Du `setjmp` et du `longjmp`

Bernard Cassagne
Daniel Veillard

1 La guerre du `goto` aura-t-elle lieu?

Après l'invention des langages de programmation de haut niveau (style ALGOL, PASCAL, etc...) les programmeurs se sont aperçus que presque tous les algorithmes pouvaient avoir leurs besoins de branchements satisfaits grâce aux instructions structurées (`for`, `while`, `repeat`, etc...) sans utiliser le branchement non structuré (`goto`).

Comme l'expérience de la programmation avait montré que l'usage des `goto` rendait les programmes difficilement lisibles, certains sont allés jusqu'à proposer la suppression pure et simple des branchements non structurés.

S'ensuivit alors une période où la controverse fit rage entre "puristes" (suppression totale des `goto`), et "pragmatiques" (ça peut servir). Au cours de cette période, fut mis en évidence l'existence de quelques rares algorithmes qui ne pouvaient être programmés avec les seules instructions structurées classiques.

Aujourd'hui, il semble que l'on s'accorde sur les points suivants:

1. il est raisonnable d'inclure une instruction de branchement non structuré dans un langage de haut niveau, quitte à mettre des restrictions à son usage.

2. les programmeurs sont incités à utiliser systématiquement les instructions structurées, et à n'utiliser le `goto` qu'en dernier recours, dans les quelques cas où il est impossible de faire autrement.

2 Le cas du langage C

Le langage C possède bien une instruction `goto`, mais la destination de ce `goto` doit être à l'intérieur de la fonction où se trouve l'instruction. On peut donc écrire:

```
f()
{
...          /*   des instructions   */
ici :
```

```
...          /*   des instructions   */
goto ici;
...          /*   des instructions   */

}
```

Une telle instruction `goto` dont la destination est dans la même fonction est dite *goto local*. Une instruction `goto` dont la destination est dans une autre fonction est dite *goto non local* et est interdite en langage C. On ne peut donc pas écrire:

```
f()
{
...          /*   des instructions   */
ici:
...          /*   des instructions   */
g();
...          /*   des instructions   */
}

g()
{
...          /*   des instructions   */
goto ici;
...          /*   des instructions   */
}
```

3 Utilité des instructions de branchement

Il exite un certain nombre de situations où l'utilisation des branchements est appréciable. L'exemple classique est le traitement des erreurs. Imaginons un programme qui lit des commandes tapées par un opérateur humain. Aprés avoir lu une commande, le programme effectue un certain traitement. Pour cela il appelle une procédure `f`, qui elle-même appelle une procédure `g`. Au beau milieu de son traitement, `g` peut se rendre compte qu'il y a une erreur dans la commande tapée par l'opérateur. On a donc envie de programmer de la manière suivante:

```
main()
{
while (non-fini)
   {
   lect:
   lire une commande
   f(cette commande)
   }
}
```

```
f(cmd)
{
...
g(cmd);
...
}

g(cmd)
{
...
si certaine condition(cmd)
   {
   emission d'un message d'erreur
   goto lect;
   }
...    /*   poursuite du traitement   */
}
```

On voit que le programmeur a envie d'utiliser un *goto non local*. Si on ne dispose pas de cette facilité, la programmation devient plus pénible. On peut procéder de la manière suivante:

```
main()
{
while (non-fini)
   {
   lect:
   lire une commande
   f(cette commande)
   }
}

f(cmd)
{
...
if (g(cmd) == ERREUR)
   return(ERREUR);
...
}
g(cmd)
...
si certaine condition(cmd)
   {
   emission d'un message d'erreur
   return(ERREUR);
   }
...    /*   poursuite du traitement   */
```

```
}
```

On voit que le programmeur doit programmer explicitement toutes les fins prématurées des procédures que le `goto` réalise d'un coup.

4 Les branchements non locaux dans le langage C

4.1 L'idée de base

On vient de voir que les branchements non locaux sont interdits dans le langage C, mais que les programmeurs ont besoin de cette possibilité dans un certain nombre de circonstances. Pour résoudre ce problème, il est traditionnel dans le monde UNIX, de disposer de 2 fonctions de la librairie standard `setjmp` et `longjmp` permettant de réaliser des goto non locaux. La fonction `setjmp` a pour but de mémoriser l'état du programme, et `longjmp` a pour but de forcer l'état du programme à un état précédemment mémorisé.

Dans `setjmp.h` il y a la définition du type `jmp_buf`. Ce type est un type opaque au programmeur (il n'a pas à savoir sa définition), il doit lui servir à déclarer des variables permettant de mémoriser l'état du programme.

Une utilisation de `setjmp` se programme de la manière suivante:

```
#include <setjmp.h>
jmp_buf env;

setjmp(env);
```

Lorsque le programmeur voudra restaurer l'état de son programme à l'état mémorisé dans `env`, il lui suffit d'écrire:

```
longjmp(env,1);
```

(Nous ignorerons pour l'instant le deuxième paramètre de `longjmp`).

4.2 Un premier exemple

Voici un petit exemple qui met en évidence ce que nous venons de voir.

```
#include <setjmp.h>
#include <stdio.h>
jmp_buf env;
/******************************************************************************/
/*                                                                            */
/*   But: Procedure recursive qui finira par faire un longjmp                 */
/*                                                                            */
/******************************************************************************/
f()
{
#define LGBUF 2
char buffer[LGBUF];
```

```
printf("Empiler encore un coup?(y/n) ");
fflush(stdout);
read(0,buffer,LGBUF);
if (buffer[0] == 'y')
   f();
else longjmp(env,1);
}

/*****************************************************************************/
/*                                                                           */
/*                                  main                                     */
/*                                                                           */
/*****************************************************************************/
main()
{
setjmp(env);
printf("Je suis dans le main\n");
f();
}
```

Et voici un exemple d'activation :

```
isis(211) essai1
Je suis dans le main
Empiler encore un coup?(y/n) y
Empiler encore un coup?(y/n) y
Empiler encore un coup?(y/n) n
Je suis dans le main
Empiler encore un coup?(y/n) n
Je suis dans le main
Empiler encore un coup?(y/n) y
Empiler encore un coup?(y/n) y
Empiler encore un coup?(y/n) y
Empiler encore un coup?(y/n) y
Empiler encore un coup?(y/n) n
Je suis dans le main
Empiler encore un coup?(y/n) ^C
isis(212)
```

4.3 Les limitations

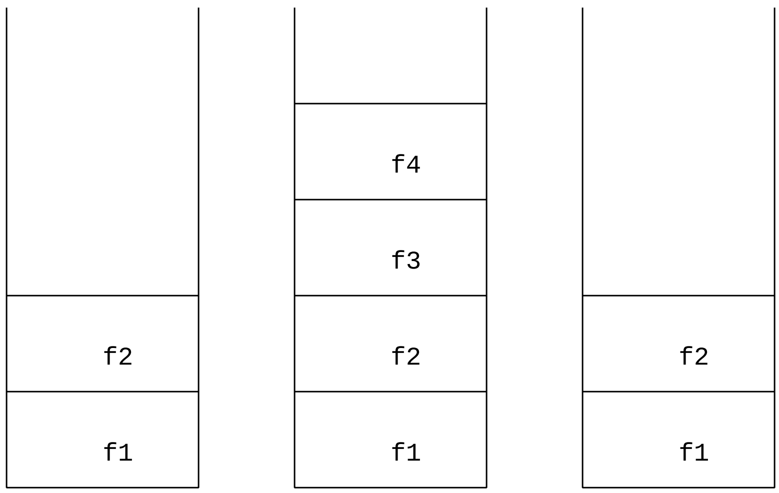
Le `longjmp` ne permet pas de revenir à n'importe quel état précédemment mémorisé grâce à un `setjmp`. La contrainte est la suivante : on ne peut faire un `longjmp` vers un état mémorisé dans un `jmp_buf env` que si la fonction qui a exécuté le `setjmp(env)` n'est pas terminée.

Cela vient du fait que `setjmp` ne mémorise pas l'état de la pile, mais seulement l'endroit où on se trouve dans la pile. En pratique, cela signifie que `setjmp` procède à une sauvegarde des registres qui pointent sur la pile. Le contenu de la pile quand à lui, n'est pas mémorisé. Cela permet de réaliser un (`setjmp`,`longjmp`) efficace et non coûteux en mémoire mais introduit des limitations à son usage.

Nous allons éclairer ceci à l'aide de deux exemples.

Un `longjmp` valide

Voici tout d'abord un exemple de `longjmp` valide : dans le schéma ci-dessous, on voit 3 états de la pile à 3 moments différents de l'exécution d'un programme.



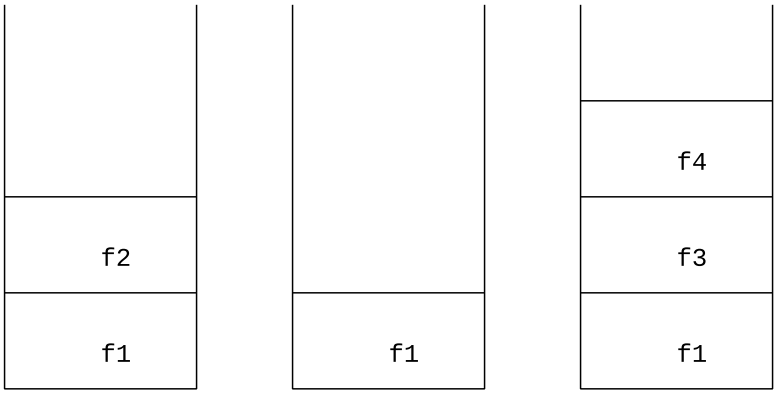
Etat 1 la fonction f1 a appellé f2 qui a exécuté un `setjmp`.

Etat 2 f2 a appellé f3 qui a appellé f4. f4 va exécuter un `longjmp`, l'état de la pile est donné juste avant cette exécution.

Etat 3 après exécution du `longjmp`, on se retrouve dans la fonction f2.

Un `longjmp` incorrect

Voici maintenant un exemple d'utilisation non valide de `longjmp`, là aussi, 3 états successifs de la pile sont donnés.



Etat 1 la fonction f1 a appellé f2 qui a exécuté un `setjmp`.

Etat 2 f2 s'est terminée.

Etat 3 f1 a ensuite appellé f3 qui a appelé f4. Si a ce moment on exécute un `longjmp`, on voit qu'il en résultera une erreur, car dans la pile, l'environ pour f2 a été écrasé par celui de f3.

4.4 Le `setjmp` et le `longjmp` à la loupe

Les bases du mécanisme étant acquises, il passons maintenant aux subtilités. Deux choses ont été passées sous silence : `setjmp` rend une valeur, et `longjmp` possède un autre paramètre, en plus de la variable de type `jmp_buf`.

La fonction `setjmp` rend une valeur, et l'état mémorisé est l'état du programme en train d'exécuter `setjmp`, juste avant de retourner cette valeur à l'appelant. Un `longjmp` remettra le programme dans cet état : juste avant le retour du `setjmp`. Il y a donc deux façons de sortir de `setjmp` : un retour normal, et un retour provoqué par un `longjmp`. Un programme entre donc une fois dans un `setjmp`, mais on en sort plusieurs fois!

Le deuxième paramètre de `longjmp` est la valeur que l'on désire faire retourner à `setjmp`, lorsque ce retour est provoqué par un `longjmp`.

Vous n'avez rien compris? Pas de panique, voici un exemple qui va vous éclairer.

```
#include <setjmp.h>
#include <stdio.h>
jmp_buf env;
/******************************************************************************/
/*                                                                            */
/*                                main                                        */
/*                                                                            */
/******************************************************************************/
main()
{
```

```
int ret;
int nb;

ret = setjmp(env);
if (ret == 0)
   printf("Retour normal de setjmp\n");
else printf("Retour de setjmp provoque par un longjmp(env,%d)\n",ret);
printf("Tapez un chiffre different de 0 : ");
fflush(stdout);
scanf("%d",&nb);
longjmp(env,nb);
}
```

Et voici une exécution de ce programme :

```
isis(138) essai2
Retour normal de setjmp
Tapez un chiffre different de 0 : 1
Retour de setjmp provoque par un longjmp(env,1)
Tapez un chiffre different de 0 : 2
Retour de setjmp provoque par un longjmp(env,2)
Tapez un chiffre different de 0 : ^C
isis(139)
```

5 Quelques cas de programmation classique

5.1 Traitement des erreurs

Voici un exemple:

```
#include <setjmp.h>
#include <stdio.h>
jmp_buf env;
/*****************************************************************************/
/*                                  f                                        */
/*****************************************************************************/
f(nb)
{
if (nb < 1000)
   longjmp(env,1);
printf("Execution correcte de f\n");
}

/*****************************************************************************/
/*                                  main                                     */
/*****************************************************************************/
main()
{
```

```
int nb;

while(1)
   {
   if (setjmp(env))
      printf("Erreur sur le nombre\n");
   printf("Entrez un nombre : ");
   fflush(stdout);
   scanf("%d",&nb);
   f(nb);
   }
}
```

Et un exemple d'exécution:

```
isis(216) essai3
Entrez un nombre : 2
Erreur sur le nombre
Entrez un nombre : 2000
Execution correcte de f
Entrez un nombre : ^C
isis(217)
```

5.2 Traitement des signaux

Nous avons vu que le programmeur peut désirer restaurer son programme à un état antérieur suite à la rencontre d'une erreur. On peut désirer faire la même chose sur un évènement asynchrone à l'exécution du programme, par exemple un signal envoyé par l'opérateur. Il arrive en effet, que l'on désire permettre à l'opérateur d'aborter une action en cours du programme, sans aborter le programme tout entier. Ceci se réalise généralement par un `longjmp` dans la routine associée au signal, comme dans l'exemple ci-dessous :

```
#include <setjmp.h>
#include <stdio.h>
#include <signal.h>
jmp_buf env;
int envset;
/****************************************************************************/
/*                              SigIntHandler                               */
/****************************************************************************/
SigIntHandler()
{
if (envset) longjmp(env,1);
}

/****************************************************************************/
/*                                    f                                     */
```

```
/****************************************************************************/
f(nb)
{
while (nb-- > 0)
   {
   printf(".");
   fflush(stdout);
   sleep(1);
   }
printf("\n");
}

/****************************************************************************/
/*                                                                          */
/*                               main                                       */
/*                                                                          */
/****************************************************************************/
main()
{
int nb;

envset=0;
signal(SIGINT,SigIntHandler);

while(1)
   {
   if (setjmp(env) == 0)
      envset = 1;
   else  printf("\n   Commande abortee\n");
   printf("Tapez un nombre : ");
   fflush(stdout);
   scanf("%d",&nb);
   f(nb);
   }
}
```

Et une exécution de ce programme :

```
isis(236) essai4
Tapez un nombre : 3
...
Tapez un nombre : 1000
.........................................^C
   Commande abortee
Tapez un nombre :
```

Exercice

Dans une première version de ce document, l'auteur avait écrit le programme de la manière suivante :

```
SigIntHandler()
{
longjmp(env,1);
}
```

Dans le `main`, la boucle était progammée comme ceci :

```
while(1)
   {
   if (setjmp(env))
      printf("\n   Commande abortee\n");
   printf("Tapez un nombre : ");
   fflush(stdout);
   scanf("%d",&nb);
   f(nb);
   }
```

Le relecteur de ce document a vu que cette version est incorrecte. Exercice (facile) : quelle séquence d'évènements pouvait faire planter la première version du programme ?