

# L3 MI — Programmation

Pierre Karpman

`pierre.karpman@univ-grenoble-alpes.fr`

`https://www-ljk.imag.fr/membres/Pierre.Karpman/tea.html`

2019-11-20

Instruction de saut : `goto`

Application (très) rapide au backtracking

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

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;
    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`

Application (très) rapide au backtracking

Fonctions de sauts non locaux

Le *backtracking* est une stratégie générale d'exploration d'espaces de recherche d'une certaine forme

- ▶ Appropriée par ex. pour trouver un assignement de  $N$  variables satisfaisant un ensemble de contraintes
  - ▶ Exemple :  $(x, y, z) \in \mathbb{B}$  t.q.  $(x \vee y) \wedge (x \vee \neg z)$
- ▶ ... s'il est facile d'éliminer certaines solutions partielles
  - ▶ Exemple :  $x = y = \perp$  ci-dessus
- ▶ L'espace de recherche peut être vu comme un arbre dont on élague les sous-arbres contenant uniquement des absences de solutions
- ▶ On espère en moyenne ne pas avoir à parcourir *tout* l'arbre avant de trouver une solution ou prouver une impossibilité

## Exemple : résolution de SUDOKU

---

Résoudre un SUDOKU revient à trouver  $(x_1, \dots, x_{81}) \in \llbracket 1, 9 \rrbracket^{81}$  vérifiant un certain nombre de contraintes

- ▶ Une solution partielle est facilement testable
  - ▶ Les variables assignées doivent vérifier les contraintes ; les variables non assignées doivent avoir au moins une assignation possible respectant les contraintes
- ▶ Elle peut facilement être étendue
  - ▶ Affecter une valeur admissible à une variable non assignée
- ▶ En cas de mauvais choix, on « backtrack »
  - ▶ On fait un autre choix pour la dernière variable affectée, ou l'avant-dernière, ou...

- ▶ Une résolution par backtracking peut s'écrire naturellement de façon récursive (terminale)
- ▶ Mais la pile risque d'exploser si le compilateur n'optimise pas la récursion
- ▶ Celle-ci peut être déroulée explicitement avec des `goto`

Exemple de structure :

- ▶ Bloc **A**: propagation de contraintes ; déclenchement de backtracking (`goto C;`) ou continuation normale (`goto B;`)
- ▶ Bloc **B**: assignation de variable (`goto A;`)
- ▶ Bloc **C**: backtracking (`goto B;`)

Instruction de saut : `goto`

Application (très) rapide au backtracking

Fonctions de sauts non locaux



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

---

Signatures :

- ▶ `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 (son environnement n'« existe plus »)

# Exemple

Qu'affiche le programme suivant ?

```
void fun2(jmp_buf e)
{
    longjmp(e, 1);
    printf("...full of raspberries\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;
}
```

- ▶ longjmp peut être utile pour implémenter une gestion d'erreurs dans une pile d'appel avec un gestionnaire d'erreur unique à haut niveau
- ▶ Mais peu utile dans la plupart des (autres) contextes