情報科学研究部用テキスト

テキストエディタを作ろう

Go Suzuki

目次

コンパイラ, OS の呼称方法

Microsoft C は Microsoft 製の最近の C++ コンパイラの C 言語部分を示す．つまり，`extern \"C\"` で指定した Visual C++(2005 以降) である．

BSD は実装上 BSD のシステムコール・ヘッダが利用できる OS，つまり，FreeBSD, NetBSD, OpenBSD, DragonflyBSD, macOS(Darwin) を示す． 古い BSD については当てはまらない事項がある可能性がある．

ライセンス

1. ファイルを読み書きしよう 楽

1.1. 標準ライブラリのセキュリティを考慮した正しい使い方

普段標準ライブラリをのほほ〜んと使っていないだろうか．危険でないプログラムを作るにはしっかりとエラーハンドリングを意識して実装することが重要である．この章では，Visual C++ しか実装していない `fscanf_s`，`fopen_s` などの話も混ぜつつ，エラーハンドリングを見ていこう．

1.1.1. 戻り値 /error_no を意識して

普段何気なく `fopen` や，`fprintf` などを使っていないだろうか．

危険な使い方

```
#include <stdio.h>

int main(){
    FILE * fp;
    fp = fopen("hoge.txt", "r");
    char buf[32]; // 現代の C 言語では関数の頭で宣言しなくても良い．
    fscanf(fp, "%s", buf);
    puts(buf);
    return 0;
}
```

(1) 戻り値は適切かを確認する

`fopen` はファイルを開くのに失敗すると，`NULL` を返す．開こうとしたファイルが存在しないことはよくあることで，正しくチェックすべきである．

`fopen` の戻り値を確認する．

```
// ファイル先頭
#include <stdlib.h>
//fp = fopen("hoge.txt", "r") == NULL を修正
if((fp = fopen("hoge.txt", "r")) == NULL) {
    puts("ERROR: Cannot open.");
    exit(EXIT_FAILURE);
}
```

普通 `NULL` は `0` であるが，これは標準 C には定義されていないので，正しく `NULL` と書くべきである．if 文の中で，エラーメッセージを表示して，`exit` 関数を呼び出している．エラー

コードは `EXIT_FAILURE` である．`1` を入れる文献もあるが，同じ理由でしっかりと識別子を書くべきである．

`fscanf` についても同様に，チェックをしてみよう．open はできたものの書き込みに失敗することはたくさんある．

`fscanf` の戻り値を確認する．

```
// ファイル先頭
#include <stdlib.h>
//fscanf(fp, "%s", buf) を修正
if(fscanf(fp, "%s", buf) == EOF) {
    fprintf(stderr, "File write error.");
}
```

(2) エラーの内容を確認する

さて，ただファイルを開けないとエラー表示するだけではユーザに不親切である．エラーの情報が分かることが望ましい．そのためには，`errno.h` を用いる．

`errno` を確認する．

```
// ファイル先頭
#include <errno.h>
// 修正したものをさらに修正
if((fp = fopen("hoge.txt", "r")) == NULL) {
    puts("ERROR: Cannot open.");
    perror("file open error");
    exit(EXIT_FAILURE);
}
```

`perror` はエラーを出力する関数である．通常 `stderr` のバッファに上記の例では以下のように出力される．

`perror` の出力

```
file open error: File Not Found
```

エラーに関する文字列を返す関数 `strerror` も存在する．

`strerror`

```
char *strerror(int errnum);
int strerror_r(int errnum, char *buf, size_t buflen);   // XSI
char *strerror_r(int errnum, char *buf, size_t buflen); // GNU
```

`strerror` はそのままバッファへのポインタを返す．しかし，この関数はスレッドセーフでない場合があり，マルチスレッド環境での使用は推奨されない．一方，`strerror_r` は XSI/GNU 拡張である．`buf` にエラーメッセージを格納する．それぞれで定義が違うのは二つの拡張によって使い方が異なる．XSI は成功すると 0 を返し，GNU は `buf` をそのまま返すか，エラーメッセージが格納された静的なバッファへのポインタを返す．全く互換性がないことに注意しよう．

Microsoft C では，全く異なるアプローチ（C オプション標準なやり方）でこの問題に対処していることを後述する．

さらに，あなたが BSD や Windows のプログラマであり，やる気があるならば Exit Code をそのエラーにあったものに変えると良いだろう．（Linux では標準の Exit Code が存在しない．）

`exit` の Exit Code を変える

```
// ファイル先頭
#include <stdlib.h>
#ifdef _WIN32
#include <winerror.h>
#elif defined(__FreeBSD__) || (defined(__NetBSD__) || \
        defined(__OpenBSD__) || defined(__DragonflyBSD__) || \
        defined(__APPLE__)
#define EXIT_RETURN
#include <sysexits.h>
#endif
// 修正したものをさらに修正
if((fp = fopen("hoge.txt", "r")) == NULL) {
    printf("ERROR: Cannot open.\n");
    perror("file open error");
    if(errno == ENOENT)
#ifdef _WIN32
        exit(ERROR_FILE_NOT_FOUND);
#elif defined(EXIT_RETURN)
```

```
        exit(EX_OSFILE);
#else
        exit(EXIT_FAILURE);
#endif
    else
        exit(EXIT_FAILURE);
}
```

大抵のプログラマはこれを怠るが，普通のことなので，気にする必要はない．

さて，あなたが C 言語のプログラマでない場合，すなわち最近の言語のプログラマである場合，事はもっと簡単である．

まず，ジャンプ例外機能のある言語だとどうだろうか．マルチスレッドのことなんか考えずにもっと簡潔に書くことができる．

ジャンプ例外機能の例： Ruby

```
begin
    file = File.open("hoge.txt", "r")
rescure => error
    p error
    exit(1) # 環境依存
end
```

ジャンプ例外機能の例： C#

```
try{
    var file = new System.IO.StreamReader("hoge.txt");
}catch(Exception ex){
    System.Console.Error.WriteLine(ex);
    System.Environment.Exit(1); // 環境依存
}
```

ファイルが存在するか否かを調べるために `fopen` を用いることがある． これは C 言語では正しい使い方であるが，ジャンプ例外機能のあるプログラミング言語では例外で条件分岐するのはやめるべきである． あくまで，エラーが起こった時のために使うべきである． 基本的に，ファイルの存在可否を調べる関数が備わっているはずであり，そちらを使うべきである．一般的に，ジャンプ例外が発生した場合，プログラムのパフォーマンスはとても悪くなってしまう．ループで複数のファイルが存在するか否かを調べると，例外を使った場合とそうでない場合で

実行速度が大幅に異なる．また，可読性の観点からもやめるべきである．

次に，直和型（Union）のある言語の場合を見てみよう．ファイルの読み書きで使われる直和型とは，例えば，Rust の `Result<File>` や Haskell の `IO モナド` である．特徴としては，普通はエラーハンドリングをしなければならない点である．C 言語やジャンプ例外機能のある言語は簡単にエラーハンドリングを無視することができる．一方で，これらの言語は意識してエラーハンドリングを無視しなければならない．例えば，Rust は `unwrap` 関数を用いて，

Union のある言語でエラーハンドリングを無視する例： Rust

```
let mut f = File::open("hoge.txt").unwrap();
```

と書かねばならない．この時，エラーが発生すれば panic を起こしてプログラムは即時強制終了される．プログラマが注意しなくても安全なプログラムが書ける一方で，気軽にプログラムが書けないことが欠点だろうか．しかし，モナドの力を使えば，簡単にエラー処理を呼び出し元に任せることができる．Rust だと `?` を付けることがそれに該当するだろう．

では，エラーハンドリングをするとどう書けるだろうか．様々なやり方があるが，Rust の場合は，`match` を用いると手続きのように書けて分かりやすいだろう．

Union のある言語での例： Rust

```
match File::open("hoge.txt") {
    Ok(file) => ..
    Err(ex) => {
        eprintln!("{}", ex);
        process::exit(1); // 環境依存
    }
}
```

他にも便利な関数が沢山あるので，使いこなすと楽しく書けるだろう．

(3) `fscanf` のバッファオーバーラン

`fscanf` では書式文字列 `\%s` を用いて文字列の読み取りをしているが，これは危険である．変数 `buf` は 32 バイト分の文字列を読み取ることができるが，これを超えた場合は，スタック上の他の変数を上書きしてしまう場合がある． つまり，`fscanf` には最大何文字読み出して良いかを渡していないから，32 バイト分の文字列を超える大きさの文字列を読み出してしまう可能性がある． そして， その文字列を変数 `buf` に書き込むが， この時に， 変数 `buf` の領域を超える場所に書き込んでしまう．これは，他の変数等を書き換えてしまう可能性がある．そのため，読み込むバイト数を指定するべきである．

読み込むバイト数を指定する

```
//fscanf(fp, "%s", buf); を修正
fscanf(fp, "%31s", buf);
```

また，C11 の Annex K Bounds-checking interfaces では，`fscanf_s` を利用することができる．（Microsoft C が対応．）

`fscanf_s` を利用する

```
//fscanf(fp, "%s", buf); を修正
fscanf_s(fp, "%s", buf, _countof(buf));
```

また，`fgets` が利用できるならば（複雑な処理でなければ），`fgets` を使った方が性能が良くなるので，検討の余地はあるだろう．

(4) お片付けは忘れずに

使ったリソースをこまめに使用後に片付けすることを心がけよう． ファイルの読み書きでは `fclose` を使用する．今回の例では，プログラム終了後，OS が片づけることになるが，大規模なプログラムを書くと，ファイルを頻繁に読み書きすることがあるだろう．そういう場合に，片付けしなければ，簡単にハードウェア資源を消費しつくしてしまう．

`fclose` を利用する

```
//return 文の前に挿入
fclose(fp);
```

ファイルの読み書き中にエラーハンドリング等で，`exit` 関数を実行する場合は，`atexit` 関数を利用すると良い．この関数は，プログラムの終了時に呼び出す関数を登録するものである．

`atexit` を活用して出来たプログラム

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _WIN32
#include <winerror.h>
#elif defined(__FreeBSD__) || (defined(__NetBSD__) || \
        defined(__OpenBSD__) || defined(__DragonflyBSD__) || \
        defined(__APPLE__)
#define EXIT_RETURN
```

```
#include <sysexits.h>
#endif
FILE * fp = NULL;

void on_exit() {
    if(fp != NULL)
        fclose(fp);
}

int main(){
    if((fp = fopen("hoge.txt", "r")) == NULL) {
        printf("ERROR: Cannot open.\n");
        perror("file open error");
        if(errno == ENOENT)
    #ifdef _WIN32
            exit(ERROR_FILE_NOT_FOUND);
    #elif defined(EXIT_RETURN)
            exit(EX_OSFILE);
    #else
            exit(EXIT_FAILURE);
    #endif
        else
            exit(EXIT_FAILURE);
    }
    atexit(on_exit); // 終了時に on_exit を呼び出す
    char buf[32]; // 現代の C 言語では関数の頭で宣言しなくても良い.
    fscanf(fp, "%31s", buf);
    puts(buf);
    // ...
    return 0;
}
```

(5) 補講 `malloc` もエラーチェックをしよう

もう一つ，特にエラーチェックしたい関数がヒープアロケーション関係の関数である．`malloc`，`calloc`，`realloc` 等のアロケーションをする関数は，利用可能な空きメモリがない場合に `NULL` を返す．

`NULL` と `malloc`

```
char * p = (char *)malloc(sizeof(char) * 32);
if(p == NULL) {
    fputs("malloc failed.", stderr);
    exit(EXIT_FAILURE);
}
```

64bit 環境の贅沢にメモリが使える環境であれば大丈夫だが，マイコンや，16bit，8bit 環境であれば，アロケーションに失敗する可能性は高まる．それらの環境では特に気を付けるべきである．

そして，片付けとして必ず `free` 関数を呼び出すことを忘れてはいけない．アロケーションをたくさんして，不要になったのに解放しないことをメモリリークと呼ぶ．連続稼働するサーバプログラムや IoT 機器等では，メモリリークするプログラムを混入させないことが非常に重要である．

`free`

```
free(p);
```

さらに，一度解放した領域を利用したり，もう一度解放させてはならない．前者を use-after-free，後者を double-free と呼ぶ．use-after-free は，解放後の領域は `malloc` によってまた別の場所で使われている可能性があるからである．その領域を上書きすれば，どこか別の関数で想定外の値の変化として障害が起きてしまう．一方，double-free は実装によってダメになる仕組みがあるので，ネットで調べてみると良い．double-free は侮ってはいけない．double-free を利用した権限昇格の脆弱性の例もあり，気を付けるべきである．

さて，対処法としては一つ，`free` の後に必ず `NULL` を代入することが挙げられるだろう．ポインタを別の場所にコピーしていた場合防げないが，マシにはなるだろう．

`NULL`-after-`free`

```
free(p);
p = NULL;
```