

# Séance 1

## Programmation C++

Cours L3 Physique. Le 2012-2013

Mourad Ismail

`ismail@ujf-grenoble.fr`

`http://www-liphy.ujf-grenoble.fr/equipe/dyfcom/ismail`

Université Joseph Fourier




## UE DPINF353

### Volume horaire

- 40h TPs/projets

### Évaluation. UE coefficient 1 : 3 ECTS

- Compte-rendu de projet
-  Pas de 2<sup>ème</sup> session



## Équipe pédagogique

- Mourad Ismail <ismail@ujf-grenoble.fr>  
Laboratoire Interdisciplinaire de Physique, bâtiment E, bureau 311
- Vincent Doyeux <vincent.doyeux@ujf-grenoble.fr>  
Laboratoire Interdisciplinaire de Physique, bâtiment E, bureau 330



## Quelques mots sur le langage C++

- 4<sup>ème</sup> langage le plus utilisé au monde (1<sup>er</sup> si on le regroupe avec C et Objective-C). D'après <http://www.tiobe.com> :  
1. C, 2. java, 3. Objective-C et 4. C++
- Multiples paradigmes : programmation procédurale, programmation orientée objet, programmation générique, etc ...
- Pas de droits d'auteurs ©
- Développé par Bjarne Stroustrup dans les années 80 pour améliorer le langage C



## Liens utiles et bibliographie I

### Page web :

`www-liphy.ujf-grenoble.fr/equipe/dyfcom/ismail`

### Bibliographie

- La plupart des exemples présentés dans ce cours sont extraits du cours de Frédéric Coquel
- Claude Delannoy, Programmer en C++ - 5e édition, Édition EYROLLES 2006, (ISBN 2-212-115024), 616 pages
- Bjarne Stroustrup, Le langage C++ - 4e édition, Édition PEARSON EDUCATION FRANCE 2003, (ISBN 978-2-7440-7003-7), 1098 pages
- Tutoriel en ligne : <http://www.cplusplus.com>



## Liens utiles et bibliographie II

### Outils

- ✓ Environnement Linux
  - g++ : le compilateur C++ de GNU (licence GPL)  
<http://gcc.gnu.org/>
  - Éditeurs de textes : emacs, KWrite, gedit, ...
- Environnement Windows
  - MinGW (Minimalist GNU for Windows) : <http://www.mingw.org>
  - MSYS (Minimal SYStem) : <http://www.mingw.org>
  - Éditeurs de textes :
    - Notepad++ <http://notepad-plus.sourceforge.net>
    - crimson <http://www.crimsoneditor.com>
- Environnement MacOS
  - Xcode Version 2.4 ou ultérieur (contient gcc4)



# Plan

## 2 Notions de base

- Un premier programme
- Identificateurs et mots-clés
- Portée de variables
- Expressions et opérateurs
- Les instructions
- Le type tableau
- Pointeurs
- Tableaux et pointeurs
- Les références
- Allocation dynamique



# Hello World!!!

## Le plus petit programme en C++

```
int main () // fonction principale
{
    /*
    toutes ces lignes
    sont ignorées
    par le compilateur
    */
}
```

|toto.C1.cpp|

- Il s'agit d'un exemple de la fonction principale (**main**) qui ne prend aucun argument et n'exécute aucune instruction !
- Tout ce qui est entre `//` et EOL est un commentaire ignoré par le compilateur
- Toutes les lignes entre `/*` et `*/` sont des commentaires



# Hello World !! II

- ⊘ Tout programme C++ doit contenir une fonction nommée `main` (et une seule !)
- ⊘ Cette fonction doit retourner un entier (`int`).
- ⊘ Une valeur de retour non nulle signale un échec
- ⊘ Une valeur de retour nulle ou l'absence de valeur de retour signalent une exécution réussie



```
int main ()  
{  
    return 0; // retour de la fonction main  
}
```

|toto1.C1.cpp|



« Hello World »



# Hello World !! III

```
#include <iostream> //directive du préprocesseur. Gestion d'E/S
int main ()
{
    std::cout << "Hello World" << std::endl; // opérateur de flux.
                                             // Utilisation de la sortie standard
    return 0;
}
```

ou bien

```
#include <iostream>
using namespace std; // utilisation de l'espace de nom std
int main ()
{
    cout << "Hello World" << endl;
    return 0;
}
```

|helloWorld1.C1.cpp|



# Hello World !! IV

## Compilation avec g++ sous Linux

```
g++ helloWorld.C1.cpp -o helloWorld
```

## Exécution sous Linux

```
./helloWorld
```

```
Hello World
```



# Déclaration et types I

## Déclaration

```
type identificateur;
```

## Exemples de types d'entiers :

- `short int`
- `int`
- `long int`

La taille de chacun de ces types dépend de la machine. Par exemple : `short int` (16 bits), `int` (32 bits), `long int` (64 bits).



## Déclaration et types II

### 📄 Vérification. Fonction `sizeof`

```
#include <iostream>

using namespace std;

int main ()
{
    cout << "Nbr d'oct. d'un char = " << sizeof(char) << endl;
    cout << "Nbr d'oct. d'un short int = " << sizeof(short int) << endl;
    cout << "Nbr d'oct. d'un int = " << sizeof(int) << endl;
    cout << "Nbr d'oct. d'un long int = " << sizeof(long int) << endl;
    cout << "Nbr d'oct. d'un float = " << sizeof(float) << endl;
    cout << "Nbr d'oct. d'un double = " << sizeof(double) << endl;
    cout << "Nbr d'oct. d'un long double = " << sizeof(long double) << endl;
    return 0;
}
```

/types.C1.cpp



## Déclaration et types III

### Exécution :

```
Nbr d'oct. d'un char = 1
Nbr d'oct. d'un short int = 2
Nbr d'oct. d'un int = 4
Nbr d'oct. d'un long int = 8
Nbr d'oct. d'un float = 4
Nbr d'oct. d'un double = 8
Nbr d'oct. d'un long double = 16
```

- Dans ce cas, un `int` occupe 32 bits ce qui permet le stockage de :

$$-2^{31}, \dots, -1, 0, 1, \dots, 2^{31} - 1.$$

- Quelques types réels :

- `float`
- `double`
- `long double`



## Déclaration et types IV

- Le type booléen :
  - `bool` : `true` OU `false`
  - `true` possède la valeur 1
  - `false` possède la valeur 0

### Exemple :

```
bool b = 5; // 5 est converti en true, donc b = true
int i = true; // true est converti en entier, donc i=1
int j = b+i; // j= 1+1 = 2
```



## Mot clé `const`

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

int main ()
{
    const double pi = 3.1415926535897;
    const int dimension = 10;
    const int nombre_elements = 100 * dimension;

    pi = 3.1; //illégal, une constante ne peut être modifiée
    const int size; //illégal, une constante doit être initialisée
    return 0;
}
```

|const.C1.cpp|





## Notions de blocs, de portées et déclarations

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

int main ()
{ // portée P0
  int n;

  { //cette accolade introduit une nouvelle portée P1
    int m = 10; // m est accessible uniquement a l'interieur
                // de cette portee
    n = m + 1; // OK, n est accessible
  } // Fin de la portée P1

  n = m ; // Erreur à la compilation, m est en dehors de
          //la presente portee
  return 0;
} // Fin de la portée P0
```

|portee.C1.cpp|



## Opérateur d'affectation. Opérateurs arithmétiques

- `i = j` désigne une expression qui réalise une action : affecte la valeur de `j` à `i`
- addition `+`, soustraction `-`, multiplication `*`, division `/`, l'opérateur modulo `%`

### Exemples :

- ✗ `double x = 3/4;`  $\implies x = 0$ , le reste de la division est ignoré
- ✓ `double y = double(3)/4;`  $\implies y = 0.75$ , l'entier 3 est converti en 3. au format `double`
- ✓ `double y = 3/4.;`  $\implies y = 0.75$
- ✓ `double y = 3./4.;`  $\implies y = 0.75$
- `int n = 4%3;`  $\implies n = 1$ , le reste de la division euclidienne de 4 par 3 est 1



## Les opérateurs d'affectation élargie

- `i += k;` équivalent à `i = i + k;`
- `i -= 2*n;` équivalent à `i = i - 2*n;`
- `a *= b;` équivalent à `a = a * b;`
- `a /= (b+1);` équivalent à `a = a / (b+1);`



# Les opérateurs d'incrémentation et de décrémentation

- `int i = 4, j = 4;`
- `i++;`  $i = i + 1;$ . Soit  $i = 5$
- `++j;`  $j = j + 1;$ . Soit  $j = 5$
- `int m = i++;`
  - 1)  $m = i,$
  - 2)  $i = i + 1.$Soit  $m = 5, i = 6.$
- `int n = ++j;`
  - 1)  $j = j + 1,$
  - 2)  $n = j.$Soit  $j = 6, n = 6.$



## Les opérateurs relationnels I

- `>` (strictement supérieur à)
- `<` (strictement inférieur à)
- `>=` (supérieur ou égal à)
- `<=` (inférieur ou égal à)
- `==` (égal à)
- `!=` (différent de)

### ATTENTION !

Il convient de distinguer l'opérateur d'affectation `=` de l'opérateur de test à l'égalité `==`. Voir `|affectation.C1.cpp|`



## Les opérateurs relationnels II



```
int x = 1;
int y = 0;
if (y=x)
{
    cout << "Test verifie"<< endl;
}
else
{
    cout << "Test non verife" << endl;
}
```

*|affectation.C1.cpp|*



### Exécution :

Test verifie



# Les opérateurs logiques

- `&&` : opérateur *et*
- `||` : opérateur *ou*
- `!` : opérateur *non*



# Les instructions conditionnelles I

## L'instruction *if-else*

```
if(condition_1) instruction_1
else
{
    if(condition_2) instruction_2
    else
    {
        if(condition_3) instruction_3
        else instruction_4
    }
}
```





# Les instructions conditionnelles II

## équivalence avec une instruction conditionnelle ternaire

```
i = a > b ? a : b;
```

```
if (a > b) i = a;  
else      i = b;
```



## Les instructions conditionnelles III

### L'instruction *switch*

```
int i, j, k
// suite des instructions attribuant,
// entre autre, une valeur à i
switch(i)
{
    case 0 :           //exécution si i = 0
        j = 1;
        k = 2;
        break; // sortie du switch
    case 2 :           //exécution si i = 0
        j = 3;
        k = 4;
        break; // sortie du switch
    default ://exec si i diff. de 0 et 2
        j = 0;
        k = 0;
        break; // sortie du switch
}
```

## Les instructions d'itérations

### L'instruction *for*

```
for(expr_1 ; expr_2 ; expr_3) instruction
```

### L'instruction *while*. Faire tant que

```
while(expression) instruction
```

### L'instruction *do . . . while*. Faire jusqu'à

```
do  
instruction  
while(expression);
```

## Tableaux statiques

- `double vec[10] = {0};` : définit un vecteur de doubles de taille 10 dont toutes les composantes sont initialisées à 0
- `vec[0] = 1.0;` : affectation sur le 1er élément de `vec`
- `vec[9] = 11.0;` : affectation du dernier élément de `vec`
- `const int dim = 5; int a[2*dim] = {};` : définit un vecteur d'entiers de taille 10 dont toutes les composantes sont initialisées à 0
- `for(int i = 0; i < 2*dim; i++) a[i] = i*i;` : remplissage de a composante par composante



## Définition et exemple 1

- Une variable de type **T\*** est une variable automatique destinée à recevoir l'adresse d'une variable de type **T**

📌 Exemple :

```
double x = 0., y = 1.;
double* ptr = 0; //ptr est initialisé par le pointeur NULL
                //il ne pointe sur rien.

ptr = &x; //on affecte l'adresse de x au pointeur ptr
cout << "je pointe sur l'objet d'adresse "<< ptr
      <<" et ma valeur est "<< *ptr <<endl;

double z = *ptr;
cout << "je suis un double d'adresse "<< &z
      <<" et de valeur " << z <<endl;

ptr = &y;
z = *ptr;
cout <<"je pointe sur l'objet d'adresse "<< ptr
```

## Définition et exemple II

```
<<" et ma valeur est " << *ptr <<endl;
```

```
/*
```

```
* Notons que les deux dernières opérations sont équivalents
```

```
* à z = y mais elles ont été réalisées via un pointeur
```

```
*/
```



|pointeurs.C1.cpp|

### Le résultat de l'exécution :

```
je pointe sur l'objet d'adresse 0xbfedbe20 et ma valeur est 0  
je suis un double d'adresse 0xbfedbe10 et de valeur 0  
je pointe sur l'objet d'adresse 0xbfedbe18 et ma valeur est 1
```



## Deux règles

-  Une expression désignant un objet de type tableau est convertie par le compilateur en un pointeur constant sur son premier élément
-  L'opérateur `[]` est défini de sorte que si `exp1` et `exp2` désignent des expressions, la première de type pointeur et la seconde de type entier. Alors les expressions suivantes sont équivalentes :

`exp1[exp2]` et `*(exp1+exp2)`

### Pointeur versus tableau

```
int tabEntiers[20];
int* ptab;
ptab = tabEntiers;
*tabEntiers = 1;
ptab[0] = 1;
*(tabEntiers+1) = 2;
ptab[1] = 2;
```

## Tableaux à 2 dimensions

⇒ Exemple de déclaration d'un tableau statique en 2D :

```
double a[5][7];
```


- Le tableau a est stocké en mémoire ligne par ligne : **a[0][0]**, a[0][1], a[0][2], a[0][3], a[0][4], a[0][5], a[0][6], **a[1][0]**, a[1][1] . . .
- ⊘ a pointe vers a[0][0]
- ⊘ a+7\*i+j pointe vers a[i][j]
- ⊘ Le tableau a aurait pu être déclaré comme un pointeur de pointeur : 


```
double** a;
```






## Définition et exemple

 une référence est un alias pour un objet déjà déclaré

 **T&** désigne une référence sur un objet de type **T**

 **Syntaxe :** `T& ref_v = v;`

### Exemple

```
int n = 5;
int& r = n; // r est une réf. sur un entier.
           //r réfère à n
int m = r; // m = 5 puisque r est un alias de n
r = 1; // n = 1 puisque r est un synonyme de n
```



## Les opérateurs `new` et `delete`

```
T* ptr = 0; // déclaration d'un pointeur sur un
           // type T et son initialisation à 0
ptr = new T; // demande d'allocation de mémoire
           // pour un type T
// forme condensée : T* ptr = new T;
*ptr = 1;    // utilisation de la variable dynamique
           // par le biais du pointeur
delete ptr;  // libération de l'espace mémoire
```

### Recommandation

```
delete ptr;
ptr = 0;
```



## Les opérateurs `new[]` et `delete[]`. Tableaux dynamiques

```
T* ptr = 0;      // déclaration d'un pointeur
                //sur un type T
ptr = new T[n]; // allocation de mémoire pour
                //un tableau de type T
                // et de taille n
// forme condensée : T* ptr = new T[n];
delete[] ptr; // libération de l'espace mémoire
ptr = 0;
```



## Exemple I

### Application : calcul des nombres premiers

```
#include <iostream>
#include <iomanip> // pour setw()
#include <cstdlib> // pour exit()
using namespace std;
int main()
{
    int max = 0; // Nbr de nombres premiers demandés
    int count = 3; // Nbr de nombres premiers trouvés
    long int test = 5; // Candidat à être un nbr premier
    bool est_premier = true; // indique qu'un nbr premier est trouvé

    cout << "Entrez le nbr de nombres premiers désirés > 3" << endl ;
    cin >> max;
    if (max < 4) exit(0);
    long* premiers = new long[max]; // allocation dynamique du tableau
    // des nbrs premiers
```



## Exemple II

```
*premiers = 2;
*(premiers+1) = 3;
*(premiers+2) = 5;

do
{
    test += 2;
    int i = 0;

    do
    {
        est_premier = (test % premiers[i]) > 0;
    } while ( (++i < count) && est_premier);

    if (est_premier)
        premiers[count++] = test;
} while (count < max);
```

## Exemple III

```
for (int i = 0; i<max; i++)
{
    if(i%5 == 0) cout << endl;
    cout << setw(10) << *(premiers+i);
}
cout << endl;
delete[] premiers;
premiers = 0;
return 0;
}
```

*[nbrPremiers.C1.cpp]*

### 🕒 Exécution :

Entrez le nbr de nombres premiers désirés > 3

10

2

3

5

7

11

13

17

19

23

29



# Plan

## 3 Fonctions

- Déclaration
- Définition
- Transmission d'arguments
- Valeurs de retour
- Fonctions mathématiques
- Fonctions récursives
- Variables locales de la classe static
- Surdéfinition de fonctions

## 4 Gestion de projets. Compilation séparée

- Makefile
- Fichiers d'en-tête (*headers*)

## 5 Mécanismes d'abstraction. Les structures



# Déclaration

## Prototype

```
type_de_retour identificateur (liste_de_paramètres);
```

### Exemples :

```
int main ();  
int main (int argc, char* argv[]); // main avec arguments de  
// ligne de commande  
double func_1 (double x);  
void func_2 (double x);
```

## Prototype complet

```
double func_4 (int nb_iter, double initial, double solution);  
double func_4 (int, double, double);
```





## Définition d'une fonction

### En-tête (*prototype complet*)

```
type_de_retour identificateur (liste_de_paramètres)
```

### Le corps d'une fonction

```
{  
    ...  
    return expression; // si type_de_retour n'est pas void  
}
```



## Exemple de définition d'une fonction

 Exemple :

```
double abs_som (double u, double v)
{
    double result = u + v;
    if(result > 0) return result;
    else return -result;
}
```

### Ordre de déclaration

Pour être utilisée, une fonction doit être déclarée avant son utilisation



## Passage par valeur

- `fonc` est une fonction qui prend un `double` comme argument et qui retourne un `double` :

```
double fonc(double ); //Déclaration de la fonction fonc
```

- Lors de l'appel de la fonction `fonc`, seule la valeur de l'argument est transmise.

```
double x = 1;  
double y = fonc(3*x+2); // utilisation de la fonction fonc
```

- Dans cet exemple c'est la valeur 5 qui est transmise en argument à la fonction `fonc`



# Passage par valeurs. Exemple I

## Échange de variables

```
void echange (int a, int b)
{
    int c;
    cout << "debut echange " << a << " " << b << endl;
    c = a;
    a = b;
    b = c;
    cout << "fin echange " << a << " " << b << endl;
}
```

*/echange.C2.cpp/*

## Utilisation



## Passage par valeurs. Exemple II

```
void echange (int a, int b); // déclaration de la fonction
int n = 1, p = 2;

cout << "avant appel " << n << " " << p << endl;
echange (n,p);
cout << "apres appel " << n << " " << p << endl;
```

/echange.C2.cpp/

### 👤 Vérification :

```
avant appel 1 2
debut echange 1 2
fin echange 2 1
apres appel 1 2
```



Pas d'échange à la sortie de la fonction !! Que s'est-il passé ?



# Transmission par pointeurs. Exemple I

## Échange de variables

```
void echange (int* a, int* b)
{
    int c;
    cout << "debut echange " << *a << " " << *b << endl;
    c = *a;
    *a = *b;
    *b = c;
    cout << "fin echange " << *a << " " << *b << endl;
}
```

|echangeAdr.C2.cpp|

## Utilisation



## Transmission par pointeurs. Exemple II

```
void echange (int* a, int* b); // déclaration de la fonction
int n = 1, p = 2;

cout << "avant appel " << n << " " << p << endl;
echange (&n, &p);
cout << "apres appel " << n << " " << p << endl;
```

|echangeAdr.C2.cpp|

### 👤 Vérification :

```
avant appel 1 2
debut echange 1 2
fin echange 2 1
apres appel 2 1
```



Échange réussi.



# Transmission par Références. Exemple I

## Échange de variables

```
void echange (int& a, int& b)
{
    int c;
    cout << "debut echange " << a << " " << b << endl;
    c = a;
    a = b;
    b = c;
    cout << "fin echange " << a << " " << b << endl;
}
```

|echangeRef.C2.cpp|

## Utilisation





## Transmission par Références. Exemple II

```
void echange (int& a, int& b); // déclaration de la fonction
int n = 1, p = 2;

cout << "avant appel " << n << " " << p << endl;
echange (n,p);
cout << "apres appel " << n << " " << p << endl;
return 0;
```

|echangeRef.C2.cpp|

### 🕒 Vérification :

```
avant appel 1 2
debut echange 1 2
fin echange 2 1
apres appel 2 1
```



Échange réussi.



## Tableaux transmis en arguments

### Rappel

L'appel `f(tab);` est équivalent à `f(&tab[0]);`

### Remarque

Seule l'adresse du premier élément d'un tableau est transmis en argument

### Définition

Si `tab` est un tableau d'entiers, le prototype d'une fonction compatible peut être `void (int* t);` ou `void (int t[]);` ou

`void (int taille, int t[]);`



## Arguments par défaut I

👉 Exemple de déclaration et d'appels :

```
// Déclaration de la fonction func  
void func(int n, int p = 1); //Prototype avec une valeur  
//par défaut  
  
// Exemples d'appels de la fonction func  
func(i, j); // Appel « classique »  
func(i); // Appel avec un seul argument ⇔ func(i, 1)  
func(); // Illégal car le premier argument n'a pas  
// de valeur par défaut
```

👉 Exemple de définition :

```
void func (int n, int p) //en-tête habituelle  
{  
    //corps de la fonction  
}
```



## Arguments par défaut II

- les valeurs par défaut peuvent dépendre d'autres variables

```
int m;  
// Suite d'instructions attribuant une valeur à m  
void func (int n, int p = 2*m+1); // La valeur par défaut  
// dépend de m
```



## Valeurs de retour

### Rappel

Une fonction dont le type de retour n'est pas `void` doit contenir une instruction `return`

### Remarque

```
type_de_retour = T  
⚡ type_de_retour = T*, T&
```

### ⚠ Attention !

Retourner un pointeur ou une référence sur une variable automatique locale à la fonction est une erreur. Pourquoi ?



## Construction et destruction de variables/objets

- ⛔ Une **variable locale** est créée dès que le flux de contrôle passe par sa déclaration. Elle est détruite dès que se termine l'exécution du bloc dans lequel elle se trouve



## Construction et destruction de variables/objets

- ⛔ Une **variable locale** est créée dès que le flux de contrôle passe par sa déclaration. Elle est détruite dès que se termine l'exécution du bloc dans lequel elle se trouve
- ⛔ Un **objet dynamique** est créé par l'opérateur `new`. Il n'est supprimé que lorsque `delete` est appelé



## Construction et destruction de variables/objets

- ⊘ Une **variable locale** est créée dès que le flux de contrôle passe par sa déclaration. Elle est détruite dès que se termine l'exécution du bloc dans lequel elle se trouve
- ⊘ Un **objet dynamique** est créé par l'opérateur `new`. Il n'est supprimé que lorsque `delete` est appelé
- ⊘ Un objet **statique local** est construit à la première rencontre entre le flux de contrôle et la définition de l'objet (voir exemple fonction factoriel recursive). Il est détruit à la fin du programme.





## Construction et destruction de variables/objets

- ⊘ Une **variable locale** est créée dès que le flux de contrôle passe par sa déclaration. Elle est détruite dès que se termine l'exécution du bloc dans lequel elle se trouve
  
- ⊘ Un objet **statique local** est construit à la première rencontre entre le flux de contrôle et la définition de l'objet (voir exemple fonction factoriel recursive). Il est détruit à la fin du programme.



## Construction et destruction de variables/objets

- ⊘ Une **variable locale** est créée dès que le flux de contrôle passe par sa déclaration. Elle est détruite dès que se termine l'exécution du bloc dans lequel elle se trouve
- ⊘ Un **objet dynamique** est créé par l'opérateur `new`. Il n'est supprimé que lorsque `delete` est appelé
- ⊘ Un objet **statique local** est construit à la première rencontre entre le flux de contrôle et la définition de l'objet (voir exemple fonction factoriel recursive). Il est détruit à la fin du programme.
- ⊘ Une **variable globale** est une variable définie en dehors de toute fonction. Elle est initialisée avant l'appel du `main()` et détruite après la fin de son exécution.



## Retour par pointeur. Exemple I

Tirage aléatoire de 10 nombres réels et leur normalisation entre 0 et 1.

Fichiers d'en-têtes :

*/funcRetourPoint.C2.cpp*

```
1 #include <iostream> // Vous y êtes déjà habitués
2 #include <iomanip> // pour la fonction setw
3 #include <cstdlib> // pour la fonction rand
```

Une variable globale :

```
1 const int taille = 10;
```

La fonction `main`. Déclaration des fonctions :

```
1 void hasard (double max, int size, double tab[]);
2 void affiche(int size, const double tab[],
3             const string & titre = "Valeurs ",
4             int largeur = 10, int par_ligne = 5);
5 double * largest (int size, double *tab);
6 double * smallest(int size, double *tab);
```



## Retour par pointeur. Exemple II

📄 La fonction `main`. Suite des instructions :

```
1  double vec[taille] = {0};
2
3  double maximum = 0.;
4  cout << "Entrez la valeur d'initialisation maximale " << endl;
5  cin >> maximum;
6
7  hasard(maximum, taille, vec);
8  cout << endl;
9  affiche(taille, vec, "Valeurs initiales",12);
10
11 double min = *smallest (taille, vec);
12
13 // Décale les valeurs de sorte a annuler la plus petite
14 for(int i=0; i<taille; i++) vec[i] -= min;
15
16 double max = *largest(taille, vec);
17 // Renormalise les valeurs à 1
```

## Retour par pointeur. Exemple III

```

18  for(int i=0; i<taille; i++) vec[i] /= max;
19
20  affiche(taille, vec, "Valeurs renormalisees");
21  return 0;

```

### Exécution :

```

1  Entrez la valeur d initialisation maximale
2  2
3
4  Valeurs initiales
5  1.68038  0.788766  1.5662  1.59688  1.82329
6  0.395103  0.670446  1.53646  0.555549  1.10794
7
8  Valeurs renormalisees
9  0.89993  0.275637  0.819985  0.841468  1
10 0  0.192791  0.799162  0.112343  0.499119

```



## Retour par pointeur. Exemple IV

Quelques Détails sur les fonctions utilisées :

### Fonction hasard

```
1 void hasard (double max, int size, double * tab)
2 {
3
4     for (int i=0; i<size; i++)
5         tab[i] = double(rand()) * max / RAND_MAX;
6 }
```

Quel est le type de retour ?



## Retour par pointeur. Exemple V

### Fonction largest

```
1 //trouver l'adresse de l'élément possédant la plus grande valeur
2 double * largest(int size, double * tab)
3 {
4     int indexMax = 0;
5     for(int i=0; i<size; i++)
6         indexMax = tab[indexMax] < tab[i] ? i : indexMax;
7     return &tab[indexMax];
8 }
```

Quelle est la valeur retournée ?

### Fonction smallest Idem !



## Retour par pointeur. Exemple VI

### Fonction affiche

```
1 void affiche(int size, const double * tab, const string & titre,
2             int largeur, int par_ligne)
3 {
4     cout << endl << titre;
5     for(int i=0; i<size; i++)
6     {
7         if(!(i%par_ligne)) cout << endl;
8         cout << setw(largeur) << tab[i];
9     }
10    cout << endl;
11 }
```

*[funcRetourPoint.C2.cpp]*

Identifier les appels de la fonction affiche dans le main.  
Commenter !





# Bibliothèque standard de fonctions mathématiques

```
double abs(double); //valeur absolue
double ceil(double d); // plus petit entier >= à d
double floor(double d); //plus grand entier >= à d
double sqrt(double); //racine carrée
double pow(double d, double e); // d à la puissance e
double pow(double d, int n); // d à la puissance n
double exp(double); //exponentielle
double log(double); //logarithme neperien
double cos(double); //cosinus
double sin(double); //sinus
double tan(double); //tangente
double acos(double); //arccosinus
double asin(double); //arcsinus
double atan(double); //arctangente
```



## Un exemple

- Une fonction peut-elle faire appel à elle-même ?

```
long double factoriel (int n)
{
    if (n == 0) return 1;
    return n*factoriel(n-1);
}
```

|factoriel.C2.cpp|

- Version non récursive :

```
long double factoriel (int n)
{
    if(n == 0) return 1;
    long double result = n;
    for( ; n>1; ) result *= --n;
    return result;
}
```

|factoriel.2.C2.cpp|

# Combien de fois la fonction `factoriel` est-elle appelée ? I

📌 Rappel de la fonction `factoriel` :

```
long double factoriel (int n)
{
    int compteur = 1;
    if (n == 0) return 1;
    cout <<"la fonction factoriel est appele " <<compteur++
         <<" fois"<<endl;

    return n*factoriel(n-1);
}
```

*|factoriel.C2.cpp|*

📌 Exemple d'utilisation :



## Combien de fois la fonction `factoriel` est-elle appelée ? II

```
int main ()
{
    int n;
    cout <<"Donnez un entier positif n = ";
    cin >> n;
    cout <<n<<"!= " << factoriel(n) <<endl;
    return 0;
}
```

|factoriel.C2.cpp|

### 🕒 Exécution :

```
Donnez un entier positif n = 3
la fonction factoriel est appele 1 fois
la fonction factoriel est appele 1 fois
la fonction factoriel est appele 1 fois
3!= 6
```



## Combien de fois la fonction `factoriel` est-elle appelée ? III

### Ajout du mot-clé `static`

```
long double factoriel (int n)
{
    static int compteur = 1;
    if (n == 0) return 1;
    cout <<"la fonction factoriel est appele " <<compteur++
         <<" fois"<<endl;
    return n*factoriel(n-1);
}
```

*|factoriel.C2.cpp|*



```
Donnez un entier positif n = 3
la fonction factoriel est appele 1 fois
la fonction factoriel est appele 2 fois
la fonction factoriel est appele 3 fois
3!= 6
```



## Combien de fois la fonction `factoriel` est-elle appelée ? IV



La variable `compteur`, de type `static`, n'est initialisée qu'une seule fois ! (à la première rencontre entre le flux de contrôle et la définition de la variable)



## Surcharge (ou surdéfinition) de fonctions I

Peut-on créer des fonctions « qui portent le même nom » et qui agissent différemment en fonction du type des objets passés en arguments ?

- On considère cette implémentation d'une fonction puissance qui calcule  $x^a$  :

```
double puissance (const double x, const int a)
{
    double result = 1;
    if(a>0)
    {
        for(int i=0; i<a; i++)
            result *= x;
        cout << "Appel de la fonction puissance (double, int) : ";
        return result;
    }
    else if(x!=0)
    {
```

## Surcharge (ou surdéfinition) de fonctions II

```
for(int i=0; i<-a; i++)
    result *= 1./x;
cout << "Appel de la fonction puissance (double, int) : ";
return result;
}
else
{
    cout << "Appel de la fonction puissance (double, int) : ";
    cout << "Indetermination!!" << endl;
    exit(1);
}
}
```

*|surchargePuissance.C2.cpp|*

 On teste notre fonction avec ces appels :





## Surcharge (ou surdéfinition) de fonctions III

```
double x = 2.;
int a1 = -1;
double a2 = 0.5;
cout << x << "^" << a1 << " = " << puissance(x, a1) << endl;
cout << x << "^" << a2 << " = " << puissance(x, a2) << endl;
int y = 2;
cout << y << "^" << a1 << " = " << puissance(x, a1) << endl;
cout << y << "^" << a2 << " = " << puissance(x, a2) << endl;
```

*|surchargePuissance.C2.cpp|*

### 🔍 Résultat :

```
1 Appel de la fonction puissance (double, int) : 2^-1 = 0.5
2 Appel de la fonction puissance (double, int) : 2^0.5 = 1
3 Appel de la fonction puissance (double, int) : 2^-1 = 0.5
4 Appel de la fonction puissance (double, int) : 2^0.5 = 1
```

Commenter les résultats des lignes 2 et 4



## Surcharge (ou surdéfinition) de fonctions IV

📖 On ajoute une autre fonction qui porte « le même nom » puissance et qui calcule  $x^a = e^{a \log(x)}$  pour  $x \in \mathbb{R}_+^*$  :

```
double puissance (const double x, const double a)
{
    if(x>0)
    {
        cout << "Appel de la fonction puissance (double, double) : ";
        return exp(a*log(x) );
    }
    else
    {
        cout << "Appel de la fonction puissance (double, double) : ";
        cout << "Indetermination!!" << endl;
        exit(1);
    }
}
```

|surchargePuissance.C2.cpp|

## Surcharge (ou surdéfinition) de fonctions V

### 👤 Résultat :

```

1 Appel de la fonction puissance (double, int) : 2^-1 = 0.5
2 Appel de la fonction puissance (double, double) : 2^0.5 = 1.41421
3 Appel de la fonction puissance (double, int) : 2^-1 = 0.5
4 Appel de la fonction puissance (double, double) : 2^0.5 = 1.41421

```

### 🔗 Que va-t-il se passer avec cet appel ?

```

int y1 = 2.; long int a3 = 2;
cout << y1 << "^" << a3 << " = " << puissance(y1,a3) << endl;

```

*|surchargePuissance.C2.cpp|*

### 📝 Compilation :

```

surchargePuissance.C2.cpp: In function 'int main()':
surchargePuissance.C2.cpp:20: error:
call of overloaded 'puissance(int&, long int&)' is ambiguous
candidates are: double puissance(double, int)
                double puissance(double, double)

```



## Conversion explicite. Mot-clé `static_cast`

- ☞ On indique explicitement quel genre de conversion on veut effectuer

```
int y1 = 2.; long int a3 = 2;
```

```
cout << y1 << "^" << a3 << " = " <<  
puissance(static_cast<double>(y1), static_cast<int>(a3)) << endl;
```

*|surchargePuissance.C2.cpp|*

- ☞ Exécution :

```
Appel de la fonction puissance (double, int) : 2^2 = 4
```



## Exemple simple de Makefile

Comment compiler un projet contenant plusieurs fichiers sources et utilisant une (ou plusieurs) bibliothèques externes ?

```

IDIR = /usr/include/qt4
ODIR = .
Cxx = g++
CFLAGS = -I$(IDIR) -Wall
LIBS = -L/usr/lib -lQtCore
OBJ = main.o fic_1.o fic_2.o
%.o : %.cpp
    $(Cxx) -c -o $@ $< $(CFLAGS)
mon_exe : $(OBJ)
    $(Cxx) -o $@ $^ $(LIBS)
clean :
    rm -f $(ODIR)/*.o *~ core
  
```

*# Quelques variables internes :*  
*# \$@ : nom de la cible*  
*# \$< : nom de la dernière dépendance*  
*# \$? : liste des dépendances récentes*  
*# que la cible*  
*# \$^ : liste des dépendances*

|Makefile|

**Exécution :** make

```

g++ -c -o main.o main.cpp -I/usr/include/qt4 -Wall
g++ -c -o fic_1.o fic_1.cpp -I/usr/include/qt4 -Wall
g++ -c -o fic_2.o fic_2.cpp -I/usr/include/qt4 -Wall
g++ -o mon_exe main.o fic_1.o fic_2.o -L/usr/lib -lQtCore
  
```



## Fichiers d'en-tête

### Définition

fichiers `.h` ou `.hpp` destinés à contenir les déclarations de fonctions

- Ils sont à inclure dans les fichiers sources `.c` ou `.cpp` :

```
#include "nomFichier.hpp"
```

- Pour éviter les doublons, on utilise les directives (`#ifndef`, `#define`, `#endif`)

```
#ifndef FIC_HPP
#define FIC_HPP
// contenu du fichier en-tête fic.hpp
#endif // FIC_HPP
```

### Remarque

Ces fichiers sont compilés implicitement par le compilateur. On ne les met pas dans le Makefile.

## Modéliser une information I

### Syntaxe générale

```
struct identificateur
{
    type_1 identificateur_1;
    type_j identificateur_j;
    type_n identificateur_n;
};
```

### Exemple. Les nombres complexes

- $z \in \mathbb{C} \Leftrightarrow z = (Re, Im) \in \mathbb{R}^2$
- Création d'une Structure Complexe composée de deux **double**
- Complexe=(**double**, **double**)

## Modéliser une information II

▶ Exemple de conception.

✍️ déclaration de la structure :

```
struct Complexe
{
    double Re;
    double Im;
};
```

*/complexe.C2.cpp*

✍️ Utilisation :





## Modéliser une information III

```

int main()
{
    Complexe z1; // Déclaration d'une variable  $z_1 \in \mathbb{C}$ 
    Complexe z2 = {1,1};
    Complexe z3 = {1};
    Complexe z4 = z2; // équivalent a Complexe z4(z2);
    z4.Im = z4.Re + 2.*z3.Im;
    Complexe * ptr = &z1;
    (*ptr).Re = 2.;
    (*ptr).Im = 1.; //  $z_1 = 2 + i$ 
    cout <<"(*ptr).Re = " << (*ptr).Re << endl;
    ptr->Re = 3.;
    ptr->Im = 4.; //  $z_1 = 3 + 4i$ 
    cout <<"ptr->Im = " << ptr->Im << endl;
    return 0;
}

```

/complexe.C2.cpp



# Structures et fonctions I

## Objectifs


- Écriture de fonctions manipulant des nombres complexes :
- une fonction qui conjugue un nombre complexe
- une fonction qui calcule la somme de deux nombres complexes
- une fonction qui affiche un nombre complexe sous la forme  $a + ib$
- Première approche. Des fonctions « standards » :
  - ✍ fonction addition :



## Structures et fonctions II

```
Complexe addition(const Complexe & z1, const Complexe & z2)
{
    Complexe result;
    result.Re = z1.Re + z2.Re;
    result.Im = z1.Im + z2.Im;
    return result;
}
```

|structFunc.C2.cpp|

 fonction conjugué :

```
Complexe conjugué(Complexe z)
{
    z.Im = -z.Im;
    return z;
}
```

|structFunc.C2.cpp|



## Structures et fonctions III

✎ fonction affiche :

```
void affiche(const Complexe & z)
{
    cout << z.Re;
    if(z.Im > 0) cout << " + i ";
    else if(z.Im < 0) cout << " - i ";
    if(z.Im) cout << abs(z.Im)<< endl;
}
```

|structFunc.C2.cpp|

✎ Utilisation :



## Structures et fonctions IV

```

Complexe conjugue(Complexe z);
Complexe addition(const Complexe & z1, const Complexe & z2);
void affiche(const Complexe & z);

Complexe z1 = {1.,2.};
Complexe z2 = conjugue(z1);
Complexe z3;
z3 = addition(z1,z2);
affiche(z1); affiche(z2); affiche(z3);

```

|structFunc.C2.cpp|

 Exécution :

```

1 + i 2
1 - i 2
2

```



# Fonctions membres I

## Définition

**Fonctions membres** (ou **méthodes**) : « attacher » des fonctions à une structure

✍ Déclaration de la structure :

```
struct Complexe
{
    // déclarations des données membres
    double Re;
    double Im;
    //déclaration des fonctions membres
    void affiche();
    Complexe conjugue();
    Complexe addition(const Complexe & z);
};
```



## Fonctions membres II

|structFuncMbre.C2.cpp|

- Définitions des fonctions membres (méthodes) :

📄 fonction affiche

```
void Complexe::affiche()
{
    cout << Re;
    if(Im > 0) cout << " + i ";
    else if(Im < 0) cout << " - i ";
    if(Im) cout << abs(Im) << endl;
}
```

|structFuncMbre.C2.cpp|

📄 fonction conjugue



## Fonctions membres III

```
Complexe Complexe::conjugue()  
{  
    Complexe result;  
    result.Re = Re;  
    result.Im = -Im;  
    return result;  
}
```

|structFuncMbre.C2.cpp|

### fonction addition

```
Complexe Complexe::addition(const Complexe & z)  
{  
    Complexe result = {Re+z.Re, Im+z.Im};  
    return result;  
}
```

|structFuncMbre.C2.cpp|





## Fonctions membres IV

### Utilisation :

```
int main ()
{
    void affiche();
    affiche();

    Complexe z1 = {1.,0.};
    z1.affiche();
    cout << endl;
}
void affiche() { cout << " Voici un exemple " << endl; }
```


|structFuncMbre.C2.cpp|

### Exécution :

```
Voici un exemple
1
```

### ● Deux fonctions `affiche()` ??

## Pointeur d'auto-référence : `this`

 Argument implicite des fonctions membres. Pointeur sur l'objet pour lequel la fonction a été appelée.

 Exemple :

```

Complexe Complexe::addition(const Complexe & z)
{
    Complexe result = {this->Re + z.Re, this->Im + z.Im};
    return result;
}
Complexe Complexe::conjugue_moi()
{
    Im = -Im;
    return *this;
}

```

 Utilisation :

```

Complexe z4 = {2., 3.}; // z4 = 2 + 3i
z4.conjugue_moi(); // z4 = 2 - 3i

```

# Plan

## 6 Les classes

## 7 Constructeurs

- Définition et exemple
- Exemple de conception



# Les classes versus structures I

- ✍ On remplace le mot-clé `struct` par `class` dans l'exemple précédent :

```
struct Complexe
{
    // déclarations des données membres
    double Re;
    double Im;
    //déclaration des fonctions membres
    void affiche();
    Complexe conjugue();
    Complexe addition(const Complexe & z);
};
```

|structFuncMbre.C2.cpp|



## Les classes versus structures II

✍ La déclaration de notre classe :

```
class Complexe
{
    // déclarations des données membres
    double Re;
    double Im;
    //déclaration des fonctions membres
    void affiche();
    Complexe conjugue();
    Complexe addition(const Complexe & z);
};
```

|classVstruct.C3.cpp|



## Les classes versus structures III

✍ On essaye de tester la classe avec ce programme simple :

```
int main ()
{
    Complexe z1;
    z1.Re = 4.;
    z1.affiche();
    Complexe z2 = z1.conjugué();
    Complexe z3 = {1., 2.};
    z3 = z1;
    return 0;
}
```

|classVstruct.C3.cpp|



## Les classes versus structures IV

### Compilation :

```
classVstruct.C3.cpp: In member function
`Complexe Complexe::addition(const Complexe&)`:
classVstruct.C3.cpp:30: error: braces around initializer for
non-aggregate type `Complexe`
classVstruct.C3.cpp: In function `int main()`:
classVstruct.C3.cpp:7:error: `double Complexe::Re` is private
classVstruct.C3.cpp:38:error:within this context
classVstruct.C3.cpp:14:error: `void Complexe::affiche()` is private
classVstruct.C3.cpp:39:error:within this context
classVstruct.C3.cpp:21:error: `Complexe Complexe::conjugue()` is private
classVstruct.C3.cpp:40:error:within this context
classVstruct.C3.cpp:41:error:braces around initializer for
non-aggregate type `Complexe`
```



## Les classes versus structures V

✗ Les numéros des lignes incriminées : 30, 38, 39, 40 et 41

```
1 Complexe Complexe::addition(const Complexe & z)
2 {
3     Complexe result = {Re+z.Re, Im+z.Im};
4     return result;
5 }
```

```
1 int main ()
2 {
3     Complexe z1;
4     z1.Re = 4.;
5     z1.affiche();
6     Complexe z2 = z1.conjugue();
7     Complexe z3 = {1., 2.};
8     z3 = z1;
9     return 0;
10 }
```

*[classVstruct.C3.cpp]*



# Les classes versus structures VI

- Rappel de la déclaration de la classe

```
1 class Complexe
2 {
3     // déclarations des données membres
4     double Re;
5     double Im;
6     //déclaration des fonctions membres
7     void affiche();
8     Complexe conjugue();
9     Complexe addition(const Complexe & z);
10 };
```

/classVstruct.C3.cpp/



# Contrôle d'accès par mots-clés

## Modèle

```
class A
{
    // membres (données ou fonctions) par défaut privés

    public :
    // membres (données ou fonctions) publics

    private :
    // membres (données ou fonctions) privés

};
```



## Modification de la déclaration de la classe `Complexe` I

- ☞ On ajoute par exemple le mot-clé `public` avant les déclarations des données et des fonctions membres

```
class Complexe
{
public :
    //membres publics
    double Re;
    double Im;
    //fonctions membres publiques
    void affiche();
    Complexe conjugue();
    Complexe addition(const Complexe & z);
};
```

|classPublic.C3.cpp|



## Modification de la déclaration de la classe `Complexe` II

- Le programme fonctionne correctement mais nous aimerions utiliser davantage les fonctionnalités du langage quant aux contrôle d'accès aux données



## Classe Complexe en représentation interne I

### Conditions :

- Les données membres `Re` et `Im` doivent être privées
- Les fonctions membres peuvent rester publiques

🔗 Reprenons le programme précédent et y apportons les modifications nécessaires


```
class Complexe
{
private: // Donnés membres privées
    double Re;
    double Im;
public: //fonctions membres publiques
    void affiche();
    Complexe conjugue();
    Complexe addition(const Complexe & z);
};
```

|classeComplexeRepInt.C3.cpp



## Classe Complexe en représentation interne II

Il faut aussi corriger les instructions devenues illégales :

 On ne peut plus utiliser `z1.Re = 4.;` et `Complexe z3 = {1.,2.};` dans la fonction `int main()`


 `z1.Re` et `z1.Im` sont inaccessibles depuis l'extérieur de la classe

- Il faut aussi modifier la fonction

```
Complexe Complexe::addition(const Complexe & z)
{
    Complexe result = {Re+z.Re, Im+z.Im};
    return result;
}
```



## Classe Complexe en représentation interne III

 Par exemple :

```
Complexe Complexe::addition(const Complexe & z)
{
    Complexe result;
    result.Re = this->Re + z.Re;
    result.Im = this->Im + z.Im;
    return result;
}
```

*|classeComplexeRepInt.C3.cpp|*



## Classe Complexe en représentation interne IV

📄 Exemple d'utilisation de la classe :

```
int main ()
{
    Complexe z1;
    z1.affiche();
    Complexe z2 = z1.conjugué(); //z2 =  $\overline{z_1}$ 
    z2.affiche();
    Complexe z3;
    z3 = z2;
    z3.affiche();
    Complexe z4;
    z4 = z1.addition(z3.conjugué()); // z4 = z1 +  $\overline{z_3}$  = z1 +  $\overline{z_2}$ 
    z4.affiche();
}
```

[classeComplexeRepInt.C3.cpp]

🛑 Tous ces objets n'ont pas pu être initialisés correctement. En **représentation interne**, on ne peut pas accéder directement aux membres  $\text{Re}$  et  $\text{Im}$  depuis l'extérieur de la classe.



## Classe Complexe en représentation interne V

👉 **Solution** : prévoir une fonction (**public**) d'initialisation

```
public: //fonctions membres publiques
    void affiche();
    Complexe conjugue();
    Complexe addition(const Complexe & z);
    void initialise (double x=0, double y=0) {Re = x; Im = y;}
};
```



|classeComplexeRepInt.C3.cpp|

👉 Exemple d'utilisation à partir de la fonction **main** :

```
z1.initialise(1.,2.); //  $z_1 = 1 + 2i$ 
z1.affiche();
z2 = z1.conjugue(); //  $z_2 = \overline{z_1} = 1 - 2i$ 
z2.affiche();
z4 = z1.addition(z2.conjugue()); //  $z_4 = z_1 + \overline{z_3} = z_1 + \overline{z_2}$ 
z4.affiche();
```

|classeComplexeRepInt.C3.cpp|



# Introduction

 Rappel : par défaut les objets de type `class` désignent des variables automatiques  Absence d'initialisation par défaut de ces objets lors de la déclaration : `Complex z1`

- Que faire ?



# Introduction

-  Rappel : par défaut les objets de type `class` désignent des variables automatiques  Absence d'initialisation par défaut de ces objets lors de la déclaration : `Complex z1`
- Que faire ? une fonction d'initialisation !






# Introduction

- ⊘ Rappel : par défaut les objets de type `class` désignent des variables automatiques ➡ Absence d'initialisation par défaut de ces objets lors de la déclaration : `Complex z1`
- Que faire ? une fonction d'initialisation ! pas très élégant et source d'erreur



# Introduction

-  Rappel : par défaut les objets de type `class` désignent des variables automatiques  Absence d'initialisation par défaut de ces objets lors de la déclaration : `Complex z1`
- Que faire ?
  - Meilleure approche ? permettre au programmeur de déclarer une fonction spéciale  **constructeur**



## Définition

### Définition

Un constructeur est une fonction membre particulière (méthode). Il

- possède le même nom que la classe,
- ne possède pas de valeur de retour (même pas `void`).



# Un constructeur simple I

## 📌 Déclaration et définition :

```
class A
{
    //objet qui ne contient aucune donnée membre
public :
    A(); //déclaration d'un constructeur
};

A::A() //définition du constructeur
{
    cout << "Je suis un constructeur du type A" << endl;
    cout << "Je viens de creer un objet A d'adresse " << this << endl;
}
```

|constructSimple.C3.cpp|

## 📌 Utilisation



## Un constructeur simple II

```
int main()
{
    void f();
    A a;
    A b;
    cout << endl;
    f();
    cout << endl;
    return 0;
}

void f()
{
    cout << "Entree dans la fonction f" << endl;
    A temp;
    cout << "Sortie de la fonction f" << endl;
}
```

/constructSimple.C3.cpp



# Un constructeur simple III

## 👤 Exécution :

```
Je suis un constructeur du type A
Je viens de creer un objet A d adresse 0x7fff9ad6a2bf
Je suis un constructeur du type A
Je viens de creer un objet A d adresse 0x7fff9ad6a2be

Entree dans la fonction f
Je suis un constructeur du type A
Je viens de creer un objet A d adresse 0x7fff9ad6a29f
Sortie de la fonction f
```



# Encapsulation

- Regrouper les données et les fonctions au sein d'une classe
- Association à un système de protection par mots clés :



# Encapsulation

- Regrouper les données et les fonctions au sein d'une classe
- Association à un système de protection par mots clés :
  - ☞ **public** : niveau le plus bas de protection, toutes les données ou fonctions membres d'une classe sont utilisables par toutes les fonctions



# Encapsulation

- Regrouper les données et les fonctions au sein d'une classe
- Association à un système de protection par mots clés :
  - ☞ **public** : niveau le plus bas de protection, toutes les données ou fonctions membres d'une classe sont utilisables par toutes les fonctions
  - ☞ **private** : niveau le plus élevé de protection, données (ou fonctions membres) d'une classe utilisables uniquement par les fonctions membre de la même classe



# Encapsulation

- Regrouper les données et les fonctions au sein d'une classe
- Association à un système de protection par mots clés :
  - ☞ **public** : niveau le plus bas de protection, toutes les données ou fonctions membres d'une classe sont utilisables par toutes les fonctions
  - ☞ **private** : niveau le plus élevé de protection, données (ou fonctions membres) d'une classe utilisables uniquement par les fonctions membre de la même classe
  - ☞ **protected** : comme **private** avec extension aux classes dérivées (voir héritage)



## Classe Cmpx. Cahier des charges

⇒ Réécriture d'une nouvelle classe modélisant les nombres complexes répondant à ces exigences :



## Classe Cmpx. Cahier des charges

⇒ Réécriture d'une nouvelle classe modélisant les nombres complexes répondant à ces exigences :

- ☞ une représentation interne (données privées représentant le complexe) sous forme polaire : **Module**  $\geq 0$  et **Phase**  $\in [-\pi, \pi]$



## Classe Cmpx. Cahier des charges

⇒ Réécriture d'une nouvelle classe modélisant les nombres complexes répondant à ces exigences :

- ☞ une représentation interne (données privées représentant le complexe) sous forme polaire : **Module**  $\geq 0$  et **Phase**  $\in [-\pi, \pi]$
- ☞ initialisation sous la forme polaire ou cartésienne





## Classe Cmpx. Cahier des charges

⇒ Réécriture d'une nouvelle classe modélisant les nombres complexes répondant à ces exigences :

- ☞ une représentation interne (données privées représentant le complexe) sous forme polaire : **Module**  $\geq 0$  et **Phase**  $\in [-\pi, \pi]$
- ☞ initialisation sous la forme polaire ou cartésienne
- ☞ une initialisation par défaut



## Classe Cmpx. Cahier des charges

⇒ Réécriture d'une nouvelle classe modélisant les nombres complexes répondant à ces exigences :

- ☞ une représentation interne (données privées représentant le complexe) sous forme polaire : **Module**  $\geq 0$  et **Phase**  $\in [-\pi, \pi]$
- ☞ initialisation sous la forme polaire ou cartésienne
- ☞ une initialisation par défaut
- ☞ accès au module et à la phase



## Classe Cmpx. Cahier des charges

⇒ Réécriture d'une nouvelle classe modélisant les nombres complexes répondant à ces exigences :

- ☞ une représentation interne (données privées représentant le complexe) sous forme polaire : **Module**  $\geq 0$  et **Phase**  $\in [-\pi, \pi]$
- ☞ initialisation sous la forme polaire ou cartésienne
- ☞ une initialisation par défaut
- ☞ accès au module et à la phase
- ☞ affichage sous la forme polaire et la forme cartésienne



## Classe Cmpx. Cahier des charges

⇒ Réécriture d'une nouvelle classe modélisant les nombres complexes répondant à ces exigences :

- ☞ une représentation interne (données privées représentant le complexe) sous forme polaire : **Module**  $\geq 0$  et **Phase**  $\in [-\pi, \pi]$
- ☞ initialisation sous la forme polaire ou cartésienne
- ☞ une initialisation par défaut
- ☞ accès au module et à la phase
- ☞ affichage sous la forme polaire et la forme cartésienne
- ☞ fonctions émulant  $*$ ,  $/$  et l'exponentiation



# Première approche I

## 👉 Déclaration de la classe

```
class Cmpx
{
private :
    double Module;
    double Phase;
public :
    // Constructeur
    Cmpx(double rho = 0, double theta = 0);
    // Fonctions membres
    void affiche();
    Cmpx Multiplication(const Cmpx &);
    Cmpx Division(const Cmpx &);
    Cmpx Exponentiation(const double & exposant);
};
```

|cmpx.C3.cpp|



## Première approche II

### Définition du constructeur

```
Cmpx::Cmpx(double rho, double theta)
{
    Module = rho;
    Phase = theta;
}
```

|cmpx.C3.cpp|

### La fonction exponentiation

```
Cmpx Cmpx::Exponentiation(const double & exposant)
{
    return Cmpx(pow(Module, exposant), exposant * Phase);
}
```

|cmpx.C3.cpp|

### La fonction division



## Première approche III

```
Cmpx Cmpx::Division(const Cmpx & z)
{
    if(!z.Module )
    {
        cout << "Division par le complexe 0" << endl;
        exit(1);
    }
    double x = Module / z.Module;
    double y = Phase - z.Phase;
    Cmpx result(x,y);
    return result;
}
```

|cmpx.C3.cpp|

### La fonction multiplication



## Première approche IV

```
Cmpx Cmpx::Multiplication(const Cmpx & z)
{
    return Cmpx(Module * z.Module, Phase + z.Phase);
}
```

*/cmpx.C3.cpp*

### La fonction affiche

```
void Cmpx::affiche()
{
    cout << "(" << Module << ", " << Phase << ")" << endl;
}
```

*/cmpx.C3.cpp*

### Utilisation





## Première approche V

```
int main()
{
    const double Pi = atan(1.)*4.;

    Cmpx z1(2,Pi/4.); //  $z_1 = \sqrt{2} + i\sqrt{2}$ 
    cout << "z1 = " ; z1.affiche();

    Cmpx z;
    z = 2.; //  $z = 2e^{0i}$  ?,  $z = 2e^{2i}$  ?,  $z = 0e^{2i}$  ?
    cout << "z = " ; z.affiche();

    Cmpx z3 = z1.Multiplication(z); //  $z_3 = z_1 z$ 
    cout << "Resultat de la multiplication :";
    cout << "z3 = " ; z3.affiche();

    z3 = z1.Multiplication(2); // Conversion de 2 en Cmpx?
    cout << "Le meme produit avec un appel different :";
    cout << "z3 = " ; z3.affiche();
}
```

## Première approche VI

```
Cmpx z4(2);  
cout << "z4 = " ; z4.affiche();  
    (z4.Exponentiation(0.5)).affiche();  
  
Cmpx z5(-2);  
cout << "z5 = " ; z5.affiche();  
    (z5.Exponentiation(0.5)).affiche();  
  
Cmpx z6;  
cout << "z6 = " ; z6.affiche();  
    (z6.Exponentiation(-0.5)).affiche();  
  
return 0;  
}
```

*/cmpx.C3.cpp*



## Première approche VII


### ⊕ Exécution :

```
z1 = (2, 0.785398)
z = (2, 0)
Resultat de la multiplication :z3 = (4, 0.785398)
Le meme produit avec un appel different :z3 = (4, 0.785398)
z4 = (2, 0)
(1.41421, 0)
z5 = (-2, 0)
(nan, 0)
z6 = (0, 0)
(inf, -0)
```

- Interdire la conversion implicite (`Cmpx z=2`)
- Pourquoi `nan` et `inf` ? Enrichir les fonctions membres pour vérifier la validité des données



## Mot clé `explicit` I

 Le mot-clé `explicit` interdit la conversion implicite. Le constructeur `Cmpx` ne sera appelé qu'explicitement

 On modifie la déclaration du constructeur :

```
class Cmpx
{
private :
    double Module;
    double Phase;
public :
    // Constructeur
    explicit Cmpx(double rho = 0, double theta = 0);
    // Fonctions membres
    void affiche();
    Cmpx Multiplication(const Cmpx &);
    Cmpx Division(const Cmpx &);
    Cmpx Exponentiation(const double & exposant);
};
```

|cmpxExplicit.C3. |pp|

## Mot clé `explicit` II

✗ Les instructions des lignes suivantes deviennent illégales

```
1 z = 2.; // z = 2e0i ?, z = 2e2i ?, z = 0e2i ?
```

```
1 z3 = z1.Multiplication(2); // Conversion de 2 en Cmpx?
```

|cmpx.C3.cpp|

✓ Par contre, on peut toujours demander **explicitement** au constructeur d'effectuer une telle conversion

```
z = Cmpx(2.); // z = 2 (conv. explicit, Phase = 0 par défaut)
```

```
z3 = z1.Multiplication(Cmpx(2)); // Conversion explicit de 2
```

|cmpxExplicit.C3.cpp|



# Constructeur et validation des données I

Réécriture du constructeur avec :

- une liste d'initialisation
- Validation des données

```
Cmpx::Cmpx(double rho, double theta)
  :Module(rho), Phase(theta)
{
  if(Module < 0)
  {
    cout << "Conctructeur invoque avec un module < 0" << endl;
    exit(1);
  }
}
```

*|cmpxExplicit.C3.cpp|*

 **Exécution :**



## Constructeur et validation des données II

```
z1 = (2, 0.785398)
z = (2, 0)
Resultat de la multiplication :z3 = (4, 0.785398)
Le meme produit avec un appel different :z3 = (4, 0.785398)
z4 = (2, 0)
(1.41421, 0)
Conctructeur invoque avec un module < 0
```

Rappel des résultats de l'ancienne version :

```
...
z4 = (2, 0)
(1.41421, 0)
z5 = (-2, 0)
(nan, 0)
z6 = (0, 0)
(inf, -0)
```



## Autres améliorations pour respecter le cahier des charges I

### 📌 La fonction exponentiation

```
Cmpx Cmpx::Exponentiation(const double & exposant)
{
    if( (exposant < 0) && (Module == 0) )
    {
        cout << "Elevation de 0 a une puissance negative " << endl;
        exit(1);
    }
    return Cmpx(pow(Module, exposant), exposant * Phase);
}
```

*/cmpxExplicit.C3.cpp*

### 📌 ou plus « rigoureusement » :

```
...
static const double epsilon_double = (10., -16);
if( (exposant < 0) && (abs(Module) <= epsilon_double) )
...

```





# Plan

- 8 **Classe Cmpx. Suite**
- 9 **Résumé. Fonctions amies**
- 10 **Résumé. Accesseurs**
- 11 **Résumé. Surcharge d'opérateurs**
- 12 **Un peu plus loin avec la classe Cmpx**
- 13 **Destructeur**
- 14 **Héritage**
  - Une première classe dérivée. Rectangle



## Modification du cahier des charges

➡ Écriture d'une classe modélisant les nombres complexes répondant à ces exigences :

- 1 une représentation interne (données privées représentant le complexe) sous forme polaire : **Module**  $\geq 0$  et **Phase**  $\in [-\pi, \pi[$
- 2 initialisation sous la forme polaire ou cartésienne
- 3 une initialisation par défaut (**sous la forme cartésienne**)
- 4 accès au module et à la phase
- 5 affichage sous la forme polaire et la forme cartésienne
- 6 fonctions émulant  $*$ ,  $/$  et l'exponentiation  
**Surcharge des opérateurs +, \*, / et une fonction émulant l'exponentiation**



# Implémentation I

- 1 On garde la même représentation interne sous la forme polaire

```
class Cmpx
{
private : // Données membres privées
    double Module;
    double Phase;
```

On ajoute une méthode privée `Angle` pour ramener la **Phase** dans l'intervalle  $[-\pi, \pi[$  :

📄 Sa déclaration :

```
// Méthodes privées
private :
    double Angle(double);
```

|cmpx.C4.hpp

📄 Sa définition :



# Implémentation II

```

double Cmpx::Angle(double phi)
{ /*
   La fonction : double modf(double d, double * integral)
   disponible dans <cmath> retourne la partie fractionnaire de d
   et stocke la partie intégrale de d dans *integral
   */
  const double Pi = 4.*atan(1.);
  double a = 0;
  double b = modf(phi/(2*Pi), &a); // On ramène  $\varphi$  dans  $[-2\pi, 2\pi]$ 
  if(b >= 0.5) b--; // puis
  if(b < -0.5) b++; // dans
  return b*2*Pi; //  $[-\pi, \pi[$ 
  // Exemples :  $\frac{3\pi}{2} \rightarrow -\frac{\pi}{2}$ ,  $2\pi \rightarrow 0$  et  $\pi \rightarrow -\pi$ 
}

```

|cmpx.C4.cpp|



## Implémentation III

- 2 Ajout de deux données membre publiques de type `Representation` permettant de choisir entre les deux représentations `Polaire` et `Cartesienne`. Le nouveau type `Representation` est défini par le mot-clé `enum` :

```
// Données et fonctions membres publiques
public :
    enum Representation { Cartesienne, Polaire };
```

|cmpx.C4.hpp|

### Remarque

La déclaration précédente signifie que `Representation` est une constante entière appartenant à l'ensemble  $\{0, 1\}$ . Grâce au mot-clé `enum`, le compilateur affecte automatiquement la valeur 0 à `Cartesienne` et 1 à `Polaire`



## Implémentation IV

✍ Écriture d'un constructeur permettant de traiter les deux représentations cartésienne et polaire. Ils seront distingués par un troisième argument de type `Représentations`.

```
// Constructeur en représentation cartésienne et polaire
Cmpx(double a=0, double b=0, Representation rep=Cartesienne);
```

*|cmpx.C4.hpp|*

Ce constructeur doit être adapté à la représentation choisie pour l'initialisation, **mais il doit toujours conduire à la définition d'un Module et d'une Phase**

✍ Définition du constructeur :



# Implémentation V

```

Cmpx::Cmpx(double a, double b, Representation rep)
{
    if(rep == Cartesienne) // x = a et y = b
    {
        const double Pi = 4.*atan(1.);
        Module = sqrt(a*a + b*b);
        Phase = Angle(a ? atan(b/a) : (b > 0 ? Pi/2.: -Pi/2.));
    }
    else // Module = a et Phase = b
    {
        if(a < 0)
        {
            cout << "Concteur invoque avec un module < 0" << endl;
            exit(1);
        }
        Module = a;
        Phase = Angle(b);
    }
}

```

## Implémentation VI

```
}

```

|cmpx.C4.cpp|

### 3 Initialisation par défaut sous la forme cartésienne

🔗 Rappel de la déclaration du constructeur :

```
Cmpx(double a=0, double b=0, Representation rep=Cartesienne);

```

|cmpx.C4.hpp|

### 4 Accès au module et à la phase : ne peut se faire à l'aide de `z.Module` et `z.Phase` quand nous sommes à l'extérieure de la classe (depuis le `main` exemple). Solution ?

Accesseurs en lecture sous la forme de méthodes publiques :

🔗 Déclaration dans l'interface publique :

```
// Accesseurs en lecture au Module et à la Phase
double GetModule();
double GetPhase();

```



## Implémentation VII

|cmpx.C4.hpp|

📄 Définition :

```
double Cmpx::GetModule()
{
    return Module;
}

double Cmpx::GetPhase()
{
    return Phase;
}
```

|cmpx.C4.cpp|

### 5 Affichage sous la forme polaire et cartésienne :

📄 Déclaration de la méthode affiche :

```
// Fonctions membres
void affiche(Representation rep = Cartesienne);
```



## Implémentation VIII

|cmpx.C4.hpp|

Pour implémenter la méthode `affiche` nous avons besoin d'accéder à la partie réelle et à la partie imaginaire de nos nombres complexes ➡ on écrit des accesseurs qui les calculent. On choisit de les déclarer comme **méthodes privées** :

```
// Accesseurs privés pour accéder à la rep. cartésienne
double GetX();
double GetY();
```

|cmpx.C4.hpp|

 Leurs définitions :



# Implémentation IX

```
double Cmpx::GetX()
{
    return Module*cos(Phase);
}

double Cmpx::GetY()
{
    return Module*sin(Phase);
}
```

|cmpx.C4.cpp|

 Retour à la méthode affiche. Sa Définition :



# Implémentation X

```
void Cmpx::affiche(Representation rep)
{
    switch(rep)
    {
        case Cartesienne :
            cout << this->GetX();
            if(this->GetY())
            {
                if(this->GetY() > 0) cout << " + i ";
                else if(this->GetY()<0) cout << " - i ";
                cout << abs(this->GetY());
            }
            cout << endl;
            break;
        case Polaire :
            cout << " " << Module << " *exp(i*" << Angle(Phase) << ")" << endl;
            break;
    }
}
```



# Implémentation XI

|cmpx.C4.cpp|

## 6 Surcharge des opérateurs utiles (\*,-,/,...) versus fonctions

Multiplication, soustraction, Division, ...

On préférerait écrire par exemple `Cmpx z3 = z1*z2;` au lieu de

`Cmpx z3 = z1.Multiplication(z2);` **OU**

`Cmpx z3 = Multiplication(z1, z2);`

📄 Déclaration des surcharges d'opérateurs :

```
public : // Surcharges d'opérateurs
        // en tant que fonctions amies
    friend Cmpx operator- (Cmpx& z1, Cmpx& z2); //  $Z_1 - Z_2$ 
    friend Cmpx operator* (Cmpx& z1, Cmpx& z2); //  $Z_1 Z_2$ 
    friend Cmpx operator/ (Cmpx& z1, Cmpx& z2); //  $\frac{Z_1}{Z_2}$ 

        // ou en tant que méthodes publiques
    Cmpx operator+ (Cmpx& z); //  $*this+Z$ 
```

## Implémentation XII

|cmpx.C4.hpp|

### Définition

Une fonction amie (**friend**) d'une classe est une fonction à laquelle on donne un droit d'accès aux données privées de ladite classe

📄 Implémentation des surcharges :

```
Cmpx operator- (Cmpx& z1, Cmpx& z2) // Fonction amie
{
    return Cmpx(z1.GetX()-z2.GetX(), z1.GetY()-z2.GetY(), Cmpx::Cartesienne)
}
```

```
Cmpx operator* (Cmpx& z1, Cmpx& z2) // Fonction amie
{
    return Cmpx(z1.Module*z2.Module, z1.Phase+z2.Phase, Cmpx::Polaire);
}
```



## Implémentation XIII

```

Cmpx operator/ (Cmpx& z1, Cmpx& z2) // Fonction amie
{
    if(!z2.Module )
    {
        cout << "Division par le complexe 0" << endl;
        exit(1);
    }
    double a = z1.Module / z2.Module;
    double b = z1.Phase - z2.Phase;
    return Cmpx(a , b, Cmpx::Polaire);
}

```

```

Cmpx Cmpx::operator+ (Cmpx& z) // Fonction membre
{
    return Cmpx(this->GetX()+z.GetX(),this->GetY()+z.GetY(),Cmpx::Cartesien);
}

```



# Implémentation XIV

|cmpx.C4.cpp|

📄 Exemple d'utilisation de la classe :

```
const double Pi = atan(1.)*4.;
```

```
Cmpx z1(2,Pi/4.,Cmpx::Polaire); //  $z_1 = \sqrt{2} + i\sqrt{2}$ 
```

```
cout << "Afficaha par default : z1 = " ; z1.affiche();
```

```
cout << "Affichage polaire : z1 = " ; z1.affiche(Cmpx::Polaire);
```

```
cout << "Affichage cartesien : z1 = " ; z1.affiche(Cmpx::Cartesienne);
```

|mainCmpx.C4.cpp|

🏃 Exécution :

```
Afficaha par default : z1 = 1.41421 + i 1.41421
```

```
Affichage polaire : z1 = 2*exp(i*0.785398)
```

```
Affichage cartesien : z1 = 1.41421 + i 1.41421
```





# Implémentation XV

```
Cmpx i(0,1,Cmpx::Cartesienne); //  $i = i$ 
cout << "i= "; i.affiche();
cout << "i= "; i.affiche(Cmpx::Polaire);
```

|mainCmpx.C4.cpp|



```
i= 6.12323e-17 + i 1
i= 1*exp(i*1.5708)
```



```
Cmpx z2=2; //  $z_2 = 2$ 
cout << "z2 = " ; z2.affiche(Cmpx::Polaire);
cout << "z2 = " ; z2.affiche(Cmpx::Cartesienne);
```

|mainCmpx.C4.cpp|



# Implémentation XVI

```
z2 = 2*exp(i*0)
z2 = 2
```



```
Cmpx z3 = z1*z2; //  $z_3 = z_1 z_2 = 2z_1 = 2\sqrt{2}(1 + i)$ 
cout << "Resultat de la multiplication :";
cout << "z3 = z1*z2 = " ; z3.affiche(Cmpx::Polaire); cout << endl;
```

|mainCmpx.C4.cpp|



```
Resultat de la multiplication :z3 = z1*z2 = 4*exp(i*0.785398)
```



# Implémentation XVII

```

Cmpx z4 = z3/i; //z4 = 2√2(1 - i)
cout << "z4 = z3/i = " ; z4.affiche();
cout << "z4 = " ; z4.affiche(Cmpx::Polaire);
cout << "z4^0.5 = ";
(z4.Exponentiation(0.5)).affiche(Cmpx::Polaire); cout << endl;

```

|mainCmpx.C4.cpp|



```

z4 = z3/i = 2.82843 - i 2.82843
z4 = 4*exp(i*-0.785398)
z4^0.5 = 2*exp(i*-0.392699)

```



## Récapitulons. Mot-clé `friend` (fonctions amies)

### Fonctions membres. Rappel

Une fonction déclarée membre d'une classe

- 1 peut accéder à sa partie privée
- 2 se trouve dans sa portée
- 3 doit être appelée sur un objet (possède un pointeur `this`)

### Fonctions amies. Définition

Une fonction est déclarée amie d'une classe pour qu'elle puisse accéder à sa partie privée (propriété numéro 1 des fonctions membres).

- Sa déclaration peut être, indifféremment, dans la partie privée ou publique de la classe
- Elle fait partie de l'interface au même titre qu'une fonction membre

## Accesseurs en lecture et en écriture des données membres privées I

- Accès en lecture à des données membres privées depuis l'extérieur de la classe
- Éventuellement en écriture !

### Accesseur en lecture. Exemple

```
double Cmpx::GetPhase()  
{  
    return Phase;  
}
```

|cmpx.C4.cpp|

On peut aussi envisager l'implémentation d'accesseurs en écriture :



## Accesseurs en lecture et en écriture des données membres privées II

### Accesseur en écriture. Exemple

```
void Cmpx::SetModule(double rho)
{
    if(rho<0)
    {
        cout << " Module negatif en argument de SetModule " << endl;
        exit(1);
    }
    Module = rho;
}
```



# Surcharge d'opérateurs I

Possibilité de surcharger les opérateurs existants pour les types simples en étendant leurs définitions aux types utilisateur.

## Exemple. Surcharge en tant que fonction amie

```
Cmpx operator* (Cmpx& z1, Cmpx& z2) // Fonction amie
{
    return Cmpx(z1.Module+z2.Module,z1.Phase+z2.Phase,Cmpx::Polaire);
}
```

|cmpx.C4.cpp|



## Surcharge d'opérateurs II

### Exemple. Surcharge en tant que méthode

```
Cmpx Cmpx::operator+ (Cmpx& z) // Fonction membre
{
    return Cmpx(this->GetX()+z.GetX(),this->GetY()+z.GetY(),Cmpx::Cartesienne);
}
```

|cmpx.C4.cpp|





Argument implicite et mot-clé `const` I

Ajoutons le mot-clé `const` devant la déclaration du nombre complexe  $i$  :

```

1  const Cmpx i(0,1,Cmpx::Cartesienne); //  $i = i$ 
2  cout << "i= "; i.affiche();
3  cout << "i= "; i.affiche(Cmpx::Polaire);
4
5  Cmpx z2=2; //  $z_2 = 2$ 
6  cout << "z2 = " ; z2.affiche(Cmpx::Polaire);
7  cout << "z2 = " ; z2.affiche(Cmpx::Cartesienne);
8
9  Cmpx z3 = z1*z2; //  $z_3 = z_1 z_2 = 2z_1 = 2\sqrt{2}(1 + i)$ 
10 cout << "Resultat de la multiplication :";
11 cout << "z3 = z1*z2 = " ; z3.affiche(Cmpx::Polaire); cout << endl;
12
13 Cmpx z4 = z3/i; //  $z_4 = 2\sqrt{2}(1 - i)$ 

```

|mainCmpx.C4.const.cpp

## Argument implicite et mot-clé `const` II

### Compilation du `main` :

```
mainCmpx.C4.const.cpp: In function 'int main()':
mainCmpx.C4.const.cpp:16: error: passing 'const Cmpx' as 'this' argument
of 'void Cmpx::affiche(Cmpx::Representation)' discards qualifiers
mainCmpx.C4.const.cpp:17: error: passing 'const Cmpx' as 'this' argument
of 'void Cmpx::affiche(Cmpx::Representation)' discards qualifiers
mainCmpx.C4.const.cpp:27: error: no match for 'operator/' in 'z3 / i'
cmpx.C4.const.hpp:33: note: candidates are: Cmpx operator/(Cmpx&, Cmpx&)
```

- Il faut modifier la fonction `affiche` pour qu'elle puisse être appelée avec un argument implicite constant



```
void affiche(Representation rep = Cartesienne) const;
```

|cmpx.C4.const.hpp



Argument implicite et mot-clé `const` III

```

void Cmpx::affiche(Representation rep) const
{
    switch(rep)
    {
        case Cartesienne :
            cout << this->GetX();
            if(this->GetY())
            {
                if(this->GetY() > 0) cout << " + i ";
                else if(this->GetY()<0) cout << " - i ";
                cout << abs(this->GetY());
            }
            cout << endl;
            break;
        case Polaire :
            cout << " " << Module << " *exp(i*" << Angle(Phase) << ")" << endl;
            break;
    }
}

```

Argument implicite et mot-clé `const` IV

}

|cmpx.C4.const.cpp|

- Il faut aussi modifier la surcharge de l'opérateur /



```
friend Cmpx operator/ (const Cmpx& z1, const Cmpx& z2); //  $\frac{z_1}{z_2}$ 
```

|cmpx.C4.const.hpp|

```
Cmpx operator/ (const Cmpx& z1, const Cmpx& z2) // Fonction amie
{
    if(!z2.Module )
    {
        cout << "Division par le complexe 0" << endl;
        exit(1);
    }
    double a = z1.Module / z2.Module;
    double b = z1.Phase - z2.Phase;
    return Cmpx(a , b, Cmpx::Polaire);
}
```

Argument implicite et mot-clé `const` V

}

/cmpx.C4.const.cpp/

 Compilation de `cmpx.C4.cpp` :

```

cmpx.C4.const.cpp: In member function
'void Cmpx::affiche(Cmpx::Representation) const':
cmpx.C4.const.cpp:56: error: passing 'const Cmpx' as 'this' argument
of 'double Cmpx::GetX()' discards qualifiers
cmpx.C4.const.cpp:57: error: passing 'const Cmpx' as 'this' argument
of 'double Cmpx::GetY()' discards qualifiers
cmpx.C4.const.cpp:59: error: passing 'const Cmpx' as 'this' argument
of 'double Cmpx::GetY()' discards qualifiers
cmpx.C4.const.cpp:60: error: passing 'const Cmpx' as 'this' argument
of 'double Cmpx::GetY()' discards qualifiers
cmpx.C4.const.cpp:61: error: passing 'const Cmpx' as 'this' argument
of 'double Cmpx::GetY()' discards qualifiers
cmpx.C4.const.cpp:66: error: passing 'const Cmpx' as 'this' argument
of 'double Cmpx::Angle(double)' discards qualifiers

```



Argument implicite et mot-clé `const` VI

```

1 void Cmpx::affiche(Representation rep) const
2 {
3     switch(rep)
4     {
5         case Cartesienne :
6             cout << this->GetX();
7             if(this->GetY())
8                 {
9                     if(this->GetY() > 0) cout << " + i ";
10                    else if(this->GetY()<0) cout << " - i ";
11                    cout << abs(this->GetY());
12                }
13            cout << endl;
14            break;
15        case Polaire :
16            cout << " " << Module << "*exp(i*" << Angle(Phase) << ")" << endl;

```

|cmpx.C4.const.cpp

## Argument implicite et mot-clé `const` VII

- De la même manière, il faut modifier `GetX`, `GetY` et `Angle`



# Destructeurs

## Définition

tout comme les constructeurs, un destructeur est une fonction membre particulière. Elle

- possède le même nom que la classe précédé de  $\sim$
- ne possède pas de valeur de retour (même pas `void`)

## Exemple. Le destructeur par défaut

```
A::~~A()  
{  
}
```

Le destructeur par défaut ne suffit toujours pas !





## Destructeur et allocation dynamique I

- ✍ **Exemple** : une classe modélisant un tableau dynamique et munie d'un pointeur statique de sauvegarde !

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

class Vecteur
{
public :
    int n;
    double* ptab;
    static double* psauv;
public :
    Vecteur(int, double*); // constructeur
};

double* Vecteur::psauv = 0; //initialisation du membre statique
```

## Destructeur et allocation dynamique II

```
Vecteur::Vecteur(int m, double* tabl)
{
    n = m;
    ptab = new double[n];
    for(int i=0; i<n; i++)
        ptab[i] = tabl[i];
    psauv = ptab;
}

int main()
{
    double taba[4]={1.1,1.2,1.3,1.4};
    {
        Vecteur v3(4,taba);
    } // Destruction de v3
    for (int i=0; i<4; i++)
        cout << *(Vecteur::psauv+i) << endl;
    return 0;
}
```

# Destructeur et allocation dynamique III

*/sansDestructeur.C4.cpp*

## Exécution :

```
1.1  
1.2  
1.3  
1.4
```

## Après l'ajout d'un destructeur correctement écrit :

```
Vecteur::~Vecteur()  
{  
    delete[] ptab;  
    psauv = NULL;  
}
```

*/sansDestructeur.2.C4.cpp*

## Destructeur et allocation dynamique IV

### 🌀 Exécution :

```
*** glibc detected *** ./a.out: double free or corruption (fasttop) :  
0x0000000000714010 ***  
===== Backtrace: =====  
/lib/libc.so.6[0x7f1ee6004948]  
/lib/libc.so.6(cfree+0x76) [0x7f1ee6006a56]  
./a.out(__gxx_personality_v0+0x180) [0x400970]  
./a.out(__gxx_personality_v0+0x29d) [0x400a8d]  
/lib/libc.so.6(__libc_start_main+0xe6) [0x7f1ee5faf1a6]  
./a.out(__gxx_personality_v0+0x39) [0x400829]
```



# Classe Polygone I

## Définition

On caractérise un polygone par une liste ordonnée de points du plan

✍ On commence par écrire une classe point

```
class Point
{
public :
    Point (double x=0, double y=0);
    double GetX() const;
    double GetY() const;
    double Distance (const Point & p) const;

    friend ostream & operator<< (ostream &, const Point &);
    friend Point operator+ (const Point &, const Point &);

private :
```

## Classe Polygone II

```
double X;  
double Y;  
};
```

*/classePoint.C4.hpp*

### Sa définition :

```
Point::Point (double x, double y)  
  : X(x),Y(y)  
{  
  
double Point::GetX() const  
{  
  return X;  
}  
  
double Point::GetY() const  
{  
  return Y;  
}
```

## Classe Polygone III

```
}  
  
double Point::Distance (const Point & p) const  
{  
    return pow((X-p.X)*(X-p.X) + (Y-p.Y)*(Y-p.Y), 0.5);  
}  
  
std::ostream & operator<< (std::ostream & s, const Point & p)  
{  
    return s << "[" << p.X << ", " << p.Y << "];"  
}  
  
Point operator+ (const Point & p1, const Point & p2)  
{  
    return Point(p1.X + p2.X, p1.Y + p2.Y);  
}
```

/classePoint.C4.cpp



## Classe Polygone IV

### La classe Polygone

```
#ifndef POLYGONE_HPP
#define POLYGONE_HPP

#include "classePoint.C4.hpp"

class Polygone
{
private :
    Point * TabPoints;
    int NbPoints;
protected :
    void Dimensionner(int, const Point *);
public :
    Polygone(int);
    Polygone(const Polygone &);
    ~Polygone();
};
```



# Classe Polygone V

```
Polygone & operator= (const Polygone &);  
int GetNbPoints () const {return NbPoints;}  
Point & operator[] (int);  
Point operator[] (int) const;  
void DecrisToi() const;  
double Surface() const;  
double Perimetre() const;  
};  
  
#endif // POLYGONE_HPP
```

|classePolygone.C4.hpp|

## Son implémentation

 Une fonction utile : Dimensionner



# Classe Polygone VI

```
void Polygone::Dimensionner(int nbp, const Point * tabp)
{
    if (TabPoints != 0)
    {
        delete [] TabPoints;
        TabPoints = 0;
    }
    if (nbp <= 0) // en cas d'erreur
    {
        NbPoints = 0;
        return;
    }
    TabPoints = new Point [NbPoints = nbp];
    for (int i=0; i<NbPoints; i++) TabPoints[i] = tabp[i];
}
```

|classePolygone.C4.cpp|

 Deux constructeurs :



## Classe Polygone VII

```
Polygone::Polygone(int nbp) : NbPoints(0), TabPoints(0)
{
    if (nbp > 0) //sinon Polygone vide
    {
        Point tabp[nbp]; //par défaut ne contient que (0,0)
        Dimensionner(nbp, tabp);
    }
}

Polygone::Polygone(const Polygone & p)
{
    Dimensionner(p.NbPoints, p.TabPoints);
}
```

|classePolygone.C4.cpp|

 Un destructeur :



## Classe Polygone VIII

```
Polygone::~~Polygone()  
{  
    delete[] TabPoints;  
}
```

*/classePolygone.C4.cpp/*



## Classe Rectangle

### Rectangle

- Un nombre fixe de sommets : 4
- Une largeur
- Une longueur

### Conception

Nous définissons un rectangle par la donnée d'une origine, d'une longueur et d'une largeur

### Modèle d'héritage

```
class B : specificateur A
```

specificateur = public, private **OU** protected



## Déclaration de la classe Rectangle I

```
#ifndef RECTANGLE_HPP
# define RECTANGLE_HPP

#include "classePolygone.C4.hpp"

class Rectangle : public Polygone
{
    // Aucune donnée membre en plus de celles héritées de Polygone
    // Méthodes d'implémentation propres à la classe Rectangle
    double Dx() const; // Taille du rectangle selon x
    double Dy() const; // Taille du rectangle selon y
public :
    // Constructeur spécialisé
    Rectangle(const Point & origine = Point(0,0),
              double dx = 0, double dy = 0);
    // Destructeur
    ~Rectangle();
    // Fonctions membres spécialisées

```

## Déclaration de la classe Rectangle II

```
void DecrisToi() const;
double Perimetre() const;
double Surface() const;
// Nouvelles fonctions membres
double Largeur() const;
double Longueur() const;
};

#endif // RECTANGLE_HPP
```

|Rectangle.C4.hpp|

