

2022-2023, Semestre d'automne
L3, Licence Sciences et Technologies
Université Lyon 1

LIFAPC: Algorithmique, Programmation et Complexité

Chaîne Raphaëlle (responsable semestre automne)
E-mail : raphaelle.chaine@iris.cnrs.fr
<http://iris.cnrs.fr/membres?idn=rchaine>

1

1

Organisation de cet enseignement

- Répartition sur 10 semaines
 - 1h30 de cours
 - 1h30 de TD
 - 3h de TP
- Auxquels sont ajoutés des créneaux correspondant à des séances de rendus de TP

2

2

Evaluation des connaissances

- Contrôle continu intégral
 - Assiduité en TD/TP
 - Petites interrogations (questions de cours, ou exercices fait en TD) (2 ou 3)
 - Evaluation de certains TPs (2 ou 3)
 - Contrôle en amphi (à la fin de l'UE)
- Enseignants référents
- Tuteurs

3

3

Bibliographie Algorithmique

- **Introduction à l'algorithmique**,
T. Cormen, C. Leiserson, R. Rivest, Dunod
- **Structures de données et algorithmes**,
A. Aho, J. Hopcroft, J. Ullman, InterEditions
- **Types de données et algorithmes**,
C. Froideveaux, M.C. Gaudel, M. Soria, Ediscience
- **The Art of Computer Programming**,
D.Knuth, Vol. 1 et 3, Addison-Wesley
- **Algorithms**, Sedgewick, Addison-Wesley
- **Algorithmes de graphes**, C. Prins, Eyrolles

4

4

Bibliographie C

- **The C Programming Language (Ansi C)**
B.W. Kernighan - D.M. Ritchie
Ed. Prentice Hall, 1988
- **Le langage C - C ANSI**
B.W. Kernighan - D.M. Ritchie
Ed. Masson - Prentice Hall, 1994
- **Langage C - Manuel de référence**
Harbison S.P., Steele Jr. G.L.
Masson, 1990
(traduction en français par J.C. Franchitti)
- **Méthodologie de la programmation en langage C, Principes et applications**
Braquelaire J.P.
Masson, 1995
- N'oubliez jamais de vous servir du **man!**

5

5

Bibliographie C++

- **C++ Primer**, Lippman & Lajoie,
Third Edition, Addison-Wesley, 1998
- **L'essentiel du C++**, Lippman & Lajoie, Vuibert
- **Le langage et la bibliothèque C++, Norme ISO**
Henri Garetta, Ellipse, 2000
- **The C++ Programming Language**,
Bjarne Stroustrup, 3. Auflage, Addison-Wesley,
1997,
(existe en version française)

6

6

Objectifs

- Méthodes de conception pour la résolution de problèmes
 - Types Abstraits
 - Collection ou Ensemble, Séquence ou Liste (triée ou non), Tableau, File, Pile, File de priorité, Table, Arbres, Graphes
- Structures de données
- Complexité des algorithmes
 - Efficacité asymptotique : temps de calcul, espace nécessaire, empreinte mémoire
 - Définitions
 - Outils théoriques d'analyse de la complexité
- Notion de preuve
- Réalisation, impl(éme)(a)ntation

7

7

Prérequis

- LIFAP1, LIFASR3, LIFAP2, LIFAP3, (et LIFAP4)
- Gestion de la mémoire
 - Organisation en **pile** des variables d'un programme
 - Allocation dynamique dans le **tas** (C++ : `new/delete`, C : `malloc/free`)
- Différents modes de passage des paramètres d'une procédure
 - Paramètres **NON MODIFIABLES** : Donnée,
 - Paramètres **MODIFIABLES** : donnée-résultat, résultat
 - Donnée : la procédure travaille sur une copie (*passage par valeur*) ou sur l'original en s'engageant à ne pas le modifier (*const référence*)
 - Donnée-Résultat : la procédure travaille sur l'original (*passage par adresse ou bien par référence*)
- Notion de pointeur et de référence
- Arithmétique des pointeurs
- Type Abstrait et Programmation Modulaire (.hpp .cpp)
- Nuance entre définition et déclaration

8

8

Retour sur la PILE et le TAS

9

9

Prérequis C/C++

- Savoir identifier le positionnement en mémoire des éléments du programme et leur durée de vie
- Pile et Tas
- Pile :
 - Lieu où sont allouées (par **empilement**) les **variables** du programme (ex : `int ma_variable_i;`)
Dépilement à la fin du {bloc} contenant la définition
Attention : Cela concerne aussi les arguments des procédures et fonctions!
 - Lieu où sont retournées les **valeurs des fonctions**
Dépilement à la fin de l'instruction contenant leur appel

10

10

Définition de fonctions en C++

```
float moyenne(float a, float b)
//Précondition : a et b initialisés avec valeurs valides
//quelconques
//Resultat : retourne la moyenne de a et b
{
    float res;
    res = (a+b)/2;
    return res;
}
```

A l'appel :
`float x=4,d;`
...
`d=moyenne(x,2);`

Question : Que se passe-t-il en mémoire ?

11

11

Evolution de la pile

```
float x=4,d;
...
d=moyenne(x,2);
```

x	4
d	

Attention : il s'agit ici d'un modèle pédagogique de l'évolution de la pile avec une simplification de la mise en œuvre des retours de fonctions!

12

12

Evolution de la pile (suite)

```

float x=4,d;
...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
  float res;
  res = (a+b)/2;
  return res;
}

```

x	4
d	
Appel moyenne(x,2)	
Valeur retour	
Paramètres formels	a 4
	b 2

13

Evolution de la pile (suite)

```

float x=4,d;
...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
  float res;
  res = (a+b)/2;
  return res;
}

```

x	4
d	
Appel moyenne(x,2)	
Valeur retour	
Paramètres formels	a 4
	b 2
res	

14

13

14

Evolution de la pile (suite)

```

float x=4,d;
...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
  float res;
  res = (a+b)/2;
  return res;
}

```

x	4
d	
Appel moyenne	
Valeur retour	
Paramètres formels	a 4
	b 2
res	3

15

15

Evolution de la pile (suite)

```

float x=4,d;
...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
  float res;
  res = (a+b)/2;
  return res;
}

```

x	4
d	
Appel moyenne	
Valeur retour	3
Paramètres formels	a 4
	b 2
res	3

16

16

Evolution de la pile (suite)

```

float x=4,d;
...
d=moyenne(x,2);

Or
float moyenne(float a,
              float b)
{
  float res;
  res = (a+b)/2;
  return res;
}

```

x	4
d	3
Valeur retour	3

17

17

Evolution de la pile (suite)

```

float x=4,d;
...
d=moyenne(x,2);

```

x	4
d	3

18

18

Prérequis C/C++

```
int ff(int a)
{ int b=4;
  return a+b;
}
int main()
{ int i=5;
  { int j = 55;
    j=ff(i);
  }
  i++;
  return 0;
}
```

19

19

Prérequis C/C++

```
int ff(int a) //5 VRff et a empilés à l'appel de ff
{ int b=4; //6 Empilement de b
  return a+b; //7 Initialisation de VRff (valeur retour)
} //8 Dépilement de b et de a
int main() //1 VRmain empilée à l'appel du main
{ int i=5; //2 Empilement de i
  { int j = 55; //3 Empilement de j
    j=ff(i); //4 Appel ff
  } //9 dépilement VRff au ;
  //10 Dépilement de j
  i=i+1; //11
  return 0; //12 Initialisation de VRmain
} //13 Dépilement de i
```

20

20

Prérequis C/C++

- Tas :
 - Lieu où des objets sont alloués par l'allocateur mémoire new ou malloc qui recherche un espace disponible et renvoie l'adresse correspondante
 - L'adresse peut/doit être stockée dans un pointeur
 - La désallocation doit être faite par delete ou free à qui on communique l'adresse de l'objet à désallouer

21

21

Prérequis C/C++

```
int main() //1 Empilement de VRmain
{ int i=5; //2 Empilement de i
  { int* pi; //3 Empilement de pi (de type pointeur)
    pi=new int; //4 Allocation dynamique de mémoire
                // à une adresse mémorisée dans pi
    *pi=i+5; // *pi est l'espace de type int alloué
    i=*pi+1; // dans le tas
    delete pi; //5 Désallocation de l'espace pointé
  } //6 Dépilement de pi
  i++;
  return 0; //7 Initialisation de VRmain
}
```

22

22

Pointeurs et adressage en C/C++

- **Opérateur &**
Accès à la valeur de l'adresse d'une variable
ex : int a;
 &a renvoie la valeur de l'adresse de a
 &a peut être stockée dans une variable de type
int *

Attention : Ne pas confondre avec le & permettant
de définir des références en C++

23

23

Retour sur les pointeurs et les références

24

24

- **Définition de variables de types pointeur**
Nom_Type* nom_variable;

La variable pointée par **nom_variable** est de type **Nom_Type**

Ex : **int*** pa;

pa destinée à contenir des adresses de variables de type int

Ex : pa = **&a** ;
// où a est une variable de type int

25

25

- **Opérateur *** (de déréférencement)
Pour accéder à une variable à partir d'une valeur d'adresse

Ex : pa = **&a** ;
//ici *pa désigne donc a

Remarque : *pa identique à ***(&a)**
et identique à a

- Remarque : Pour déréférencer un pointeur, on peut aussi utiliser l'opérateur [] avec argument 0

*pa identique à pa[0] (cf transparents ultérieurs sur les pointeurs et les tableaux C/C++)

26

26

Prérequis C/C++

- Savoir identifier la signification des symboles * et & en tout lieu

NomType* p; //définition d'une variable p
//de type pointeur sur NomType

*p //déréférencement du pointeur p,
//désigne la variable pointée par p

NomType & a=b; //définition d'une référence a
//sur la variable b

&c //valeur de l'adresse de la variable c

- Différence entre (type) pointeur (LIFAP3) et (pseudo-type) référence (LIFAP1)

27

27

Retour sur les références du C++

- Etant donné un **type T**, C++ offre au programmeur le **pseudo-type T &**

- Référence sur un objet de type T
- Une référence correspond à un synonyme ou alias

Ex :
int a;
int & b=a; //b est un alias de a
(référence sur la variable a)

– MAIS b N'EST PAS UNE NOUVELLE VARIABLE!

28

28

- La référence n'est **pas un vrai type**
- La définition d'une référence ne correspond pas à la définition d'une nouvelle variable
- Toute opération effectuée « sur » une référence est effectuée sur la variable référée (et inversement)

– Ex :
int a=4;
int & b=a;
b++;
a++;
b=a;

– Avec const, on peut également créer des références sur des valeurs!!!!

const int & c=15; //Das ce cas un temporaire
// est empilé pour stocker 15

29

29

- Une référence **doit** être initialisée au moment de sa définition

– Par un identifiant d'un **emplacement mémoire de valeur modifiable (-value)** dans le cas d'une référence **T &**

– Par un identifiant d'un emplacement mémoire de valeur modifiable ou non, ou bien par une valeur constante dans le cas d'une référence **const T &**

– double d;
double& dr1=5; //Erreur
double& dr2=d; //OK
const double& cdr2=d; //OK mais pas de
//modif. de d via cdr2
const double& cdr1=5; //OK

30

30

- Attention, la référence est associée à une variable au moment de sa définition, cette association ne peut pas être modifiée par la suite

31

31

```
int a=3;
int &b=a;
```

b ≅ a



32

32

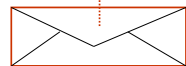
```
int a=3;
int &b=a;
```

b ≅ a



- Secret de Polichinelle :
Réalisation d'une référence à travers un «pointeur masqué»

pm_b
(pointeur masqué)



Chaque occurrence de b dans le programme est remplacée par *pm_b

33

33

Rappel LIFAP1 : Mise en œuvre du passage de paramètres MODIFIABLES (résultat ou donnée résultat) à une procédure en C++

On veut travailler sur les variables originales, pas sur des copies

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

A l'appel
int aa=4, bb=5;
swap(aa,bb);

34

34

A utiliser en lieu et place des pointeurs pour les passages de paramètres MODIFIABLES (résultat ou donnée résultat) en C

```
void swap(int* pa, int* pb)
{ int c;
  c = *pa;
  *pa = *pb;
  *pb = c;
}
```

Utiliser les pointeurs supprime l'ambiguïté de savoir si on travaille sur des références ou sur des int, mais il faut faire le travail d'indirection!

A l'appel
int aa=4, bb=5;
swap(&aa,&bb);

35

35

Passages de paramètres des procédures C++

```
void exemple(int & a, //paramètre MODIFIABLE
             // (donnée-résultat
             //ou résultat)
            int x, const int & z //paramètre
                               //NON MODIFIABLE
                               //(donnée)
)
//Précondition : a variable initialisée quelconque
//Postcondition : a+ a pour valeur x*z
{
  a=x*z;
}
```

36

36

Prérequis C/C++

- Spécificité des tableaux statiques et des chaînes de caractères en C

```
int tab[4];
char st1[20]={ 'S', 'o', 'l', 'e', 'l', 'l', '\0' }; char st2[ ]="Jojo";
```
- Lecture / écriture
 - scanf, printf, ... (entrée/sortie standard)
 - fscanf, fprintf, fread, fwrite, feof, fopen, fclose, ... (entrée/sortie sur fichiers)
 - Entrées sorties avec << et >>
- Différentes étapes de la compilation
- Makefile

37

37

Types composites

- Possibilité de créer des types composites : class et struct
- Dans une class (resp. struct) les données membres et fonctions membres sont private (resp. public) si non spécifié.

```
class Complexe
{ private :
    double re, im;
    double ro, theta;
public :
    void affiche() const;
};
```

38

38

Types composites

```
class Complexe
{ private :
    double re, im;
    double ro, theta;
public :
    void affiche();
};
void Complexe::affiche() const
{ std::cout << this->ro << theta << std::endl; }
```

this est un pointeur sur le Complexe sur lequel on exécute la fonction membre affiche()

Définition et utilisation d'une variable de type Complexe
Complexe z; //Création puis initialisation de z sur la pile
z.affiche(); //Invocation de affiche() sur z

39

39

Mécanisme des Constructeurs et Destructeurs

- **Mécanisme des constructeurs :**
Permet d'**initialiser automatiquement** une instance d'une classe **au moment de sa création** (dans la pile ou dans le tas), sans faire d'appel explicite à la fonction utilisée pour cela.
- Les fonctions d'initialisation automatique sont appelées constructeurs.
- **Mécanisme des destructeurs :**
Permet l'exécution **automatique** d'un ensemble d'instructions **juste avant la disparition** d'une instance d'une classe, sans appel explicite. La fonction appelée automatiquement est le destructeur.

40

```
class Complexe
{ private :
    double re, im, ro, theta;
public :
    Complexe(double r, double i); // Constructeur
    ~Complexe() // Destructeur
        {std::cout << « je vais mourir »;}
};
Complexe::Complexe(double r, double i)
: re(r), im(i)
{ro=std::sqrt(re*re+im*im); theta = std::atan(i/r); }
```

Définition et initialisation automatique d'une variable de type Complexe
Complexe z(3.0, 5.5);

41

41

```
int main() //1 VRmain créée sur la pile
{
    int i; //2 Variable i créée sur la pile
    Complexe z(3.0, 5.5); //3 Variable z créée sur la pile
                        // et initialisée par appel
                        // automatique au constructeur
                        // de la classe Complexe
    z.affiche() //4 appel à la fonction membre affiche
                sur z
}; //5 appel au destructeur sur z
//6 z dépilé de la pile
//7 i dépilé
```

42

42

Mécanisme d'affectation

- On peut aussi spécifier comment on entend mettre en œuvre l'affectation des variables d'une classe.
- Surcharge de l'opérateur d'affectation
operator=

43

43

```
class Complexe
{ private :
  double re, im, ro, theta;
  public :
  ...
  Complexe & operator =(const Complexe & zz)
  {this->re=zz.re; etc...}
};
```

Utilisation :
Complexe z1(3.0, 5.5);
Complexe z2(0.0, 12.5);
z2=z1; // correspond à l'appel z2.operator=(z1);

44

44

Transparent
Optionnel

Compléments de révisions

- Une fois que les prérequis précédents sont acquis vous pourrez optionnellement aborder les 5 slides suivants

45

45

Peut-on voir un appel de fonction à gauche
d'un opérateur d'affectation?

Transparent
Optionnel

46

46

OUI!!!
S'il s'agit d'une fonction retournant une référence

```
Ex : int& elt(int tab[],int i,int t)
    //précondition : tab de taille t (au moins)
    //              : 0<=i<=t-1
    //Résultat : retourne une référence
    //          sur le ième élément de tab.
    //          Attention cette référence pourra ensuite
    //          permettre d'en modifier le contenu
    {return tab[i];}
```

A l'appel :

```
int tatab[4];
elt(tatab,0,4)=2;
```

47

47

Que penser de :

```
int& inc(int i)
{
  int temp=i+1;
  return temp;
}
```

Transparent
Optionnel

48

48


```
int quizz(int & a, int b)
{
    int z=a+b;
    a=a+1;
    return z;
}
```

Transparent
Optionnel

Est-ce une fonction au sens algorithmique du terme?
(ie. qui n'a pas d'effet de bord)

```
float moyenne(float & a, float & b)
{
    float res;
    res = (a+b)/2;
    return res;
}
Même question?
```

49

49

Pour éviter toute ambiguïté :

```
float moyenne(const float & a, const float & b)
{
    float res;
    res = (a+b)/2;
    return res;
}
```

Transparent
Optionnel

50

50

Retour sur les Types Abstraites et leur mise en œuvre en C++

51

51

Types abstraits

- Type :
 - Ensemble des valeurs codées par le type
 - Ensemble des opérations que l'on peut effectuer sur les valeurs et variables de ce type
- Il n'est pas nécessaire de connaître la manière dont les valeurs et les opérations sont codées pour pouvoir les utiliser.
- On utilise les types de façon abstraite, sans connaître leur implantation interne
- Ex : Entier + - * /

52

52

- Il existe un certain nombre de types scalaires de base
- On souhaite construire de nouveaux types abstraits :
 - Constructibles à partir de types existants
 - Manipulables à travers un jeu d'opérations, sans avoir à connaître leur structuration interne
- Ex : Les chaînes de caractères C/C++ sont un piètre exemple de type abstrait
 - Leur utilisation est étroitement liée à leur implantation

53

53

• Description des Types Abstraites dans le cadre de modules

- Module :
 - Regroupe un ensemble de définitions de constantes, de variables globales, de types, de procédures et de fonctions qui forment un ensemble cohérent.
 - Interface du module :
 - Présentation claire des constantes, variables, types, procédures et fonctions offertes par le module
 - Implantation du module :
 - Mise en oeuvre des types, procédures et fonctions proposées dans l'interface. Définition des constantes et variables globales du module.

54

54

- Pourquoi la séparation interface / implantation ?
(Déclarations / définitions)

- Eviter l'introduction (parfois inconsciente) de dépendances entre l'utilisation d'un type et son implantation (idem pour les procédures et les fonctions)
- Possibilité de modifier l'implantation du module sans toucher à son interface
- Possibilité de mettre en place des bibliothèques logicielles

- Si mise en œuvre en C/C++

- Possibilité de compiler l'implantation du module indépendamment des programmes utilisateurs
- Possibilité de transmettre uniquement l'interface et l'implantation compilée au programme utilisateur d'un module

55

55

- Modules et types abstraits : Pseudo langage utilisé

- **module** nom_module { rôle du module }

- **importer**

Déclaration des éléments de modules extérieurs utilisés dans l'interface de nom_module

- **exporter**

Déclaration des éléments offerts par nom_module

- **implantation**

• **Déclaration** des éléments de modules extérieurs utilisés dans l'implantation de nom_module

• **Définition** éventuelle d'éléments internes au module (utiles pour l'implantation de nom_module mais non exportés)

• **Définition** des éléments offerts par nom_module

- **initialisation**

• Actions à exécuter au début du programme pour garantir une utilisation correcte du module

- finmodule**

Eléments = constantes, variables globales (au module), types, procédure et fonction

56

56

module nombre_complexe

- **importer**

Module nombre_réel {offrant le type MonRéel}

- **exporter**

constante PI : MonRéel

variable cpteComplexe : entier {nombre de Complexes manipulés}

Type Complexe

procédure initialiser(Résultat c : Complexe) {pré/postconditions ...}

procédure initialiser(Résultat c1 : Complexe, Donnée c2 : Complexe) {pré/postconditions ...}

procédure initialiser(Résultat c : Complexe, Donnée a,b : MonRéel) {pré/postconditions ...}

procédure testament (donnée-résultat Complexe c) {pré/postconditions}

procédure affectation(donnée-résultat Complexe c1, donnée Complexe c2) {pré/postconditions de L'AFFECTION...}

procédure affiche(donnée Complexe c) {pré/postconditions ...}
Complexe **fonction** addition(donnée Complexe c1, Complexe c2) {préconditions et résultat...}

57

57

- **implantation**

constante PI : MonRéel ← 3,141 59

variable cpteComplexe : entier ← 0

Type Complexe = structure
x : MonRéel
y : MonRéel
fin Complexe

procédure initialiser(Résultat c : Complexe, Donnée a,b : MonRéel)
début
c.x ← a c.y ← b cpteComplexe++
fin initialiser

procédure initialiser(Résultat c : Complexe)
début
initialiser(c,0,0)
fin initialiser

procédure initialiser(Résultat c1 : Complexe, Donnée c2 : Complexe)
début
initialiser(c1,c2.x,c2.y)
fin initialiser

procédure testament(donnée-résultat Complexe c2)
début
cpteComplexe--
fin testament

58

58

procédure affectation(donnée-résultat Complexe c1, donnée Complexe c2)

début
c1.x ← c2.x
c1.y ← c2.y
fin affectation

procédure affiche(donnée Complexe c)

début
affiche (c.x), affiche(" +i "), affiche (c.y)
fin affiche

Complexe **fonction** addition(Complexe c1, Complexe c2)

début
Complexe c
initialiser(c,c1.x+c2.x,c1.y+c2.y)
résultat c
fin affiche

- **Initialisation** {Rien à signaler ici }

finmodule nombre_complexe

59

59

Programme utilisateur

- **importer**

module nombre_réel, nombre_complexe

- **variable**

r : MonRéel

z1 : Complexe

z2 : Complexe

z3 : Complexe

60

60

- Début
 - $r \leftarrow 5$
 - initialiser(z1,r,PI)
 - initialiser(z2,z1)
 - initialiser(z3)
 - affectation(z2, z3)
 - affectation(z3,addition(z1,z2))
 - affiche(z2)
 - testament(z1)
 - testament(z2)
 - testament(z3)
- Fin

61

61

Transparent Optionnel

Les quelques transparents qui suivent sont optionnels et visent à expliquer pourquoi le mécanisme des constructeurs et destructeurs a été mis en place en C++ pour permettre une logique fonctionnelle.

- > On ne peut pas invoquer explicitement le testament sur la valeur de retour d'une fonction!!!!
- > Besoin d'un mécanisme automatique

62

62

- Problème : dans le programme précédent, la valeur de cmpte_complexe n'est pas nulle à la fin de l'exécution du programme!
- La procédure testament n'a pas pu être invoquée sur la valeur de retour de l'addition...
- Solutions :
 - A la place d'une fonction d'addition, possibilité d'opter pour une procédure addition à trois paramètres dont un paramètre résultat sera le Complexe dans lequel on souhaitera stocker le résultat

63

63

Une solution plus élégante

Le programme sera plus élégant si les procédures d'initialisation et de testament peuvent être appelées de manière automatique...

Et encore plus élégant si on peut récupérer les symboles des opérateurs existants (<-, +, -, *, /)

64

64

Programme utilisateur

- **importer**
 - module nombre_réel, nombre_complexe
 - **variable**
 - r : MonRéal $\leftarrow 5$
 - z1 : Complexe $\leftarrow (r,PI)$
 - z2 : Complexe $\leftarrow z1$
 - z3 : Complexe
 - Début
 - z2 $\leftarrow z3$
 - z3 $\leftarrow z1+z2$
 - affiche(z2)
 - Fin
- Appel implicite à la procédure d'initialisation adéquate
- Appel implicite à la procédure testament sur z1, z2 et z3 avant leur disparition

65

65

Transparent Optionnel

Le mécanisme d'appel automatique des procédures d'initialisation et de testament pourra être mis en œuvre en C++ mais pas en C (dans ce cas prévoir des appels explicites), grâce aux constructeurs et destructeurs

66

66

Mise en oeuvre en C++

- Répartition du module sur 2 fichiers
 - **Interface du module** : Fichier d'entête (.h)
Contenant la déclaration des éléments importés et exportés
 - Le fichier d'entête contient **malheureusement** aussi les définitions de type ...
 - **Implantation du module** : Fichier source (.cpp)

67

67

```

#ifndef __NBCOMP
#define __NBCOMP
#include "NombreReel.h" //type MonReel
extern const MonReel PI;
extern int cmpteComplexe;
struct Complexe
{
    MonReel x,y;//données membres

    // Procédures d'initialisation (constructeurs)
    Complexe(); //Pré/postconditions...constructeur par défaut
    Complexe(const Complexe & z); // Pré/postconditions...
    // constructeur par copie
    Complexe(const MonReel& a, const MonReel& b); // Pré/postconditions

    Procédure testament (destructeur)
    ~Complexe(); //Pré/postconditions...

    void affiche() const; //Préconditions et résultat

    Complexe& operator = (const Complexe & z); //Pré/postconditions...
};
// pour permettre les enchaînements d'affectation
Complexe operator +(const Complexe & z1, const Complexe & z2);
#endif
    
```

NombreComplexe.h
Fichier de promesses!

68

68

```

#include "NombreReel.h"
#include "NombreComplexe.h "
const MonReel PI=3,141 59;
int cmpteComplexe=0;

Complexe::Complexe() {(*this).x=(*this).y=0; cmpteComplexe++;
//ie. le x et le y du Complexe à initialiser

Complexe::Complexe(const Complexe & z) {...}

Complexe::Complexe(const MonReel& a, const MonReel& b) {...}

Complexe::~Complexe() {cmpteComplexe--;}

Complexe& Complexe::operator = (const Complexe & z)
{if (this!=& z) //this : adresse du Complexe à affecter
{(*this).x=z.x; (*this).y=z.y;
return *this;
}

Complexe operator +(const Complexe & z1, const Complexe & z2)
{...}
void Complexe::affiche() const
{...}
    
```

NombreComplexe.cpp

69

69

```

main.cpp

#include "NombreReel.h"
#include "NombreComplexe.h "
int main()
{
    Monreel r=5;
    Complexe z1(r,PI);
    Complexe z2(z1); //appel constructeur par copie
    Complexe z3; //appel constructeur par défaut
    z2=z1+z1;
    return 0;
} // appel au destructeur des variables
//z1,z2 et z3 avant qu'elles ne soient détruites
    
```

70

70

Remarque 1 :

- Le type Complexe est implanté sous forme d'une structure à 2 champs x et y (**données membres**)

Remarque 2 :

- Les procédures d'**initialisation automatique (constructeurs)** d'un Complexe ne font pas apparaître le Complexe sur lequel elles agissent dans la liste de leurs paramètres.
- Accès au Complexe concerné via un pointeur : **this**
- Idem pour la procédure de **testament automatique** avant disparition (**destructeur**) d'un Complexe
- Idem pour la surcharge de l'opérateur d'affectation

On dit que ce sont des **fonctions membres** de la struct Complexe

71

71

Caractéristiques des fonctions membres :

- Déclaration dans la portée de struct Complexe
- Nom complet préfixé par **Complexe::** (c'est **ce** nom complet qui est utilisé au moment de la **définition** de la fonction membre)
- Une fonction membre est une fonction qui possède un **argument implicite** supplémentaire **this** (adresse du Complexe sur lequel est invoquée la fonction membre)
- Fonctions membres à ajouter : accès à ou modification des données membre (accesseurs *get* et modifieurs *set*)

72

72

Attention :

- L'opérateur + n'a pas été surchargé en fonction membre de la struct Complexe :
 - 2 opérandes
 - aucune raison de privilégier son opérande de gauche plutôt que celui de droite
 - Ainsi on conserve le caractère symétrique de l'opérateur +

73

73

Mot clef *static*

- 2 usages possibles :
 - Utilisation pour définir des éléments **internes** à un module
 - Utilisations pour définir des variables de classes (pas forcément internes, elles peuvent être exportées à l'extérieur du module)

74

74

Mot clef *static*

- Utilisation pour définir des éléments **internes** à un module
 - Éléments non exportés (non utilisables dans d'autres modules)
- Exemple :
 - Pour l'instant la variable globale cmpteComplexe du module NombreComplexe est exportée,
 - Possibilité de la modifier depuis un programme utilisateur et donc de lui faire perdre son intégrité
 - Il serait préférable :
 - Que cmpteComplexe soit une variable globale **interne** au module NombreComplexe
 - et que l'on exporte juste une fonction permettant d'accéder à sa valeur depuis un programme utilisateur

75

75

```
#ifndef __NBCOMP
#define __NBCOMP
#include "NombreReel.h" //type MonReel

extern const MonReel PI;

// La variable globale cmpteComplexe a été retirée de l'interface
// au profit d'une fonction d'accès à sa valeur

struct Complexe
{
    MonReel x,y;//données membres

    // déclaration procédures d'initialisation (constructeurs)
    ...

    // déclaration procédure testament (destructeur)
    ...

    // déclaration surcharge opération d'affectation (operator =)
};

// Procédures, fonctions et opérations offertes sur le type abstrait Complexe
...
int nbComplexesVivants();
//Préconditions : aucune
//Résultat : nombre de Complexes existant en mémoire au moment de l'appel
#endif
```

76

76

```
#include "NombreReel.h"
#include "NombreComplexe.h"
#include <iostream>
```

```
const MonReel PI=3,141 59;
static int cmpteComplexe=0; //définition variable globale interne
```

```
//On peut aussi définir des fonctions internes à un module
```

```
static int exempleFonctionInterne()
{
    std::cout << " Hello " << std::endl;
    return 8;
}
```

cmpteComplexe et
exempleFonctionInterne
accessibles **uniquement** dans
NombreComplexe.cpp

```
//Définition des constructeurs de Complexe
```

```
//Définition du destructeur de Complexe
```

```
//Définition de la surcharge de l'opérateur d'affectation
```

```
//Définition des procédures, fonctions et opérations  
// offertes sur le type abstrait Complexe
```

```
int nbComplexesVivants()
{
    return cmpteComplexe;
}
```

77

77

Mot clef *static*

- Le mot clé static peut être utilisé pour définir des variables de classes ie. des variables globales définies **DANS** la portée d'une classe.
- **class Complexe**

```
{
    static int compte_complexe;
    ...
};
```

 // déclaration dans le fichier complexe.h
- Le nom complet de notre variable globale est `Complexe::compte_complexe`
- `int Complexe::compte_complexe = 0;` // définition dans le fichier complexe.cpp
- **Les variables de classes peuvent être exportées**

78

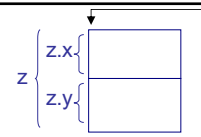
78

Constructeurs

- Constructeur : fonction membre portant le nom de la struct, dont C++ génère l'appel, au moment de la création d'instances du type ainsi créé
 - avant le main pour les variables globales,
 - au sein d'un bloc pour les variables locales,
 - lors d'un appel à l'opérateur new pour les instances allouées dynamiquement

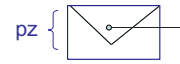
79

```
Complexe z;
//Création sur la pile
//et initialisation par
//le constructeur par défaut
```

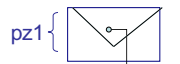


```
Complexe *pz=&z;
```

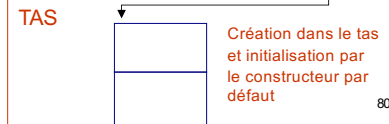
Notations :
(*pz).x peut s'écrire pz->x



```
Complexe *pz1=new Complexe;
```



```
...
delete pz1;
```



80

79

80

- Il peut y avoir plusieurs constructeurs
- Constructeurs usuels
 - constructeur par défaut (sans argument)
 - constructeur par copie
- Pas de type de retour dans la signature des constructeurs

```
Echier.h
struct CC
{
    CC(); //par défaut
    CC(const CC &); //par copie
};
```

```
Echier.cpp
CC::CC()
{ corps constructeur par défaut}
...
```

81

81

- Arguments du constructeur : fournis à la définition de l'instance
 - Complexe z1; //Initialisation par défaut
 - Complexe z2(z1); //Initialisation par copie
 - Complexe * pz=new Complexe(2,5);

82

82

- Remarque : un constructeur peut être invoqué de manière **explicite** pour créer une variable temporaire

- Exemple :
z1=z+Complexe(2,7);

83

83

Constructeur par copie

- Initialisation d'une instance à partir d'une référence sur une instance de même type
 - Complexe z(zz);
 - Complexe z=zz;
 - Complexe z=Complexe(zz);
 - Complexe *pz=new Complexe(zz);
- Appel au constructeur par copie de signature
Complexe::Complexe(const Complexe &)

84

84

Constructeur par copie

- Si vous n'avez pas prévu de constructeur par copie dans votre module :
 - Existence d'un constructeur par copie implicite qui fait une copie des champs en suivant leur propre lois de copie
 - Pour un type primitif la copie employée est une copie bit à bit

85

85

- Attention : Le constructeur par copie est appelé en de multiples circonstances
- Ex :

```
Complexe UneFct(Complexe t)
{return t;}
```

```
Complexe z;
UneFct(z);
Combien d'appel au constructeur par copie?
```

86

86

- Appel au constructeur par copie :
 - quand une instance est définie et initialisée à partir d'une autre,
 - quand on passe une valeur de type Complexe à une fonction,
 - quand une fonction retourne une valeur de type Complexe

87

87

Destructeur

- Fonction membre appelée **automatiquement**, au moment de la destruction d'une instance, **avant** libération de l'espace mémoire correspondant
 - A la fin du programme, pour les instances globales
 - A la fin du bloc, pour les instances locales (automatic)
 - Lors de l'appel à delete, pour les instances allouées dynamiquement
- Un destructeur est
 - **unique**,
 - sans argument
- Syntaxe
Nom de la struct précédé de ~

88

88

```
struct CC      CC::CC()
{              { ad=new int[5]; }
  CC();       CC:: ~CC()
  ~CC();      {
              delete [] ad;
              std::cout<<"Je n'ai plus rien "
              "a restituer dans le tas \n";
              }
  int *ad;
};
```

Attention : Dans ce cas là, un constructeur par copie serait également nécessaire! Savez vous pourquoi?

89

89

Destructeur

- Si vous n'avez pas prévu de destructeur dans votre module :
 - Existence d'un destructeur implicite qui appelle le destructeur des données membres qui en possèdent un

90

90

L'opérateur d'affectation

- Opérateur membre appelé dans les instructions du type

```
CC a,b;  
a=b; //appel à l'opérateur d'affectation de CC  
// a.operator=(b)
```

- Prototype de l'opérateur membre d'affectation

```
CC & CC::operator = (const CC &);  
const CC & CC::operator = (const CC &);
```

- Existence d'un opérateur d'affectation implicite qui fait une copie des champs

91