

2015-2016, Semestre d'automne
L3, Licence Sciences et Technologies
Université Lyon 1

LIF9: Algorithmique, Programmation et Complexité

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

1

Organisation de cet enseignement

- Répartition sur 10 semaines
 - 1h30 de cours
 - 1h30 de TD
 - 3h de TP
- Auxquelles il faut ajouter des créneaux correspondant à des rendus de TP et un TP complémentaire

2

Evaluation des connaissances

- Plan « Réussir en Licence »
- 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)
- TDs de soutien
- Enseignants référents

3

Un point important

- Cours Magistral écologique :
 - Veuillez éteindre vos portables (téléphone et ordinateur)
 - Brouillon conseillé

4

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

5

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**!

6

Bibliographie C++

- **C++ Primer**, Lippman & Lajoie, Third Edition, Addison-Wesley, 1998
- **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)

7

Objectifs

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

8

Prérequis

- **LIF1, LIF5, LIF7** (et dans une moindre mesure **LIF3**)
- Gestion de la mémoire
 - Organisation en pile des variables d'un programme
 - Allocation dynamique dans le tas (*new/delete, malloc/free*)
- Différents modes de passage des paramètres d'une procédure
 - Donnée, donnée-résultat, résultat
- Notion de pointeur
- Arithmétique des pointeurs
- Type Abstrait et **Programmation Modulaire**
- Nuance entre **définition** et **déclaration**

9

Prérequis

- Spécificité des tableaux statiques et des chaînes de caractères en C/C++
- Lecture / écriture
 - scanf, printf, ... (entrée/sortie standard)
 - fscanf, fprintf, fread, fwrite, feof, fopen, fclose, ... (entrée/sortie sur fichiers)
- Différentes étapes de la compilation
- Makefile
- ...

10

Prérequis

- 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 (**LIF5**) et (pseudo-type) référence (**LIF1**)

11

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)
```

12

- 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;
```
- On peut également créer des références sur des valeurs :  

```
const int &c=15;
```

13

- Une référence **doit** être initialisée au moment de sa définition
  - Par un identifiant de valeur modifiable (*l-value*) dans le cas d'une référence **T &**
  - Par un identifiant de valeur modifiable ou constante dans le cas d'une référence **const T &**
- Ex :  


```
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
```

14

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

15


```
int a=3;
int &b=a; b ≡ a
```



16

```
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

17

Rappel LIF1 : Mise en œuvre du passage de paramètres **résultat** ou **donnée résultat** à une procédure en C++

```
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);
```

18

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

19

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;
```

20

Que penser de :

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

21

## Retour sur les principaux paradigmes de programmation

- **Programmation fonctionnelle LIF3 (Scheme,...)**
  - Un programme est une fonction qui fournit une **valeur à partir des valeurs** d'arguments
  - Fondé sur le lambda-calcul
  - Exemple :
 

```
(list 8 16 32 40)
(cons 8 (cons 16 (cons 32 (cons 40 '()))))
```

    - correspond, dans les 2 cas, à la **valeur** de la liste contenant les valeurs 8 16 32 et 40
    - on peut la désigner par un nom
 

```
(define lili (list 8 16 32 40))
```

22

## Retour sur les principaux paradigmes de programmation

- **Programmation procédurale (ou impérative) LIF1, LIF5, LIF7 (C, C/C++,...)**

- Le calcul est effectué par **effet de bord** (changement d'état) sur les **variables** d'un programme
- Modèle très répandu car proche de la structure des ordinateurs (modèle de Von Neumann)

– Exemple :

```
Liste lili;
initialisationDefault(lili);
ajoutEnTete(lili,40);
ajoutEnTete(lili,32);
ajoutEnTete(lili,16);
```

Extension du langage C (ANSI89-ISO90) auquel sont ajoutés certains éléments C++ (ISO98) : référence, surcharge fonction

- La variable lili contient alors les valeurs 16, 32 et 40
- Ne pas oublier d'exécuter `testament(lili)` avant la disparition de la variable lili !!!!!

23

## Retour sur les principaux paradigmes de programmation

- **Programmation objet LIF13 (Java, C++,...)**
  - Variante du modèle de Von Neumann
  - Calcul résultant de **l'interaction entre objets**
  - Gestion du **polymorphisme**

24

## Retour sur les principaux paradigmes de programmation

- Avantages de la programmation fonctionnelle :
  - Proche du raisonnement mathématique
  - Ecriture concise
  - Possibilité d'**enchaîner** les appels de fonctions sans avoir à stocker des résultats temporaires dans des variables
- Avantages de la programmation procédurale :
  - Possibilité de stocker un unique exemplaire d'une valeur dans une variable, sur laquelle on aura ensuite des effets de bord en différents points d'interaction du programme

25

## Retour sur les principaux paradigmes de programmation

- Certains langages sont plus étudiés que d'autres pour permettre la mise en œuvre de tel ou tel paradigme
- La plupart des langages fonctionnels offrent aussi la notion de variable ainsi que des mécanismes procéduraux
- La plupart des langages procéduraux permettent de programmer de manière fonctionnelle, **dans une certaine mesure ...**

26

## Retour sur les principaux paradigmes de programmation

- ... et conclusions à en tirer pour une saine programmation C/C++
  - Possibilité de tirer avantage des 2 modes de programmation fonctionnelle et procédurale sans en enfreindre les principes
  - En C/C++, on garde généralement une sémantique procédurale, tout en s'offrant la possibilité de retourner une valeur qui pourra intervenir dans d'autres traitements

27

## Que penser de :

```
struct Liste
{
 Cellule * ad;
 unsigned int nb_elem;
};

struct Cellule
{
 Element info;
 Cellule *suivant;
};
```

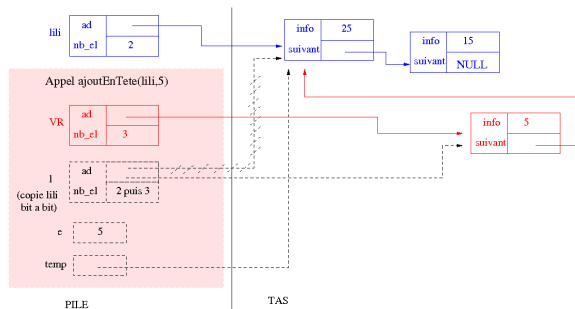
Bernard souhaite privilégier un mécanisme fonctionnel :

Liste ajoutEnTete(Liste l, Element e)

```
{ Cellule * temp=l.ad;
 l.ad=new Cellule; if(l.ad==NULL) exit(1);
 l.ad->info=e;
 l.ad->suivant=temp;
 l.nb_elem++;
 return l;
}
```

28

- Trace d'un appel à ajoutEnTete sur une Liste lili (cliché de la mémoire juste avant la sortie de la fonction)



29

- Tout se passe bien si l'appel à ajout en tête se fait dans le cadre d'une instruction de la forme :

```
lili = ajoutEnTete(lili,5);
lili = ajoutEnTete(ajouteEnTete(lili,4),7);
lili = ajoutEnTete(((ajouteEnTete(lili,6),...),15);
etc.
```

où lili est une variable de type Liste (préalablement initialisée en un point du programme)

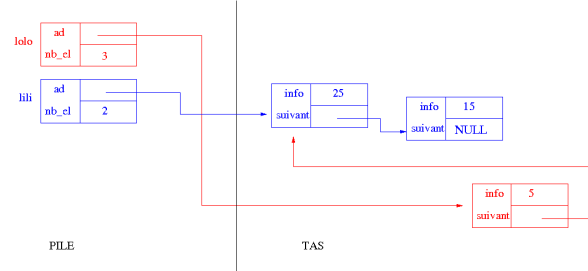
30

Problème :

- Si la valeur renvoyée n'est pas récupérée :
  - ajouteEnTete(lili,5);
  - impossibilité d'opérer une gestion saine de la mémoire (aucune trace de l'adresse de la nouvelle Cellule créée dans le tas)
- Si on appelle ajouteEnTete dans une instruction du type lolo = ajouteEnTete(lili,5);
  - lili et lolo ne sont plus des listes indépendantes ...
  - on court à la catastrophe, notamment à la destruction d'une des 2 listes

31

Résultat de l'instruction lolo = ajouteEnTete(lili,5);



32

Conseil de programmation :

- Eviter les fonctions dont il faut ABSOLUMENT récupérer la valeur de retour pour permettre une gestion saine de la mémoire...
- Dans ce cas, il est préférable de privilégier une formulation procédurale.

La possibilité de faire des enchaînements de traitements en cascade n'est pas perdue pour autant!

- Faire une procédure qui a un effet de bord sur la liste que l'on souhaite modifier et qui retourne
  - une référence sur cette liste (C/C++)
  - ou un pointeur sur cette liste (C)

33

Que penser de :

```
struct Liste {
 Cellule * ad;
 unsigned int nb_elem;
};

struct Cellule {
 Element info;
 Cellule *suivant;
};
```

Programmation procédurale avec un parfum fonctionnel

```
Liste & ajoutEnTete (Liste & l, Element e)
{
 Cellule * temp=l.ad;
 l.ad=new Cellule; if(l.ad==NULL) exit(1);
 l.ad->info=e;
 l.ad->suivant=temp;
 l.nb_elem++;
 return l;
}
```

34

Exemple :

- Utilisation sur une variable lili de type Liste (préalablement initialisée en un point du programme)

```
ajoutEnTete(lili,5); //Pas de pb cette fois!
lili=ajoutEnTete(lili,5); //eventuellement, mais inutile!
ajoutEnTete(ajoutEnTete(ajoutEnTete(lili,55),33),22);
```

- Attention toutefois de ne toujours pas écrire une instruction du type :

```
lolo=ajoutEnTete(lili,10);
```

Pour la même raison que précédemment!

Dans ce cas, il faut écrire et utiliser une procédure d'affectation qui affecte à une liste une copie d'une autre (cf. LIFS)

```
affectation(lolo,lili);
```

35

Les remarques précédentes s'étendent bien sûr à des traitements comme l'initialisation, l'affectation ...

Programmation procédurale avec un parfum fonctionnel

```
Liste & initialisationDéfaut (Liste & l)
{
 l.ad = NULL;
 l.nb_elem=0;
 return l;
}

Liste & affectation (Liste & l1, const Liste & l2)
{
 //on vide l1
 //pour chaque élément de l2
 //(parcours à l'aide d'un pointeur de travail)
 //insertion en queue de l1 (création de cellules)
 return l1;
}
```

36

## Le(s) langage(s) de l'UE

- C/C++: pour une utilisation du C plus proche de l'algo
  - mise en œuvre facile des passages de paramètre donnée-résultat et résultat (utilisation de référence)
  - simplification du passage de ces paramètres au moment de l'appel à la procédure
  - Pour que l'adaptation à C++ soit plus facile le moment venu

37

## • Rappel C/C++ (LIF1, LIF5, LIF7) :

- Passages paramètres donnée , donnée-résultat ou résultat:

```
void proc(type1 par1, //paramètre donnée
 const type2 & par2, //paramètre donnée ENCOMBRANT
 type3 & par3, //paramètre donnée-résultat
 type4 & par4); //paramètre résultat
```

```
• Exemple d'utilisation :
type1 p1; type2 p2; type3 p3; type4 p4;
// initialisation de p1, p2, p3 mais pas de p4 :
initialisationXouY1(p1);
initialisationXouY2(p2);
initialisationXouY3(p3);
```

```
...
proc(p1, p2, p3, p4);
```

Paramètres formels :  
Valeur ou variables

Paramètres formels :  
**Variables** uniquement

38

## Le(s) langage(s) de l'UE

- Le langage C/C++
- Et si on ne dispose que d'un compilateur C?
- Il faut régler les passages de paramètre en contournant le problème de l'absence des références :
  - Utilisation de pointeurs
  - L'utilisateur de la procédure devra s'adapter au mécanisme d'indirection ainsi introduit...

39

## • Rappel C (LIF5) :

- Passages paramètres donnée , donnée-résultat ou résultat:

```
void proc(type1 par1, //paramètre donnée
 const type2 * adp2, //paramètre donnée ENCOMBRANT
 type3 * adp3, //paramètre donnée-résultat
 type4 * adp4); //paramètre résultat
```

```
• Exemple d'utilisation :
type1 p1; type2 p2; type3 p3; type4 p4;
// initialisation de p1, p2, p3 mais pas de p4 :
initialisationXouY1(&p1);
initialisationXouY2(&p2);
initialisationXouY3(&p3);
```

```
...
proc(p1, &p2, &p3, &p4);
```

Paramètres formels :  
Valeur ou variables

Paramètres formels :  
**Adresses de variables** uniquement

40