KOOC - 4ième Partie

Sommaire

1) Traitement des erreurs : exceptions.

a) Traitement des erreurs dans un programme classique.

A l'origine, les programmes ne sont qu'une tentative de traduction en langage informatique d'un problème mathématique. Ils ont donc la prétention d'être sans défaut si le problème est bien posé (preuve). Laissons de côté les bugs logiques qui sont des erreurs de conception. Un programme s'exécute sur une machine et répond à des contraintes physiques, dispose de ressources limitées (dans le sens finies), interagit avec un utilisateur (aux réactions imprévues). Il n'est donc pas certain que dans un programme toutes les fonctions puissent être exécutées comme le concepteur l'avait prévu. Les fonctions peuvent échouer sans avoir réalisé l'ensemble des tâches qui leur sont dévolues. Le fil d'exécution est dérouté vers une voie de garage.

Le terme 'dérouté' est bien choisi, car en pratique et bien avant l'avènement de la programmation dite 'structurée', les programmes (souvent en assembleur) étaient conçus telles des routes, par pavés successifs de blocs de code assurant l'exécution d'une tâche. Le programme s'exécutait de manière implacable. Lorsqu'une erreur système survenait, le processeur arrêtait la voiture (le pointeur de programme) et prenait une 'déviation' (changement de sa valeur via une instruction de saut 'jmp') pour exécuter une procédure d'arrêt du système, ou de diagnostique (core dump).

Au début, sur ces ordinausaures les seules fonctions systèmes étaient mathématiques (Colossus). Et les premières erreurs, des problèmes de débordements pendant les calculs. Donc on a appris à tester par programmation

(valeur de registre d'état) ces cas de débordements et à appeler des fonctions de traitement d'erreurs. Ensuite quand l'interaction avec les périphériques est apparue, on a gardé l'habitude de tester les cas critiques par retour de valeur.

b) Cascade de if versus exceptions

Avec l'avènement de la programmation structurée (apparition de langage avec: if, while, fonctions), on imposa par des règles de programmation la résolution et la gestion d'erreurs. Que ce soit par un paramètre en mise-à-jours ou par une valeur de retour, chaque fonction devait pouvoir être auditée sur sa bonne exécution. Cette règle perdure encore dans les langages tels que le C. Bon nombre d'API ont adopté la règle de la valeur de retour pour tester la bonne exécution de la fonction.

Ex: API win32, Appel système Linux.

L'avantage d'une règle de programmation telle que celle précédemment citée, c'est qu'elle est simple à mettre en place et ne change absolument rien dans la définition d'un langage.

Cependant, on pourra trouver 2 effets pervers à cette règle :

- Les erreurs peuvent passer inaperçues Dans beaucoup de langages et surtout en C, on peut ignorer la valeur de retour. Par mauvaise habitude et par fainéantise, on ne teste pas systématiquement le retour des appels systèmes pour savoir si tout s'est bien passé.
- Lourdeur de programmation. À l'opposé, si l'on veut tenir compte de toutes les erreurs, il va falloir tout tester. Sur un programme simple, cela n'est pas gênant, mais pour un produit industriel cela se relève extrêmement fastidieux.

Pour contrer cette lourdeur et changer de technique, le principe d'exception s'est imposé. Il s'agit d'introduire de nouvelles structures de contrôles masquant quelques opérations basiques sur la gestion des contextes de fonctions pour ne 'dérouter' un programme que lorsque cela est nécessaire. On parle de mécanisme d'exception (j'espère que pour vous une erreur est par nature "exceptionnelle", c'est-à-dire "rare", contrairement au basic "users" complètement habitués aux erreurs Windows et convaincu que l'informatique ne marche pas...ou si peu... cf. parent ou grand-parent). Il s'agit généralement de :

- throw (lance) Qui permet concrètement de générer une exception.
- try (essaie) Pour délimiter un nouveau bloc de code susceptible de générer des exceptions.

- catch (attrape, sous-entendu ce qui a été lancé) Ce qui permet d'associer pour chaque type d'exception un bloc de code à exécuter quand les erreurs surviennent.

c) Contexte de fonction

Pour réaliser un mécanisme d'exception, il faut apprendre à manipuler les contextes d'exécution de fonction. Pour cela, 2 fonctions de la libC existent : **setjmp** et **longjmp**. D'abord, faisons un petit rappel sur ce qui se passe quand on appelle une fonction.

Premièrement, le processeur possède un registre compteur de programme (PC : program counter ou IP: Instruction Pointer) qui indique l'adresse mémoire de l'instruction à exécuter. Ensuite il possède un registre indiquant une zone particulière de la mémoire du processus qu'il est en train d'exécuter. Ce pointeur, c'est le pointeur de pile (SP: Stack Pointer). Bien que l'on parle de pile, on représente celle-ci la tête en bas. Car lors d'une instruction d'empilement la valeur du registre décroît. Pourquoi? Dans les vieilles architectures à mémoire flat (modèle plat) cela permettait de loger facilement toutes ces différentes zones de mémoire dans le même espace de mémoire contiguë. En bas, le code. Ensuite le segment de données. Ensuite le tas (heap) pour la mémoire allouée dynamiquement. Et tout en haut la pile. Avec son pointeur qui descend vers le tas. Cette structure très simple est restée.

A quoi sert la pile? En fait cette organisation mémoire a été mise au point pour gérer lors du passage à la programmation structurée, la zone pour contenir le contexte d'appel et les variables locales.

Comment cela marche ?

Dans un premier temps, voyons comment le compilateur assemble un appel de fonction.

```
function(p1, p2, p3, p4);
```

Il évalue la valeur réelle de p1, p2, p3, p4. Ensuite il empile dans un certain ordre p1, p2, p3, p4. C'est-à-dire qu'il les recopie sur la pile. Cet ordre dépend de la convention d'appel utilisée par le langage de compilation. En pascal ou en ada, il va le faire dans l'ordre p1 puis p2 etc.. En C, dans l'ordre inverse, d'abord p4. Pourquoi? Le "C" est le seul langage gérant un nombre variable d'arguments (du coups le C++ aussi) et cette convention est la seule façon d'implémenter cela. En effet, p1 le premier paramètre est toujours accessible quel que soit le nombre de paramètres optionnels (son adresse est toujours la même par rapport au haut de la pile pendant l'exécution d'un bloc de code). Cela permet par exemple dans printf de se baser sur ce premier paramètre pour savoir combien de paramètres ont été empilés.

Ensuite le processeur exécute une instruction de saut vers la fonction : un 'call'. Cette instruction va sauver sur la pile l'adresse de l'instruction suivante, puis saute à l'adresse indiquée en tant qu'opérande de l'instruction.

On se retrouve donc en début de fonction avec un certain nombre de paramètres sur la pile, suivi de l'adresse de retour. Maintenant, il faut réfléchir au code de la fonction. Cette fonction va utiliser les paramètres formels. Comme le compilateur ne peut pas connaître leur adresse réelle en mémoire, il va devoir utiliser un registre autre que celui de la pile, pour qu'à n'importe quelle instruction de la fonction l'on puisse accéder aux paramètres via ce registre (BP: Base Pointer). Après avoir chargé celui-ci de la valeur du registre de pile, le compilateur pourra l'utiliser pour accéder à tous les paramètres. Il peut ensuite réserver un espace dans la pile pour y stocker les variables locales (par simple soustraction de SP). Après tout ceci, le code spécifique à la fonction va être exécuté. À la fin, la fonction restituera la valeur du registre utilisée comme index et chargera la valeur de l'adresse de retour dans le compteur de programme.

L'ensemble des registres utilisés (IP, SP, BP) constitue le contexte de la fonction. Leurs valeurs sont sauvées ainsi que la valeur des registres courants dans une structure en utilisant la fonction '**setjmp**'.

2) Exception en KOOC

a) setjmp et longjmp

Synopsis:

```
#include <setjmp.h>

int setjmp (jmp_buf env);
void longjmp (jmp_buf env, int val);
```

La fonction '**setjmp**' va nous servir à sauver le contexte d'exécution d'une fonction et '**longjmp**' à retourner au point précédemment sauvé. Pour plus d'infos sur leur utilisation : man **setjmp** et man **longjmp**. Cependant, voici un exemple de l'utilisation qui peut en être faite.

```
#include <stdio.h>
#include <setjmp.h>
#include <math.h> /* a compiler avec -lm */

jmp_buf         env;

#define         NEG     1
#define         NE      2
#define         CRIT    3

double          root(double x)
```

```
{
  if (x < 0)
    longjmp(env, NEG);
  return (sqrt(x));
}

void            print(char *s)
{
  if (s == NULL)
    longjmp(env, NE);
  printf("%s", s);
}

void            level2()
{
  jmp_buf       local;
  int           ret;

  /* sauve le contexte appelant */
  memcpy(&local, &env, sizeof(jmp_buf));
  if (!(ret = setjmp(env)))
    {
      print("coucou\n");
      print(NULL);
    }
  else
    {
      /* restitue le contexte appelant */
      memcpy(&env, &local, sizeof(jmp_buf));
      switch (ret)
        {
        case NE:
          printf("NULL exception\n");
          longjmp(env, CRIT);
          break;
        default:
          /* je connais pas je fait suivre */
          longjmp(env, ret);
        }
    }
}

void    level1()
{
  jmp_buf       local;
  int           ret;

  /* sauve le contexte appelant */
  memcpy(&local, &env, sizeof(jmp_buf));
  if (!(ret = setjmp(env)))
    {
      double r;

      r = root(4);
      printf("racine de 4 :%d\n", (int)r);
```

```
      r = root(-1);
      printf("racine de -1 :%d\n", (int)r);
      r = root(9);
      printf("racine de 9 :%d\n", (int)r);
    }
  else
    {
      /* restitue le contexte appelant */
      memcpy(&env, &local, sizeof(jmp_buf));
      switch (ret)
        {
        case NEG:
          printf("NEG exception\n");
          break;
        default:
          /* je connais pas je fait suivre */
          longjmp(env, ret);
        }
    }
}

int     main(int ac, char **av)
{
 int   ret;

  if (!(ret = setjmp(env)))
    {
      level1();
      level2();
    }
  else
    {
      switch (ret)
        {
        case CRIT:
          printf("CRITICAL exception: abort program\n");
        default:
          printf("UNKNOWN exception: abort program\n");
        }
      exit(1);
    }
}
```

Lors de l'appel à **longjmp** les variables locales sont détruites. Vous pouvez le mettre en évidence en utilisant **__attribute__((cleanup(....))**, qui est une extension C offerte par GCC, permettant d'appeler la fonction passée en paramètre lors de la destruction du cadre de pile contenant la variable. La fonction prend un paramètre du même type que la variable à un niveau d'indirection en plus (+ un niveau de pointeur).

Comme ci-dessous :

```
void aSingleDestructor(char ** s)
{
      printf("the String variable at %p containing %s was destructed.\n",
      s, *s);
      free(*s);
}

void    level1()
{
  jmp_buf       local;
  char *    aLocalVariable __attribute__((cleanup(aSingleDestructor)))
            = strdup("Ceci est une donnee locale");
  int           ret;
  ....
}
```

Sur ces bases vous pouvez imaginer l'ajout de **@try**, **@catch** et **@throw** à KOOC. La syntaxe est spécifiée dans le sujet mais, vous pouvez sur ces bases créer votre propre mécanisme.

b) Syntaxe

Voici la syntaxe imposée :

```
@class FileNotFoundException : Exception
{
@member {
      const char *msg;
      void  init(const char *);
}

}

@implementation FileNotFoundException
{
@member void  init(const char *msg)
{
      [self.msg] = msg;
}

}

FILE  *OpenRead(char *filename)
{
  FILE      *f;

  f = fopen(filename, "r");
  if (!f)
      @throw (FileNotFoundException*)("Fucking bastard!");
  return (f);
```

```
}

int   main(int ac, char *av)
{
  @try
  {
      OpenRead("SomeFile");
  }
  @catch (FileNotFoundException *ex)
  {
      printf("%s\n", [ex.msg]);
      exit(1);
  }
}
```

Le mot-clef **@throw** va créer automatiquement un objet de type **FileNotFoundException** sur le tas (heap). Dans le catch, cet objet **Exception** sera accessible et doit être automatiquement détruit à la fin du catch. Toutes les exceptions dérivent d'une interface unique **Exception** définit par le runtime. Par défaut cette interface n'est juste qu'une interface de convention. Elle est vide et sert d'ancêtre commun à toutes les exceptions. Libre à vous de lui ajouter d'autres fonctionnalités.