TD5 — Gestion d'erreurs, robustesse

Support enseignant

L'objet de cette séance est de montrer que, de la même manière qu'on ne peut jamais être sûr de la correction complète d'un programme (absence totale de bugs), il faut aussi prévoir autant que possible l'utilisation erronée d'un programme ou d'un module/paquetage.

L'exercice est donc :

1. *identifier et spécifier ce qu'est la «bonne utilisation» du programme/paquetage (préconditions des fonctions, format des entrées)*
2. *à partir de cette spécification (qui doit faire partie, au moins sous forme de commentaires, de la «spécification» du paquetage), identifier les utilisations «erronées», savoir ce que ces utilisations vont modifier du comportement du programme (comportement non attendu : lequel ? ou défaillance complète par arrêt du programme...)*
3. *savoir mettre en œuvre une méthode de gestion des erreurs*

On considère le paquetage `type_pile` du TD/TP n° 4, dont voici ci-dessous une implémentation.

```c
#include "type_pile.h"
#include <stdio.h>

/* Créer une pile vide */
void creer_pile(PileEntiers *p) { p->n = 0; }

/* Retourne vrai ssi p est vide */
int est_vide(PileEntiers *p) { return (p->n == 0); }

/* Renvoie l'entier en haut de la pile */
int sommet(PileEntiers *p) { return p->tab[p->n - 1]; }

/* Renvoie le nombre d'éléments dans la pile */
int taille(PileEntiers *p) { return p->n; }

/* Vider la pile p */
void vider(PileEntiers *p) { p->n = 0; }

/* Empiler un entier x */
void empiler(PileEntiers *p, int x) {
  p->tab[p->n] = x;
  p->n = p->n + 1;
}

/* Supprimer et renvoyer l'entier en haut de la pile */
int depiler(PileEntiers *p) {
  p->n = p->n - 1;
  return p->tab[p->n];
}
```

Exercice 1. Donner les préconditions requises pour l'utilisation de chaque fonction de ce paquetage.

— `creer_pile` *: le pointeur* `p` *doit être non nul*

NB : `p` *doit aussi pointer effectivement sur une structure de type* `PileEntier`*... Ce qu'il n'est pas possible de vérifier. On va supposer dans la suite que l'utilisateur ne fait que des erreurs «raisonnables». On peut mentionner le fait que la gestion des erreurs repose de toutes façons sur un «modèle de fautes/d'erreurs», explicite ou non : on ne gère en fait qu'un sous-ensemble d'erreurs possibles, sous-ensemble plus ou moins grand en fonction du niveau de* robustesse *nécessaire dans le contexte d'exécution prévu du programme.*

— *toutes les autres fonctions ne doivent être appelées avec* `p` *qu'une fois* `creer_pile(p)` *fait.*

— `sommet` *:* `p` *doit être non vide*

— `vider` *:* `p` *doit être non nul*

— `empiler` *:* `p` *non nul, et nombre d'éléments* $<$ `TAILLE_MAX`

— `depiler` *:* `p` *non nul, et pile non vide*

Exercice 2. Ces préconditions peuvent ne pas être satisfaites, dans le cas d'une mauvaise utilisation de ce paquetage. Quel sera alors le comportement des fonctions ?

— `creer_pile`, `vider`, `empiler`, `depiler` *: si* `p==NULL`*, l'opération de déréférencement (*`p->...` *ou* `*p`*) provoque l'arrêt du programme avec l'erreur «Segmentation fault» (bien le redire pour que les étudiants sachent d'où vient cette erreur...).*

— *si* `creer_pile` *n'est pas appelée au préalable : cela dépend du contenu initial de la structure* `PileEntier` *déclarée, or ce contenu est arbitraire en C (dépend de l'état de la zone mémoire allouée au moment de la déclaration)...*

— `sommet(p)` *si* `p` *est vide : accès à l'élément d'indice* -1 *du tableau* `p.tab`*.*

En C : pas de vérification dynamique de la valeur des indices, pas d'erreur systématique (dans la plupart des autres langages : erreur sous la forme d'une exception*). Si la zone mémoire adjacente est aussi allouée au programme (déclarations adjacentes par exemple), la valeur retournée sera celle contenue dans cette zone mémoire. (En l'occurrence : dans notre cas, la zone mémoire adjacente au tableau est celle allouée pour* `p.n` *: sommet(p) renverra donc 0)*

— `empiler(p)` *si la pile est pleine (*`p->n == TAILLE_MAX`*) : modification d'un élément en-dehors du tableau (à l'indice* `TAILLE_MAX`*). Idem : dépend de l'état de la mémoire.*

— `depiler(p)` *si la pile est vide :* `p->n` *devient négatif. Le prochain accès au tableau* `p->tab` *se fera en-dehors de l'intervalle des indices.*

Exercice 3. Modifiez le paquetage pour permettre la gestion des erreurs.

À minima : impression d'un message d'erreur, par exemple :

```
int sommet(PileEntiers * p) {
  if (est_vide(p)) {
    printf("ERREUR : fonction sommet, p est vide\n");
  } else {
    return p->tab[p->n-1];
  }
}
```

Sauf que :

- *la fonction doit quand même retourner une valeur : le compilateur va éventuellement avertir le programmeur (possibilité d'absence d'exécution d'une instruction* `return` *dans la fonction* `sommet`*...) ; et la valeur retournée en cas d'erreur est arbitraire ;*
- *pour des raisons de* séparation des préoccupations*, il vaut mieux permettre de gérer les erreurs à l'extérieur du paquetage : en cas d'erreur, ce qu'on souhaite n'est pas forcément d'afficher un message, en tout cas pas forcément sur la sortie standard ou le terminal.*

Il faut donc un mécanisme qui permet :

- *de prévenir l'utilisateur du paquetage de la présence d'une erreur ;*
- *de permettre à cet utilisateur de pouvoir* identifier *l'erreur (code d'erreur, éventuellement un message standard associé...).*

NB : dans des langages de programmation un peu plus évolués que le C, ce sont les exceptions*, qui permettent de contourner l'utilisation standard de fonctions.*

En C : on peut par exemple modifier le profil des fonctions : on met le résultat normal en paramètre, et la valeur retournée indique si une erreur s'est produite ou non (et éventuellement, indique quelle *erreur s'est produite). Il faut évidemment ajouter cette gestion d'erreur à la spécification !*

```
typedef enum { OK, ERREUR_PILE_VIDE, ERREUR_PILE_PLEINE } TypeErreur;

/* Renvoie l'entier en haut de la pile.
Précondition : p est non vide.
Si la fonction s'exécute correctement, resultat contient l'entier en
haut de la pile p, et la valeur OK est retournée.
Si la fonction est appelée avec une pile p vide, la valeur de resultat
n'est pas spécifiée, et la fonction retourne la valeur
ERREUR_PILE_VIDE. */
TypeErreur sommet(PileEntiers * p, int * resultat) {
  if (!est_vide(p)) {
    *resultat = p->tab[p->n-1];
    return OK;
  } else {
    return ERREUR_PILE_VIDE;
}
```

De même pour les autres fonctions...

On peut enfin donner des exemples d'utilisation du paquetage ainsi modifié.