CM 15 - INITIATION À LA PROGRAMMATION MODULAIRE

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

PRINCIPES GLOBAUX

▷ Une première forme de modularité consiste, comme nous l'avons déjà évoqué, au découpage d'un code en plusieurs fichiers

▷ Ce découpage permet de meilleures lisibilité et réutilisabilité du code

▷ Il faut donc éviter l'effet "couteau suisse"

▷ Chaque fichier correspond à un ensemble de services portant sur une thématique précise (e.g., des fonctions mathématiques ou de traitement de fichiers)

▷ On parle alors de cohérence

PRINCIPES GLOBAUX

▷ Une autre forme de modularité consiste à rendre le plus indépendant possible notre code de celui qui pourra l'utiliser

▷ Cela est rendu possible en découplant les choix d'implémentation de la définition globale des services; souvent dénommée interface

▷ Mais également en rendant indépendant l'utilisation de notre interface des choix d'implémentation

▷ On parle alors d'encapsulation

▷ Cela permet, comme nous le verrons, de modifier l'implémentation de notre interface sans répercussion sur le code l'utilisant

FIL D'ARIANE

- On se propose de considérer la mise en oeuvre d'un module de Pile d'entiers
- Comme abordé dans l'UE d'Algorithmique, une pile est une structure de données de type LIFO (Last-In, First-Out) où seul le dernier élément inséré est accessible
- Une pile est définie par deux opérations d'accès
  - `int peek()`
  - `bool isEmpty()`
- Et deux opérations de modification
  - `void push(int element)`
  - `int pop()`

MODÉLISATION D'UNE PILE VIA UN TABLEAU

▷ On définit une structure

```
struct istack{
    int content[MAX+1];
    int topIndex;
};
```

▷ Le contenu de la pile est représenté par un tableau de taille fixée (définie par une constante **MAX**)

▷ **topIndex** représente l'indice du sommet de la pile dans le tableau **content**

Modélisation d'une pile via un tableau

▷ Afin de pouvoir être appliquées sur plusieurs piles, les fonctions doivent prendre en paramètre la pile sur laquelle elles s'appliquent

▷ Tout argument étant passé par copie en C, nous utiliserons l'adresses de piles comme paramètre

▷ Cela évitera la recopie de la structure sur la pile d'appel dans le cas des fonctions d'accès

▷ Cela permettra la modification de la structure dans le cas des fonctions de modification

FONCTIONS PEEK ET POP

```
int peekIStack(struct istack * p_s){
    if(isEmptyIStack(p_s)){
        fprintf(stderr,"Stack is empty!\n");
        exit(EXIT_FAILURE);
    }
    return (*p_s).content[(*p_s).topIndex];
}

int popIStack(struct istack * p_s){
    int elt=peekIStack(p_s);
    (*p_s).topIndex--;
    return elt;
}
```

FONCTION ISEMPTY OU LE DÉBUT DES ENNUIS

▷ L'implémentation de la fonction `isEmpty` nécessite déjà des choix d'implémentation

```
bool isEmptyIStack(struct istack * p_s){
    return ((*p_s).topIndex<0);
}
```

▷ Cela créé une dépendance avec l'initialisation de la pile

▷ En effet, notre fonction considère que, a priori, une pile nouvellement créée devra avoir le champs **topIndex** initialisé à une valeur négative

FONCTION PUSH ET LES ENNUIS CONTINUENT

▷ L'implémentation de la fonction **push** impose des choix d'implémentation supplémentaires

```
void pushIStack(int elt, struct istack * p_s){
    if((*p_s).topIndex>=MAX){
        fprintf(stderr,"Stack is full!\n");
        exit(EXIT_FAILURE);
    }
    (*p_s).topIndex++;
    (*p_s).content[(*p_s).topIndex]=elt;
}
```

▷ Pour fonctionner, une pile vide devrait avoir le champs **topIndex** initialisé à **-1**

▷ On créé également une dépendance à la macro **MAX**

Exemple de code d'utilisation

▷ Notre pile peut être actuellement utilisée ainsi

```
...
#include "istack.h"
int main(void){
    struct istack s;
    s.topIndex=-1;
    pushIStack(1,&s);
    pushIStack(2,&s);
    while(!isEmptyIStack(&s)){
        printf("%d ",peekIStack(&s));
        printf("%d\n",popIStack(&s));
    }
    return EXIT_SUCCESS;
}
```

▷ La pile est de taille limitée mais l'utilisateur n'en sait trop rien et il doit faire attention à l'initialisation

▷ C'est peu satisfaisant !

PREMIÈRES AMÉLIORATIONS

▷ Il devient nécessaire de maîtriser la création d'une pile et de limiter la dépendance à la macro **MAX**

▷ Pour ce faire, on se débarrasse de **MAX** et modifie la structure comme suit

```
struct istack{
    int * content;
    int topIndex;
    int maxSize;
};
void pushIStack(int elt, struct istack * p_s){
    if((*p_s).topIndex>=(*p_s).maxSize-1){...}
    ...
}
```

PREMIÈRES AMÉLIORATIONS

▷ Il devient nécessaire de maîtriser la création d'une pile et de limiter la dépendance à la macro **MAX**

▷ Nous ajoutons également une fonction de création de pile

```
struct istack * createIStack(int size){
    struct istack * p_s = malloc(sizeof(struct istack));
    if(p_s==NULL){
        fprintf(stderr,"Not enough memory!\n");
        exit(EXIT_FAILURE);
    }
    (*p_s).topIndex=-1;
    (*p_s).maxSize=size;
    (*p_s).content= (int*) malloc(size*sizeof(int));
    if((*p_s).content==NULL){
        fprintf(stderr,"Not enough memory!\n");
        exit(EXIT_FAILURE);
    }
    return p_s;
}
```

Fichier d'en-tête

▷ Notre fichier d'en-tête ressemble à ceci maintenant

```
#ifndef _ISTACK_H_
    #define _ISTACK_H_
    struct istack{
        int * content;
        int topIndex;
        int maxSize;
    };
    struct istack * createIStack(int size);
    bool isEmptyIStack(struct istack * p_s);
    int peekIStack(struct istack * p_s);
    void pushIStack(int elt, struct istack * p_s);
    int popIStack(struct istack * p_s);
#endif
```

Exemple de code d'utilisation

▷ Notre pile peut alors être utilisée ainsi

```
...
#include "istack.h"
int main(void){
    struct istack * p_s = createIStack(20);
    pushIStack(1,p_s);
    pushIStack(2,p_s);
    while(!isEmptyIStack(p_s)){
        printf("%d ",peekIStack(p_s));
        printf("%d\n",popIStack(p_s));
    }
    return EXIT_SUCCESS;
}
```

▷ C'est mieux mais il subsiste plusieurs problèmes important

▷ Lesquels à votre avis ?

PROBLÈMES INHÉRENTS

▷ ⚠Rien ne contraint l'utilisateur d'utiliser notre fonction de création de pile

```
...
#include "istack.h"
int main(void){
    struct istack s;
    struct istack * p_s = &s;
    pushIStack(1,p_s);
    pushIStack(2,p_s);
    while(!isEmptyIStack(p_s)){
        printf("%d ",peekIStack(p_s));
        printf("%d\n",popIStack(p_s));
    }
    return EXIT_SUCCESS;
}
```

PROBLÈMES INHÉRENTS

▷ ⚠Rien n'interdit l'utilisateur de manipuler les champs de la pile directement

```
...
#include "istack.h"
int main(void){
    struct istack * p_s =  createIStack(20);
    pushIStack(1,p_s);
    pushIStack(2,p_s);
    (*p_s).topIndex++;
    while(!isEmptyIStack(p_s)){
        printf("%d ",peekIStack(p_s));
        printf("%d\n",popIStack(p_s));
    }
    return EXIT_SUCCESS;
}
```

▷ On souhaite qu'il connaisse l'existence de notre structure mais pas son implémentation ...

ENCAPSULATION

▷ L'utilisateur peut ne manipuler que des adresses vers des piles !

▷ On déplace la définition de la structure dans le fichier d'implémentation (i.e., dans le **.c**)

▷ Notre fichier d'en-tête devient

```
#ifndef _ISTACK_H_
    #define _ISTACK_H_
    typedef struct istack * IStack;
    IStack createIStack(int size);
    bool isEmptyIStack(IStack p_s);
    int peekIStack(IStack p_s);
    void pushIStack(int elt, IStack p_s);
    int popIStack(IStack p_s);
#endif
```

QUID DE LA LIBÉRATION DE MÉMOIRE

▷ Ne sachant pas ce qu'est vraiment une pile, l'utilisateur ne peut plus la détruire correctement

▷ On ajoute donc une fonction à notre module

```
void destroyIStack(IStack p_s){
    free((*p_s).content);
    free(p_s);
}
```

▷ Notons que cela respecte le principe "celui qui alloue et également celui qui désalloue"

GAIN DE CETTE SOLUTION

▷ L'interface ainsi obtenue permet de respecter le principe d'encapsulation

▷ Elle permet également aisément de pouvoir changer l'implémentation de notre pile sans impacter les codes utilisant notre interface

▷ Nous allons donc en profiter pour proposer une implémentation plus flexible basée sur les listes

▷ L'idée de base est de pouvoir extraire facilement le dernier élément inséré dans la liste

▷ Comme les opérations sur la tête de liste sont généralement plus avantageuses, nous appliquerons les insertions et extractions sur celle-ci

RAPPELS D'ALGORITHMIQUE

▷ Une liste correspond à un ensemble extensible d'éléments ordonnés

▷ Il existe de multiples implémentations de ces dernières (e.g., simplement/doublement chaînée, à sentinelle, circulaire, ...)

▷ Nous nous contenterons ici de l'interface suivante

```
#ifndef _ILIST_H_
    #define _ILIST_H_
    typedef struct ilist * IList;
    IList createEmptyIList();
    void destroyIList(IList p_l);
    int getSizeIList(IList p_l);
    void addAsFirstElementIList(int elt, IList p_l);
    int getFirstElementIList(IList p_l);
    void removeFirstElementIList(IList p_l);
#endif
```

MODÉLISATION D'UNE PILE VIA UNE LISTE

```
...
#include "istack.h"
#include "ilist.h"

struct istack{
    IList content;
    int maxSize;
}

IStack createIStack(int size){
    IStack p_s = malloc(sizeof(struct istack));
    if(p_s==NULL){
        fprintf(stderr,"Not enough memory!\n");
        exit(EXIT_FAILURE);
    }
    (*p_s).maxSize=size;
    (*p_s).content= createEmptyIList();
    return p_s;
}
```

MODÉLISATION D'UNE PILE VIA UNE LISTE

```
int peekIStack(IStack p_s){
    if(isEmptyIStack(p_s)){
        fprintf(stderr,"Stack is empty!\n");
        exit(EXIT_FAILURE);
    }
    return getFirstElementIList((*p_s).content);
}

int popIStack(IStack p_s){
    int elt=peekIStack(p_s);
    removeFirstElementIList((*p_s).content);
    return elt;
}

bool isEmptyIStack(IStack p_s){
    return (getSizeIList((*p_s).content)==0);
}
```

MODÉLISATION D'UNE PILE VIA UNE LISTE

```
void pushIStack(int elt, IStack p_s){
    if(getSizeIList((*p_s).content)==(*p_s).maxSize){
        fprintf(stderr,"Stack is full!\n");
        exit(EXIT_FAILURE);
    }
    addAsFirstElementIList(elt,(*p_s).content);
}
```

▷ La taille d'une liste n'étant pas, a priori, bornée, nous devons gérer la limitation de la taille de la pile nous-même

▷ N'ayant pas apporté de modification à l'interface, le code utilisateur n'est pas impacté

▷ Seul l'implémentation de notre module est à re-compiler

▷ Pour que notre nouvelle implémentation soit effective, l'édition des "nouveaux" liens sera nécessaire

PILE DE DOUBLE

▷ Fort de notre expérience, nous souhaitons proposer une implémentation d'une pile de réels dont l'interface sera

```
#ifndef _DSTACK_H_
    #define _DSTACK_H_
    typedef struct dstack * DStack;
    DStack createDStack(int size);
    bool isEmptyDStack(DStack p_s);
    double peekDStack(DStack p_s);
    void pushDStack(double elt, DStack p_s);
    double popDStack(DStack p_s);
#endif
```

▷ A l'aise avec le copier-coller et le chercher-remplacer, nous ne tardons pas, après de la correction de bugs, à obtenir une implémentation fonctionnelle

▷ Pourquoi est-ce mal ?

DUPLICATION ? QUELLE DUPLICATION ?

```
$>diff -y --suppress-common-lines dstack.c istack.c
#include ”dstack.h”                  | #include ”istack.h”
struct dstack{                       | struct istack{
double * content;                    | int * content;
DStack createDStack(int size){       | IStack createIStack(int size){
DStack p_s = malloc(... dstack));    | IStack p_s = malloc(... istack));
(double*) malloc(... double));       | (int*) malloc(... int));
void destroyDStack(DStack p_s){      | void destroyIStack(IStack p_s){
bool isEmptyDStack(DStack p_s){      | bool isEmptyIStack(IStack p_s){
double peekDStack(DStack p_s){       | int peekIStack(IStack p_s){
if(isEmptyDStack(p_s)){              | if(isEmptyIStack(p_s)){
void pushDStack(double elt, DStack|  | void pushIStack(int elt, IStack
double popDStack(DStack p_s){        | int popIStack(IStack p_s){
double elt=peekDStack(p_s);          |     int elt=peekIStack(p_s);

$>wc -l dstack.c
    57 dstack.c
```

UN PEU DE GÉNÉRICITÉ

▷ Les deux implémentations stockent des pointeurs

▷ Etant donné que tous les pointeurs sont unifiables sur le type `void *`, on peut écrire une pile générique

```
#ifndef _STACK_H_
    #define _STACK_H_
    typedef struct stack * Stack;
    Stack createStack(int size);
    void destroyStack(Stack p_s);
    bool isEmptyStack(Stack p_s);
    void * peekStack(Stack p_s);
    void pushStack(void * elt, Stack p_s);
    void * popStack(Stack p_s);
#endif
```

UN PEU DE GÉNÉRICITÉ

```
#include "stack.h"
struct stack{
    void* * content;
    int topIndex;
    int maxSize;
};
Stack createStack(int size){
    Stack p_s = malloc(sizeof(struct stack));
    if(p_s==NULL){
        fprintf(stderr,"Not enough memory!\n");
        exit(EXIT_FAILURE);
    }
    (*p_s).topIndex = -1;
    (*p_s).maxSize = size;
    (*p_s).content = (void**) malloc(size*sizeof(void *));
    if((*p_s).content==NULL){
        fprintf(stderr,"Not enough memory!\n");
        exit(EXIT_FAILURE);
    }
    return p_s;
}
```

UN PEU DE GÉNÉRICITÉ

```
void destroyStack(Stack p_s){
    free((*p_s).content);
    free(p_s);
}

bool isEmptyStack(Stack p_s){
    return ((*p_s).topIndex<0);
}

void * peekStack(Stack p_s){
    if(isEmptyStack(p_s)){
        fprintf(stderr,"Stack is empty!\n");
        exit(EXIT_FAILURE);
    }
    return (*p_s).content[(*p_s).topIndex];
}
```

UN PEU DE GÉNÉRICITÉ

```
void pushStack(void * elt, Stack p_s){
    if((*p_s).topIndex==(*p_s).maxSize){
        fprintf(stderr,"Stack is full!\n");
        exit(EXIT_FAILURE);
    }
    (*p_s).topIndex++;
    (*p_s).content[(*p_s).topIndex]=elt;
}

void * popIStack(Stack p_s){
    void* elt=peekStack(p_s);
    (*p_s).topIndex--;
    return elt;
}
```

▷ On a maintenant un code générique qui permet de représenter une pile de pointeurs

▷ On peut, en revanche, y insérer tout et n'importe quoi ...

UN PEU DE GÉNÉRICITÉ

▷ On va l'utiliser dans l'implémentation pour déléguer le stockage des informations

```
struct dstack {
    Stack content;
};
DStack createDStack(int size){
    DStack p_s = (DStack) malloc(sizeof(struct dstack));
    if(p_s==NULL){
        fprintf(stderr,"Not enough memory!\n");
        exit(EXIT_FAILURE);
    }
    (*p_s).content = createStack(size);
    return p_s;
}
void destroyDStack(DStack p_s){
    destroyStack((*p_s).content);
    free(p_s);
}
```

UN PEU DE GÉNÉRICITÉ

```
bool isEmptyDStack(DStack p_s){
    return isEmptyStack((*p_s).content);
}
double peekDStack(DStack p_s){
    double * p = (double*) peekStack((*p_s).content);
    return (*p);
}
void pushDStack(double elt, DStack p_s){
    double * p = (double*) malloc(sizeof(double));
    (*p)=elt;
    pushStack(p,(*p_s).content);
}
double popDStack(DStack p_s){
    double * p = (double*) popStack((*p_s).content);
    double d = (*p);
    free(p);
    return d;
}
```

DOGGY BAG

TO TAKE AWAY ...

▷ Les principes de cohérence, encapsulation et généricité peuvent être mis en oeuvre en langage C au prix de certains efforts

▷ Une partie du code reste fortement redondant car en langage C un symbole non local ne peut pas être dupliqué

▷ Contrairement à d'autres langages, le langage C n'a pas de mécanisme de polymorphisme – du grec ancien polús (plusieurs) et morphê (forme) – qui est le concept consistant à fournir une interface unique à des entités pouvant avoir différents types

Questions?