L3 MI — Programmation

Pierre Karpman
pierre.karpman@univ-grenoble-alpes.fr
https://membres-ljk.imag.fr/Pierre.Karpman/tea.html

2022–11–23

Instruction de saut : `goto`

Fonctions de sauts non locaux

Sauts (in)conditionnels

- Les changements de flot de contrôle (`if`, `for`...) sont implémentés en machine par des *sauts*
  - Conditionnels (par ex. en x86 : `JE`, `JNE`, `JNAE`, `JA`...)
  - Non conditionnels (par ex. en x86 : `JMP`)
- En C, les sauts non conditionnels sont accessibles via l'instruction `goto`
- Syntaxe : `goto label;`, où `label` indique une instruction *de la même fonction*
- Un label est déclaré avec la syntaxe `identifiant:` en début de ligne ; il n'a aucune influence directe sur le flot de contrôle

« Mauvaise » utilisation de goto

Une simple boucle....

```
        int i = 0, acc = 0;
A:
        acc += i++;
        if (i < 20)
                goto A;
```

Alternative moins lisible à :

```
        int i = 0, acc = 0;
        do
        {
                acc += i++;
        } while (i < 20);
```

Le code compilé est (essentiellement) identique dans les deux cas

GO TO statement considered harmful

- L'utilisation de `goto` à la place de blocs structurés est découragée
- Opinion popularisée par Dijkstra (1968)
- Instruction généralement non présente dans les langages haut niveau

« Bonnes » utilisations de `goto`

Les instructions de saut restent utiles dans certains cas, par ex. :

- Pour défaire proprement des initialisations (par ex. courant dans le noyau Linux), ou éviter la duplication de code de gestion d'erreurs
- Pour échapper plusieurs blocs en une fois (`break` « généralisé »)
- Pour éviter une pile d'appels récursifs importante, par ex. en *backtracking*

Exemple 1

Contexte :

- Une fonction qui doit allouer trois ressources via `allocA`, `allocB`, `allocC`
- Doit les désallouer à la fin de l'exécution via `freeA`, `freeB`, `freeC`
- Les allocations peuvent échouer ; dans ce cas, les ressources potentiellement déjà allouées doivent être libérées

Problématique classique en programmation bas niveau

Exemple 1 (cont.)

Une solution à base de `goto` :

```
        ...
        ok = allocA(...);
        if (!ok)
                goto failA;
        ok = allocB(...);
        if (!ok)
                goto failB;
        ok = allocC(...);
        if (!ok)
                goto failC;
        ...
        free(C);
failC:
        free(B);
failB:
        free(A);
failA:
        return ok;
```

- Sauts unidirectionnels ; le flot des `goto` est simple

Exemple 2

Contexte :

- Une fonction doit vérifier plusieurs causes possibles d'erreur à l'exécution
- En cas d'erreur, le traitement à appliquer est identique (ou très proche)
- Les vérifications peuvent être à une profondeur importante relativement au code de traitement
  - L'utilisation de `break` n'est pas (forcément) possible
  - On veut éviter la duplication du code de gestion

Exemple 2 (cont.)

Extrait de la bibliothèque SSL/TLS d'Apple (via wikipedia) :

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
         SSLBuffer signedParams, uint8_t *signature, UInt16 signatureLen) {
    OSStatus        err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err; }
```

Exercice : quel problème grave se trouve dans ce code ? Comment le résoudre ?

Instruction de saut : `goto`

Fonctions de sauts non locaux

Les fonctions `setjmp`, `longjmp`...

Disponibles dans `"setjmp.h"` :

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

Comportement :

- `setjmp` sauvegarde l'environnement courant (c-à-d l'état des registres et de la pile) dans `env` (un tableau d'`int`) et retourne `0`
- `longjmp` saute au `setjmp` ayant fourni `env`, et fait retourner `val` à celui-ci
- Ces appels peuvent avoir lieu dans des fonctions différentes (contrairement à un `goto` dont le saut est local)
- Mais l'appel à `longjmp` ne doit pas se faire après que la fonction ayant appelé `setjmp` a retourné (car alors son environnement n'≪ existe plus ≫)

Exemple

Qu'affiche le programme suivant ?

```
void fun2(jmp_buf e)
{
        longjmp(e, 1);
        printf("...full of Mustelidae\n");
}
void fun(void)
{
        jmp_buf e;
        if (setjmp(e))
        {
                printf("a good world\n");
                return;
        }
        printf("This is ");
        fun2(e);
}
int main()
{
        fun();
        return 0;
}
```

Utilité de `longjmp` *et al.*

- `longjmp` peut être utile pour par ex. implémenter une gestion d'erreurs dans une pile d'appel avec un gestionnaire d'erreur unique à haut niveau
- Aussi proche de la notion de *continuation* et des *callbacks* en programmation asynchrone
- Mais peu utile dans la plupart des (autres) contextes