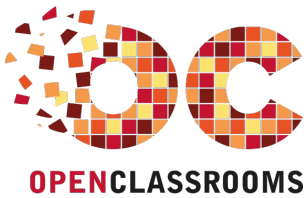
La gestion des erreurs en C

Par Lucas Pesenti (Lucas-84)



www.openclassrooms.com

Dernière mise à jour le 31/08/2012

Sommaire

La gestion des erreurs en C

Par  Lucas Pesenti (Lucas-84)

Mise à jour : 31/08/2012
Difficulté : Intermédiaire Durée d'étude : 2 heures

Introduction

Tout au long de votre apprentissage progressif du langage C, vous avez pu remarquer que certaines fonctions échouaient souvent et pour des raisons variées. Jusqu'à présent, vous vous êtes peut-être contentés de tester le retour de la fonction et d'afficher un simple message d'erreur en conséquence. Bien que ce soit une très bonne habitude, cela ne suffit souvent pas. L'objet de ce tutoriel sera de gérer des diagnostics plus précis quant aux erreurs détectées, et ainsi vous permettre d'agir en conséquence.

A l'origine, le langage C était conçu pour la programmation système. Pour simplifier, cette dernière consiste au développement d'applications plus bas niveau qui utilisent des primitives proches du noyau de la machine. Vous pouvez donc imaginer que les erreurs détectées y sont souvent complexes et variées.

Plan du cours

Pour traiter les nombreuses erreurs qui subsistent à l'exécution, la bibliothèque standard de C dispose d'un fichier d'en-tête spécialisé dans la gestion des erreurs : `<errno.h>`. Cette partie de la bibliothèque standard, ainsi que certaines fonctions situées dans des fichiers d'en-têtes différents (`<string.h>`, `<stdio.h>`) fera l'objet de la première partie du cours, plus théorique, qui résume les pages de documentation et les principes de base d'un programme C gérant basiquement les erreurs

La suite du cours sera constitué d'un débat, plus ou moins subjectif, sur la remontée des erreurs. Nous analyserons les deux grandes méthodes de remontée des erreurs (un seul point de sortie par fonction ou plusieurs), et nous redécouvrirons le subtil `goto`, qui trouve presque toute son utilité dans la gestion des erreurs. Enfin, nous examinerons les sauts non-locaux, souvent délaissés mais parfois utiles.

Pour terminer, la troisième partie sera plus pratique. Nous créerons une petite bibliothèque de gestion des erreurs. Dans un premier temps, nous créerons les fondements de la bibliothèque : initialisation, affichage d'un message d'erreur selon un niveau d'erreur, fermeture. Dans un second temps, nous créerons un système de *stacktrace* de la pile d'appel, qui, à l'aide de mots-clés transformés, permettront au programmeur d'afficher la pile d'appel des fonctions de son programme. Ce sera sans doute la partie la plus technique, puisqu'elles feront appel à une partie plus algorithmique de la programmation en C. Enfin, dans un troisième temps, nous créerons notre propre système d'erreurs courantes, une sorte d'`errno` en plus complet, en leur associant un descriptif et en reproduisant un mécanisme d'exception.

C'est donc le programme qui nous attend. Je ne vous en dis pas plus, la suite est dans le reste du cours. Bon courage !
Sommaire du tutoriel :

[Théorie] Gestion simple et standard des erreurs

Dans cette première partie, nous analyserons les fonctions et autres outils que nous propose la bibliothèque standard de C pour réaliser une gestion des erreurs, certes sommaire, mais nécessaire. Après un bref retour sur le flux d'erreurs standard `stderr` et

son utilité, nous étudierons l'unité de base de la gestion des erreurs : `errno`, puis nous manipulerons et interpréterons son contenu à l'aide d'autres fonctions externes.

Le flux d'erreurs standard `stderr`

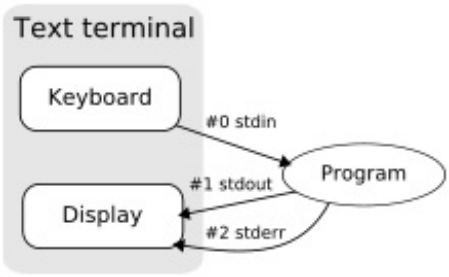
Le flux d'erreurs standard `stderr` se modélise par un fichier dans lequel on peut écrire nos erreurs. Grâce à cette écriture séparée, l'utilisateur pourra rediriger le flux standard afin d'isoler les erreurs et ainsi mieux les traiter, en nombre moins importants, devant la masse d'informations contenues en sortie. Bien que ce passage peut sembler inutile à certains, il me semblait important de revenir sur la notion de flux, qui, au-delà de l'aspect de culture générale informatique qui y est associé, vous permettra de comprendre les trois flux standards qui régissent l'utilisation d'un processus d'un programme écrit en langage C.

Définition d'un flux

Les flux peuvent être représentés comme des canaux dans lesquels circulent des informations. Dans l'informatique moderne, ces flux sont représentés par l'abstraction de base du disque dur : le fichier. Ainsi, tout ce qui sera écrit dans un des flux sera écrit dans le fichier associé. A l'origine modélisés sur des systèmes d'exploitation de type *UNIX*, cette technique s'est largement généralisée auprès des systèmes d'exploitation modernes. D'une manière plus générale, on peut associer la notion de flux à d'autres flux plus renommés : *flux RSS*, *flux de paquets*, etc.

Présentation des trois flux standards

En programmation, l'utilité principale des flux réside dans les entrées/sorties. L'abstraction qu'est le processus nous permet de distinguer trois flux standards : le flux d'entrée standard (`stdin`), le flux de sortie standard (`stdout`) et le flux de sortie d'erreurs standard (`stderr`). Le fichier d'en-tête `<stdio.h>` déclare ces identificateurs comme étant de type pointeurs sur `FILE`.



Les trois flux standards (Wikipedia)

Ces trois descripteurs de fichiers sont initialisés au lancement du processus, et sont libérés à sa destruction. Il n'est donc pas nécessaire de les ouvrir et de les fermer comme on pourrait le faire avec des fichiers classiques.

Le flux d'entrée standard est tout ce qui est envoyé en entrée au programme. La plupart du temps, cela prend la forme d'informations écrites au clavier. Toutefois, comme nous le verrons dans la partie suivante, l'entrée standard peut prendre une forme différente dans le cadre d'une redirection.

Le flux de sortie standard et le flux de sortie d'erreurs standard sont par défaut associés à la console. Là encore, il est possible de les isoler l'un de l'autre, afin de différencier messages d'informations, envoyés à la sortie standard, et messages d'erreurs ou avertissements, envoyés à la sortie d'erreurs standards.

Il est possible de vider le tampon associé à un flux, c'est-à-dire de forcer l'écriture de toutes les données contenues dans celui-ci, à l'aide la fonction `fflush`. Attention, appeler `fflush` avec comme argument un flux d'entrée comme `stdin` est un comportement indéterminé.

Utilité de la redirection d'un flux standard

L'utilité concrète de ces trois flux est attribuée à l'utilisateur. En effet, celui-ci peut rediriger un ou plusieurs des trois flux vers un fichier. L'objet de ce cours n'étant pas de manipuler les différentes commandes consoles, nous ne nous étalerons pas sur les différents outils de redirection, vous laissant le droit d'en savoir plus grâce à une rapide recherche.

De manière succincte, nous pouvons signaler trois sigles permettant de rediriger les trois flux standards. Le chiffre « 1 » et le chevron fermant : « 1> » redirige la sortie standard vers un fichier passé en paramètre à la suite du symbole. Le chiffre « 2 » et le chevron fermant : « 2> » redirige la sortie d'erreurs standards vers le fichier. Le chevron ouvrant « < » redirige quant à lui l'entrée standard vers un fichier.

Par exemple, prenons le code source du célèbre programme « Hello world » qui affiche le message éponyme sur la sortie standard.

Code : C - Hello world

```c
#include <stdio.h>

int main(void) {
    printf("hello world\n");
    return 0;
}
```

Code : Console

```
$ gcc test.c -o hello.out
$ ./hello.out 1> hello.txt
$ cat hello.txt
hello world !
```

Ces essais ont été réalisés sous GNU/Linux. Sous Windows, le fonctionnement est à peu près similaire.

L'utilité concrète est multiple. Premièrement, la redirection du flux d'entrée standard peut permettre de soumettre à votre programme une batterie de tests. Ainsi, vous n'avez pas à retaper insatiablement les mêmes entrées. Deuxièmement, la redirection du flux de sortie standard peut permettre d'enlever tous les messages inutiles que certaines applications affichent. Troisièmement, la redirection de flux de sortie d'erreurs standard peut permettre à l'utilisateur d'isoler les erreurs, et ainsi de les traiter pertinemment.

Vous pouvez vous renseigner sur la fonction `freopen` pour rediriger un des flux standards vers un fichier directement dans votre programme.

La variable globale `errno`

`errno` est le point de départ de la gestion des erreurs standard offerte par la bibliothèque standard de C. C'est une variable globale déclarée dans `<errno.h>`, n'oubliez donc pas d'inclure ce fichier d'en-tête ultérieurement.

Présentation de `errno`

`errno` prend généralement la forme d'une variable globale de type `int`.

Avec les nouvelles technologies de *multi-threading*, `errno` est très souvent implémentée sous forme de macro. N'essayez donc pas de récupérer son adresse.

Contenu standard possible de `errno`

Elle contient une valeur correspondant au code de la dernière erreur s'étant produite. Hélas, on voit là les limites d'`errno`, puisque la bibliothèque standard ne définit que trois codes d'erreur : `EDOM` (passage en paramètre en dehors du domaine

attendu), ERANGE (résultat trop grand ou trop petit) et EILSEQ (erreur de transcodage).

Extensions du contenu de errno

Heureusement, les systèmes d'exploitation communs et actuels proposent souvent certaines extensions au contenu de errno. La norme *POSIX* définit par exemple une trentaine de constantes numériques supplémentaires (elles sont ici).

Interprétation du contenu de errno

Il est donc difficile d'associer directement le contenu de errno à des codes d'erreur, car la portabilité risque de nous faire défaut. C'est pourquoi la bibliothèque standard de C met à notre disposition deux fonctions qui interprètent le contenu de errno, et nous évitent ainsi de passer par un code non standard, ou bien considérablement allongé.

strerror

La fonction strerror associe au code d'erreur passé en paramètre une description de celui-ci en sortie. La fonction est déclarée comme suit dans le fichier d'en-tête de la bibliothèque standard <string.h>.

Code : C - Fichier d'en-tête standard <string.h>

```
char *strerror(int errnum);
```

Son utilisation est assez simple. Par exemple, prenons la fonction strtol, qui convertit une chaîne de caractère en nombre. Le standard indique que si le résultat dépasse les limites du type, LONG_MIN ou LONG_MAX est retourné.

Code : C

```
#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char s[32];

    snprintf(s, sizeof s, "%lu", LONG_MAX + 1UL);

    if (strtol(s, NULL, 10) == LONG_MAX)
        fprintf(stderr, "%s\n", strerror(errno));

    return 0;
}
```

Code : Console

```
Numerical result out of range
```

Mais que se passe-t-il si nous essayons de convertir la valeur LONG_MAX ?

Code : C

```
#include <errno.h>
#include <limits.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char s[32];

    snprintf(s, sizeof s, "%ld", LONG_MAX);

    if (strtol(s, NULL, 10) == LONG_MAX)
        fprintf(stderr, "%s\n", strerror(errno));

    return 0;
}
```

Code : Console

```
Success
```

Dans ce cas-là, il nous faut interpréter le contenu d'`errno` pour savoir si il faut afficher une erreur ou pas.

Code : C

```c
#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char s[32];

    snprintf(s, sizeof s, "%llu", LONG_MAX + 1ULL);

    errno = 0;

    if (strtol(s, NULL, 10) == LONG_MAX && errno)
        fprintf(stderr, "%s\n", strerror(errno));

    return 0;
}
```

perror

L'autre fonction que je tenais à vous présenter est encore plus simple d'usage (et elle est aussi plus utilisée). Il s'agit de la fonction `perror`, qui associe à la valeur courante de `errno` sa description, l'affichant sur la sortie d'erreurs standard. Il est également possible de placer un préfixe devant cette description, que l'on pourra passer en paramètre.

Code : C - Fichier d'en-tête standard <stdio.h>

```c
void perror(const char *s);
```

On reprend le même exemple que tout à l'heure :

Code : C

```c
#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char s[32];

    snprintf(s, sizeof s, "%lu", LONG_MAX + 1UL);

    errno = 0;

    if (strtol(s, NULL, 10) == LONG_MAX && errno)
        perror("strtol");

    return 0;
}
```

Code : Console

```
strtol: Numerical argument out of domain
```

Pour clore cette sous-partie, sachez que vous pouvez également profiter des variables globales `sys_errlist` (tableau de chaînes de caractère indiquant les différentes descriptions pouvant être indexées par `errno`) et `sys_nerr` (nombre de codes d'erreur supportés par le système). Attention, ces deux variables ne sont que conformes BSD.

[Débat] La remontée des erreurs

Après cette première sous-partie particulièrement théorique, vous aurez le droit à une deuxième sous-partie un peu plus subjective, basée sur des débats qui régissent l'utilisation du langage C sur les forums. Chacun est libre d'avoir son propre avis, les thèses qui seront présentées ici seront simplement basées sur l'expérience globale des personnes ayant participé à ce tutoriel. Nous resterons cependant le plus objectif possible.

SESE vs SEME

Dans la programmation depuis toujours, on constate souvent que deux thèses s'opposent quant au nombre de points de sortie (`return`) qu'une fonction peut avoir. Certains prônent la lisibilité, au profit de la règle SESE, et d'autres préfèrent avoir un code moins complexe, avec la règle SEME.

SEME

Commençons par celle qui peut vous sembler la plus naturelle : les multiples points de sortie. Pour les exemples, nous prendrons une fonction qui alloue deux éléments dans une structure retournée en tant que pointeur. C'est sans doute le plus marquant pour différencier les deux techniques. Avec la théorie SEME, voici ce que cela pourrait donner :

Code : C

```c
#define TABSIZE 32
#define NAMESIZE 24

typedef struct {
    int *tab;
    char *name;
```

```
} board;

board *allocBoard(void) {
    board *this;

    if (!(this = malloc(sizeof *this)))
        return NULL;

    if (!(this->tab = malloc(TABSIZE * sizeof *this->tab))) {
        free(this);
        return NULL;
    }

    if (!(this->name = malloc(NAMESIZE))) {
        free(this->tab);
        free(this);
        return NULL;
    }

    return this;
}
```

Un code assez simple qui devrait quand même faire frémir quelques puristes. On constate une lancinante répétition des libérations mémoire et d'une valeur de retour nulle. De plus, en cas d'imbrication importante, il peut être difficile de prendre connaissance des points de sortie excessifs.

SESE

La deuxième théorie consiste à n'utiliser qu'un seul point de sortie pour une fonction donnée. Cela donne parfois lieu à des codes un peu plus longs, mais s'avère souvent plus lisible (à condition de ne pas trop abuser) et moins répétitif.

Code : C

```
#define TABSIZE 32
#define NAMESIZE 24

typedef struct {
    int *tab;
    char *name;
} board;

board *allocBoard(void) {
    board *this = malloc(sizeof *this);

    if (this) {
        this->name = malloc(TABSIZE * sizeof *this->tab);
        this->tab = malloc(NAMESIZE);
    }

    if (!this || !this->name || !this->tab) {
        if (this) {
            free(this->name);
            free(this->tab);
        }
        free(this), this = NULL;
    }

    return this;
}
```

L'étape d'allocation des ressources s'en voit simplifiée. Côté libération des ressources, ça peut parfois devenir un peu tordu. En outre, bien qu'ici ce ne soit pas forcément le cas car notre fonction est simple, l'utilisation de cette théorie peut mener à des

imbrications assez importantes. Imaginons par exemple qu'on veuille rajouter un traitement après chaque allocation, on risque de se retrouver avec beaucoup de branchements conditionnels. C'est pourquoi l'utilisation d'un SESE avec **goto** est souvent recommandée (à condition de savoir ce que l'on fait).

SESE et goto

C'est dans la gestion des erreurs que **goto** trouve sa plus grande utilité. Associé à la théorie SESE, on peut se retrouver avec un code concis et compact. Bien que ce mot-clé soit souvent déconseillé aux débutants à cause des horreurs non structurées qui s'ensuivent quelques fois, dans notre situation, il est presque incontournable.

Code : C

```c
#define TABSIZE 32
#define NAMESIZE 24

typedef struct {
    int *tab;
    char *name;
} board;

board *allocBoard(void) {
    board *this;
    int error = 1;

    if (!(this = malloc(sizeof *this)))
        goto end;

    this->tab = NULL;

    if (!(this->name = malloc(TABSIZE * sizeof *this->tab)))
        goto end;

    if (!(this->tab = malloc(NAMESIZE)))
        goto end;

    error = 0;

  end:
    if (error) {
        if (this) {
            free(this->name);
            free(this->tab);
        }
        free(this), this = NULL;
    }

    return this;
}
```

Ici, nous devons initialiser `tab` à une valeur nulle pour que sa valeur soit fixée si jamais l'allocation de `this->name` échoue. Nous pouvons nous contenter d'une initialisation champ par champ car notre structure est simple, mais lorsque la structure s'avère plus complexe, il peut s'avérer intéressant d'utiliser les littéraux agrégats.

Code : C

```c
*this = (board) { .tab = NULL };
```

Ceux qui préfèrent rester conforme à la norme C89 pourront utiliser une variable statique, dont les membres sont automatiquement initialisés à 0.

Code : C

```
static board init;

if (!(this = malloc(sizeof *this)))
        goto end;

*this = init;
```

Les effets ne se font pas immédiatement ressentir (pas de raccourcissement à court terme), mais sur un code un peu plus long, ce sera inévitable. Notamment lorsque des allocations sont faites dans des imbrications importantes.

SEME et goto

Il est également possible de coupler SEME et **goto**, par exemple :

Code : C

```
#define TABSIZE 32
#define NAMESIZE 24

typedef struct {
    int *tab;
    char *name;
} board;

board *allocBoard(void) {
    board *this;

    if (!(this = malloc(sizeof *this)))
        goto error;

    this->tab = NULL;

    if (!(this->name = malloc(TABSIZE * sizeof *this->tab)))
        goto error;

    if (!(this->tab = malloc(NAMESIZE)))
        goto error;

    return this;

  error:
    if (this) {
        free(this->name);
        free(this->tab);
    }
    free(this);

    return NULL;
}
```

On peut également utiliser plusieurs étiquettes :

Code : C

```
#define TABSIZE 32
#define NAMESIZE 24

typedef struct {
```

```
    int *tab;
    char *name;
} board;

board *allocBoard(void) {
    board *this;

    if (!(this = malloc(sizeof *this)))
        goto mallocThisError;
    if (!(this->name = malloc(TABSIZE * sizeof *this->tab)))
        goto mallocNameError;
    if (!(this->tab = malloc(NAMESIZE)))
        goto mallocTaberror;

    return this;

  mallocTabError:
    free(this->name);
  mallocNameError:
    free(this);
  mallocThisError:
    return NULL;
}
```

Le fonctionnement de **goto** étant assez simple, il ne semble pas nécessaire d'isoler son utilisation dans une bibliothèque. Ce sera donc plus pour une gestion des erreurs « au quotidien ».

Les sauts non-locaux, viables ou pas ?

Alors que **goto**, bien que critiqué, reste quand même souvent accepté, les sauts non-locaux, relativement moins connus, sont beaucoup moins prônés par les utilisateurs récurrents du langage C. En effet, les normes d'utilisation sont plutôt complexes, et la programmation peut très vite devenir complètement déstructurée. Les sauts non-locaux sont néanmoins utilisés de temps en temps, par exemple pour recréer un mécanisme de gestion d'exceptions, c'est pourquoi nous reviendrons quelque peu dessus.

setjmp et longjmp

Le fichier d'en-tête standard `<setjmp.h>` définit plusieurs macros et un type, permettant de sauvegarder puis de restaurer un environnement (cela se manifeste la plupart du temps par une copie de la pile, c'est-à-dire du contenu des différentes variables automatiques au moment de l'appel pour la sauvegarde). Lors de la restauration du contexte, le processus recommencera son exécution avec l'exécution du `setjmp`.

Le fichier définit un type : `jmp_buf`, un tableau dont nous n'avons pas à nous soucier de l'implémentation, puisque nous ne le manipulerons pas directement.

La première fonction que nous allons découvrir, `setjmp`, permet de sauvegarder l'environnement dans son paramètre :

Code : C

```
int setjmp(jmp_buf env);
```

Pour information, la valeur de retour est non nulle (nous pourrons la spécifier tout à l'heure) si elle a été fait dans le cadre d'un environnement spécifié par `longjmp`, ou nulle si elle a été réalisée dans des conditions normales.

Pour restaurer l'environnement ainsi sauvé, c'est une autre fonction qui va nous servir : `longjmp`.

Code : C

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

Le premier argument est l'environnement précédemment sauvé avec un appel à `setjmp`. Le second argument est la valeur que retournera `setjmp` lorsqu'elle sera appelée (valeur non nulle). Petit exemple :

Code : C

```
#include <stdio.h>
#include <setjmp.h>

#define VALUE 5
#define PRINTFUNC() printf("We are into %s\n", __func__)

static jmp_buf env;

void foo(void) {
    PRINTFUNC();
    longjmp(env, VALUE);
}

int main(void) {
    int ret;
    if ((ret = setjmp(env))) {
        printf("%d/%d\n", ret, VALUE);
        PRINTFUNC();
    }
    else
        foo();

    return 0;
}
```

C'est assez simple. On appelle `setjmp` une première fois. On teste sa valeur de retour afin de savoir si elle a été effectuée dans le cadre d'une restauration d'environnement. Si celle-ci est nulle, on appelle la fonction `foo` qui va restaurer l'environnement précédemment enregistré. On envoie comme deuxième argument `VALUE`, valeur qui sera retournée par `setjmp` lors du second appel, effectué juste après le saut non-local. L'affichage de la ligne 17 vous permettra de vérifier cette indication.

Code : Console - Sortie

```
We are into foo
5/5
We are into main
```

Nous réutiliserons les sauts non-locaux dans le cadre de la construction de la bibliothèque, afin de recoder un mécanisme de gestion d'exceptions.

Les sauts non-locaux, leurs défauts

En théorie, les variables restaurées ont la même valeur que lors de l'appel de `longjmp`. Néanmoins, la norme émet un comportement indéterminé lorsqu'une variable automatique est modifiée entre l'appel de `setjmp` et de `longjmp`. Pour le retirer, il suffit de déclarer la variable avec le qualificatif **`volatile`**, ce qui peut se révéler embêtant lorsque vous ne contrôlez pas les déclarations des variables envoyés à une sauvegarde d'environnement. Enfin, notez que ces fonctions peuvent nuire aux performances du programme.

[Pratique] Notre bibliothèque de gestion des erreurs

Dans cette troisième sous-partie pratique, nous allons coder une petite bibliothèque de gestion des erreurs que vous pourrez peut-être utiliser dans vos projets ultérieurs. N'hésitez à l'améliorer par vous-même pour qu'elle réponde mieux à vos propres

besoins. Afin d'avoir des conventions de nommages, j'ai nommé la bibliothèque SLOGL.

Base de la bibliothèque

Dans un premier temps, nous allons nous occuper de toute la base de la librairie : initialisation, arrêt, et puis nous définirons l'implémentation de l'unité de base de notre bibliothèque : le fichier de journalisation, qui servira à l'écriture de toutes les erreurs détectées à l'exécution. Enfin, nous terminerons par un point d'entrée avec une fonction permettant l'écriture dans le fichier de journalisation, selon un niveau d'erreur donné.

Initialisation et arrêt de la bibliothèque

Comme précisé précédemment, la base de notre bibliothèque sera le fichier de journalisation. L'initialisation de celle-ci constituera simplement l'ouverture du fichier, et l'arrêt sa fermeture. Afin de masquer l'implémentation, commençons par définir un alias de type, qui sera le descripteur du fichier de journalisation.

Code : C

```c
#include <stdio.h>

/* Fichier de journalisation. */
typedef FILE *SLOGL_file;
```

C'est donc un objet de ce type que nous allons manipuler lors de l'ouverture et de la fermeture. Dans un premier temps, notre bibliothèque ouvrira le fichier, dont le chemin sera passé en paramètre. Afin de pouvoir réutiliser notre fichier sans le passer en permanence en paramètre, nous implémenterons une sauvegarde du descripteur de fichier.

Code : C

```c
#include <stdio.h>

/* Sauvegarde du descripteur de fichier de journalisation. */
static SLOGL_file fpSave = NULL;

/* Initialise la bibliothèque. */
int SLOGL_init(const char *path) {
    return path && (fpSave = fopen(path, "a"));
}
```

On ouvre notre fichier en mode ajout afin de pouvoir écrire à la fin. Si le fichier n'existe pas, il sera créé, donc cela ne devrait pas poser de problème au programmeur.

L'arrêt de la bibliothèque s'avère également simple :

Code : C

```c
#include <stdio.h>

/* Quitte la bibliothèque. */
int SLOGL_quit(void) {
    return fpSave && fclose(fpSave) != EOF;
}
```

Gestion des fichiers de journalisation

Afin de pouvoir limiter la taille du fichier de journalisation (et ainsi améliorer la lecture du fichier), il faut pouvoir fixer une rotation des fichiers de journalisation. On pourrait effectuer cette rotation selon un critère de taille, mais j'ai choisi la simplicité avec la date. Il suffira ainsi de créer un fichier de journalisation par jour. On ajoutera la date du fichier dans son nom, comme cela la rotation se fera automatiquement.

Pour cela, il nous faut une fonction qui détermine la date actuelle. Cette dernière sera implémentée sous la forme d'un pointeur de structure de type `struct tm`. On commence donc par créer une fonction privée pour obtenir cette date :

Code : C

```
#include <time.h>

/* Récupère la date courante. */
static struct tm *getDate(void) {
    time_t t = time(NULL);
    return localtime(&t);
}
```

Ensuite, il suffit de modifier un peu notre fonction d'initialisation pour obtenir un nom de fichier complet.

Code : C

```
#include <stdio.h>
#include <time.h>

/* Taille maximale du chemin d'un fichier. */
#define SLOGL_MAXFILEPATH 1024

/* Initialise la bibliothèque. */
int SLOGL_init(const char *fileName) {
    int ret = 0;

    if (fileName) {
        char fullFileName[SLOGL_MAXFILEPATH];
        struct tm *t = getDate();

        snprintf(fullFileName, sizeof fullFileName, "%s.%d%d%d.log",
                 fileName, t->tm_mday, t->tm_mon + 1, t->tm_year +
1900);
        ret = (fpSave = fopen(fullFileName, "a")) != NULL;
    }

    return ret;
}
```

Implémentation de différents niveaux de journalisation

Les informations écrites dans un fichier de journalisation peuvent être plus ou moins importantes. Elles peuvent rapporter une erreur comme indiquer simplement la valeur d'une variable pour le débogage d'un programme. C'est pourquoi nous allons construire une hiérarchie de niveaux de journalisation, allant de la simple information de débogage à l'erreur fatale. Lors de la compilation, le programmeur choisit le niveau d'erreur dont il veut être informé. Toutes les erreurs dont la priorité est supérieure à celle choisie seront affichées dans le fichier.

Commençons donc par l'implémentation de ces différents niveaux de journalisation. Des commentaires ont été insérées en face de chaque champ de l'énumération pour indiquer leur signification.

Code : C

```
/* Niveau d'erreur. */
typedef enum {
```

```
    SLOGL_LVL_DEBUG,        /* Déboguage du programme. */
    SLOGL_LVL_INFO,         /* Informations diverses. */
    SLOGL_LVL_NOTICE,       /* Informations remarquables */
    SLOGL_LVL_WARNING,      /* Message d'avertissement. */
    SLOGL_LVL_ERROR,        /* Erreur d'exécution. */
    SLOGL_LVL_FATAL         /* Erreur fatale. */
} SLOGL_level;

/* Niveau d'erreur du programme. */
extern SLOGL_level programLevel;
```

Nous allons regarder les constantes définies lors de la phase du préprocesseur et modifier la valeur du niveau de journalisation.

Code : C

```
/* Niveau d'erreur du programme. */
#ifdef INFO
    SLOGL_level SLOGL_programLevel = SLOGL_LVL_INFO;
#elif defined NOTICE
    SLOGL_level SLOGL_programLevel = SLOGL_LVL_NOTICE;
#elif defined WARNING
    SLOGL_level SLOGL_programLevel = SLOGL_LVL_WARNING;
#elif defined ERROR
    SLOGL_level SLOGL_programLevel = SLOGL_LVL_ERROR;
#elif defined FATAL
    SLOGL_level SLOGL_programLevel = SLOGL_LVL_FATAL;
#else
    SLOGL_level SLOGL_programLevel = SLOGL_LVL_DEBUG;
#endif
```

A partir de là, on peut implémenter notre fonction d'écriture dans le fichier de journalisation. On veut écrire une chaîne du type :

Code : Autre

```
NIVEAU - HHhMMmSS : msg
```

On sépare donc les trois écritures. Une première écrit le niveau de journalisation.

Code : C

```
#include <stdio.h>

/* Ecrit le niveau d'erreur dans le fichier de journalisation. */
static void printLevel(SLOGL_level msgLevel) {
    if (fpSave && msgLevel <= SLOGL_LVL_FATAL) {
        static const char *t[] =
            {"Debug", "Info", "Notice", "Warning", "Error", "Fatal
error"};
        fprintf(fpSave, "\n%s - ", t[msgLevel]);
    }
}
```

Une deuxième fonction écrit l'heure.

Code : C

```
#include <stdio.h>
#include <time.h>

/* Ecrit la date dans le fichier de journalisation. */
static void printDate(void) {
    if (fpSave) {
        struct tm *t = getDate();
        fprintf(fpSave, "%dh%dm%d  :  ", t->tm_hour, t->tm_min, t->tm_sec);
    }
}
```

Une troisième réunit ces informations d'en-tête :

Code : C

```
/* Ecrit les informations d'en-tête dans le fichier de journalisation. */
void SLOGL_printHeader(SLOGL_level msgLevel) {
    if (msgLevel >= SLOGL_programLevel) {
        printLevel(msgLevel);
        printDate();
    }
}
```

Une quatrième écrit le message formaté, grâce au mécanisme des fonctions à nombre variable d'arguments :

Code : C

```
#include <stdarg.h>
#include <stdio.h>

/* Ecrit une chaîne dans le fichier de journalisation. */
void SLOGL_vprint(SLOGL_level msgLevel, const char *msg, ...) {
    if (fpSave && msg && msgLevel >= programLevel) {
        va_list ap;
        va_start(ap, msg);
        vfprintf(fpSave, msg, ap);
    }
}
```

Enfin, j'ai choisi d'implémenter la fonction principale sous forme de *macro variadic*.

Code : C

```
/* Affiche un message dans le fichier de journalisation. */

#define SLOGL_print(n, ...) \
do { \
SLOGL_printHeader(n); \
SLOGL_vprint(n, __VA_ARGS__); \
} while (0)
```

Gestion d'une *stacktrace* d'une pile d'appel

Notre objectif va maintenant être de pouvoir permettre au programmeur d'afficher la pile d'appel des fonctions. Pour cela, nous allons modifier certains mots-clés pour savoir quand on rentre dans une fonction donnée puis lorsqu'on en sort. Grâce à ces informations, le programmeur peut savoir quand son programme a cessé de fonctionner, et ainsi agir en conséquence.

Structure de donnée de la pile d'appel

Notre pile d'appel devra reconstituer les différentes fonctions qui commencent et se terminent à une adresse et dans un fichier donné. Nous pourrions donc appréhender le problème sous la forme d'une pile doublement chaînée. Chaque élément de la pile pointe vers la fonction supérieure et inférieure.

Code : C

```
/* Nombre maximal de fonctions appelées pour une fonction donnée.
*/
#define SLOGL_MAXCALLFUNCTION 16

/* Fonction dans la pile d'appel du programme. */
typedef struct frame {
    char *name;             /* Nom de la fonction. */
    void *addr;             /* Adresse de la fonction. */
    char *file;             /* Fichier de la fonction. */
    int startLine;          /* Ligne de début de la fonction. */
    int endLine;            /* Ligne de fin de la fonction. */
    int depth;              /*  Profondeur de la fonction dans la
pile. */
    int iUp;                /*   Nombre   courants   de   fonctions
appelées. */
    int iMax;               /* Nombre de fonctions appelées. */
    struct frame *up        /* Fonctions supérieures. */
        [SLOGL_MAXCALLFUNCTION];
    struct frame *down;     /* Fonction inférieure. */
} frame;

/* Structure de la pile d'appel du programme. */
typedef struct {
    frame *head;            /* Début de la pile. */
    frame *current;         /* Element courant de la pile. */
    int depth;              /* Profondeur de la pile. */
} stack;

/* Pile d'appel du programme. */
static stack currentStack = {NULL, NULL, 0};
```

Ajout d'un élément sur la pile

Afin d'obtenir le suivi des débuts et des fins de fonction, nous allons implémenter quelques mots-clés sous forme de macros, qui permettront l'appel automatique de certaines fonctions pour empiler et dépiler certaines fonctions. Nous implémenterons les fonctions d'empilage et de dépilage un peu plus tard.

Code : C

```
/* Début d'une fonction. */
#define START(x) \
{ \
SLOGL_pushStack(#x, &x, __FILE__, __LINE__)

/* Retour d'une fonction. */
#define RETURN(x) \
do { \
SLOGL_popStack(__LINE__); \
return x; \
} while (0)
```

```
/* Sortie du programme à partir d'une fonction. */
#define EXIT(x) \
do { \
SLOGL_popStack(__LINE__); \
exit(x); \
} while (0)

/* Fin d'une fonction. */
#define END() \
SLOGL_popStack(__LINE__); \
}
```

Les macros `START` et `END` sont à appeler au début de la fonction, à la place des accolades ouvrantes. Les macros `RETURN` et `EXIT` remplacent les fonctionnalités éponymes.

Ensuite, il nous faut les fonctions de manipulation de la pile : empilage, dépilage, affichage, destruction. L'utilisateur de la bibliothèque aura ainsi la possibilité, s'il déclare la macro-constante `SLOGL_PRINTSTACKTRACE`, d'afficher la pile au fur et à mesure des appels de fonction. Comme on ne peut pas se servir du descripteur de fichier de journalisation que le programmeur n'a pas passé en paramètre des pseudos-macros, nous utiliserons la sauvegarde de celui-ci, que nous avons initialisé précédemment.

Code : C

```
/* Afficher la pile au fur et à mesure ? */
static int printStackTrace = 0;


/* Détruit la pile d'appel du programme. */
static void deleteStack(void) {
    frame *tmp = currentStack.head;

    while (tmp) {
        if (tmp->up[tmp->iUp]) {
            tmp = tmp->up[tmp->iUp];
            ++tmp->down->iUp;
        }
        else if (tmp->down) {
            frame *buf = tmp->down;
            free(tmp);
            tmp = buf;
        }
        else
            break;
    }

    free(currentStack.head);
}

/* Ecrit une description d'une fonction dans le fichier de *
* journalisation. */
static void printFunction(frame *this, int started) {
    SLOGL_vprint(SLOGL_LVL_DEBUG, "\n\t%s:%d : ", this->file,
                 started ? this->endLine : this->startLine);

    for (int j = 0; j < this->depth; ++j)
        SLOGL_vprint(SLOGL_LVL_DEBUG, "\t");

    SLOGL_vprint(SLOGL_LVL_DEBUG, "%s %s[%p]", started ?
           "END" : "START", this->name, this->addr);
}

/* Positionne l'affichage ou non de la pile d'appel au fur et à
mesure. */
void SLOGL_setStackTrace(int n) {
    printStackTrace = n;
```

```
}

/* Empile une fonction. */
void SLOGL_pushStack(char *name, void *addr, char *file, int line) {
    SLOGL_frame *tmp = currentStack.current;
    SLOGL_frame *this = malloc(sizeof *this);

    if (this) {
        this->name = name;
        this->addr = addr;
        this->file = file;
        this->startLine = line;
        this->iUp = this->iMax = 0;
        this->endLine = 0;
        this->depth = ++currentStack.depth;
        this->down = tmp;

        for (int i = 0; i < SLOGL_MAXCALLFUNCTION; ++i)
            this->up[i] = NULL;

        if (!currentStack.head) {
            if (printStackTrace)
                SLOGL_printHeader(SLOGL_LVL_DEBUG);
            currentStack.head = this;
        }

        if (tmp)
            tmp->up[tmp->iMax++] = this;

        currentStack.current = this;

        if (printStackTrace)
            printFunction(currentStack.current, 0);
    }
}

/* Dépile une fonction. */
void SLOGL_popStack(int endLine) {
    if (currentStack.current) {
        currentStack.current->endLine = endLine;

        if (printStackTrace)
            printFunction(currentStack.current, 1);

        currentStack.current = currentStack.current->down;
    }

    --currentStack.depth;
}
```

La destruction de la pile sera automatisée lors de l'arrêt de la bibliothèque.

Code : C

```
/* Quitte la bibliothèque. */
int SLOGL_quit(void) {
deleteStack();
    return fpSave && fclose(fpSave) != EOF;
}
```

Affichage de la pile

Pour afficher la pile, il suffit de parcourir cette dernière à l'aide d'un pointeur temporaire.

Code : C

```
/* Affiche la pile d'appel du programme. */
void SLOGL_displayStack(void) {
    frame *tmp = currentStack.head;

    SLOGL_printHeader(SLOGL_LVL_DEBUG);
    printFunction(currentStack.head, 0);

    while (tmp) {
        if (tmp->up[tmp->iUp]) {
            tmp = tmp->up[tmp->iUp];
            printFunction(tmp, 0);
            ++tmp->down->iUp;
        }
        else if (tmp->down) {
            if (tmp->endLine)
                printFunction(tmp, 1);
            tmp->iUp = 0;
            tmp = tmp->down;
        }
        else
            break;
    }

    if (currentStack.head->endLine)
        printFunction(tmp, 1);
    currentStack.head->iUp = 0;
}
```

Prenons exemple avec une fonction récursive, qui convertit une valeur décimale en binaire :

Code : C

```
#include <stdio.h>
#include <stdlib.h>
#include "SLOGL_main.h"

void foo(unsigned val)
  START(foo);
    if(val) {
        foo(val / 2);
        putchar((val % 2) ? '1' : '0');
    }
  END();

int main(void) {
    SLOGL_init("mylog");

    SLOGL_setStackTrace(1);

  START(main);

    foo(14);

  END();

    SLOGL_quit();
    return 0;
}
```

Cela donnera, dans le fichier de journalisation, quelque chose comme cela :

Code : Autre

```
Debug - 12h5m33 :
        test.c:18 :         START main[0x80486cf]
        test.c:6 :                 START foo[0x80485e4]
        test.c:6 :                         START foo[0x80485e4]
        test.c:6 :                                 START foo[0x80485e4]
        test.c:6 :                                         START foo[0x80485e4]
        test.c:6 :                                                 START foo[0x80485e4]
        test.c:6 :                                                         START foo[0x8
        test.c:11 :                                                         END foo[0x80
        test.c:11 :                                                 END foo[0x80485e4]
        test.c:11 :                                         END foo[0x80485e4]
        test.c:11 :                                 END foo[0x80485e4]
        test.c:11 :                         END foo[0x80485e4]
        test.c:11 :                 END foo[0x80485e4]
        test.c:22 :         END main[0x80486cf]
```

Notez également l'existence de la fonction `backtrace` des extensions GNU.

Extension de `errno` : gestion d'un ensemble d'erreurs courantes

Étant donné que la variable globale `errno` ne possède que trois valeurs possibles de manière standard, nous allons standardiser d'autres codes d'erreur et les manipulerons dans notre propre bibliothèque. Nous en profiterons pour coupler ce mécanisme avec un mécanisme d'exception.

Erreurs courantes

Pour l'occasion, j'ai réuni quelques erreurs courantes avec lesquelles vous pourriez positionner notre variable globale : allocation mémoire, ouverture d'un fichier, fermeture d'un fichier et paramètre invalide.

Code : C

```c
/* Codes d'erreur. */
typedef enum {
    SLOGL_ERR_NOERR,        /* Aucune erreur. */
    SLOGL_ERR_MEM,          /* Allocation mémoire. */
    SLOGL_ERR_FOPEN,        /* Ouverture d'un fichier. */
    SLOGL_ERR_FCLOSE,       /* Fermeture d'un fichier. */
    SLOGL_ERR_PARAM,        /* Passage en paramètre. */
    SLOGL_ERR_DIVNUL,       /* Division par zéro. */
    SLOGL_ERR_LAST          /* Nombre d'erreurs. */
} SLOGL_err;
```

La liste est loin d'être exhaustive, c'est ensuite à vous de la compléter pour qu'elle soit mieux adaptée à vos besoins.

Implémentation d'un système complet, semblable à `errno`

Nous allons maintenant créer les équivalents de `strerror` et de `perror`. Ils s'avèrent relativement simple à recoder.

Code : C

```c
/* Erreur courante du programme. */
```

```
extern SLOGL_err SLOGL_currentError;

/* Récupère une description de l'erreur. */
char *SLOGL_printDesError(SLOGL_err n) {
    static const char *t[] = {
        "aucune erreur",
        "allocation mémoire",
        "ouverture d'un fichier",
        "fermeture d'un fichier",
        "paramètre invalide",
        "division par zéro"
    };
    return n < SLOGL_ERR_LAST ? (char *)t[n] : NULL;
}

/* Ecrit la description de l'erreur courante. */
void SLOGL_printError(const char *msg) {
    if (msg)
        SLOGL_print(SLOGL_LVL_ERROR, "%s: %s", msg,
                    SLOGL_printDesError(SLOGL_currentError));
}
```

Un mécanisme d'exception avec les sauts non-locaux

Nous allons maintenant réutiliser les quelques notions que nous avons apprises sur les sauts non-locaux pour recoder un mécanisme d'exception, semblable à celui du C++. Le principe est simple ; on détermine une zone où une erreur peut survenir. On la délimite par un bloc « try ». Pour signaler une erreur, on utilise le mot-clé « throw », qui va envoyer une valeur à un bloc « catch » situé en-dessous, dans lequel nous pourrons traiter tranquillement l'erreur. Voilà par exemple ce que l'on pourrait faire :

Code : C

```
#include <stdio.h>
#include "SLOGL_main.h"

int divideNumbers(int a, int b) {
    if (!b)
        throw(SLOGL_ERR_DIVNUL);

    return a / b;
}

int main(void) {
    int a, b;

    scanf("%d%d", &a, &b);

    try {
        printf("%d\n", divideNumbers(a, b));
    }
    catch(SLOGL_err val) {
        fprintf(stderr, "Erreur : %s\n", SLOGL_printDesError(val));
    }

    return 0;
}
```

Afin de pouvoir imbriquer plusieurs blocs, nous allons créer une petite pile, allouée statiquement pour que ce ne soit pas trop lourd à gérer, contenant le type de l'exception rencontrée et la sauvegarde de l'environnement.

Code : C

```
/* Taille maximale de la pile d'exceptions. */
#define SLOGL_STACKMAXLEN 16

/* Exception. */
extern struct SLOGL_exception {
    struct {
        int type;           /* Type de l'exception. */
        jmp_buf env;        /* Environnement sauvergardé. */
    } tab[SLOGL_STACKMAXLEN],
      *current;
} SLOGL_catch;

/* Essai d'un bloc. */
#define try \
if  (!((++SLOGL_catch.current)->type  =  setjmp(SLOGL_catch.current-
>env)))

/* Bloc de traitement d'erreur. */
#define catch(id) \
int SLOGL_tmp = 1; \
if(!SLOGL_catch.current->type) \
--SLOGL_catch.current; \
else for (id = (SLOGL_catch.current--)->type; SLOGL_tmp; \
SLOGL_tmp = 0)

/* Signalement d'une erreur. */
#define throw(t) \
longjmp(SLOGL_catch.current->env, t)
```

Le code ne supporte au maximum que 16 imbrications, ce qui reste néanmoins suffisant. Toutefois, contrairement au C++, on ne peut qu'envoyer une variable de type entier avec un `throw` (ce sont les limites de `setjmp`). De plus, certaines situations peuvent causer des bogues (lors d'imbrications compliquées). N'hésitez donc pas à l'améliorer.

D'autres exemples de mécanisme d'exception ont été postés sur ce sujet. N'hésitez pas y aller, et pourquoi pas de partager votre solution.

Code final

Secret (cliquez pour afficher)

Code : C - SLOGL_main.c

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "SLOGL_main.h"

#if !defined (__STDC_VERSION__) || __STDC_VERSION__ < 199901L
#error "Use of SLOGL requires C99"
#endif

/* Taille maximale du chemin d'un fichier. */
#define SLOGL_MAXFILEPATH 1024


/* Fichier de journalisation. */
typedef FILE *SLOGL_file;


/* Erreur courante du programme. */
SLOGL_err SLOGL_currentError = 0;
```

```
/* Exception. */
struct SLOGL_exception SLOGL_catch = {.current = SLOGL_catch.tab -
1} ;

/* Niveau d'erreur du programme. */
SLOGL_level SLOGL_programLevel = SLOGL_LVL_DEBUG;

/* Pile d'appel du programme. */
static SLOGL_stack currentStack = {NULL, NULL, 0};

/* Sauvegarde du descripteur de fichier de journalisation. */
static SLOGL_file fpSave = NULL;

/* Afficher la pile au fur et à mesure ? */
static int printStackTrace = 0;


/* Récupère la date courante. */
static struct tm *getDate(void) {
    time_t t = time(NULL);
    return localtime(&t);
}

/* Ecrit le niveau d'erreur dans le fichier de journalisation. */
static void printLevel(SLOGL_level msgLevel) {
    if (fpSave && msgLevel <= SLOGL_LVL_FATAL) {
        static const char *t[] =
            {"Debug", "Info", "Notice", "Warning", "Error", "Fatal
error"};
        fprintf(fpSave, "\n%s - ", t[msgLevel]);
    }
}

/* Ecrit la date dans le fichier de journalisation. */
static void printDate(void) {
    if (fpSave) {
        struct tm *t = getDate();
        fprintf(fpSave, "%dh%dm%d : ", t->tm_hour, t->tm_min, t-
>tm_sec);
    }
}

/* Détruit la pile d'appel du programme. */
static void deleteStack(void) {
    SLOGL_frame *tmp = currentStack.head;

    while (tmp) {
        if (tmp->up[tmp->iUp]) {
            tmp = tmp->up[tmp->iUp];
            ++tmp->down->iUp;
        }
        else if (tmp->down) {
            SLOGL_frame *buf = tmp->down;
            free(tmp);
            tmp = buf;
        }
        else
            break;
    }

    free(currentStack.head);
}

/* Ecrit une description d'une fonction dans le fichier de *
* journalisation. */
static void printFunction(SLOGL_frame *this, int started) {
    SLOGL_vprint(SLOGL_LVL_DEBUG, "\n\t%s:%d : ", this->file,
                 started ? this->endLine : this->startLine);

    for (unsigned j = 0; j < this->depth; ++j)
```

```
            SLOGL_vprint(SLOGL_LVL_DEBUG, "\t");

        SLOGL_vprint(SLOGL_LVL_DEBUG, "%s %s[%p]", started ?
                "END" : "START", this->name, this->addr);
    }

    /* Récupère une description de l'erreur. */
    char *SLOGL_printDesError(SLOGL_err n) {
        static const char *t[] = {
            "aucune erreur",
            "allocation mémoire",
            "ouverture d'un fichier",
            "fermeture d'un fichier",
            "paramètre invalide",
            "division par zéro"
        };
        return n < SLOGL_ERR_LAST ? (char *)t[n] : NULL;
    }

    /* Ecrit la description de l'erreur courante. */
    void SLOGL_printError(const char *msg) {
        if (msg)
            SLOGL_print(SLOGL_LVL_ERROR, "%s: %s", msg,
                        SLOGL_printDesError(SLOGL_currentError));
    }

    /* Positionne l'affichage ou non de la pile d'appel au fur et à
    mesure. */
    void SLOGL_setStackTrace(int n) {
        printStackTrace = n;
    }

    /* Positionne le niveau d'erreur du programme. */
    void SLOGL_setProgramLevel(SLOGL_level lvl) {
        SLOGL_programLevel = lvl;
    }

    /*  Ecrit  les  informations  d'en-tête  dans  le  fichier  de
    journalisation. */
    void SLOGL_printHeader(SLOGL_level msgLevel) {
        if (msgLevel >= SLOGL_programLevel) {
            printLevel(msgLevel);
            printDate();
        }
    }

    /* Ecrit une chaîne dans le fichier de journalisation. */
    void SLOGL_vprint(SLOGL_level msgLevel, const char *msg, ...) {
        if (fpSave && msg && msgLevel >= SLOGL_programLevel) {
            va_list ap;
            va_start(ap, msg);
            vfprintf(fpSave, msg, ap);
        }
    }

    /* Empile une fonction. */
    void SLOGL_pushStack(char *name, void *addr, char *file, int line)
    {
        SLOGL_frame *tmp = currentStack.current;
        SLOGL_frame *this = malloc(sizeof *this);

        if (this) {
            this->name = name;
            this->addr = addr;
            this->file = file;
            this->startLine = line;
            this->iUp = this->iMax = 0;
            this->endLine = 0;
            this->depth = ++currentStack.depth;
            this->down = tmp;
```

```
            for (int i = 0; i < SLOGL_MAXCALLFUNCTION; ++i)
                this->up[i] = NULL;

            if (!currentStack.head) {
                if (printStackTrace)
                    SLOGL_printHeader(SLOGL_LVL_DEBUG);
                currentStack.head = this;
            }

            if (tmp)
                tmp->up[tmp->iMax++] = this;

            currentStack.current = this;

            if (printStackTrace)
                printFunction(currentStack.current, 0);
        }
    }

    /* Dépile une fonction. */
    void SLOGL_popStack(int endLine) {
        if (currentStack.current) {
            currentStack.current->endLine = endLine;

            if (printStackTrace)
                printFunction(currentStack.current, 1);

            currentStack.current = currentStack.current->down;
        }

        --currentStack.depth;
    }

    /* Affiche la pile d'appel du programme. (nécessite le niveau
    DEBUG) */
    void SLOGL_displayStack(void) {
        SLOGL_frame *tmp = currentStack.head;

        SLOGL_printHeader(SLOGL_LVL_DEBUG);
        printFunction(currentStack.head, 0);

        while (tmp) {
            if (tmp->up[tmp->iUp]) {
                tmp = tmp->up[tmp->iUp];
                printFunction(tmp, 0);
                ++tmp->down->iUp;
            }
            else if (tmp->down) {
                if (tmp->endLine)
                    printFunction(tmp, 1);

                tmp->iUp = 0;
                tmp = tmp->down;
            }
            else
                break;
        }

        if (currentStack.head->endLine)
            printFunction(tmp, 1);
        currentStack.head->iUp = 0;
    }

    /* Initialise la bibliothèque. */
    int SLOGL_init(const char *fileName) {
        int ret = 0;

        if (fileName) {
            char fullFileName[SLOGL_MAXFILEPATH];
```

```
            struct tm *t = getDate();

            snprintf(fullFileName, sizeof fullFileName,
"%s.%d%d%d.log",
                     fileName, t->tm_mday, t->tm_mon + 1, t->tm_year +
1900);
            ret = (fpSave = fopen(fullFileName, "a")) != NULL;
        }

        return ret;
}

/* Quitte la bibliothèque. */
int SLOGL_quit(void) {
    deleteStack();
    return fpSave && fclose(fpSave) != EOF;
}
```

Secret (cliquez pour afficher)

Code : C - SLOGL_main.h

```
#ifndef H_LP_MAIN_20120424183153
#define H_LP_MAIN_20120424183153


#if !defined (__STDC_VERSION__) || __STDC_VERSION__ < 199901L
#error "Use of SLOGL requires C99"
#endif


#include <setjmp.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>


/* Nombre maximal de fonctions appelées pour une fonction donnée.
*/
#define SLOGL_MAXCALLFUNCTION 16

/* Taille maximale de la pile d'exceptions. */
#define SLOGL_STACKMAXLEN 16


/* Affiche un message dans le fichier de journalisation. */
#define SLOGL_print(n, ...) \
do { \
SLOGL_printHeader(n); \
SLOGL_vprint(n, __VA_ARGS__); \
} while (0)


/* Début d'une fonction. */
#define START(x) \
{ \
SLOGL_pushStack(#x, &x, __FILE__, __LINE__)

/* Retour d'une fonction. */
#define RETURN(x) \
do { \
SLOGL_popStack(__LINE__); \
return x; \
} while (0)

/* Sortie du programme à partir d'une fonction. */
#define EXIT(x) \
```

```
do { \
SLOGL_popStack(__LINE__); \
exit(x); \
} while (0)

/* Fin d'une fonction. */
#define END() \
SLOGL_popStack(__LINE__); \
}

/* Essai d'un bloc. */
#define try \
if (!((++SLOGL_catch.current)->type = setjmp(SLOGL_catch.current-
>env)))

/* Bloc de traitement d'erreur. */
#define catch(id) \
if(!SLOGL_catch.current->type) \
--SLOGL_catch.current; \
else for (int SLOGL_tmp = 1, id = (SLOGL_catch.current--)->type;
SLOGL_tmp; \
SLOGL_tmp = 0)

/* Signalement d'une erreur. */
#define throw(t) \
longjmp(SLOGL_catch.current->env, t)


/* Fonction dans la pile d'appel du programme. */
typedef struct frame {
    char *name;             /* Nom de la fonction. */
    void *addr;             /* Adresse de la fonction. */
    char *file;             /* Fichier de la fonction. */
    int startLine;          /* Ligne de début de la fonction. */
    int endLine;            /* Ligne de fin de la fonction. */
    unsigned depth;         /* Profondeur de la fonction dans la
pile. */
    unsigned iUp;           /* Nombre courants de fonctions
appelées. */
    unsigned iMax;          /* Nombre de fonctions appelées. */
    struct frame *up        /* Fonctions supérieures. */
        [SLOGL_MAXCALLFUNCTION];
    struct frame *down;     /* Fonction inférieure. */
} SLOGL_frame;

/* Structure de la pile d'appel du programme. */
typedef struct {
    SLOGL_frame *head;      /* Début de la pile. */
    SLOGL_frame *current;   /* Element courant de la pile. */
    unsigned depth;         /* Profondeur de la pile. */
} SLOGL_stack;

/* Exception. */
extern struct SLOGL_exception {
    struct {
        int type;           /* Type de l'exception. */
        jmp_buf env;        /* Environnement sauvergardé. */
    } tab[SLOGL_STACKMAXLEN], *current;
} SLOGL_catch;

/* Niveau d'erreur. */
typedef enum {
    SLOGL_LVL_DEBUG,        /* Déboguage du programme. */
    SLOGL_LVL_INFO,         /* Informations diverses. */
    SLOGL_LVL_NOTICE,       /* Informations remarquables */
    SLOGL_LVL_WARNING,      /* Message d'avertissement. */
    SLOGL_LVL_ERROR,        /* Erreur d'exécution. */
    SLOGL_LVL_FATAL         /* Erreur fatale. */
} SLOGL_level;
```

```
/* Codes d'erreur. */
typedef enum {
    SLOGL_ERR_NOERR,        /* Aucune erreur. */
    SLOGL_ERR_MEM,          /* Allocation mémoire. */
    SLOGL_ERR_FOPEN,        /* Ouverture d'un fichier. */
    SLOGL_ERR_FCLOSE,       /* Fermeture d'un fichier. */
    SLOGL_ERR_PARAM,        /* Passage en paramètre. */
    SLOGL_ERR_DIVNUL,       /* Division par zéro. */
    SLOGL_ERR_LAST          /* Nombre d'erreurs. */
} SLOGL_err;


/* Erreur courante du programme. */
extern SLOGL_err SLOGL_currentError;

/* Niveau d'erreur du programme. */
extern SLOGL_level SLOGL_programLevel;


/* Positionne l'affichage ou non de la pile d'appel au fur et à
mesure. */
void SLOGL_setStackTrace(int n);

/* Positionne le niveau d'erreur du programme. */
void SLOGL_setProgramLevel(SLOGL_level lvl);

/* Récupère une description de l'erreur. */
char *SLOGL_printDesError(SLOGL_err n);

/* Ecrit la description de l'erreur courante. */
void SLOGL_printError(const char *msg);

/*  Ecrit  les  informations  d'en-tête  dans  le  fichier  de
journalisation. */
void SLOGL_printHeader(SLOGL_level msgLevel);

/* Ecrit une chaîne dans le fichier de journalisation. */
void SLOGL_vprint(SLOGL_level msgLevel, const char *msg, ...);

/* Empile une fonction. */
void SLOGL_pushStack(char *name, void *addr, char *file, int
line);

/* Dépile une fonction. */
void SLOGL_popStack(int endLine);

/* Affiche  la  pile  d'appel  du  programme.  (nécessite  le  niveau
DEBUG) */
void SLOGL_displayStack(void);

/* Initialise la bibliothèque. */
int SLOGL_init(const char *fileName);

/* Quitte la bibliothèque. */
int SLOGL_quit(void);

#endif
```

Ce cours est maintenant terminé, j'espère que vous savez désormais effectuer une gestion des erreurs conséquente et pertinente. N'hésitez pas à me contacter si vous avez des remarques ou des critiques.

Merci à tous ceux qui ont participé à ce tutoriel (notamment Arthurus pour les idées de la bibliothèque ; Maëlan et Taurre pour leur relecture lors de la phase d'élaboration).