
```

```
// Version avec friend (vu plus tard)
// = accès possible aux membres de l'instance
ostream & operator<<(ostream &o, const Chaîne &a)
{
    o << a.tab;
    return o;
}
```

Opérateur []

- Extraire un élément d'une chaîne de caractères
- Méthode
- 2 versions nécessaires
 - Une version non constante qui renvoie une référence vers un caractère que l'on pourra modifier
 - Une version constante qui renvoie une référence constante vers un caractère non modifiable

Implémentation de l'opérateur []

```
char& Chaîne::operator[](int index)
{
    if ((index < 0) || (index >= taille))
    {
        cerr << "Erreur, index en dehors des limites" << endl;
        exit(1); // on verra comment mieux faire plus tard
    }
    return tab[index];
}

const char & Chaîne::operator[](int index) const
{
    if ((index < 0) || (index >= taille))
    {
        cerr << "Erreur, index en dehors des limites" << endl;
        exit(1); // meme remarque
    }
    return tab[index];
}
```

Utilisation

```
int main(int, char **)
{
    Chaîne chaîne1("toto");
    const Chaîne chaîne2("titi");
    char c;

    // Version non const
    chaîne1[2]='a';
    cout << chaîne1 << endl; // Imprimera "tato"
    cout << chaîne1[2] << endl; // Imprimera "a"

    // Version const
    cout << chaîne2[2] << endl; // Imprimera << "i"
    c=chaîne2[3]; // Pas de pbm
    chaîne2[1]=c; // Erreur de compilation car
                  // reference constante
    chaîne1[1]=c; // ok
    return 0;
}
```

Concaténation : opérateur +

- Opérateur + pour concaténer 2 chaînes
- Résultat souhaité :
`Chaîne a("vivement ");`
`Chaîne b("l'été");`
`cout << a+b << endl; // vivement l'été`
- Rôles symétriques de a et de b dans a+b => méthode
- Prototypage :

```
Chaîne operator+(const Chaîne&, const Chaîne&);
```

Implémentation de l'opérateur +

- Si l'on n'utilise pas une méthode, comment accéder aux membres `private` ?
- Solution : les amis (`friend`)
 - A spécifier dans la déclaration de la classe
 - Impossible donc d'en rajouter dans une classe déjà compilée
 - Classes ou fonctions
 - Accèdent à tous les membres même privés !

```
class A
{
    ...
    friend class B;
    friend void fonction(const A&, const B&);
};
```

Attention !
l'amitié n'est pas transitive !

De la bonne utilisation des friends

- Avantages
 - 👉 Possibilité de rendre amies des classes fortement liées (notion de package)
 - 👉 Efficacité du code
- Inconvénients
 - 👉 Violation du principe d'encapsulation
 - 👉 Possibilité de faire n'importe quoi au nom de l'efficacité
 - 👉 Attention à l'intégrité de l'objet lorsque l'on accède aux attributs !

Concaténer sans `friend`

- les méthodes suivantes sont disponibles
 - `int getTaille();` // renvoie la taille de la chaîne
 - `const char *toCstr();` // pointeur sur le tableau de chars // stockant la chaîne

```
Chaine operator+(const Chaine &a, const Chaine &b) {
    char *tab = new char [a.getTaille()+b.getTaille()-1];
    strcpy(tab, a.toCstr());
    strcat(tab, b.toCstr());
    Chaine temp(tab);
    delete [] tab;
    return temp;
}
```

- 👉 Aucune modification de `Chaine` nécessaire
- 👉 N'utilise que des méthodes
- 👉 Performance car 2 copies de chaînes

Concaténation avec `friend`

```
Chaine operator+(const Chaine &a, const Chaine &b)
{
    chaine c(a.taille+b.taille-2);
    strcpy(c.tab, a.tab);
    strcat(c.tab, b.tab);
    return c;
}
```

- 👉 Code plus efficace que le précédent
- 👉 Moins de méthodes requises que précédent
- 👉 Nécessite de connaître la logique interne de `Chaine` quand à l'attribut `taille`
- 👉 On a rajouté la ligne suivante dans la déclaration de `Chaine`

```
friend Chaine operator+(const Chaine &, const Chaine &);
```

GENERICITE

Généricité

- Implémentation de fonctions et classes paramétrées par des types ou des constantes
- Mécanisme orthogonal au paradigme objet

Fonctions paramétrées (1)

- Exemple : maximum de 2 nombres
- 1^{ère} solution : fonctions dédiées

```
int max(int a,int b)
{
    return ((a > b) ? a : b);
}
```

```
double max(double a,double b)
{
    return ((a > b) ? a : b);
}
```

- 👉 Vérification de types
- 👉 Code dupliqué
 - Erreurs
 - Taille de l'exécutable

Fonctions paramétrées (2)

- 2^{ème} solution : Macro

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

- 👉 Code réutilisé
- 👉 Performance
- 👉 Aucune vérification de types
- 👉 Syntaxe biscornue
- 👉 Taille de l'exécutable ?

Fonctions paramétrées (3)

- 3^{ème} (et bonne) solution : une fonction paramétrée

```
template <class T>
T max (const T&a, const T&b)
{
    return ((a > b) ? a : b);
}
```

- 👉 Code réutilisé
- 👉 Possibilité de mettre en `inline` automatiques pour performances
- 👉 Pas de conversions de types

Exemple

```
int i;
int j;
long k;
long z;
T a;
T b;
T c;

cout << max(i,j) << endl;
cout << max(k,z) << endl;
cout << max(i,k) << endl;
```

int max(int, int)

long max(long, long)

Erreur, car les 2 arguments doivent être de même type et il n'y a pas de conversion automatique

```
c = max(a,b);
```

Correct si les opérateurs de comparaison et d'affectation existent

Pile

- Structure de données classique : classe paramétrée
- Simplification : tableau de taille fixe pour stocker les éléments

```
void push(const T&); // Empilage d'un élément
const T& top(void) const; // Consultation de l'élément du dessus
void pop(void); // Dépileage d'un élément
```

```
#ifndef __PILE_HPP__
#define __PILE_HPP__

template <class T>
// template <typename T>
class Pile
{
public:
    Pile(int laCapacite=256) :
        capacite_(laCapacite),
        taille_(0)
    {
        tab_=new T[capacite_];
    }
    const T& top() const
    {
        return tab_[taille_-1];
    }
    bool isEmpty()
    {
        return (taille_==0);
    }

private:
    int taille_;
    int capacite_;
    T *tab_;
};
#endif
```

Organisation du code source

- Le code de la classe paramétrée n'est jamais compilé tel quel
- Il subit une première instantiation
- Il doit donc être placé entier dans le **.hpp**
Y compris les méthodes déportées qui sont placées après la déclaration !
- Comme ça, il est entièrement généré
- C'est totalement différent d'un code classique

Organisation typique

```
#ifndef __CLASSE_PARAM_HPP__
#define __CLASSE_PARAM_HPP__

template <parametre en template>
class PARAM
{
    // Déclaration de la classe
};

// Définition des méthodes déportées

#endif
```

```
#include <iostream>
#include "pile.hpp"
int main(int, char **)
{
    typedef Pile<int> PileEntiers;

    PileEntiers p1;

    p1.push(10);
    p1.push(20);

    while (!p1.isEmpty())
    {
        cout << p1.top() << endl;
        p1.pop();
    }
    return 0;
}
```

Instantiation !

```
typedef Pile<Point> PilePoints;
PilePoints pp;
pp.push(Point(12,32));
Point unPoint(2,3);
pp.push(unPoint);
```

Autre exemple

Constante en paramètre template !

```
#ifndef __PILE_HPP__
#define __PILE_HPP__

template <class T, const int TAILLE = 256 >
class Pile
{
public:
    Pile() : taille_(0) {};
    ~Pile() {};
private:
    T tab_[TAILLE];
};
#endif
```

Constante entière en template
Attention ! Certains compilateurs n'acceptent pas la valeur par défaut

- Tout le reste est identique

Avantages et inconvénients

- Tableau dynamique
- Tableau statique
- Possibilité de réallocation dynamique
- Rapidité d'exécution car la mémoire est allouée dans l'exécutable
- Taille de l'exécutable
- Taille de l'exécutable
- Lenteur inhérente au new
- Pas de réallocation dynamique possible

Fonction template très pratique

- Écriture de comparateurs
- A partir du moment où l'on dispose de == < ou >, on peut tout écrire :

```
template <class T>
bool operator <=(const T&a, const T&b)
{
    return ((a < b) || (a == b));
}

template <class T>
bool operator >=(const T&a, const T&b)
{
    return !(a < b);
}
```

Les exceptions

- Mécanisme de gestion des erreurs
- Inconvénients
 - 👉 Coûteux en CPU
 - 👉 Dur à programmer
- Avantages
 - 👉 Rigoureux
 - 👉 Ne peut être ignoré !
 - ⇒ Toute exception ignorée résulte en la terminaison du programme !

Mécanismes classiques (1)

- Fonction renvoyant un statut
 - 👉 Facile à mettre en œuvre
 - 👉 Permet un diagnostic avancé
 - 👉 Type de retour monopolisé par le statut
 - 👉 Rien n'oblige l'utilisateur à vérifier le statut
 - 👉 switch sur le type de retour
- Variante : variable globale positionnée
 - Type de retour non monopolisé
 - Conflits possibles sur la variable
 - Énormément de valeurs possibles

GESTION DES ERREURS : EXCEPTIONS

```
int fonction(parametres formels)
{
    if (ConditionErreur1)
        return CONSTANCE_ERREUR1;
    if (ConditionErreur2)
        return CONSTANCE_ERREUR2;
    ...
    return CONSTANCE_SUCCES;
}
...
switch (fonction(parametres effectifs))
{
    case CONSTANCE_ERREUR1:
        ...
        break;
    case CONSTANCE_ERREUR2:
        ...
        break;
    case SUCCES:
        ...
        break;
}
```

Exemple de mécanismes classiques

Mécanismes classiques (2)

- Message d'erreur et terminaison
 - 👉 Possibilité de terminer proprement avec la mise en place de fonctions de terminaison (atexit)
 - 👉 Termine toujours le programme
 - 👉 Impossible d'adapter le fonctionnement du programme
 - 👉 Rendu d'allocation dynamique ?

```
if (condition erreur)
{
    cerr << "Message d'erreur" << endl;
    exit(1);
}
```

Exceptions

- Lancement d'exceptions
 - Par fonctions ou méthodes
 - Théoriquement tout type de données
 - Habituellement des classes spécialisées
- Traitement des exceptions
 - Blocs de codes surveillés
 - Gestionnaires d'exceptions : code spécialisé

Une exception non traitée entraîne l'arrêt du programme

Exemple

```
class Chaîne
{
public:
    class ExceptionBornes {};
    ...
    char &operator[](int index)
    {
        if ((index < 0) || (index > taille_))
            throw ExceptionBornes();

        return tab_[index];
    }
};

try
{
    c = chaîne[3];
}
catch (Chaîne::ExceptionBornes &e)
{
    // traitement
}
```

Lancement de l'exception c'est un **objet**

Début du bloc d'instructions surveillées

Traite exceptions

Type d'exception traitée

Le type des exceptions

- Éviter d'encombrer l'espace de nommage
 - Classes imbriquées
- Exceptions hiérarchisées
- Bibliothèque standard du C++

```
class exception {
public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

- Bon comportement = dériver de exception
 - #include <exception>

Exemple de hiérarchies d'exceptions

```
class Vecteur
{
public:
    class ExptVecteur : public exception
    {
    public:
        const char *what() const throw()
        {
            return "Exception de vecteur";
        }
    };
};

class ExptVecteurBorneSup : public Vecteur::ExptVecteur
{
    // ...
};

class ExptVecteurBorneInf : public Vecteur::ExptVecteurBorneSup
{
    // ...
};

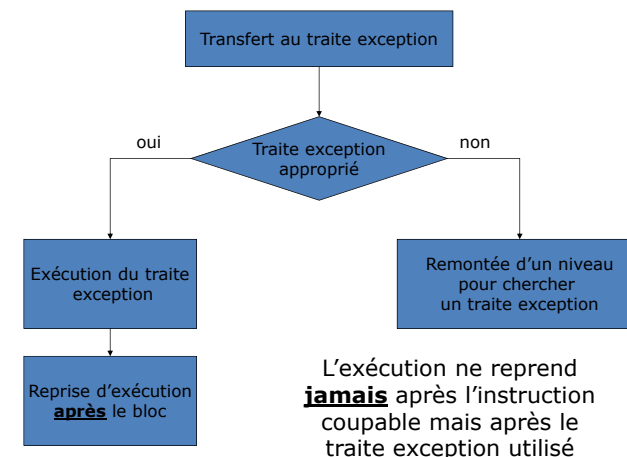
class ExptVecteurAllocation : public Vecteur::ExptVecteur
{
    // ...
};
```

Traitement des hiérarchies d'exception

- Toujours traiter les exceptions les plus spécialisées d'abord
- Utiliser un objet passé par référence pour éviter une copie
- Prévoir un traitement pour **exception**
- Gestionnaire spécialisé `catch(...)` qui ramasse « tout ce qui traîne »

```
try
{
  // Code susceptible de lancer une exception
}
catch (Vecteur::ExptVecteurAllocation &e)
{
  // Traitement exception spécialisée ExptVecteurAllocation
}
catch (Vecteur::ExptVecteurBorneInf &e)
{
  // Traitement exception spécialisée ExptVecteurAllocation
}
catch (Vecteur::ExptVecteur &e)
{
  // Traitement exception plus générale ExptVecteur
}
catch (exception)
{
  // Traitement sommet hiérarchie
}
catch (...)
{ /* Fourre tout */ }
```

Que se passe t'il lorsqu'une exception est levée ?



Relancer une exception

- Pourquoi ?
 - Traite exception incapable d'assurer le retour à la normale
- Conséquences
 - Traitement d'une même exception à plusieurs niveaux
 - Terminaison éventuelle du programme
- Syntaxe
 - Est-il possible de faire plus simple ? ;-)

`throw;`

terminate

- Met fin au programme lorsqu'une exception n'a pas été traitée
- Ne ferme pas les fichiers ni ne libère les ressources
 - ⇒ Prévoir les libérations de ressources dans chaque classe ...
 - ⇒ Ou prévoir une alternative à l'aide de la fonction `set_terminate`
- ⇒ Prototype :

```
void fRemplacement(void);
```

Spécificateurs d'exceptions

- But
 - Définir l'ensemble d'exceptions que l'utilisateur peut s'attendre à voir lever par une méthode ou une fonction
- Difficulté
 - Difficile à définir car le spécificateur doit prévoir tous les sous-appels !
- Si une exception non prévue est levée :
 - Appel de la procédure `unexpected`
 - Par défaut celle-ci appelle `terminate`
 - Possible de redéfinir ce comportement (`set_unexpected` similaire à `set_terminate`)

Syntaxe des spécificateurs d'exceptions

- Syntaxe
 - `typeretour ident(params) throw (liste);`
- Exemple
 - `char & Chaine::operator[](int index) throw (Vecteur::ExptVecteurBornes);`
- Remarques
 - Spécification de `ExptVecteurBornes`
 - Inclut les sous classes
 - Les méthodes propres des classes dérivées de `exception` ne sont pas sensées pouvoir lever une exception