Modularité

Il n'y a pas de classes, mais on peut obtenir la séparation entre l'interface et l'implémentation d'un type abstrait

Une instance d'une classe est remplacée par un pointeur sur une structure contenant les détails de l'implémentation

L'interface est une collection de fonctions dont l'un des paramètres (souvent le premier) est un tel pointeur

Les fonctions de l'interface sont déclarées (et non définies) dans un fichier d'inclusion (extension .h)

Les autres fonctions sont définies dans le .c avec le mot clé "static" afin de les cacher de l'extérieur

Pour les conteneurs on peut utiliser le pointeur générique (void *) pour les éléments

Les champs de la structure font partie de l'implémentation. Il faut donc déclarer la structure dans le fichier d'implémentation (extension .c)

Modularité (exemple)

stack.h

```
#ifndef STACK_H
#define STACK_H

struct stack;
typedef struct stack *stack;

/* create an empty stack */
extern stack stack_create(void);

/* destroy a (not necessarily empty) stack */
extern void stack_destroy(stack s);

/* return a true value iff the stack is empty */
extern int stack_empty(stack s);

/* push an element on top of the stack */
extern void stack_push(stack s, void *element);

/* pop the top element of the stack and return it */
extern void *stack_pop(stack s);

#endif
```

Modularité (exemple)

stack.h

```
#ifndef STACK_H
#define STACK_H

struct stack;
typedef struct stack *stack;

/* create an empty stack */
extern stack stack_create(void);

/* destroy a (not necessarily empy) stack */
extern void stack_destroy(stack s);

/* return a true value iff the stack is empty */
extern int stack_empty(stack s);

/* push an element on top of the stack */
extern void stack_push(stack s, void *element);

/* pop the top element of the stack and return it */
extern void *stack_pop(stack s);

#endif
```

éviter plusieurs inclusions du fichier dans un programm

il y a une struct stack, mais on ne connait pas les détail

le type principal manipulé sera un pointeur

expliquer le protocole avec des commentaires

pour indiquer zéro paramètres

expliciter que c'est l'interface

utilisation de la forme impérative du verbe pour avoir moins de te

mettre le nom des paramèt

utilisation du pointeur générique pour les éléments

éviter plusieurs inclusions du fichier dans un programme

Modularité (exemple)

```
#include "stack.h"
#include <stdlib.h>

static void **
grow_buffer(void **old_buffer, int old_size, int new_size)
{
  ...
}
...

struct stack
{
  ...
};

stack
stack_create(void)
{
  stack s = malloc(sizeof(struct stack));
  ...
  return s;
}
```

Modularité (exemple)

```
#include "stack.h"
#include <stdlib.h>

static void **
grow_buffer(void **old_buffer, int old_size, int new_size)
{
  ...
}
...

struct stack
{
  ...
};

stack
stack_create(void)
{
  stack s = malloc(sizeof(struct stack));
  ...
  return s;
}
...
```

inclusion du .h pour permettre la vérification de types

inclusion d'autres .h nécessaires

les fonctions internes sont déclarées "static"

le nom d'une fonction interne n'est pas préfixé

la déclaration complète de la structure utilisée pour l'implémentation du type abstrait

le nom de la fonction en première colonne

la taille de la structure à allouer

D’autres utilisations de la séparation .h .c

Attention à ne pas aller trop loin.

La règle n’est pas "ne jamais mettre la déclaration de la struct dans le .h", mais "pour un type abstrait, ne jamais...."

On peut utiliser le .h pour d’autres choses que pour les types abstraits par exemple pour éviter la duplication de code

Pour avoir un type abstrait, il faut une interface non ambiguë

Exemples de types non abstraits:

- **Liste chaînée (c’est un type d’implémentation, un type concret)**
- **Liste doublement chaînée, liste circulaire (même chose)**
- **Ensemble (souvent pas assez spécifique)**

Exemple de types abstraits: ensemble union-find, ensemble insert-member-delete

Gestion de la mémoire

La gestion manuelle de la mémoire est source d'erreurs (malloc(), free())

- **Corruption des structures de données**
- **Fuites de mémoire**

Ces problèmes sont possibles à corriger, mais difficilement

La gestion manuelle de la mémoire "casse les barrières d'abstraction"

Ce problème est impossible à corriger et rend l'application difficile à maintenir

Conseil: pour les applications dont la quantité de mémoire allouée pendant la durée de l'exécution est bornée et pas trop importante, ne pas utiliser free().

Conseil: pour les autres applications, utiliser un ramasse-miettes, soit intégrer le code avec un système comme RScheme ou Guile, soit utiliser le ramasse-miettes de Hans Boehm

Le ramasse-miettes de Hans Boehm

Un ramasse-miettes pour le langage C (et C++)

Peu de restrictions sur la programmation

Existe en version incrémentale pour Linux

Logiciel libre

Possible à intégrer dans un logiciel existant

Disponible par http://reality.sgi.com/employees/boehm_mti/

Remplacer malloc() par GC_malloc()

Linker avec gc.a

Situations exceptionnelles

Il y en a trois types:

Les échecs

Ce sont des situations dues aux défauts de programmation.

L'épuisement de ressources

C'est quand malloc() renvoie NULL ou quand le disque est plein.

Les autres

Ce sont des situations normales, prévues par le programmeur, mais dont le traitement nécessite une structure de contôle particulière

Les échecs

Il n'y rien à faire

Il faut arrêter l'exécution le plus vite possible

Avec la macro "assert", on peut parfois tester les échecs et arrêter l'exécution avec un message standard (fichier, ligne)

Il ne sert à rien de faire mieux, car on ne sait pas dans quel état se trouve le programme

La macro "assert" est aussi une bonne méthode pour documenter l'interface d'un type abstrait

À l'aide du compilateur, on peut supprimer les tests quand on est "sûr" qu'il n'y a plus de défauts

Il est souvent meiux de les laisser (sauf ceux qui coûtent cher)

L'épuisement de ressources

**Dans un programme non interactif, facile à relancer,
arrêter l'exécution avec un message destiné à l'utilisateur
(et non comme assert destiné au programmeur)**

**Dans un programme interactif, c'est un peu trop brutal.
Relancer la boucle d'intercation après un message indiquant
comment faire pour libérer des ressources**

**La programmation dans le cas interactif est très délicate. Il ne faut pas
que le message ou les commandes pour la libération de ressources
nécessitent de la ressource épuisée (printf peut appeler malloc, par exemple)**

Les autres

Ouverture d'un fichier qui n'existe pas, mais c'est normal

Parcour d'un tableau à la recherche d'un élément particulier, et on l'a trouvé

Tentative de reculer d'un caractère, alors que le point est au début du tampon

Multiplication d'une suite de nombres, et l'un des nombres est zéro

Etc...

Le problème des situations exceptionnelles

La situation est souvent détectée par du code de bas niveau

Mais c'est du code haut niveau qui possède la connaissance pour son traitement

Exemple (épuisement de ressources):

> **Le problème est détecté par malloc (très bas niveau)**
>
> **Mais malloc ne sais pas si l'application est interactif ou non (ce qui influence le traitement du problème)**

Exemple (ouverture de fichier non existant):

> **Le problème est détecté par open (très bas niveau)**
>
> **Si c'est un fichier d'installation c'est un défaut. Si c'est un fichier utilisateur c'est une situation de type "autre".**

Il faut donc communiquer le problème du bas niveau au haut niveau

Utilisation directe de setjmp/longjmp

interaction.c

```
#include <setjmp.h>

static jmp_buf errbuf;

void
fatal(char *message)
{
  fprintf(stderr, message);
  longjmp(errbuf);
}

void
cmd_loop(void)
{
  setjmp(errbuf);
  while(1)
    execute_command();
}
...
```

command.c

```
...
void
execute_command(void)
{
  ...
  advance_one_char();
  ...
}
...
```

charcommands.c

```
...
void
advance_one_char(void)
{
  if(beginning_of_buffer())
    fatal("begining of buffer");
  else
    point_location--;
  ...
}
...
```

Utilisation d'une bibliothèque d'exceptions

C'est une bibliothèque de plus haut niveau que setjmp/longjmp

Elle utilise setjmp/longjmp comme mécanisme de base

Il y en a plusieurs:

- **Celle écrite par J.-P. Braquelaire**
- **Celle publiée dans Software, Practice and Experience**