



Généralités

- Auteur: Bjarne Stroustrup
- Crée à la fin des années 80.
 - V2.0 1989, V2.1 1990, V3.0 1991 ...
- Langage normalisée (ISO 14882)
 - Normes publiées C++98, C++03, C++11, C++14, C++17, C++21
- Le langage C++ peut être perçu comme une évolution du langage C
 - C, M.Ritchie et B.W.Kernighan, début des années 70



Programme principal (Main)

- Point d'entrée d'un programme c++

Modules

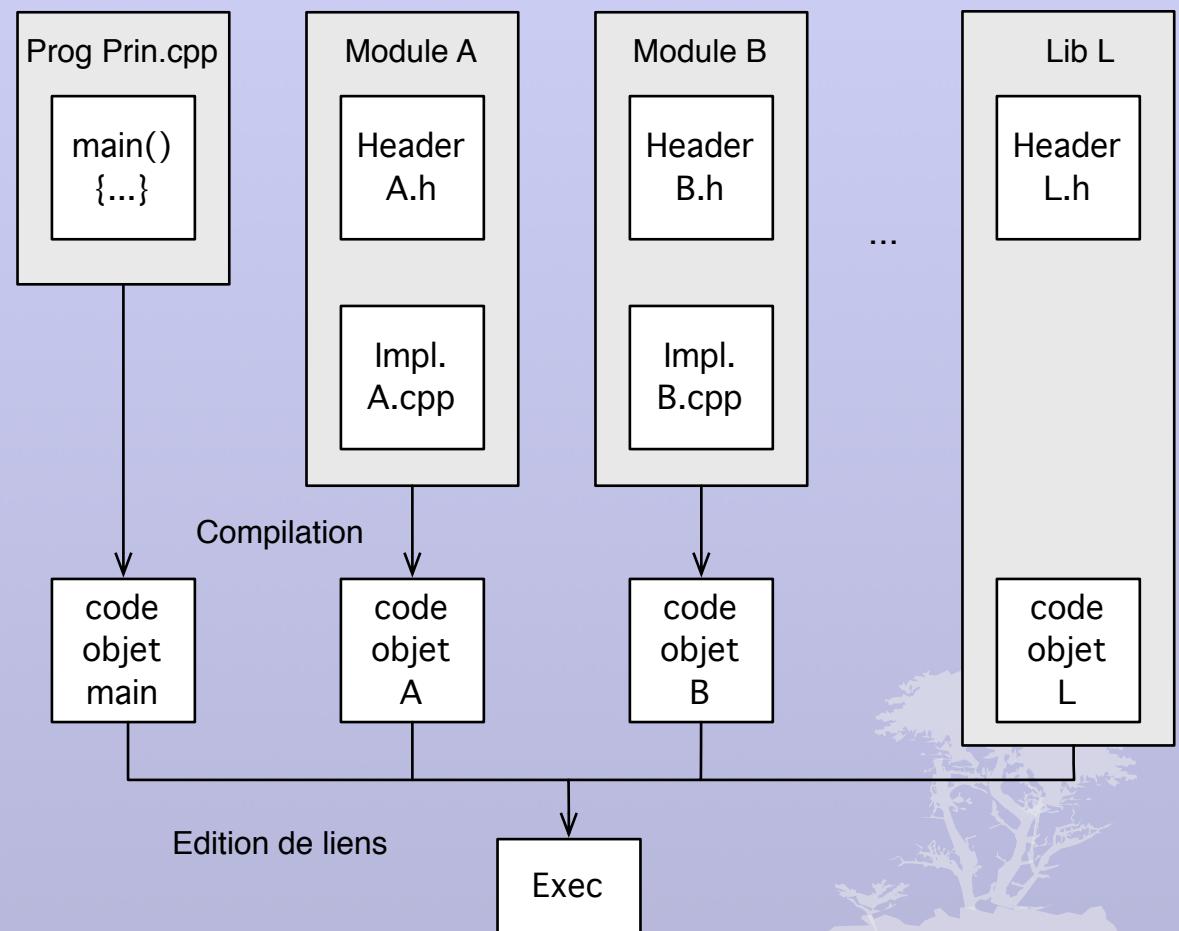
- Entête ou header : Interface contractuelle du module

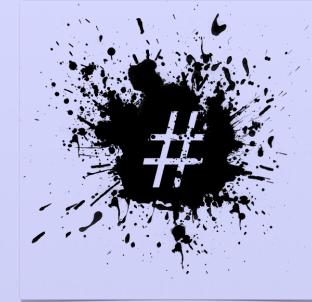
- Implémentation

Librairies

- Entête: Interface contractuelle de la librairie

- Code objet





Définition

- **Le préprocesseur est un programme qui analyse un fichier texte et qui lui fait subir certaines transformations.**
- **Ces transformations peuvent être l'inclusion d'un fichier, la suppression d'une zone de texte ou le remplacement d'une zone de texte.**
- **Le préprocesseur effectue ces opérations en suivant des ordres qu'il lit dans le fichier en cours d'analyse.**
- **Il est appelé automatiquement par le compilateur, avant la compilation, pour traiter les fichiers à compiler.**



Les commandes courantes

● Toutes les commandes commencent : en début de ligne par un signe dièse (#)

● Inclusion de fichier

```
//inclusion d'un fichier "normal"  
#include "fileName"  
//inclusion d'un fichier de librairie standard  
#include <fileName>
```

● Constantes de compilation et remplacement de texte

```
#define aSymbol  
#define First 1
```

● Compilation conditionnelle

#ifdef identificateur ... #endif	#ifndef (if not defined ...) #elif (sinon, si ...) #if (si ...)
--	---

● Exemples

```
#if (__cplusplus==199711L)  
...  
#endif
```

```
#ifndef ASYMBOL  
#define ASYMBOL  
  
//Texte à n'inclure qu'une seule fois  
  
#endif
```

Autres commandes

- **#pragma arguments**
- Permet de donner des ordres spécifiques à une implémentation de compilateur tout en conservant la portabilité du programme.

Les constantes

- **__LINE__** : donne le numéro de la ligne courante ;
- **__FILE__** : donne le nom du fichier courant ;
- **__DATE__** : renvoie la date du traitement du fichier par le préprocesseur ;
- **__TIME__** : renvoie l'heure du traitement du fichier par le préprocesseur ;
- **__cplusplus** : renvoie l'identificateur du compilateur.



Exemple Macro d'assertion

```
#ifndef ASSERT_ON
    #define ASSERT(x)
#else
    #define ASSERT(x) \
        if (!(x)) \
        { \
            std::cout << std ::endl << "ASSERT ERROR " << #x << " failed" << std::endl; \
            std::cout << "on line " << __LINE__ << std::endl; \
            std::cout << "in file " << __FILE__ << std::endl; \
        }
#endif
```

Mots clefs typiquement C++

bool	catch	class	const_cast
delete	dynamic_cast	explicit	export
false	friend	inline	mutable
namespace	new	operator	private
protected	public	reinterpret_cast	static_cast
template	this	throw	true
try	typeid	typename	using
virtual	wchar_t		

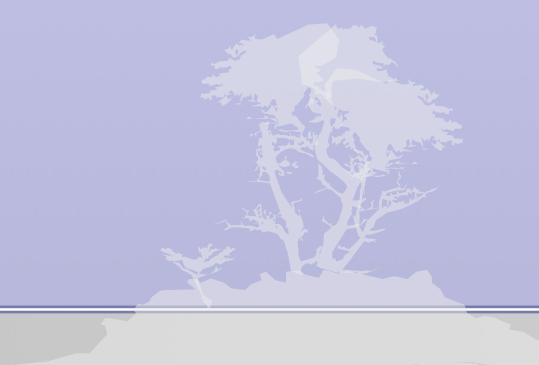
Commentaires

Sur plusieurs lignes

```
/* this  
   is a comment */
```

Sur une seule ligne

```
// this is a comment
```





Types élémentaires C++

Type	Signification	Taille (octets)	Plage de valeurs
char	caractère	1	-128 à 127
unsigned char	caractère non signé	1	0 à 255
short int	entier court	2	-32768 à 32767
int	entier	2 (proc 16 bits) 4 (proc 32 bits)	-32768 à 32767 -2147483648 à 2147483647
unsigned int	entier non signé	2 (proc 16 bits) 4 (proc 32 bits)	0 à 65535 0 à 4 294 967 295
long int	entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	entier long non signé	4	0 à 4 294 967 295
float	flottant (réel)	4	-3.4*10-38 à 3.4*1038
double	flottant double précision	10	-3.4*10-4932 à 3.4*104932
bool	booléen	Même taille que int ou 1 selon compilateur	true ou false

Création d'un nouveau type

Mot clef `typedef`

```
typedef type_original nouveau_type;  
typedef int Integer; //creation du type Integer
```

Utilisation de l'instruction `using (c++11)`

```
using nouveau_type = type_original;  
using Integer = int; //creation du type Integer
```

Conversion de type

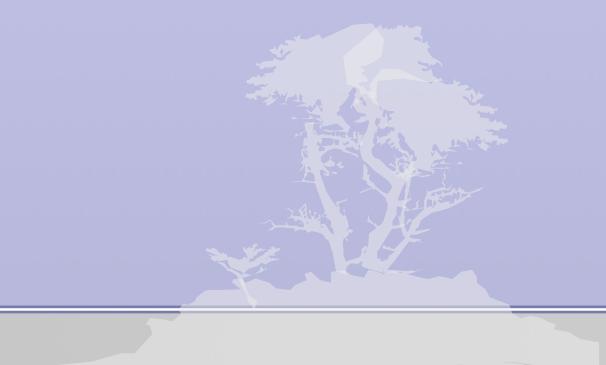
Implicit

```
int i = 2.5;
```

Explicit

Utilisation de l'opérateur de “transtypage” ()

```
int i = (int) 2.5;
```



Déclaration des variables

Portée

- Le bloc d'instruction entre { ... } ou globale

Points de déclarations

- Pas de lieux prédéfinis

```
int a;  
int c,d;  
:  
char c;
```

- La déclaration peut se faire dans une expression :

```
for(int i=0; i < MAX; ++i) {... }
```

- Initialisation de variables avec des valeurs connues ou non à la compilation

```
(C++11) int a = 1;  
int b = a;  
int c{};// int c = 0  
int d{5}// int d=5 ->Attention n'accepte pas de conversion dégradante
```

Déclaration de constantes

```
#define CTE 1  
const int MAX = 10;  
const float CTEF = 1.0;  
int tabInt[MAX] //valid in c++
```



L'inférence de type (C++11)

Utilisation des mots clefs *auto* et *decltype*

- La variable *varInt* sera un entier
- L'utilisation de *decltype* permet de déclarer une variable d'un type identique

Attention

- une variable déclarée avec *auto* doit être initialisée
- A n'utiliser qu'en cas de nécessité
- Son utilisation intempestive nuit à la lisibilité et la maintenabilité du code
- Son utilisation est souvent justifiée lors de l'utilisation de « template »

```
auto varInt = 12;
decltype(varInt) anOtherVarInt = 25;
std::cout << varInt + anOtherVarInt;
```



37

Opérateurs

- les opérateurs de calcul: +, -, *, /, %
- les opérateurs d'incrémentation: ++, -- (pre et post fixés)
- les opérateurs de comparaison: ==, !=, >, <, >=, <=
- les opérateurs logiques booléens: ||, &&, !
- les opérateurs bit-à-bit: &, |, ^, >>, <<
- les opérateurs d'assignation: +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=
- opérateur ternaire: ?
- opérateur pour les tableaux: []
- opérateur de cast: ()



Nouveau for (C++11)

Une boucle fondée sur un intervalle

- L'utilisation d'une *référence* pour *v* permet le cas échéant de modifier la valeur de *v*

```
int tabInt[] = {2,4,6,8,10};  
for(int &v:tabInt)  
    std::cout << v << ' ';
```



2 4 6 8 10

Déclaration

```
type_de_donnee Nom_De_La_Fonction(type1 argument1, type2 argument2, ...)  
{  
    liste d'instructions  
}
```

Appel

//fonction à deux arguments

```
Nom_De_La_Fonction(argument1, argument2);
```

//fonction sans argument

```
Nom_De_La_Fonction();
```

//mémorisation du résultat d'une fonction retournant un int

```
int a;  
a= Nom_De_La_Fonction();
```

exemple.h

```
void f();  
float pi()  
int max(int, int);
```

exemple.cpp

```
#include "exemple.h"  
  
void f()  
{  
}  
  
float pi()  
{  
    retrun pi;  
}  
  
int max(int a, int b)  
{  
    return (a>=b) ?a:b;  
}
```



Ecriture sur la sortie standard

cout : flot de sortie prédéfini

<< opérateur

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

int main()
{
    cout << "hello ";

    //-----
    int a = 1;
    cout << "a ";
    cout << a;
    cout << endl << "a " << a;

    //-----
    int n = 25;
    char c = 'a';
    double x = 3.15;
    char *mess = "hello";

    cout << "value of n : " << n << endl;
    cout << "value of c : " << c << endl;
    cout << "mess : " << mess << endl;
    cout << "adresse de n : " << &n;
    return 0;
}
```



hello

a 1
a 1value of n : 25
value of c : a
mess : hello
adresse de n : 0012FF6C

Lecture sur l'entrée standard

cin : flot d'entrée prédefini

>> opérateur

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

int main()
{
    int a;
    cin >> a;
    cout << endl << "a :" << a;

//-----
    char string[80] ;

    for (unsigned int n=0; n<79; n++)
    {
        cin >> string[n];

        if (string[n]=='.')
        {
            string[n] = '\0';
            break;
        }
    }
    cout << "inputs : " << string;
}
```

14
a :14

voici une
suite de caractères
lus.
inputs : voiciunesuitedecharactèreslus.

Arguments par défaut

```
void printPrompt(char* p = "login > ")  
{  
    cout << endl << p;  
}
```



login >
->

```
int main()  
{  
    printPrompt();  
    printPrompt(" -> ");  
    return 0;  
}
```

Surcharge de fonctions

C++ permet d'avoir un même nom pour plusieurs fonctions.

La différence se fait au niveau des arguments.

```
void whoAmI(int i)  
{  
    cout << "I am the integer :" << i << endl;  
}  
  
void whoAmI(char c)  
{  
    cout << "I am the char :" << c << endl;  
}
```



I am the integer :12
I am the char :a

```
int main()  
{  
    whoAmI(12);  
    whoAmI('a');  
    return 0;  
}
```

Tableaux

Déclaration

```
type Nom_du_tableau [Nombre d'éléments]
```

```
//tableau de 5 entiers  
int tabInt1[5];
```

int	int	int	int	int
-----	-----	-----	-----	-----

```
//tableau de 2 entiers initialisés à la déclaration  
int tabInt2[2] = {12, 5};
```

12	5
----	---

```
//tableau à 2 dimensions initialisés  
int matrix[2][2] = {{12, 5}, {3, 8}};
```

12	5
3	8

```
//itération sur un tableau  
for(int i=0; i<2; i++)  
    cout << tabInt2[i];
```

Un tableau est également un pointeur

- L'arithmétique des pointeurs s'applique aux tableaux



Pointeurs

Définitions

- Un pointeur est une variable dont la valeur est une adresse.
- Les pointeurs sont typés
- Une arithmétique sur les pointeurs existe.

Exemples

```
int *a;                                //a est un pointeur sur un entier
int b= 12;
a = &b;                                //a prend la valeur de l'adresse de b;

cout << a;                            //affichage de a -> adresse de b      0xbfffffa7c
cout << *a;                            //affichage du "contenu" de a (valeur de b) 12

int tab[5]={1,3,5,7,11};
int *ptr = tab;                        //le pointeur ptr prend la valeur de tab

for(int i=0; i<5; ++i, ptr++) //première itération sur le tableau
    cout << ' ' << *ptr;
    1 3 5 7 11

ptr = tab;
for(int i=0; i<5; ++i)                //seconde itération sur le tableau
    cout << ' ' << *ptr++;
    1 3 5 7 11
```

Initialisation d'un pointeur (C++11)

Avant

- Utilisation de la macro **NULL** ayant pour valeur 0

```
int *ptr_old = NULL;
```

Maintenant

- Utilisation de la valeur **nullptr** de type **std::nullptr_t**

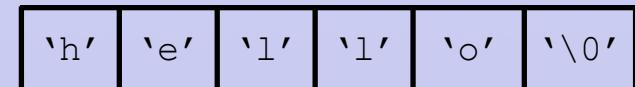
```
int *ptr_new = nullptr;
```

Chaînes de caractères

- En C++ une chaîne de caractères est un tableau ou un “buffer” de caractères terminé par le caractère nul ‘\0’.
- Exemple: la chaîne “hello”

- déclaration sous forme de pointeur

```
char *mess = "hello";
```



- déclaration sous forme de tableau

```
char mess[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

Caractères spéciaux



- La librairie <cstring> fournit un ensemble de fonctions permettant la manipulation des chaînes de caractères.

- copie, comparaison, concaténation, ...

Caractère	Signification
\0	caractère nul
\”	guillemet
\\\	\
\a	bip
\b	retour en arrière
\f	saut de page
\n	retour à la ligne
\t	tabulation
\v	tabulation verticale



Les références

- C'est une alternative aux pointeurs
- C'est une adresse
- Toute manipulation de la référence agit sur l'objet référencé
- Une référence doit être initialisée à sa déclaration
- Elle peut être perçue comme un pointeur constant
- Sa valeur ne peut être modifiée (l'adresse)
- Utilisation comme synonyme

```
int a = 1;
int &aref = a;

cout << aref << endl;
aref = 2;
cout << a << endl;
```



1
2



Passage d'arguments à une fonction

- Pour pouvoir tenir compte du changement de valeurs d'un argument à la sortie d'une fonction, le passage de paramètres doit se faire par adresse.

Par pointeur

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

int a=1,b=2;
swap(&a,&b);
cout << "a " << a;
cout << " b " << b;
```

Par référence

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

int a=1,b=2;
swap(a,b);
cout << "a " << a;
cout << " b " << b;
```



a 2 b 1



Pointeur de fonction

- Un pointeur de fonction contient l'adresse du début du code binaire constituant la fonction

Déclaration

```
type (*identificateur) (type arg1, type arg2, ...);  
  
//Pointeur sur une fonction  
//retournant un int et ayant deux int en paramètre  
int (*ptrFct) (int, int);
```

Exemple

- Utilisation de l'opérateur “**sizeof**” pour déterminer la taille d'une entité

```
PtrFunc tabFuncs[]={subb, sdd, max};  
  
for(int i=0; i<=sizeof(tabFuncs)/sizeof(PtrFunc); ++i)  
    cout << endl << do(tabFuncs[i],2,3);
```

-1
5
3



```
#ifndef PTRFUNC_H  
#define PTRFUNC_H  
  
typedef int (*PtrFunc) (int, int);  
  
int max(int, int);  
int add(int, int);  
int subb(int, int);  
  
int do(PtrFunc, int ,int);  
  
#endif
```



```
#include "ptrfunc.h"  
  
int max(int left, int right)  
{  
    return (left>=right)? left:right;  
}  
  
int add(int left, int right)  
{  
    return left+right;  
}  
  
int subb(int left, int right)  
{  
    return left-right;  
}  
  
int do(PtrFunc pf, int left, int right)  
{  
    return pf(left, right);  
}
```

Fonctions anonymes et fermetures (C++11)

- Fonctions qui n'ont pas de nom (lambda expressions)

- Une fermeture est une fonction qui capture des variables à l'extérieur de celle-ci

- Il faut préciser quelles sont les variables capturées.

```
typedef std::function<int (int ,int)> PtrFunc; ou using PtrFunc = std::function<int (int ,int)>;
```

```
PtrFunc max = [](int left ,int right)->int { return (left>=right) ? left : right;};
PtrFunc add = [](int left ,int right)->int { return left + right;};
PtrFunc subs = [](int left ,int right)->int { return left - right;};

std::cout << std::endl << max(12,5);
```



12

```
auto do = [](PtrFunc f, int left, int right)->int {return f(left,right);};
PtrFunc tabFuncs []={max, add, subb};
```



3

5

-1

```
int x=100, y = 200;
auto do_fermeture = [x, y](PtrFunc f)->int {return f(x,y);};
```



200

300

-100



Fonctions anonymes et fermetures (C++11)

• Fermeture -> syntaxe de la capture des variables

- [] : pas de variable externe
- [x, &y] : x capturée par valeur, y capturée par référence
- [&] : toutes les variables externes sont capturées par référence
- [=] : toutes les variables externes sont capturées par valeur
- [&, x] : x capturée par valeur, toutes les autres variables, par référence
- [=, &x] : x capturée par référence, toutes les autres variables, par valeur



Structure

- Permet de créer un type composé dont les champs n'ont pas le même type

Déclaration

```
struct NomStructure
{
    typeChamp1 champs1;
    typeChamp2 champs1;
    :
    typeChampN champs1;
};
```

Exemple

```
struct Point
{
    int x;
    int y;
};

struct ColoredPoint
{
    Point point;
    int color;
};
```

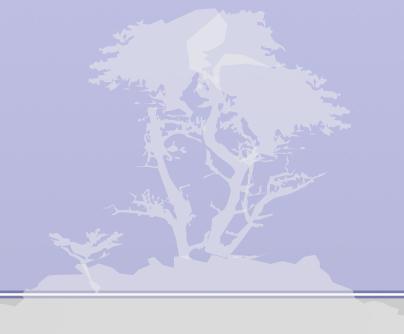
Déclaration d'une variable structurée et utilisation de l'opérateur “.”

```
Point p1;
Point *p2 = &p1;
p1.x = 10;
p1.y = 20;
```

```
ColoredPoint cp;
cp.point.x = 10;
cp.point.y = 20;
cp.color = 0;
```

Opérateur “->” pour les pointeurs

```
Point *p2 = &p1; // p2 pointe sur p1
cout << p2->x // <=> cout << (*p2).x, imprime x de p1
```



Types énumérés

- Les types énumérés sont des types intégraux (ils sont basés sur les entiers)
- Chaque valeur de l'énumération dispose d'un nom unique.
- Ils permettent de définir les constantes entières dans un programme et de les nommer.

Déclaration

```
enum NomType {v1,v2, ..., vn};  
  
//avec initialisions  
  
//initialisions implicites  
enum NomType {v1 = 5,v2 , ..., vn};  
  
//Initialisions individuelles  
enum NomType {v1 = 5, v2=7 , ...}
```

Exemple



```
enum Position {top, bottom, left, right};  
  
Position pos = left;  
  
cout << "pos: ";  
switch (pos)  
{  
    case top:  
        cout << "top";  
        break;  
    case bottom:  
        cout << "bottom";  
        break;  
    case left:  
        cout << "left";  
        break;  
    case right:  
        cout<< "right";  
        break;  
}
```



pos: left

Types énumérés fortement typés (c++11)

Motivations

- Plus de conversions implicites vers un entier
- Il n'est pas possible de faire de l'arithmétique avec! :-)
- Il est possible de choisir la taille de l'entier sous-jacent

Différences

- Utilisation du mot clef **class** après **enum**
- Utilisation de l'opérateur de portée **::**

```
enum class Direction: short { up, down, left, right };

Direction dir = Direction::up;

switch (dir) {
    case Direction::up : std::cout << "up"; break;
    case Direction::down : std::cout << "down"; break;
    case Direction::left : std::cout << "left"; break;
    case Direction::right : std::cout << "right"; break;
}
```



Opérateurs de gestion dynamique de la mémoire

- Deux opérateurs redéfinissables : new et delete

- Allocation mémoire (new)

```
//utilisation  
new Type;  
//allocation d'un tableau de types  
new Type[n];
```

- Libération de la mémoire (delete)

```
//libération d'un objet unique  
delete adresse;  
//libération d'un tableau d'objets  
delete[] adresse;  
  
const int N = 10;  
int *ptr,*tab;  
ptr = new int;  
tab = new int[N];  
delete ptr;  
delete[] tab;
```

- 2 opérateurs de placement prédéfini

- Ils renvoient l'adresse passée en paramètre, on ne doit pas utiliser delete

```
new(adresse) type;  
new(adresse) type[n];
```



utilisation de extern C

Exemple

- lib.h est le header d'une libraririe écrite en C
- f() est une fonction C

```
extern "C"  
{  
    #include "alib.h"  
    void f();  
}
```

Déclaration d'une classe

- Un fichier header: interface contractuelle de la classe
- Un fichier d'implémentation
- Support au principe d'encapsulation
- L'accès aux entités d'une classe est défini par les mots clefs:
 - **public**: accès à tous
 - **protected**: accès aux classes d'une même famille
 - **private**: accès restreint à la classe
- L'héritage module la visibilité des composantes des classes "ancêtres"

		type d'héritage		
Accès aux données membres et méthodes		public	protected	private
public	public	protected	private	
protected	protected	protected	private	
private	inaccessible	inaccessible	inaccessible	



Anatomie

Entête/Header

```
#ifndef UN_SYMBOLE_UNIQUE
#define UN_SYMBOLE_UNIQUE

class NomClasse: public|protected|private NomClasseMere, ...
{
    public|protected|private:
        //déclaration de données membres et/ou méthodes

    public|protected|private:
        //déclaration de données membres et/ou méthodes

    ...
};

#endif
```

nomfichier.h

Implémentation

```
#include "nomfichier.h"

//implémentation des méthodes de la classe
//déclarations spécifiques
```

nomfichier.cpp

Exemple

Classe Point

```
#ifndef POINT_H           point.h
#define POINT_H

class Point
{
public:
    Point();
    Point(int, int);

    int getX() const;
    int getY() const;
    void setX(int);
    void setY(int);

private:
    int x,y;
};

#endif
```

```
#include "point.h"          point.cpp

Point::Point():
x(0), y(0)
{
}

Point::Point(int _x, int _y):
x(_x), y(_y)
{
}

int Point::getX() const
{
    return x;
}

int Point::getY() const
{
    return y;
}

void Point::setX(int _x)
{
    x=_x;
}

void Point::setY(int _y)
{
    y=_y;
}
```

```
int main ()
{
    Point p1, p2(10,20);
    Point p3(p2);
    std::cout << endl << "p1:" << p1.getX() << ',' << p1.getY();
    std::cout << endl << "p2:" << p2.getX() << ',' << p2.getY();
    std::cout << endl << "p3:" << p3.getX() << ',' << p3.getY();
    return 0;
}
```

p1:0,0
p2:10,20
p3:10,20



Constructeurs

- Trois types : **Défaut, Valué, Par recopie**
- Ils portent le même nom que la classe.
- Quelques règles d'existence
 - Le constructeur par défaut est généré tant qu'un constructeur n'aït été explicitement défini.
 - Le constructeur par recopie est généré tant qu'il n'a pas été pas redéfini.
- Quelques règles de comportement
 - Les constructeurs sont invoqués par le compilateur lors de l'instanciation des objets.
 - Tout constructeur vide (sauf par recopie) fait appel au constructeur par défaut de la classe mère et initialise les données membre en invoquant leur constructeur par défaut.
 - Le constructeur par recopie fait appel au constructeur par recopie de la classe mère et, par défaut, initialise les données membres “membre à membre”.



Classe Point et redéfinition des trois constructeurs

```
int main ()
{
    Point p1, p2(10,20);
    Point p3(p2);
    std::cout << endl << "p1:" << p1.getX() << ',' << p1.getY();
    std::cout << endl << "p2:" << p2.getX() << ',' << p2.getY();
    std::cout << endl << "p3:" << p3.getX() << ',' << p3.getY();
    return 0;
}
```



-> défaut-> valué-> copie
 p1:0,0
 p2:10,20
 p3:10,20

```
#ifndef POINT_H
#define POINT_H

class Point
{
public:
    Point();
    Point(int, int);
    Point(const Point&);
    ~Point();
    ...
};

#endif

#include "point.h"
#include <iostream>

Point::Point():
x(0), y(0)
{
    std::cout << "-> défaut";
}

Point::Point(int _x, int _y):
x(_x), y(_y)
{
    std::cout << "-> valué";
}

Point::Point(const Point& p):
x(p.x), y(p.y)
{
    std::cout << "-> copie";
}

Point::~Point()
{
    std::cout << "-> destructeur";
}
...
```

Destructeur

- C'est une fonction membre appelée à la destruction d'un objet.
- Il porte le même nom que celui de la classe et est précédé de ~ .
- Il ne possède ni arguments ni valeur de retour.
- Il permet de libérer des ressources acquises

Règle d'existence

- Il est généré tant qu'il n'a pas été redéfini.

Quelques règles de comportement

- Les destructeurs sont invoqués par le compilateur lors de la destruction des objets
- Les destructeurs invoquent par défaut les destructeurs des données membre et invoquent le destructeur de la classe "Mère"



Classe Point et redéfinition du destructeur

```
int main ()
{
    Point p1, p2(10,20);
    Point p3(p2);
    std::cout << endl << "p1:" << p1.getX() << ',' << p1.getY();
    std::cout << endl << "p2:" << p2.getX() << ',' << p2.getY();
    std::cout << endl << "p3:" << p3.getX() << ',' << p3.getY();
    return 0;
}
```

-> défaut-> valué-> copie
 p1:0,0
 p2:10,20
 p3:10,20-> destructeur-> destructeur-> destructeur

```
#ifndef POINT_H
#define POINT_H

class Point
{
public:
    Point();
    Point(int, int);
    Point(const Point&);
    ~Point();
    ...
};

#endif
```

Exemple de créations dynamiques d'objets

```
int main ()
{
    Point *p4 = new Point[3];
    Point *p5 = new Point(1,2);
    delete[] p4;
    delete p5;
    return 0;
}
```

```
#include "point.h"
#include <iostream>

...
Point::~Point()
{
    std::cout << "-> destructeur";
}
...
```

-> défaut-> défaut-> défaut-> valué-> destructeur-> destructeur-> destructeur-> destructeur

Membres constants

- Une donnée membre constante se définit par le mot clef : **const**
- Elle est initialisée à la construction et ne peut plus être changée.

```
class Truc           Truc::Truc():  
{                      N(10)  
public:                {  
    Truc();  
  
private:  
    const int N;  
};
```

Fonctions constantes

- Une fonction constante est une fonction qui ne doit pas modifier l'état d'un objet.
- Exemple les fonctions de consultation (Get) de la classe Point



Membres mutables

- La notion de constance dépend de la perception que l'on en a.
- La constance peut être physique ou logique.
- Un objet peut paraître cte pour ses clients et peut en fait changer son état interne pour assurer son comportement.

```
class Point
{
public:
    ...
    int getX() const;
    int getY() const;

private:
    int x,y;
    mutable int access;
};

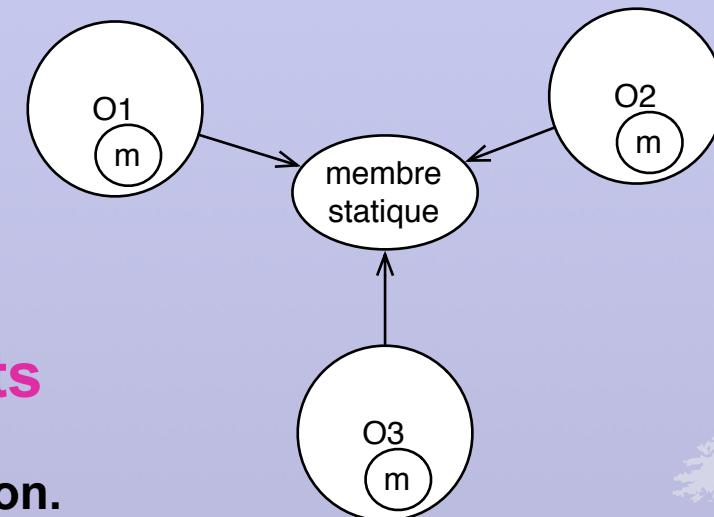
int Point::getX() const
{
    access++;
    return x;
}

int Point::getY() const
{
    access++;
    return y;
}
```

- Sans mutable, le compilateur n'aurait pas permis l'écriture des fonctions get().

Méthodes et données membres statiques

- Membres statiques
- Ils servent à implanter les variables, constantes et fonctions de classes.
- Habituellement chaque objet possède ses fonctions et membres.
- Si un membre est statique alors tous les objets le partage.
- Ce membre peut être utilisé sans qu'il n'existe d'instance de la classe.



Exemple d'un compteur d'objets

- On incrémente le compteur à la création.
- On décrémente le compteur à la destruction.

Comptage d'objets: classe Class

En faisant hériter la classe Point de la classe Class

```
#ifndef POINT_H
#define POINT_H

#include "point.h"

class Point: public Class
{
    ...
};

#endif
```

```
int main ()
{
    Point *p4 = new Point[3];
    Point *p5 = new Point(1,2);
    Point p6(*p5);
    cout << endl << Class::getRef() << endl;
    delete[] p4;
    delete p5;
    cout << endl << Class::getRef() << endl;
    return 0;
}
```

-> défaut-> défaut-> défaut-> valué-> copie
 5
 ->destructeur->destructeur->destructeur->destructeur
 1
 ->destructeur

```
#ifndef CLASS_H
#define CLASS_H

class Class
{
public:
    Class();
    Class(const Class&);
    virtual ~Class();

    static unsigned int getRef();
private:
    static unsigned int ref;
};

#endif
```

```
#include "class.h"

unsigned int Class::ref = 0;

Class::Class()
{
    ref++;
}
Class::Class(const Class& c)
{
    ref++;
}
Class::~Class()
{
    ref--;
}

unsigned int Class::getRef()
{
    return ref;
}
```

Affection d'objets

- L'opérateur d'affection par défaut existe.
- L'affection se fait membre à membre.
- Cet opérateur est rédéfinissable.

Redéfinition de l'opérateur =

Affection de point sans redéfinition

```
Point p1(10,20), p2;
cout << endl << "p1:" << p1.getX() << ',' << p1.getY();
cout << endl << "p2:" << p2.getX() << ',' << p2.getY();
p2=p1;
cout << endl << "p2:" << p2.getX() << ',' << p2.getY();
```

p1:10,20
p2:0,0
p2:10,20



Après redéfinition

```
class Point: public Class          point.h
{
public:
...
Point& operator = (const Point&);
...
};
```

```
Point& Point::operator = (const Point& p)
{
    if(&p != this)
    {
        x=p.x;
        y=p.y;
        std::cout << std::endl << "-> =";
    }
    return *this;
}
```

point.cpp

p1:10,20
p2:0,0
-> =
p2:10,20



Classe typique

Attention

- Les redéfinitions du constructeur par recopie et de l'opérateur d'affectation vont de pair.

```
#ifndef POINT_H
#define POINT_H

#include "class.h"

class Point: public Class
{
public:
    Point();
    Point(int, int);
    Point(const Point&);
    ~Point();

    Point& operator = (const Point&);

    int getX() const;
    int getY() const;
    void setX(int);
    void setY(int);

private:
    int x,y;
};

#endif
```

Conversion de type

Par un constructeur implicite

```
//Définition du constructeur  
Point(int x, int y = 0);  
  
-----  
  
//utilisation  
Point p = 2;  
cout << "p:" << p.getX() << ',' << p.getY();
```



p:2,0

Par un constructeur explicite

```
//Définition du constructeur  
explicit Point(int x, int y = 0);  
  
-----  
  
//utilisation  
Point p = 2; //illegal  
Point p = (Point) 2; // ok  
cout << "p:" << p.getX() << ',' << p.getY();
```

Par un opérateur de conversion

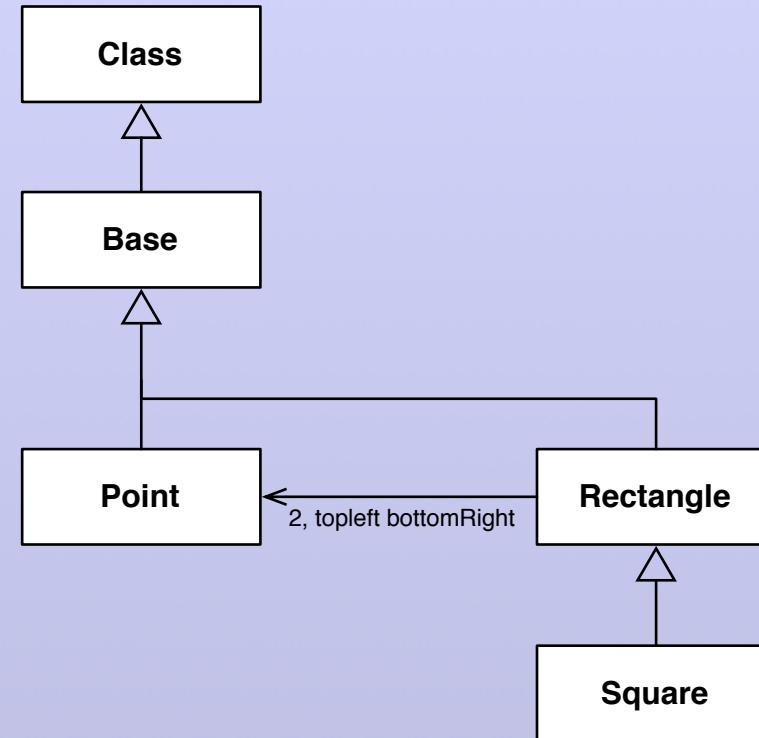
```
//Déclaration  
operator int ();  
  
-----  
  
//Implémentation  
Point::operator int()  
{  
    return sqrt(x*x+y*y);  
}
```

2



```
//utilisation  
Point p(2,2);  
int i = p;  
cout << i;
```

Diagramme de classes



Classe Base (Abstraite)

Mécanisme d'impression

- `o.printOn(cout); // o étant une sorte de Base`
- `cout << o;`

• Toutes les classes héritant de **Base** publiquement auront la capacité d'impression sur un flux standard de sortie.

• Pour avoir une impression spécialisée il faut redéfinir la méthode virtuelle pure:

`virtual ostream& printOn(ostream&) const = 0`

```
#ifndef Base_h
#define Base_h

#include <iostream>
#include "class.h"

class Base : public Class
{
public :
    virtual std::ostream& printOn(std::ostream&) const = 0;

    friend std::ostream& operator << (std::ostream&, const Base&);
};

#endif /* Base_hp */
```

Abstraite

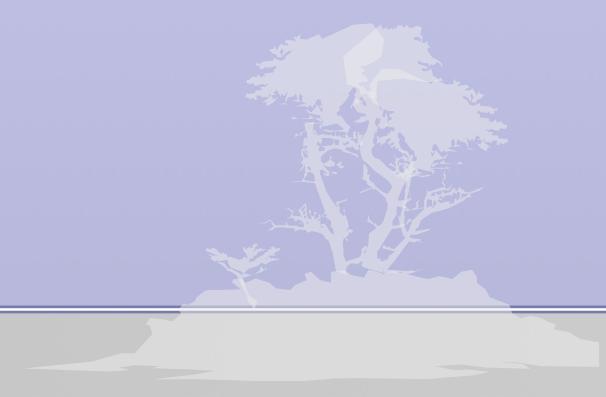
Virtuelle pure

```
#include "Base.hpp"

std::ostream& operator << (std::ostream& os, const Base& o)
{
    return o.printOn(os);
}
```

La classe Base est abstraite

• Elle possède une méthode virtuelle pure



Classe Point

```
#ifndef POINT_H
#define POINT_H

#include "base.h"

class Point: public Base
{
public:
    Point();
    Point(int, int);
    Point(const Point&);
    virtual ~Point();

    virtual Point& operator = (const Point&);

    virtual int getX() const;
    virtual int getY() const;
    virtual void getX(int);
    virtual void getY(int);
    virtual void translate(int, int);

    virtual std::ostream& printOn(std::ostream&) const;

private:
    int x,y;
};

#endif
```

Classe Point

```
#include "point.h"
#include <iostream>

Point::Point():
x(0), y(0)
{}

Point::Point(int _x, int _y):
x(_x), y(_y)
{}

Point::Point(const Point& p):
x(p.x), y(p.y)
{}

Point::~Point()
{}

Point& Point::operator = (const Point& p){
    if(&p != this)
    {
        x=p.x;
        y=p.y;
    }
    return *this;
}
```

```
Point::Point():
Point(0,0)
{}

Point::Point(int _x, int _y):
x(_x), y(_y)
{}

Point::Point(const Point& p):
Point(p.x, p.y)
{}
```

C++11

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

int Point::getY() const{
    return y;
}

void Point::setX(int _x){
    x=_x;
}

void Point::setY(int _y){
    y=_y;
}

void Point::translate(int dx, int dy){
    x+=dx;
    y+=dy;
}

std::ostream& Point::printOn(std::ostream& os) const{
    os << "Point(";
    os << x << ',' << y << ')';
    return os;
}
```

Point(10,20)

Classe Rectangle

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

#include "base.h"
#include "point.h"

class Rectangle: public Base
{
public:
    Rectangle();
    Rectangle(const Point&, const Point&);
    Rectangle(int, int, int, int);

    virtual Point getTopLeft() const;
    virtual Point getBottomRight() const;
    virtual void setTopLeft(const Point&);
    virtual void setBottomRight(const Point&);
    virtual void translate(int dx, int dy);

protected:
    virtual std::ostream& printOn(std::ostream&) const;

private:
    Point topLeft, bottomRight;
};

#endif
```

topLeft



bottomRight



Classe Rectangle

```
#include "Rectangle.h"

Rectangle::Rectangle()
{}

Rectangle::Rectangle(const Point& tl, const Point& br):
topLeft(tl), bottomRight(br)
{}

Rectangle::Rectangle(int x, int y, int width, int height):
topLeft(Point(x,y)), bottomRight(x+width, y+height)
{}

Point Rectangle::getTopLeft() const
{
    return topLeft;
}

Point Rectangle::getBottomRight() const
{
    return bottomRight;
}
```

```
void Rectangle::setTopLeft(const Point& r)
{
    topLeft = r;
}

void Rectangle::setBottomRight(const Point& r)
{
    bottomRight = r;
}

void Rectangle::translate(int dx, int dy)
{
    topLeft.translate(dx,dy);
    bottomRight.translate(dx,dy);
}

std::ostream& Rectangle::printOn(std::ostream& os) const
{
    os << "Rectangle(";
    os << topLeft << ',' << bottomRight << ')';
    return os;
}
```

Rectangle(Point(10,20), Point(30,25))



Classe Square

- ➊ Héritage privé
- ➋ Un carré est une sorte de rectangle
- ➌ Le contrat d'utilisation du rectangle est masqué et inaccessible.
- ➍ Le contrat d'utilisation du carré est spécifique et doit être complètement redéfini à partir de celui du rectangle.

```
#ifndef SQUARE_H
#define SQUARE_H

#include "rectangle.h"

class Square: private Rectangle
{
public:
    Square();
    Square(const Point&, int);
    Square(int, int, int);

    virtual Point getLocation() const;
    virtual int getSize() const;
    virtual void setLocation(const Point&);
    virtual void setSize(int);
    virtual void Translate(int, int);

    friend std::ostream& operator << (std::ostream&, const Square&);

};

#endif
```

Square

```
#include "square.h"

Square::Square(const Point& loc, int size):
    Rectangle(Point(loc), Point(loc.getX()+size, loc.getY()+size))
{}

Square::Square(int x, int y, int size):
    Rectangle(x, y, size, size)
{}

Point Square::getLocation() const
{
    return getTopLeft();
}

int Square::getSize() const
{
    return getBottomRight().getX()-getTopLeft().getX();
}

void Square:: setLocation(const Point& loc)
{
    int dx = loc.getX() - getTopLeft().getX();
    int dy = loc.getY() - getTopLeft().getY();

    Rectangle::translate(dx,dy);
}

void Square::setSize(int size)
{
    int x = getBottomRight().getX()+size;
    int y = getBottomRight().getY()+size;

    setBottomRight(Point(x,y));
}

void Square::Translate(int dx, int dy)
{
    Rectangle::Translate(dx,dy);
}

std::ostream& operator << (std::ostream& os, const Square& s)
{
    return s.printOn(os);
}
```

Rectangle (Point(10,20), Point(30,40))

● Mécanisme d'interception des erreurs permettant leur gestion.

● Prennent en Charge :

- La remontée du signal d'erreur vers les fonctions appelantes.
- La détection du signal par une fonction réceptive.
- L'exécution du traitement approprié.

● Mise en œuvre de la gestion des erreurs en 2 temps :

- Mise en place du bloc réceptif à l'aide de l'instruction : **try**.
- Définition du traitement à apporter -> définition des gestionnaires d'exceptions à l'aide de l'instruction : **catch**.

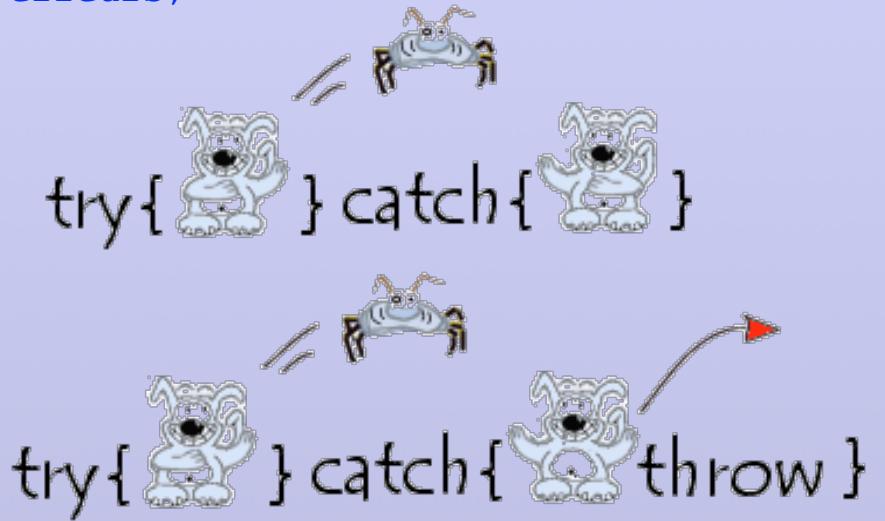
● Déclenchement de l'erreur

- Utilisation de l'instruction : **throw**.



Cas général

```
// Instructions inoffensives ;  
  
try  
{  
    // Instructions susceptibles de provoquer des erreurs;  
    ...  
}  
catch (TypeException1 e1)  
{  
    // Instructions de traitement de l'exception;  
    ...  
}  
catch (TypeException2 e2)  
{  
    // Instructions de traitement de l'exception;  
    ...  
}  
catch(...)  
{  
    //instructions exécutées si aucun bloc catch ne convient  
    ...  
}  
  
// Instructions si aucune nouvelle exception n'est apparue dans un bloc catch
```



© <http://www.commentcamarche.com>



Exemple: classe MonthConverter

```
#ifndef MONTHCONVERTER_H
#define MONTHCONVERTER_H

#include "illegalmonthexception.h"

class Monthconverter
{
public:
    static const char* convert(unsigned short);

private:
    static const char* month[];
};

#endif
```

```
#include "monthconverter.h"

const char*
Monthconverter::month[]={"","","january","february"
,"march","april","may","june","july","aout","september"
,"october","november","december"};

const char* Monthconverter::convert(unsigned short m)
{
    if(m<1 || m>12)
        throw IllegalMonthException(m);

    return month[m];
}
```

```
#ifndef ILEGALMONTHEXCEPTION_H
#define ILEGALMONTHEXCEPTION_H

#include <string>
#include "base.h"

class IllegalMonthException : public Base
{
public:
    IllegalMonthException(unsigned char);

    virtual std::ostream& printOn(std::ostream&) const;

private:
    const unsigned short illegalMonth;
};

#endif
```

```
#include "illegalmonthexception.h"

IllegalMonthException::IllegalMonthException(unsigned char m):
illegalMonth(m)
{ }

std::ostream& IllegalMonthException::printOn(std::ostream& os)
const
{
    return os << "illegal month: " << illegalMonth;
}
```

 Exemple

```
int main (int argc, char * const argv[])
{
    unsigned short month;

    while(true)
    {
        try
        {
            cout << endl << "month to convert: ";
            cin >> month;
            cout << "-> " << Monthconverter::convert(month);
        }
        catch(const IllegalMonthException& e)
        {
            cout << endl << e;
        }
    }
    return 0;
}
```



month to convert: 6
-> june
month to convert: 0
->
illegal month: 0
month to convert: 3
-> march
month to convert:

Les fonctions

```
template<class T>
void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}

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

template<class T>
void test(T& x, T& y)
{
    cout << endl << "x: " << x << " y: " << y;
    swap(x,y);
    cout << endl << "x: " << x << " y: " << y;
    cout << endl << "max(x,y): " << max(x,y);
}
```

```
int x=1, y=10;
test(x, y);
string s1="s1", s2="s2";
test(s1, s2);
```



```
x: 1 y: 10
x: 10 y: 1
max(x,y) : 10
x: s1 y: s2
x: s2 y: s1
max(x,y) : s2
```

Les classes

Tableau dynamique

```
#ifndef ARRAY_H
#define ARRAY_H

#include "base.h"

template <class T>
class Array: public Base
{
public:
    typedef unsigned int size_a;
public:
    Array();
    Array(size_a);
    virtual ~Array();

    virtual T& operator [] (size_a);
    virtual ostream& printOn(ostream &) const;
private:
    T *array;
    size_a size;
};
```

```
template <class T>
Array<T>::Array():
size(1), array(new T[1])
{
}

template <class T>
Array<T>::Array(size_a sz):
size(sz), array(new T[sz])
{
}

template <class T>
Array<T>::~Array()
{
    delete[] array;
}

template <class T>
ostream& Array<T>::printOn(ostream& s) const
{
    for (size_a i=0;i<size;++i)
        s << array[i] << ' ';
    return s;
}

template <class T>
T& Array<T>::operator [] (size_a i)
{
    if (i > (size-1))
        throw "invalid index";
    return array[i];
}

#endif
```

Les classes

Tableau dynamique (suite)

```
try
{
    Array<int> aint(3);
    aint[0] = 1; aint[1] = 2; aint[2] = 3;

    Array<string> astring(2);
    astring[0] = "st1"; astring[1] = "st2";

    cout << aint << endl;
    cout << astring << endl;
    astring[3] = "st3";
    cout << astring << endl;
}
catch(const char* msg)
{
    cout << endl << msg;
}
```

1 2 3
st1 st2

invalid index

Catégories

Support du langage

Classes Exceptions de base

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

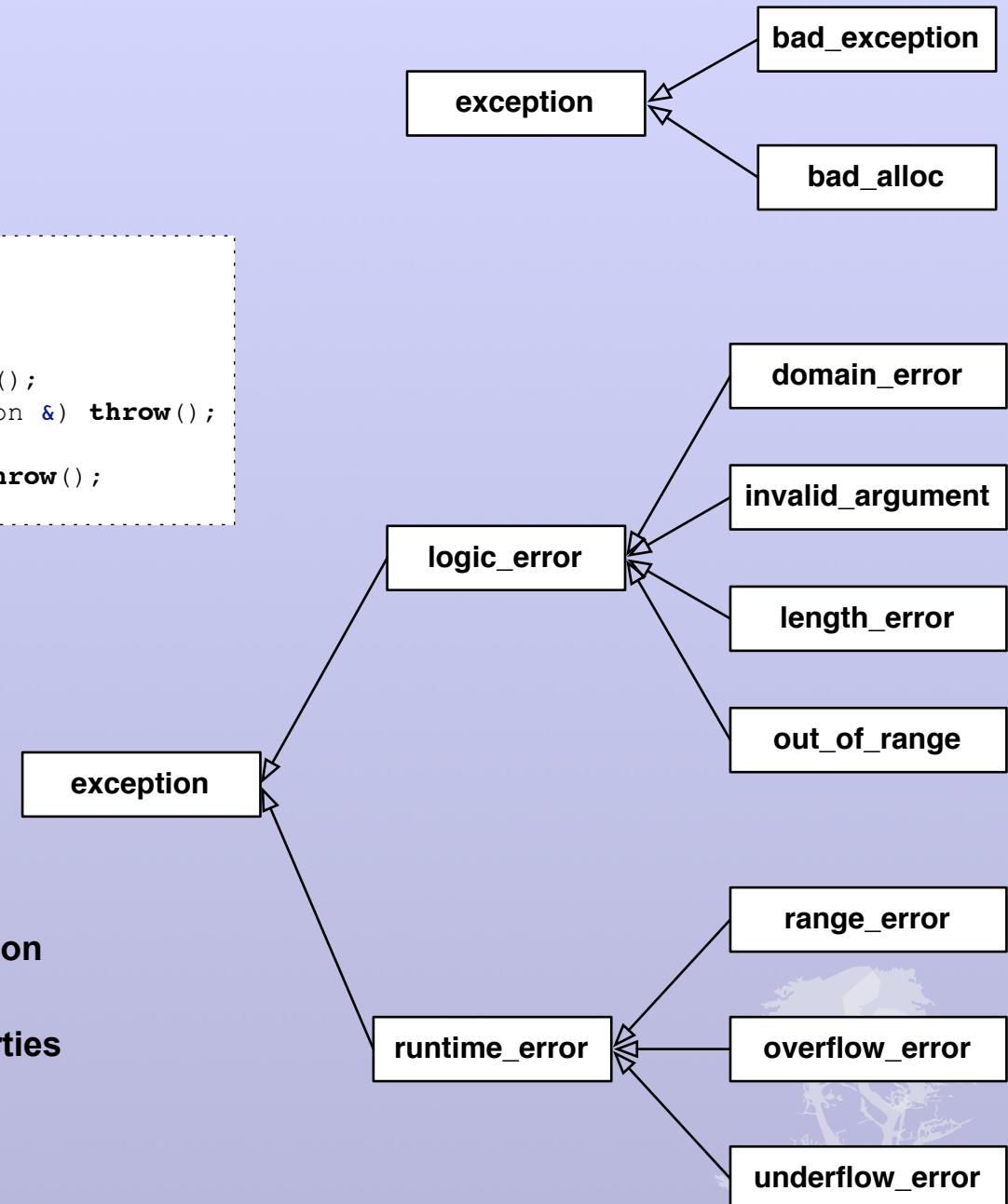
Identification des types dynamiques

Diagnostic <stdexcept>

Classes Exceptions complémentaires:

- logic_error* : erreur de programmation

- runtime_error* : erreur d'entrées/sorties

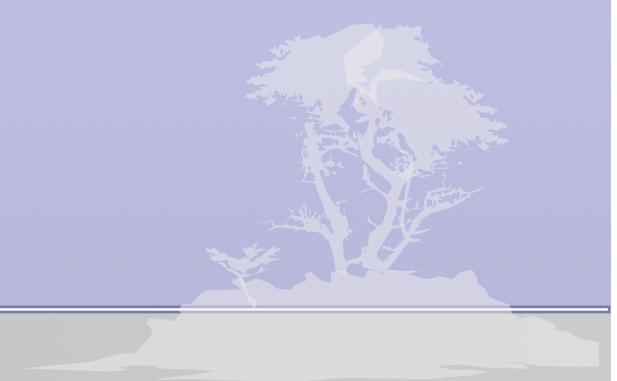




Catégories

Utilitaires généraux

- Générateurs d'opérateurs
- La structure pair
- Les Classes Fonctions
- Chaînes string



Catégories

Conteneurs

- Séquences de base

- **vector, list, deque**

- Types abstraits

- **queue, priority_queue, stack**

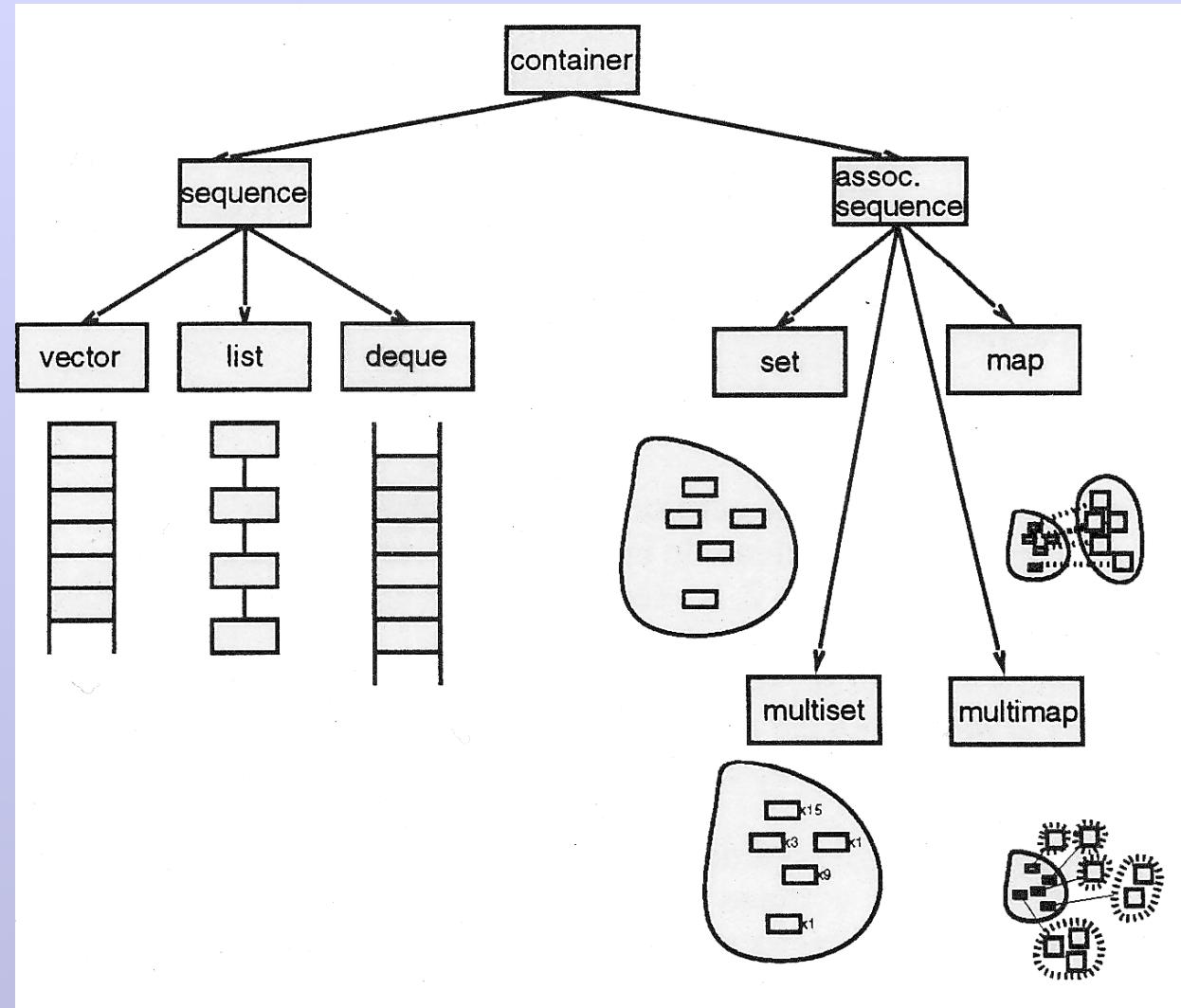
- Containers associatifs

- **set, multiset**

- **map, multimap**

- Itérateurs

- Permettent d'itérer sur les conteneurs de façon uniforme et générique.



Catégories

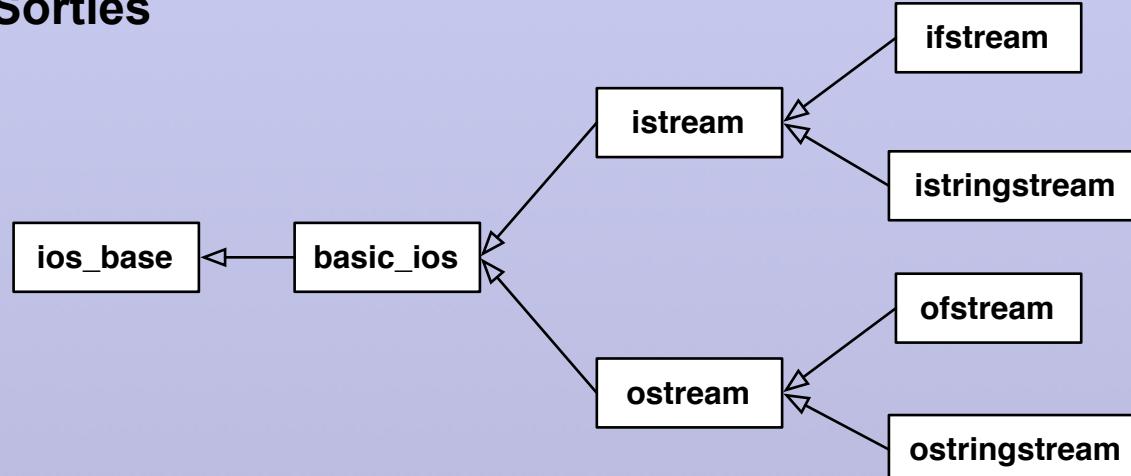
Algorithmes

- Composants réalisant des traitements sur ou à partir des conteneurs.

Numériques

- **complex**
- **valarray**

Entrées/Sorties



Manipulation de formats

• Fichier à inclure : <iomanip>

• dec, hex, oct

- Permettent de modifier la base courante pour l'entrée ou la sortie d'entiers

• showbase, noshowbase

- Spécifient si un indicateur de base doit être afficher.

• fixed, scientific

- Spécifient si les réels sont sortis en notation fixe ou scientifique.

• left, right

- Spécifient le cadrage de sortie

• showpoint, noshowpoint

- Spécifient si un point doit être généré inconditionnellement lors de la sortie des réels.

Manipulations de format

showpos, noshowpos

- Spécifient si le signe plus doit être sorti.

uppercase, nouppercase

- Spécifient si les majuscules remplacent les minuscules.

skipws, noskipws

- Spécifient si les espaces doivent être ignorés lors de certaines entrées

setw(int)

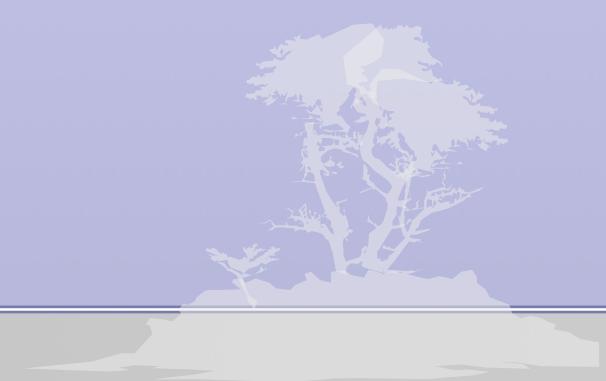
- Permet d'obtenir la largeur de champs spécifié

setfill(int)

- Permet de définir le caractère de remplissage

setprecision(int)

- Permet de définir le nombre de décimal après la virgule.



Manipulations de formats

resetiosflags(ios_base::fmtflags)

- Reset les indicateurs spécifiés.

setiosflag(ios_base::fmtflags)

- Valide les indicateurs spécifiés.

Exemples

```
cout << endl << "15 decimal-> " << 15;
cout << endl << hex << "15 hexa-> " << 15;
cout << showbase;
cout << endl << hex << "15 hexa with base-> " << 15;
cout << noshowbase;
cout << fixed << endl << "fixed notation-> " << 10.234;
cout << scientific << endl << "scientific notation-> " << 10.234;
cout << endl << "width 15 & fill - -> " << fixed;
cout << setfill('-') << setw(15) << 10.234;
cout << setprecision(2) << endl << "precision 2-> " << 10.234;
```

15 decimal-> 15
 15 hexa-> f
 15 hexa with base-> 0xf
 fixed notation-> 10.234000
 scientific notation-> 1.023400e+001
 width 15 & fill - -> -----10.234000
 precision 2-> 10.23



Utilisation de fichier

Fichier à inclure : <fstream>

Recopie simple d'un fichier caractère par caractère.

```
ifstream fin("demo.txt");
ofstream fout("demo-copy.txt");
fin >> noskipws;
char c;
while (fin)
{
    fin >> c ;
    fout << c;
}
```

Utilisation des flux connectés aux chaînes

Fichier à inclure : <sstream>

```
template<class T>
string value2string(const T& v)
{
    ostringstream os;
    os << v;
    return os.str();
}
```

```
string s("result: ");
cout << s +value2string(string2value<int>("2") + 3);
```

```
template <class T>
T string2value(const string& s)
{
    istringstream is(s);
    T v;
    is >> v;
    return v;
}
```



Exemples

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

```
float tab[] = {1,2,3,4,5};

vector<float> v1(tab,tab+5);
vector<float>::iterator it = v1.begin();

for(;it != v1.end();++it)
    cout << *it << " ";

cout << endl;
for_each(v1.begin(),v1.end(),print);

cout << endl;
printFunc<float> p = for_each(v1.begin(),v1.end(),printFunc<float>(cout));
cout << endl << "count: " << p.count;

cout << endl;
ostream_iterator<float> outPut(cout, "-");
copy(v1.begin(),v1.end(),outPut);

cout << endl << "enter v2 values" << endl;
vector<float> v2;
istream_iterator<float> inIter(cin), endIter;
copy(inIter, endIter, back_inserter(v2));
copy(v2.begin(),v2.end(),outPut);
```

```
void print(float f)
{
    cout << f << ',';
}

template<class T>
```

```
auto print = [] (float f) {std::cout << f << ',';};
std::vector<float> v1 = {1,2,3,4,5};

for(float v : v1)
    std::cout << v << " ";

std::cout << std::endl;
for_each(v1.begin(),v1.end(), print);
//for_each(v1.begin(),v1.end(), [] (float f) {std::cout << f << ',';});
```

```
int count;
};
```

C++11



```
1 2 3 4 5
1,2,3,4,5,
1 2 3 4 5
count: 5
1-2-3-4-5-
enter v2 values
5
6
0
5-6-
```

Les tableaux à taille fixe (C++11)

Permet d'utiliser un tableau comme un conteneur STL

```
#include <array>
std::array<float,5> a1 = {1,2,3,4,5};
std::cout << "size : " << a1.size();

std::cout << std::endl;
(a1.empty())? std::cout << "a1 empty " : std::cout << "a1 not empty ";
std::cout << std::endl << "front() : " << a1.front();
std::cout << std::endl << "back() : " << a1.back();
std::cout << std::endl << "at() with limits control: " << a1.at(1);
std::cout << std::endl << "[] : " << a1[1];

std::cout << std::endl;
copy(begin(a1), end(a1), std::ostream_iterator<float>(std::cout, "-"));
```



```
size : 5
a1 not empty
front() : 1
back() : 5
at() with limits control: 2
[] : 2
1-2-3-4-5-
```

Evaluateur de fonctions -> Courbes discrètes

```
#ifndef EVALUATOR_H
#define EVALUATOR_H

#include <vector>
using namespace std;

template <class T>
class Evaluator
{
public:
    class EvalFunc
    {
public:
        virtual T operator () (const T&)=0;
    };

    typedef pair<vector<T>,vector<T> > Shape;

    static Shape buildShape(const T& min, const T& max, const T& step, EvalFunc&);
    static ostream& printShape(ostream&, const Shape& s);
};

...
```

Evaluateur de fonctions -> Courbes discrètes

```

...
template <class T>
typename Evaluator<T>::Shape Evaluator<T>:: buildShape(const T& min, const T& max, const T& step, EvalFunc& f)
{
    vector<T> x, y;
    for (T i= min ;i <= max ;i += step)
    {
        y.push_back(f(i));
        x.push_back(i);
    }
    return Shape(x,y);
}

template<class T>
ostream& Evaluator<T>::printShape(ostream& os, const Shape& s)
{
    os << '[';
    typename vector<T>::const_iterator it= s.first.begin();
    for(; it!=s.first.end(); ++it)
        os << *it << ' ';
    os << ']';
    os << endl;
    os << '[';
    it = s.second.begin();
    for(; it!=s.second.end(); ++it)
        os << *it << ' ';
    os << ']';
    return os;
}

#endif

```

C++11

```

os << '[';
for(auto val : s.first)
    os << val << ' ';
os << ']';
os << endl;
os << '[';
for(auto val : s.second)
    os << val << ' ';
os << ']';
return os;

```

Evaluateur de fonctions -> Courbes discrètes

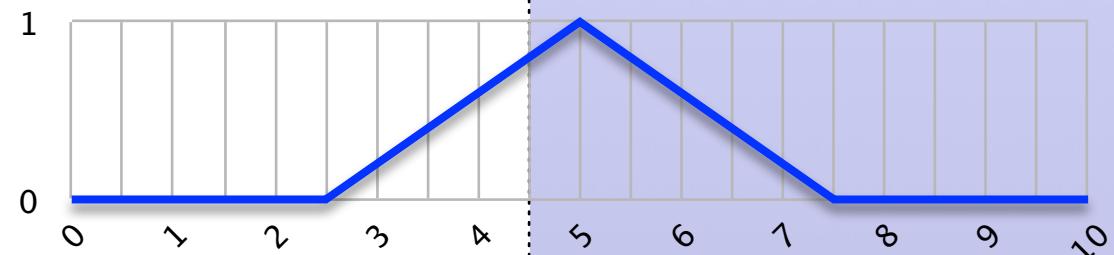
```
template<class T>
class triangleFunc : public Evaluator<T>::EvalFunc
{
public:
    triangleFunc(const T& _min, const T& _mid, const T& _max) :
        min(_min), mid(_mid), max(_max)
    {}

    virtual T operator()(const T& x)
    {
        if (x < min || x > max)
            return 0;

        return
            (x <= mid) ?
            (x-min)/(mid-min):
            (max-x)/(max-mid);
    }

    T min, mid, max;
};

void testShape()
{
    triangleFunc<float> func(2.5f, 5, 7.5f);
    Evaluator<float>::Shape s = Evaluator<float>::buildShape(0, 10, 0.5f, func);
    Evaluator<float>::printShape(cout,s);
}
```



```
[0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.5 9 9.5 10 ]
[0 0 0 0 0 0.2 0.4 0.6 0.8 1 0.8 0.6 0.4 0.2 0 0 0 0 0 ]
```



Compteur de lignes

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>

using namespace std;

int main (int argc, char * const argv[])
{
    if (argc<2)
    {
        cout << endl << "filename missing";
        return 0;
    }

    ifstream file(argv[1]);
    istream_iterator<char> it(file), itEnd;

    if (file)
    {
        file >> noskipws;
        cout << endl << "lines number: " << count(it, itEnd, '\n');
    }
    return 0;
}
```

Bilan des occurrences de mots

```
#ifndef WORDCOUNTER_H
#define WORDCOUNTER_H

#include <iostream>
#include <string>
#include <map>

class WordCounter
{
public:
    WordCounter() = default;
    WordCounter(std::istream& s);

    void count(std::istream& s);
    std::ostream& printOn(std::ostream& os) const;

private:
    typedef std::map<std::string, unsigned long> w_map;
    typedef w_map::value_type w_value;
    typedef w_map::iterator iterator;
    typedef w_map::const_iterator const_iterator;

    w_map words;
};

#endif
```

```
#include "wordcounter.h"
#include <sstream>

WordCounter::WordCounter(std::istream& s)
{
    count(s);
}

void WordCounter::count(std::istream& s)
{
    std::string current;

    while(s)
    {
        s >> current;
        (*words.insert(w_value(current,0)).first).second++;
    }
}

std::ostream& WordCounter::printOn(std::ostream& os) const
{
    for(const_iterator it = words.begin(); it != words.end(); ++it)
        os << (*it).first << "->" << (*it).second << std::endl;
}

return os;
```

`pair<iterator, bool> insert(const value_type& _Val);`

_Val :

The value of an element to be inserted into the map unless the map already contains that element or, more generally, an element whose key is equivalently ordered.

Return Value :

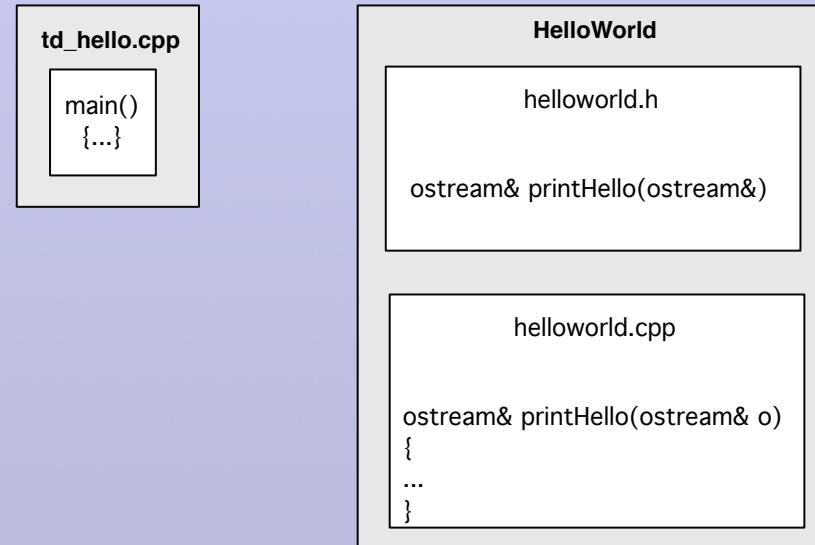
The first insert member function returns a pair whose bool component returns true if an insertion was made and false if the map already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

Premier Hello World

- Un simple programme principal imprimant la chaîne de caractères “Hello world” sur le standard de sortie !

Second Hello World

- Un programme composé d'un programme principal faisant appel à un module pour l'impression de la chaîne de caractères “Hello world” sur *stream* de sortie.



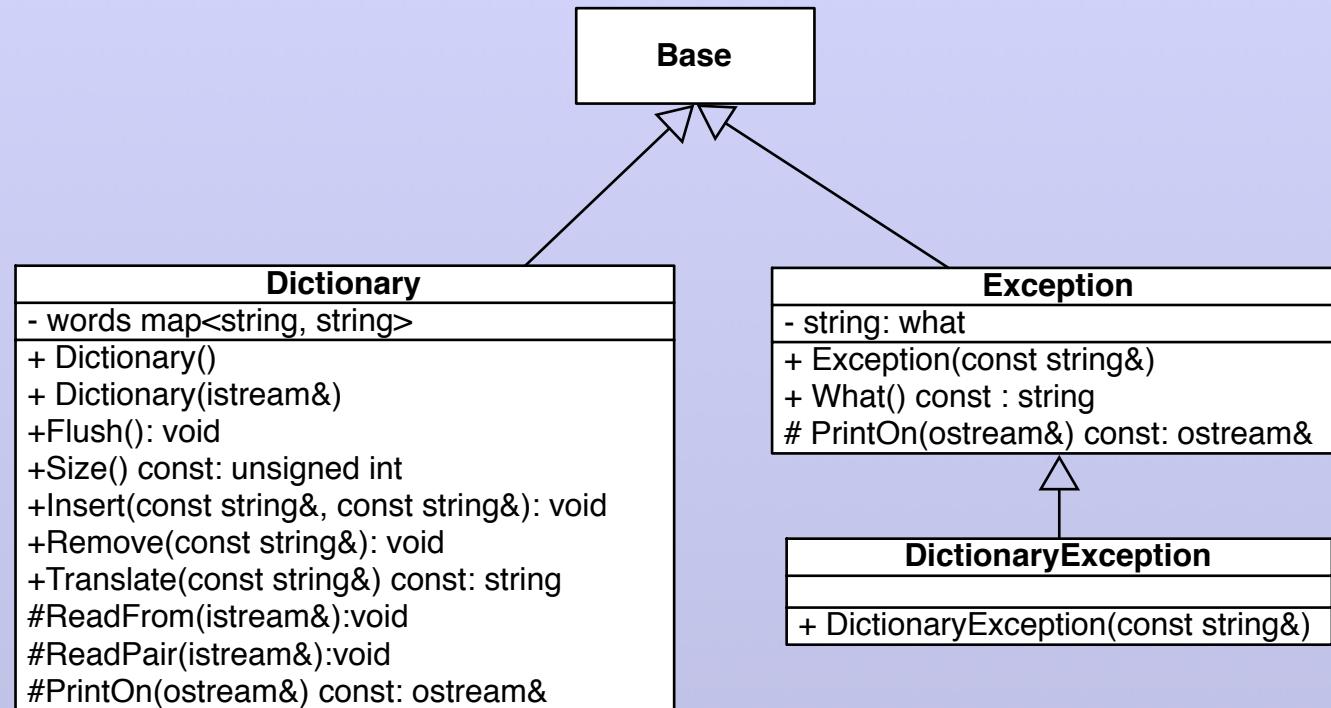
Ecrire et tester la classe DynamicArray

- Elle représente un tableau dynamique d'entier
- Elle doit permettre:
 - une instantiation par défaut, en précisant la taille ou encore par recopie,
 - fournir des information sur sa taille et la taille des blocs d'allocation,
 - permettre de changer ou d'accéder aux valeurs du tableau par des méthodes *get*, *set* et opérateur `[]`,
 - permettre d'augmenter sa taille sur demande,
 - permettre une représentation textuelles sur un *stream* de sortie.

Ecrire la classe Array<T>

- Elle représente un tableau dynamique d'un type abstrait T
- Elle doit permettre les mêmes fonctionnalités que la classe *DynamicArray*

Diagramme de classes



Classe Dictionnaire

```
#include "dictionary.h"

ostream& Dictionary::PrintOn(ostream& os) const
{
    for(const_iterator it=words.begin();it != words.end();++it)
        os << endl << (*it).first << '\t' << (*it).second;
    return os;
}
```

```
#ifndef DICTIONARY_H
#define DICTIONARY_H

#include "base.h"
#include <map>
using namespace std;

class Dictionary : public Base
{
public:
    Dictionary();
    Dictionary(istream& );
    virtual ~Dictionary() {};

    virtual void flush();
    virtual unsigned int Size() const;
    virtual void insert(const string&, const string&);
    virtual void remove(const string&);
    virtual string translate(const string&) const;

protected:
    virtual ostream& printOn(ostream&) const;
    virtual istream& readFrom(istream& );
    virtual istream& readPair(istream& );
    friend istream& operator >> (istream&, Dictionary&);

private:
    typedef map<string, string> Words;
    typedef Words::value_type value_type;
    typedef Words::iterator iterator;
    typedef Words::const_iterator const_iterator;

private:
    Words words;
};

#endif
```