# 第14講 例外処理・ファイル

# 14.1 例外処理

プログラムの実行中には、配列の添え字範囲外アクセスやメモリ確保の失敗など、予期しない様々なエラーが発生する可能性があります。本章では、C++が提供する強力で柔軟なエラー処理メカニズムである例外処理 **(exception handling)** について学びます。

---

## 1. 例外処理の基本概念

### エラー処理の課題

以前の`IntArray`クラスを例にとると、不正な添え字が指定された場合に何が起こるでしょうか？
`a[24] = 156; // aのサイズが10の場合、これは配列の範囲外アクセスを引き起こす`

このような範囲外アクセスはコンパイラではチェックされず、実行時にプログラムのクラッシュやデータ破損の原因となる可能性があります。直接的な方法として、コードに`if`文によるチェックを追加することが考えられます。

```
C/C++
int& IntArray::operator[](int i) {
    if (i < 0 || i >= size) {
        // エラー処理...
    }
    return vec[i];
}
```

しかし、「エラー処理」では具体的に何をすべきでしょうか？

1. プログラムを終了させる？ (あまりに強引で、データ損失の可能性がある)
2. エラーメッセージを出力して続行する？ (プログラムが不安定な状態に陥る可能性がある)
3. 特別な値を返す？ (呼び出し側が戻り値をチェックする必要があり、見落とされやすい)

これらの方法は柔軟性に欠けます。例外処理メカニズムは、この問題に対してより良い解決策を提供します。

### `throw`, `try`, `catch` キーワード

C++の例外処理は、主に以下の3つのキーワードを中心に展開されます。

- `try`: `try`ブロックは、例外を発生させる可能性のあるコードを囲みます。
- `catch`: `catch`ブロック（例外ハンドラとも呼ばれる）は、`try`ブロック内でスローされた例外を捕捉し、処理するために使用されます。
- `throw`: エラーが検出されたときに、`throw`式を用いて例外をスロー **(throw)** します。

典型的な例として、`new`演算子によるメモリ確保の失敗を処理するコードを見てみましょう。

```C/C++
try {
    double* a = new double[3000000000U]; // 大量のメモリを確保しようと試みる
} catch (const std::bad_alloc& e) {
    std::cout << "メモリ確保に失敗しました。\n";
    // 回復処理またはプログラムの終了処理を行う
}
```

もし`new`が必要なメモリを確保できなかった場合、`std::bad_alloc`型の例外をスローし、プログラムは対応する`catch`ブロックにジャンプして実行を続けます。

---

## 2. 例外の捕捉と伝播

### `try...catch` 構造

`try`ブロックの後には、一つまたは複数の`catch`ブロックを続けることができます。各`catch`ブロックは、特定の型の例外に対応します。

`try`ブロック内のコードが例外をスローすると、プログラムは現在の実行フローを直ちに中断し、その例外の型を処理できる`catch`ブロックを探します。

### 例外の伝播メカニズム

ある関数の内部で例外がスローされたが、その関数内に`try...catch`ブロックがない、あるいは一致する`catch`ブロックがない場合、例外は呼び出し元の関数へと伝播 **(propagate)** し、一致する`catch`ブロックを探し続けます。このプロセスは、一致するハンドラが見つかるまで、あるいは最終的にハンドラが見つからずにプログラムが終了するまで、関数呼び出しのスタックを逆にたどっていきます。

## 3. 例外のスロー

### throw 式

throw式は、例外をスローするために使用されます。スローされるオブジェクトは、基本データ型（intなど）、文字列リテラル、あるいはカスタムクラスのオブジェクトなど、どのような型でも構いません。通常は、異なる種類のエラーを表すために、専門の例外クラスを定義します。

### 例外クラス

以下のリストは、簡単な例外クラスOverflowを定義し、関数内でそれをスローする方法を示しています。

C/C++

```
// throw.cpp
#include <iostream>

// 空の例外クラスを定義
class Overflow { };

// 例外をスローする可能性のある関数
int f(int x) {
    if (x < 0) throw "値が負です"; // const char* 型の例外をスロー
    if (x > 30000) throw Overflow(); // Overflow オブジェクトをスロー
    return x * 2;
}

int main() {
    int a;
    std::cout << "整数：";
    std::cin >> a;

    try {
        int b = f(a);
        std::cout << "その数の2倍は" << b << "です。\n";
    }
    catch (const char* str) {        // 文字列の例外を捕捉
        std::cout << "例外発生：" << str << '\n';
    }
    catch (const Overflow&) {        // Overflow型の例外はここで捕捉
        std::cout << "オーバフローしました。プログラムを終了します。\n";
        return 1;
```

```
    }
}
```

コード解説:

- `class Overflow { };`: 特定のエラータイプを表すためだけの、簡単な空のクラスを定義しています。
- `throw "値が負です";`: `const char*`型の例外をスローしています。
- `throw Overflow();`: `Overflow`クラスの一時オブジェクトを生成してスローしています。
- `catch (const char* str)` と `catch (const Overflow&)`: それぞれ2つの異なる型の例外を捕捉し、処理します。

---

## 4. `IntArray`クラスへの例外処理の応用

ここで、`IntArray`クラスに例外処理を適用し、添え字の範囲外エラーを処理します。

### ネストされた例外クラス `IdxRngErr`

添え字の範囲外エラーを表すために、`IntArray`クラスの内部にネストクラス (nested class)として`IdxRngErr`を定義します。例外クラスをメインクラス内に定義することで、この例外がメインクラスと密接に関連していることを明確に示せます。

この例外クラスは、エラー情報を記録するために2つのメンバを持ちます。

- `ident`: エラーが発生した`IntArray`オブジェクトへのポインタ。
- `idx`: エラーを引き起こした不正な添え字の値。

### `operator[]`の修正

```C/C++
// IntArray.h (一部)
#ifndef ___Class_IntArray
#define ___Class_IntArray

class IntArray {
    int nelem;
    int* vec;

public:
```

```
    class IdxRngErr {
    private:
        const IntArray* ident;
        int idx;
    public:
        IdxRngErr(const IntArray* p, int i) : ident(p), idx(i) { }
        int index() const { return idx; }
    };

    explicit IntArray(int size) : nelem(size) { vec = new int[nelem]; }
    ~IntArray() { delete[] vec; }

    int& operator[](int i) {
        if (i < 0 || i >= nelem)
            throw IdxRngErr(this, i); // 添字範囲エラー送出
        return vec[i];
    }
};
#endif
```

コード解説:

- `class IdxRngErr { ... };`: `IntArray`の`public`部分にネストクラス`IdxRngErr`を定義しています。
- `throw IdxRngErr(this, i);`: `operator[]`内で、添え字`i`が範囲外の場合、`IdxRngErr`オブジェクトを生成してスローします。`this`ポインタと不正な添え字`i`が`IdxRngErr`のコンストラクタに渡され、エラーハンドラが詳細なエラー情報を取得できるようになっています。

---

## 5. `IntArray`例外処理の使用例

```C/C++
// IntArrayTest.cpp
#include <new>
#include <iostream>
#include "IntArray.h"

// IntArrayを作成し、データを埋める関数
void f(int size, int num) {
    try {
```

```
        IntArray x(size);
        for (int i = 0; i < num; i++) {
            x[i] = i;
            std::cout << "x[" << i << "] = " << x[i] << '\n';
        }
    }
    catch (const IntArray::IdxRngErr& x) {
        std::cout << "添え字範囲外エラー：不正な添え字 " << x.index() << '\n';
        return;
    }
    catch (const std::bad_alloc&) {
        std::cout << "メモリ確保に失敗しました。\n";
        exit(1);
    }
}

int main() {
    int size, num;
    std::cout << "配列のサイズを入力してください："; std::cin >> size;
    std::cout << "データ数を入力してください："; std::cin >> num;
    f(size, num);
    std::cout << "main関数が終了しました。\n";
}
```

実行例:

- 例**1 (**正常実行**)**: 配列サイズに`5`、データ数に`5`を入力すると、正常に実行されます。
- 例**2 (**添え字範囲外**)**: 配列サイズに`5`、データ数に`6`を入力すると、`i`が`5`のときに`IdxRngErr`例外がスローされ、catchブロックで捕捉されます。

---

# 6. 例外の適切な使用法と例外安全性

## 例外と戻り値の使い分け

エラーを通知する方法として、例外と戻り値（ステータスコード）のどちらを使うべきか、という問題は重要です。

- 例外が適しているケース:「予期しない、かつ局所的に即時処理することが困難な」失敗。
    - 例：メモリ確保の失敗 (`std::bad_alloc`)、ディスク満杯によるファイル書き込み失敗など。これらはプログラムの前提条件が崩れる重大なエラーです。

- 戻り値が適しているケース:「ビジネスロジック上、十分にあり得る、かつ予期されるべき正常系の分岐」。
    - 例:「検索したが該当なし」「ユーザーが操作をキャンセル」など。これらは「エラー」ではなく「正常な結果の一つ」です。

### 例外安全なコードとRAII

例外がスローされると、確保したリソース(メモリ、ファイル等)が解放されないリソースリークが発生する危険があります。

**RAII (Resource Acquisition Is Initialization)** は、リソースの管理をオブジェクトの生存期間に紐付けるC++の設計原則です。リソースの確保をコンストラクタで、解放をデストラクタで行います。`std::ifstream`や`std::unique_ptr`のようにRAIIに従ったクラスを使えば、例外発生時にデストラクタが自動的に呼ばれ、リソースが安全に解放されます。

### 重要なルール:デストラクタから例外をスローしてはいけない

理由: ある例外の処理中(スタック巻き戻し中)に、デストラクタがさらに別の例外をスローすると、未処理の例外が2つ同時に存在することになります。この状況はC++では解決不能であり、`std::terminate()`が呼ばれプログラムが強制終了します。

対策: デストラクタ内では、リソース解放に失敗しても例外をスローせず、ログに記録するなどの対応に留めるべきです。

---

# 14.2※ ファイル入出力

※ 補足資料(期末試験範囲外)

プログラムは、計算結果やデータを永続的に保存するために、外部の記憶媒体(ハードディスク、SSDなど)上のファイルを利用します。本節では、C言語スタイルとC++スタイルの両方で、テキストファイルとバイナリファイルの基本的な読み書き方法を学びます。

# 1. ファイルの基本概念

## ファイルとは？

コンピュータにおける「ファイル」とは、OS（オペレーティングシステム）がデータを保存・管理するための基本単位です。ファイルには名前や種類（拡張子）、保存場所（パス）があり、HDD/SSD などの記憶装置に保存されるため、電源を切っても内容は消えません。

また、OSの立場ではファイルは「連続したバイト列」として扱われます。プログラムにとってファイルは、物理的なディスク領域そのものではなく、OSが提供する抽象化されたリソースとして、読み書きの対象になります。

- ファイルは何からできているか
    - 内容（データ）：実際に保存されている文字・画像・プログラム・動画など。形式に関わらず、内部的には 0/1 からなるバイナリデータ（連続したバイト列）として記録されます。
    - メタ情報（属性）：ファイル名、サイズ、作成／更新日時、アクセス権、所有者、隠し属性、読み取り専用など。これらはOSが管理し、プログラムはOSを介して参照・更新します。

- ファイルの種類
    - テキスト（**text**）ファイル: 人間が読める文字（ASCIIやUTF-8など）で構成されています。メモ帳などのテキストエディタで直接開いて編集できます。
    - バイナリ（**binary**）ファイル: データ（数値、構造体、画像データなど）をそのままのバイナリ形式で保存します。テキストファイルより効率的ですが、専用のプログラムなしでは内容を解読できません。

- ファイル名と拡張子
    - ファイル名には通常、拡張子が含まれます。たとえば、.txt、.jpg、.xlsx、.cpp などです。
    - 拡張子は主に、
        - そのファイルがどのような内容であるかを、OSやユーザーが判断する指標となる。
        - 既定（デフォルト）でどのアプリケーションで開くかを決めるために使われる。
    - ただし、拡張子は絶対に信頼できるものではありません。拡張子を変更しても、ファイルの内部構造が変わるわけではありません。

- ファイルとフォルダ（ディレクトリ）
    - ファイル (File)

        - 実際にデータが保存されている実体です（例：テキスト、画像、プログラム）。
    - フォルダ (Folder / Directory)
        - 管理用の容器: 複数のファイルを整理・保管するための入れ物です。
        - 階層構造（ツリー構造）: フォルダの中にさらに別のフォルダ（サブフォルダ）を作成することで、データを階層的に管理できます。

- プログラムによるファイル操作の基本フロー

    - オープン **(Open)**: パスを指定してファイルとプログラムを関連付け、操作ハンドル（ストリームオブジェクト）を取得します。
    - 読み書き **(Read/Write)**: バイト単位、または特定のフォーマットでデータを読み書きします。
    - シーク **(Seek)**: ファイル内の現在の読み書き位置を移動させます。
    - クローズ **(Close)**: ファイルハンドルを解放し、バッファに残っているデータをディスクに書き込みます（フラッシュ）。

## ファイルパス：絶対パスと相対パス

ファイルを開く際には、その場所を示すパスを指定する必要があります。

- 絶対パス **(Absolute Path)**: ルートディレクトリ（階層の最上位）から始まり、対象ファイルまでの完全な経路を示します。環境に依存せず、常に一意にファイルを特定できます。

    - Windowsの例: `C:\Users\Taro\Documents\report.txt`
    - Linux/macOSの例: `/home/user/documents/report.txt`

- 相対パス **(Relative Path)**: プログラムの現在の作業ディレクトリを基準として、対象ファイルまでの相対的な経路を示します。記述が短くなりますが、プログラムを実行する場所によって解釈が変わります。

    - 例1: `report.txt`（作業ディレクトリにある`report.txt`）
    - 例2: `data/input.csv`（作業ディレクトリ内の`data`フォルダにある`input.csv`）
    - 例3: `../backup/log.txt`（作業ディレクトリの一つ上の階層にある`backup`フォルダの`log.txt`）

パスの文字列表現:
プログラム内でパスを文字列として記述する場合、特に Windows 環境では注意が必要です。
エスケープシーケンスの回避: C/C++などの言語では、バックスラッシュ（`\`）はエスケープシーケンス（`\n` や `\t` など）として解釈されます。そのため、Windowsのパスを記述する際は、バックスラッシュを2つ重ねる必要があります。また、文字列中ではスラッシュ（`/`）を使って書くことで代用することもできます。

- NG: `"C:\temp\new_data.txt"`　　（`\n` や `\t` と誤認される）

- OK: `"C:/temp/new_data.txt"`　　（推奨。読みやすく、他OSでも動く）
- OK: `"C:\\temp\\new_data.txt"`

---

## 2. テキストファイルの読み書き

### C言語スタイル

- C言語では、`FILE`ポインタと`fprintf`、`fgets`といった関数群を用いてファイル操作を行います。
- 使用するヘッダファイル：`#include <stdio.h>`

**C言語スタイルでの書き込み**
例：

```C/C++
// TxtWrite.c
#include <stdio.h>

int main() {
    FILE *fp = NULL;

    // 書き込むデータを準備
    char str[] = "INFO";
    int n = 10;
    double x = 3.5;

    // ファイルを書き込みモード("w")でオープンする
    fp = fopen("test_c.txt", "w");
    if (fp == NULL) {
        printf("ファイルをオープンできませんでした！\n");
        return 1;
    }

    // fprintfを使用してフォーマット済みデータを書き込む
    // 最初の引数はファイルポインタ（fp）です。使い方は printf とほぼ同じです。
    fprintf(fp, "This is testing for fprintf...\n");

    // 変数の書き込み：文字列(%s)、整数(%d)、浮動小数点数(%.2f)
    fprintf(fp, "Setting: %s\n", str);
    fprintf(fp, "Count: %d\n", n);
    fprintf(fp, "Value: %.2f\n", x);

    // ファイルをクローズする
    fclose(fp);
```

```
    printf("データが test_c.txt へ書き込まれました。\n");
}
```

主な関数:

- `FILE* fopen(const char* filename, const char* mode)`: ファイルを開く。
  - `filename`: ファイルのパス。
  - `mode`: ファイルへの「アクセス権限」と「操作方法」を決定する重要なパラメータです。よく使われる fopen のモードは以下の通りです:
    - "r" (読み込み): 既存のファイルを開きます。ファイルが存在しない場合はエラー（NULL）となる。
    - "w" (書き込み): 新しくファイルを作成して書き込みます。既にファイルがある場合は中身をすべて消去して上書きするため、注意が必要です。
    - "a" (追記): ファイルの末尾にデータを付け足します。既存の内容は保持され、ファイルが存在しない場合は新規に作成されます。
    - "b"(バイナリ): テキスト変換を行わず、変換なしの生データとして扱います。"rb" や "wb" のように他のモードと組み合わせて使用します。
  - 成功すると`FILE`ポインタを、失敗すると`NULL`を返す。
- `int fprintf(FILE* stream, const char* format, ...)`: ファイルにフォーマットされた文字列を書き込む。`printf`のファイル版。
- `int fclose(FILE* stream)`: 開いたファイルを閉じる。リソースを解放し、バッファをフラッシュする。

**`fclose(fp)`**を省略した場合の挙動
`fclose()`を呼び出さないと、プログラムはすぐにはクラッシュしませんが、深刻な問題を引き起こす可能性があります。

1. バッファ問題: パフォーマンス向上のため、`fprintf`などで書き込んだデータは一度メモリ上の「バッファ」に溜められ、すぐにはディスクに書き込まれません。`fclose()`は、このバッファの内容をディスクに書き込む（フラッシュする）役割を担います。これを呼び出さないと、バッファに残ったデータがファイルに保存されず、データが欠損する可能性があります。
2. リソースリーク: OSが同時に開けるファイル数には上限があります。ループ内でファイルを何度も開いて閉じないでいると、この上限に達し、やがて新しいファイルが一切開けなくなります（`fopen`が`NULL`を返すようになる）。
3. ファイルロック: OSによっては、開かれているファイルがロックされ、他のプログラムから編集・削除・移動ができなくなることがあります。

**C言語スタイルでの読み込み**
例:

```C/C++
// TxtRead.c
#include <stdio.h>

int main() {
    FILE *fp = NULL;
    char buffer[255]; // 読み込んだ1行を格納するバッファ

    // ファイルを読み込みモード("r")で開く
    fp = fopen("test_c.txt", "r");
    if (fp == NULL) {
        printf("ファイルを開く際にエラーが発生しました");
        return 1;
    }

    // ファイル終端(EOF)に達するまで1行ずつ読み込む
    printf("--- ファイルの内容 ---\n");
    while (fgets(buffer, 255, fp) != NULL) {
        printf("%s", buffer);
    }
    printf("--- 読み込み終了 ---\n");

    // ファイルをクローズする
    fclose(fp);
}
```

主な関数:

- `char* fgets(char* str, int num, FILE* stream)`: ファイルから1行（または`num-1`文字まで）を読み込む。
    - `str`: 読み込んだ文字列を格納する文字配列。
    - `num`: バッファのサイズ。
    - `stream`: `fopen`で得た`FILE`ポインタ。
    - ファイルの終端に達すると`NULL`を返す。

## C++スタイル

C++では、ファイル操作を「ストリーム（データの流れ）」として扱います。
使用するヘッダファイル: `#include <fstream>`

用途に応じて以下の2つのクラスを使い分けます。

- **`ofstream` (output file stream)**:
    - 役割: ファイルへの書き込み専用クラスです。

    - 特徴: プログラム内のデータをファイルに出力します。ファイルが存在しない場合は新規作成し、存在する場合は通常上書きします。
    - ofs << data:  coutと同様に、<< 演算子を使って様々な型のデータをファイルに書き込める。
- **`ifstream` (input file stream)**:
    - 役割: ファイルからの読み込み専用クラスです。
    - 特徴: ファイル内のデータをプログラムの変数に読み込みます。ファイルが存在しない場合はエラーになります。
    - cin と同様に、>> 演算子 を使ってファイルから変数へデータを読み込みます。

C++では、**RAII (Resource Acquisition Is Initialization)** の原則に基づいた`ofstream`(出力)と`ifstream`(入力)クラスを使います。これにより、オブジェクトがスコープを抜ける際に自動的にファイルが閉じられるため、より安全です。

**C++**スタイルでの書き込み
例:

```C/C++
// TxtWrite.cpp
#include <iostream>
#include <fstream> // ファイル操作のためのヘッダ
#include <string>

int main() {
    // 1. ofstreamオブジェクトを作成すると、自動的にファイルが開かれる
    std::ofstream ofs("test_cpp.txt");

    // 開けたかどうかのチェック
    if (!ofs) {
        std::cout << "ファイルをオープンできませんでした！" << std::endl;
        return 1;
    }

    std::string str = "INFO";
    int n = 10;
    double x = 3.5;

    // 2. << 演算子でcoutのように書き込める
    ofs << "This is testing for C++ ofstream..." << std::endl;
    ofs << "Setting: " << str << std::endl;
    ofs << "Count: " << n << std::endl;
    ofs << "Value: " << x << std::endl;

    // 3. close()は明示的に呼べるが、デストラクタが自動で呼ぶため省略可能
    // ofs.close();
```

```
    std::cout << "データが正常に書き込まれました。" << std::endl;
}
```

**C++スタイルでの読み込み**
例:

```C/C++
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::string line;

    // 1. ifstreamオブジェクトを作成し、ファイルを開く
    std::ifstream ifs("test_cpp.txt");
    if (!ifs) {
        std::cout << "ファイルを開けませんでした。" << std::endl;
        return 1;
    }

    // 2. getlineで1行ずつ読み込む
    std::cout << "--- ファイルの内容 ---" << std::endl;
    while (std::getline(ifs, line)) {
        std::cout << line << std::endl;
    }
    std::cout << "--- 読み込み終了 ---" << std::endl;

    // RAIIにより、ifsオブジェクトが破棄される際に自動でファイルが閉じられる
}
```

**主なクラス/関数:**

- `std::ifstream`: ファイルからの読み込みを行うためのクラス。
- `bool std::getline(std::istream& is, std::string& str)`: 入力ストリーム`is`から1行読み込み、`str`に格納する。読み込みが成功すれば`true`を返す。

## 文字コードの問題

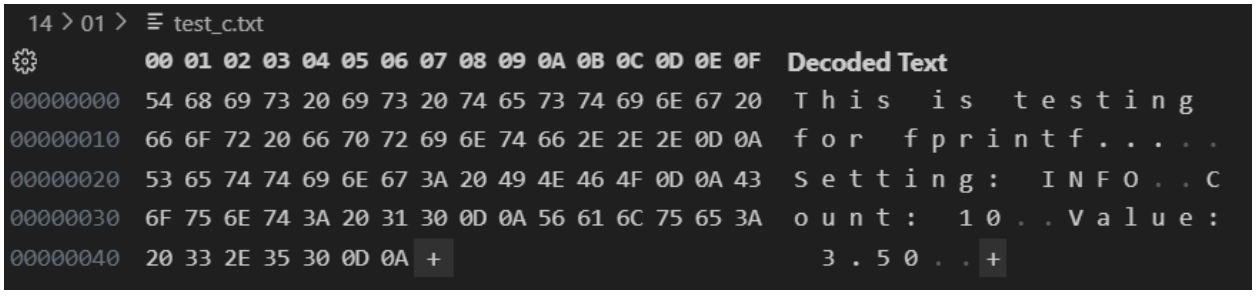
テキストファイルを扱う際、文字コード（エンコーディング）の違いが問題になることがあります。

- **ASCII**（初期の標準）: 最初期のコンピュータ標準で、1バイト（Byte）のみを使用して文字を表現します。ASCIIは7ビット（0〜127）の128文字で、英数字や基本記号のみを扱え、漢字や日本語のような複雑な文字を表現することは全くできません。
- 各国語による分断: 中国語、日本語、韓国語、アラビア語などを表示するために、各国が独自のエンコーディング方式（日本の **Shift_JIS** や中国の **GBK** など）を導入しました。これらの標準には互換性がないため、異なる言語環境のコンピュータでファイルを開くと「文字化け」が発生します。
- **UTF-8**（世界統一標準）: 言語間の互換性の問題を解決するために誕生した、現在の国際的な主流方式です。文字によって長さが変わる可変長符号を採用しており、英文字は1バイト、日本語の文字などは通常3バイトで表現されます（一部の文字は4バイトになることがあります）。世界中のほぼ全ての文字に対応しており、マルチプラットフォーム開発における推奨標準となっています。

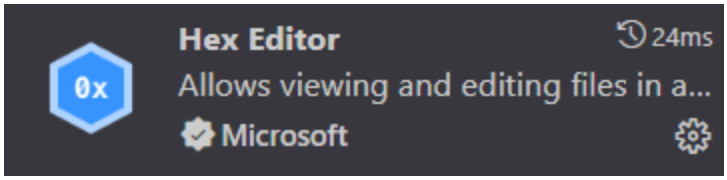
例：「国」という一文字を例に、異なるエンコーディングでどのようにバイトデータ（バイナリ）が変わるかを見てみましょう。

| エンコーディング方式 | バイト表現（16進数） | 特徴 |
|---|---|---|
| ASCII | (表現不可) | 1バイト(0-127)しかなく、日本語は扱えません。 |
| Shift_JIS (日本) | 0x8D 0xFB | 2バイト。かつてのWindows（日本語版）の標準。 |
| GBK / GB2312 (中国) | 0xB9 0xFA | 2バイト。中国大陸の旧標準。Shift_JISとは互換性なし。 |
| UTF-8 (世界標準) | 0xE5 0x9B 0xBD | 3バイト。現在の主流。世界中の文字を混在させることが可能。 |

例：test_c.txt ファイルのデータは、以下の図の通りです。英字、記号、数字は、ASCII形式で保存されます。




Tool: VS Code Extension → Hex Editor




※ ASCIIコードについては、以下をご参照ください
https://images.saymedia-content.com/.image/t_share/MTc2MjU5OTkxNjc3MjQ4Njg1/what-are-ascii-codes.gif

---

## 3. バイナリファイルの読み書き

バイナリファイルは、メモリ上のデータをそのままの形式でファイルに保存します。高速でサイズも小さいですが、異なる環境（OSやCPUアーキテクチャ）間での互換性には注意が必要です。

### C言語スタイル

主な関数:

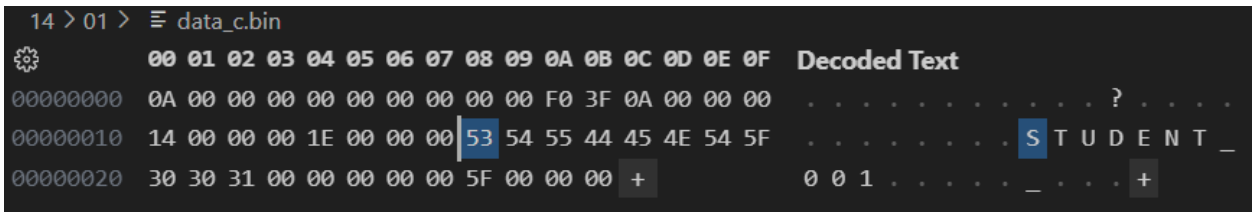
- `size_t fwrite(const void* ptr, size_t size, size_t count, FILE* stream)`: `ptr`が指すメモリから、`size`バイトのデータを`count`個、ファイルに書き込む。
- `size_t fread(void* ptr, size_t size, size_t count, FILE* stream)`: ファイルから`size`バイトのデータを`count`個読み込み、`ptr`が指すメモリに格納する。

バイナリ書き込み
例:

```C/C++
// BinWrite.c
#include <stdio.h>
#include <string.h>

typedef struct { char id[16]; int score; } Record;

int main() {
    FILE *fp = fopen("data_c.bin", "wb"); // "wb": バイナリ書き込みモード
    if (fp == NULL) {
        printf("ファイルをオープンできませんでした！\n");
        return 1;
    }

    // データの準備
```

```c
    int n = 10;
    double x = 1.00;
    int arr[3] = {10, 20, 30};
    Record rec = {"STUDENT_001", 95};

    // 整数(int)の書き込み
    fwrite(&n, sizeof(int), 1, fp);

    // 浮動小数点数(double)の書き込み
    fwrite(&x, sizeof(double), 1, fp);

    // 配列(int array)の書き込み
    fwrite(arr, sizeof(int), 3, fp);

    // 構造体(Record)の書き込み
    fwrite(&rec, sizeof(Record), 1, fp);

    fclose(fp); // クローズしてバッファをフラッシュ
    printf("バイナリデータの保存が完了しました。\n");
}
```

例：data_c.bin ファイルのデータは、以下の図の通りです。




| | | |
|---|---|---|
| Hex: 00 00 00 0A | ⇒ `int` 10 | n |
| Hex: 3F F0 00 00 00 00 00 00 | ⇒ `double` 1.0 | x |
| Hex: 00 00 00 0A | ⇒ `int` 10 | arr[0] |
| Hex: 00 00 00 14 | ⇒ `int` 20 | arr[1] |
| Hex: 00 00 00 1E | ⇒ `int` 30 | arr[2] |
| Hex: 53 | ⇒ `char` 'S' | rec.id[0] |
| Hex: 54 | ⇒ `char` 'T' | rec.id[1] |
| … | | |
| Hex: 00 00 00 5F | ⇒ `int` 95 | rec.score |

※ float 型および double 型の16進数表現については、以下をご参照ください
[Base Convert: IEEE 754 Floating Point]

バイナリ読み込み
例：

C/C++

```c
// BinRead.c
#include <stdio.h>

// 書き込み時と完全に一致する構造体定義が必要です
typedef struct
{
    char id[16];
    int score;
} Record;

int main()
{
    // 1. ファイルをバイナリ読み込みモード ("rb") でオープン
    FILE *fp = fopen("data_c.bin", "rb");
    if (!fp)
    {
        printf("ファイルを開けませんでした。\n");
        return 1;
    }

    // 読み込んだデータを格納するための変数（メモリ領域）を準備
    int n_in;
    double x_in;
    int arr_in[3];
    Record rec_in;

    // 2. データの読み込み (fread)
    // 書き込んだ時と同じ順番で、各データ型のサイズ（sizeof(...)）単位で読み戻します。

    // 整数(int)の読み込み
    fread(&n_in, sizeof(int), 1, fp);

    // 浮動小数点数(double)の読み込み
    fread(&x_in, sizeof(double), 1, fp);

    // 配列(int array)の読み込み
    fread(arr_in, sizeof(int), 3, fp);

    // 構造体(Record)の読み込み
    fread(&rec_in, sizeof(Record), 1, fp);

    // 3. 読み込んだデータの分析と表示
    printf("--- 読み込んだデータ ---\n");
    printf("整数: %d\n", n_in);
    printf("実数: %.2f\n", x_in);
    printf("配列: %d, %d, %d\n", arr_in[0], arr_in[1], arr_in[2]);
    printf("構造体 ID: %s, Score: %d\n", rec_in.id, rec_in.score);
```

```
    // 4. ファイルをクローズ
    fclose(fp);
}
```

ランダムアクセス
ファイルは先頭から順に読むだけでなく、特定の位置に直接ジャンプすることも可能です。これをランダムアクセスと呼びます。

主な関数 **(C言語スタイル):**

- `int fseek(FILE* stream, long offset, int origin)`: ファイルの読み書き位置を移動させる。
    - `origin`: `SEEK_SET` (ファイルの先頭), `SEEK_CUR` (現在位置), `SEEK_END` (ファイルの末尾) のいずれかを指定。
    - `offset`:`origin` で指定した基準位置（先頭／現在位置／末尾）からの移動量（バイト単位）。正の値で前方へ、負の値で後方へ移動する。
- `long ftell(FILE* stream)`: ファイルの先頭からの現在位置をバイト単位で返す。

例:**C言語スタイルで特定のレコードを読み込む**

```C/C++
// BinRead2.c
#include <stdio.h>

typedef struct { char id[16]; int score; } Record;

int main() {

    FILE *fp = fopen("data_c.bin", "rb");
    if (!fp)
    {
        printf("ファイルを開けませんでした。\n");
        return 1;
    }

    // 書き込み済みの「int型1つ」と「double型1つ」、さらに「int型3つ（配列）」をスキップ
    // 合計：int × 4個 + double × 1個 分のサイズを先頭から移動
    fseek(fp, sizeof(int) * 4 + sizeof(double), SEEK_SET);

    // 現在位置から1レコード分を読み込む
    Record rec_in;
```

```
    fread(&rec_in, sizeof(Record), 1, fp);

    printf("構造体 ID: %s, Score: %d\n", rec_in.id, rec_in.score);
    fclose(fp);
}
```

## C++スタイル

主なメンバ関数:

- `ostream& write(const char* s, streamsize n)`: sが指す位置からnバイトのデータをバイナリとして書き込む。
- `istream& read(char* s, streamsize n)`: ファイルからnバイトのデータを読み込み、sが指すメモリ位置に格納する。
- `reinterpret_cast`: バイナリI/Oでは、任意の型のポインタを`char*`に変換するためによく使われる。

バイナリ書き込み

```C/C++
// BinWrite.cpp
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>

using namespace std;

// 構造体の定義
struct Record
{
    char id[16];
    int score;
};

int main()
{
    // 1. バイナリモードでファイルを開く
    ofstream ofs("data_cpp.bin", ios::binary);
    if (!ofs)
    {
        cout << "書き込みファイルのオープンに失敗しました" << endl;
```

```
        return 1;
    }

    // データの準備
    int n = 10;
    double x = 1.00;
    int arr[3] = {10, 20, 30};
    Record rec = {"STUDENT_001", 95};

    // 2. メモリ内容をそのまま書き出す
    // write関数は char* 型を要求するため、reinterpret_cast を使用します
    ofs.write(reinterpret_cast<const char *>(&n), sizeof(n));
    ofs.write(reinterpret_cast<const char *>(&x), sizeof(x));
    ofs.write(reinterpret_cast<const char *>(arr), sizeof(arr));
    ofs.write(reinterpret_cast<const char *>(&rec), sizeof(rec));

    cout << "C++スタイルでバイナリ保存完了（RAIIにより自動クローズ）" << endl;
}
```

バイナリ読み込み

C/C++

```
// BinRead.cpp
#include <iostream>
#include <fstream>

using namespace std;

// 書き込み時と完全に一致する構造体定義が必要です
struct Record
{
    char id[16];
    int score;
};

int main()
{
    // 1. バイナリモードでファイルを読み込む
    ifstream ifs("data_cpp.bin", ios::binary);
    if (!ifs)
    {
        cout << "読み込みファイルのオープンに失敗しました" << endl;
        return 1;
    }
```

```
    int n_in;
    double x_in;
    int arr_in[3];
    Record rec_in;

    // 2. 書き込み時と同じ順序で読み込む
    ifs.read(reinterpret_cast<char *>(&n_in), sizeof(n_in));
    ifs.read(reinterpret_cast<char *>(&x_in), sizeof(x_in));
    ifs.read(reinterpret_cast<char *>(arr_in), sizeof(arr_in));
    ifs.read(reinterpret_cast<char *>(&rec_in), sizeof(rec_in));

    // 3. データの表示
    cout << "--- 読み込んだデータ ---" << endl;
    cout << "整数: " << n_in << endl;
    cout << "実数: " << x_in << endl;
    cout << "配列: " << arr_in[0] << ", " << arr_in[1] << ", " << arr_in[2] <<
endl;
    cout << "構造体 ID: " << rec_in.id << ", Score: " << rec_in.score << endl;

    cout << "C++スタイルでバイナリ読み込み完了（RAIIにより自動クローズ）" << endl;
}
```

バイナリI/Oの経験とアドバイス

- 厳密な型と順序: 読み書きは、完全に同じデータ構造と順序で行う必要があります。少しでもずれると、データ全体が壊れてしまいます。
- バージョン管理: 将来、構造体にメンバを追加・削除する可能性がある場合は、ファイルの先頭にバージョン情報を書き込んでおき、読み込み時にバージョンに応じた処理を行う設計が必要です。
- 互換性問題: バイナリファイルは特定の環境（CPU、OS、コンパイラ）に強く依存します。例えば、エンディアン（バイトオーダ）の違いや、構造体のアライメント（パディング）の違いにより、ある環境で書き込んだファイルが別の環境で正しく読めないことがあります。異なる環境間でデータをやり取りする場合は、JSONやXMLのようなテキストベースの形式か、Protocol Buffersのようなシリアライズライブラリを使うのが安全です。