C言語プログラミング教材

プログラミング学習教材

2024年

Table of Contents

# C言語プログラミング教材

## 目次

1. [第1章: C言語入門](#第1章-c言語入門)
2. [第2章: 基本構文](#第2章-基本構文)
3. [第3章: データ型と変数](#第3章-データ型と変数)
4. [第4章: 演算子](#第4章-演算子)
5. [第5章: 条件分岐](#第5章-条件分岐)
6. [第6章: 繰り返し処理](#第6章-繰り返し処理)
7. [第7章: 配列](#第7章-配列)
8. [第8章: 文字列処理](#第8章-文字列処理)
9. [第9章: 関数](#第9章-関数)
10. [第10章: ポインタ](#第10章-ポインタ)
11. [第11章: 構造体](#第11章-構造体)
12. [第12章: 関数ポインタ](#第12章-関数ポインタ)
13. [第13章: 高度なトピック](#第13章-高度なトピック)
14. [第14章: C23の新機能](#第14章-c23の新機能)

# 第1章: C言語入門

## 対応C規格

* **主要対象:** 全規格共通
* **学習内容:** C言語の基礎知識、開発環境構築、規格解説

## 学習目標

この章を完了すると、以下のことができるようになります：

* C言語の特徴と歴史を理解する
* 開発環境を構築し、基本的なコンパイル手順を覚える
* C言語の規格（C90, C99, C11, C17）の違いを理解する
* 簡単なプログラムをコンパイル・実行できる

## 概要と詳細

### C言語とは

C言語は1972年にデニス・リッチーがベル研究所で開発したプログラミング言語です。現在でも世界中で広く使われている、とても重要なプログラミング言語の一つです。

#### なぜC言語を学ぶのか？

初めてプログラミングを学ぶ方は「なぜC言語から始めるの？」と思うかもしれません。実は、C言語を学ぶことには大きなメリットがあります：

1. **コンピュータの仕組みが理解できる**
   * C言語はコンピュータのハードウェアに近い言語です
   * メモリやCPUがどのように動作するか実感できます
   * 他の言語を学ぶときの基礎知識になります
2. **多くのプログラミング言語の元になっている**
   * C++、Java、C#などはC言語の影響を受けています
   * C言語の文法を知っていれば、他の言語も学びやすくなります
3. **今でも現役で使われている**
   * 50年以上の歴史がありますが、今でも重要な場面で使われています
   * 特に高速性や効率性が求められる場面では欠かせません

#### 主な特徴

C言語には以下のような特徴があります。初心者には少し難しく感じるかもしれませんが、一つずつ理解していけば大丈夫です：

* **システムプログラミング向け**:
  + OS（オペレーティングシステム）の開発に使われます
  + ハードウェアを直接制御できます
  + 例：LinuxカーネルはC言語で書かれています
* **高い移植性**:
  + 一度書いたプログラムが様々なコンピュータで動きます
  + Windows、Mac、Linuxなど、異なるOSでも同じコードが使えます
  + これを「Write Once, Run Anywhere（一度書けば、どこでも動く）」と言います
* **効率性**:
  + 実行速度が非常に速い
  + メモリの使用量を最小限に抑えられる
  + 限られたリソースで動作させる必要がある場面で重宝されます
* **シンプルな文法**:
  + 言語の機能が基本的なものに絞られています
  + 覚えることが比較的少ない
  + ただし、その分プログラマーが考えることは多くなります

#### 使用分野

C言語は以下のような分野で活躍しています：

* **オペレーティングシステム（OS）**
  + Linux：世界中のサーバーで使われているOS
  + Windows：一部のコア部分
  + macOS：カーネル部分
* **組込みシステム**
  + 家電製品（エアコン、洗濯機、電子レンジなど）
  + 自動車の制御システム
  + IoTデバイス（スマートウォッチ、センサーなど）
* **データベースシステム**
  + MySQL、PostgreSQLなどの有名なデータベース
  + 高速なデータ処理が必要なため
* **ゲームエンジン**
  + ゲームの基盤となる部分
  + 3Dグラフィックスの処理など
* **科学技術計算**
  + 物理シミュレーション
  + 数値解析
  + 機械学習の基礎ライブラリ

### C言語の規格

C言語は時代とともに進化し、複数の規格（バージョン）が策定されています。プログラミング言語にも「バージョン」があることを不思議に思うかもしれませんが、これは言語をより使いやすく、より強力にするための改良の歴史です。

#### C言語規格の歴史

| 規格名 | 発表年 | 正式名称 | 主な特徴 |
| --- | --- | --- | --- |
| **C90** | 1990年 | ISO/IEC 9899:1990 | 初の国際標準、ANSI C |
| **C99** | 1999年 | ISO/IEC 9899:1999 | \_Bool型、可変長配列、inline関数 |
| **C11** | 2011年 | ISO/IEC 9899:2011 | 匿名構造体、\_Generic、マルチスレッド |
| **C17** | 2018年 | ISO/IEC 9899:2018 | C11のバグ修正版 |

#### 各規格の詳しい説明

**C90（ANSI C）** - C言語の最初の国際標準規格です - 最も基本的で、すべてのC言語コンパイラが対応しています - シンプルで学習しやすく、初心者にお勧めです - 古い組込みシステムでも確実に動作します

**C99** - 多くの便利な機能が追加されました - 例：forループの中で変数を宣言できるようになりました c for (int i = 0; i < 10; i++) { /\* C99から可能 \*/ } - 現在最も広く使われている規格です

**C11** - マルチスレッド（並列処理）のサポートが追加されました - より安全なプログラミングのための機能が強化されました - 比較的新しいため、すべての環境で使えるとは限りません

**C17** - C11の細かいバグを修正したマイナーアップデート - 新機能の追加はほとんどありません

#### どの規格を選ぶべきか？

初学者の方への推奨：

* **学習目的なら**: **C90から始める**
  + 最も基本的な機能を確実に習得できます
  + どこでも動く汎用的なコードが書けるようになります
  + 段階的に新しい機能を学んでいけます
* **実用的なプログラムを書くなら**: **C99を使う**
  + 便利な機能が使えて、生産性が向上します
  + ほとんどの環境で問題なく動作します
  + 現在の業界標準と言えます
* **組込みシステムを作るなら**: **C90またはC99**
  + 使用するマイコンやコンパイラの対応状況を確認してください
  + 古い環境ではC90しか使えない場合があります

#### 規格による違いの例

/\* C90スタイル \*/  
int main(void) {  
 int i; /\* 変数宣言は関数の先頭で \*/  
   
 for (i = 0; i < 10; i++) {  
 printf("%d\n", i);  
 }  
 return 0;  
}  
  
/\* C99以降のスタイル \*/  
int main(void) {  
 for (int i = 0; i < 10; i++) { /\* forループ内で宣言可能 \*/  
 printf("%d\n", i);  
 }  
 return 0;  
}

この教材では、基本的にC90準拠のコードを示し、必要に応じてC99版も提供します。これにより、どんな環境でも動作するコードの書き方を学びながら、新しい機能も理解できるようになっています。

### 開発環境

C言語でプログラミングを始めるには、いくつかのツール（ソフトウェア）が必要です。最初は難しく感じるかもしれませんが、一度セットアップすればずっと使えます。

#### 必須ツール

プログラミングには以下の4つのツールが必要です：

1. **テキストエディター**: プログラムを書くためのソフト
   * メモ帳でも書けますが、専用エディターが便利です
   * 推奨：Visual Studio Code（無料）、Sublime Text、Atom
   * プログラミング用エディターは色分け表示などの便利機能があります
2. **コンパイラ**: 人間が書いたコードをコンピュータが理解できる形に変換
   * C言語で書いたテキストを実行可能なプログラムに変換します
   * 主なコンパイラ：GCC（無料）、Clang（無料）、Visual C++（Windows）
   * この教材ではGCCを使用します
3. **リンカー**: プログラムの部品を組み合わせる
   * 通常はコンパイラに含まれているので、別途インストールは不要です
   * 複数のファイルからなるプログラムを一つにまとめます
4. **デバッガー**: プログラムの誤り（バグ）を見つけるツール
   * プログラムを一行ずつ実行して動作を確認できます
   * GDB（GCCに付属）が一般的です

#### OS別の環境構築手順

お使いのOSに応じて、以下の手順で環境を構築してください：

**Windows:**

最も簡単な方法はMinGW-w64をインストールすることです：

1. MinGW-w64のインストーラーをダウンロード
2. インストール時の設定：
   * Architecture: x86\_64（64ビット版）
   * Threads: posix
   * Exception: seh
3. 環境変数PATHにMinGWのbinフォルダを追加
4. コマンドプロンプトで gcc --version と入力して確認

または、Microsoft Visual Studio Community（無料）も使えます： - C++の開発環境をインストール - 統合開発環境（IDE）なので初心者に優しい

**macOS:**

macOSは開発者に優しいOSです：

# ターミナルを開いて以下のコマンドを実行  
xcode-select --install

これだけで、GCCとその他の開発ツールがインストールされます。

**Linux（Ubuntu/Debian）:**

Linuxは最初から開発環境が整っていることが多いです：

# パッケージリストを更新  
sudo apt update  
  
# 開発ツール一式をインストール  
sudo apt install build-essential  
  
# 確認  
gcc --version

#### 環境構築の確認

インストールが完了したら、以下のコマンドで確認しましょう：

# コンパイラのバージョン確認  
gcc --version  
  
# 簡単なプログラムを作成してテスト  
echo 'int main(void) { return 0; }' > test.c  
gcc test.c -o test  
./test # Windowsの場合は test.exe  
echo $? # 0 が表示されれば成功

### コンパイル手順

C言語のプログラムが実行されるまでの流れを理解しましょう。料理に例えると：

1. **ソースコード作成**（レシピを書く）
   * 人間が理解できるC言語でプログラムを書きます
   * ファイルの拡張子は .c です（例：hello.c）
2. **プリプロセッサ処理**（材料の下準備）
   * #include で指定したファイルを取り込みます
   * #define で定義した値を置き換えます
   * コンパイラが処理する前の準備作業です
3. **コンパイル**（調理）
   * C言語のコードをアセンブリ言語に変換します
   * 文法エラーがあればここで検出されます
   * まだ実行はできません
4. **アセンブル**（盛り付け）
   * アセンブリ言語を機械語（0と1）に変換します
   * オブジェクトファイル（.o または .obj）が作成されます
5. **リンク**（完成）
   * 必要なライブラリと結合します
   * 実行可能ファイルが生成されます
   * Windowsでは .exe ファイル、Unix系では拡張子なし
6. **実行**（食べる）
   * 作成したプログラムを実行します

#### 基本的なコンパイルコマンド

実際のコンパイル方法を見てみましょう：

# シンプルな方法（すべての処理を一度に）  
gcc hello.c -o hello  
# hello.c をコンパイルして hello という実行ファイルを作成  
  
# 実行  
./hello # Unix/Linux/macOS  
hello # Windows  
  
# より詳細なオプション付き  
gcc -Wall -Wextra -o hello hello.c  
# -Wall: すべての警告を表示  
# -Wextra: 追加の警告も表示  
# -o hello: 出力ファイル名を指定  
  
# 段階的なコンパイル（大きなプログラムで使用）  
gcc -c hello.c # hello.o を作成  
gcc -c another.c # another.o を作成  
gcc hello.o another.o -o program # リンクして実行ファイル作成

#### よくあるエラーと対処法

初心者がよく遭遇するエラー：

1. **“gcc: command not found”**
   * コンパイラがインストールされていません
   * PATHが設定されていません
2. **“hello.c: No such file or directory”**
   * ファイル名が間違っています
   * 現在のディレクトリにファイルがありません
3. **“undefined reference to `main’”**
   * main関数が定義されていません
   * C言語のプログラムには必ずmain関数が必要です

## サンプルコード

### 環境確認プログラム

プログラムファイル: examples/environment\_check.c

このプログラムは開発環境が正しく設定されているかを確認します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic environment\_check.c -o env\_check\_c90  
  
# C99準拠でコンパイル  
gcc -std=c99 -Wall -Wextra -pedantic environment\_check.c -o env\_check\_c99  
  
# C11準拠でコンパイル  
gcc -std=c11 -Wall -Wextra -pedantic environment\_check.c -o env\_check\_c11  
  
# C17準拠でコンパイル  
gcc -std=c17 -Wall -Wextra -pedantic environment\_check.c -o env\_check\_c17  
  
# 実行  
./env\_check\_c90

## 演習課題

### 基礎問題

1. **環境構築確認**
   * サンプルプログラムを正常にコンパイル・実行できることを確認してください
   * C90、C99、C11、C17の各規格でコンパイルを試してください
2. **コンパイラ情報の確認**
   * 使用しているコンパイラの名前とバージョンを調べてください
   * 対応している規格を確認してください

### 応用問題

1. **規格比較**
   * C90とC99の主な違いを3つ以上挙げてください
   * 組込み開発でC90が今でも使われる理由を考えてください
2. **コンパイル手順の理解**
   * プリプロセッサ、コンパイラ、リンカーの役割を説明してください
   * -E, -S, -c オプションの効果を確認してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# C90準拠でコンパイル  
make STANDARD=c90  
  
# C99準拠でコンパイル   
make STANDARD=c99  
  
# C11準拠でコンパイル  
make STANDARD=c11  
  
# C17準拠でコンパイル  
make STANDARD=c17  
  
# クリーンアップ  
make clean

## 規格による違い

この章では基本的な環境確認のため、規格による大きな違いはありません。ただし、コンパイラの警告やエラーメッセージに違いが出る場合があります。

## 次の章へ

環境構築が完了したら、[基本文法・Hello World](../basics-syntax/README.md) に進んでください。

## 参考資料

* [GCC公式ドキュメント](https://gcc.gnu.org/documentation.html)
* [C言語規格書 (ISO/IEC 9899:2018)](https://www.iso.org/standard/74528.html)
* [GNU Make マニュアル](https://www.gnu.org/software/make/manual/)

## サンプルコード

### environment\_check.c

/\*  
 \* environment\_check.c - C言語開発環境確認プログラム  
 \*  
 \* このプログラムは開発環境が正しく設定されているかを確認します。  
 \* 各種C規格でコンパイルして動作を確認してください。  
 \*/  
  
#include <stdio.h>  
#include <limits.h>  
  
/\*  
 \* 環境情報を表示する関数  
 \*/  
void print\_environment\_info(void)  
{  
 printf("=== C言語開発環境確認 ===\n\n");  
  
 /\* コンパイラ情報 \*/  
 printf("コンパイラ情報:\n");  
  
#ifdef \_\_GNUC\_\_  
 printf("- GCC version: %d.%d.%d\n", \_\_GNUC\_\_, \_\_GNUC\_MINOR\_\_, \_\_GNUC\_PATCHLEVEL\_\_);  
#endif  
  
#ifdef \_MSC\_VER  
 printf("- Microsoft Visual C++ version: %d\n", \_MSC\_VER);  
#endif  
  
#ifdef \_\_clang\_\_  
 printf("- Clang version: %d.%d.%d\n", \_\_clang\_major\_\_, \_\_clang\_minor\_\_, \_\_clang\_patchlevel\_\_);  
#endif  
  
 /\* C規格情報 \*/  
 printf("\nC規格情報:\n");  
  
#ifdef \_\_STDC\_VERSION\_\_  
 printf("- \_\_STDC\_VERSION\_\_: %ldL\n", \_\_STDC\_VERSION\_\_);  
  
 if (\_\_STDC\_VERSION\_\_ >= 201710L)  
 {  
 printf("- 対応規格: C17 (C18)\n");  
 }  
 else if (\_\_STDC\_VERSION\_\_ >= 201112L)  
 {  
 printf("- 対応規格: C11\n");  
 }  
 else if (\_\_STDC\_VERSION\_\_ >= 199901L)  
 {  
 printf("- 対応規格: C99\n");  
 }  
 else  
 {  
 printf("- 対応規格: C90 (ANSI C)\n");  
 }  
#else  
 printf("- 対応規格: C90 (ANSI C) または未定義\n");  
#endif  
  
 /\* システム情報 \*/  
 printf("\nシステム情報:\n");  
  
 printf("- int型サイズ: %lu bytes\n", (unsigned long)sizeof(int));  
 printf("- long型サイズ: %lu bytes\n", (unsigned long)sizeof(long));  
 printf("- ポインタサイズ: %lu bytes\n", (unsigned long)sizeof(void \*));  
 printf("- char型範囲: %d 〜 %d\n", CHAR\_MIN, CHAR\_MAX);  
 printf("- int型範囲: %d 〜 %d\n", INT\_MIN, INT\_MAX);  
}  
  
/\*  
 \* 基本的な計算テスト  
 \*/  
void test\_basic\_operations(void)  
{  
 int a = 10;  
 int b = 3;  
 int result;  
  
 printf("\n=== 基本演算テスト ===\n");  
  
 result = a + b;  
 printf("%d + %d = %d\n", a, b, result);  
  
 result = a - b;  
 printf("%d - %d = %d\n", a, b, result);  
  
 result = a \* b;  
 printf("%d \* %d = %d\n", a, b, result);  
  
 result = a / b;  
 printf("%d / %d = %d\n", a, b, result);  
  
 result = a % b;  
 printf("%d %% %d = %d\n", a, b, result);  
  
 printf("基本演算テスト: 正常完了\n");  
}  
  
/\*  
 \* メイン関数  
 \*/  
int main(void)  
{  
 printf("C言語開発環境確認プログラム\n");  
 printf("コンパイル日時: %s %s\n\n", \_\_DATE\_\_, \_\_TIME\_\_);  
  
 /\* 環境情報の表示 \*/  
 print\_environment\_info();  
  
 /\* 基本演算のテスト \*/  
 test\_basic\_operations();  
  
 printf("\n=== 環境確認完了 ===\n");  
 printf("この出力が正常に表示されれば、開発環境は正しく設定されています。\n");  
  
 return 0;  
}

# 第2章: 基本構文

## 対応C規格

* **主要対象:** C90
* **学習内容:** プログラムの基本構造、main関数、printf関数

## 学習目標

この章を完了すると、以下のことができるようになります：

* C言語プログラムの基本構造を理解する
* main関数の役割を理解する
* printf関数を使った文字列出力ができる
* コメントの書き方を覚える
* 基本的なエスケープシーケンスを使える

## 概要と詳細

### はじめてのC言語プログラム

プログラミングの学習は「Hello, World!」と画面に表示するプログラムから始まるのが伝統です。この章では、最初のC言語プログラムを作成し、その構造を詳しく学びます。

### プログラムの基本構造

まず、最もシンプルなC言語プログラムを見てみましょう：

#include <stdio.h> /\* ヘッダーファイルのインクルード \*/  
  
int main(void) /\* メイン関数の定義 \*/  
{  
 printf("Hello, World!\n"); /\* 処理文 \*/  
 return 0; /\* 戻り値 \*/  
}

このたった6行のプログラムにも、C言語の重要な要素がすべて含まれています。料理のレシピのように、各行には重要な役割があります。

#### 各部分の詳しい説明

1. **プリプロセッサ指令** (#include <stdio.h>)
   * プログラムの「準備」をする行です
   * stdio.h（Standard Input/Output Header）は「標準入出力ヘッダー」という意味
   * printfのような入出力関数を使うために必要です
   * 例えるなら、料理を始める前に必要な道具を準備するようなものです
2. **main関数** (int main(void))
   * C言語プログラムの「心臓部」です
   * プログラムは必ずここから実行が始まります
   * intは「整数を返す」という意味
   * voidは「引数を受け取らない」という意味
   * どんなに大きなプログラムでも、main関数は1つだけです
3. **関数本体**（{ と } で囲まれた部分）
   * 実際の処理を書く場所です
   * {（開き波括弧）で始まり、}（閉じ波括弧）で終わります
   * この中に書かれた命令が順番に実行されます
   * インデント（字下げ）をして読みやすくするのが慣習です
4. **処理文** (printf("Hello, World!\n");)
   * 実際に何かをする命令です
   * printfは画面に文字を表示する関数
   * 文の最後には必ず;（セミコロン）を付けます
   * これを忘れるとコンパイルエラーになります
5. **return文** (return 0;)
   * プログラムの終了を示します
   * 0は「正常終了」を意味する特別な数字
   * エラーが発生した場合は0以外の数字を返すことがあります

#### なぜこの構造なのか？

初心者の方は「なぜこんなに複雑なの？」と思うかもしれません。実は、この構造には理由があります：

* **#include**: 車輪の再発明を避けるため
  + すでに作られた便利な機能を使えるようにします
  + printf関数も誰かが作ってくれた機能です
* **main関数**: プログラムの開始点を明確にするため
  + コンピュータがどこから実行すればよいか迷わないようにします
* **波括弧{}**: プログラムの範囲を明確にするため
  + どこからどこまでがmain関数なのかを示します
* **セミコロン;**: 文の終わりを明確にするため
  + 日本語の「。」のような役割です

### main関数の詳細

main関数はプログラムの**エントリーポイント**（実行開始点）です。野球で言えばピッチャーマウンド、演劇で言えば主役が最初に登場する場所のようなものです。

#### main関数の2つの形式

C言語のmain関数には主に2つの書き方があります：

**1. 基本形式（引数なし）:**

int main(void)  
{  
 /\* プログラムの処理 \*/  
 return 0;  
}

この形式は最もシンプルで、初心者の学習に適しています。voidは「何も受け取らない」という意味です。

**2. コマンドライン引数を受け取る形式:**

int main(int argc, char \*argv[])  
{  
 /\* プログラムの処理 \*/  
 return 0;  
}

この形式は、プログラム実行時に追加情報を受け取りたい場合に使います。例えば：

./myprogram file.txt -v

このように実行すると、file.txtや-vという情報をプログラム内で使えます。 （詳しくは後の章で学習します）

#### 戻り値の意味と重要性

main関数のreturnは、プログラムがどのように終了したかをOSに伝える重要な役割があります：

* **0**: 「すべて正常に終わりました」という合図
  + 成功を示す世界共通の値です
  + プログラムが期待通りに動作したことを意味します
* **0以外**: 「何か問題がありました」という合図
  + 1, 2, 3…など、異なる数字で異なるエラーを表現できます
  + 例：1 = ファイルが見つからない、2 = メモリ不足など

#### なぜ戻り値が重要なのか？

プログラムは単独で動くだけでなく、他のプログラムと連携することがあります：

# プログラムAが成功したらプログラムBを実行  
./programA && ./programB  
  
# プログラムCが失敗したらエラーメッセージを表示  
./programC || echo "エラーが発生しました"

このような連携を可能にするのが、main関数の戻り値です。

#### よくある疑問

**Q: なぜintなの？** A: 戻り値として整数（integer）を返すためです。成功/失敗を数字で表現します。

**Q: void main()と書いている例を見たけど？** A: 古い書き方で、現在は推奨されません。必ずint main()を使いましょう。

**Q: return 0;を忘れたらどうなる？** A: C99以降では自動的に0が返されますが、明示的に書くことを推奨します。

### printf関数

printf関数は画面に文字を表示するための関数です。「print formatted（フォーマット付き出力）」の略で、単純な文字列だけでなく、数値や変数の値も表示できる強力な機能です。

#### 基本的な使い方

最もシンプルな使い方から始めましょう：

printf("Hello, World!\n");

この1行で「Hello, World!」と表示して改行します。\nは改行を意味する特殊な記号です。

#### なぜprintf関数を使うのか？

コンピュータとの対話には出力が欠かせません： - 計算結果を確認したい - プログラムの動作状況を知りたい - エラーメッセージを表示したい

これらすべてにprintf関数を使います。

#### フォーマット指定子（変数の値を表示する魔法）

printf関数の真の力は、変数の値を表示できることです。%記号を使った「フォーマット指定子」がその鍵です：

| 指定子 | データ型 | 説明 | 例 |
| --- | --- | --- | --- |
| %d | int | 整数（decimal） | printf("年齢は%d歳です", 20); → 年齢は20歳です |
| %c | char | 1文字（character） | printf("成績は%cです", 'A'); → 成績はAです |
| %s | char\* | 文字列（string） | printf("名前は%sです", "太郎"); → 名前は太郎です |
| %f | double | 小数（float） | printf("円周率は%fです", 3.14); → 円周率は3.140000です |

#### 複数の値を同時に表示

printf関数では、複数の値を一度に表示できます：

int age = 20;  
double height = 170.5;  
printf("年齢: %d歳、身長: %.1fcm\n", age, height);  
/\* 出力: 年齢: 20歳、身長: 170.5cm \*/

%.1fは「小数点以下1桁まで表示」という意味です。

#### エスケープシーケンス（特殊文字の表現）

画面に表示できない特殊な文字を表現するための記号です。\（バックスラッシュ）で始まります：

| シーケンス | 意味 | 使用例と効果 |
| --- | --- | --- |
| \n | 改行（newline） | printf("1行目\n2行目"); → 2行に分けて表示 |
| \t | タブ（tab） | printf("名前\t年齢"); → 名前と年齢の間に大きな空白 |
| \" | ダブルクォート | printf("彼は\"天才\"だ"); → 彼は”天才”だ |
| \\ | バックスラッシュ | printf("C:\\Program Files"); → C:Files |

#### よくある使用例

/\* 基本的な挨拶 \*/  
printf("こんにちは！\n");  
  
/\* 計算結果の表示 \*/  
int a = 10, b = 20;  
printf("%d + %d = %d\n", a, b, a + b); /\* 10 + 20 = 30 \*/  
  
/\* 表形式での表示 \*/  
printf("商品名\t\t価格\t個数\n");  
printf("りんご\t\t100円\t3個\n");  
printf("みかん\t\t80円\t5個\n");  
  
/\* エラーメッセージ \*/  
printf("エラー: ファイル \"%s\" が見つかりません。\n", "data.txt");

#### 初心者がよく間違えるポイント

1. **%の数と値の数が一致しない**

* /\* NG: %dが2つあるのに値が1つしかない \*/  
  printf("%d + %d = ", 10); /\* エラーまたは予期しない動作 \*/  
    
  /\* OK: 正しい書き方 \*/  
  printf("%d + %d = %d", 10, 20, 30);

1. \*\*\*

* printf("Hello");  
  printf("World"); /\* HelloWorld と続けて表示される \*/  
    
  /\* 改行したい場合 \*/  
  printf("Hello\n");  
  printf("World\n");

1. **%と変数の型が一致しない**

* double pi = 3.14;  
  /\* NG: doubleに%dを使用 \*/  
  printf("円周率は%dです", pi); /\* おかしな値が表示される \*/  
    
  /\* OK: doubleには%f \*/  
  printf("円周率は%fです", pi);

### コメント

プログラムに説明を追加するためのコメント記法：

#### C90スタイル

/\* これは複数実行にわたる  
 コメントです \*/  
  
/\* 1行コメント \*/

#### C99以降のスタイル

// これは1行コメントです（C99以降）

**重要:** この章ではC90準拠のため、/\* \*/ スタイルを使用します。

### 基本的なプログラム例

#### 例: 単純な文字列出力

#include <stdio.h>  
  
int main(void)  
{  
 printf("Hello, World!\n");  
 return 0;

#### 例: 複数実行の出力

#include <stdio.h>  
  
int main(void)  
{  
 printf("C言語学習教材\n");  
 printf("基本文法・Hello World\n");  
 printf("プログラムが正常に動作しています！\n");  
 return 0;

#### 例: エスケープシーケンスの使用

#include <stdio.h>  
  
int main(void)  
{  
 printf("タブ区切り:\tアイテム1\tアイテム2\n");  
 printf("引用符の表示: \"Hello, World!\"\n");  
 printf("パス表示: C:\\Program Files\\MyApp\n");  
 return 0;

## サンプルコード

### Hello Worldプログラム

プログラムファイル: examples/hello\_world.c

もっとも基本的なC言語プログラムです。

### フォーマット出力プログラム

プログラムファイル: examples/formatted\_output.c

printf関数のさまざまな使い方を学習します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic hello\_world.c -o hello\_world  
  
# 実行  
./hello\_world

## 演習課題

### 基礎問題

1. **Hello World**
   * 自分の名前を表示するプログラムを作成してください
2. **複数実行出力**
   * 好きな詩や歌詞を3実行以上で表示するプログラムを作成してください
3. **エスケープシーケンス**
   * タブとダブルクォートを使った表示プログラムを作成してください

### 応用問題

1. **フォーマット文字列**
   * printf関数のさまざまなフォーマット指定子を試してください
2. **プログラム構造の理解**
   * main関数の戻り値を変更して動作を確認してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make hello\_world  
  
# 実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* コメントは /\* \*/ 形式のみ
* 変数宣言は関数やブロックの先頭で実行う
* forループ内での変数宣言は不可

### C99以降の拡張

* // 形式のコメント追加
* forループ内での変数宣言が可能
* 複合リテラル等の新機能

## 次の章へ

基本文法を理解したら、[データ型と変数](../data-types/README.md) に進んでください。

## 参考資料

* [printf関数リファレンス](https://ja.cppreference.com/w/c/io/fprintf)
* [エスケープシーケンス一覧](https://ja.cppreference.com/w/c/language/escape)
* [C90規格仕様](https://www.iso.org/standard/17782.html)

## サンプルコード

### formatted\_output.c

/\*  
 \* ファイル名: formatted\_output.c  
 \* 説明: printf関数のさまざまなフォーマット指定子を学習  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void)  
{  
 /\* 基本的な文字列出力 \*/  
 printf("=== C言語フォーマット出力 ===\n");  
  
 /\* 整数の出力 \*/  
 printf("整数: %d\n", 42);  
 printf("負の整数: %d\n", -123);  
  
 /\* 文字の出力 \*/  
 printf("文字: %c\n", 'A');  
 printf("文字コード: %c\n", 65); /\* Aのアスキーコード \*/  
  
 /\* 文字列の出力 \*/  
 printf("文字列: %s\n", "Hello, World!");  
  
 /\* 浮動小数点数の出力 \*/  
 printf("浮動小数点数: %f\n", 3.14159);  
 printf("小数点以下2桁: %.2f\n", 3.14159);  
  
 /\* エスケープシーケンスの使用 \*/  
 printf("\n--- エスケープシーケンス ---\n");  
 printf("改行\nとタブ\tの使用例：\n");  
 printf("項目1\t項目2\t項目3\n");  
 printf("値A\t値B\t値C\n");  
  
 /\* 特殊文字の表示 \*/  
 printf("\n--- 特殊文字の表示 ---\n");  
 printf("ダブルクォート: \"Hello\"\n");  
 printf("バックスラッシュ: C:\\Program Files\\\n");  
  
 /\* フォーマット指定子の組み合わせ \*/  
 printf("\n--- 複合表示 ---\n");  
 printf("名前: %s, 年齢: %d歳, 身長: %.1fcm\n", "田中太郎", 25, 175.5);  
  
 return 0;  
}

C99版は[こちら](formatted_output_c99.c)を参照してください。

### hello\_world.c

/\*  
 \* ファイル名: hello\_world.c  
 \* 説明: C言語の最も基本的なプログラム  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void)  
{  
 printf("Hello, World!\n");  
 return 0;  
}

C99版は[こちら](hello_world_c99.c)を参照してください。

# 第3章: データ型と変数

## 対応C規格

* **主要対象:** C90
* **学習内容:** 基本データ型、変数宣言、初期化、スコープ、変数の生存期間

## 学習目標

この章を完了すると、以下のことができるようになります：

* C言語の基本データ型を理解する
* 変数の宣言と初期化ができる
* 変数のスコープ（有効範囲）を理解する
* データ型のサイズと範囲を把握する
* 適切な型を選択してプログラムを作成できる

## 概要と詳細

### データ型と変数の世界へようこそ

プログラミングでは、様々な種類のデータを扱います。数字、文字、文章など、それぞれのデータには適切な「入れ物」が必要です。この「入れ物」がデータ型で、その入れ物に名前を付けたものが変数です。

例えば、年齢を記録するなら整数の入れ物（int型）、身長を記録するなら小数の入れ物（double型）を使います。この章では、C言語で使えるデータ型と、変数の使い方を学びます。

### C言語の基本データ型

C言語には、扱うデータの種類に応じて様々なデータ型が用意されています。まずは最も基本的な型から見ていきましょう。

#### なぜデータ型が必要なのか？

コンピュータのメモリは「0」と「1」の集まりです。データ型は、この0と1の並びをどのように解釈するかを決める「ルール」です。同じ0と1の並びでも、整数として読むか、文字として読むかで意味が変わります。

また、データ型によってメモリの使用量も変わります。大きな数を扱うには大きな入れ物が必要ですが、小さな数なら小さな入れ物で十分です。適切なデータ型を選ぶことで、効率的なプログラムが書けます。

#### 整数型（整数を扱う型）

整数型は、小数点のない数（…、-2、-1、0、1、2、…）を扱うための型です。

| 型名 | サイズ | 範囲 | 使用例 |
| --- | --- | --- | --- |
| char | 1バイト | -128 〜 127 | 文字（‘A’、‘B’）、小さな数 |
| unsigned char | 1バイト | 0 〜 255 | 文字コード、画像のピクセル値 |
| short | 2バイト | -32,768 〜 32,767 | 音声データ、座標値 |
| unsigned short | 2バイト | 0 〜 65,535 | ポート番号、色情報 |
| int | 4バイト | 約-21億 〜 約21億 | 一般的な整数（年齢、個数など） |
| unsigned int | 4バイト | 0 〜 約42億 | 大きな正の数、ビット演算 |
| long | 4/8バイト | システム依存 | より大きな整数 |
| unsigned long | 4/8バイト | システム依存 | より大きな正の整数 |

**初心者のための型選択ガイド：** - 普通の整数なら int を使う - 文字を扱うなら char を使う - 負の数が不要なら unsigned を付ける - より大きな数が必要なら long を使う

#### 浮動小数点数型（小数を扱う型）

小数点を含む数（3.14、-0.5など）を扱うための型です。

| 型名 | サイズ | 精度 | 使用例 |
| --- | --- | --- | --- |
| float | 4バイト | 約7桁 | ゲームの座標、簡単な計算 |
| double | 8バイト | 約15桁 | 科学計算、金額計算 |
| long double | 12/16バイト | システム依存 | 超高精度計算 |

**初心者のための型選択ガイド：** - 通常は double を使う（精度が高い） - メモリを節約したい場合のみ float を使う - long double は特殊な用途のみ

#### その他の型

| 型名 | 説明 | 使用例 |
| --- | --- | --- |
| void | 「型なし」を表す | 値を返さない関数、汎用ポインタ |

#### 型のサイズについて

「バイト」はコンピュータのメモリの単位です： - 1バイト = 8ビット = 8個の0または1 - 1バイトで表現できる数：256通り（2の8乗） - 4バイトで表現できる数：約42億通り（2の32乗）

サイズが大きいほど、より大きな数や、より精密な小数を扱えます。

### 変数の宣言と初期化

変数は、データを入れる「箱」のようなものです。箱を使う前に、「どんな種類の箱か」「何という名前の箱か」を決める必要があります。これが変数の宣言です。

#### 変数とは何か？

変数を現実世界に例えると： - **引き出し**：それぞれに名前（ラベル）が付いていて、中に物を入れられる - **貯金箱**：お金（数値）を入れたり、取り出したりできる - **メモ帳**：情報を書いたり、書き換えたりできる

プログラムの中で、変数は値を記憶し、必要なときに取り出せる仕組みです。

#### 基本的な変数宣言

変数を使うには、まず「宣言」が必要です：

int age; /\* 年齢を入れる整数型の箱を用意 \*/  
double height; /\* 身長を入れる小数型の箱を用意 \*/  
char grade; /\* 成績（A、B、Cなど）を入れる文字型の箱を用意 \*/

宣言の構造：

データ型 変数名;

**変数名のルール：** 1. 英字（a-z、A-Z）、数字（0-9）、アンダースコア（\_）が使える 2. 数字から始めることはできない 3. C言語の予約語（int、if、forなど）は使えない 4. 大文字と小文字は区別される（ageとAgeは別の変数）

**良い変数名の例：**

int studentAge; /\* 学生の年齢 \*/  
double roomTemperature; /\* 室温 \*/  
char firstInitial; /\* 名前のイニシャル \*/

**悪い変数名の例：**

int 2ndPlace; /\* NG: 数字で始まっている \*/  
double my-score; /\* NG: ハイフンは使えない \*/  
char int; /\* NG: 予約語は使えない \*/

#### 初期化付き宣言

変数は宣言と同時に初期値を設定できます。これを「初期化」といいます：

int count = 10; /\* 最初から10が入った箱を用意 \*/  
double pi = 3.14159; /\* 円周率の値で初期化 \*/  
char letter = 'A'; /\* 文字'A'で初期化 \*/

**なぜ初期化が重要なのか？**

初期化しない変数には「ゴミ値」（予測できない値）が入っています：

int x; /\* 初期化なし：何が入っているか分からない \*/  
printf("%d\n", x); /\* 予測不能な値が表示される \*/  
  
int y = 0; /\* 初期化あり：確実に0 \*/  
printf("%d\n", y); /\* 0が表示される \*/

**推奨：変数は必ず初期化しましょう！**

#### 複数変数の同時宣言

同じ型の変数は、まとめて宣言できます：

/\* 3つの整数変数を一度に宣言 \*/  
int x, y, z;  
  
/\* 宣言と初期化を混ぜることも可能 \*/  
int a = 1, b = 2, c; /\* aは1、bは2、cは未初期化 \*/  
  
/\* より読みやすい書き方（推奨） \*/  
int width = 100;  
int height = 200;  
int depth = 50;

#### 変数の命名規則（ベストプラクティス）

良いプログラムは、変数名を見ただけで何を表すか分かります：

**1. 意味のある名前を使う**

/\* 悪い例 \*/  
int a, b, c;  
  
/\* 良い例 \*/  
int width, height, area;

**2. 命名スタイルを統一する**

/\* キャメルケース（推奨） \*/  
int studentAge;  
double averageScore;  
  
/\* スネークケース \*/  
int student\_age;  
double average\_score;

**3. 適切な長さにする**

/\* 短すぎる \*/  
int s; /\* sが何を表すか不明 \*/  
  
/\* ちょうど良い \*/  
int score;  
  
/\* 長すぎる \*/  
int theScoreOfTheStudentInTheMathematicsExam;

### 変数の代入と演算

#### 代入演算子

int number;  
number = 10; /\* 代入 \*/  
number += 5; /\* number = number + 5 と同じ \*/  
number -= 3; /\* number = number - 3 と同じ \*/  
number \*= 2; /\* number = number \* 2 と同じ \*/  
number /= 4; /\* number = number / 4 と同じ \*/

#### 基本演算子

int a = 10, b = 3;  
int sum = a + b; /\* 加算: 13 \*/  
int diff = a - b; /\* 減算: 7 \*/  
int product = a \* b; /\* 乗算: 30 \*/  
int quotient = a / b; /\* 除算: 3 (整数除算) \*/  
int remainder = a % b; /\* 剰余: 1 \*/

### 型変換（キャスト）

#### 暗黙の型変換

int i = 42;  
double d;  
d = i; /\* int から double への自動変換 \*/

#### 明示的な型変換

double pi = 3.14159;  
int rounded;  
rounded = (int)pi; /\* 明示的にintに変換（3になる） \*/

#### 型変換の注意点

int a = 5, b = 2;  
double result;  
  
result = a / b; /\* 結果: 2.0 (整数除算後に変換) \*/  
result = (double)a / b; /\* 結果: 2.5 (実数除算) \*/

### 変数のスコープ（有効範囲）

スコープとは、変数がプログラム中のどの部分でアクセス可能かを示す概念です。

#### 1. ファイルスコープ（グローバルスコープ）

ファイル全体でアクセス可能な変数です。

#include <stdio.h>  
  
int global\_count = 100; /\* ファイルスコープ変数 \*/  
double global\_rate; /\* 初期化されていない場合は0になる \*/  
  
void function(void)  
{  
 global\_count = 200; /\* 関数内からアクセス可能 \*/  
}  
  
int main(void)  
{  
 global\_count = 150; /\* main関数からもアクセス可能 \*/  
 function();  
 printf("global\_count = %d\n", global\_count); /\* 200が出力 \*/  
 return 0;  
}

#### 2. 関数スコープ

関数内で宣言された変数は、その関数内でのみアクセス可能です。

void function\_a(void)  
{  
 int local\_a = 10; /\* function\_a内でのみ有効 \*/  
 printf("local\_a = %d\n", local\_a);  
}  
  
void function\_b(void)  
{  
 int local\_a = 20; /\* 異なる関数なので同じ名前でも別変数 \*/  
 printf("local\_a = %d\n", local\_a);  
 /\* function\_aのlocal\_aにはアクセス不可 \*/  
}

#### 3. ブロックスコープ

{ と } で囲まれたブロック内でのみアクセス可能な変数です。

int main(void)  
{  
 int main\_var = 5;  
   
 if (main\_var > 0) {  
 int block\_var = 10; /\* このifブロック内でのみ有効 \*/  
 printf("block\_var = %d\n", block\_var);  
   
 {  
 int nested\_var = 20; /\* さらに内側のブロック \*/  
 printf("nested\_var = %d\n", nested\_var);  
 /\* block\_varとmain\_varにもアクセス可能 \*/  
 }  
 /\* nested\_varはここではアクセス不可 \*/  
 }  
 /\* block\_varはここではアクセス不可 \*/  
   
 return 0;  
}

#### 4. スコープの隠蔽（シャドウイング）

内側のスコープで同じ名前の変数を宣言すると、外側の変数が隠蔽されます。

int x = 100; /\* グローバル変数 \*/  
  
int main(void)  
{  
 printf("グローバルx = %d\n", x); /\* 100が出力 \*/  
   
 {  
 int x = 200; /\* ローカル変数がグローバル変数を隠蔽 \*/  
 printf("ローカルx = %d\n", x); /\* 200が出力 \*/  
   
 {  
 int x = 300; /\* さらに内側で隠蔽 \*/  
 printf("内側x = %d\n", x); /\* 300が出力 \*/  
 }  
   
 printf("ローカルx = %d\n", x); /\* 200が出力（復活） \*/  
 }  
   
 printf("グローバルx = %d\n", x); /\* 100が出力（復活） \*/  
 return 0;  
}

### 変数の生存期間（Storage Duration）

変数がメモリ上に存在する期間を生存期間といいます。

#### 1. 自動変数（Automatic Variables）

関数やブロックに入るときに作成され、出るときに破棄される変数です。

void demo\_function(void)  
{  
 int auto\_var = 5; /\* 自動変数（autoは省略可能） \*/  
 auto int explicit\_auto = 10; /\* 明示的にauto指定 \*/  
   
 printf("auto\_var = %d\n", auto\_var);  
 /\* 関数終了時にauto\_varは破棄される \*/  
}  
  
int main(void)  
{  
 demo\_function();  
 demo\_function(); /\* 毎回新しくauto\_varが作成される \*/  
 return 0;  
}

#### 2. 静的変数（Static Variables）

プログラム開始時に作成され、プログラム終了まで存在し続ける変数です。

void counter\_function(void)  
{  
 static int count = 0; /\* 静的ローカル変数、初期化は11回のみ \*/  
   
 count++;  
 printf("関数呼出回数: %d\n", count);  
}  
  
int main(void)  
{  
 counter\_function(); /\* 1が出力 \*/  
 counter\_function(); /\* 2が出力 \*/  
 counter\_function(); /\* 3が出力 \*/  
 return 0;  
}

#### 3. 静的グローバル変数

ファイル内でのみアクセス可能なグローバル変数です。

static int file\_private = 10; /\* このファイル内でのみアクセス可能 \*/  
int file\_public = 20; /\* 他のファイルからもアクセス可能 \*/  
  
static void private\_function(void) /\* このファイル内でのみ呼出可能 \*/  
{  
 printf("file\_private = %d\n", file\_private);  
}  
  
int main(void)  
{  
 private\_function();  
 return 0;  
}

#### 4. レジスタ変数（Register Variables）

可能であればレジスタに格納される変数です（C90では推奨のみ）。

int main(void)  
{  
 register int fast\_var = 100; /\* レジスタ格納を要求 \*/  
   
 /\* レジスタ変数のアドレスは取得できない \*/  
 /\* int \*ptr = &fast\_var; <- エラー \*/  
   
 return 0;  
}

### ストレージクラス指定子まとめ

| 指定子 | スコープ | 生存期間 | 初期化 | 説明 |
| --- | --- | --- | --- | --- |
| auto | ブロック | 自動 | 未定義値 | デフォルト（省略可能） |
| static（ローカル） | ブロック | 静的 | 0 | 関数呼出間で値保持 |
| static（グローバル） | ファイル | 静的 | 0 | ファイル内限定 |
| extern | グローバル | 静的 | 他で定義 | 他ファイルの変数参照 |
| register | ブロック | 自動 | 未定義値 | レジスタ格納要求 |

### 変数の初期化規則

#### 自動変数の初期化

int main(void)  
{  
 int uninitialized; /\* 未定義値（危険） \*/  
 int initialized = 0; /\* 明示的初期化（推奨） \*/  
   
 printf("uninitialized = %d\n", uninitialized); /\* 予測不能 \*/  
 printf("initialized = %d\n", initialized); /\* 0 \*/  
   
 return 0;  
}

#### 静的変数の初期化

int global\_int; /\* 自動的に0で初期化 \*/  
static int static\_int; /\* 自動的に0で初期化 \*/  
  
int main(void)  
{  
 static int local\_static; /\* 自動的に0で初期化 \*/  
   
 printf("global\_int = %d\n", global\_int); /\* 0 \*/  
 printf("static\_int = %d\n", static\_int); /\* 0 \*/  
 printf("local\_static = %d\n", local\_static); /\* 0 \*/  
   
 return 0;  
}

### 実践的なスコープ活用例

#### カウンター関数の実装

int get\_next\_id(void)  
{  
 static int id\_counter = 1000; /\* 初期値 \*/  
   
 return id\_counter++; /\* 現在値を返してからインクリメント \*/  
}  
  
int main(void)  
{  
 printf("ID: %d\n", get\_next\_id()); /\* 1000 \*/  
 printf("ID: %d\n", get\_next\_id()); /\* 1001 \*/  
 printf("ID: %d\n", get\_next\_id()); /\* 1002 \*/  
   
 return 0;  
}

#### モジュール内状態管理

/\* 設定管理モジュールの例 \*/  
static int debug\_mode = 0; /\* モジュール内部状態 \*/  
static int max\_connections = 10;  
  
void set\_debug\_mode(int mode)  
{  
 debug\_mode = mode;  
}  
  
int is\_debug\_enabled(void)  
{  
 return debug\_mode;  
}  
  
void set\_max\_connections(int max)  
{  
 if (max > 0 && max <= 100) {  
 max\_connections = max;  
 }  
}  
  
int get\_max\_connections(void)  
{  
 return max\_connections;  
}

### 定数の定義

#### constキーワード

const int MAX\_SIZE = 100; /\* 定数の定義 \*/  
const double PI = 3.14159; /\* 浮動小数点定数 \*/

#### #defineプリプロセッサ

#define MAX\_STUDENTS 50 /\* マクロ定数 \*/  
#define TITLE "C Language Tutorial"

### データ型のサイズ確認

sizeof演算子を使ってデータ型のサイズを確認できます：

#include <stdio.h>  
  
int main(void)  
{  
 printf("char: %lu バイト\n", sizeof(char));  
 printf("int: %lu バイト\n", sizeof(int));  
 printf("double: %lu バイト\n", sizeof(double));  
 return 0;

## サンプルコード

### 基本データ型の使用例

**プログラムファイル:** <examples/data_types_demo.c> | [C99版](examples/data_types_demo_c99.c)

各データ型の宣言、初期化、出力を学習します。

/\*  
 \* ファイル名: data\_types\_demo.c  
 \* 説明: C言語の基本データ型の使用例  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void)  
{  
 /\* 整数型の変数宣言と初期化 \*/  
 char letter = 'A';  
 short small\_num = 1000;  
 int age = 25;  
 long big\_num = 1234567L;  
 unsigned int positive = 100U;  
  
 /\* 浮動小数点型の変数宣言と初期化 \*/  
 float pi\_f = 3.14f;  
 double pi\_d = 3.141592653589793;  
  
 printf("=== データ型のサイズと値 ===\n");  
  
 /\* 各データ型のサイズを表示 \*/  
 printf("char : %lu バイト, 値: %c (%d)\n",  
 sizeof(char), letter, letter);  
 printf("short : %lu バイト, 値: %d\n",  
 sizeof(short), small\_num);  
 printf("int : %lu バイト, 値: %d\n",  
 sizeof(int), age);  
 printf("long : %lu バイト, 値: %ld\n",  
 sizeof(long), big\_num);  
 printf("unsigned : %lu バイト, 値: %u\n",  
 sizeof(unsigned int), positive);  
 printf("float : %lu バイト, 値: %.6f\n",  
 sizeof(float), pi\_f);  
 printf("double : %lu バイト, 値: %.15f\n",  
 sizeof(double), pi\_d);  
  
 /\* 演算の例 \*/  
 printf("\n=== 基本演算 ===\n");  
 printf("10 + 3 = %d\n", 10 + 3);  
 printf("10 - 3 = %d\n", 10 - 3);  
 printf("10 \* 3 = %d\n", 10 \* 3);  
 printf("10 / 3 = %d (整数除算)\n", 10 / 3);  
 printf("10 %% 3 = %d (剰余)\n", 10 % 3);  
 printf("10.0 / 3.0 = %.2f (実数除算)\n", 10.0 / 3.0);  
  
 return 0;  
}

### 数値計算の応用例

**プログラムファイル:** <solutions/numerical_calc.c> | [C99版](solutions/numerical_calc_c99.c)

math.hライブラリを使用した高度な数値計算を学習します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic data\_types\_demo.c -o data\_types\_demo  
  
# 実行  
./data\_types\_demo

## 演習課題

### 基礎問題

1. **変数宣言と初期化**
   * さまざまなデータ型の変数を宣言し、初期化してください
   * 各変数の値をprintf関数で出力してください
2. **四則演算計算機**
   * 2つの数値を変数に格納し、四則演算の結果を表示するプログラムを作成してください
3. **データ型サイズの確認**
   * sizeof演算子を使って、各データ型のサイズを表示するプログラムを作成してください

### 応用問題

1. **型変換の理解**
   * 整数除算と実数除算の違いを確認するプログラムを作成してください
2. **スコープの実験**
   * グローバル変数とローカル変数を使い分けるプログラムを作成してください
3. **定数の活用**
   * constと#defineを使って定数を定義し、計算に使用してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make data\_types\_demo  
  
# 実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* 変数宣言は関数やブロックの先頭で行う必要がある
* long long型は使用できない
* \_Bool型は使用できない

### C99以降の拡張

* long long型の追加（64ビット整数）
* \_Bool型の追加（論理型）
* 複素数型（\_Complex）の追加
* 変数宣言位置の制限緩和

### 型サイズの注意点

データ型のサイズはシステムに依存するため、portable（移植可能）なプログラムを書く際は注意が必要です：

/\* 推奨: 具体的なサイズが必要な場合 \*/  
#include <stdint.h>  
int32\_t exact\_32bit; /\* C99以降 \*/

## よくある間違い

### 1. 初期化忘れ

/\* NG: 初期化せずに使用 \*/  
int count;  
printf("%d\n", count); /\* 未定義値が出力される \*/  
  
/\* OK: 初期化してから使用 \*/  
int count = 0;  
printf("%d\n", count);

### 2. 整数除算の結果

/\* NG: 期待した結果にならない \*/  
int result = 5 / 2; /\* 結果: 2 \*/  
  
/\* OK: 実数除算を使用 \*/  
double result = 5.0 / 2.0; /\* 結果: 2.5 \*/

### 3. 型の範囲超過

/\* NG: charの範囲を超える \*/  
char big\_num = 300; /\* オーバーフロー \*/  
  
/\* OK: 適切な型を使用 \*/  
int big\_num = 300;

## 次の章へ

データ型と変数を理解したら、[演算子と式](../operators/README.md) に進んでください。

## 参考資料

* [C言語データ型リファレンス](https://ja.cppreference.com/w/c/language/type)
* [sizeof演算子](https://ja.cppreference.com/w/c/language/sizeof)
* [型変換](https://ja.cppreference.com/w/c/language/conversion)

## サンプルコード

### data\_types\_demo.c

/\*  
 \* ファイル名: data\_types\_demo.c  
 \* 説明: C言語の基本データ型の使用例  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void)  
{  
 /\* 整数型の変数宣言と初期化 \*/  
 char letter = 'A';  
 short small\_num = 1000;  
 int age = 25;  
 long big\_num = 1234567L;  
 unsigned int positive = 100U;  
  
 /\* 浮動小数点型の変数宣言と初期化 \*/  
 float pi\_f = 3.14f;  
 double pi\_d = 3.141592653589793;  
  
 printf("=== データ型のサイズと値 ===\n");  
  
 /\* 各データ型のサイズを表示 \*/  
 printf("char : %lu バイト, 値: %c (%d)\n",  
 sizeof(char), letter, letter);  
 printf("short : %lu バイト, 値: %d\n",  
 sizeof(short), small\_num);  
 printf("int : %lu バイト, 値: %d\n",  
 sizeof(int), age);  
 printf("long : %lu バイト, 値: %ld\n",  
 sizeof(long), big\_num);  
 printf("unsigned : %lu バイト, 値: %u\n",  
 sizeof(unsigned int), positive);  
 printf("float : %lu バイト, 値: %.6f\n",  
 sizeof(float), pi\_f);  
 printf("double : %lu バイト, 値: %.15f\n",  
 sizeof(double), pi\_d);  
  
 /\* 演算の例 \*/  
 printf("\n=== 基本演算 ===\n");  
 printf("10 + 3 = %d\n", 10 + 3);  
 printf("10 - 3 = %d\n", 10 - 3);  
 printf("10 \* 3 = %d\n", 10 \* 3);  
 printf("10 / 3 = %d (整数除算)\n", 10 / 3);  
 printf("10 %% 3 = %d (剰余)\n", 10 % 3);  
 printf("10.0 / 3.0 = %.2f (実数除算)\n", 10.0 / 3.0);  
  
 return 0;  
}

C99版は[こちら](data_types_demo_c99.c)を参照してください。

# 第4章: 演算子

## 対応C規格

* **主要対象:** C90
* **学習内容:** 算術演算子、関係演算子、論理演算子、ビット演算子、演算子の優先順位

## 学習目標

この章を完了すると、以下のことができるようになります：

* さまざまな演算子の種類と使い方を理解する
* 演算子の優先順位と結合規則を把握する
* 複雑な式を正しく記述できる
* インクリメント・デクリメント演算子を適切に使える
* ビット演算の基本を理解する

## 概要と詳細

### 演算子の世界へようこそ

プログラミングでは、数学の計算だけでなく、データの比較、条件判断、ビット操作など、様々な「演算」を行います。演算子は、これらの操作を行うための「道具」です。

電卓を使ったことがある人なら、すでに演算子を使っています。+、-、×、÷などの記号が演算子です。C言語では、これらに加えて、プログラミング特有の便利な演算子がたくさん用意されています。

この章では、C言語の演算子を一つずつ学び、それらを組み合わせて複雑な計算や判断ができるようになることを目指します。

### 算術演算子（計算の基本）

算術演算子は、日常的な計算を行うための演算子です。小学校で習った四則演算がそのまま使えます。

| 演算子 | 意味 | 例 | 結果 | 日常での例 |
| --- | --- | --- | --- | --- |
| + | 加算（足し算） | 5 + 3 | 8 | 5個のリンゴに3個追加 |
| - | 減算（引き算） | 5 - 3 | 2 | 5個のリンゴから3個食べる |
| \* | 乗算（掛け算） | 5 \* 3 | 15 | 5個入りの箱を3箱 |
| / | 除算（割り算） | 7 / 3 | 2 | 7個のお菓子を3人で分ける |
| % | 剰余（余り） | 7 % 3 | 1 | 7個を3人で分けた余り |

#### なぜ剰余（%）演算子が重要なのか？

剰余演算子は日常生活でも意外と使われています： - **曜日の計算**：今日から10日後は何曜日？ → (今日の曜日 + 10) % 7 - **偶数・奇数の判定**：number % 2 が 0 なら偶数、1 なら奇数 - **時計の計算**：15時から10時間後は？ → (15 + 10) % 24 = 1時

/\* 偶数・奇数の判定 \*/  
int number = 17;  
if (number % 2 == 0) {  
 printf("%dは偶数です\n", number);  
} else {  
 printf("%dは奇数です\n", number);  
}

#### 整数除算の罠（初心者が必ず遭遇する問題）

C言語では、整数同士の割り算は「整数除算」になり、小数点以下が切り捨てられます：

/\* よくある間違い \*/  
int a = 7, b = 3;  
double result = a / b; /\* 期待: 2.333... 実際: 2.0 \*/  
  
/\* なぜ？ \*/  
/\* a / b の計算時点で整数除算（7 ÷ 3 = 2）が行われ、  
 その後でdoubleに変換されるため \*/  
  
/\* 正しい方法1：どちらかをdoubleにキャスト \*/  
double result = (double)a / b; /\* 結果: 2.333... \*/  
  
/\* 正しい方法2：最初からdoubleで計算 \*/  
double x = 7.0, y = 3.0;  
double result = x / y; /\* 結果: 2.333... \*/

**覚え方のコツ**： - 整数 ÷ 整数 = 整数（小数点以下は消える） - 小数 ÷ 整数 = 小数（小数として計算される） - 整数 ÷ 小数 = 小数（小数として計算される）

### 代入演算子（値を保存する）

代入演算子は、計算結果や値を変数に保存するための演算子です。=は数学の「等しい」ではなく、「右の値を左の変数に入れる」という意味です。

#### 基本の代入演算子

int score = 85; /\* 「scoreに85を入れる」と読む \*/  
score = 90; /\* scoreの値を90に更新 \*/

**重要な違い**： - 数学：x = 5 は「xは5に等しい」 - C言語：x = 5 は「xに5を代入する（入れる）」

#### 複合代入演算子（ショートカット）

プログラミングでは、「現在の値に何かを足す/引く」という操作が頻繁に行われます。複合代入演算子はこれを簡潔に書くための便利な記法です：

| 演算子 | 意味 | 例 | 等価な記述 | 使用場面 |
| --- | --- | --- | --- | --- |
| = | 代入 | a = 5 | - | 初期値設定 |
| += | 加算して代入 | a += 3 | a = a + 3 | カウントアップ、合計計算 |
| -= | 減算して代入 | a -= 2 | a = a - 2 | カウントダウン、在庫減少 |
| \*= | 乗算して代入 | a \*= 2 | a = a \* 2 | 倍増処理、累乗計算 |
| /= | 除算して代入 | a /= 3 | a = a / 3 | 平均値計算、分割処理 |
| %= | 剰余を代入 | a %= 4 | a = a % 4 | 循環処理、範囲制限 |

#### 実用的な使用例

/\* ゲームのスコア計算 \*/  
int score = 0;  
score += 100; /\* 敵を倒した：100点追加 \*/  
score += 50; /\* アイテムゲット：50点追加 \*/  
score -= 30; /\* ダメージを受けた：30点減少 \*/  
printf("現在のスコア: %d\n", score); /\* 120 \*/  
  
/\* 在庫管理 \*/  
int stock = 100;  
stock -= 15; /\* 15個販売 \*/  
stock += 50; /\* 50個入荷 \*/  
printf("在庫数: %d\n", stock); /\* 135 \*/  
  
/\* レベルアップシステム \*/  
int experience = 80;  
experience += 40; /\* 経験値40獲得 \*/  
if (experience >= 100) {  
 printf("レベルアップ！\n");  
 experience %= 100; /\* 100で割った余りが次レベルへの経験値 \*/  
}

#### なぜ複合代入演算子を使うのか？

1. **コードが短くなる**

* /\* 長い書き方 \*/  
  total\_price = total\_price + item\_price;  
    
  /\* 短い書き方 \*/  
  total\_price += item\_price;

1. **意図が明確になる**

* health -= damage; /\* 「ヘルスからダメージを引く」と直感的に理解できる \*/

1. **タイプミスを防げる**

* /\* 変数名が長い場合、タイプミスしやすい \*/  
  very\_long\_variable\_name = very\_long\_variable\_name + 1;  
    
  /\* こちらの方が安全 \*/  
  very\_long\_variable\_name += 1;

### インクリメント・デクリメント演算子（1つずつ増減）

プログラミングでは「1増やす」「1減らす」という操作が非常に多く使われます。そのため、専用の演算子が用意されています。

| 演算子 | 意味 | 前置形 | 後置形 | 使用場面 |
| --- | --- | --- | --- | --- |
| ++ | 1増やす（インクリメント） | ++a | a++ | カウンター、ループ制御 |
| -- | 1減らす（デクリメント） | --a | a-- | カウントダウン、在庫減少 |

#### なぜこの演算子が必要なのか？

/\* カウンターの例（よくあるパターン） \*/  
int count = 0;  
count = count + 1; /\* 長い書き方 \*/  
count += 1; /\* 複合代入演算子を使った書き方 \*/  
count++; /\* 最も簡潔な書き方 \*/

ループ処理では特に頻繁に使われます：

/\* 10回繰り返す \*/  
int i;  
for (i = 0; i < 10; i++) { /\* i++ で毎回1増やす \*/  
 printf("繰り返し %d 回目\n", i + 1);  
}

#### 前置と後置の違い（重要！）

この演算子には「前置形」と「後置形」があり、微妙に動作が異なります：

int a = 5, b = 5;  
int x, y;  
  
/\* 前置形：先に増やしてから使う \*/  
x = ++a; /\* ①aを6に増やす → ②xに6を代入 \*/  
printf("x = %d, a = %d\n", x, a); /\* x = 6, a = 6 \*/  
  
/\* 後置形：使ってから増やす \*/  
y = b++; /\* ①yに5を代入 → ②bを6に増やす \*/  
printf("y = %d, b = %d\n", y, b); /\* y = 5, b = 6 \*/

#### 実用的な例で理解する

/\* 商品番号の発行（後置形の活用） \*/  
int product\_id = 1000;  
printf("商品A: ID=%d\n", product\_id++); /\* 1000を表示、その後1001に \*/  
printf("商品B: ID=%d\n", product\_id++); /\* 1001を表示、その後1002に \*/  
printf("商品C: ID=%d\n", product\_id++); /\* 1002を表示、その後1003に \*/  
  
/\* 在庫チェック（前置形の活用） \*/  
int stock = 1;  
if (--stock >= 0) { /\* 先に1減らして0に、その後で判定 \*/  
 printf("在庫あり、販売可能\n");  
} else {  
 printf("在庫なし\n");  
}

#### よくある間違いと対策

/\* 間違い例1：意図しない動作 \*/  
int n = 5;  
int result = n++ + n++; /\* 未定義動作！使ってはいけない \*/  
  
/\* 正しい書き方 \*/  
int n = 5;  
int result = n + (n + 1); /\* 明確に意図を表現 \*/  
n += 2;  
  
/\* 間違い例2：条件文での使用 \*/  
int count = 10;  
if (count-- > 0) { /\* 10 > 0 を判定してからcountを9に \*/  
 /\* この時点でcountは9 \*/  
}  
  
/\* より明確な書き方 \*/  
int count = 10;  
if (count > 0) {  
 count--;  
 /\* 処理 \*/  
}

#### 使い分けのガイドライン

* **単独で使う場合**：前置・後置どちらでも同じ
* i++; /\* これと \*/  
  ++i; /\* これは同じ結果 \*/
* **値を使いながら増減する場合**：意図に応じて選択
* array[index++] = value; /\* 現在の位置に代入してから次へ \*/  
  array[++index] = value; /\* 次の位置に移動してから代入 \*/

**C99版での詳細:** [increment\_decrement\_c99.c](solutions/increment_decrement_c99.c)

### 関係演算子（比較する）

関係演算子は、2つの値を比較して「正しい（真）」か「正しくない（偽）」かを判定します。日常生活での「大きい・小さい」「同じ・違う」の判断をプログラムで行うための演算子です。

#### C言語における真偽の表現

C言語では： - **真（True）** = 0以外の値（通常は1） - **偽（False）** = 0

| 演算子 | 意味 | 例 | 結果 | 日常での例 |
| --- | --- | --- | --- | --- |
| == | 等しい | 5 == 3 | 0 (偽) | パスワードが一致するか |
| != | 等しくない | 5 != 3 | 1 (真) | 在庫が0でないか |
| < | より小さい | 5 < 3 | 0 (偽) | 年齢が18歳未満か |
| <= | 以下 | 5 <= 3 | 0 (偽) | 体重が標準以下か |
| > | より大きい | 5 > 3 | 1 (真) | 得点が合格点より高いか |
| >= | 以上 | 5 >= 3 | 1 (真) | 残高が購入金額以上か |

#### 実用的な例

/\* パスワードチェック \*/  
int password = 1234;  
int input = 1234;  
if (password == input) {  
 printf("ログイン成功\n");  
}  
  
/\* 在庫管理 \*/  
int stock = 5;  
if (stock > 0) {  
 printf("在庫あり（%d個）\n", stock);  
} else {  
 printf("在庫切れ\n");  
}  
  
/\* 年齢制限チェック \*/  
int age = 16;  
if (age >= 18) {  
 printf("成人向けコンテンツを表示\n");  
} else {  
 printf("アクセス制限中\n");  
}

#### よくある間違い：= と == の混同

int x = 5;  
  
/\* 間違い：代入になってしまう \*/  
if (x = 10) { /\* xに10を代入し、10（真）として評価 \*/  
 printf("この部分は必ず実行される\n");  
}  
  
/\* 正しい：比較 \*/  
if (x == 10) { /\* xが10と等しいか比較 \*/  
 printf("xは10です\n");  
}

**覚え方のコツ**： - = は「代入（入れる）」の矢印 → - == は「等しいか確認」の二重チェック

### 論理演算子（条件を組み合わせる）

論理演算子は、複数の条件を組み合わせて複雑な判断を行うための演算子です。

| 演算子 | 意味 | 例 | 説明 | 日常での例 |
| --- | --- | --- | --- | --- |
| && | かつ（AND） | a && b | 両方とも真なら真 | 「晴れ」かつ「暖かい」なら外出 |
| || | または（OR） | a || b | どちらか真なら真 | 「土曜」または「日曜」なら休み |
| ! | でない（NOT） | !a | 真偽を反転 | 「雨でない」なら洗濯 |

#### 真理値表で理解する

**AND（&&）の動作** | A | B | A && B | |—|—|——–| | 真 | 真 | 真 | | 真 | 偽 | 偽 | | 偽 | 真 | 偽 | | 偽 | 偽 | 偽 |

**OR（||）の動作** | A | B | A || B | |—|—|———-| | 真 | 真 | 真 | | 真 | 偽 | 真 | | 偽 | 真 | 真 | | 偽 | 偽 | 偽 |

#### 実用的な例

/\* 営業時間チェック（9時〜17時） \*/  
int hour = 14;  
if (hour >= 9 && hour <= 17) {  
 printf("営業中です\n");  
}  
  
/\* 休日判定 \*/  
int day = 6; /\* 1=月曜...7=日曜 \*/  
if (day == 6 || day == 7) {  
 printf("週末です\n");  
}  
  
/\* 割引条件（学生でない一般客） \*/  
int is\_student = 0;  
if (!is\_student) {  
 printf("一般料金です\n");  
}  
  
/\* 複雑な条件：優待条件 \*/  
int age = 65;  
int is\_member = 1;  
if ((age >= 65 || age < 18) && is\_member) {  
 printf("優待料金が適用されます\n");  
}

#### 短絡評価（ショートサーキット）

論理演算子には「短絡評価」という重要な特性があります。結果が確定した時点で、残りの評価をスキップします：

/\* &&の短絡評価：左が偽なら右は評価しない \*/  
int a = 0, b = 10;  
if (a != 0 && b / a > 5) { /\* a が 0 なので b/a は計算されない \*/  
 /\* ゼロ除算エラーを回避できる \*/  
}  
  
/\* ||の短絡評価：左が真なら右は評価しない \*/  
int is\_admin = 1;  
int has\_permission = 0;  
if (is\_admin || check\_permission()) { /\* is\_adminが真なので関数は呼ばれない \*/  
 printf("アクセス許可\n");  
}

#### 論理演算子の優先順位

/\* 括弧なしの場合（&&が||より優先） \*/  
if (a || b && c) /\* a || (b && c) として評価 \*/  
  
/\* 明確にするために括弧を使う \*/  
if ((a || b) && c) /\* 意図を明確に表現 \*/

### ビット演算子

ビットレベルで値を操作する演算子です。

| 演算子 | 意味 | 例 | 説明 |
| --- | --- | --- | --- |
| & | ビットAND | a & b | 対応するビットが両方1の場合1 |
| | | ビットOR | a | b | 対応するビットのいずれかが1の場合1 |
| ^ | ビットXOR | a ^ b | 対応するビットが異なる場合1 |
| ~ | ビット反転 | ~a | 各ビットを反転 |
| << | 左シフト | a << 2 | ビットを左に2つシフト |
| >> | 右シフト | a >> 1 | ビットを右に1つシフト |

unsigned char a = 5; /\* 00000101 \*/  
unsigned char b = 3; /\* 00000011 \*/  
  
printf("a & b = %d\n", a & b); /\* 1 (00000001) \*/  
printf("a | b = %d\n", a | b); /\* 7 (00000111) \*/  
printf("a ^ b = %d\n", a ^ b); /\* 6 (00000110) \*/  
printf("~a = %d\n", ~a); /\* 250 (11111010) \*/

**C99版での詳細:** [bitwise\_demo\_c99.c](examples/bitwise_demo_c99.c)

### 条件演算子（三項演算子）

条件に基づいて値を選択する演算子です。

条件 ? 真の場合の値 : 偽の場合の値

int a = 3, b = 7;  
int max = (a > b) ? a : b; /\* b が大きいので max = 7 \*/  
  
printf("大きい方: %d\n", max);

**C99版での詳細:** [conditional\_operator\_c99.c](solutions/conditional_operator_c99.c)

### sizeof演算子

データ型や変数のサイズを取得する演算子です。

printf("int のサイズ: %lu バイト\n", (unsigned long)sizeof(int));  
printf("double のサイズ: %lu バイト\n", (unsigned long)sizeof(double));  
  
int arr[10];  
printf("配列のサイズ: %lu バイト\n", (unsigned long)sizeof(arr));

### 演算子の優先順位

演算子には優先順位があり、計算の順序に影響します。

| 優先順位 | 演算子 | 結合規則 |
| --- | --- | --- |
| 1 | () [] -> . | 左から右 |
| 2 | ! ~ ++ -- + - \* & sizeof (型) | 右から左 |
| 3 | \* / % | 左から右 |
| 4 | + - | 左から右 |
| 5 | << >> | 左から右 |
| 6 | < <= > >= | 左から右 |
| 7 | == != | 左から右 |
| 8 | & | 左から右 |
| 9 | ^ | 左から右 |
| 10 | | | 左から右 |
| 11 | && | 左から右 |
| 12 | || | 左から右 |
| 13 | ?: | 右から左 |
| 14 | = += -= \*= /= %= &= ^= |= <<= >>= | 右から左 |
| 15 | , | 左から右 |

#### 優先順位の例

int result = 2 + 3 \* 4; /\* 結果: 14 (乗算が先) \*/  
int result = (2 + 3) \* 4; /\* 結果: 20 (括弧が先) \*/

**C99版での詳細:** [precedence\_demo\_c99.c](examples/precedence_demo_c99.c)

## サンプルコード

### 演算子の基本使用例

プログラムファイル: [operators\_demo.c](examples/operators_demo.c)  
C99版: [operators\_demo\_c99.c](examples/operators_demo_c99.c)

さまざまな演算子の使用方法を学習します。

### ビット演算の実例

プログラムファイル: [bitwise\_demo.c](examples/bitwise_demo.c)  
C99版: [bitwise\_demo\_c99.c](examples/bitwise_demo_c99.c)

ビット演算の動作を可視化して確認します。

### 演算子優先順位の確認

プログラムファイル: [precedence\_demo.c](examples/precedence_demo.c)  
C99版: [precedence\_demo\_c99.c](examples/precedence_demo_c99.c)

演算子の優先順位による計算結果の違いを確認します。

### コンパイルと実行

# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic examples/operators\_demo.c -o operators\_demo  
  
# C99準拠でコンパイル  
gcc -std=c99 -Wall -Wextra -pedantic examples/operators\_demo\_c99.c -o operators\_demo\_c99  
  
# 実行  
./operators\_demo

## 演習課題

演習課題の詳細は <exercises/README.md> を参照してください。

### 基礎問題

1. **四則演算計算機**
   * 2つの数値に対してすべての算術演算を実行するプログラムを作成してください
2. **比較と論理演算**
   * 2つの数値の中から最大値を求めるプログラムを作成してください
3. **インクリメント・デクリメント**
   * 前置と後置の違いを確認するプログラムを作成してください

### 応用問題

1. **ビット操作**
   * 整数の各ビットを表示するプログラムを作成してください
2. **条件演算子の活用**
   * 三項演算子を使って複雑な条件分岐を実装してください
3. **演算子優先順位**
   * 複雑な式の計算順序を確認するプログラムを作成してください

## 解答例

各演習問題の解答例は <solutions/> ディレクトリにあります：

* [increment\_decrement.c](solutions/increment_decrement.c) / [C99版](solutions/increment_decrement_c99.c)
* [operator\_precedence.c](solutions/operator_precedence.c) / [C99版](solutions/operator_precedence_c99.c)
* [conditional\_operator.c](solutions/conditional_operator.c) / [C99版](solutions/conditional_operator_c99.c)

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make operators\_demo  
  
# C99版をコンパイル  
make operators\_demo\_c99  
  
# 全て実行  
make run-all  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* ビット演算子は整数型に対してのみ使用可能
* 論理演算子の結果は0または1
* ブロック先頭でのみ変数宣言可能

### C99以降の拡張

* \_Bool型の追加により、論理値の取り扱いが明確化
* 複素数型に対する演算子の追加
* forループ内での変数宣言が可能
* インライン関数の使用が可能

## よくある間違い

### 1. 代入と比較の混同

/\* NG: 代入を条件文で使用 \*/  
if (a = 5) /\* 常に真になる \*/  
 /\* ... \*/  
  
/\* OK: 比較演算子を使用 \*/  
if (a == 5) /\* a が 5 と等しいかチェック \*/  
 /\* ... \*/

### 2. 整数除算の結果

/\* NG: 期待した結果にならない \*/  
double result = 5 / 2; /\* 結果: 2.0 \*/  
  
/\* OK: 実数除算を使用 \*/  
double result = 5.0 / 2.0; /\* 結果: 2.5 \*/

### 3. 演算子の優先順位

/\* NG: 意図しない計算順序 \*/  
int result = a + b \* c + d; /\* b \* c が先に計算される \*/  
  
/\* OK: 括弧で明示 \*/  
int result = (a + b) \* (c + d);

### 4. ビット演算の優先順位

/\* NG: 期待しない結果 \*/  
if (flags & MASK == 1) /\* flags & (MASK == 1) として評価される \*/  
  
/\* OK: 括弧で明示 \*/  
if ((flags & MASK) == 1) /\* 正しい評価順序 \*/

## 次の章へ

演算子と式を理解したら、[制御構造（条件分岐）](../control-if/README.md) に進んでください。

## 参考資料

* [C言語演算子リファレンス](https://ja.cppreference.com/w/c/language/operator_precedence)
* [ビット演算詳細](https://ja.cppreference.com/w/c/language/operator_arithmetic)
* [演算子優先順位表](https://ja.cppreference.com/w/c/language/operator_precedence) ## サンプルコード

### bitwise\_demo.c

/\*  
 \* bitwise\_demo.c  
 \* ビット演算の動作を可視化して確認  
 \* C90準拠  
 \*/  
  
#include <stdio.h>  
  
/\* ビットパターンを表示する関数 \*/  
void print\_bits(unsigned char value) {  
 int i;  
 for (i = 7; i >= 0; i--) {  
 printf("%d", (value >> i) & 1);  
 if (i == 4) printf(" "); /\* 読みやすさのため4ビットごとに空白 \*/  
 }  
}  
  
int main(void) {  
 unsigned char a = 5; /\* 00000101 \*/  
 unsigned char b = 3; /\* 00000011 \*/  
 unsigned char result;  
   
 printf("===== ビット演算のデモ =====\n");  
 printf("初期値:\n");  
 printf(" a = %3d : ", a);  
 print\_bits(a);  
 printf("\n");  
 printf(" b = %3d : ", b);  
 print\_bits(b);  
 printf("\n\n");  
   
 /\* ビットAND \*/  
 result = a & b;  
 printf("ビットAND (a & b):\n");  
 printf(" ");  
 print\_bits(a);  
 printf("\n& ");  
 print\_bits(b);  
 printf("\n---------------\n ");  
 print\_bits(result);  
 printf(" = %d\n\n", result);  
   
 /\* ビットOR \*/  
 result = a | b;  
 printf("ビットOR (a | b):\n");  
 printf(" ");  
 print\_bits(a);  
 printf("\n| ");  
 print\_bits(b);  
 printf("\n---------------\n ");  
 print\_bits(result);  
 printf(" = %d\n\n", result);  
   
 /\* ビットXOR \*/  
 result = a ^ b;  
 printf("ビットXOR (a ^ b):\n");  
 printf(" ");  
 print\_bits(a);  
 printf("\n^ ");  
 print\_bits(b);  
 printf("\n---------------\n ");  
 print\_bits(result);  
 printf(" = %d\n\n", result);  
   
 /\* ビット反転 \*/  
 result = ~a;  
 printf("ビット反転 (~a):\n");  
 printf("~ ");  
 print\_bits(a);  
 printf("\n---------------\n ");  
 print\_bits(result);  
 printf(" = %d\n\n", result);  
   
 /\* 左シフト \*/  
 printf("===== シフト演算のデモ =====\n");  
 a = 5; /\* リセット \*/  
   
 result = a << 1;  
 printf("左シフト (a << 1):\n");  
 printf(" ");  
 print\_bits(a);  
 printf(" << 1\n= ");  
 print\_bits(result);  
 printf(" = %d (2倍)\n\n", result);  
   
 result = a << 2;  
 printf("左シフト (a << 2):\n");  
 printf(" ");  
 print\_bits(a);  
 printf(" << 2\n= ");  
 print\_bits(result);  
 printf(" = %d (4倍)\n\n", result);  
   
 /\* 右シフト \*/  
 a = 20; /\* 00010100 \*/  
   
 result = a >> 1;  
 printf("右シフト (a >> 1):\n");  
 printf(" ");  
 print\_bits(a);  
 printf(" >> 1\n= ");  
 print\_bits(result);  
 printf(" = %d (1/2)\n\n", result);  
   
 result = a >> 2;  
 printf("右シフト (a >> 2):\n");  
 printf(" ");  
 print\_bits(a);  
 printf(" >> 2\n= ");  
 print\_bits(result);  
 printf(" = %d (1/4)\n\n", result);  
   
 /\* 実用例：フラグ管理 \*/  
 printf("===== ビット演算の実用例（フラグ管理） =====\n");  
 #define FLAG\_A 0x01 /\* 0000 0001 \*/  
 #define FLAG\_B 0x02 /\* 0000 0010 \*/  
 #define FLAG\_C 0x04 /\* 0000 0100 \*/  
 #define FLAG\_D 0x08 /\* 0000 1000 \*/  
   
 unsigned char flags = 0;  
   
 printf("初期状態: ");  
 print\_bits(flags);  
 printf("\n");  
   
 /\* フラグを立てる \*/  
 flags |= FLAG\_A;  
 printf("FLAG\_A を立てる: ");  
 print\_bits(flags);  
 printf("\n");  
   
 flags |= FLAG\_C;  
 printf("FLAG\_C を立てる: ");  
 print\_bits(flags);  
 printf("\n");  
   
 /\* フラグのチェック \*/  
 if (flags & FLAG\_A) {  
 printf("FLAG\_A は ON です\n");  
 }  
 if (flags & FLAG\_B) {  
 printf("FLAG\_B は ON です\n");  
 } else {  
 printf("FLAG\_B は OFF です\n");  
 }  
   
 /\* フラグをクリア \*/  
 flags &= ~FLAG\_A;  
 printf("FLAG\_A をクリア: ");  
 print\_bits(flags);  
 printf("\n");  
   
 /\* フラグをトグル \*/  
 flags ^= FLAG\_B;  
 printf("FLAG\_B をトグル: ");  
 print\_bits(flags);  
 printf("\n");  
   
 return 0;  
}

C99版は[こちら](bitwise_demo_c99.c)を参照してください。

### operators\_demo.c

/\*  
 \* operators\_demo.c  
 \* さまざまな演算子の使用方法を学習  
 \* C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void) {  
 int a = 10, b = 3;  
 double x = 10.0, y = 3.0;  
 int result;  
   
 printf("===== 算術演算子のデモ =====\n");  
 printf("整数演算 (a = %d, b = %d):\n", a, b);  
 printf(" a + b = %d\n", a + b);  
 printf(" a - b = %d\n", a - b);  
 printf(" a \* b = %d\n", a \* b);  
 printf(" a / b = %d (整数除算)\n", a / b);  
 printf(" a %% b = %d (剰余)\n", a % b);  
   
 printf("\n実数演算 (x = %.1f, y = %.1f):\n", x, y);  
 printf(" x / y = %.2f (実数除算)\n", x / y);  
   
 printf("\n===== 代入演算子のデモ =====\n");  
 result = a;  
 printf("result = %d\n", result);  
 result += 5;  
 printf("result += 5 -> %d\n", result);  
 result -= 3;  
 printf("result -= 3 -> %d\n", result);  
 result \*= 2;  
 printf("result \*= 2 -> %d\n", result);  
 result /= 4;  
 printf("result /= 4 -> %d\n", result);  
 result %= 3;  
 printf("result %%= 3 -> %d\n", result);  
   
 printf("\n===== インクリメント・デクリメント演算子 =====\n");  
 a = 5;  
 printf("初期値: a = %d\n", a);  
 printf("a++ = %d (後置インクリメント)\n", a++);  
 printf("現在の a = %d\n", a);  
 printf("++a = %d (前置インクリメント)\n", ++a);  
 printf("現在の a = %d\n", a);  
   
 a = 5;  
 printf("\n初期値: a = %d\n", a);  
 printf("a-- = %d (後置デクリメント)\n", a--);  
 printf("現在の a = %d\n", a);  
 printf("--a = %d (前置デクリメント)\n", --a);  
 printf("現在の a = %d\n", a);  
   
 printf("\n===== 関係演算子のデモ =====\n");  
 a = 5; b = 3;  
 printf("a = %d, b = %d のとき:\n", a, b);  
 printf(" a == b : %d\n", a == b);  
 printf(" a != b : %d\n", a != b);  
 printf(" a < b : %d\n", a < b);  
 printf(" a <= b : %d\n", a <= b);  
 printf(" a > b : %d\n", a > b);  
 printf(" a >= b : %d\n", a >= b);  
   
 printf("\n===== 論理演算子のデモ =====\n");  
 a = 1; b = 0;  
 printf("a = %d (真), b = %d (偽) のとき:\n", a, b);  
 printf(" a && b : %d\n", a && b);  
 printf(" a || b : %d\n", a || b);  
 printf(" !a : %d\n", !a);  
 printf(" !b : %d\n", !b);  
   
 /\* 短絡評価のデモ \*/  
 printf("\n短絡評価のデモ:\n");  
 a = 0; b = 5;  
 printf("a = %d, b = %d のとき:\n", a, b);  
 if (a != 0 && b / a > 2) {  
 printf(" 条件は真です\n");  
 } else {  
 printf(" 条件は偽です (a が 0 なので b/a は評価されない)\n");  
 }  
   
 printf("\n===== 条件演算子（三項演算子）のデモ =====\n");  
 a = 10; b = 20;  
 result = (a > b) ? a : b;  
 printf("a = %d, b = %d のとき:\n", a, b);  
 printf(" (a > b) ? a : b = %d (大きい方の値)\n", result);  
   
 printf("\n===== sizeof演算子のデモ =====\n");  
 printf("基本データ型のサイズ:\n");  
 printf(" sizeof(char) = %lu バイト\n", (unsigned long)sizeof(char));  
 printf(" sizeof(short) = %lu バイト\n", (unsigned long)sizeof(short));  
 printf(" sizeof(int) = %lu バイト\n", (unsigned long)sizeof(int));  
 printf(" sizeof(long) = %lu バイト\n", (unsigned long)sizeof(long));  
 printf(" sizeof(float) = %lu バイト\n", (unsigned long)sizeof(float));  
 printf(" sizeof(double) = %lu バイト\n", (unsigned long)sizeof(double));  
   
 return 0;  
}

C99版は[こちら](operators_demo_c99.c)を参照してください。

### precedence\_demo.c

/\*  
 \* precedence\_demo.c  
 \* 演算子の優先順位による計算結果の違いを確認  
 \* C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void) {  
 int a = 2, b = 3, c = 4, d = 5;  
 int result;  
   
 printf("===== 演算子優先順位のデモ =====\n");  
 printf("初期値: a = %d, b = %d, c = %d, d = %d\n\n", a, b, c, d);  
   
 /\* 算術演算子の優先順位 \*/  
 printf("【算術演算子の優先順位】\n");  
   
 result = a + b \* c;  
 printf("a + b \* c = %d + %d \* %d = %d\n", a, b, c, result);  
 printf(" → 乗算が先: %d + (%d \* %d) = %d + %d = %d\n\n", a, b, c, b \* c, b \* c, result);  
   
 result = (a + b) \* c;  
 printf("(a + b) \* c = (%d + %d) \* %d = %d\n", a, b, c, result);  
 printf(" → 括弧が先: (%d + %d) \* %d = %d \* %d = %d\n\n", a, b, c, a + b, a + b, result);  
   
 result = a \* b + c \* d;  
 printf("a \* b + c \* d = %d \* %d + %d \* %d = %d\n", a, b, c, d, result);  
 printf(" → 乗算が先: (%d \* %d) + (%d \* %d) = %d + %d = %d\n\n",   
 a, b, c, d, a \* b, c \* d, result);  
   
 /\* 複雑な式の例 \*/  
 printf("【複雑な式の評価順序】\n");  
   
 result = a + b \* c - d;  
 printf("a + b \* c - d = %d + %d \* %d - %d = %d\n", a, b, c, d, result);  
 printf(" 評価順序:\n");  
 printf(" 1) b \* c = %d \* %d = %d\n", b, c, b \* c);  
 printf(" 2) a + (b \* c) = %d + %d = %d\n", a, b \* c, a + b \* c);  
 printf(" 3) (a + b \* c) - d = %d - %d = %d\n\n", a + b \* c, d, result);  
   
 /\* 関係演算子と論理演算子 \*/  
 printf("【関係演算子と論理演算子の優先順位】\n");  
   
 result = a < b && c > d;  
 printf("a < b && c > d = %d < %d && %d > %d = %d\n", a, b, c, d, result);  
 printf(" 評価順序:\n");  
 printf(" 1) a < b = %d < %d = %d (真)\n", a, b, a < b);  
 printf(" 2) c > d = %d > %d = %d (偽)\n", c, d, c > d);  
 printf(" 3) (a < b) && (c > d) = %d && %d = %d\n\n", a < b, c > d, result);  
   
 /\* ビット演算子と比較演算子 \*/  
 printf("【ビット演算子と比較演算子の優先順位】\n");  
   
 /\* 注意：この例は優先順位を示すためのもの \*/  
 result = (a & b) == 2; /\* 括弧を使って意図を明確にする \*/  
 printf("(a & b) == 2 = (%d & %d) == 2 = %d\n", a, b, result);  
 printf(" 評価順序:\n");  
 printf(" 1) a & b = %d & %d = %d\n", a, b, a & b);  
 printf(" 2) (a & b) == 2 = %d == 2 = %d\n\n", a & b, result);  
   
 /\* 代入演算子の結合性 \*/  
 printf("【代入演算子の結合性（右結合）】\n");  
   
 a = b = c = 10;  
 printf("a = b = c = 10 の後: a = %d, b = %d, c = %d\n", a, b, c);  
 printf(" → 右から左に評価: a = (b = (c = 10))\n\n");  
   
 /\* インクリメントと他の演算子 \*/  
 printf("【インクリメント演算子の優先順位】\n");  
   
 a = 5;  
 result = ++a \* 2;  
 printf("a = 5; result = ++a \* 2;\n");  
 printf(" result = %d (前置++が先: 6 \* 2)\n", result);  
 printf(" a = %d\n\n", a);  
   
 a = 5;  
 result = a++ \* 2;  
 printf("a = 5; result = a++ \* 2;\n");  
 printf(" result = %d (後置++は後: 5 \* 2)\n", result);  
 printf(" a = %d\n\n", a);  
   
 /\* 括弧を使った明確な表現 \*/  
 printf("【推奨：括弧で意図を明確にする】\n");  
   
 a = 2; b = 3; c = 4; d = 1;  
   
 /\* 曖昧な式 \*/  
 result = a + b << 2 & c;  
 printf("曖昧: a + b << 2 & c = %d\n", result);  
   
 /\* 明確な式 \*/  
 result = ((a + b) << 2) & c;  
 printf("明確: ((a + b) << 2) & c = %d\n", result);  
 printf(" 評価順序:\n");  
 printf(" 1) a + b = %d + %d = %d\n", a, b, a + b);  
 printf(" 2) (a + b) << 2 = %d << 2 = %d\n", a + b, (a + b) << 2);  
 printf(" 3) ((a + b) << 2) & c = %d & %d = %d\n", (a + b) << 2, c, result);  
   
 return 0;  
}

C99版は[こちら](precedence_demo_c99.c)を参照してください。

# 第5章: 条件分岐

## 対応C規格

* **主要対象:** C90
* **学習内容:** if文、else文、switch文、条件演算子による分岐処理

## 学習目標

この章を完了すると、以下のことができるようになります：

* if文を使った条件分岐ができる
* else if文で複数条件を処理できる
* switch文による多分岐処理ができる
* 条件演算子を適切に使える
* ネストした条件分岐を理解する

## 概要と詳細

### 条件分岐の世界へようこそ

私たちは日常生活で常に「もし〜なら」という判断をしています。「もし雨が降ったら傘を持つ」「もし電車が遅れたらバスで行く」など、状況に応じて行動を変えています。

プログラミングでも同じように、条件によって処理を変える必要があります。この章では、C言語で条件分岐を実現する方法を学びます。

### なぜ条件分岐が必要なのか？

条件分岐がないプログラムは、いつも同じ処理しかできません。それでは： - ユーザーの入力に応じた処理ができない - エラーが発生しても対処できない - 状況に応じた最適な処理を選べない

条件分岐を使うことで、プログラムに「知能」を与えることができます。

### if文の基本（もし〜なら）

if文は最も基本的な条件分岐です。日本語の「もし〜なら〜する」をそのままプログラムで表現できます。

#### 基本構文

if (条件式) {  
 /\* 条件が真（正しい）の場合に実行される処理 \*/  
}

**重要なポイント**： - 条件式は括弧 () で囲む - 処理部分は波括弧 {} で囲む（1行でも推奨） - 条件が偽（正しくない）の場合は、if文の中は実行されない

#### 実生活に例えた単純なif文

#include <stdio.h>  
  
int main(void)  
{  
 int rain\_probability = 70; /\* 降水確率70% \*/  
   
 if (rain\_probability >= 50) {  
 printf("傘を持っていきましょう\n");  
 }  
   
 int temperature = 35; /\* 気温35度 \*/  
   
 if (temperature >= 30) {  
 printf("熱中症に注意！水分補給を忘れずに\n");  
 }  
   
 return 0;  
}

#### 波括弧の重要性

/\* 危険な書き方（波括弧なし） \*/  
if (score >= 80)  
 printf("合格です\n");  
 printf("おめでとう！\n"); /\* これは常に実行される！ \*/  
  
/\* 安全な書き方（波括弧あり） \*/  
if (score >= 80) {  
 printf("合格です\n");  
 printf("おめでとう！\n"); /\* 両方とも条件が真の時だけ実行 \*/  
}

### if-else文（そうでなければ）

現実の判断では「もし〜なら〜する、そうでなければ〜する」という二者択一の場面がよくあります。if-else文はまさにこの状況を表現します。

#### 基本構文

if (条件式) {  
 /\* 条件が真（正しい）の場合の処理 \*/  
} else {  
 /\* 条件が偽（正しくない）の場合の処理 \*/  
}

**ポイント**： - どちらか一方の処理が必ず実行される - elseは「それ以外すべて」を意味する

#### 実用的なif-else文の例

/\* 成人判定 \*/  
int age = 19;  
  
if (age >= 20) {  
 printf("成人です\n");  
 printf("選挙権があります\n");  
} else {  
 printf("未成年です\n");  
 printf("あと%d年で成人です\n", 20 - age);  
}  
  
/\* パスワード認証 \*/  
int password = 1234;  
int input = 1235;  
  
if (password == input) {  
 printf("ログイン成功！\n");  
 printf("ようこそ！\n");  
} else {  
 printf("パスワードが違います\n");  
 printf("もう一度お試しください\n");  
}  
  
/\* 営業判定 \*/  
int hour = 18;  
  
if (hour >= 9 && hour < 17) {  
 printf("営業中です\n");  
} else {  
 printf("営業時間外です\n");  
}

#### よくある使い方：エラー処理

int balance = 1000; /\* 残高 \*/  
int withdraw = 1500; /\* 引き出し額 \*/  
  
if (balance >= withdraw) {  
 balance -= withdraw;  
 printf("引き出し成功\n");  
 printf("残高: %d円\n", balance);  
} else {  
 printf("残高不足です\n");  
 printf("現在の残高: %d円\n", balance);  
}

### if-else if文（複数の選択肢）

3つ以上の選択肢から選ぶ場面では、if-else if文を使います。上から順番に条件をチェックし、最初に真になった処理を実行します。

#### 基本構文

if (条件1) {  
 /\* 条件1が真の場合 \*/  
} else if (条件2) {  
 /\* 条件1は偽で、条件2が真の場合 \*/  
} else if (条件3) {  
 /\* 条件1,2は偽で、条件3が真の場合 \*/  
} else {  
 /\* すべての条件が偽の場合 \*/  
}

**重要な特徴**： - 上から順番にチェックされる - 最初に真になった部分だけ実行される - それ以降の条件はチェックされない

#### 成績判定システムの例

int score = 85;  
  
printf("点数: %d点\n", score);  
  
if (score >= 90) {  
 printf("成績: A（優秀）\n");  
 printf("素晴らしい成績です！\n");  
} else if (score >= 80) {  
 printf("成績: B（良好）\n");  
 printf("よく頑張りました！\n");  
} else if (score >= 70) {  
 printf("成績: C（普通）\n");  
 printf("合格です\n");  
} else if (score >= 60) {  
 printf("成績: D（可）\n");  
 printf("ギリギリ合格です\n");  
} else {  
 printf("成績: F（不可）\n");  
 printf("もう少し頑張りましょう\n");  
}

#### 時間帯による挨拶の例

int hour = 14; /\* 14時（午後2時） \*/  
  
if (hour >= 5 && hour < 12) {  
 printf("おはようございます\n");  
} else if (hour >= 12 && hour < 17) {  
 printf("こんにちは\n");  
} else if (hour >= 17 && hour < 21) {  
 printf("こんばんは\n");  
} else {  
 printf("夜遅いですね、お疲れさまです\n");  
}

#### 順序が重要な理由

/\* 間違った順序 \*/  
int age = 15;  
  
if (age >= 0) {  
 printf("生まれています\n"); /\* 15歳でもここで止まる！ \*/  
} else if (age >= 13) {  
 printf("中学生以上\n"); /\* 実行されない \*/  
} else if (age >= 18) {  
 printf("成人\n"); /\* 実行されない \*/  
}  
  
/\* 正しい順序（厳しい条件から） \*/  
if (age >= 18) {  
 printf("成人\n");  
} else if (age >= 13) {  
 printf("中学生以上\n");  
} else if (age >= 0) {  
 printf("生まれています\n");  
}

### ネストしたif文（入れ子構造）

if文の中にさらにif文を書くことで、より複雑な条件判断ができます。

/\* 天気と気温による服装アドバイス \*/  
int temperature = 25;  
int is\_raining = 0; /\* 0:晴れ、1:雨 \*/  
  
if (temperature >= 25) {  
 printf("暖かい日です\n");  
 if (is\_raining) {  
 printf("半袖で傘を持っていきましょう\n");  
 } else {  
 printf("半袖で大丈夫です\n");  
 }  
} else if (temperature >= 15) {  
 printf("過ごしやすい気温です\n");  
 if (is\_raining) {  
 printf("長袖で傘を持っていきましょう\n");  
 } else {  
 printf("長袖がおすすめです\n");  
 }  
} else {  
 printf("寒い日です\n");  
 if (is\_raining) {  
 printf("コートと傘が必要です\n");  
 } else {  
 printf("コートを着ていきましょう\n");  
 }  
}

#### ネストを避ける工夫

ネストが深くなると読みにくくなるので、論理演算子を使って簡潔に書くこともできます：

/\* ネストを使った場合 \*/  
if (age >= 18) {  
 if (has\_license) {  
 printf("運転できます\n");  
 }  
}  
  
/\* 論理演算子を使った場合 \*/  
if (age >= 18 && has\_license) {  
 printf("運転できます\n");  
}

### 条件式の詳細

#### 比較演算子

int a = , b = ;  
  
if (a == b) printf("等しいn");   
if (a != b) printf("等しくないn");   
if (a < b) printf("a は b より小さいn");   
if (a <= b) printf("a は b 以下n");   
if (a > b) printf("a は b より大きいn");   
if (a >= b) printf("a は b 以上n");

#### 論理演算子の組み合わせ

int age = ;  
int income = ;  
  
/\* AND演算子 \*/  
if (age >= && income >= )   
 printf("ローン審査に通りましたn");  
  
  
/\* OR演算子 \*/  
if (age < || age > )   
 printf("特別料金が適用されますn");  
  
  
/\* NOT演算子 \*/  
if (!(age >= ))   
 printf("歳未満ですn");

### switch文

複数の値に対する分岐処理を効率的に記述できます。

#### switch文って何？

プログラムを書いていると、「この変数の値が1なら処理A、2なら処理B、3なら処理C…」というように、一つの変数の値によって多くの分岐をしたい場面があります。

これをif-else ifで書くと：

if (choice == 1) {  
 printf("処理A\n");  
} else if (choice == 2) {  
 printf("処理B\n");  
} else if (choice == 3) {  
 printf("処理C\n");  
} else if (choice == 4) {  
 printf("処理D\n");  
} else {  
 printf("その他の処理\n");  
}

このように長くなってしまいます。switch文を使うと、もっとスッキリ書けます！

#### 日常生活での例え

switch文は「自動販売機」のようなものです：

1. ボタンを押す（変数の値）
2. 押したボタンに応じて違う商品が出る（caseごとの処理）
3. 存在しないボタンを押したら何も出ない（default処理）

int button = 2; /\* 2番のボタンを押した \*/  
  
switch (button) {  
 case 1:  
 printf("コーラが出ました\n");  
 break;  
 case 2:  
 printf("オレンジジュースが出ました\n");  
 break;  
 case 3:  
 printf("お茶が出ました\n");  
 break;  
 default:  
 printf("そのボタンには商品がありません\n");  
 break;  
}

#### 基本構文

switch (変数または式)   
 case 値:  
 /\* 値の場合の処理 \*/  
 break;  
 case 値:  
 /\* 値の場合の処理 \*/  
 break;  
 case 値:  
 /\* 値の場合の処理 \*/  
 break;  
 default:  
 /\* どの値にも一致しない場合の処理 \*/  
 break;

#### 曜日判定の例

int day = ;  
  
switch (day)   
 case :  
 printf("月曜日n");  
 break;  
 case :  
 printf("火曜日n");  
 break;  
 case :  
 printf("水曜日n");  
 break;  
 case :  
 printf("木曜日n");  
 break;  
 case :  
 printf("金曜日n");  
 break;  
 case :  
 printf("土曜日n");  
 break;  
 case :  
 printf("日曜日n");  
 break;  
 default:  
 printf("無効な曜日ですn");  
 break;

#### break文の重要性

break文を忘れると、次のcaseも実行されます（フォールスルー）：

**初心者がよくやるミス：breakを忘れる**

switch文で最も多いミスが「break文の書き忘れ」です。これは本当によくあるミスなので、必ず確認しましょう！

int grade = '';  
  
switch (grade)   
 case 'A':  
 printf("優秀n");  
 /\* break がないので次のcaseも実行される \*/  
 case '':  
 printf("良好n");  
 /\* break がないので次のcaseも実行される \*/  
 case 'C':  
 printf("普通n");  
 break;  
 default:  
 printf("要努力n");  
 break;  
  
/\* grade が '' の場合、"良好" と "普通" の両方が出力される \*/

#### 意図的なフォールスルー

時には意図的にbreakを省略することもあります：

**フォールスルーが便利な場面**

「複数の値で同じ処理をしたい」ときは、あえてbreakを書かないことで、コードを簡潔に書けます：

char ch = 'a';  
  
switch (ch)   
 case 'a':  
 case 'e':  
 case 'i':  
 case 'o':  
 case '':  
 printf("母音ですn");  
 break;  
 default:  
 printf("子音ですn");  
 break;

### 条件演算子（三項演算子）の復習

条件分岐の簡潔な書き方として条件演算子があります。

int a = , b = ;  
int max;  
  
/\* if-else文での記述 \*/  
if (a > b)   
 max = a;  
 else   
 max = b;  
  
  
/\* 条件演算子での記述 \*/  
max = (a > b) ? a : b;  
  
printf("最大値: %d\n", max);

### 条件式での注意点

初心者が陥りやすい間違いをいくつか紹介します。これらを知っておけば、デバッグ時間を大幅に短縮できます！

#### 代入と比較の混同

int x = ;  
  
/\* NG: 代入になってしまう \*/  
if (x = 10) { /\* ← これは間違い！ \*/  
 printf("常に実行される\n"); /\* x に 10 が代入され、常に真 \*/  
}  
  
/\* OK: 比較演算子を使用 \*/  
if (x == 10) { /\* ← 正しい書き方 \*/  
 printf("x が 10 の場合のみ実行\n");  
}  
  
/\* 防御的プログラミング：定数を左側に書く \*/  
if (10 == x) { /\* もし = を1つしか書かなかったらコンパイルエラーになる \*/  
 printf("x が 10 の場合のみ実行\n");  
}

#### 浮動小数点数の比較

コンピュータでは小数を完全に正確に表現できないことがあります。これは初心者にとって意外な落とし穴です：

double d = 0.1 + 0.2;  
  
/\* NG: 浮動小数点の誤差で期待通りにならない可能性 \*/  
if (d == 0.3) {  
 printf("等しい\n"); /\* 実行されないかも！ \*/  
}  
  
/\* OK: 誤差を考慮した比較 \*/  
if (d >= 0.299999 && d <= 0.300001) {  
 printf("ほぼ等しい\n");  
}  
  
/\* より実践的な方法：許容誤差を定義 \*/  
#define EPSILON 0.00001  
if (fabs(d - 0.3) < EPSILON) {  
 printf("実質的に等しい\n");  
}

**なぜこんなことが起きるの？**

コンピュータは2進数で計算するため、10進数の0.1や0.2を正確に表現できません。人間にとっての「0.1」は、コンピュータにとっては「0.1に限りなく近い値」なのです。

## サンプルコード

### if文の基本使用例

プログラムファイル: examples/if\_basic.c

基本的なif文、if-else文の使用方法を学習します。

### 複雑な条件分岐

プログラムファイル: examples/complex\_conditions.c

複数条件を組み合わせた分岐処理を学習します。

### switch文の実例

プログラムファイル: examples/switch\_demo.c

switch文を使った多分岐処理を学習します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic if\_basic.c -o if\_basic  
  
# 実行  
./if\_basic

## 演習課題

### 基礎問題

. **年齢判定プログラム** - 年齢を入力して、幼児・児童・学生・成人・高齢者に分類するプログラムを作成してください

. **電卓プログラム** - 2つの数値と演算子（+, -, \*, /）を入力して計算結果を表示するプログラムをswitch文で作成してください

. **成績判定システム** - 点数を入力してA〜の成績を判定するプログラムを作成してください

### 応用問題

. **うるう年判定** - 年を入力してうるう年かどうかを判定するプログラムを作成してください - 条件: で割り切れる、ただしで割り切れる年は平年、で割り切れる年はうるう年

. **MI計算と判定** - 身長と体重を入力してMIを計算し、肥満度を判定するプログラムを作成してください

. **三角形の判定** - 2つの辺の長さを入力して、三角形として成立するか、どの種類の三角形かを判定してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make if\_basic  
  
# 実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* switch文の case ラベルは整数定数のみ
* 複合文内での変数宣言は先頭のみ

### C99以降の拡張

* switch文でlong long型が使用可能
* ブロック内での変数宣言位置の制限緩和

## よくある間違い

### . セミコロンの位置

/\* NG: if文の後にセミコロン \*/  
if (condition);  
  
 printf("常に実行されるn"); /\* 条件に関係なく実行 \*/  
  
  
/\* OK: 正しい記述 \*/  
if (condition)   
 printf("条件が真の場合のみ実行n");

### . switch文でのbreak忘れ

/\* NG: break忘れでフォールスルー \*/  
switch (vale)   
 case :  
 printf("ですn");  
 /\* break; を忘れると次のcaseも実行される \*/  
 case :  
 printf("ですn");  
 break;  
  
  
/\* OK: 適切なbreak文 \*/  
switch (vale)   
 case :  
 printf("ですn");  
 break;  
 case :  
 printf("ですn");  
 break;

### . 複雑な条件式での括弧不足

/\* NG: 意図しない優先順位 \*/  
if (a == || b == && c == )   
 /\* && が || より優先される \*/  
  
  
/\* OK: 括弧で明示 \*/  
if ((a == ) || (b == && c == ))   
 /\* 意図が明確 \*/

## 次の章へ

条件分岐を理解したら、[制御構造（ループ）](../control-loop/README.md) に進んでください。

## 参考資料

* [if文詳細](https://ja.cppreference.com/w/c/language/if)
* [switch文詳細](https://ja.cppreference.com/w/c/language/switch)
* [条件演算子](https://ja.cppreference.com/w/c/language/operautor_other)

## サンプルコード

### conditional\_operator.c

/\*  
 \* 条件演算子（三項演算子）の使い方（C90準拠）  
 \*   
 \* このプログラムは、条件演算子を使った  
 \* 簡潔な条件分岐の書き方を示します。  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 /\* 変数の宣言（C90では先頭で宣言） \*/  
 int a, b;  
 int max, min;  
 int age;  
 int score;  
 int bonus;  
 int number;  
 int abs\_value;  
 char \*result;  
 char \*status;  
   
 /\* 2つの数の最大値・最小値を求める \*/  
 printf("2つの整数を入力してください: ");  
 scanf("%d %d", &a, &b);  
   
 /\* 条件演算子を使った最大値・最小値の判定 \*/  
 max = (a > b) ? a : b;  
 min = (a < b) ? a : b;  
   
 printf("最大値: %d\n", max);  
 printf("最小値: %d\n", min);  
   
 /\* 年齢による判定 \*/  
 printf("\n年齢を入力してください: ");  
 scanf("%d", &age);  
   
 status = (age >= 20) ? "成人" : "未成年";  
 printf("あなたは%sです。\n", status);  
   
 /\* 成績による合否判定 \*/  
 printf("\n点数を入力してください (0-100): ");  
 scanf("%d", &score);  
   
 result = (score >= 60) ? "合格" : "不合格";  
 printf("判定: %s\n", result);  
   
 /\* ネストした条件演算子（読みやすさには注意） \*/  
 printf("成績評価: ");  
 printf("%c\n", (score >= 90) ? 'A' :  
 (score >= 80) ? 'B' :  
 (score >= 70) ? 'C' :  
 (score >= 60) ? 'D' : 'F');  
   
 /\* 条件演算子と通常の計算を組み合わせる \*/  
 bonus = (score >= 80) ? 10 : 0;  
 printf("ボーナス点: %d\n", bonus);  
 printf("最終得点: %d\n", score + bonus);  
   
 /\* 絶対値の計算 \*/  
 printf("\n整数を入力してください: ");  
 scanf("%d", &number);  
   
 abs\_value = (number >= 0) ? number : -number;  
 printf("%d の絶対値は %d です。\n", number, abs\_value);  
   
 return 0;  
}```  
  
> C99版は[こちら](conditional\_operator\_c99.c)を参照してください。  
  
### if\_basic.c  
  
```c  
/\*  
 \* if文の基本的な使い方（C90準拠）  
 \*   
 \* このプログラムは、if文、if-else文、if-else if文の  
 \* 基本的な使い方を示します。  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 /\* 変数の宣言（C90では先頭で宣言） \*/  
 int score;  
   
 printf("点数を入力してください (0-100): ");  
 scanf("%d", &score);  
   
 /\* 単純なif文 \*/  
 if (score == 100) {  
 printf("満点です！素晴らしい！\n");  
 }  
   
 /\* if-else文 \*/  
 if (score >= 60) {  
 printf("合格です\n");  
 } else {  
 printf("不合格です\n");  
 }  
   
 /\* if-else if文による成績判定 \*/  
 printf("成績評価: ");  
 if (score >= 90) {  
 printf("A (優秀)\n");  
 } else if (score >= 80) {  
 printf("B (良好)\n");  
 } else if (score >= 70) {  
 printf("C (普通)\n");  
 } else if (score >= 60) {  
 printf("D (可)\n");  
 } else {  
 printf("F (不可)\n");  
 }  
   
 /\* ネストしたif文 \*/  
 if (score >= 60) {  
 printf("おめでとうございます！");  
 if (score >= 80) {  
 printf("優秀な成績です！");  
 }  
 printf("\n");  
 }  
   
 return 0;  
}```  
  
> C99版は[こちら](if\_basic\_c99.c)を参照してください。  
  
### switch\_basic.c  
  
```c  
/\*  
 \* switch文の基本的な使い方（C90準拠）  
 \*   
 \* このプログラムは、switch文による多分岐処理の  
 \* 基本的な使い方を示します。  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 /\* 変数の宣言（C90では先頭で宣言） \*/  
 int menu\_choice;  
 char grade;  
 int month;  
   
 /\* メニュー選択の例 \*/  
 printf("=== レストランメニュー ===\n");  
 printf("1. ハンバーガー (500円)\n");  
 printf("2. ピザ (800円)\n");  
 printf("3. パスタ (700円)\n");  
 printf("4. サラダ (400円)\n");  
 printf("選択してください (1-4): ");  
 scanf("%d", &menu\_choice);  
   
 switch (menu\_choice) {  
 case 1:  
 printf("ハンバーガーを注文しました。500円です。\n");  
 break;  
 case 2:  
 printf("ピザを注文しました。800円です。\n");  
 break;  
 case 3:  
 printf("パスタを注文しました。700円です。\n");  
 break;  
 case 4:  
 printf("サラダを注文しました。400円です。\n");  
 break;  
 default:  
 printf("無効な選択です。\n");  
 break;  
 }  
   
 /\* 成績評価の例 \*/  
 printf("\n成績を入力してください (A-F): ");  
 scanf(" %c", &grade); /\* 空白文字を読み飛ばすため、%cの前に空白を入れる \*/  
   
 switch (grade) {  
 case 'A':  
 case 'a':  
 printf("優秀: 90点以上\n");  
 break;  
 case 'B':  
 case 'b':  
 printf("良好: 80-89点\n");  
 break;  
 case 'C':  
 case 'c':  
 printf("普通: 70-79点\n");  
 break;  
 case 'D':  
 case 'd':  
 printf("可: 60-69点\n");  
 break;  
 case 'F':  
 case 'f':  
 printf("不可: 60点未満\n");  
 break;  
 default:  
 printf("無効な成績です。\n");  
 break;  
 }  
   
 /\* 月から季節を判定する例（フォールスルーの活用） \*/  
 printf("\n月を入力してください (1-12): ");  
 scanf("%d", &month);  
   
 printf("季節: ");  
 switch (month) {  
 case 12:  
 case 1:  
 case 2:  
 printf("冬\n");  
 break;  
 case 3:  
 case 4:  
 case 5:  
 printf("春\n");  
 break;  
 case 6:  
 case 7:  
 case 8:  
 printf("夏\n");  
 break;  
 case 9:  
 case 10:  
 case 11:  
 printf("秋\n");  
 break;  
 default:  
 printf("無効な月です\n");  
 break;  
 }  
   
 return 0;  
}```  
  
> C99版は[こちら](switch\_basic\_c99.c)を参照してください。  
  
---  
  
# 第6章: 繰り返し処理  
  
  
## 対応C規格  
- \*\*主要対象:\*\* C90  
- \*\*学習内容:\*\* for文、while文、do-while文、break文、continue文、ネストしたループ  
  
## 学習目標  
  
この章を完了すると、以下のことができるようになります：  
  
- for文を使った繰り返し処理ができる  
- while文とdo-while文の違いを理解する  
- break文とcontinue文を適切に使える  
- ネストしたループを理解して活用できる  
- ループを使った実践的なプログラムを作成できる  
  
## 概要と詳細  
  
### 繰り返し処理とは？  
  
プログラミングでは、同じ処理を何度も繰り返したい場面がたくさんあります。例えば：  
  
- 1から100までの数を全て表示する  
- 成績データを100人分処理する   
- ゲームのメインループ（ユーザーが終了するまで続ける）  
- ファイルのデータを最後まで読み込む  
  
これらを一つ一つ手作業で書くのは大変です。そこで「ループ（繰り返し）」を使います！  
  
#### なぜループが重要なの？  
  
コンピュータの強みは「高速で正確な繰り返し処理」です。人間が苦手とする単調な繰り返し作業を、コンピュータは疲れることなく実行できます。この能力を活用するのがループです。  
  
### for文   
  
最も一般的なループ文で、初期化・条件・更新を一箇所で記述できます。  
  
#### for文の仕組みを理解しよう  
  
for文は「カウンター付きの繰り返し」と考えると分かりやすいです。例えば、ラジオ体操を10回やるとき：  
  
1. 最初は1回目から始める（初期化）  
2. 10回以下なら続ける（条件）  
3. 1回終わったら回数を増やす（更新）  
  
これをコードで表現すると：  
  
```c  
for (int i = 1; i <= 10; i++) {  
 printf("%d回目のラジオ体操\n", i);  
}

#### 基本構文

for (初期化; 条件式; 更新式)   
 /\* 繰り返し実行される文 \*/

#### 基本的な使用例

#include <stdio.h>  
  
int main(void)  
{  
  
 int i;  
   
 /\* 1から10まで出力 \*/  
 for (i = 1; i <= 10; i++)   
 printf("%d ", i);  
   
 printf("\n");  
   
 return 0;

#### for文の詳細動作

for文がどのように動作するか、ステップごとに見てみましょう：

for (i = 0; i < 5; i++) {  
 printf("i = %d\n", i);  
}  
  
/\* 上記は以下と同等 \*/  
i = 0; /\* ① 初期化（1回のみ実行） \*/  
while (i < 5) { /\* ② 条件チェック \*/  
 printf("i = %d\n", i); /\* ③ ループ本体 \*/  
 i++; /\* ④ 更新式 \*/  
}

**実行の流れ：** 1. 初期化 → i = 0 2. 条件チェック → 0 < 5は真、続行 3. ループ本体実行 → “i = 0”を表示 4. 更新 → i = 1 5. 条件チェック → 1 < 5は真、続行 6. …これを繰り返し… 7. 更新 → i = 5 8. 条件チェック → 5 < 5は偽、ループ終了

#### さまざまなfor文のパターン

for文は非常に柔軟で、色々な書き方ができます：

/\* 逆順ループ（カウントダウン） \*/  
for (i = 10; i >= 1; i--) {  
 printf("%d ", i);  
}  
printf("発射！\n");  
  
/\* 2つずつ増加（偶数だけ表示） \*/  
for (i = 0; i <= 20; i += 2) {  
 printf("%d ", i);  
}  
  
/\* 複数変数の制御（両端から中央へ） \*/  
for (i = 0, j = 10; i < j; i++, j--) {  
 printf("i=%d, j=%d\n", i, j);  
}  
  
/\* 無限ループ（条件式を省略） \*/  
for (;;) {  
 /\* Ctrl+Cで止めるまで続く \*/  
 printf("無限ループ中...\n");  
}

**初心者のためのヒント：** - 最初は基本形（for (i = 0; i < n; i++)）をマスターしましょう - 複雑なfor文は読みにくくなりがちなので、シンプルに保つことが大切です

### while文

条件が真である間、繰り返し処理を実行します。

#### while文って何？

while文は「〜である間、繰り返す」という意味です。条件が満たされている限り、処理を続けます。

**日常生活での例：** - お風呂にお湯を入れる → 「満タンになるまで」続ける - 宿題をする → 「全部終わるまで」続ける - 電車を待つ → 「電車が来るまで」待つ

これらは全て「ある条件が満たされるまで続ける」パターンです。

#### 基本構文

while (条件式)   
 /\* 繰り返し実行される文 \*/

#### while文の使用例

#include <stdio.h>  
  
int main(void)  
{  
  
 int count = ;  
   
 while (count <= )   
 printf("count = %d\n", count);  
 count++;  
   
   
 return ;

#### while文の実践例

while文は「回数が分からない繰り返し」に最適です：

/\* ユーザー入力の処理 \*/  
int number;  
printf("正の数を入力してください（0で終了）: ");  
  
while (scanf("%d", &number) == 1 && number > 0) {  
 printf("入力された数: %d\n", number);  
 printf("次の数を入力してください（0で終了）: ");  
}  
  
printf("プログラムを終了します。\n");

**for文とwhile文の使い分け：** - **for文**：繰り返し回数が分かっている場合（例：10回繰り返す） - **while文**：条件で終了を判断する場合（例：ユーザーが0を入力するまで）

### do-while文

最低1回は実行され、その後条件をチェックするループです。

#### do-while文の特徴

while文との大きな違いは「条件チェックのタイミング」です：

* **while文**：最初に条件をチェック（条件が偽なら一度も実行されない）
* **do-while文**：最後に条件をチェック（必ず一度は実行される）

**日常生活での例：**

レストランでの注文を考えてみましょう： 1. メニューを見せる（必ず一度は見せる） 2. 注文を聞く 3. 「他に注文はありますか？」と聞く 4. 「はい」なら1に戻る、「いいえ」なら終了

これがdo-while文の考え方です！

#### 基本構文

do   
 /\* 最低1回は実行される文 \*/  
 while (条件式);

#### do-while文の使用例

#include <stdio.h>  
  
int main(void)  
{  
  
 int choice;  
   
 do   
 printf("nメニュー:n");  
 printf(". オプションn");  
 printf(". オプションn");  
 printf(". 終了n");  
 printf("選択してください: ");  
 scanf("%d", &choice);  
   
 switch (choice)   
 case :  
 printf("オプションが選択されましたn");  
 break;  
 case :  
 printf("オプションが選択されましたn");  
 break;  
 case :  
 printf("プログラムを終了しますn");  
 break;  
 default:  
 printf("無効な選択ですn");  
 break;  
   
 while (choice != );  
   
 return ;

### break文とcontinue文 ⏭

ループの流れを制御する特別な文です。

#### 途中でループを抜けたいとき

プログラムを書いていると、「ある条件を満たしたらループを抜けたい」という場面があります。そんなときに使うのがbreak文とcontinue文です。

**日常生活での例：** - **break**：探し物が見つかったら探すのをやめる - **continue**：嫌いな野菜は飛ばして次の料理を食べる

#### break文

ループを強制的に終了します。

#include <stdio.h>  
  
int main(void)  
{  
  
 int i;  
   
 for (i = ; i <= ; i++)   
 if (i == )   
 break; /\* i が のときループを抜ける \*/  
   
 printf("%d ", i);  
   
 printf("nループを抜けましたn");  
   
 return ;  
  
/\* 出力: \*/

#### continue文

現在の繰り返しをスキップして、次の繰り返しに進みます。

#include <stdio.h>  
  
int main(void)  
{  
  
 int i;  
   
 for (i = ; i <= ; i++)   
 if (i % == )   
 continue; /\* 偶数の場合はスキップ \*/  
   
 printf("%d ", i);  
   
 printf("n");  
   
 return ;  
  
/\* 出力: 9 \*/

### ネストしたループ

ループの中にさらにループを含む構造です。

#### 二重ループを理解しよう

ネストしたループは「ループの中にループ」がある構造です。

**日常生活での例：**

学校の時間割を考えてみましょう： - 外側のループ：月曜日から金曜日まで（5回） - 内側のループ：1時間目から6時間目まで（6回） - 合計：5日 × 6時間 = 30コマ

これがネストしたループの考え方です！

#### 二重ループの例

#include <stdio.h>  
  
int main(void)  
{  
  
 int i, j;  
   
 /\* 九九表の作成 \*/  
 for (i = ; i <= 9; i++)   
 for (j = ; j <= 9; j++)   
 printf("%d ", i \* j);  
   
 printf("n");  
   
   
 return ;

#### 三角形パターンの出力

#include <stdio.h>  
  
int main(void)  
{  
  
 int i, j;  
   
 /\* 星印の三角形 \*/  
 for (i = ; i <= ; i++)   
 for (j = ; j <= i; j++)   
 printf("\* ");  
   
 printf("n");  
   
   
 return ;  
  
/\*  
出力:  
\*   
\* \*   
\* \* \*   
\* \* \* \*   
\* \* \* \* \*   
\*/

### ループでのbreak・continueの応用

#### ネストしたループでのbreak

#include <stdio.h>  
  
int main(void)  
{  
  
 int i, j;  
 int fond = ;  
   
 for (i = ; i <= && !fond; i++)   
 for (j = ; j <= ; j++)   
 if (i \* j == )   
 printf("発見: %d \* %d = n", i, j);  
 fond = ;  
 break; /\* 内側のループを抜ける \*/  
   
   
   
   
 return ;

#### ラベル付きbreak（goto文を使用）

#include <stdio.h>  
  
int main(void)  
{  
  
 int i, j;  
   
 for (i = ; i <= ; i++)   
 for (j = ; j <= ; j++)   
 if (i \* j == )   
 printf("発見: %d \* %d = n", i, j);  
 goto exit\_loops; /\* 両方のループを抜ける \*/  
   
   
   
   
exit\_loops:  
 printf("ループ終了n");  
 return ;

### 無限ループ ♾

意図的に終了しないループを作成することもあります。

#### 無限ループが必要な場面

「無限ループ」と聞くと怖く感じるかもしれませんが、実は多くのプログラムで使われています：

* ゲームのメインループ（ユーザーが終了するまで続く）
* サーバープログラム（常に要求を待ち受ける）
* 組み込みシステム（電源が入っている間ずっと動作）

**重要：** 無限ループを作るときは、必ず脱出方法（breakやプログラム終了条件）を用意しましょう！

#### 無限ループの作成方法

/\* 方法: for文 \*/  
for (;;)   
 /\* 無限ループ \*/  
 if (条件) break;  
  
  
/\* 方法: while文 \*/  
while ()   
 /\* 無限ループ \*/  
 if (条件) break;  
  
  
/\* 方法: do-while文 \*/  
do   
 /\* 無限ループ \*/  
 if (条件) break;  
 while ();

#### 無限ループの実用例

#include <stdio.h>  
  
int main(void)  
{  
  
 int choice;  
   
 while () /\* 無限ループ \*/  
 printf("n=== 計算機 ===n");  
 printf(". 足し算n");  
 printf(". 引き算n");  
 printf(". 終了n");  
 printf("選択: ");  
   
 if (scanf("%d", &choice) != )   
 printf("入力エラーn");  
 break;  
   
   
 if (choice == )   
 printf("終了しますn");  
 break;  
   
   
 /\* 計算処理... \*/  
   
   
 return ;

### ループの最適化とベストプラクティス

初心者がループを書く際に気をつけるべきポイントを紹介します。

#### 効率的なループ

/\* NG: 毎回strlen()を呼び出し \*/  
for (i = ; i < strlen(str); i++)   
 /\* 処理 \*/  
  
  
/\* OK: 長さを事前に計算 \*/  
len = strlen(str);  
for (i = ; i < len; i++)   
 /\* 処理 \*/

#### ループ変数の適切な使用

int main(void)  
{  
  
 int i; /\* C90では関数の先頭で宣言 \*/  
   
 for (i = ; i < ; i++)   
 /\* iはループ外でも有効 \*/  
   
   
 printf("最終的なi = %d\n", i); /\* が出力 \*/  
   
 return ;

## サンプルコード

### 基本的なループ例

プログラムファイル: examples/loop\_basic.c

for、while、do-while文の基本的な使用方法を学習します。

### ネストしたループ

プログラムファイル: examples/nested\_loops.c

二重・三重ループの使用例を学習します。

### break・continue文

プログラムファイル: examples/break\_continue.c

ループ制御文の使い方を学習します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic loop\_basic.c -o loop\_basic  
  
# 実行  
./loop\_basic

## 演習課題

### 基礎問題

. **数列の出力** - からまでの数を出力するプログラムを種類のループで作成してください

. **偶数・奇数の判定** - からまでの数で、偶数のみを出力するプログラムを作成してください

. **階乗の計算** - 入力された数の階乗を計算するプログラムを作成してください

### 応用問題

. **素数判定** - 入力された数が素数かどうかを判定するプログラムを作成してください

. **フィボナッチ数列** - n項目までのフィボナッチ数列を出力するプログラムを作成してください

. **パターン出力** - 以下のようなパターンを出力するプログラムを作成してください： \* \*\* \*\*\* \*\*\*\* \*\*\*\*\* \*\*\*\* \*\*\* \*\* \*

### 発展問題

. **成績管理システム** - 複数の学生の成績を入力し、平均・最高・最低点を計算するプログラム

. **簡易ゲーム** - 数当てゲームを作成してください（-の範囲で乱数を生成）

. **データ検索** - 配列の中から特定の値を検索するプログラムを作成してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make loop\_basic  
  
# 実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* ループ変数は関数の先頭で宣言する必要がある
* for文内での変数宣言は不可

### C99以降の拡張

* for文内での変数宣言が可能
* for (int i = ; i < ; i++) /\* C99以降 \*/  
   /\* 処理 \*/

## よくある間違い

初心者がループで陥りやすいミスと、その対策を紹介します。

### . 無限ループの作成

/\* NG: 終了条件の記述ミス \*/  
for (i = ; i > ; i++) /\* 最初から条件が偽 \*/  
 /\* 実行されない \*/  
  
  
/\* NG: 更新式の記述ミス \*/  
for (i = ; i < ; i--) /\* 負の方向に進む \*/  
 /\* 無限ループ \*/

### . セミコロンの位置

/\* NG: for文の後にセミコロン \*/  
for (i = ; i < ; i++); /\* 空文が実行される \*/  
  
 printf("%d\n", i); /\* ループ外で実行 \*/  
  
  
/\* OK: 正しい記述 \*/  
for (i = ; i < ; i++)   
 printf("%d\n", i);

### . ループ変数の範囲外アクセス

配列を使うときに特に注意が必要です：

/\* NG: 配列の範囲を超える \*/  
int arr[10];  
for (i = 0; i <= 10; i++) { /\* i=10で範囲外！ \*/  
 arr[i] = i; /\* 配列は0～9までしかない \*/  
}  
  
/\* OK: 正しい範囲 \*/  
for (i = 0; i < 10; i++) {  
 arr[i] = i;  
}

**覚え方：** 配列のサイズがNなら、インデックスは0からN-1まで！

## 次の章へ

ループ処理を理解したら、[配列](../arrays/README.md) に進んでください。

## 参考資料

* [for文詳細](https://ja.cppreference.com/w/c/language/for)
* [while文詳細](https://ja.cppreference.com/w/c/language/while)
* [break・continue文](https://ja.cppreference.com/w/c/language/break)

## サンプルコード

### break\_continue.c

/\*  
 \* break文とcontinue文の使い方  
 \*  
 \* このプログラムは、ループ制御文である  
 \* break文とcontinue文の使い方を示します。  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 int i, j;  
 int sum;  
 int num;  
 int is\_prime;  
 int count;  
 int value;  
 char ch;  
  
 /\* 1. break文の基本 \*/  
 printf("=== break文の基本 ===\n");  
 printf("1から10まで出力（5で中断）:\n");  
 for (i = 1; i <= 10; i++)  
 {  
 if (i == 5)  
 {  
 printf("\n5に到達したので終了します\n");  
 break;  
 }  
 printf("%d ", i);  
 }  
 printf("\n\n");  
  
 /\* 2. continue文の基本 \*/  
 printf("=== continue文の基本 ===\n");  
 printf("1から10まで出力（偶数をスキップ）:\n");  
 for (i = 1; i <= 10; i++)  
 {  
 if (i % 2 == 0)  
 {  
 continue; /\* 偶数の場合は次の繰り返しへ \*/  
 }  
 printf("%d ", i);  
 }  
 printf("\n\n");  
  
 /\* 3. 素数の判定（breakの活用） \*/  
 printf("=== 素数判定 ===\n");  
 printf("整数を入力してください: ");  
 scanf("%d", &num);  
  
 is\_prime = 1; /\* 素数フラグ（1:素数、0:素数でない） \*/  
  
 if (num <= 1)  
 {  
 is\_prime = 0;  
 }  
 else  
 {  
 for (i = 2; i \* i <= num; i++)  
 {  
 if (num % i == 0)  
 {  
 is\_prime = 0;  
 break; /\* 約数が見つかったら終了 \*/  
 }  
 }  
 }  
  
 if (is\_prime)  
 {  
 printf("%d は素数です\n", num);  
 }  
 else  
 {  
 printf("%d は素数ではありません\n", num);  
 }  
 printf("\n");  
  
 /\* 4. 負の数を除外した合計（continueの活用） \*/  
 printf("=== 正の数の合計計算 ===\n");  
 printf("5つの数を入力してください（負の数は無視されます）:\n");  
  
 sum = 0;  
 count = 0;  
 for (i = 0; i < 5; i++)  
 {  
 printf("%d番目の数: ", i + 1);  
 scanf("%d", &value);  
  
 if (value < 0)  
 {  
 printf(" → 負の数なのでスキップします\n");  
 continue;  
 }  
  
 sum += value;  
 count++;  
 }  
  
 printf("正の数の合計: %d（%d個）\n\n", sum, count);  
  
 /\* 5. ネストしたループでのbreak \*/  
 printf("=== ネストしたループでのbreak ===\n");  
 printf("九九の表（5x5で中断）:\n");  
  
 for (i = 1; i <= 9; i++)  
 {  
 for (j = 1; j <= 9; j++)  
 {  
 if (i == 5 && j == 5)  
 {  
 printf("\n5x5で中断します\n");  
 break; /\* 内側のループのみ終了 \*/  
 }  
 printf("%2d ", i \* j);  
 }  
 if (i == 5 && j == 5)  
 {  
 break; /\* 外側のループも終了 \*/  
 }  
 printf("\n");  
 }  
 printf("\n");  
  
 /\* 6. 無限ループとbreak \*/  
 printf("=== 無限ループの制御 ===\n");  
 printf("'q'を入力すると終了します:\n");  
 while (1)  
 { /\* 無限ループ \*/  
 printf("文字を入力: ");  
 scanf(" %c", &ch);  
  
 if (ch == 'q' || ch == 'Q')  
 {  
 printf("終了します\n");  
 break;  
 }  
  
 printf("入力された文字: %c\n", ch);  
 }  
 printf("\n");  
  
 /\* 7. 3の倍数と5の倍数をスキップ \*/  
 printf("=== 3の倍数と5の倍数以外を表示 ===\n");  
 printf("1から30まで（3の倍数と5の倍数を除く）:\n");  
  
 for (i = 1; i <= 30; i++)  
 {  
 if (i % 3 == 0 || i % 5 == 0)  
 {  
 continue;  
 }  
 printf("%d ", i);  
 }  
 printf("\n");  
  
 return 0;  
}```  
  
### for\_loop\_basics.c  
  
```c  
/\*  
 \* for文の基本的な使い方  
 \*  
 \* このプログラムは、for文を使った様々な  
 \* ループパターンを示します。  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 int i, j;  
 int sum;  
 char ch;  
 int n;  
 int factorial;  
  
 /\* 1. 基本的なfor文：1から10まで出力 \*/  
 printf("1から10まで: ");  
 for (i = 1; i <= 10; i++)  
 {  
 printf("%d ", i);  
 }  
 printf("\n\n");  
  
 /\* 2. 逆順ループ：10から1まで出力 \*/  
 printf("10から1まで（逆順）: ");  
 for (i = 10; i >= 1; i--)  
 {  
 printf("%d ", i);  
 }  
 printf("\n\n");  
  
 /\* 3. 偶数だけを出力 \*/  
 printf("0から20までの偶数: ");  
 for (i = 0; i <= 20; i += 2)  
 {  
 printf("%d ", i);  
 }  
 printf("\n\n");  
  
 /\* 4. 累計の計算 \*/  
 sum = 0;  
 for (i = 1; i <= 100; i++)  
 {  
 sum += i;  
 }  
 printf("1から100までの合計: %d\n\n", sum);  
  
 /\* 5. 九九の表（ネストしたfor文） \*/  
 printf("九九の表:\n");  
 printf(" ");  
 for (i = 1; i <= 9; i++)  
 {  
 printf("%3d ", i);  
 }  
 printf("\n");  
 printf(" ");  
 for (i = 1; i <= 9; i++)  
 {  
 printf("----");  
 }  
 printf("\n");  
  
 for (i = 1; i <= 9; i++)  
 {  
 printf("%2d |", i);  
 for (j = 1; j <= 9; j++)  
 {  
 printf("%3d ", i \* j);  
 }  
 printf("\n");  
 }  
 printf("\n");  
  
 /\* 6. 文字の出力 \*/  
 printf("アルファベット（大文字）: ");  
 for (ch = 'A'; ch <= 'Z'; ch++)  
 {  
 printf("%c ", ch);  
 }  
 printf("\n\n");  
  
 /\* 7. 複数変数を使ったfor文 \*/  
 printf("複数変数の例:\n");  
 for (i = 0, j = 10; i < j; i++, j--)  
 {  
 printf("i = %d, j = %d, 合計 = %d\n", i, j, i + j);  
 }  
 printf("\n");  
  
 /\* 8. 階乗の計算 \*/  
 n = 5;  
 factorial = 1;  
 printf("%d! = ", n);  
 for (i = n; i >= 1; i--)  
 {  
 factorial \*= i;  
 printf("%d", i);  
 if (i > 1)  
 printf(" x ");  
 }  
 printf(" = %d\n", factorial);  
  
 return 0;  
}```  
  
### while\_dowhile.c  
  
```c  
/\*  
 \* while文とdo-while文の使い方  
 \*  
 \* このプログラムは、while文とdo-while文の  
 \* 基本的な使い方と違いを示します。  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 int i;  
 int num;  
 int sum;  
 char choice;  
 int target;  
 int guess;  
 int attempts;  
  
 /\* 1. 基本的なwhile文 \*/  
 printf("=== while文の基本 ===\n");  
 i = 1;  
 while (i <= 5)  
 {  
 printf("カウント: %d\n", i);  
 i++;  
 }  
 printf("\n");  
  
 /\* 2. ユーザー入力を使ったwhile文 \*/  
 printf("=== 合計計算（0で終了）===\n");  
 sum = 0;  
 printf("数値を入力してください（0で終了）: ");  
 scanf("%d", &num);  
  
 while (num != 0)  
 {  
 sum += num;  
 printf("現在の合計: %d\n", sum);  
 printf("次の数値を入力してください（0で終了）: ");  
 scanf("%d", &num);  
 }  
 printf("最終合計: %d\n\n", sum);  
  
 /\* 3. do-while文の基本 \*/  
 printf("=== do-while文の基本 ===\n");  
 i = 1;  
 do  
 {  
 printf("カウント: %d\n", i);  
 i++;  
 } while (i <= 5);  
 printf("\n");  
  
 /\* 4. メニュー選択（do-whileの典型的な使用例） \*/  
 printf("=== メニューシステム ===\n");  
 do  
 {  
 printf("\n--- メインメニュー ---\n");  
 printf("1. 挨拶\n");  
 printf("2. 現在時刻の表示（ダミー）\n");  
 printf("3. カウントダウン\n");  
 printf("Q. 終了\n");  
 printf("選択してください: ");  
 scanf(" %c", &choice);  
  
 switch (choice)  
 {  
 case '1':  
 printf("こんにちは！\n");  
 break;  
 case '2':  
 printf("現在時刻: 12:00（ダミー）\n");  
 break;  
 case '3':  
 printf("カウントダウン開始: ");  
 for (i = 5; i >= 1; i--)  
 {  
 printf("%d... ", i);  
 }  
 printf("発射！\n");  
 break;  
 case 'Q':  
 case 'q':  
 printf("プログラムを終了します。\n");  
 break;  
 default:  
 printf("無効な選択です。\n");  
 }  
 } while (choice != 'Q' && choice != 'q');  
  
 /\* 5. while vs do-whileの違い \*/  
 printf("\n=== while vs do-while ===\n");  
  
 /\* 条件が最初から偽の場合のwhile \*/  
 printf("while文（条件が最初から偽）:\n");  
 i = 10;  
 while (i < 5)  
 {  
 printf("これは実行されません\n");  
 i++;  
 }  
 printf("while文は一度も実行されませんでした\n");  
  
 /\* 条件が最初から偽の場合のdo-while \*/  
 printf("\ndo-while文（条件が最初から偽）:\n");  
 i = 10;  
 do  
 {  
 printf("do-while文は最低1回実行されます（i = %d）\n", i);  
 i++;  
 } while (i < 5);  
  
 /\* 6. 数当てゲーム（簡易版） \*/  
 printf("\n=== 数当てゲーム ===\n");  
 target = 42; /\* 正解の数 \*/  
 attempts = 0;  
  
 printf("1から100までの数を当ててください!\n");  
  
 do  
 {  
 printf("予想を入力: ");  
 scanf("%d", &guess);  
 attempts++;  
  
 if (guess < target)  
 {  
 printf("もっと大きいです\n");  
 }  
 else if (guess > target)  
 {  
 printf("もっと小さいです\n");  
 }  
 else  
 {  
 printf("正解！ %d回で当たりました!\n", attempts);  
 }  
 } while (guess != target);  
  
 return 0;  
}```  
  
---  
  
# 第7章: 配列  
  
  
## 対応C規格  
- \*\*主要対象:\*\* C90  
- \*\*学習内容:\*\* 次元配列、多次元配列、文字列配列、配列の初期化、配列とポインタの関係  
  
## 学習目標  
  
この章を完了すると、以下のことができるようになります：  
  
- 次元配列の宣言・初期化・使用ができる  
- 多次元配列を理解して活用できる  
- 文字列配列の扱い方を完全に理解する  
- char配列とchar\*配列の違いを把握する  
- 配列を使った実践的なプログラムを作成できる  
  
## 概要と詳細  
  
### 配列とは？  
  
配列（はいれつ）は、プログラミングで最も重要なデータ構造の一つです。  
  
#### 日常生活での配列  
  
配列を理解するために、身近な例を考えてみましょう：  
  
1. \*\*教室の座席\*\*  
 - 5行6列の座席 = 2次元配列  
 - 各座席には生徒が座る = 配列の要素  
 - 「3行目の4番目の席」= 配列の[3][4]  
  
2. \*\*アパートの部屋番号\*\*  
 - 101号室、102号室... = 1次元配列  
 - 各部屋に住人がいる = 配列の要素  
  
3. \*\*カレンダー\*\*  
 - 曜日×週 = 2次元配列  
 - 各マスに日付 = 配列の要素  
  
#### なぜ配列が必要なの？  
  
配列がないと、たくさんのデータを扱うのが大変になります：  
  
```c  
/\* 配列を使わない場合（5人の点数） \*/  
int score1 = 85;  
int score2 = 92;  
int score3 = 78;  
int score4 = 91;  
int score5 = 88;  
  
/\* 配列を使う場合 \*/  
int scores[5] = {85, 92, 78, 91, 88};

配列を使えば、100人でも1000人でも簡単に管理できます！

### 配列の基本概念

配列は同じデータ型の要素を連続したメモリ領域に格納するデータ構造です。

#### 次元配列の宣言

データ型 配列名[要素数];

#### 基本的な配列の使用例

#include <stdio.h>  
  
int main(void)  
{  
  
 int numbers[5]; /\* 5個のint型要素を持つ配列 \*/  
 int i;  
   
 /\* 配列への値の代入 \*/  
 numbers[0] = 10;  
 numbers[1] = 20;  
 numbers[2] = 30;  
 numbers[3] = 40;  
 numbers[4] = 50;  
   
 /\* 配列の値を出力 \*/  
 for (i = 0; i < 5; i++) {  
 printf("numbers[%d] = %d\n", i, numbers[i]);  
 }  
   
   
 return 0;

### 配列の初期化

配列を使うときは、最初に値を設定（初期化）することが大切です。

#### 初心者が陥りやすいミス

int numbers[5]; /\* 初期化なし \*/  
printf("%d", numbers[0]); /\* 何が表示される？ \*/

初期化していない配列には「ゴミ値」が入っています。必ず初期化しましょう！

#### 宣言時の初期化

/\* 方法1: 全要素を明示的に初期化 \*/  
int scores[5] = {85, 92, 78, 91, 88};  
  
/\* 方法2: 部分的な初期化（残りは0で初期化） \*/  
int values[10] = {1, 2, 3}; /\* values[0]=1, values[1]=2, values[2]=3, 残りは0 \*/  
  
/\* 方法3: サイズを省略（要素数から自動決定） \*/  
int grades[] = {95, 87, 92, 76, 89}; /\* サイズは5になる \*/  
  
/\* 方法4: 全要素を0で初期化 \*/  
int zeros[10] = {0}; /\* 全要素が0 \*/

#### C90での初期化の制限

int main(void)  
{  
  
 int size = 5;  
 /\* int arr[size]; <- C90では不可（Variable Length Array） \*/  
   
 /\* C90では定数でサイズを指定 \*/  
 #define ARRAY\_SIZE 5  
 int arr[ARRAY\_SIZE] = {1, 2, 3, 4, 5};  
   
 return 0;

### 配列の操作

#### 配列要素へのアクセス

#include <stdio.h>  
  
int main(void)  
{  
  
 int data[5] = {10, 20, 30, 40, 50};  
 int i;  
   
 /\* 読み取り \*/  
 printf("3番目の要素: %d\n", data[2]); /\* 30が出力 \*/  
   
 /\* 書き込み \*/  
 data[2] = 35;  
   
 /\* ループでのアクセス \*/  
 for (i = 0; i < 5; i++) {  
 data[i] = data[i] \* 2; /\* 各要素を2倍 \*/  
 }  
   
   
 return 0;

#### 配列のサイズ計算

#include <stdio.h>  
  
int main(void)  
{  
  
 int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
 int size;  
 int i;  
   
 /\* 配列のサイズを計算 \*/  
 size = sizeof(numbers) / sizeof(numbers[0]);  
   
 printf("配列のサイズ: %d\n", size); /\* 10が出力 \*/  
   
 for (i = 0; i < size; i++) {  
 printf("%d ", numbers[i]);  
 }  
   
 printf("\n");  
   
 return 0;

### 文字配列（文字列）

C言語では、文字列は文字（char）の配列として扱います。これは初心者にとって重要な概念です。

#### 文字列の仕組み

文字列は「文字の並び + 終端文字（\0）」です：

H e l l o \0  
↑ ↑ ↑ ↑ ↑ ↑  
0 1 2 3 4 5番目

終端文字（\0）は「ここで文字列が終わり」という目印です。

#### char配列による文字列

#include <stdio.h>  
  
int main(void)  
{  
  
 char str1[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; /\* 明示的初期化 \*/  
 char str2[6] = "Hello"; /\* 文字列リテラル \*/  
 char str3[] = "Hello"; /\* サイズ自動決定 \*/  
   
 printf("str1: %s\n", str1);  
 printf("str2: %s\n", str2);  
 printf("str3: %s\n", str3);  
   
 return 0;

#### 文字配列の操作

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 char buffer[50];  
 char name[20];  
 int i;  
   
 /\* 文字列のコピー \*/  
 strcpy(buffer, "Hello, World!");  
 printf("buffer: %s\n", buffer);  
   
 /\* 文字づつのアクセス \*/  
 strcpy(name, "Alice");  
 for (i = 0; name[i] != '\0'; i++) {  
 printf("name[%d] = '%c'\n", i, name[i]);  
 }  
   
   
 return 0;

### 文字列配列の扱い

複数の文字列を扱う方法は、C言語学習者がよく迷うポイントです。ここでは2つの方法を詳しく説明します。

#### 方法: 次元char配列

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 /\* 各文字列の最大長を20文字とする5つの文字列 \*/  
 char names[5][20]; /\* 19文字+null終端文字 \*/  
 int i;  
   
 /\* 文字列配列への代入 \*/  
 strcpy(names[0], "Alice");  
 strcpy(names[1], "Bob");  
 strcpy(names[2], "Charlie");  
 strcpy(names[3], "David");  
 strcpy(names[4], "Eve");  
   
 /\* 文字列配列の表示 \*/  
 for (i = 0; i < 5; i++) {  
 printf("names[%d] = %s\n", i, names[i]);  
 }  
   
   
 return 0;

#### 方法: char\*配列（文字列ポインタ配列）

#include <stdio.h>  
  
int main(void)  
{  
  
 /\* 文字列ポインタの配列 \*/  
 char \*fruits[] = {  
 "Apple",  
 "Banana",   
 "Cherry",  
 "Date"  
 };  
 int i;  
   
 /\* 文字列ポインタ配列の表示 \*/  
 for (i = 0; i < 4; i++) {  
 printf("fruits[%d] = %s\n", i, fruits[i]);  
 }  
   
   
 return 0;

#### 文字列配列の初期化パターン

/\* パターン1: 2次元char配列の初期化 \*/  
char cities[3][10] = {  
 "Tokyo",  
 "Osaka",   
 "Kyoto"  
};  
  
/\* パターン2: char\*配列の初期化 \*/  
char \*colors[] = {  
 "Red",  
 "Green",  
 "Blue",  
 "Yellow"  
};  
  
/\* パターン3: 混合初期化 \*/  
char languages[4][15] = {  
 "C", /\* 文字列リテラル \*/  
 "Python", /\* 文字列リテラル \*/  
 "Java", /\* 文字列リテラル \*/  
 "JavaScript" /\* 文字列リテラル \*/  
};

### char配列 vs char\*配列の違い

これは文字列配列を扱う上で重要な概念です。初心者には少し難しいですが、違いを理解することで適切に使い分けられるようになります。

#### どちらを使うべきか？

**簡単な目安：** - 文字列を変更する必要がある → 2次元char配列 - 読み取り専用でメモリを節約したい → char\*配列 - 初心者のうちは → 2次元char配列が安全

#### メモリ配置の違い

#include <stdio.h>  
  
int main(void)  
{  
  
 /\* 2次元char配列 \*/  
 char matrix[3][4] = {"ABC", "DEF", "GHI"};  
   
 /\* char\*配列 \*/  
 char \*pointers[] = {"ABC", "DEF", "GHI"};  
   
 printf("=== 2次元char配列 ===\n");  
 printf("全体サイズ: %lu バイト\n", sizeof(matrix)); /\* 12バイト \*/  
 printf("1行のサイズ: %lu バイト\n", sizeof(matrix[0])); /\* 4バイト \*/  
   
 printf("\n=== char\*配列 ===\n");  
 printf("全体サイズ: %lu バイト\n", sizeof(pointers)); /\* 24バイト（8バイトx3） \*/  
 printf("要素のサイズ: %lu バイト\n", sizeof(pointers[0])); /\* 8バイト（ポインタサイズ） \*/  
   
 return 0;

#### 変更可能性の違い

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 /\* 2次元char配列（変更可能） \*/  
 char mutable\_array[3][10] = {"Hello", "World", "Test"};  
   
 /\* char\*配列（文字列リテラルは変更不可） \*/  
 char \*immutable\_array[] = {"Hello", "World", "Test"};  
   
 /\* 配列の内容変更 \*/  
 strcpy(mutable\_array[0], "Hi"); /\* OK: 内容変更可能 \*/  
 /\* strcpy(immutable\_array[0], "Hi"); <- NG: 文字列リテラルは変更不可 \*/  
   
 /\* ポインタの変更 \*/  
 /\* mutable\_array[0] = "New"; <- NG: 配列名は変更不可 \*/  
 immutable\_array[0] = "New"; /\* OK: ポインタ変更可能 \*/  
   
 printf("mutable\_array[0]: %s\n", mutable\_array[0]); /\* "Hi" \*/  
 printf("immutable\_array[0]: %s\n", immutable\_array[0]); /\* "New" \*/  
   
 return 0;

#### 用途に応じた使い分け

/\* 固定文字列の配列（読み取り専用） - char\*配列が適している \*/  
char \*error\_messages[] = {  
 "Success",  
 "File not found",  
 "Permission denied",  
 "Out of memory"  
};  
  
/\* 動的に変更する文字列配列 - 2次元char配列が適している \*/  
char user\_inputs[10][100]; /\* ユーザー入力を格納 \*/  
  
/\* 設定項目など（変更の可能性あり） - char\*配列が適している \*/  
char \*config\_items[] = {  
 "debug=on",  
 "log\_level=info",   
 "max\_connections=100"  
};

### 多次元配列

2次元以上の配列を多次元配列と呼びます。表やマトリックスを扱うときに便利です。

#### 2次元配列を理解しよう

2次元配列は「配列の配列」です。表計算ソフトのような構造をイメージしてください：

列0 列1 列2 列3  
行0 1 2 3 4  
行1 5 6 7 8   
行2 9 10 11 12

これをコードで表すと：

int matrix[3][4]; /\* 3行4列の2次元配列 \*/

#### 次元配列の基本

#include <stdio.h>  
  
int main(void)  
{  
  
 int matrix[3][4] = {  
 {1, 2, 3, 4},  
 {5, 6, 7, 8},  
 {9, 10, 11, 12}  
 };  
 int i, j;  
   
 /\* 2次元配列の表示 \*/  
 for (i = 0; i < 3; i++) {  
 for (j = 0; j < 4; j++) {  
 printf("%d ", matrix[i][j]);  
 }  
 printf("\n");  
 }  
   
 return 0;

#### 次元配列の例

#include <stdio.h>  
  
int main(void)  
{  
  
 /\* 3次元配列: [2][3][4] \*/  
 int cube[2][3][4] = {  
 {  
 {1, 2, 3, 4},  
 {5, 6, 7, 8},  
 {9, 10, 11, 12}  
 },  
 {  
 {13, 14, 15, 16},  
 {17, 18, 19, 20},  
 {21, 22, 23, 24}  
 }  
 };  
 int i, j, k;  
   
 for (i = 0; i < 2; i++) {  
 printf("=== Layer %d ===\n", i);  
 for (j = 0; j < 3; j++) {  
 for (k = 0; k < 4; k++) {  
 printf("%d ", cube[i][j][k]);  
 }  
 printf("\n");  
 }  
 printf("\n");  
 }  
   
 return 0;

### 配列の実践的応用

#### 成績管理システム

#include <stdio.h>  
  
#define MAX\_STUDENTS 5  
#define MAX\_SUBJECTS 3  
  
int main(void)  
{  
  
 char students[MAX\_STUDENTS][20] = {  
 "Alice", "Bob", "Charlie", "David", "Eve"  
 };  
 char subjects[MAX\_SUBJECTS][10] = {  
 "Math", "English", "Science"  
 };  
 int scores[MAX\_STUDENTS][MAX\_SUBJECTS] = {  
 {85, 92, 78}, /\* Alice \*/  
 {90, 87, 85}, /\* Bob \*/  
 {95, 91, 89}, /\* Charlie \*/  
 {82, 79, 88}, /\* David \*/  
 {91, 88, 93} /\* Eve \*/  
 };  
 int i, j;  
 int total;  
   
 /\* 成績表の表示 \*/  
 printf("Student ");  
 for (j = 0; j < MAX\_SUBJECTS; j++) {  
 printf("%-10s", subjects[j]);  
 }  
 printf("Average\n");  
 printf("----------------------------------------\n");  
   
 for (i = 0; i < MAX\_STUDENTS; i++) {  
 printf("%-10s ", students[i]);  
 total = 0;  
 for (j = 0; j < MAX\_SUBJECTS; j++) {  
 printf("%-10d", scores[i][j]);  
 total += scores[i][j];  
 }  
 printf("%.1f\n", (double)total / MAX\_SUBJECTS);  
 }  
   
 return 0;

#### 文字列配列のソート

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_NAMES 5  
#define MAX\_LENGTH 20  
  
int main(void)  
{  
  
 char names[MAX\_NAMES][MAX\_LENGTH] = {  
 "Charlie", "Alice", "Eve", "Bob", "David"  
 };  
 char temp[MAX\_LENGTH];  
 int i, j;  
   
 printf("ソート前:\n");  
 for (i = 0; i < MAX\_NAMES; i++) {  
 printf("%s ", names[i]);  
 }  
 printf("\n");  
   
 /\* バブルソート \*/  
 for (i = 0; i < MAX\_NAMES - 1; i++) {  
 for (j = 0; j < MAX\_NAMES - 1 - i; j++) {  
 if (strcmp(names[j], names[j + 1]) > 0) {  
 strcpy(temp, names[j]);  
 strcpy(names[j], names[j + 1]);  
 strcpy(names[j + 1], temp);  
 }  
 }  
 }  
   
 printf("ソート後:\n");  
 for (i = 0; i < MAX\_NAMES; i++) {  
 printf("%s ", names[i]);  
 }  
 printf("\n");  
   
 return 0;

### 配列とポインタの関係

配列名はその先頭要素へのポインタとして動作します。これは少し高度な話題ですが、C言語を深く理解するために重要です。

#### 初心者のための説明

配列名は「配列の最初の要素の住所」を表します：

int arr[5] = {10, 20, 30, 40, 50};  
/\* arr は &arr[0] と同じ意味 \*/

詳しくは第10章（ポインタ）で学習します。今は「配列名は特別な意味を持つ」ことだけ覚えておきましょう。

#include <stdio.h>  
  
int main(void)  
{  
  
 int arr[5] = {10, 20, 30, 40, 50};  
 int \*ptr;  
 int i;  
   
 ptr = arr; /\* arr は &arr[0] と同じ \*/  
   
 printf("=== 配列表記 ===\n");  
 for (i = 0; i < 5; i++) {  
 printf("arr[%d] = %d\n", i, arr[i]);  
 }  
   
 printf("\n=== ポインタ表記 ===\n");  
 for (i = 0; i < 5; i++) {  
 printf("\*(ptr + %d) = %d\n", i, \*(ptr + i));  
 }  
   
 /\* 配列名とポインタの関係 \*/  
 printf("\n=== アドレス比較 ===\n");  
 printf("arr = %p\n", (void\*)arr);  
 printf("&arr[0] = %p\n", (void\*)&arr[0]);  
 printf("ptr = %p\n", (void\*)ptr);  
   
 return 0;

## 実践的なプログラム例

### 基本的な配列操作（C90準拠）

#### 1次元配列の基本

#include <stdio.h>  
  
int main(void)  
{  
 /\* C90では先頭で全ての変数を宣言 \*/  
 int numbers[5] = {10, 20, 30, 40, 50};  
 int total = 0;  
 int i;  
   
 /\* 配列の全要素を表示 \*/  
 printf("=== 配列の内容 ===\n");  
 for (i = 0; i < 5; i++) {  
 printf("numbers[%d] = %d\n", i, numbers[i]);  
 total += numbers[i];  
 }  
   
 printf("合計: %d\n", total);  
 printf("平均: %.1f\n", (double)total / 5);  
   
 return 0;  
}

📁 **ファイル**: <examples/array_basics.c>  
📁 **C99版**: <examples/array_basics_c99.c>

### 文字列配列の実例（C90準拠）

#### char配列 vs char\*配列の違い

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
 /\* C90では先頭で全ての変数を宣言 \*/  
 char cities[3][20] = {"Tokyo", "Osaka", "Kyoto"}; /\* 2次元char配列 \*/  
 char \*fruits[] = {"Apple", "Banana", "Cherry"}; /\* char\*配列 \*/  
 int i;  
   
 printf("=== 2次元char配列 ===\n");  
 for (i = 0; i < 3; i++) {  
 printf("cities[%d] = %s\n", i, cities[i]);  
 }  
   
 printf("\n=== char\*配列 ===\n");  
 for (i = 0; i < 3; i++) {  
 printf("fruits[%d] = %s\n", i, fruits[i]);  
 }  
   
 /\* メモリサイズの違い \*/  
 printf("\n=== メモリサイズ比較 ===\n");  
 printf("cities配列のサイズ: %lu バイト\n", (unsigned long)sizeof(cities));  
 printf("fruits配列のサイズ: %lu バイト\n", (unsigned long)sizeof(fruits));  
   
 return 0;  
}

📁 **ファイル**: <examples/string_arrays.c>  
📁 **C99版**: <examples/string_arrays_c99.c>

### 多次元配列の活用（C90準拠）

#### 3x4行列の操作

#include <stdio.h>  
  
int main(void)  
{  
 /\* C90では先頭で全ての変数を宣言 \*/  
 int matrix[3][4] = {  
 {1, 2, 3, 4},  
 {5, 6, 7, 8},  
 {9, 10, 11, 12}  
 };  
 int i, j;  
 int row\_sum;  
   
 printf("=== 行列の表示 ===\n");  
 for (i = 0; i < 3; i++) {  
 row\_sum = 0;  
 for (j = 0; j < 4; j++) {  
 printf("%3d ", matrix[i][j]);  
 row\_sum += matrix[i][j];  
 }  
 printf("| 行の合計: %d\n", row\_sum);  
 }  
   
 return 0;  
}

📁 **ファイル**: <examples/multidimensional_arrays.c>  
📁 **C99版**: <examples/multidimensional_arrays_c99.c>

### コンパイル方法

#### 基本的なコンパイル（C90準拠）

gcc -std=c90 -Wall -Wextra -pedantic array\_basics.c -o array\_basics

#### Makefileを使用した場合

# 全てのプログラムをコンパイル  
make all  
  
# 特定のプログラムをコンパイル   
make array\_basics  
  
# C99版をコンパイル  
make array\_basics\_c99  
  
# プログラムを実行  
make run-all  
  
# クリーンアップ  
make clean

## 演習課題と解答例

### 利用可能な演習問題

[演習問題詳細](exercises/README.md)では、配列と文字列配列の理解を深める課題が提供されています。

#### 演習6-1: 配列の基本操作（C90準拠）

#include <stdio.h>  
  
int main(void)  
{  
 /\* C90では先頭で全ての変数を宣言 \*/  
 int numbers[10] = {15, 23, 8, 42, 7, 31, 19, 5, 28, 36};  
 int i;  
 int max, min, sum;  
 double average;  
   
 /\* 最大値、最小値、合計の初期化 \*/  
 max = min = sum = numbers[0];  
   
 /\* 配列の全要素を処理 \*/  
 for (i = 1; i < 10; i++) {  
 if (numbers[i] > max) max = numbers[i];  
 if (numbers[i] < min) min = numbers[i];  
 sum += numbers[i];  
 }  
   
 average = (double)sum / 10;  
   
 printf("=== 配列の統計情報 ===\n");  
 printf("最大値: %d\n", max);  
 printf("最小値: %d\n", min);  
 printf("合計: %d\n", sum);  
 printf("平均: %.1f\n", average);  
   
 return 0;  
}

📁 **解答例**: <solutions/ex6_1.c>  
📁 **C99版**: <solutions/ex6_1_c99.c>

#### 演習6-2: 成績管理システム（C90準拠）

#include <stdio.h>  
  
#define NUM\_STUDENTS 5  
#define NUM\_SUBJECTS 4  
  
/\* 学生名と科目名 \*/  
char students[NUM\_STUDENTS][20] = {  
 "田中太郎", "佐藤花子", "鈴木一郎", "高橋美咲", "伊藤健太"  
};  
  
char subjects[NUM\_SUBJECTS][10] = {  
 "国語", "数学", "英語", "理科"  
};  
  
int main(void)  
{  
 /\* C90では先頭で全ての変数を宣言 \*/  
 int grades[NUM\_STUDENTS][NUM\_SUBJECTS];  
 int i, j;  
   
 printf("=== 成績管理システム ===\n");  
 printf("5人の学生の4科目の成績を入力してください。\n\n");  
   
 /\* 成績の入力 \*/  
 for (i = 0; i < NUM\_STUDENTS; i++) {  
 printf("--- %s の成績 ---\n", students[i]);  
 for (j = 0; j < NUM\_SUBJECTS; j++) {  
 printf("%s: ", subjects[j]);  
 scanf("%d", &grades[i][j]);  
 }  
 printf("\n");  
 }  
   
 /\* 成績表の表示処理などが続く... \*/  
   
 return 0;  
}

📁 **解答例**: <solutions/ex6_2.c>  
📁 **C99版**: <solutions/ex6_2_c99.c>

#### 演習6-3: 文字列配列の操作（C90準拠）

#include <stdio.h>  
#include <string.h>  
  
#define NUM\_PREFECTURES 10  
#define MAX\_NAME\_LENGTH 20  
  
/\* 都道府県名のサンプルデータ \*/  
char prefectures[NUM\_PREFECTURES][MAX\_NAME\_LENGTH] = {  
 "東京都", "神奈川県", "大阪府", "愛知県", "埼玉県",  
 "千葉県", "兵庫県", "北海道", "福岡県", "静岡県"  
};  
  
int main(void)  
{  
 /\* C90では先頭で全ての変数を宣言 \*/  
 char working\_array[NUM\_PREFECTURES][MAX\_NAME\_LENGTH];  
 int i;  
   
 printf("=== 都道府県管理システム ===\n\n");  
   
 /\* 作業用配列にコピー \*/  
 for (i = 0; i < NUM\_PREFECTURES; i++) {  
 strcpy(working\_array[i], prefectures[i]);  
 }  
   
 /\* ソート、検索処理などが続く... \*/  
   
 return 0;  
}

📁 **解答例**: <solutions/ex6_3.c>  
📁 **C99版**: <solutions/ex6_3_c99.c>

### 演習問題一覧

1. **演習6-1**: 配列の基本操作（最大値・最小値・平均値）
2. **演習6-2**: 成績管理システム（5人×4科目）
3. **演習6-3**: 文字列配列の操作（都道府県管理）
4. **演習6-4**: 行列演算（加算・減算・乗算・転置）
5. **演習6-5**: 簡易辞書プログラム（英和・和英検索）

### チャレンジ問題

* **チャレンジ1**: ソートアルゴリズムの実装と比較
* **チャレンジ2**: ライフゲーム（Conway’s Game of Life）

すべての演習には**C90版**と**C99版**の両方の解答例が提供されています。

## 学習フローとコンパイル方法

### 推奨学習順序

1. **理論学習**: このREADME.mdで配列の概念を理解
2. **サンプルコード**: examples/の基本例を確認
3. **演習課題**: exercises/README.mdで課題を確認
4. **実装練習**: solutions/の解答例を参考に自分で実装

### Makefileを使用したコンパイル

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make array\_basics # C90版  
make array\_basics\_c99 # C99版  
  
# 全てのプログラムを実行  
make run-all  
  
# C90準拠でのテスト  
make test  
  
# 複数のC標準でのテスト  
make test-standards  
  
# クリーンアップ  
make clean  
  
# ヘルプ表示  
make help

### 手動コンパイル

# C90準拠でコンパイル（推奨）  
gcc -std=c90 -Wall -Wextra -pedantic examples/array\_basics.c -o array\_basics  
  
# C99準拠でコンパイル  
gcc -std=c99 -Wall -Wextra -pedantic examples/array\_basics\_c99.c -o array\_basics\_c99  
  
# 実行  
./array\_basics

## C90とC99の違い

### C90の特徴（このチュートリアルの基準）

int main(void)  
{  
 /\* すべての変数を関数の先頭で宣言 \*/  
 int numbers[5] = {1, 2, 3, 4, 5};  
 int i;  
 int sum = 0;  
   
 /\* 配列サイズは定数で指定 \*/  
 #define ARRAY\_SIZE 10  
 int arr[ARRAY\_SIZE];  
   
 /\* forループは従来のスタイル \*/  
 for (i = 0; i < 5; i++) {  
 sum += numbers[i];  
 }  
   
 return 0;  
}

### C99の拡張機能

int main(void)  
{  
 int numbers[5] = {1, 2, 3, 4, 5};  
   
 /\* forループ内での変数宣言（C99） \*/  
 for (int i = 0; i < 5; i++) {  
 printf("%d ", numbers[i]);  
 }  
   
 /\* 必要な時点での変数宣言（C99） \*/  
 int sum = 0;  
 for (int i = 0; i < 5; i++) {  
 sum += numbers[i];  
 }  
   
 /\* 可変長配列（VLA）も可能（C99） \*/  
 int size = 10;  
 int vla[size]; /\* C90では不可 \*/  
   
 return 0;  
}

### このチュートリアルでの方針

* **C90版**: 基準となる実装、互換性重視
* **C99版**: 現代的な書き方、読みやすさ重視
* **学習推奨**: まずC90で基礎を固めてからC99を学習

## よくある間違いとデバッグ方法

初心者が配列でつまずきやすいポイントと、その解決方法を紹介します。

### 1. 配列の境界外アクセス

**最も危険で、最もよくあるミス**です！

#### 間違った例

/\* NG: 配列の範囲を超える \*/  
int arr[5] = {1, 2, 3, 4, 5};  
printf("%d\n", arr[5]); /\* 範囲外アクセス - 未定義動作 \*/

#### 正しい例

/\* OK: 正しい範囲でアクセス \*/  
#include <stdio.h>  
  
int main(void)  
{  
 int arr[5] = {1, 2, 3, 4, 5};  
 int i;  
   
 /\* 安全な配列アクセス \*/  
 for (i = 0; i < 5; i++) { /\* i < 5 であることを確認 \*/  
 printf("arr[%d] = %d\n", i, arr[i]);  
 }  
   
 return 0;  
}

### 2. 文字列配列の初期化忘れ

文字列を扱うときは、必ず初期化しましょう。

#### 間違った例

/\* NG: 初期化されていない文字配列 \*/  
char str[20];  
printf("%s\n", str); /\* 未定義動作 \*/

#### 正しい例

/\* OK: 適切な初期化 \*/  
#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
 char str[20] = ""; /\* 空文字列で初期化 \*/  
 strcpy(str, "Hello"); /\* 後から代入 \*/  
 printf("%s\n", str);  
   
 return 0;  
}

### 3. 配列代入の間違い

配列は変数とは違って、=で代入できません。これは初心者がよく間違えるポイントです。

#### 間違った例

/\* NG: 配列は直接代入できない \*/  
char str1[20], str2[20] = "Hello";  
str1 = str2; /\* コンパイルエラー \*/

#### 正しい例

/\* OK: strcpyを使用 \*/  
#include <string.h>  
  
char str1[20], str2[20] = "Hello";  
strcpy(str1, str2); /\* 正しい文字列コピー \*/

### 4. C90とC99の混在エラー

#### C90でコンパイル時のエラー例

/\* C90ではエラーになる書き方 \*/  
int main(void)  
{  
 int arr[5] = {1, 2, 3, 4, 5};  
 for (int i = 0; i < 5; i++) { /\* C90ではNG \*/  
 printf("%d ", arr[i]);  
 }  
 return 0;  
}

#### 正しいC90の書き方

/\* C90準拠の書き方 \*/  
int main(void)  
{  
 int arr[5] = {1, 2, 3, 4, 5};  
 int i; /\* 先頭で宣言 \*/  
   
 for (i = 0; i < 5; i++) {  
 printf("%d ", arr[i]);  
 }  
 return 0;  
}

### デバッグのコツ

* **境界チェック**: 配列アクセス時は必ずインデックスが範囲内かチェック
* **初期化の確認**: 配列や文字列は使用前に必ず初期化
* **コンパイル標準の確認**: -std=c90 または -std=c99 の指定を確認
* **解答例との比較**: solutions/の正解コードと比較して違いを確認

## 次の章へ

配列と文字列配列を理解したら、[文字列処理](../strings/README.md) に進んでください。

## 参考資料

* [C言語配列リファレンス](https://ja.cppreference.com/w/c/language/array)
* [文字列操作関数](https://ja.cppreference.com/w/c/string/byte)
* [多次元配列の詳細](https://ja.cppreference.com/w/c/language/array)

## サンプルコード

### array\_basics.c

/\*  
 \* 配列の基本的な使い方（C90準拠版）  
 \*   
 \* このプログラムは、1次元配列の宣言、初期化、  
 \* 操作の基本的な方法を示します。  
 \*/  
#include <stdio.h>  
  
#define MAX\_SIZE 10  
  
int main(void)  
{  
 /\* C90では先頭で全ての変数を宣言 \*/  
 int numbers[5];  
 int i;  
 int scores[5] = {85, 92, 78, 96, 88};  
 int partial[10] = {1, 2, 3};  
 int auto\_size[] = {5, 10, 15, 20, 25, 30};  
 int size;  
 int test\_scores[] = {75, 88, 92, 67, 85, 91, 78, 84, 90, 86};  
 int n;  
 int sum = 0;  
 int max;  
 int min;  
 int search\_array[] = {15, 23, 8, 42, 16, 4, 11, 29, 36, 19};  
 int search\_value = 42;  
 int found = 0;  
 int position = -1;  
 int original[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
 int source[5] = {11, 22, 33, 44, 55};  
 int destination[5];  
 int arr[MAX\_SIZE];  
   
 printf("=== 配列の基本的な使用 ===\n");  
   
 /\* 配列に値を代入 \*/  
 for (i = 0; i < 5; i++) {  
 numbers[i] = (i + 1) \* 10;  
 }  
   
 /\* 配列の内容を表示 \*/  
 printf("配列の内容: ");  
 for (i = 0; i < 5; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n\n");  
   
 /\* 2. 配列の初期化 \*/  
 printf("=== 配列の初期化 ===\n");  
   
 /\* 完全な初期化 \*/  
 printf("scores配列: ");  
 for (i = 0; i < 5; i++) {  
 printf("%d ", scores[i]);  
 }  
 printf("\n");  
   
 /\* 部分的な初期化 \*/  
 printf("partial配列: ");  
 for (i = 0; i < 10; i++) {  
 printf("%d ", partial[i]);  
 }  
 printf("\n");  
   
 /\* サイズを省略した初期化 \*/  
 size = sizeof(auto\_size) / sizeof(auto\_size[0]);  
 printf("auto\_size配列（要素数%d）: ", size);  
 for (i = 0; i < size; i++) {  
 printf("%d ", auto\_size[i]);  
 }  
 printf("\n\n");  
   
 /\* 3. 配列の統計計算 \*/  
 printf("=== 配列の統計計算 ===\n");  
   
 n = sizeof(test\_scores) / sizeof(test\_scores[0]);  
 sum = 0;  
 max = test\_scores[0];  
 min = test\_scores[0];  
   
 /\* 合計、最大値、最小値を計算 \*/  
 for (i = 0; i < n; i++) {  
 sum += test\_scores[i];  
 if (test\_scores[i] > max) {  
 max = test\_scores[i];  
 }  
 if (test\_scores[i] < min) {  
 min = test\_scores[i];  
 }  
 }  
   
 printf("テストの点数: ");  
 for (i = 0; i < n; i++) {  
 printf("%d ", test\_scores[i]);  
 }  
 printf("\n");  
 printf("要素数: %d\n", n);  
 printf("合計: %d\n", sum);  
 printf("平均: %.2f\n", (double)sum / n);  
 printf("最高点: %d\n", max);  
 printf("最低点: %d\n\n", min);  
   
 /\* 4. 配列の要素検索 \*/  
 printf("=== 配列の検索 ===\n");  
   
 /\* 線形探索 \*/  
 for (i = 0; i < 10; i++) {  
 if (search\_array[i] == search\_value) {  
 found = 1;  
 position = i;  
 break;  
 }  
 }  
   
 if (found) {  
 printf("値 %d は位置 %d にあります\n", search\_value, position);  
 } else {  
 printf("値 %d は見つかりませんでした\n", search\_value);  
 }  
 printf("\n");  
   
 /\* 5. 配列の逆順表示 \*/  
 printf("=== 配列の逆順表示 ===\n");  
   
 printf("元の配列: ");  
 for (i = 0; i < 10; i++) {  
 printf("%d ", original[i]);  
 }  
 printf("\n");  
   
 printf("逆順: ");  
 for (i = 9; i >= 0; i--) {  
 printf("%d ", original[i]);  
 }  
 printf("\n\n");  
   
 /\* 6. 配列のコピー \*/  
 printf("=== 配列のコピー ===\n");  
   
 /\* 要素を1つずつコピー \*/  
 for (i = 0; i < 5; i++) {  
 destination[i] = source[i];  
 }  
   
 printf("source配列: ");  
 for (i = 0; i < 5; i++) {  
 printf("%d ", source[i]);  
 }  
 printf("\n");  
   
 printf("destination配列: ");  
 for (i = 0; i < 5; i++) {  
 printf("%d ", destination[i]);  
 }  
 printf("\n\n");  
   
 /\* 7. 配列サイズの注意点 \*/  
 printf("=== 配列サイズの情報 ===\n");  
   
 printf("配列のサイズ: %lu バイト\n", sizeof(arr));  
 printf("要素1つのサイズ: %lu バイト\n", sizeof(arr[0]));  
 printf("要素数: %lu\n", sizeof(arr) / sizeof(arr[0]));  
   
 return 0;  
}

C99版は[こちら](array_basics_c99.c)を参照してください。

### multidimensional\_arrays.c

/\*  
 \* 多次元配列の使い方（C90準拠版）  
 \*  
 \* このプログラムは、2次元配列と3次元配列の  
 \* 宣言、初期化、操作方法を示します。  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 /\* C90では先頭で全ての変数を宣言 \*/  
 int i, j;  
 int matrix[3][4] = {  
 {1, 2, 3, 4},  
 {5, 6, 7, 8},  
 {9, 10, 11, 12}};  
 int grades[5][3] = {  
 {85, 92, 78}, /\* 学生1 \*/  
 {90, 88, 85}, /\* 学生2 \*/  
 {78, 85, 92}, /\* 学生3 \*/  
 {92, 76, 88}, /\* 学生4 \*/  
 {88, 90, 82} /\* 学生5 \*/  
 };  
 char \*subjects[] = {"国語", "数学", "英語"};  
 int partial[4][5] = {  
 {1, 2}, /\* 残りの要素は0 \*/  
 {3, 4, 5}, /\* 残りの要素は0 \*/  
 {6} /\* 残りの要素は0 \*/  
 /\* 4行目は全て0 \*/  
 };  
 int identity[5][5];  
 int image[2][3][3] = {  
 {  
 {255, 0, 0}, /\* 赤 \*/  
 {0, 255, 0}, /\* 緑 \*/  
 {0, 0, 255} /\* 青 \*/  
 },  
 {  
 {255, 255, 0}, /\* 黄 \*/  
 {255, 0, 255}, /\* マゼンタ \*/  
 {0, 255, 255} /\* シアン \*/  
 }};  
 int original\_matrix[3][4] = {  
 {1, 2, 3, 4},  
 {5, 6, 7, 8},  
 {9, 10, 11, 12}};  
 int transposed[4][3];  
 int sum;  
  
 /\* 1. 2次元配列の基本 \*/  
 printf("=== 2次元配列の基本 ===\n");  
  
 /\* 3x4の2次元配列 \*/  
  
 /\* 2次元配列の表示 \*/  
 printf("3x4の行列:\n");  
 for (i = 0; i < 3; i++)  
 {  
 for (j = 0; j < 4; j++)  
 {  
 printf("%3d ", matrix[i][j]);  
 }  
 printf("\n");  
 }  
 printf("\n");  
  
 /\* 2. 成績表の管理 \*/  
 printf("=== 成績表（学生x科目）===\n");  
  
 /\* 5人の学生、3科目（国語、数学、英語）の成績 \*/  
  
 /\* 成績表の表示 \*/  
 printf(" ");  
 for (j = 0; j < 3; j++)  
 {  
 printf("%-6s ", subjects[j]);  
 }  
 printf("平均\n");  
 printf("----------------------------------------\n");  
  
 for (i = 0; i < 5; i++)  
 {  
 printf("学生%d: ", i + 1);  
 sum = 0;  
 for (j = 0; j < 3; j++)  
 {  
 printf("%-6d ", grades[i][j]);  
 sum += grades[i][j];  
 }  
 printf("%.1f\n", sum / 3.0);  
 }  
  
 /\* 科目別平均 \*/  
 printf("----------------------------------------\n");  
 printf("平均: ");  
 for (j = 0; j < 3; j++)  
 {  
 sum = 0;  
 for (i = 0; i < 5; i++)  
 {  
 sum += grades[i][j];  
 }  
 printf("%-6.1f ", sum / 5.0);  
 }  
 printf("\n\n");  
  
 /\* 3. 部分的な初期化 \*/  
 printf("=== 2次元配列の部分初期化 ===\n");  
  
 /\* 部分的に初期化（残りは0） \*/  
  
 printf("部分的に初期化された配列:\n");  
 for (i = 0; i < 4; i++)  
 {  
 for (j = 0; j < 5; j++)  
 {  
 printf("%2d ", partial[i][j]);  
 }  
 printf("\n");  
 }  
 printf("\n");  
  
 /\* 4. 単位行列の作成 \*/  
 printf("=== 単位行列の作成 ===\n");  
  
 /\* 単位行列を作成 \*/  
 for (i = 0; i < 5; i++)  
 {  
 for (j = 0; j < 5; j++)  
 {  
 if (i == j)  
 {  
 identity[i][j] = 1;  
 }  
 else  
 {  
 identity[i][j] = 0;  
 }  
 }  
 }  
  
 /\* 単位行列を表示 \*/  
 printf("5x5の単位行列:\n");  
 for (i = 0; i < 5; i++)  
 {  
 for (j = 0; j < 5; j++)  
 {  
 printf("%d ", identity[i][j]);  
 }  
 printf("\n");  
 }  
 printf("\n");  
  
 /\* 5. 3次元配列の例 \*/  
 printf("=== 3次元配列（RGB画像データの例）===\n");  
  
 /\* 2x3ピクセル、各ピクセルはRGB（3値） \*/  
  
 printf("画像データ（2x3ピクセル）:\n");  
 for (i = 0; i < 2; i++)  
 {  
 printf("行 %d:\n", i);  
 for (j = 0; j < 3; j++)  
 {  
 printf(" ピクセル[%d][%d]: R=%3d, G=%3d, B=%3d\n",  
 i, j, image[i][j][0], image[i][j][1], image[i][j][2]);  
 }  
 }  
 printf("\n");  
  
 /\* 6. 転置行列 \*/  
 printf("=== 行列の転置 ===\n");  
  
 /\* 転置を計算 \*/  
 for (i = 0; i < 3; i++)  
 {  
 for (j = 0; j < 4; j++)  
 {  
 transposed[j][i] = original\_matrix[i][j];  
 }  
 }  
  
 printf("元の行列（3x4）:\n");  
 for (i = 0; i < 3; i++)  
 {  
 for (j = 0; j < 4; j++)  
 {  
 printf("%3d ", original\_matrix[i][j]);  
 }  
 printf("\n");  
 }  
  
 printf("\n転置行列（4x3）:\n");  
 for (i = 0; i < 4; i++)  
 {  
 for (j = 0; j < 3; j++)  
 {  
 printf("%3d ", transposed[i][j]);  
 }  
 printf("\n");  
 }  
  
 return 0;  
}```  
  
> C99版は[こちら](multidimensional\_arrays\_c99.c)を参照してください。  
  
### string\_arrays.c  
  
```c  
/\*  
 \* 文字列配列の使い方（C90準拠版）  
 \*   
 \* このプログラムは、文字列配列の宣言、初期化、  
 \* 操作方法、およびchar配列とchar\*配列の違いを示します。  
 \*/  
#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
 /\* C90では先頭で全ての変数を宣言 \*/  
 int i;  
 char str1[20];  
 char str2[] = "World";  
 char fruits[5][20] = {  
 "Apple",  
 "Banana",  
 "Orange",  
 "Grape",  
 "Strawberry"  
 };  
 char \*months[] = {  
 "January", "February", "March", "April",  
 "May", "June", "July", "August",  
 "September", "October", "November", "December"  
 };  
 char cities1[3][20] = {"Tokyo", "New York", "London"};  
 char \*cities2[] = {"Tokyo", "New York", "London"};  
 char \*programming\_languages[] = {  
 "C", "C++", "Java", "Python", "JavaScript",  
 "Ruby", "Go", "Rust", "Swift", "Kotlin"  
 };  
 int n\_languages;  
 char search\_lang[20];  
 int found = 0;  
 char \*words[] = {"cat", "elephant", "dog", "hippopotamus", "ant", "butterfly"};  
 int n\_words;  
 int len;  
 char \*menu\_items[] = {  
 "1. ファイルを開く",  
 "2. ファイルを保存",  
 "3. 設定",  
 "4. ヘルプ",  
 "5. 終了"  
 };  
 int n\_items;  
   
 /\* 1. char配列（文字配列）の基本 \*/  
 printf("=== char配列（文字配列）===\n");  
   
 strcpy(str1, "Hello"); /\* C90では初期化後に代入 \*/  
   
 printf("str1: %s（サイズ: %lu）\n", str1, sizeof(str1));  
 printf("str2: %s（サイズ: %lu）\n", str2, sizeof(str2));  
   
 /\* 文字配列の変更 \*/  
 str1[0] = 'h'; /\* 最初の文字を小文字に \*/  
 printf("変更後のstr1: %s\n\n", str1);  
   
 /\* 2. 文字列配列（2次元char配列）\*/  
 printf("=== 2次元char配列による文字列配列 ===\n");  
   
 printf("果物リスト:\n");  
 for (i = 0; i < 5; i++) {  
 printf("%d: %s\n", i + 1, fruits[i]);  
 }  
   
 /\* 文字列の変更 \*/  
 strcpy(fruits[2], "Mango"); /\* OrangeをMangoに変更 \*/  
 printf("\n変更後（3番目をMangoに）:\n");  
 for (i = 0; i < 5; i++) {  
 printf("%d: %s\n", i + 1, fruits[i]);  
 }  
 printf("\n");  
   
 /\* 3. char\*配列（ポインタ配列）\*/  
 printf("=== char\*配列による文字列配列 ===\n");  
   
 printf("月の名前:\n");  
 for (i = 0; i < 12; i++) {  
 printf("%2d月: %s\n", i + 1, months[i]);  
 }  
 printf("\n");  
   
 /\* 4. char配列とchar\*配列の違い \*/  
 printf("=== char配列とchar\*配列の違い ===\n");  
   
 printf("char配列のサイズ: %lu バイト\n", sizeof(cities1));  
 printf("char\*配列のサイズ: %lu バイト\n", sizeof(cities2));  
 printf("（ポインタのサイズx要素数）\n\n");  
   
 /\* 5. 文字列の検索と操作 \*/  
 printf("=== 文字列配列の検索 ===\n");  
   
 n\_languages = sizeof(programming\_languages) / sizeof(programming\_languages[0]);  
   
 printf("検索する言語名を入力: ");  
 scanf("%s", search\_lang);  
   
 found = 0;  
 for (i = 0; i < n\_languages; i++) {  
 if (strcmp(programming\_languages[i], search\_lang) == 0) {  
 printf("%s は %d 番目にあります\n", search\_lang, i + 1);  
 found = 1;  
 break;  
 }  
 }  
   
 if (!found) {  
 printf("%s は見つかりませんでした\n", search\_lang);  
 }  
 printf("\n");  
   
 /\* 6. 文字列の長さ順にソート（簡易版）\*/  
 printf("=== 文字列の長さ順表示 ===\n");  
   
 n\_words = sizeof(words) / sizeof(words[0]);  
   
 printf("元の順序:\n");  
 for (i = 0; i < n\_words; i++) {  
 printf("%s ", words[i]);  
 }  
 printf("\n");  
   
 /\* 長さ順に表示（実際の配列は変更しない） \*/  
 printf("\n長さ順:\n");  
 for (len = 1; len <= 20; len++) {  
 for (i = 0; i < n\_words; i++) {  
 if ((int)strlen(words[i]) == len) { /\* 型キャストで警告を回避 \*/  
 printf("%s（%d文字） ", words[i], len);  
 }  
 }  
 }  
 printf("\n\n");  
   
 /\* 7. 動的な文字列配列の例 \*/  
 printf("=== コマンドメニューシステム ===\n");  
   
 n\_items = sizeof(menu\_items) / sizeof(menu\_items[0]);  
   
 printf("メニュー:\n");  
 for (i = 0; i < n\_items; i++) {  
 printf("%s\n", menu\_items[i]);  
 }  
   
 return 0;  
}```  
  
> C99版は[こちら](string\_arrays\_c99.c)を参照してください。  
  
---  
  
# 第8章: 文字列処理  
  
  
## 対応C規格  
- \*\*主要対象:\*\* C90  
- \*\*学習内容:\*\* 文字列の基本、文字列操作関数、文字列配列の詳細操作、文字列の比較・検索・変換  
  
## 学習目標  
  
この章を完了すると、以下のことができるようになります：  
  
- 文字列の基本概念を完全に理解する  
- 標準ライブラリの文字列操作関数を使いこなせる  
- 文字列配列の詳細な操作ができる  
- 次元文字配列と文字列ポインタ配列を使い分けられる  
- 実践的な文字列処理プログラムを作成できる  
  
## 概要と詳細  
  
### 文字列処理とは？  
  
文字列処理は、プログラミングで最も頻繁に行う作業の一つです。名前、メッセージ、ファイル名など、私たちが扱うデータの多くは文字列です。  
  
#### 日常生活での文字列処理  
  
身近な例を考えてみましょう：  
  
1. \*\*メールアドレスの検証\*\*  
 - "@"が含まれているか？  
 - 適切な形式か？  
  
2. \*\*パスワードのチェック\*\*   
 - 8文字以上か？  
 - 大文字・小文字・数字が含まれているか？  
  
3. \*\*名前の整形\*\*  
 - 姓と名を分ける  
 - 大文字・小文字を整える  
  
これらはすべて文字列処理の例です！  
  
### 文字列の基本概念   
  
C言語では文字列は文字の配列として表現され、null文字（'\0'）で終端されます。  
  
#### 文字列の表現方法  
  
```c  
#include <stdio.h>  
  
int main(void)  
{  
  
 /\* 方法: 文字配列として宣言 \*/  
 char str[] = 'H', 'e', 'l', 'l', 'o', '';  
   
 /\* 方法: 文字列リテラルで初期化 \*/  
 char str[] = "Hello";  
   
 /\* 方法: サイズを自動決定 \*/  
 char str[] = "Hello";  
   
 /\* 方法: 文字列ポインタ \*/  
 char \*str = "Hello";  
   
 printf("str: %s\n", str);  
 printf("str: %s\n", str);  
 printf("str: %s\n", str);  
 printf("str: %s\n", str);  
   
 return ;

#### 文字列の終端文字について

文字列の最後には必ず「\0」（ヌル文字）が必要です。これを忘れると大変なことになります！

/\* 間違い：終端文字なし \*/  
char bad[] = {'H', 'e', 'l', 'l', 'o'}; /\* 危険！ \*/  
  
/\* 正しい：終端文字あり \*/  
char good[] = {'H', 'e', 'l', 'l', 'o', '\0'}; /\* 安全 \*/

#### 文字列の長さとサイズ

初心者がよく混乱するのが「文字列の長さ」と「配列のサイズ」の違いです：

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 char str[] = "Hello";  
   
 printf("文字列: "%s"n", str);  
 printf("strlen(str): %lu\n", strlen(str)); /\* (文字数) \*/  
 printf("sizeof(str): %lu\n", sizeof(str)); /\* (配列サイズ) \*/  
   
 return ;

### 標準文字列操作関数

C言語には便利な文字列操作関数が用意されています。これらを使いこなすことで、文字列処理が楽になります。

#### 重要な注意点

文字列関数を使うときは、必ず #include <string.h> を忘れずに！

#### strcpy() - 文字列のコピー

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 char sorce[] = "Hello, World!";  
 char destination[];  
   
 /\* 文字列のコピー \*/  
 strcpy(destination, sorce);  
 printf("destination: %s\n", destination);  
   
 /\* 部分的なコピー \*/  
 struncpy(destination, sorce, );  
 destination[] = ''; /\* nll終端を明示的に追加 \*/  
 printf("partial copy: %s\n", destination); /\* "Hello" \*/  
   
 return ;

#### strcat() - 文字列の連結

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 char str[] = "Hello";  
 char str[] = ", World!";  
 char str[] = " How are yo?";  
   
 /\* 文字列の連結 \*/  
 strcat(str, str);  
 printf("After strcat: %s\n", str); /\* "Hello, World!" \*/  
   
 /\* 部分的な連結 \*/  
 struncat(str, str, );  
 printf("After struncat: %s\n", str); /\* "Hello, World! How" \*/  
   
 return ;

#### strcmp() - 文字列の比較

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 char str[] = "Apple";  
 char str[] = "anana";  
 char str[] = "Apple";  
 int result;  
   
 /\* 文字列の比較 \*/  
 result = strcmp(str, str);  
 if (result < )   
 printf(""%s" < "%s"n", str, str);  
 else if (result > )   
 printf(""%s" > "%s"n", str, str);  
 else   
 printf(""%s" == "%s"n", str, str);  
   
   
 /\* 等価性の確認 \*/  
 if (strcmp(str, str) == )   
 printf(""%s" と "%s" は同じですn", str, str);  
   
   
 /\* 部分比較 \*/  
 if (struncmp(str, str, ) != )   
 printf("最初の文字が異なりますn");  
   
   
 return ;

#### 文字列検索関数

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 char text[] = "Hello, World! Welcome to C programming.";  
 char \*fond;  
   
 /\* 文字の検索 \*/  
 fond = strchr(text, 'W');  
 if (fond != NULL)   
 printf("'W' fond at position: %ldn", fond - text);  
   
   
 /\* 文字列の検索 \*/  
 fond = strstr(text, "World");  
 if (fond != NULL)   
 printf(""World" fond at position: %ldn", fond - text);  
 printf("ond: %s\n", fond);  
   
   
 /\* 最後の文字を検索 \*/  
 fond = strrchr(text, 'o');  
 if (fond != NULL)   
 printf("Last 'o' at position: %ldn", fond - text);  
   
   
 return ;

### 文字列配列の詳細操作

複数の文字列を管理する方法を詳しく見ていきましょう。これは実際のプログラムでよく使う重要な技術です。

#### 次元文字配列での文字列配列

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_STRINGS   
#define MAX\_LNGTH   
  
int main(void)  
{  
  
 /\* 次元文字配列による文字列配列 \*/  
 char languages[MAX\_STRINGS][MAX\_LNGTH];  
 int count = ;  
 int i;  
   
 /\* 文字列配列への代入 \*/  
 strcpy(languages[count++], "C");  
 strcpy(languages[count++], "Python");  
 strcpy(languages[count++], "Java");  
 strcpy(languages[count++], "JavaScript");  
 strcpy(languages[count++], "C++");  
   
 /\* 表示 \*/  
 printf("プログラミング言語一覧:n");  
 for (i = ; i < count; i++)   
 printf("%d. %s\n", i + , languages[i]);  
   
   
 /\* 文字列の変更 \*/  
 strcpy(languages[], "C言語");  
 printf("n変更後の番目: %s\n", languages[]);  
   
 return ;

#### 文字列ポインタ配列での文字列配列

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 /\* 文字列ポインタ配列 \*/  
 char \*frits[] =   
 "Apple",  
 "anana",   
 "Cherry",  
 "Date",  
 "lderberry"  
 ;  
 int count = sizeof(frits) / sizeof(frits[]);  
 int i;  
   
 /\* 表示 \*/  
 printf("果物一覧:n");  
 for (i = ; i < count; i++)   
 printf("%d. %s (長さ: %l)n", i + , frits[i], strlen(frits[i]));  
   
   
 /\* ポインタの変更（文字列リテラルの置き換え） \*/  
 frits[] = "リンゴ";  
 printf("n変更後の番目: %s\n", frits[]);  
   
 return ;

#### 文字列配列の初期化と動的変更

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_ITMS   
#define MAX\_LNGTH   
  
int main(void)  
{  
  
 /\* 動的に変更可能な文字列配列 \*/  
 char men\_items[MAX\_ITMS][MAX\_LNGTH];  
 char \*categories[] = "前菜", "メイン", "デザート", "飲み物";  
 int item\_count = ;  
 int i;  
   
 /\* メニュー項目の追加 \*/  
 strcpy(men\_items[item\_count++], "サラダ");  
 strcpy(men\_items[item\_count++], "ステーキ");  
 strcpy(men\_items[item\_count++], "アイスクリーム");  
 strcpy(men\_items[item\_count++], "コーヒー");  
   
 /\* カテゴリ別表示 \*/  
 printf("=== レストランメニュー ===n");  
 for (i = ; i < item\_count && i < ; i++)   
 printf("[%s] %s\n", categories[i], men\_items[i]);  
   
   
 /\* 項目の変更 \*/  
 strcpy(men\_items[], "ハンバーグ");  
 printf("nメイン料理を変更: %s\n", men\_items[]);  
   
 return ;

### 文字列配列の比較と使い分け

2次元char配列とchar\*配列、どちらを使うべきでしょうか？それぞれの特徴を理解して、適切に使い分けましょう。

#### メモリ使用量の比較

#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
  
 /\* 次元文字配列 \*/  
 char matrix\_strings[][] =   
 "Cat", "Dog", "ird", "ish", "Rabbit"  
 ;  
   
 /\* 文字列ポインタ配列 \*/  
 char \*pointer\_strings[] =   
 "Cat", "Dog", "ird", "ish", "Rabbit"  
 ;  
   
 printf("=== メモリ使用量比較 ===n");  
 printf("次元文字配列: %lu バイトn", sizeof(matrix\_strings));  
 printf("文字列ポインタ配列: %lu バイトn", sizeof(pointer\_strings));  
   
 printf("n=== 実際の文字列長 ===n");  
 int i;  
 int total\_chars = ;  
 for (i = ; i < ; i++)   
 int len = strlen(matrix\_strings[i]);  
 printf("%s: %d文字n", matrix\_strings[i], len);  
 total\_chars += len;  
   
 printf("総文字数: %d文字n", total\_chars);  
 printf("未使用領域: %lu バイトn", sizeof(matrix\_strings) - total\_chars - );  
   
 return ;

#### 動的な文字列配列の管理

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_STUDENTS   
#define MAX\_NAM\_LNGTH   
  
typedef strct   
 char name[MAX\_NAM\_LNGTH];  
 int age;  
 float gpa;  
 Stdent;  
  
int main(void)  
{  
  
 Stdent stdents[MAX\_STUDENTS];  
 char \*stats\_messages[] =   
 "優秀", "良好", "普通", "要努力"  
 ;  
 int stdent\_count = ;  
 int i;  
   
 /\* 学生データの追加 \*/  
 strcpy(stdents[stdent\_count].name, "田中太郎");  
 stdents[stdent\_count].age = ;  
 stdents[stdent\_count].gpa = .;  
 stdent\_count++;  
   
 strcpy(stdents[stdent\_count].name, "佐藤花子");  
 stdents[stdent\_count].age = 9;  
 stdents[stdent\_count].gpa = .;  
 stdent\_count++;  
   
 strcpy(stdents[stdent\_count].name, "鈴木一郎");  
 stdents[stdent\_count].age = ;  
 stdents[stdent\_count].gpa = .;  
 stdent\_count++;  
   
 /\* 学生情報の表示 \*/  
 printf("=== 学生一覧 ===n");  
 for (i = ; i < stdent\_count; i++)   
 char \*stats;  
 if (stdents[i].gpa >= .) stats = stats\_messages[];  
 else if (stdents[i].gpa >= .) stats = stats\_messages[];  
 else if (stdents[i].gpa >= .) stats = stats\_messages[];  
 else stats = stats\_messages[];  
   
 printf("%s (%d歳) - GPA: %.f [%s]n",   
 stdents[i].name, stdents[i].age, stdents[i].gpa, stats);  
   
   
 return ;

### 文字列配列のソートと検索

文字列の配列を整理したり、特定の文字列を探したりする方法を学びましょう。

#### 文字列配列のバブルソート

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_WORDS   
#define MAX\_LNGTH   
  
int main(void)  
{  
  
 char words[MAX\_WORDS][MAX\_LNGTH] =   
 "Zebra", "Apple", "Monkey", "anana",  
 "Cat", "Dog", "lephant", "ish"  
 ;  
 char temp[MAX\_LNGTH];  
 int i, j;  
   
 printf("ソート前:n");  
 for (i = ; i < MAX\_WORDS; i++)   
 printf("%s ", words[i]);  
   
 printf("nn");  
   
 /\* バブルソート \*/  
 for (i = ; i < MAX\_WORDS - ; i++)   
 for (j = ; j < MAX\_WORDS - - i; j++)   
 if (strcmp(words[j], words[j + ]) > )   
 strcpy(temp, words[j]);  
 strcpy(words[j], words[j + ]);  
 strcpy(words[j + ], temp);  
   
   
   
   
 printf("ソート後:n");  
 for (i = ; i < MAX\_WORDS; i++)   
 printf("%s ", words[i]);  
   
 printf("n");  
   
 return ;

#### 文字列配列での検索

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_CITIS   
#define MAX\_LNGTH   
  
int search\_city(char cities[][MAX\_LNGTH], int count, char \*target)  
{  
  
 int i;  
 for (i = ; i < count; i++)   
 if (strcmp(cities[i], target) == )   
 return i; /\* 見2つかった場合のインデックス \*/  
   
   
 return -; /\* 見2つからない場合 \*/  
  
  
int main(void)  
{  
  
 char cities[MAX\_CITIS][MAX\_LNGTH] =   
 "Tokyo", "Osaka", "Kyoto", "Nagoya", "Sapporo",  
 "koka", "Kobe", "Sendai", "Hiroshima", "Yokohama"  
 ;  
 char search\_target[] = "Kyoto";  
 int result;  
 int i;  
   
 printf("都市一覧:n");  
 for (i = ; i < MAX\_CITIS; i++)   
 printf("%d. %s\n", i + , cities[i]);  
   
   
 /\* 検索実行 \*/  
 result = search\_city(cities, MAX\_CITIS, search\_target);  
 if (result != -)   
 printf("n"%s" は %d番目にあります。n", search\_target, result + );  
 else   
 printf("n"%s" は見2つかりませんでした。n", search\_target);  
   
   
 return ;

### 実践的な文字列処理

実際のプログラムでよく使う文字列処理のテクニックを紹介します。

#### 文字列の分割（トークン化）

CSVファイルの処理など、カンマで区切られたデータを扱うときに便利です：

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_TOKNS   
#define MAX\_LNGTH   
  
int main(void)  
{  
  
 char inpt[] = "apple,banana,cherry,date,elderberry";  
 char tokens[MAX\_TOKNS][MAX\_LNGTH];  
 char temp[];  
 char \*token;  
 int count = ;  
 int i;  
   
 /\* 入力文字列をコピー（strtokは元の文字列を変更するため） \*/  
 strcpy(temp, inpt);  
   
 /\* カンマで分割 \*/  
 token = strtok(temp, ",");  
 while (token != NULL && count < MAX\_TOKNS)   
 strcpy(tokens[count], token);  
 count++;  
 token = strtok(NULL, ",");  
   
   
 printf("分割結果:n");  
 for (i = ; i < count; i++)   
 printf("%d: %s\n", i + , tokens[i]);  
   
   
 return ;

#### 文字列の変換（大文字・小文字）

#include <stdio.h>  
#include <string.h>  
#include <ctype.h>  
  
#define MAX\_STRINGS   
#define MAX\_LNGTH   
  
void to\_uuppercase(char \*str)  
{  
  
 int i;  
 for (i = ; str[i] != ''; i++)   
 str[i] = topper(str[i]);  
   
  
  
void to\_lowercase(char \*str)  
{  
  
 int i;  
 for (i = ; str[i] != ''; i++)   
 str[i] = tolower(str[i]);  
   
  
  
int main(void)  
{  
  
 char original[MAX\_STRINGS][MAX\_LNGTH] =   
 "Hello World",  
 "Programming Langage",  
 "C Langage Ttutorial",  
 "String Processing",  
 "Array Maniplation"  
 ;  
 char uuppercase[MAX\_STRINGS][MAX\_LNGTH];  
 char lowercase[MAX\_STRINGS][MAX\_LNGTH];  
 int i;  
   
 /\* 文字列をコピーして変換 \*/  
 for (i = ; i < MAX\_STRINGS; i++)   
 strcpy(uuppercase[i], original[i]);  
 strcpy(lowercase[i], original[i]);  
   
 to\_uuppercase(uuppercase[i]);  
 to\_lowercase(lowercase[i]);  
   
   
 /\* 結果の表示 \*/  
 printf("元の文字列 -> 大文字 -> 小文字n");  
 printf("=====================================n");  
 for (i = ; i < MAX\_STRINGS; i++)   
 printf("%-s -> %-s -> %s\n",   
 original[i], uuppercase[i], lowercase[i]);  
   
   
 return ;

#### 文字列配列を使った簡易データベース

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_RCORDS   
#define MAX\_ILD\_LNGTH   
  
typedef strct   
 char name[MAX\_ILD\_LNGTH];  
 char department[MAX\_ILD\_LNGTH];  
 char position[MAX\_ILD\_LNGTH];  
 int salary;  
 mployee;  
  
int main(void)  
{  
  
 mployee employees[MAX\_RCORDS];  
 char \*departments[] = "営業", "開発", "人事", "経理";  
 char \*positions[] = "部長", "課長", "主任", "一般";  
 int employee\_count = ;  
 int i, j;  
   
 /\* サンプルデータの追加 \*/  
 strcpy(employees[employee\_count].name, "田中太郎");  
 strcpy(employees[employee\_count].department, departments[]); /\* 開発 \*/  
 strcpy(employees[employee\_count].position, positions[]); /\* 課長 \*/  
 employees[employee\_count].salary = ;  
 employee\_count++;  
   
 strcpy(employees[employee\_count].name, "佐藤花子");  
 strcpy(employees[employee\_count].department, departments[]); /\* 営業 \*/  
 strcpy(employees[employee\_count].position, positions[]); /\* 部長 \*/  
 employees[employee\_count].salary = ;  
 employee\_count++;  
   
 strcpy(employees[employee\_count].name, "鈴木一郎");  
 strcpy(employees[employee\_count].department, departments[]); /\* 開発 \*/  
 strcpy(employees[employee\_count].position, positions[]); /\* 一般 \*/  
 employees[employee\_count].salary = ;  
 employee\_count++;  
   
 /\* 全従業員の表示 \*/  
 printf("=== 従業員一覧 ===n");  
 printf("%-s %-s %-s %s\n", "名前", "部署", "役職", "給与");  
 printf("--------------------------------------------------n");  
 for (i = ; i < employee\_count; i++)   
 printf("%-s %-s %-s %d円n",  
 employees[i].name,  
 employees[i].department,  
 employees[i].position,  
 employees[i].salary);  
   
   
 /\* 部署別集計 \*/  
 printf("n=== 部署別従業員数 ===n");  
 for (i = ; i < ; i++)   
 int count = ;  
 for (j = ; j < employee\_count; j++)   
 if (strcmp(employees[j].department, departments[i]) == )   
 count++;  
   
   
 printf("%s: %d人n", departments[i], count);  
   
   
 return ;

## サンプルコード

### 基本的な文字列操作

プログラムファイル: examples/string\_basic.c

標準ライブラリの文字列操作関数の使用方法を学習します。

### 文字列配列の実践例

プログラムファイル: examples/string\_array\_operations.c

次元文字配列と文字列ポインタ配列の使い分けを学習します。

### 文字列処理の応用

プログラムファイル: examples/advanced\_string\_processing.c

実践的な文字列処理テクニックを学習します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic string\_basic.c -o string\_basic  
  
# 実行  
./string\_basic

## 演習課題

### 基礎問題

. **文字列の基本操作** - 2つの文字列を入力して連結・比較・長さ計算を実行うプログラムを作成してください

. **文字列配列の管理** - 学生名を格納する文字列配列を作成し、追加・削除・表示機能を実装してください

. **文字列の検索** - 文字列配列から特定の文字列を検索するプログラムを作成してください

### 応用問題

. **単語カウンタ** - 文章を入力して、各単語の出現1回数をカウントするプログラムを作成してください

. **文字列ソートプログラム** - 複数の文字列を入力してアルファベット順にソートするプログラムを作成してください

. **簡易テキストエディタ** - 文字列配列を使って行単位でテキストを管理するプログラムを作成してください

### 発展問題

. **文字列データベース** - 商品情報（名前、価格、カテゴリ）を文字列配列で管理するシステムを作成してください

. **文字列パーサー** - CSV形式の文字列を解析して構造化データに変換するプログラムを作成してください

. **文字列暗号化** - 簡単な暗号化・復号化機能を持2つプログラムを作成してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make string\_basic  
  
# 実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* 文字列操作はnll終端が前提
* 可変長文字列配列は使用不可
* Unicode対応は限定的

### C99以降の拡張

* snprintf()などの安全な文字列操作関数
* 可変長配列による動的な文字列配列
* より豊富な文字分類関数

## よくある間違い

文字列処理で初心者がよく陥るミスとその対策を紹介します。

### 1. null終端の忘れ

/\* NG: nll終端なし \*/  
char str[];  
struncpy(str, "Hello", ); /\* nll終端なし \*/  
printf("%s\n", str); /\* 未定義動作 \*/  
  
/\* OK: 明示的にnll終端 \*/  
char str[];  
struncpy(str, "Hello", );  
str[] = '';  
printf("%s\n", str);

### 2. バッファオーバーフロー

最も危険なミスの一つです。十分な領域を確保しましょう：

/\* NG: バッファサイズ不足 \*/  
char small[5];  
strcpy(small, "Hello, World!"); /\* バッファオーバーフロー！ \*/  
  
/\* OK: 適切なサイズ \*/  
char large[20];  
strcpy(large, "Hello, World!");

**覚え方：** 文字列の長さ + 1（null終端分）の領域が必要！

### 3. 文字列比較の間違い

C言語では、== で文字列を比較できません！

/\* NG: ポインタの比較 \*/  
char \*str1 = "Hello";  
char \*str2 = "Hello";  
if (str1 == str2) { /\* 間違い！アドレスを比較している \*/  
}  
  
/\* OK: 文字列内容の比較 \*/  
if (strcmp(str1, str2) == 0) { /\* 正しい！文字列の内容を比較 \*/  
 printf("文字列は同じです\n");  
}

**覚え方：** 文字列の比較は必ず strcmp() を使う！

## 次の章へ

文字列処理を理解したら、[関数](../functions/README.md) に進んでください。

## 参考資料

* [C言語文字列リファレンス](https://ja.cppreference.com/w/c/string/byte)
* [文字分類関数](https://ja.cppreference.com/w/c/string/ctype)
* [文字列操作のベストプラクティス](https://ja.cppreference.com/w/c/string)

## サンプルコード

### advanced\_string\_processing.c

/\*  
 \* 文字列処理の応用  
 \* ファイル名: advanced\_string\_processing.c  
 \* 説明: 実践的な文字列処理テクニックを学習  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
#include <ctype.h>  
  
#define MAX\_TOKENS 10  
#define MAX\_LENGTH 100  
#define MAX\_STRINGS 5  
  
/\* 文字列を大文字に変換 \*/  
void to\_uppercase(char \*str)  
{  
 int i;  
   
 if (str == NULL) {  
 return;  
 }  
   
 for (i = 0; str[i] != '\0'; i++) {  
 str[i] = toupper((unsigned char)str[i]);  
 }  
}  
  
/\* 文字列を小文字に変換 \*/  
void to\_lowercase(char \*str)  
{  
 int i;  
   
 if (str == NULL) {  
 return;  
 }  
   
 for (i = 0; str[i] != '\0'; i++) {  
 str[i] = tolower((unsigned char)str[i]);  
 }  
}  
  
/\* 文字列配列のバブルソート \*/  
void bubble\_sort\_strings(char arr[][MAX\_LENGTH], int n)  
{  
 int i, j;  
 char temp[MAX\_LENGTH];  
   
 for (i = 0; i < n - 1; i++) {  
 for (j = 0; j < n - 1 - i; j++) {  
 if (strcmp(arr[j], arr[j + 1]) > 0) {  
 strcpy(temp, arr[j]);  
 strcpy(arr[j], arr[j + 1]);  
 strcpy(arr[j + 1], temp);  
 }  
 }  
 }  
}  
  
/\* 文字列配列での検索 \*/  
int search\_string(char arr[][MAX\_LENGTH], int count, char \*target)  
{  
 int i;  
   
 for (i = 0; i < count; i++) {  
 if (strcmp(arr[i], target) == 0) {  
 return i; /\* 見つかった場合のインデックス \*/  
 }  
 }  
 return -1; /\* 見つからない場合 \*/  
}  
  
int main(void)  
{  
 /\* 文字列の分割（トークン化） \*/  
 printf("=== 文字列の分割（トークン化） ===\n");  
 char input[] = "apple,banana,cherry,date,elderberry";  
 char tokens[MAX\_TOKENS][MAX\_LENGTH];  
 char temp[100];  
 char \*token;  
 int token\_count = 0;  
 int i;  
   
 /\* 入力文字列をコピー（strtokは元の文字列を変更するため） \*/  
 strcpy(temp, input);  
   
 /\* カンマで分割 \*/  
 token = strtok(temp, ",");  
 while (token != NULL && token\_count < MAX\_TOKENS) {  
 strcpy(tokens[token\_count], token);  
 token\_count++;  
 token = strtok(NULL, ",");  
 }  
   
 printf("分割結果:\n");  
 for (i = 0; i < token\_count; i++) {  
 printf("%d: %s\n", i + 1, tokens[i]);  
 }  
   
 /\* 文字列の変換（大文字・小文字） \*/  
 printf("\n=== 文字列の変換（大文字・小文字） ===\n");  
 char original[MAX\_STRINGS][MAX\_LENGTH] = {  
 "Hello World",  
 "Programming Language",  
 "C Language Tutorial",  
 "String Processing",  
 "Array Manipulation"  
 };  
 char uppercase\_copy[MAX\_STRINGS][MAX\_LENGTH];  
 char lowercase\_copy[MAX\_STRINGS][MAX\_LENGTH];  
   
 /\* 文字列をコピーして変換 \*/  
 for (i = 0; i < MAX\_STRINGS; i++) {  
 strcpy(uppercase\_copy[i], original[i]);  
 strcpy(lowercase\_copy[i], original[i]);  
   
 to\_uppercase(uppercase\_copy[i]);  
 to\_lowercase(lowercase\_copy[i]);  
 }  
   
 /\* 結果の表示 \*/  
 printf("元の文字列 -> 大文字 -> 小文字\n");  
 printf("=====================================\n");  
 for (i = 0; i < MAX\_STRINGS; i++) {  
 printf("%-20s -> %-20s -> %s\n",   
 original[i], uppercase\_copy[i], lowercase\_copy[i]);  
 }  
   
 /\* 文字列配列のソートと検索 \*/  
 printf("\n=== 文字列配列のソート ===\n");  
 char words[8][MAX\_LENGTH] = {  
 "Zebra", "Apple", "Monkey", "Banana",  
 "Cat", "Dog", "Elephant", "Fish"  
 };  
   
 printf("ソート前:\n");  
 for (i = 0; i < 8; i++) {  
 printf("%s ", words[i]);  
 }  
 printf("\n\n");  
   
 /\* バブルソート \*/  
 bubble\_sort\_strings(words, 8);  
   
 printf("ソート後:\n");  
 for (i = 0; i < 8; i++) {  
 printf("%s ", words[i]);  
 }  
 printf("\n");  
   
 /\* 検索のテスト \*/  
 printf("\n=== 文字列の検索 ===\n");  
 char cities[10][MAX\_LENGTH] = {  
 "Tokyo", "Osaka", "Kyoto", "Nagoya", "Sapporo",  
 "Fukuoka", "Kobe", "Sendai", "Hiroshima", "Yokohama"  
 };  
 char search\_target[] = "Kyoto";  
 int result;  
   
 printf("都市一覧:\n");  
 for (i = 0; i < 10; i++) {  
 printf("%d. %s\n", i + 1, cities[i]);  
 }  
   
 /\* 検索実行 \*/  
 result = search\_string(cities, 10, search\_target);  
 if (result != -1) {  
 printf("\n\"%s\" は %d番目にあります。\n", search\_target, result + 1);  
 } else {  
 printf("\n\"%s\" は見つかりませんでした。\n", search\_target);  
 }  
   
 /\* 簡易データベースの例 \*/  
 printf("\n=== 簡易従業員データベース ===\n");  
 char employee\_names[3][MAX\_LENGTH] = {  
 "田中太郎", "佐藤花子", "鈴木一郎"  
 };  
 char departments[3][MAX\_LENGTH] = {  
 "開発", "営業", "開発"  
 };  
 char positions[3][MAX\_LENGTH] = {  
 "課長", "部長", "一般"  
 };  
 int salaries[3] = {800000, 1200000, 500000};  
   
 /\* 全従業員の表示 \*/  
 printf("%-12s %-8s %-8s %s\n", "名前", "部署", "役職", "給与");  
 printf("--------------------------------------------------\n");  
 for (i = 0; i < 3; i++) {  
 printf("%-12s %-8s %-8s %d円\n",  
 employee\_names[i],  
 departments[i],  
 positions[i],  
 salaries[i]);  
 }  
   
 /\* 部署別集計 \*/  
 printf("\n=== 部署別従業員数 ===\n");  
 char unique\_depts[4][MAX\_LENGTH] = {"営業", "開発", "人事", "経理"};  
 int dept\_count;  
   
 for (i = 0; i < 4; i++) {  
 int count = 0;  
 int j;  
   
 for (j = 0; j < 3; j++) {  
 if (strcmp(departments[j], unique\_depts[i]) == 0) {  
 count++;  
 }  
 }  
 printf("%s: %d人\n", unique\_depts[i], count);  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 文字列の分割:  
 - strtok()関数による分割  
 - 元の文字列が変更されることに注意  
 - 区切り文字の指定方法  
  
2. 文字列変換:  
 - toupper()/tolower()による大文字小文字変換  
 - ctype.hライブラリの活用  
 - 文字列全体の変換処理  
  
3. 文字列ソート:  
 - バブルソートアルゴリズムの実装  
 - strcmp()による文字列比較  
 - 配列要素の交換処理  
  
4. 実用的な応用:  
 - データベース的な処理  
 - 集計・分析機能  
 - 検索・フィルタリング  
  
注意点:  
- strtok()は元の文字列を変更するため注意  
- 文字のキャストにはunsigned charを使用  
- バッファサイズの管理  
- NULL文字の適切な処理  
\*/

C99版は[こちら](advanced_string_processing_c99.c)を参照してください。

### string\_array\_operations.c

/\*  
 \* 文字列配列の実践例  
 \* ファイル名: string\_array\_operations.c  
 \* 説明: 二次元文字配列と文字列ポインタ配列の使い分けを学習  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
  
#define MAX\_STRINGS 5  
#define MAX\_LENGTH 20  
  
int main(void)  
{  
 /\* 二次元文字配列での文字列配列 \*/  
 printf("=== 二次元文字配列による文字列配列 ===\n");  
 char languages[MAX\_STRINGS][MAX\_LENGTH];  
 int count = 0;  
 int i;  
   
 /\* 文字列配列への代入 \*/  
 strcpy(languages[count++], "C");  
 strcpy(languages[count++], "Python");  
 strcpy(languages[count++], "Java");  
 strcpy(languages[count++], "JavaScript");  
 strcpy(languages[count++], "C++");  
   
 /\* 表示 \*/  
 printf("プログラミング言語一覧:\n");  
 for (i = 0; i < count; i++) {  
 printf("%d. %s\n", i + 1, languages[i]);  
 }  
   
 /\* 文字列の変更 \*/  
 strcpy(languages[0], "C言語");  
 printf("\n変更後の1番目: %s\n", languages[0]);  
   
 /\* 文字列ポインタ配列での文字列配列 \*/  
 printf("\n=== 文字列ポインタ配列 ===\n");  
 char \*fruits[] = {  
 "Apple",  
 "Banana",   
 "Cherry",  
 "Date",  
 "Elderberry"  
 };  
 int fruit\_count = sizeof(fruits) / sizeof(fruits[0]);  
   
 /\* 表示 \*/  
 printf("果物一覧:\n");  
 for (i = 0; i < fruit\_count; i++) {  
 printf("%d. %s (長さ: %lu)\n", i + 1, fruits[i],   
 (unsigned long)strlen(fruits[i]));  
 }  
   
 /\* ポインタの変更（文字列リテラルの置き換え） \*/  
 fruits[0] = "リンゴ";  
 printf("\n変更後の1番目: %s\n", fruits[0]);  
   
 /\* 文字列配列の初期化と動的変更 \*/  
 printf("\n=== 動的に変更可能な文字列配列 ===\n");  
 char menu\_items[4][MAX\_LENGTH];  
 char \*categories[] = {"前菜", "メイン", "デザート", "飲み物"};  
 int item\_count = 0;  
   
 /\* メニュー項目の追加 \*/  
 strcpy(menu\_items[item\_count++], "サラダ");  
 strcpy(menu\_items[item\_count++], "ステーキ");  
 strcpy(menu\_items[item\_count++], "アイスクリーム");  
 strcpy(menu\_items[item\_count++], "コーヒー");  
   
 /\* カテゴリ別表示 \*/  
 printf("=== レストランメニュー ===\n");  
 for (i = 0; i < item\_count && i < 4; i++) {  
 printf("[%s] %s\n", categories[i], menu\_items[i]);  
 }  
   
 /\* 項目の変更 \*/  
 strcpy(menu\_items[1], "ハンバーグ");  
 printf("\nメイン料理を変更: %s\n", menu\_items[1]);  
   
 /\* メモリ使用量の比較 \*/  
 printf("\n=== メモリ使用量比較 ===\n");  
 char matrix\_strings[5][10] = {  
 "Cat", "Dog", "Bird", "Fish", "Rabbit"  
 };  
   
 char \*pointer\_strings[] = {  
 "Cat", "Dog", "Bird", "Fish", "Rabbit"  
 };  
   
 printf("二次元文字配列: %lu バイト\n",   
 (unsigned long)sizeof(matrix\_strings));  
 printf("文字列ポインタ配列: %lu バイト\n",   
 (unsigned long)sizeof(pointer\_strings));  
   
 printf("\n=== 実際の文字列長 ===\n");  
 int total\_chars = 0;  
 for (i = 0; i < 5; i++) {  
 int len = strlen(matrix\_strings[i]);  
 printf("%s: %d文字\n", matrix\_strings[i], len);  
 total\_chars += len;  
 }  
 printf("総文字数: %d文字\n", total\_chars);  
 printf("未使用領域: %lu バイト\n",   
 sizeof(matrix\_strings) - total\_chars - 5);  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 二次元文字配列:  
 - 固定サイズのメモリ確保  
 - 文字列の変更が可能  
 - メモリ使用効率は低い場合がある  
   
2. 文字列ポインタ配列:  
 - ポインタのみを格納  
 - 文字列リテラルは変更不可  
 - メモリ使用効率が良い  
   
3. 使い分けの基準:  
 - 文字列を変更する必要がある: 二次元配列  
 - 文字列が固定: ポインタ配列  
 - メモリ効率を重視: ポインタ配列  
   
4. 実用的な応用:  
 - メニュー管理システム  
 - 設定項目の管理  
 - データの分類・整理  
  
注意点:  
- 文字列リテラルの変更は未定義動作  
- バッファサイズの確認  
- 配列の境界チェック  
\*/

C99版は[こちら](string_array_operations_c99.c)を参照してください。

### string\_basic.c

/\*  
 \* 文字列操作の基本例  
 \* ファイル名: string\_basic.c  
 \* 説明: 標準ライブラリの文字列操作関数の使用方法を学習  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
  
int main(void)  
{  
 /\* 文字列の表現方法 \*/  
 char str1[] = {'H', 'e', 'l', 'l', 'o', '\0'};  
 char str2[] = "Hello";  
 char str3[10] = "Hello";  
 char \*str4 = "Hello";  
   
 printf("=== 文字列の表現方法 ===\n");  
 printf("str1: %s\n", str1);  
 printf("str2: %s\n", str2);  
 printf("str3: %s\n", str3);  
 printf("str4: %s\n", str4);  
   
 /\* 文字列の長さとサイズ \*/  
 printf("\n=== 文字列の長さとサイズ ===\n");  
 printf("文字列: \"%s\"\n", str2);  
 printf("strlen(str2): %lu文字\n", (unsigned long)strlen(str2));  
 printf("sizeof(str2): %lu バイト\n", (unsigned long)sizeof(str2));  
   
 /\* strcpy() - 文字列のコピー \*/  
 printf("\n=== 文字列のコピー ===\n");  
 char source[] = "Hello, World!";  
 char destination[50];  
   
 strcpy(destination, source);  
 printf("destination: %s\n", destination);  
   
 /\* 部分的なコピー \*/  
 strncpy(destination, source, 5);  
 destination[5] = '\0'; /\* null終端を明示的に追加 \*/  
 printf("partial copy: %s\n", destination); /\* "Hello" \*/  
   
 /\* strcat() - 文字列の連結 \*/  
 printf("\n=== 文字列の連結 ===\n");  
 char str\_a[50] = "Hello";  
 char str\_b[] = ", World!";  
 char str\_c[] = " How are you?";  
   
 strcat(str\_a, str\_b);  
 printf("After strcat: %s\n", str\_a); /\* "Hello, World!" \*/  
   
 /\* 部分的な連結 \*/  
 strncat(str\_a, str\_c, 4);  
 printf("After strncat: %s\n", str\_a); /\* "Hello, World! How" \*/  
   
 /\* strcmp() - 文字列の比較 \*/  
 printf("\n=== 文字列の比較 ===\n");  
 char apple[] = "Apple";  
 char banana[] = "Banana";  
 char apple2[] = "Apple";  
 int result;  
   
 result = strcmp(apple, banana);  
 if (result < 0) {  
 printf("\"%s\" < \"%s\"\n", apple, banana);  
 } else if (result > 0) {  
 printf("\"%s\" > \"%s\"\n", apple, banana);  
 } else {  
 printf("\"%s\" == \"%s\"\n", apple, banana);  
 }  
   
 /\* 等価性の確認 \*/  
 if (strcmp(apple, apple2) == 0) {  
 printf("\"%s\" と \"%s\" は同じです\n", apple, apple2);  
 }  
   
 /\* 部分比較 \*/  
 if (strncmp(apple, banana, 1) != 0) {  
 printf("最初の文字が異なります\n");  
 }  
   
 /\* 文字列検索関数 \*/  
 printf("\n=== 文字列検索関数 ===\n");  
 char text[] = "Hello, World! Welcome to C programming.";  
 char \*found;  
   
 /\* 文字の検索 \*/  
 found = strchr(text, 'W');  
 if (found != NULL) {  
 printf("'W' found at position: %ld\n", found - text);  
 }  
   
 /\* 文字列の検索 \*/  
 found = strstr(text, "World");  
 if (found != NULL) {  
 printf("\"World\" found at position: %ld\n", found - text);  
 printf("Found: %s\n", found);  
 }  
   
 /\* 最後の文字を検索 \*/  
 found = strrchr(text, 'o');  
 if (found != NULL) {  
 printf("Last 'o' at position: %ld\n", found - text);  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 文字列の表現方法:  
 - 文字配列による初期化  
 - 文字列リテラルによる初期化  
 - ポインタによる文字列参照  
  
2. 標準ライブラリ関数:  
 - strlen(): 文字列の長さを取得  
 - strcpy()/strncpy(): 文字列のコピー  
 - strcat()/strncat(): 文字列の連結  
 - strcmp()/strncmp(): 文字列の比較  
  
3. 文字列検索:  
 - strchr(): 文字の検索  
 - strrchr(): 文字の後方検索  
 - strstr(): 部分文字列の検索  
  
注意点:  
- strncpy()使用時はnull終端文字の確認が必要  
- バッファオーバーフローの防止  
- NULLポインタのチェック  
\*/

C99版は[こちら](string_basic_c99.c)を参照してください。

# 第9章: 関数

## 対応C規格

* **主要対象:** C90
* **学習内容:** 関数の基本、引数と戻り値、関数のスコープ、再帰関数、関数プロトタイプ

## 学習目標

この章を完了すると、以下のことができるようになります：

* 関数の定義と呼び出しができる
* 引数と戻り値を適切に使える
* 関数のスコープを理解する
* 再帰関数を作成できる
* 関数プロトタイプの重要性を理解する

## 概要と詳細

### 関数とは？

関数は、プログラムを「部品」に分けるための仕組みです。大きな問題を小さな部分に分割して解決する「分割統治」の考え方を実現します。

#### 日常生活での関数

関数を理解するために、料理を例に考えてみましょう：

**カレーライスを作る場合：** 1. ご飯を炊く（関数1） 2. 野菜を切る（関数2） 3. 肉を炒める（関数3） 4. カレーを煮込む（関数4） 5. 盛り付ける（関数5）

それぞれの作業を「関数」として独立させることで： - 同じ作業（ご飯を炊く）を他の料理でも使える - 問題があったとき、どの工程か特定しやすい - 複数人で分担できる

### 関数の基本概念

関数は特定の処理をまとめたコードブロックで、プログラムの再利用性と保守性を向上させます。

#### 関数を使う理由

1. **コードの再利用性**: 同じ処理を何度も書く必要がない
2. **プログラムの構造化**: 複雑な問題を小さな部分に分割
3. **保守性の向上**: 修正が必要な箇所を特定しやすい
4. **可読性の向上**: プログラムの意図が明確になる
5. **デバッグの容易さ**: 問題のある部分を特定しやすい

#### 関数の構成要素

戻り値の型 関数名(引数リスト)  
{  
 /\* 関数本体 \*/  
 return 戻り値; /\* 戻り値がある場合 \*/  
}

### 基本的な関数の例

まずは簡単な例から始めましょう：

#include <stdio.h>  
  
/\* 2つの数の合計を計算する関数 \*/  
int add(int a, int b)  
{  
 int result = a + b;  
 return result;  
}  
  
int main(void)  
{  
 int num1 = 10;  
 int num2 = 20;  
 int sum;  
   
 /\* 関数の呼び出し \*/  
 sum = add(num1, num2);  
   
 printf("%d + %d = %d\n", num1, num2, sum);  
   
 return 0;  
}

**このプログラムの流れ：** 1. main関数が実行開始 2. add関数を呼び出す（10と20を渡す） 3. add関数が計算して結果（30）を返す 4. 結果をsumに格納 5. 結果を表示

### 関数の定義と宣言

関数を使うには、「宣言」と「定義」を理解する必要があります。

#### なぜ関数プロトタイプが必要？

C言語では「使う前に宣言する」というルールがあります。関数プロトタイプは「この関数がありますよ」とコンパイラに教える役割があります。

#### 関数プロトタイプ（前方宣言）

#include <stdio.h>  
  
/\* 関数プロトタイプ \*/  
int multiply(int x, int y);  
void print\_result(int value);  
double calculate\_average(int \*array, int size);  
  
int main(void)  
{  
 int a = 5, b = 3;  
 int product;  
   
 product = multiply(a, b);  
 print\_result(product);  
   
 return 0;  
}  
  
/\* 関数の実装 \*/  
int multiply(int x, int y)  
{  
 return x \* y;  
}  
  
void print\_result(int value)  
{  
 printf("結果: %d\n", value);  
}  
  
double calculate\_average(int \*array, int size)  
{  
 int i;  
 int sum = 0;  
   
 if (size <= 0)  
 {  
 return 0.0;  
 }  
   
 for (i = 0; i < size; i++)  
 {  
 sum += array[i];  
 }  
   
 return (double)sum / size;  
}

### 引数の渡し方

関数に値を渡す方法は2つあります。この違いを理解することは非常に重要です！

#### 値渡し（Call by Value）

C言語では、通常の変数を渡すと「値のコピー」が渡されます：

/\* 値渡しの例（間違った交換関数） \*/  
void swap\_wrong(int a, int b)  
{  
 int temp = a;  
 a = b;  
 b = temp;  
 /\* ⚠️ 呼び出し元の変数は変更されない！ \*/  
}  
  
/\* ポインタを使った参照渡し（正しい交換関数） \*/  
void swap\_correct(int \*a, int \*b)  
{  
 int temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
 /\* ✅ 呼び出し元の変数が変更される！ \*/  
}  
  
int main(void)  
{  
 int x = 10, y = 20;  
   
 printf("交換前: x = %d, y = %d\n", x, y);  
   
 swap\_wrong(x, y);  
 printf("値渡し後: x = %d, y = %d\n", x, y);  
   
 swap\_correct(&x, &y);  
 printf("ポインタ渡し後: x = %d, y = %d\n", x, y);  
   
 return 0;  
}

### 様々な関数の種類

関数にはいろいろな種類があります。用途に応じて使い分けましょう。

#### 戻り値のない関数（void関数）

処理だけ行って、結果を返さない関数です：

void print\_header(void)  
{  
 printf("====================\n");  
 printf(" プログラム開始\n");  
 printf("====================\n");  
}  
  
void greet\_user(char \*name)  
{  
 printf("こんにちは、%sさん！\n", name);  
}

#### 配列を扱う関数

/\* 配列の要素数を計算できないため、サイズを別途渡す必要がある \*/  
int array\_sum(int arr[], int size)  
{  
 int i, sum = 0;  
   
 for (i = 0; i < size; i++)  
 {  
 sum += arr[i];  
 }  
   
 return sum;  
}  
  
/\* 配列を初期化する関数 \*/  
void initialize\_array(int arr[], int size, int value)  
{  
 int i;  
   
 for (i = 0; i < size; i++)  
 {  
 arr[i] = value;  
 }  
}

### 再帰関数

再帰関数は自分自身を呼び出す関数です。

#### 再帰を理解する日常例

**ロシア人形（マトリョーシカ）**を想像してください： 1. 一番外側の人形を開ける 2. 中に人形があったら、それも開ける 3. さらに中に人形があったら、それも開ける 4. 最後の人形（これ以上開けない）に到達したら終了

これが再帰の考え方です！

/\* 階乗を計算する再帰関数 \*/  
int factorial(int n)  
{  
 /\* 基底条件（終了条件）\*/  
 if (n <= 1)  
 {  
 return 1; /\* ここで再帰が止まる！ \*/  
 }  
   
 /\* 再帰呼び出し \*/  
 return n \* factorial(n - 1);  
}  
  
/\*   
 factorial(5)の動き：  
 factorial(5) = 5 \* factorial(4)  
 = 5 \* 4 \* factorial(3)  
 = 5 \* 4 \* 3 \* factorial(2)  
 = 5 \* 4 \* 3 \* 2 \* factorial(1)  
 = 5 \* 4 \* 3 \* 2 \* 1  
 = 120  
\*/  
  
/\* フィボナッチ数列を計算する再帰関数 \*/  
int fibonacci(int n)  
{  
 /\* 基底条件 \*/  
 if (n <= 1)  
 {  
 return n;  
 }  
   
 /\* 再帰呼び出し \*/  
 return fibonacci(n - 1) + fibonacci(n - 2);  
}

### 関数のスコープと生存期間

変数には「見える範囲」と「生きている期間」があります。これを理解することで、バグを防げます。

#### ローカル変数とグローバル変数

**家族の例で考えてみましょう：** - **ローカル変数**：自分の部屋にあるもの（他の人は使えない） - **グローバル変数**：リビングにあるもの（家族全員が使える）

#include <stdio.h>  
  
/\* グローバル変数 \*/  
int global\_count = 0;  
  
void increment\_global(void)  
{  
 global\_count++; /\* グローバル変数にアクセス \*/  
}  
  
void local\_example(void)  
{  
 int local\_var = 10; /\* ローカル変数 \*/  
   
 printf("ローカル変数: %d\n", local\_var);  
 /\* 関数を抜けるとlocal\_varは消滅 \*/  
}  
  
/\* static変数の例 \*/  
void counter(void)  
{  
 static int count = 0; /\* static変数は値を保持 \*/  
 count++;  
 printf("呼び出し回数: %d\n", count);  
}  
  
int main(void)  
{  
 int i;  
   
 /\* staticの効果を確認 \*/  
 for (i = 0; i < 3; i++)  
 {  
 counter();  
 }  
   
 return 0;  
}

### 関数ポインタの基礎

関数へのポインタを使うことで、関数を変数のように扱えます。これは少し高度な話題ですが、基本だけ紹介します。

#### 関数ポインタのイメージ

テレビのリモコンを想像してください： - リモコンのボタン = 関数ポインタ - ボタンを押す = 関数を実行 - ボタンの割り当てを変える = 別の関数を指すようにする

#include <stdio.h>  
  
/\* 計算用の関数 \*/  
int add(int a, int b) { return a + b; }  
int subtract(int a, int b) { return a - b; }  
int multiply(int a, int b) { return a \* b; }  
  
int main(void)  
{  
 /\* 関数ポインタの宣言 \*/  
 int (\*operation)(int, int);  
 int x = 10, y = 5;  
   
 /\* 関数ポインタに関数を代入 \*/  
 operation = add;  
 printf("%d + %d = %d\n", x, y, operation(x, y));  
   
 operation = subtract;  
 printf("%d - %d = %d\n", x, y, operation(x, y));  
   
 operation = multiply;  
 printf("%d \* %d = %d\n", x, y, operation(x, y));  
   
 return 0;  
}

## 実例コード

完全な実装例は以下のファイルを参照してください：

### 基本的な関数の使い方

* [function\_basics.c](examples/function_basics.c) - C90準拠版
* [function\_basics\_c99.c](examples/function_basics_c99.c) - C99準拠版

### 高度な関数の使い方

* [advanced\_functions.c](examples/advanced_functions.c) - C90準拠版
* [advanced\_functions\_c99.c](examples/advanced_functions_c99.c) - C99準拠版

## コンパイルと実行

# 基本的な関数の例をコンパイル  
gcc -Wall -Wextra -pedantic -std=c90 examples/function\_basics.c -o function\_basics  
  
# 実行  
./function\_basics  
  
# C99版をコンパイル  
gcc -Wall -Wextra -pedantic -std=c99 examples/function\_basics\_c99.c -o function\_basics\_c99  
  
# 数学関数を使う場合は-lmを追加  
gcc -Wall -Wextra -pedantic examples/advanced\_functions.c -lm -o advanced\_functions

## 演習問題

この章の演習問題は<exercises/README.md>にあります。

## 注意事項

初心者が関数で間違えやすいポイント：

1. **関数プロトタイプ**: main関数より後に定義する関数は、必ずプロトタイプ宣言が必要

* /\* NG: プロトタイプなし \*/  
  int main() {   
   add(1, 2); /\* エラー！ \*/  
  }  
  int add(int a, int b) { return a + b; }

1. **配列の扱い**: 配列を関数に渡すときは、サイズ情報も一緒に渡す

* /\* NG: サイズが分からない \*/  
  void print\_array(int arr[]) { /\* 要素数不明 \*/ }  
    
  /\* OK: サイズも渡す \*/  
  void print\_array(int arr[], int size) { /\* OK \*/ }

1. **再帰の深さ**: 再帰関数は深くなりすぎるとスタックオーバーフローを起こす
2. **グローバル変数**: できるだけ使用を避け、必要な場合は慎重に使用する
3. **static変数**: 関数内でstatic変数を使うと、値が保持される

## 次のステップ

関数の基本を理解したら、以下のトピックに進みましょう：

1. より複雑な関数の設計パターン
2. 関数ポインタと高階関数
3. 可変長引数関数（stdarg.h）
4. インライン関数（C99以降）
5. ライブラリ関数の作成 ## サンプルコード

### advanced\_functions.c

/\*  
 \* 高度な関数の使い方  
 \*   
 \* このプログラムは、再帰関数、ローカル変数とグローバル変数、  
 \* 配列を引数に取る関数の使い方を示します。  
 \*/  
#include <stdio.h>  
  
/\* グローバル変数 \*/  
int global\_counter = 0;  
  
/\* 関数プロトタイプ \*/  
int factorial(int n);  
int fibonacci(int n);  
void display\_array(int arr[], int size);  
int sum\_array(int arr[], int size);  
double average\_array(int arr[], int size);  
void increment\_counter(void);  
int local\_vs\_global\_demo(int param);  
  
/\* 階乗を計算する再帰関数 \*/  
int factorial(int n)  
{  
 if (n <= 1) {  
 return 1; /\* 基底条件 \*/  
 }  
 return n \* factorial(n - 1); /\* 再帰呼び出し \*/  
}  
  
/\* フィボナッチ数列を計算する再帰関数 \*/  
int fibonacci(int n)  
{  
 if (n <= 0) {  
 return 0;  
 }  
 if (n == 1) {  
 return 1;  
 }  
 return fibonacci(n - 1) + fibonacci(n - 2);  
}  
  
/\* 配列を表示する関数 \*/  
void display\_array(int arr[], int size)  
{  
 int i;  
 printf("配列の内容: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
/\* 配列の合計を計算する関数 \*/  
int sum\_array(int arr[], int size)  
{  
 int sum = 0;  
 int i;  
   
 for (i = 0; i < size; i++) {  
 sum += arr[i];  
 }  
 return sum;  
}  
  
/\* 配列の平均を計算する関数 \*/  
double average\_array(int arr[], int size)  
{  
 if (size == 0) {  
 return 0.0;  
 }  
 return (double)sum\_array(arr, size) / size;  
}  
  
/\* グローバル変数を操作する関数 \*/  
void increment\_counter(void)  
{  
 global\_counter++;  
 printf("グローバルカウンター: %d\n", global\_counter);  
}  
  
/\* ローカル変数とグローバル変数のデモ \*/  
int local\_vs\_global\_demo(int param)  
{  
 int local\_var = 10; /\* ローカル変数 \*/  
   
 printf(" 関数内 - ローカル変数: %d\n", local\_var);  
 printf(" 関数内 - グローバル変数: %d\n", global\_counter);  
 printf(" 関数内 - 引数: %d\n", param);  
   
 local\_var += param;  
 global\_counter += param;  
   
 return local\_var;  
}  
  
int main(void)  
{  
 int i;  
   
 printf("=== 高度な関数の使用例 ===\n\n");  
   
 /\* 1. 再帰関数（階乗） \*/  
 printf("=== 再帰関数: 階乗 ===\n");  
 for (i = 0; i <= 7; i++) {  
 printf("%d! = %d\n", i, factorial(i));  
 }  
 printf("\n");  
   
 /\* 2. 再帰関数（フィボナッチ数列） \*/  
 printf("=== 再帰関数: フィボナッチ数列 ===\n");  
 printf("最初の10個のフィボナッチ数: ");  
 for (i = 0; i < 10; i++) {  
 printf("%d ", fibonacci(i));  
 }  
 printf("\n\n");  
   
 /\* 3. 配列を引数に取る関数 \*/  
 printf("=== 配列を引数に取る関数 ===\n");  
 int numbers[] = {5, 12, 8, 23, 16, 9, 14, 7};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
   
 display\_array(numbers, size);  
 printf("合計: %d\n", sum\_array(numbers, size));  
 printf("平均: %.2f\n", average\_array(numbers, size));  
 printf("\n");  
   
 /\* 4. ローカル変数とグローバル変数 \*/  
 printf("=== ローカル変数とグローバル変数 ===\n");  
 printf("初期状態のグローバル変数: %d\n", global\_counter);  
   
 printf("\n1回目の関数呼び出し:\n");  
 int result1 = local\_vs\_global\_demo(5);  
 printf(" 戻り値: %d\n", result1);  
   
 printf("\n2回目の関数呼び出し:\n");  
 int result2 = local\_vs\_global\_demo(3);  
 printf(" 戻り値: %d\n", result2);  
   
 printf("\n最終的なグローバル変数: %d\n\n", global\_counter);  
   
 /\* 5. グローバル変数を操作する関数 \*/  
 printf("=== グローバル変数の操作 ===\n");  
 for (i = 0; i < 5; i++) {  
 increment\_counter();  
 }  
 printf("\n");  
   
 /\* 6. 関数の組み合わせ使用 \*/  
 printf("=== 関数の組み合わせ使用 ===\n");  
   
 /\* 配列の各要素の階乗を計算 \*/  
 int sumall\_numbers[] = {1, 2, 3, 4, 5};  
 int sumall\_size = sizeof(sumall\_numbers) / sizeof(sumall\_numbers[0]);  
   
 printf("元の配列: ");  
 display\_array(sumall\_numbers, sumall\_size);  
   
 printf("各要素の階乗:\n");  
 for (i = 0; i < sumall\_size; i++) {  
 printf("%d! = %d\n", sumall\_numbers[i], factorial(sumall\_numbers[i]));  
 }  
 printf("\n");  
   
 /\* 7. 再帰の深さの制限例 \*/  
 printf("=== 再帰の制限例 ===\n");  
 printf("大きな数での階乗計算:\n");  
 for (i = 10; i <= 12; i++) {  
 printf("%d! = %d\n", i, factorial(i));  
 }  
 printf("注意: 大きな数では整数オーバーフローが発生する可能性があります\n");  
   
 return 0;  
}```  
  
> C99版は[こちら](advanced\_functions\_c99.c)を参照してください。  
  
### function\_basics.c  
  
```c  
/\*  
 \* 関数の基本的な使い方  
 \*   
 \* このプログラムは、関数の定義、呼び出し、  
 \* 引数と戻り値の基本的な使い方を示します。  
 \*/  
#include <stdio.h>  
  
/\* 関数プロトタイプ宣言 \*/  
int add(int a, int b);  
int subtract(int a, int b);  
int multiply(int a, int b);  
double divide(int a, int b);  
void print\_message(void);  
void print\_number(int num);  
int square(int n);  
int max(int a, int b);  
  
/\* 2つの数の加算 \*/  
int add(int a, int b)  
{  
 return a + b;  
}  
  
/\* 2つの数の減算 \*/  
int subtract(int a, int b)  
{  
 return a - b;  
}  
  
/\* 2つの数の乗算 \*/  
int multiply(int a, int b)  
{  
 return a \* b;  
}  
  
/\* 2つの数の除算（浮動小数点で返す） \*/  
double divide(int a, int b)  
{  
 if (b == 0) {  
 printf("エラー: ゼロで割ることはできません\n");  
 return 0.0;  
 }  
 return (double)a / b;  
}  
  
/\* 戻り値なし、引数なしの関数 \*/  
void print\_message(void)  
{  
 printf("これは関数から出力されたメッセージです\n");  
}  
  
/\* 戻り値なし、引数ありの関数 \*/  
void print\_number(int num)  
{  
 printf("数値: %d\n", num);  
}  
  
/\* 数の平方を計算 \*/  
int square(int n)  
{  
 return n \* n;  
}  
  
/\* 2つの数の最大値を返す \*/  
int max(int a, int b)  
{  
 if (a > b) {  
 return a;  
 } else {  
 return b;  
 }  
}  
  
int main(void)  
{  
 int x = 15;  
 int y = 7;  
 int result;  
 double dresult;  
   
 printf("=== 関数の基本操作 ===\n");  
 printf("x = %d, y = %d\n\n", x, y);  
   
 /\* 1. 基本的な算術関数 \*/  
 printf("=== 基本的な算術関数 ===\n");  
 result = add(x, y);  
 printf("%d + %d = %d\n", x, y, result);  
   
 result = subtract(x, y);  
 printf("%d - %d = %d\n", x, y, result);  
   
 result = multiply(x, y);  
 printf("%d x %d = %d\n", x, y, result);  
   
 dresult = divide(x, y);  
 printf("%d / %d = %.2f\n", x, y, dresult);  
 printf("\n");  
   
 /\* 2. void関数の使用 \*/  
 printf("=== void関数の使用 ===\n");  
 print\_message();  
 print\_number(42);  
 print\_number(x + y);  
 printf("\n");  
   
 /\* 3. 関数の戻り値を直接使用 \*/  
 printf("=== 関数の戻り値を直接使用 ===\n");  
 printf("%d の平方は %d です\n", x, square(x));  
 printf("%d と %d の最大値は %d です\n", x, y, max(x, y));  
   
 /\* 関数呼び出しをネスト \*/  
 printf("(%d + %d) の平方は %d です\n", x, y, square(add(x, y)));  
 printf("\n");  
   
 /\* 4. 関数を使った計算の例 \*/  
 printf("=== 複雑な計算例 ===\n");  
 {  
 /\* 三角形の面積計算（底辺 x 高さ / 2） \*/  
 int base = 10;  
 int height = 8;  
 double area = divide(multiply(base, height), 2);  
 printf("底辺%d、高さ%dの三角形の面積: %.1f\n", base, height, area);  
   
 /\* 円の面積の近似計算 \*/  
 int radius = 5;  
 double pi = 3.14159;  
 double circle\_area = pi \* square(radius);  
 printf("半径%dの円の面積（近似）: %.2f\n", radius, circle\_area);  
 printf("\n");  
 }  
   
 /\* 5. デモ用の固定値を使った関数 \*/  
 printf("=== デモ用固定値での関数使用 ===\n");  
 int num1 = 15, num2 = 4;  
   
 printf("使用する値: %d と %d\n", num1, num2);  
   
 printf("\n計算結果:\n");  
 printf("%d + %d = %d\n", num1, num2, add(num1, num2));  
 printf("%d - %d = %d\n", num1, num2, subtract(num1, num2));  
 printf("%d x %d = %d\n", num1, num2, multiply(num1, num2));  
   
 if (num2 != 0) {  
 printf("%d / %d = %.2f\n", num1, num2, divide(num1, num2));  
 }  
   
 printf("最大値: %d\n", max(num1, num2));  
   
 return 0;  
}

C99版は[こちら](function_basics_c99.c)を参照してください。

# 第10章: ポインタ

## 対応C規格

* **主要対象:** C90
* **学習内容:** ポインタの基本概念、アドレス演算子、間接参照演算子、ポインタと配列、ポインタ演算

## 学習目標

この章を完了すると、以下のことができるようになります：

* ポインタの基本概念を理解する
* アドレス演算子（&）と間接参照演算子（\*）を使える
* ポインタと配列の関係を理解する
* ポインタ演算ができる
* ポインタを関数の引数として使える

## 概要と詳細

### ポインタとは？

ポインタは、C言語で最も重要かつ難しいと言われる概念です。しかし、適切な例えを使えば理解できます！

#### 日常生活でのポインタ

ポインタを理解するために、「住所」の例えを使いましょう：

1. **家 = 変数**
   * 実際に人が住んでいる場所
   * 中に値（住人）が入っている
2. **住所 = ポインタ**
   * 家の場所を示す情報
   * 住所を知っていれば、その家に行ける
3. **住所録 = ポインタ変数**
   * 住所を書き留めておく手帳
   * 違う住所に書き換えることもできる

int house = 100; /\* 家（変数）に100という値が住んでいる \*/  
int \*address = &house; /\* 住所録にhouseの住所を記録 \*/

### ポインタの基本概念

ポインタは他の変数のメモリアドレスを格納する変数です。これにより間接的に他の変数にアクセスできます。

#### なぜポインタが必要？

1. **大きなデータを効率的に扱える**
   * データのコピーではなく、場所だけを教える
2. **関数で複数の値を返せる**
   * 通常の関数は1つしか値を返せないが、ポインタなら複数可能
3. **動的なメモリ管理ができる**
   * プログラム実行中に必要なメモリを確保

#### メモリとアドレス

#include <stdio.h>  
  
int main(void)  
{  
  
 int number = 42;  
   
 printf("変数numberの値: %d\n", number);  
 printf("変数numberのアドレス: %p\n", (void\*)&number);  
 printf("変数numberのサイズ: %lu バイト\n", (unsigned long)sizeof(number));  
   
 return 0;

#### ポインタ変数の宣言

#include <stdio.h>  
  
int main(void)  
{  
  
 int value = 100; /\* 通常の整数変数 \*/  
 int \*ptr; /\* 整数を指すポインタ変数 \*/  
   
 ptr = &value; /\* valueのアドレスをptrに代入 \*/  
   
 printf("value = %d\n", value);  
 printf("&value = %p\n", (void\*)&value);  
 printf("ptr = %p\n", (void\*)ptr);  
 printf("\*ptr = %d\n", \*ptr); /\* ポインタが指す値 \*/  
   
 return 0;

### アドレス演算子（&）と間接参照演算子（\*）

この2つの演算子は、ポインタを使う上で最も重要です。

#### 演算子の意味を覚える方法

* **& = “の住所”**（address of）
  + &house = “houseの住所”
* **\* = “が指す場所の中身”**（value at）
  + \*address = “addressが指す場所の中身”

#### アドレス演算子（&）

#include <stdio.h>  
  
int main(void)  
{  
  
 int a = 10;  
 double b = 3.14;  
 char c = 'A';  
   
 printf("変数のアドレス:\n");  
 printf("&a = %p\n", (void\*)&a);  
 printf("&b = %p\n", (void\*)&b);  
 printf("&c = %p\n", (void\*)&c);  
   
 /\* ポインタ変数の宣言と初期化 \*/  
 int \*ptr\_a = &a;  
 double \*ptr\_b = &b;  
 char \*ptr\_c = &c;  
   
 printf("\nポインタの値（アドレス）:\n");  
 printf("ptr\_a = %p\n", (void\*)ptr\_a);  
 printf("ptr\_b = %p\n", (void\*)ptr\_b);  
 printf("ptr\_c = %p\n", (void\*)ptr\_c);  
   
 return 0;

#### 間接参照演算子（\*）

#include <stdio.h>  
  
int main(void)  
{  
  
 int original = 50;  
 int \*pointer = &original;  
   
 printf("=== 元の状態 ===\n");  
 printf("original = %d\n", original);  
 printf("\*pointer = %d\n", \*pointer);  
   
 /\* ポインタを通じて値を変更 \*/  
 \*pointer = 75;  
   
 printf("\n=== \*pointer = 75 実行後 ===\n");  
 printf("original = %d\n", original); /\* 75に変更される \*/  
 printf("\*pointer = %d\n", \*pointer); /\* 75 \*/  
   
 /\* 元の変数を直接変更 \*/  
 original = 99;  
   
 printf("\n=== original = 99 実行後 ===\n");  
 printf("original = %d\n", original); /\* 99 \*/  
 printf("\*pointer = %d\n", \*pointer); /\* 99 \*/  
   
 return 0;

### ポインタのデータ型

#### さまざまなデータ型のポインタ

#include <stdio.h>  
  
int main(void)  
{  
  
 /\* 各データ型の変数 \*/  
 char char\_var = 'X';  
 int int\_var = 123;  
 float float\_var = 3.14f;  
 double double\_var = 2.718;  
   
 /\* 各データ型のポインタ \*/  
 char \*char\_ptr = &char\_var;  
 int \*int\_ptr = &int\_var;  
 float \*float\_ptr = &float\_var;  
 double \*double\_ptr = &double\_var;  
   
 printf("=== 値の表示 ===\n");  
 printf("char: %c\n", \*char\_ptr);  
 printf("int: %d\n", \*int\_ptr);  
 printf("float: %.2f\n", \*float\_ptr);  
 printf("double: %.3f\n", \*double\_ptr);  
   
 printf("\n=== ポインタのサイズ ===\n");  
 printf("char\*: %lu バイト\n", (unsigned long)sizeof(char\_ptr));  
 printf("int\*: %lu バイト\n", (unsigned long)sizeof(int\_ptr));  
 printf("float\*: %lu バイト\n", (unsigned long)sizeof(float\_ptr));  
 printf("double\*: %lu バイト\n", (unsigned long)sizeof(double\_ptr));  
   
 return 0;

#### void\*ポインタ（汎用ポインタ）

#include <stdio.h>  
  
int main(void)  
{  
  
 int int\_value = 456;  
 double double\_value = 1.618;  
   
 void \*generic\_ptr; /\* 汎用ポインタ \*/  
   
 /\* intを指す \*/  
 generic\_ptr = &int\_value;  
 printf("int値: %d\n", \*(int\*)generic\_ptr); /\* キャストが必要 \*/  
   
 /\* doubleを指す \*/  
 generic\_ptr = &double\_value;  
 printf("double値: %.3f\n", \*(double\*)generic\_ptr); /\* キャストが必要 \*/  
   
 return 0;

### ポインタと配列の関係

ポインタと配列は非常に密接な関係があります。実は、配列名そのものがポインタとして扱われることを理解すると、C言語がより深く理解できます。

#### 配列名はポインタ

配列名は「配列の最初の要素を指すポインタ」として扱われます。これは重要な概念です！

#include <stdio.h>  
  
int main(void)  
{  
  
 int arr[] = {10, 20, 30, 40, 50};  
 int \*ptr = arr; /\* arr は &arr[0] と同じ \*/  
 int i;  
   
 printf("=== 配列とポインタの関係 ===\n");  
 printf("arr = %p\n", (void\*)arr);  
 printf("&arr[0] = %p\n", (void\*)&arr[0]);  
 printf("ptr = %p\n", (void\*)ptr);  
   
 printf("\n=== 配列要素へのアクセス ===\n");  
 for (i = 0; i < 5; i++) {  
 printf("arr[%d] = %d, \*(arr + %d) = %d, \*(ptr + %d) = %d\n",  
 i, arr[i], i, \*(arr + i), i, \*(ptr + i));  
 }  
   
 return 0;

#### ポインタを使った配列操作

#include <stdio.h>  
  
void print\_array\_index(int arr[], int size)  
{  
 int i;  
   
 printf("インデックス記法: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
void print\_array\_pointer(int \*ptr, int size)  
{  
 int i;  
   
 printf("ポインタ記法: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*(ptr + i));  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 int numbers[] = {11, 22, 33, 44, 55};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
   
 print\_array\_index(numbers, size);  
 print\_array\_pointer(numbers, size);  
   
 return 0;

### ポインタ演算

ポインタには通常の数値とは異なる特殊な演算規則があります。これを理解することで、配列やメモリを効率的に操作できるようになります。

#### なぜポインタ演算が特殊なのか？

ポインタに1を足しても、アドレスが1増えるわけではありません。代わりに「次の要素」に移動します。

int配列の場合：  
ptr + 1 → 次のint（通常4バイト先）へ移動  
  
char配列の場合：  
ptr + 1 → 次のchar（1バイト先）へ移動

#### ポインタの加算・減算

#include <stdio.h>  
  
int main(void)  
{  
  
 int arr[] = {5, 15, 25, 35, 45};  
 int \*ptr = arr;  
 int i;  
   
 printf("=== ポインタ演算の例 ===\n");  
 for (i = 0; i < 5; i++) {  
 printf("ptr + %d = %p, \*(ptr + %d) = %d\n",  
 i, (void\*)(ptr + i), i, \*(ptr + i));  
 }  
   
 printf("\n=== ポインタのインクリメント ===\n");  
 ptr = arr; /\* 先頭に戻す \*/  
 for (i = 0; i < 5; i++) {  
 printf("\*ptr = %d (アドレス: %p)\n", \*ptr, (void\*)ptr);  
 ptr++; /\* 次の要素に移動 \*/  
 }  
   
 return 0;

#### ポインタ同士の差

#include <stdio.h>  
  
int main(void)  
{  
  
 int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
 int \*start = &arr[2]; /\* arr[2]を指す \*/  
 int \*end = &arr[7]; /\* arr[7]を指す \*/  
   
 printf("start が指す値: %d (インデックス2)\n", \*start);  
 printf("end が指す値: %d (インデックス7)\n", \*end);  
 printf("ポインタの差: %ld\n", end - start); /\* 5 \*/  
 printf("バイト差: %ld\n", (char\*)end - (char\*)start);  
   
 return 0;

### ポインタと関数

関数とポインタを組み合わせることで、C言語の真の力を発揮できます。特に、関数で複数の値を変更したり、大きなデータを効率的に扱えるようになります。

#### なぜ関数でポインタを使うのか？

1. **複数の値を変更できる**
   * 通常の関数は1つの値しか返せない
   * ポインタなら複数の値を同時に変更可能
2. **効率的なデータ処理**
   * 大きな配列や構造体をコピーせずに処理
   * メモリと処理時間を節約
3. **実際の変数を変更できる**
   * 値渡しでは元の変数は変わらない
   * ポインタ渡しなら元の変数を直接変更

#### ポインタを引数とする関数

#include <stdio.h>  
  
/\* 値を交換する関数 \*/  
void swap(int \*a, int \*b)  
{  
 int temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
}  
  
/\* 値を倍にする関数 \*/  
void double\_value(int \*value)  
{  
 \*value = (\*value) \* 2;  
}  
  
/\* 配列の要素を変更する関数 \*/  
void modify\_array(int arr[], int size)  
{  
 int i;  
   
 for (i = 0; i < size; i++) {  
 arr[i] \*= 3; /\* 各要素を3倍 \*/  
 }  
}  
  
int main(void)  
{  
 int x = 8, y = 12;  
 int numbers[] = {1, 2, 3, 4, 5};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
 int i;  
   
 printf("=== swap関数のテスト ===\n");  
 printf("交換前: x = %d, y = %d\n", x, y);  
 swap(&x, &y);  
 printf("交換後: x = %d, y = %d\n", x, y);  
   
 printf("\n=== double\_value関数のテスト ===\n");  
 printf("2倍前: x = %d\n", x);  
 double\_value(&x);  
 printf("2倍後: x = %d\n", x);  
   
 printf("\n=== modify\_array関数のテスト ===\n");  
 printf("変更前: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 modify\_array(numbers, size);  
   
 printf("変更後: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 return 0;

#### ポインタを戻り値とする関数

#include <stdio.h>  
  
/\* 配列から最大値のアドレスを返す \*/  
int\* find\_max\_ptr(int arr[], int size)  
{  
  
 int \*max\_ptr = &arr[];  
 int i;  
   
 for (i = ; i < size; i++)   
 if (arr[i] > \*max\_ptr)   
 max\_ptr = &arr[i];  
   
   
   
 return max\_ptr;  
  
  
/\* 配列から指定値を検索してアドレスを返す \*/  
int\* search\_vale(int arr[], int size, int target)  
{  
  
 int i;  
   
 for (i = ; i < size; i++)   
 if (arr[i] == target)   
 return &arr[i]; /\* 見2つかった要素のアドレス \*/  
   
   
   
 return NULL; /\* 見2つからない場合 \*/  
  
  
int main(void)  
{  
  
 int data[] = , , , 9, , , ;  
 int size = sizeof(data) / sizeof(data[]);  
 int \*result\_ptr;  
 int search\_target = ;  
 int i;  
   
 /\* 配列の表示 \*/  
 printf("配列: ");  
 for (i = ; i < size; i++)   
 printf("%d ", data[i]);  
   
 printf("n");  
   
 /\* 最大値の検索 \*/  
 result\_ptr = find\_max\_ptr(data, size);  
 printf("最大値: %d (アドレス: %p)n", \*result\_ptr, (void\*)result\_ptr);  
   
 /\* 特定値の検索 \*/  
 result\_ptr = search\_vale(data, size, search\_target);  
 if (result\_ptr != NULL)   
 printf("%d が見2つかりました (アドレス: %p)n",   
 \*result\_ptr, (void\*)result\_ptr);  
 else   
 printf("%d は見2つかりませんでしたn", search\_target);  
   
   
 return ;

### 文字列とポインタ

文字列とポインタの関係は、C言語でよく使われる重要なトピックです。文字列は実質的に文字の配列であり、ポインタを使って効率的に操作できます。

#### 文字列の2つの表現方法

1. **文字配列として**

* char str[] = "Hello"; /\* 変更可能 \*/

1. **文字列ポインタとして**

* char \*str = "Hello"; /\* 変更不可（読み取り専用） \*/

この違いを理解することが重要です！

#### 文字列リテラルとポインタ

#include <stdio.h>  
  
int main(void)  
{  
  
 char \*str = "Hello, World!"; /\* 文字列リテラル \*/  
 char str[] = "Hello, World!"; /\* 文字配列 \*/  
 char \*ptr = str;  
   
 printf("str: %s\n", str);  
 printf("str: %s\n", str);  
 printf("ptr: %s\n", ptr);  
   
 printf("n=== アドレスの比較 ===n");  
 printf("strのアドレス: %pn", (void\*)str);  
 printf("strのアドレス: %pn", (void\*)str);  
 printf("ptrの値: %pn", (void\*)ptr);  
   
 /\* 文字配列は変更可能 \*/  
 str[] = 'h';  
 printf("n変更後のstr: %s\n", str);  
   
 /\* 文字列リテラルは変更不可 \*/  
 /\* str[] = 'h'; <- 実行時エラーの可能性 \*/  
   
 return ;

#### ポインタを使った文字列操作

#include <stdio.h>  
  
/\* 文字列の長さを計算（ポインタ版） \*/  
int string\_length(char \*str)  
{  
  
 int length = 0;  
   
 while (\*str != '\0') {  
 length++;  
 str++;  
 }  
   
   
 return length;  
  
  
/\* 文字列をコピー（ポインタ版） \*/  
void string\_copy(char \*dest, char \*src)  
{  
  
 while (\*src != '')   
 \*dest = \*src;  
 dest++;  
 src++;  
   
 \*dest = ''; /\* nll終端文字を追加 \*/  
  
  
/\* 文字列を連結（ポインタ版） \*/  
void string\_concat(char \*dest, char \*src)  
{  
  
 /\* destの末尾を見2つける \*/  
 while (\*dest != '')   
 dest++;  
   
   
 /\* srcをdestの末尾に追加 \*/  
 while (\*src != '')   
 \*dest = \*src;  
 dest++;  
 src++;  
   
 \*dest = '';  
  
  
int main(void)  
{  
  
 char str[] = "Hello";  
 char str[] = "World";  
 char bffer[];  
 char result[];  
   
 printf("元の文字列: "%s", "%s"n", str, str);  
   
 /\* 長さの計算 \*/  
 printf("strの長さ: %d\n", string\_length(str));  
 printf("strの長さ: %d\n", string\_length(str));  
   
 /\* 文字列のコピー \*/  
 string\_copy(bffer, str);  
 printf("コピー結果: "%s"n", bffer);  
   
 /\* 文字列の連結 \*/  
 string\_copy(result, str); /\* まずstrをコピー \*/  
 string\_concat(result, ", ");  
 string\_concat(result, str);  
 string\_concat(result, "!");  
 printf("連結結果: "%s"n", result);  
   
 return ;

### ポインタの配列

ポインタも配列にできます。これは複数の文字列を管理したり、複数の異なる配列を扱う場合に便利です。

#### ポインタ配列が役立つ場面

1. **複数の文字列を管理**
   * メニュー項目
   * エラーメッセージ
   * コマンドライン引数
2. **異なる長さのデータを効率的に扱う**
   * 固定長配列より柔軟
   * メモリを無駄にしない

#### ポインタ配列の基本

#include <stdio.h>  
  
int main(void)  
{  
  
 int a = , b = , c = , d = ;  
 int \*ptr\_array[]; /\* ポインタの配列 \*/  
 int i;  
   
 /\* ポインタ配列に各変数のアドレスを格納 \*/  
 ptr\_array[] = &a;  
 ptr\_array[] = &b;  
 ptr\_array[] = &c;  
 ptr\_array[] = &d;  
   
 printf("=== ポインタ配列の内容 ===n");  
 for (i = ; i < ; i++)   
 printf("ptr\_array[%d] = %p, \*ptr\_array[%d] = %d\n",  
 i, (void\*)ptr\_array[i], i, \*ptr\_array[i]);  
   
   
 /\* ポインタを通じて値を変更 \*/  
 \*ptr\_array[] = ;  
 \*ptr\_array[] = ;  
   
 printf("n=== 変更後の値 ===n");  
 printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);  
   
 return ;

### よくあるポインタの間違いと対策

初心者がつまずきやすいポインタの落とし穴を知って、安全にプログラミングしましょう。

#### 1. 初期化していないポインタの使用

/\* 危険！未初期化ポインタ \*/  
int \*ptr;  
\*ptr = 10; /\* どこを指しているか不明！ \*/  
  
/\* 安全：NULLで初期化 \*/  
int \*ptr = NULL;  
if (ptr != NULL) {  
 \*ptr = 10;  
}

#### 2. NULLポインタの参照

/\* 危険！NULLチェックなし \*/  
void print\_string(char \*str) {  
 printf("%s\n", str); /\* strがNULLだとクラッシュ \*/  
}  
  
/\* 安全：NULLチェックあり \*/  
void print\_string\_safe(char \*str) {  
 if (str != NULL) {  
 printf("%s\n", str);  
 } else {  
 printf("(null)\n");  
 }  
}

#### 3. 配列の範囲外アクセス

/\* 危険！範囲外アクセス \*/  
int arr[5] = {1, 2, 3, 4, 5};  
int \*ptr = arr;  
ptr += 10; /\* 配列の範囲外！ \*/  
\*ptr = 100; /\* 危険な書き込み \*/  
  
/\* 安全：範囲チェック \*/  
int index = 10;  
if (index >= 0 && index < 5) {  
 arr[index] = 100;  
}

#### 4. ポインタ型の不一致

/\* 危険！型が違う \*/  
int value = 100;  
char \*ptr = &value; /\* 警告が出る \*/  
  
/\* 安全：正しい型を使用 \*/  
int value = 100;  
int \*ptr = &value;

### ポインタを学ぶためのステップ

1. **基本から始める**
   * まずアドレス演算子(&)を理解
   * 次に間接参照演算子(\*)を理解
2. **図を描いて理解**
   * メモリの箱とアドレスの図を描く
   * ポインタがどこを指しているか視覚化
3. **小さなプログラムで練習**
   * 値の交換（swap関数）から始める
   * 徐々に複雑な例に挑戦
4. **デバッガを活用**
   * 変数のアドレスを確認
   * ポインタの値を追跡

#### 文字列ポインタの配列

#include <stdio.h>  
  
int main(void)  
{  
  
 char \*frits[] =   
 "Apple",  
 "anana",  
 "Cherry",  
 "Date",  
 "lderberry"  
 ;  
 int count = sizeof(frits) / sizeof(frits[]);  
 int i;  
   
 printf("=== 果物リスト ===n");  
 for (i = ; i < count; i++)   
 printf("%d. %s\n", i + , frits[i]);  
   
   
 /\* ポインタの変更 \*/  
 frits[] = "leberry";  
   
 printf("n=== 変更後 ===n");  
 for (i = ; i < count; i++)   
 printf("%d. %s\n", i + , frits[i]);  
   
   
 return ;

### 実践的なポインタ活用例

#### 動的配列の操作

#include <stdio.h>  
  
/\* 配列内の要素を逆順にする \*/  
void reverse\_array(int \*arr, int size)  
{  
  
 int \*start = arr;  
 int \*end = arr + size - ;  
 int temp;  
   
 while (start < end)   
 temp = \*start;  
 \*start = \*end;  
 \*end = temp;  
 start++;  
 end--;  
   
  
  
/\* 配列を1回転させる（右につシフト） \*/  
void rotate\_right(int \*arr, int size)  
{  
  
 int last = \*(arr + size - );  
 int i;  
   
 for (i = size - ; i > ; i--)   
 \*(arr + i) = \*(arr + i - );  
   
 \*arr = last;  
  
  
int main(void)  
{  
  
 int numbers[] = , , , , , , , ;  
 int size = sizeof(numbers) / sizeof(numbers[]);  
 int i;  
   
 printf("元の配列: ");  
 for (i = ; i < size; i++)   
 printf("%d ", numbers[i]);  
   
 printf("n");  
   
 /\* 配列を逆順にする \*/  
 reverse\_array(numbers, size);  
 printf("逆順後: ");  
 for (i = ; i < size; i++)   
 printf("%d ", numbers[i]);  
   
 printf("n");  
   
 /\* 配列を右に1回転 \*/  
 rotate\_right(numbers, size);  
 printf("右1回転後: ");  
 for (i = ; i < size; i++)   
 printf("%d ", numbers[i]);  
   
 printf("n");  
   
 return ;

#### ポインタを使ったソート

#include <stdio.h>  
  
/\* ポインタを使ったバブルソート \*/  
void bbble\_sort\_ptr(int \*arr, int size)  
{  
  
 int i, j;  
 int \*ptr, \*ptr;  
 int temp;  
   
 for (i = ; i < size - ; i++)   
 for (j = ; j < size - - i; j++)   
 ptr = arr + j;  
 ptr = arr + j + ;  
   
 if (\*ptr > \*ptr)   
 temp = \*ptr;  
 \*ptr = \*ptr;  
 \*ptr = temp;  
   
   
   
  
  
/\* 2つの配列を比較 \*/  
int compare\_arrays(int \*arr, int \*arr, int size)  
{  
  
 int i;  
   
 for (i = ; i < size; i++)   
 if (\*(arr + i) != \*(arr + i))   
 return ; /\* 異なる \*/  
   
   
   
 return ; /\* 同じ \*/  
  
  
int main(void)  
{  
  
 int original[] = , , , , , , 9;  
 int copy[] = , , , , , , 9;  
 int size = sizeof(original) / sizeof(original[]);  
 int i;  
   
 printf("ソート前: ");  
 for (i = ; i < size; i++)   
 printf("%d ", original[i]);  
   
 printf("n");  
   
 /\* ソート実行 \*/  
 bbble\_sort\_ptr(original, size);  
   
 printf("ソート後: ");  
 for (i = ; i < size; i++)   
 printf("%d ", original[i]);  
   
 printf("n");  
   
 /\* 配列の比較 \*/  
 if (compare\_arrays(original, copy, size))   
 printf("配列は同じですn");  
 else   
 printf("配列は異なりますn");  
   
   
 return ;

## サンプルコード

### ポインタの基本操作

プログラムファイル: examples/pointer\_basic.c

ポインタの宣言、初期化、基本的な操作を学習します。

**C99版**: [pointer\_basic\_c99.c](examples/pointer_basic_c99.c) - bool型、restrict修飾子、固定幅整数型を使用

### ポインタと配列

プログラムファイル: examples/pointer\_arrays.c

ポインタと配列の関係、ポインタ演算を学習します。

**C99版**: [pointer\_arrays\_c99.c](examples/pointer_arrays_c99.c) - 可変長配列、複合リテラル、指定初期化子を使用

### ポインタと関数

プログラムファイル: examples/pointer\_functions.c

関数引数、戻り値としてのポインタの使用方法を学習します。

**C99版**: [pointer\_functions\_c99.c](examples/pointer_functions_c99.c) - インライン関数、restrict修飾子、可変長配列パラメータを使用

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic pointer\_basic.c -o pointer\_basic  
  
# 実行  
./pointer\_basic

## 演習課題

### 基礎問題

1. **ポインタの基本操作**
   * 2つの変数の値をポインタを使って交換するプログラムを作成してください
   * 解答例: <solutions/ex1_pointer_swap.c> (C90版) / <solutions/ex1_pointer_swap_c99.c> (C99版)
2. **配列とポインタ**
   * ポインタ演算を使って配列の要素を逆順に表示するプログラムを作成してください
   * 解答例: <solutions/ex2_array_reverse.c> (C90版) / <solutions/ex2_array_reverse_c99.c> (C99版)
3. **文字列操作**
   * ポインタを使って文字列の長さを計算し、文字列を逆順にするプログラムを作成してください

### 応用問題

1. **配列操作関数**
   * ポインタを使って配列の最大値、最小値、平均値を計算する関数を作成してください
2. **文字列処理**
   * ポインタを使って文字列の検索、置換、分割を実行する関数群を作成してください
3. **データ変換**
   * ポインタを使って配列のデータ型変換を実行するプログラムを作成してください

### 発展問題

1. **メモリ操作**
   * ポインタを使って任意のデータ型の配列をコピーする汎用関数を作成してください
2. **アルゴリズム実装**
   * ポインタを使って各種ソートアルゴリズムを実装してください
3. **データ構造**
   * ポインタを使って簡単なリンクリスト構造を実装してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make pointer\_basic  
  
# 実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* void\*からの暗黙的な型変換は制限的
* ポインタ演算は整数型との組み合わせのみ
* 関数ポインタの表記法は限定的

### C99以降の拡張

* restrict修飾子の追加
* 可変長配列とポインタの関係改善
* より厳密なポインタ型チェック

## よくある間違い

### 1. 未初期化ポインタの使用

/\* NG: 初期化されていないポインタ \*/  
int \*ptr;  
\*ptr = 10; /\* 危険: 不明なメモリへの書き込み \*/  
  
/\* OK: 適切に初期化 \*/  
int value = 0;  
int \*ptr = &value;  
\*ptr = 10;

### 2. NULLポインタの参照

/\* NG: NULLポインタの参照 \*/  
int \*ptr = NULL;  
printf("%d\n", \*ptr); /\* セグメンテーション違反 \*/  
  
/\* OK: NULLチェック \*/  
if (ptr != NULL) {  
 printf("%d\n", \*ptr);  
}

### 3. スコープ外変数へのポインタ

/\* NG: ローカル変数のアドレス返し \*/  
int\* bad\_function(void)  
{  
 int local = 100;  
 return &local; /\* 危険: ローカル変数のアドレス \*/  
}  
  
/\* OK: 静的変数または引数で渡された変数を使用 \*/  
int\* good\_function(int \*param)  
{  
 return param; /\* 引数で渡されたアドレスを返す \*/  
}

### 4. 配列境界の越えた参照

/\* NG: 配列の範囲外アクセス \*/  
int arr[] = {1, 2, 3, 4, 5};  
int \*ptr = arr;  
printf("%d\n", \*(ptr + 10)); /\* 範囲外アクセス \*/  
  
/\* OK: 範囲チェック \*/  
int index = 2;  
if (index >= 0 && index < 5) {  
 printf("%d\n", \*(ptr + index));  
}

## 次の章へ

ポインタの基礎を理解したら、「構造体とポインタ」の実装をお待ちください。現在実装中です。

## 参考資料

* [C言語ポインタリファレンス](https://ja.cppreference.com/w/c/language/pointer)
* [ポインタ演算の詳細](https://ja.cppreference.com/w/c/language/operautor_arithmetic)
* [メモリモデルとポインタ](https://ja.cppreference.com/w/c/language/memory_model)

## サンプルコード

### pointer\_arrays.c

/\*  
 \* pointer\_arrays.c  
 \* ポインタと配列の関係、ポインタ演算を学習  
 \* C90準拠  
 \*/  
  
#include <stdio.h>  
  
void print\_array\_with\_index(int arr[], int size)  
{  
 int i;  
 printf("インデックス記法: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
void print\_array\_with\_pointer(int \*ptr, int size)  
{  
 int i;  
 printf("ポインタ記法: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*(ptr + i));  
 }  
 printf("\n");  
}  
  
void print\_array\_with\_increment(int \*ptr, int size)  
{  
 int i;  
 printf("インクリメント: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*ptr);  
 ptr++; /\* ポインタを次の要素に進める \*/  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 int numbers[6] = {10, 20, 30, 40, 50, 60};  
 int \*ptr = numbers; /\* 配列名は先頭要素のアドレス \*/  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
 int i;  
 int \*start; /\* C90: ポインタ変数の宣言を先頭に配置 \*/  
 int \*end;  
 char message[] = "Hello, World!";  
 char \*str\_ptr;  
 int str\_len = 0;  
 int matrix[3][4] = {  
 {1, 2, 3, 4},  
 {5, 6, 7, 8},  
 {9, 10, 11, 12}  
 };  
 int \*matrix\_ptr;  
 int total\_elements = 3 \* 4;  
 int a = 100, b = 200, c = 300;  
 int \*ptr\_array[3]; /\* ポインタの配列 \*/  
   
 printf("===== 配列とポインタの関係 =====\n");  
 printf("配列名numbers: %p\n", (void\*)numbers);  
 printf("&numbers[0]: %p\n", (void\*)&numbers[0]);  
 printf("ポインタptr: %p\n", (void\*)ptr);  
 printf("これらはすべて同じアドレスです\n");  
   
 printf("\n===== 配列要素へのアクセス方法 =====\n");  
 print\_array\_with\_index(numbers, size);  
 print\_array\_with\_pointer(ptr, size);  
   
 /\* ポインタをリセット（print\_array\_with\_incrementでptrが変更されるため） \*/  
 ptr = numbers;  
 print\_array\_with\_increment(ptr, size);  
   
 printf("\n===== ポインタ演算の詳細 =====\n");  
 ptr = numbers; /\* 先頭に戻す \*/  
   
 for (i = 0; i < size; i++) {  
 printf("ptr + %d = %p, \*(ptr + %d) = %d\n",  
 i, (void\*)(ptr + i), i, \*(ptr + i));  
 }  
   
 printf("\n===== アドレスの差による要素間隔の確認 =====\n");  
 printf("numbers[0]のアドレス: %p\n", (void\*)&numbers[0]);  
 printf("numbers[1]のアドレス: %p\n", (void\*)&numbers[1]);  
 printf("アドレス差: %ld バイト\n",   
 (char\*)&numbers[1] - (char\*)&numbers[0]);  
 printf("int型のサイズ: %lu バイト\n", (unsigned long)sizeof(int));  
   
 printf("\n===== ポインタによる配列の逆順表示 =====\n");  
 ptr = numbers + size - 1; /\* 最後の要素を指す \*/  
 printf("逆順: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*ptr);  
 ptr--; /\* ポインタを前の要素に戻す \*/  
 }  
 printf("\n");  
   
 printf("\n===== ポインタ同士の演算 =====\n");  
 start = &numbers[1]; /\* numbers[1]を指す \*/  
 end = &numbers[4]; /\* numbers[4]を指す \*/  
   
 printf("start が指す値: %d (numbers[1])\n", \*start);  
 printf("end が指す値: %d (numbers[4])\n", \*end);  
 printf("ポインタの差: %ld (要素数)\n", end - start);  
 printf("バイト差: %ld バイト\n", (char\*)end - (char\*)start);  
   
 printf("\n===== 文字列（文字配列）とポインタ =====\n");  
 str\_ptr = message;  
   
 printf("文字列: %s\n", message);  
 printf("文字ごとの表示: ");  
   
 /\* 文字列の終端（'\0'）まで表示 \*/  
 while (\*str\_ptr != '\0') {  
 printf("%c", \*str\_ptr);  
 str\_ptr++;  
 str\_len++;  
 }  
 printf("\n");  
 printf("文字列の長さ: %d文字\n", str\_len);  
   
 printf("\n===== 2次元配列とポインタ =====\n");  
 matrix\_ptr = (int\*)matrix; /\* 1次元ポインタとして扱う \*/  
   
 printf("2次元配列を1次元ポインタで表示:\n");  
 for (i = 0; i < total\_elements; i++) {  
 printf("%2d ", \*(matrix\_ptr + i));  
 if ((i + 1) % 4 == 0) {  
 printf("\n");  
 }  
 }  
   
 printf("\n===== ポインタ配列の例 =====\n");  
   
 ptr\_array[0] = &a;  
 ptr\_array[1] = &b;  
 ptr\_array[2] = &c;  
   
 printf("ポインタ配列の内容:\n");  
 for (i = 0; i < 3; i++) {  
 printf("ptr\_array[%d] = %p, \*ptr\_array[%d] = %d\n",  
 i, (void\*)ptr\_array[i], i, \*ptr\_array[i]);  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 配列名は先頭要素のポインタと同等  
2. ポインタ演算: ptr + n は n個後の要素を指す  
3. \*(ptr + i) と ptr[i] は同等  
4. ポインタのインクリメント/デクリメント  
5. ポインタ同士の差は要素数を表す  
6. 文字列もポインタで操作可能  
7. 2次元配列も1次元ポインタで扱える  
8. ポインタの配列の活用  
  
重要な概念:  
- 配列の記法とポインタ記法の等価性  
- ポインタ演算の自動的な型サイズ調整  
- メモリレイアウトの理解  
- 安全な境界チェックの重要性  
\*/

C99版は[こちら](pointer_arrays_c99.c)を参照してください。

### pointer\_basic.c

/\*  
 \* pointer\_basic.c  
 \* ポインタの宣言、初期化、基本的な操作を学習  
 \* C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void)  
{  
 int value = 42; /\* 通常の整数変数 \*/  
 int \*ptr; /\* 整数ポインタ変数の宣言 \*/  
 int another\_value = 100;  
 char char\_var = 'A'; /\* C90: 全変数宣言を先頭に配置 \*/  
 float float\_var = 3.14f;  
 double double\_var = 2.718;  
 char \*char\_ptr;  
 float \*float\_ptr;  
 double \*double\_ptr;  
 int \*null\_ptr; /\* C90: NULLポインタ用変数 \*/  
 void \*generic\_ptr; /\* C90: void\*ポインタ用変数 \*/  
 int int\_val = 123;  
 double double\_val = 45.67;  
   
 printf("===== ポインタの基本操作 =====\n");  
   
 /\* ポインタの初期化 \*/  
 ptr = &value; /\* valueのアドレスをptrに代入 \*/  
   
 printf("変数valueの値: %d\n", value);  
 printf("変数valueのアドレス: %p\n", (void\*)&value);  
 printf("ポインタptrの値（アドレス）: %p\n", (void\*)ptr);  
 printf("ポインタptrが指す値: %d\n", \*ptr);  
   
 printf("\n===== 間接参照による値の変更 =====\n");  
 printf("変更前: value = %d, \*ptr = %d\n", value, \*ptr);  
   
 \*ptr = 75; /\* ポインタを通じて値を変更 \*/  
 printf("\*ptr = 75 実行後:\n");  
 printf("変更後: value = %d, \*ptr = %d\n", value, \*ptr);  
   
 printf("\n===== ポインタの再代入 =====\n");  
 printf("another\_value = %d\n", another\_value);  
   
 ptr = &another\_value; /\* ptrを別の変数に向ける \*/  
 printf("ptr = &another\_value 実行後:\n");  
 printf("ポインタptrが指す値: %d\n", \*ptr);  
 printf("ポインタptrのアドレス: %p\n", (void\*)ptr);  
   
 printf("\n===== 様々なデータ型のポインタ =====\n");  
   
 /\* C90: ポインタの初期化は宣言後に \*/  
 char\_ptr = &char\_var;  
 float\_ptr = &float\_var;  
 double\_ptr = &double\_var;  
   
 printf("char値: %c (アドレス: %p)\n", \*char\_ptr, (void\*)char\_ptr);  
 printf("float値: %.2f (アドレス: %p)\n", \*float\_ptr, (void\*)float\_ptr);  
 printf("double値: %.3f (アドレス: %p)\n", \*double\_ptr, (void\*)double\_ptr);  
   
 printf("\n===== ポインタのサイズ =====\n");  
 printf("int\*のサイズ: %lu バイト\n", (unsigned long)sizeof(int\*));  
 printf("char\*のサイズ: %lu バイト\n", (unsigned long)sizeof(char\*));  
 printf("float\*のサイズ: %lu バイト\n", (unsigned long)sizeof(float\*));  
 printf("double\*のサイズ: %lu バイト\n", (unsigned long)sizeof(double\*));  
   
 printf("\n===== NULLポインタ =====\n");  
 null\_ptr = NULL; /\* C90: 初期化は宣言後に \*/  
 printf("NULLポインタの値: %p\n", (void\*)null\_ptr);  
   
 if (null\_ptr == NULL) {  
 printf("null\_ptrはNULLです（安全チェック）\n");  
 }  
   
 /\* NULLポインタの参照は危険なのでコメントアウト \*/  
 /\* printf("NULL参照: %d\n", \*null\_ptr); // 危険！ \*/  
   
 printf("\n===== void\*ポインタ（汎用ポインタ） =====\n");  
   
 generic\_ptr = &int\_val;  
 printf("intを指すvoid\*: %d\n", \*(int\*)generic\_ptr);  
   
 generic\_ptr = &double\_val;  
 printf("doubleを指すvoid\*: %.2f\n", \*(double\*)generic\_ptr);  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. ポインタの宣言: int \*ptr;  
2. アドレス演算子: &variable  
3. 間接参照演算子: \*pointer  
4. ポインタの初期化と再代入  
5. NULLポインタの安全な使用  
6. void\*ポインタと型キャスト  
7. ポインタのサイズは型に関係なく一定  
8. 安全なプログラミングのためのNULLチェック  
  
注意点:  
- 未初期化ポインタの使用は危険  
- NULLポインタの参照は避ける  
- void\*ポインタは使用時に適切な型キャストが必要  
\*/

C99版は[こちら](pointer_basic_c99.c)を参照してください。

### pointer\_functions.c

/\*  
 \* pointer\_functions.c  
 \* 関数引数、戻り値としてのポインタの使用方法を学習  
 \* C90準拠  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
  
/\* 2つの値を交換する関数 \*/  
void swap\_values(int \*a, int \*b)  
{  
 int temp;  
   
 printf(" swap関数内: 交換前 \*a = %d, \*b = %d\n", \*a, \*b);  
   
 temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
   
 printf(" swap関数内: 交換後 \*a = %d, \*b = %d\n", \*a, \*b);  
}  
  
/\* 値を2倍にする関数 \*/  
void double\_value(int \*value)  
{  
 printf(" double\_value関数内: 変更前 \*value = %d\n", \*value);  
 \*value = (\*value) \* 2;  
 printf(" double\_value関数内: 変更後 \*value = %d\n", \*value);  
}  
  
/\* 配列の合計を計算する関数 \*/  
int sum\_array(int \*arr, int size)  
{  
 int sum = 0;  
 int i;  
   
 for (i = 0; i < size; i++) {  
 sum += \*(arr + i); /\* ポインタ記法 \*/  
 }  
   
 return sum;  
}  
  
/\* 配列の要素を変更する関数 \*/  
void modify\_array(int \*arr, int size, int multiplier)  
{  
 int i;  
   
 printf(" modify\_array関数内: 変更前\n ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
   
 for (i = 0; i < size; i++) {  
 arr[i] \*= multiplier;  
 }  
   
 printf(" modify\_array関数内: 変更後（%d倍）\n ", multiplier);  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
/\* 配列から最大値のポインタを返す関数 \*/  
int\* find\_max\_pointer(int \*arr, int size)  
{  
 int \*max\_ptr = arr; /\* 最初の要素を最大とする \*/  
 int i;  
   
 for (i = 1; i < size; i++) {  
 if (\*(arr + i) > \*max\_ptr) {  
 max\_ptr = arr + i;  
 }  
 }  
   
 return max\_ptr;  
}  
  
/\* 配列から指定値を検索してポインタを返す関数 \*/  
int\* search\_value(int \*arr, int size, int target)  
{  
 int i;  
   
 for (i = 0; i < size; i++) {  
 if (\*(arr + i) == target) {  
 return arr + i; /\* 見つかった要素のポインタを返す \*/  
 }  
 }  
   
 return NULL; /\* 見つからない場合はNULLを返す \*/  
}  
  
/\* 文字列の長さを計算する関数（ポインタ版） \*/  
int string\_length\_ptr(char \*str)  
{  
 int length = 0;  
   
 while (\*str != '\0') {  
 length++;  
 str++;  
 }  
   
 return length;  
}  
  
/\* 文字列をコピーする関数（ポインタ版） \*/  
void string\_copy\_ptr(char \*dest, char \*src)  
{  
 while (\*src != '\0') {  
 \*dest = \*src;  
 dest++;  
 src++;  
 }  
 \*dest = '\0'; /\* null終端文字を追加 \*/  
}  
  
/\* 配列を初期化する関数 \*/  
void init\_array(int \*arr, int size, int init\_value)  
{  
 int i;  
   
 for (i = 0; i < size; i++) {  
 \*(arr + i) = init\_value + i; /\* 初期値 + インデックス \*/  
 }  
}  
  
/\* 配列を表示する関数 \*/  
void print\_array(char \*label, int \*arr, int size)  
{  
 int i;  
   
 printf("%s: ", label);  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 /\* C90: 全変数宣言を先頭に配置 \*/  
 int x = 10, y = 20;  
 int numbers[6];  
 int size;  
 int total;  
 int data[] = {45, 67, 23, 89, 12, 56, 78};  
 int data\_size;  
 int \*result\_ptr;  
 int search\_target = 56;  
 char original[] = "Hello, World!";  
 char copy\_buffer[50];  
 char \*fruits[] = {"Apple", "Banana", "Cherry", "Date"};  
 int fruit\_count;  
 int i;  
 int \*arrays[2]; /\* C90では可変初期化ができないため配列サイズを固定 \*/  
 int sizes[2];  
 char \*labels[] = {"numbers配列", "data配列"};  
 int array\_sum;  
   
 printf("===== ポインタを引数とする関数 =====\n");  
   
 /\* 変数の初期化 \*/  
 size = sizeof(numbers) / sizeof(numbers[0]);  
 data\_size = sizeof(data) / sizeof(data[0]);  
 fruit\_count = sizeof(fruits) / sizeof(fruits[0]);  
   
 /\* swap関数のテスト \*/  
 printf("\nswap関数のテスト:\n");  
 printf("交換前: x = %d, y = %d\n", x, y);  
 swap\_values(&x, &y);  
 printf("交換後: x = %d, y = %d\n", x, y);  
   
 /\* double\_value関数のテスト \*/  
 printf("\ndouble\_value関数のテスト:\n");  
 printf("2倍前: x = %d\n", x);  
 double\_value(&x);  
 printf("2倍後: x = %d\n", x);  
   
 printf("\n===== 配列を操作する関数 =====\n");  
   
 /\* 配列の初期化 \*/  
 init\_array(numbers, size, 10);  
 print\_array("初期化後", numbers, size);  
   
 /\* 配列の合計計算 \*/  
 total = sum\_array(numbers, size);  
 printf("配列の合計: %d\n", total);  
   
 /\* 配列の変更 \*/  
 printf("\n配列変更のテスト:\n");  
 modify\_array(numbers, size, 3);  
 print\_array("main関数内での変更後", numbers, size);  
   
 printf("\n===== ポインタを戻り値とする関数 =====\n");  
   
 print\_array("検索対象配列", data, data\_size);  
   
 /\* 最大値の検索 \*/  
 result\_ptr = find\_max\_pointer(data, data\_size);  
 if (result\_ptr != NULL) {  
 printf("最大値: %d (アドレス: %p)\n", \*result\_ptr, (void\*)result\_ptr);  
 printf("最大値のインデックス: %ld\n", result\_ptr - data);  
 }  
   
 /\* 特定値の検索 \*/  
 result\_ptr = search\_value(data, data\_size, search\_target);  
 if (result\_ptr != NULL) {  
 printf("%d が見つかりました (アドレス: %p)\n",   
 \*result\_ptr, (void\*)result\_ptr);  
 printf("見つかった位置のインデックス: %ld\n", result\_ptr - data);  
   
 /\* 見つかった値を変更 \*/  
 \*result\_ptr = 999;  
 print\_array("値変更後", data, data\_size);  
 } else {  
 printf("%d は見つかりませんでした\n", search\_target);  
 }  
   
 printf("\n===== 文字列操作関数 =====\n");  
   
 printf("元の文字列: \"%s\"\n", original);  
 printf("文字列の長さ: %d文字\n", string\_length\_ptr(original));  
   
 /\* 文字列のコピー \*/  
 string\_copy\_ptr(copy\_buffer, original);  
 printf("コピーした文字列: \"%s\"\n", copy\_buffer);  
   
 printf("\n===== ポインタによる高度な操作 =====\n");  
   
 /\* ポインタの配列を使った複数の文字列 \*/  
   
 printf("果物リスト:\n");  
 for (i = 0; i < fruit\_count; i++) {  
 printf(" %d. %s (長さ: %d文字)\n",   
 i + 1, fruits[i], string\_length\_ptr(fruits[i]));  
 }  
   
 /\* 複数の配列を処理 \*/  
 /\* C90: 配列初期化は宣言時のみ可能なため、手動で代入 \*/  
 arrays[0] = numbers;  
 arrays[1] = data;  
 sizes[0] = size;  
 sizes[1] = data\_size;  
   
 printf("\n複数配列の合計:\n");  
 for (i = 0; i < 2; i++) {  
 array\_sum = sum\_array(arrays[i], sizes[i]);  
 printf(" %s の合計: %d\n", labels[i], array\_sum);  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 関数引数としてのポインタ:  
 - 値の変更が可能（参照渡し）  
 - 配列の受け渡し  
 - NULLチェックの重要性  
  
2. 戻り値としてのポインタ:  
 - 配列要素のアドレス返し  
 - NULLによるエラー表現  
 - ローカル変数のアドレス返しは危険  
  
3. 実用的なポインタ活用:  
 - 配列操作の効率化  
 - 文字列処理  
 - データ検索と変更  
 - 複数データの一括処理  
  
4. メモリ安全性:  
 - 境界チェック  
 - NULLポインタチェック  
 - 適切な初期化  
  
注意点:  
- ローカル変数のアドレスを返してはいけない  
- 配列の範囲外アクセスに注意  
- NULLポインタの参照は避ける  
- ポインタ演算の境界を意識する  
\*/

C99版は[こちら](pointer_functions_c99.c)を参照してください。

# 第11章: 構造体

## 対応C規格

* **主要対象:** C90
* **学習内容:** 構造体の基本、構造体ポインタ、構造体配列、->演算子、メモリ管理

## 学習目標

この章を完了すると、以下のことができるようになります：

* 構造体の基本概念を理解する
* 構造体ポインタを効果的に使える
* ->演算子と.演算子の使い分けができる
* 構造体配列とポインタ配列を活用できる
* 構造体を使ったデータ管理ができる

## 概要と詳細

### 構造体とは？

構造体は、複数の異なるデータ型をまとめて1つの新しいデータ型を作る機能です。現実世界の「モノ」をプログラムで表現するのに最適です！

#### 日常生活での構造体

身の回りにあるものを考えてみましょう：

1. **学生証の情報**
   * 学籍番号（整数）
   * 名前（文字列）
   * 学部（文字列）
   * 入学年（整数）
2. **商品の情報**
   * 商品コード（整数）
   * 商品名（文字列）
   * 価格（小数）
   * 在庫数（整数）

これらの「関連する情報の集まり」を1つにまとめるのが構造体です！

### なぜ構造体が必要なのか？

構造体を使わないと、関連するデータをバラバラに管理することになります：

/\* 構造体を使わない場合（管理が大変！） \*/  
int student\_id[100];  
char student\_name[100][50];  
int student\_age[100];  
  
/\* 構造体を使う場合（スッキリ！） \*/  
struct Student students[100];

### 構造体の基本概念

構造体は異なるデータ型を組み合わせて新しいデータ型を作る仕組みです。

#### 構造体の作り方の3ステップ

1. **定義する**：どんな情報を含むか決める
2. **宣言する**：構造体の変数を作る
3. **使う**：メンバーにアクセスする

#### 構造体の定義と宣言

#include <stdio.h>  
#include <string.h>  
  
/\* 学生情報を表す構造体 \*/  
struct Student {  
 int id;  
 char name[50];  
 int age;  
 double gpa;  
};  
  
int main(void)  
{  
 struct Student student;  
   
 /\* メンバーに値を代入 \*/  
 student.id = 1001;  
 strcpy(student.name, "田中太郎");  
 student.age = 20;  
 student.gpa = 3.5;  
   
 /\* メンバーの値を表示 \*/  
 printf("学生ID: %d\n", student.id);  
 printf("名前: %s\n", student.name);  
 printf("年齢: %d\n", student.age);  
 printf("GPA: %.1f\n", student.gpa);  
   
 return 0;  
}

#### 構造体の初期化

#include <stdio.h>  
  
struct Point {  
 int x;  
 int y;  
};  
  
struct Rectangle {  
 struct Point top\_left;  
 struct Point bottom\_right;  
};  
  
int main(void)  
{  
 /\* 初期化方法1: 順序指定 \*/  
 struct Point p1 = {10, 20};  
   
 /\* 初期化方法2: 部分初期化 \*/  
 struct Point p2 = {5}; /\* x=5, y=0 \*/  
   
 /\* 初期化方法3: ネストした構造体 \*/  
 struct Rectangle rect = {{0, 0}, {100, 50}};  
   
 /\* 初期化方法4: 個別代入 \*/  
 struct Point p3;  
 p3.x = 30;  
 p3.y = 40;  
   
 printf("p1: (%d, %d)\n", p1.x, p1.y);  
 printf("p2: (%d, %d)\n", p2.x, p2.y);  
 printf("p3: (%d, %d)\n", p3.x, p3.y);  
 printf("矩形: (%d,%d) から (%d,%d)\n",   
 rect.top\_left.x, rect.top\_left.y,  
 rect.bottom\_right.x, rect.bottom\_right.y);  
   
 return 0;  
}

### 構造体ポインタの基本

構造体もポインタで操作できます。大きな構造体を関数に渡す際に効率的です。

#### なぜ構造体ポインタを使うのか？

1. **メモリ効率**
   * 構造体全体をコピーするより、アドレスだけ渡す方が高速
   * 特に大きな構造体では効果的
2. **関数での値の変更**
   * ポインタなら関数内で元の構造体を変更できる
3. **動的メモリ管理**
   * 実行時に必要な数だけ構造体を作成可能

#### 構造体ポインタの宣言と使用

#include <stdio.h>  
#include <string.h>  
  
struct Book {  
 char title[100];  
 char author[50];  
 int pages;  
 double price;  
};  
  
int main(void)  
{  
 struct Book book;  
 struct Book \*book\_ptr;  
   
 /\* 構造体ポインタを構造体のアドレスに設定 \*/  
 book\_ptr = &book;  
   
 /\* ポインタを通じてメンバーにアクセス（方法1: (\*ptr).member） \*/  
 strcpy((\*book\_ptr).title, "C言語入門");  
 strcpy((\*book\_ptr).author, "山田花子");  
 (\*book\_ptr).pages = 350;  
 (\*book\_ptr).price = 2800.0;  
   
 /\* 直接アクセスとポインタアクセスの比較 \*/  
 printf("=== 直接アクセス ===\n");  
 printf("書名: %s\n", book.title);  
 printf("著者: %s\n", book.author);  
 printf("ページ数: %d\n", book.pages);  
 printf("価格: %.0f円\n", book.price);  
   
 printf("\n=== ポインタアクセス ===\n");  
 printf("書名: %s\n", (\*book\_ptr).title);  
 printf("著者: %s\n", (\*book\_ptr).author);  
 printf("ページ数: %d\n", (\*book\_ptr).pages);  
 printf("価格: %.0f円\n", (\*book\_ptr).price);  
   
 return 0;  
}

### アロー演算子（->）の使用

構造体ポインタには便利なアロー演算子（->）が用意されています。

#### アロー演算子の覚え方

構造体変数 → . (ドット)を使う  
構造体ポインタ → -> (矢印)を使う

**記憶のコツ**：ポインタは「矢印」で指すから「->」（矢印記号）を使う！

#### ->演算子と.演算子の比較

#include <stdio.h>  
#include <string.h>  
  
struct Employee {  
 int id;  
 char name[50];  
 char department[30];  
 double salary;  
};  
  
void print\_employee\_info(struct Employee \*emp)  
{  
 printf("=== 社員情報 ===\n");  
 printf("ID: %d\n", emp->id); /\* emp->id は (\*emp).id と同じ \*/  
 printf("名前: %s\n", emp->name); /\* emp->name は (\*emp).name と同じ \*/  
 printf("部署: %s\n", emp->department); /\* emp->department は (\*emp).department と同じ \*/  
 printf("給与: %.0f円\n", emp->salary); /\* emp->salary は (\*emp).salary と同じ \*/  
}  
  
int main(void)  
{  
 struct Employee emp;  
 struct Employee \*emp\_ptr = &emp;  
   
 /\* ->演算子を使用してメンバーに値を設定 \*/  
 emp\_ptr->id = 2001;  
 strcpy(emp\_ptr->name, "佐藤次郎");  
 strcpy(emp\_ptr->department, "開発部");  
 emp\_ptr->salary = 350000.0;  
   
 /\* 3つの方法でアクセス \*/  
 printf("=== 直接アクセス（.演算子） ===\n");  
 printf("ID: %d\n", emp.id);  
   
 printf("\n=== ポインタ経由（(\*ptr).member） ===\n");  
 printf("ID: %d\n", (\*emp\_ptr).id);  
   
 printf("\n=== ポインタ経由（->演算子） ===\n");  
 printf("ID: %d\n", emp\_ptr->id);  
   
 printf("\n");  
 print\_employee\_info(emp\_ptr);  
   
 return 0;  
}

#### ->演算子の実践例

#include <stdio.h>  
#include <string.h>  
  
struct Car {  
 char brand[30];  
 char model[30];  
 int year;  
 double mileage;  
 int is\_electric;  
};  
  
/\* 車の情報を更新する関数 \*/  
void update\_mileage(struct Car \*car, double new\_mileage)  
{  
 if (car != NULL && new\_mileage >= car->mileage) {  
 car->mileage = new\_mileage;  
 printf("%s %s の走行距離を %.1f km に更新しました\n",   
 car->brand, car->model, car->mileage);  
 }  
}  
  
/\* 車の詳細情報を表示する関数 \*/  
void display\_car\_details(struct Car \*car)  
{  
 if (car == NULL) {  
 printf("無効な車両データです\n");  
 return;  
 }  
   
 printf("=== 車両情報 ===\n");  
 printf("ブランド: %s\n", car->brand);  
 printf("モデル: %s\n", car->model);  
 printf("年式: %d年\n", car->year);  
 printf("走行距離: %.1f km\n", car->mileage);  
 printf("電気自動車: %s\n", car->is\_electric ? "はい" : "いいえ");  
 printf("\n");  
}  
  
int main(void)  
{  
 struct Car my\_car;  
 struct Car \*car\_ptr = &my\_car;  
   
 /\* ->演算子で車の初期設定 \*/  
 strcpy(car\_ptr->brand, "Toyota");  
 strcpy(car\_ptr->model, "Prius");  
 car\_ptr->year = 2020;  
 car\_ptr->mileage = 15000.0;  
 car\_ptr->is\_electric = 0; /\* ハイブリッド車 \*/  
   
 display\_car\_details(car\_ptr);  
   
 /\* 走行距離を更新 \*/  
 update\_mileage(car\_ptr, 18500.0);  
   
 display\_car\_details(car\_ptr);  
   
 return 0;  
}

### 構造体配列とポインタ

構造体の配列とそのポインタ操作について学習します。

#### 構造体配列の使いどころ

構造体配列は、同じ種類のデータを複数管理する場合に便利です：

* **学生名簿**：100人の学生情報
* **商品在庫**：1000個の商品データ
* **座標点**：図形を構成する頂点の集まり

/\* 1人分 \*/  
struct Student student;  
  
/\* 100人分 \*/  
struct Student students[100];

#### 構造体配列の基本操作

#include <stdio.h>  
#include <string.h>  
  
struct Product {  
 int code;  
 char name[50];  
 double price;  
 int stock;  
};  
  
void print\_product(struct Product \*prod)  
{  
 printf("商品コード: %d\n", prod->code);  
 printf("商品名: %s\n", prod->name);  
 printf("価格: %.0f円\n", prod->price);  
 printf("在庫: %d個\n", prod->stock);  
 printf("---\n");  
}  
  
void print\_all\_products(struct Product products[], int count)  
{  
 int i;  
 printf("=== 全商品リスト ===\n");  
 for (i = 0; i < count; i++) {  
 print\_product(&products[i]); /\* 配列要素のアドレスを渡す \*/  
 }  
}  
  
int main(void)  
{  
 struct Product inventory[3];  
 int i;  
   
 /\* 商品データの初期化 \*/  
 inventory[0] = (struct Product){101, "ノートPC", 98000, 5};  
   
 strcpy(inventory[1].name, "マウス");  
 inventory[1].code = 102;  
 inventory[1].price = 2980;  
 inventory[1].stock = 20;  
   
 inventory[2].code = 103;  
 strcpy(inventory[2].name, "キーボード");  
 inventory[2].price = 4500;  
 inventory[2].stock = 15;  
   
 print\_all\_products(inventory, 3);  
   
 /\* ポインタを使った配列操作 \*/  
 printf("\n=== ポインタを使った操作 ===\n");  
 struct Product \*ptr = inventory; /\* 配列の先頭要素を指す \*/  
   
 for (i = 0; i < 3; i++) {  
 printf("%d番目: %s (%.0f円)\n",   
 i + 1, (ptr + i)->name, (ptr + i)->price);  
 }  
   
 return 0;  
}

#### ポインタ演算による構造体配列操作

#include <stdio.h>  
#include <string.h>  
  
struct Score {  
 char subject[30];  
 int points;  
 char grade;  
};  
  
/\* 成績を計算してグレードを設定 \*/  
void calculate\_grade(struct Score \*score)  
{  
 if (score->points >= 90) {  
 score->grade = 'A';  
 } else if (score->points >= 80) {  
 score->grade = 'B';  
 } else if (score->points >= 70) {  
 score->grade = 'C';  
 } else if (score->points >= 60) {  
 score->grade = 'D';  
 } else {  
 score->grade = 'F';  
 }  
}  
  
/\* 配列内の最高得点を見つける \*/  
struct Score\* find\_best\_score(struct Score scores[], int count)  
{  
 struct Score \*best = &scores[0];  
 int i;  
   
 for (i = 1; i < count; i++) {  
 if (scores[i].points > best->points) {  
 best = &scores[i];  
 }  
 }  
   
 return best;  
}  
  
int main(void)  
{  
 struct Score my\_scores[4];  
 struct Score \*ptr;  
 struct Score \*best;  
 int i;  
   
 /\* 成績データの入力 \*/  
 strcpy(my\_scores[0].subject, "数学");  
 my\_scores[0].points = 85;  
   
 strcpy(my\_scores[1].subject, "英語");  
 my\_scores[1].points = 92;  
   
 strcpy(my\_scores[2].subject, "理科");  
 my\_scores[2].points = 78;  
   
 strcpy(my\_scores[3].subject, "社会");  
 my\_scores[3].points = 88;  
   
 /\* 各科目のグレードを計算 \*/  
 ptr = my\_scores; /\* 配列の先頭を指すポインタ \*/  
 for (i = 0; i < 4; i++) {  
 calculate\_grade(ptr + i); /\* ポインタ演算でi番目の要素 \*/  
 }  
   
 /\* 結果表示 \*/  
 printf("=== 成績表 ===\n");  
 for (i = 0; i < 4; i++) {  
 printf("%s: %d点 (評価: %c)\n",   
 my\_scores[i].subject,   
 my\_scores[i].points,   
 my\_scores[i].grade);  
 }  
   
 /\* 最高得点の科目を見つける \*/  
 best = find\_best\_score(my\_scores, 4);  
 printf("\n最高得点: %s (%d点)\n", best->subject, best->points);  
   
 return 0;  
}

### 構造体ポインタ配列

ポインタの配列で複数の構造体を効率的に管理します。

#### 構造体ポインタ配列の基本

#include <stdio.h>  
#include <string.h>  
  
struct Person {  
 char name[50];  
 int age;  
 char city[30];  
};  
  
void print\_person\_info(struct Person \*person)  
{  
 printf("名前: %s, 年齢: %d歳, 居住地: %s\n",   
 person->name, person->age, person->city);  
}  
  
int main(void)  
{  
 /\* 個別の構造体変数 \*/  
 struct Person person1 = {"山田太郎", 25, "東京"};  
 struct Person person2 = {"佐藤花子", 30, "大阪"};  
 struct Person person3 = {"田中次郎", 28, "名古屋"};  
 struct Person person4 = {"鈴木美咲", 22, "福岡"};  
   
 /\* 構造体ポインタの配列 \*/  
 struct Person \*people[4];  
 int i;  
   
 /\* ポインタ配列に各構造体のアドレスを設定 \*/  
 people[0] = &person1;  
 people[1] = &person2;  
 people[2] = &person3;  
 people[3] = &person4;  
   
 printf("=== 全員の情報 ===\n");  
 for (i = 0; i < 4; i++) {  
 printf("%d. ", i + 1);  
 print\_person\_info(people[i]);  
 }  
   
 /\* 特定の条件で検索 \*/  
 printf("\n=== 25歳未満の人 ===\n");  
 for (i = 0; i < 4; i++) {  
 if (people[i]->age < 25) {  
 print\_person\_info(people[i]);  
 }  
 }  
   
 return 0;  
}

#### 動的な構造体ポインタ配列

#include <stdio.h>  
#include <string.h>  
  
struct Student {  
 int id;  
 char name[50];  
 double gpa;  
};  
  
/\* 学生データを作成する関数 \*/  
struct Student create\_student(int id, const char\* name, double gpa)  
{  
 struct Student student;  
 student.id = id;  
 strcpy(student.name, name);  
 student.gpa = gpa;  
 return student;  
}  
  
/\* GPA順でソート（バブルソート） \*/  
void sort\_students\_by\_gpa(struct Student \*students[], int count)  
{  
 int i, j;  
 struct Student \*temp;  
   
 for (i = 0; i < count - 1; i++) {  
 for (j = 0; j < count - 1 - i; j++) {  
 if (students[j]->gpa < students[j + 1]->gpa) {  
 temp = students[j];  
 students[j] = students[j + 1];  
 students[j + 1] = temp;  
 }  
 }  
 }  
}  
  
/\* 学生一覧を表示 \*/  
void display\_students(struct Student \*students[], int count)  
{  
 int i;  
 printf("ID\t名前\t\tGPA\n");  
 printf("------------------------\n");  
 for (i = 0; i < count; i++) {  
 printf("%d\t%s\t\t%.1f\n",   
 students[i]->id,   
 students[i]->name,   
 students[i]->gpa);  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 /\* 学生データの作成 \*/  
 struct Student student1 = create\_student(1001, "田中太郎", 3.2);  
 struct Student student2 = create\_student(1002, "佐藤花子", 3.8);  
 struct Student student3 = create\_student(1003, "山田次郎", 2.9);  
 struct Student student4 = create\_student(1004, "鈴木美咲", 3.5);  
 struct Student student5 = create\_student(1005, "高橋健太", 3.1);  
   
 /\* 構造体ポインタ配列 \*/  
 struct Student \*class\_roster[] = {  
 &student1, &student2, &student3, &student4, &student5  
 };  
   
 int class\_size = sizeof(class\_roster) / sizeof(class\_roster[0]);  
   
 printf("=== 入学順（元の順序） ===\n");  
 display\_students(class\_roster, class\_size);  
   
 /\* GPA順でソート \*/  
 sort\_students\_by\_gpa(class\_roster, class\_size);  
   
 printf("=== GPA順（高い順） ===\n");  
 display\_students(class\_roster, class\_size);  
   
 /\* 優秀な学生を見つける \*/  
 printf("=== GPA 3.5以上の学生 ===\n");  
 int i;  
 for (i = 0; i < class\_size; i++) {  
 if (class\_roster[i]->gpa >= 3.5) {  
 printf("%s (GPA: %.1f)\n",   
 class\_roster[i]->name,   
 class\_roster[i]->gpa);  
 }  
 }  
   
 return 0;  
}

### ネストした構造体とポインタ

構造体の中に他の構造体を含む場合のポインタ操作について学習します。

#### ネスト構造の現実例

現実世界では、情報が階層的になることがよくあります：

**会社の組織**

会社  
 └─ 部署  
 └─ 社員  
 └─ 住所  
 └─ 連絡先

このような階層構造を構造体のネストで表現できます！

#### ネストした構造体の基本

#include <stdio.h>  
#include <string.h>  
  
struct Address {  
 char street[100];  
 char city[50];  
 char postal\_code[20];  
};  
  
struct Contact {  
 char phone[20];  
 char email[50];  
};  
  
struct Employee {  
 int id;  
 char name[50];  
 struct Address address; /\* ネストした構造体 \*/  
 struct Contact contact; /\* ネストした構造体 \*/  
 double salary;  
};  
  
void print\_employee\_details(struct Employee \*emp)  
{  
 printf("=== 社員詳細情報 ===\n");  
 printf("ID: %d\n", emp->id);  
 printf("名前: %s\n", emp->name);  
 printf("住所: %s, %s %s\n",   
 emp->address.street,   
 emp->address.city,   
 emp->address.postal\_code);  
 printf("電話: %s\n", emp->contact.phone);  
 printf("メール: %s\n", emp->contact.email);  
 printf("給与: %.0f円\n", emp->salary);  
 printf("\n");  
}  
  
void update\_address(struct Employee \*emp, const char\* street,   
 const char\* city, const char\* postal\_code)  
{  
 strcpy(emp->address.street, street);  
 strcpy(emp->address.city, city);  
 strcpy(emp->address.postal\_code, postal\_code);  
 printf("%s の住所を更新しました\n", emp->name);  
}  
  
int main(void)  
{  
 struct Employee emp;  
 struct Employee \*emp\_ptr = &emp;  
   
 /\* 基本情報の設定 \*/  
 emp\_ptr->id = 5001;  
 strcpy(emp\_ptr->name, "田中太郎");  
 emp\_ptr->salary = 450000.0;  
   
 /\* ネストした構造体メンバーの設定 \*/  
 strcpy(emp\_ptr->address.street, "新宿区西新宿1-2-3");  
 strcpy(emp\_ptr->address.city, "東京都");  
 strcpy(emp\_ptr->address.postal\_code, "160-0023");  
   
 strcpy(emp\_ptr->contact.phone, "03-1234-5678");  
 strcpy(emp\_ptr->contact.email, "tanaka@company.co.jp");  
   
 print\_employee\_details(emp\_ptr);  
   
 /\* 住所更新 \*/  
 update\_address(emp\_ptr, "渋谷区渋谷2-3-4", "東京都", "150-0002");  
   
 print\_employee\_details(emp\_ptr);  
   
 return 0;  
}

#### 構造体ポインタのネスト

#include <stdio.h>  
#include <string.h>  
  
struct Engine {  
 char type[50];  
 double displacement;  
 int horsepower;  
};  
  
struct Car {  
 char model[50];  
 int year;  
 struct Engine \*engine; /\* エンジン情報へのポインタ \*/  
 double price;  
};  
  
struct Dealership {  
 char name[50];  
 struct Car \*inventory; /\* 車の在庫配列へのポインタ \*/  
 int car\_count;  
};  
  
void print\_car\_info(struct Car \*car)  
{  
 printf("=== 車両情報 ===\n");  
 printf("モデル: %s (%d年)\n", car->model, car->year);  
 if (car->engine != NULL) {  
 printf("エンジン: %s %.1fL %dHP\n",   
 car->engine->type,   
 car->engine->displacement,   
 car->engine->horsepower);  
 }  
 printf("価格: %.0f万円\n", car->price);  
 printf("\n");  
}  
  
void print\_dealership\_inventory(struct Dealership \*dealer)  
{  
 int i;  
 printf("=== %s の在庫 ===\n", dealer->name);  
 for (i = 0; i < dealer->car\_count; i++) {  
 printf("%d. %s (%.0f万円)\n",   
 i + 1,   
 (dealer->inventory + i)->model,   
 (dealer->inventory + i)->price);  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 /\* エンジン情報 \*/  
 struct Engine engine1 = {"V6ガソリン", 3.5, 280};  
 struct Engine engine2 = {"直4ハイブリッド", 1.8, 120};  
 struct Engine engine3 = {"V8ガソリン", 5.0, 450};  
   
 /\* 車両情報 \*/  
 struct Car cars[3] = {  
 {"セダンLX", 2023, &engine1, 450},  
 {"ハイブリッドEC", 2024, &engine2, 380},  
 {"スポーツST", 2023, &engine3, 850}  
 };  
   
 /\* ディーラー情報 \*/  
 struct Dealership dealer = {  
 "トーキョー自動車",  
 cars,  
 3  
 };  
   
 /\* ディーラーの在庫一覧 \*/  
 print\_dealership\_inventory(&dealer);  
   
 /\* 各車両の詳細情報 \*/  
 int i;  
 for (i = 0; i < dealer.car\_count; i++) {  
 print\_car\_info(dealer.inventory + i);  
 }  
   
 return 0;  
}

### 構造体とメモリ管理

構造体のメモリレイアウトとアライメントについて理解します。

#### メモリアライメントとは？

コンピュータはメモリを効率的に読み書きするため、データを特定の境界に揃えて配置します。これをアライメント（整列）と呼びます。

**例えば**： - intは4バイト境界に配置 - doubleは8バイト境界に配置

そのため、構造体のサイズは各メンバーのサイズの合計より大きくなることがあります！

#### 構造体のメモリサイズとアライメント

#include <stdio.h>  
  
struct Example1 {  
 char a; /\* 1バイト \*/  
 int b; /\* 4バイト \*/  
 char c; /\* 1バイト \*/  
};  
  
struct Example2 {  
 char a; /\* 1バイト \*/  
 char c; /\* 1バイト \*/  
 int b; /\* 4バイト \*/  
};  
  
struct Example3 {  
 double d; /\* 8バイト \*/  
 char a; /\* 1バイト \*/  
 int b; /\* 4バイト \*/  
};  
  
void analyze\_struct\_memory(void)  
{  
 struct Example1 ex1;  
 struct Example2 ex2;  
 struct Example3 ex3;  
   
 printf("=== 構造体メモリ分析 ===\n");  
   
 printf("Example1 (char, int, char):\n");  
 printf(" sizeof: %lu バイト\n", sizeof(struct Example1));  
 printf(" a のオフセット: %lu\n", (unsigned long)&ex1.a - (unsigned long)&ex1);  
 printf(" b のオフセット: %lu\n", (unsigned long)&ex1.b - (unsigned long)&ex1);  
 printf(" c のオフセット: %lu\n", (unsigned long)&ex1.c - (unsigned long)&ex1);  
 printf("\n");  
   
 printf("Example2 (char, char, int):\n");  
 printf(" sizeof: %lu バイト\n", sizeof(struct Example2));  
 printf(" a のオフセット: %lu\n", (unsigned long)&ex2.a - (unsigned long)&ex2);  
 printf(" c のオフセット: %lu\n", (unsigned long)&ex2.c - (unsigned long)&ex2);  
 printf(" b のオフセット: %lu\n", (unsigned long)&ex2.b - (unsigned long)&ex2);  
 printf("\n");  
   
 printf("Example3 (double, char, int):\n");  
 printf(" sizeof: %lu バイト\n", sizeof(struct Example3));  
 printf(" d のオフセット: %lu\n", (unsigned long)&ex3.d - (unsigned long)&ex3);  
 printf(" a のオフセット: %lu\n", (unsigned long)&ex3.a - (unsigned long)&ex3);  
 printf(" b のオフセット: %lu\n", (unsigned long)&ex3.b - (unsigned long)&ex3);  
 printf("\n");  
}  
  
int main(void)  
{  
 analyze\_struct\_memory();  
   
 printf("=== 基本データ型のサイズ ===\n");  
 printf("char: %lu バイト\n", sizeof(char));  
 printf("int: %lu バイト\n", sizeof(int));  
 printf("double: %lu バイト\n", sizeof(double));  
 printf("ポインタ: %lu バイト\n", sizeof(void\*));  
   
 return 0;  
}

#### 構造体の配列とポインタの効率性

#include <stdio.h>  
#include <string.h>  
#include <time.h>  
  
struct LargeData {  
 char description[256];  
 double values[100];  
 int status;  
};  
  
/\* 値渡しで構造体を処理（非効率） \*/  
double process\_by\_value(struct LargeData data)  
{  
 int i;  
 double sum = 0.0;  
   
 for (i = 0; i < 100; i++) {  
 sum += data.values[i];  
 }  
   
 return sum / 100.0; /\* 平均値 \*/  
}  
  
/\* ポインタ渡しで構造体を処理（効率的） \*/  
double process\_by\_pointer(struct LargeData \*data)  
{  
 int i;  
 double sum = 0.0;  
   
 for (i = 0; i < 100; i++) {  
 sum += data->values[i];  
 }  
   
 return sum / 100.0; /\* 平均値 \*/  
}  
  
/\* 構造体にテストデータを設定 \*/  
void setup\_test\_data(struct LargeData \*data)  
{  
 int i;  
 strcpy(data->description, "テストデータセット");  
   
 for (i = 0; i < 100; i++) {  
 data->values[i] = (double)(i + 1) \* 1.5;  
 }  
   
 data->status = 1;  
}  
  
int main(void)  
{  
 struct LargeData test\_data;  
 clock\_t start, end;  
 double result;  
 int iterations = 10000;  
 int i;  
   
 setup\_test\_data(&test\_data);  
   
 printf("=== 構造体サイズ情報 ===\n");  
 printf("LargeData のサイズ: %lu バイト\n", sizeof(struct LargeData));  
 printf("テスト回数: %d 回\n\n", iterations);  
   
 /\* 値渡しのテスト \*/  
 printf("値渡しでのテスト開始...\n");  
 start = clock();  
 for (i = 0; i < iterations; i++) {  
 result = process\_by\_value(test\_data);  
 }  
 end = clock();  
 printf("値渡し結果: %.1f\n", result);  
 printf("実行時間: %.3f 秒\n\n", (double)(end - start) / CLOCKS\_PER\_SEC);  
   
 /\* ポインタ渡しのテスト \*/  
 printf("ポインタ渡しでのテスト開始...\n");  
 start = clock();  
 for (i = 0; i < iterations; i++) {  
 result = process\_by\_pointer(&test\_data);  
 }  
 end = clock();  
 printf("ポインタ渡し結果: %.1f\n", result);  
 printf("実行時間: %.3f 秒\n", (double)(end - start) / CLOCKS\_PER\_SEC);  
   
 return 0;  
}

### 構造体を使う際の重要なポイント

初心者が覚えておくべき構造体の重要な点をまとめます。

#### 1. メンバーアクセスの使い分け

struct Point p;  
struct Point \*ptr = &p;  
  
/\* 構造体変数の場合：ドット(.) \*/  
p.x = 10;  
  
/\* 構造体ポインタの場合：アロー(->) \*/  
ptr->x = 10;

#### 2. 構造体の初期化パターン

/\* パターン1：宣言時に初期化 \*/  
struct Point p1 = {10, 20};  
  
/\* パターン2：個別に代入 \*/  
struct Point p2;  
p2.x = 10;  
p2.y = 20;  
  
/\* パターン3：関数で作成 \*/  
struct Point create\_point(int x, int y) {  
 struct Point p = {x, y};  
 return p;  
}

#### 3. 関数への渡し方

/\* 小さい構造体：値渡しでOK \*/  
void print\_point(struct Point p) {  
 printf("(%d, %d)\n", p.x, p.y);  
}  
  
/\* 大きい構造体：ポインタ渡しが効率的 \*/  
void print\_large\_data(struct LargeData \*data) {  
 /\* 処理 \*/  
}

#### 4. よくある間違いと対策

**間違い1：構造体の代入**

/\* 間違い：文字配列は代入できない \*/  
struct Student s;  
s.name = "太郎"; /\* エラー！ \*/  
  
/\* 正しい：strcpyを使う \*/  
strcpy(s.name, "太郎");

**間違い2：ポインタの初期化忘れ**

/\* 間違い：初期化なし \*/  
struct Point \*ptr;  
ptr->x = 10; /\* 危険！ \*/  
  
/\* 正しい：アドレスを設定 \*/  
struct Point p;  
struct Point \*ptr = &p;  
ptr->x = 10;

### 構造体を学ぶステップ

1. **基本から始める**
   * 単純な構造体（2-3個のメンバー）から練習
   * ドット演算子でのアクセスに慣れる
2. **ポインタとの組み合わせ**
   * アロー演算子の使い方を覚える
   * 関数への渡し方を練習
3. **実用的な例で練習**
   * 学生管理システム
   * 商品在庫管理
   * 座標計算プログラム
4. **高度な使い方へ**
   * ネストした構造体
   * 構造体の配列
   * 動的メモリ確保（次章以降）

### 実践的な構造体活用例

#### データベース風の学生管理システム

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_STUDENTS 100  
#define MAX\_NAME\_LEN 50  
  
struct Student {  
 int id;  
 char name[MAX\_NAME\_LEN];  
 int age;  
 double gpa;  
 char major[30];  
};  
  
struct StudentDatabase {  
 struct Student \*students[MAX\_STUDENTS];  
 int count;  
};  
  
/\* 学生データベースの初期化 \*/  
void init\_database(struct StudentDatabase \*db)  
{  
 int i;  
 db->count = 0;  
 for (i = 0; i < MAX\_STUDENTS; i++) {  
 db->students[i] = NULL;  
 }  
}  
  
/\* 学生を追加 \*/  
int add\_student(struct StudentDatabase \*db, struct Student \*student)  
{  
 if (db->count >= MAX\_STUDENTS) {  
 printf("エラー: データベースが満杯です\n");  
 return 0;  
 }  
   
 db->students[db->count] = student;  
 db->count++;  
 printf("学生 %s を追加しました (ID: %d)\n", student->name, student->id);  
 return 1;  
}  
  
/\* IDで学生を検索 \*/  
struct Student\* find\_student\_by\_id(struct StudentDatabase \*db, int id)  
{  
 int i;  
 for (i = 0; i < db->count; i++) {  
 if (db->students[i]->id == id) {  
 return db->students[i];  
 }  
 }  
 return NULL;  
}  
  
/\* 専攻で学生を検索 \*/  
void find\_students\_by\_major(struct StudentDatabase \*db, const char\* major)  
{  
 int i;  
 int found = 0;  
   
 printf("=== %s専攻の学生 ===\n", major);  
 for (i = 0; i < db->count; i++) {  
 if (strcmp(db->students[i]->major, major) == 0) {  
 printf("ID: %d, 名前: %s, GPA: %.1f\n",   
 db->students[i]->id,  
 db->students[i]->name,  
 db->students[i]->gpa);  
 found = 1;  
 }  
 }  
   
 if (!found) {  
 printf("該当する学生が見つかりませんでした\n");  
 }  
 printf("\n");  
}  
  
/\* 全学生の一覧表示 \*/  
void display\_all\_students(struct StudentDatabase \*db)  
{  
 int i;  
   
 printf("=== 全学生一覧 (%d名) ===\n", db->count);  
 printf("ID\t名前\t\t年齢\tGPA\t専攻\n");  
 printf("------------------------------------------------\n");  
   
 for (i = 0; i < db->count; i++) {  
 printf("%d\t%s\t\t%d\t%.1f\t%s\n",  
 db->students[i]->id,  
 db->students[i]->name,  
 db->students[i]->age,  
 db->students[i]->gpa,  
 db->students[i]->major);  
 }  
 printf("\n");  
}  
  
/\* 平均GPAを計算 \*/  
double calculate\_average\_gpa(struct StudentDatabase \*db)  
{  
 double total = 0.0;  
 int i;  
   
 if (db->count == 0) return 0.0;  
   
 for (i = 0; i < db->count; i++) {  
 total += db->students[i]->gpa;  
 }  
   
 return total / db->count;  
}  
  
int main(void)  
{  
 struct StudentDatabase db;  
 init\_database(&db);  
   
 /\* 学生データの作成 \*/  
 struct Student student1 = {2001, "田中太郎", 20, 3.2, "コンピュータ"};  
 struct Student student2 = {2002, "佐藤花子", 21, 3.8, "数学"};  
 struct Student student3 = {2003, "山田次郎", 19, 2.9, "コンピュータ"};  
 struct Student student4 = {2004, "鈴木美咲", 20, 3.5, "物理"};  
 struct Student student5 = {2005, "高橋健太", 22, 3.1, "数学"};  
   
 /\* データベースに学生を追加 \*/  
 add\_student(&db, &student1);  
 add\_student(&db, &student2);  
 add\_student(&db, &student3);  
 add\_student(&db, &student4);  
 add\_student(&db, &student5);  
   
 /\* 全学生表示 \*/  
 display\_all\_students(&db);  
   
 /\* 特定IDの学生を検索 \*/  
 struct Student \*found = find\_student\_by\_id(&db, 2003);  
 if (found != NULL) {  
 printf("ID 2003 の学生: %s (%s専攻)\n\n", found->name, found->major);  
 }  
   
 /\* 専攻別検索 \*/  
 find\_students\_by\_major(&db, "コンピュータ");  
 find\_students\_by\_major(&db, "数学");  
   
 /\* 統計情報 \*/  
 printf("=== 統計情報 ===\n");  
 printf("登録学生数: %d名\n", db.count);  
 printf("平均GPA: %.2f\n", calculate\_average\_gpa(&db));  
   
 return 0;  
}

## サンプルコード

### 構造体の基本操作

プログラムファイル: examples/struct\_basics.c

構造体の定義、初期化、メンバーアクセスの基本を学習します。

### 構造体ポインタと->演算子

プログラムファイル: examples/struct\_pointers.c

構造体ポインタと->演算子の使用方法を学習します。

### 構造体配列の操作

プログラムファイル: examples/struct\_arrays.c

構造体配列とポインタを使ったデータ管理を学習します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic struct\_basics.c -o struct\_basics  
  
# 実行  
./struct\_basics

## 演習課題

### 基礎問題

1. **学生情報管理**
   * 学生の情報（ID、名前、年齢、成績）を格納する構造体を定義し、ポインタを使って情報を表示・更新するプログラムを作成してください
2. **座標計算**
   * 2D座標を表す構造体を定義し、2点間の距離を計算する関数をポインタを使って実装してください
3. **商品管理**
   * 商品情報（コード、名前、価格、在庫）の構造体を作成し、構造体配列で複数商品を管理するプログラムを作成してください

### 応用問題

1. **従業員データベース**
   * 従業員情報と部署情報をネストした構造体で管理し、部署別の給与統計を算出するプログラムを作成してください
2. **図書管理システム**
   * 本の情報（タイトル、著者、出版年、貸出状況）を管理し、検索・貸出・返却機能を実装してください
3. **成績管理システム**
   * 学生と科目の構造体を使って、学生別・科目別の成績統計を管理するプログラムを作成してください

### 発展問題

1. **動的配列システム**
   * 構造体ポインタ配列を動的に拡張できるシステムを実装してください
2. **データソート**
   * 構造体ポインタ配列を複数の条件（名前、年齢、成績など）でソートできるプログラムを作成してください
3. **階層データ構造**
   * 会社組織（部署→チーム→従業員）のような階層構造を構造体とポインタで表現してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make struct\_basics  
  
# 実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* 構造体の初期化は順序通りのみ
* 可変長配列メンバーは不可
* 匿名構造体・共用体は不可

### C99以降の拡張

* 指定イニシャライザが使用可能
* 可変長配列メンバー（C99）
* 匿名構造体・共用体（C11）

## よくある間違い

### 1. NULLポインタの参照

/\* NG: NULLポインタの参照 \*/  
struct Student \*student = NULL;  
printf("%s\n", student->name); /\* セグメンテーション違反 \*/  
  
/\* OK: NULLチェック \*/  
if (student != NULL) {  
 printf("%s\n", student->name);  
}

### 2. スタック上の構造体アドレスの返却

/\* NG: ローカル構造体のアドレス返し \*/  
struct Point\* create\_point(int x, int y) {  
 struct Point p = {x, y};  
 return &p; /\* 危険: ローカル変数のアドレス \*/  
}  
  
/\* OK: 静的変数または呼び出し元で用意 \*/  
void create\_point(struct Point \*p, int x, int y) {  
 p->x = x;  
 p->y = y;  
}

### 3. 構造体の比較

/\* NG: 構造体の直接比較 \*/  
struct Point p1 = {10, 20};  
struct Point p2 = {10, 20};  
if (p1 == p2) { /\* コンパイルエラー \*/  
}  
  
/\* OK: メンバーごとの比較 \*/  
if (p1.x == p2.x && p1.y == p2.y) {  
 printf("同じ座標です\n");  
}

## 次の章へ

構造体を理解したら、[関数ポインタ](../function-pointers/README.md) に進んでください。

## 参考資料

* [C言語構造体リファレンス](https://ja.cppreference.com/w/c/language/struct)
* [メモリアライメントの詳細](https://ja.cppreference.com/w/c/language/object)
* [構造体の初期化](https://ja.cppreference.com/w/c/language/struct_initialization) ## サンプルコード

### struct\_arrays.c

/\*  
 \* struct\_arrays.c - 構造体配列の操作  
 \*   
 \* このプログラムでは以下を学習します：  
 \* - 構造体配列の宣言と初期化  
 \* - 構造体配列とポインタ演算  
 \* - 構造体ポインタ配列  
 \* - 実践的なデータ管理  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
  
#define MAX\_STUDENTS 50  
#define MAX\_BOOKS 100  
#define MAX\_NAME\_LEN 50  
  
/\* 学生情報を表す構造体 \*/  
struct Student {  
 int id;  
 char name[MAX\_NAME\_LEN];  
 double scores[5]; /\* 5教科の点数 \*/  
 double average;  
 char grade;  
};  
  
/\* 書籍情報を表す構造体 \*/  
struct Book {  
 int isbn;  
 char title[100];  
 char author[50];  
 int year;  
 int is\_borrowed;  
 int borrower\_id; /\* 借りている学生のID \*/  
};  
  
/\* 図書館を表す構造体 \*/  
struct Library {  
 struct Book books[MAX\_BOOKS];  
 int book\_count;  
 struct Student \*students[MAX\_STUDENTS];  
 int student\_count;  
};  
  
/\* 学生の平均点を計算する関数 \*/  
void calculate\_average(struct Student \*student)  
{  
 int i;  
 double sum = 0.0;  
   
 for (i = 0; i < 5; i++) {  
 sum += student->scores[i];  
 }  
   
 student->average = sum / 5.0;  
   
 /\* 成績評価 \*/  
 if (student->average >= 90.0) {  
 student->grade = 'A';  
 } else if (student->average >= 80.0) {  
 student->grade = 'B';  
 } else if (student->average >= 70.0) {  
 student->grade = 'C';  
 } else if (student->average >= 60.0) {  
 student->grade = 'D';  
 } else {  
 student->grade = 'F';  
 }  
}  
  
/\* 学生情報を表示する関数 \*/  
void print\_student(const struct Student \*student)  
{  
 int i;  
   
 printf("ID: %d, 名前: %s\n", student->id, student->name);  
 printf("点数: ");  
 for (i = 0; i < 5; i++) {  
 printf("%.0f ", student->scores[i]);  
 }  
 printf("\n平均: %.1f, 評価: %c\n", student->average, student->grade);  
 printf("---\n");  
}  
  
/\* 成績順にソート（バブルソート） \*/  
void sort\_students\_by\_average(struct Student students[], int count)  
{  
 int i, j;  
 struct Student temp;  
   
 for (i = 0; i < count - 1; i++) {  
 for (j = 0; j < count - 1 - i; j++) {  
 if (students[j].average < students[j + 1].average) {  
 temp = students[j];  
 students[j] = students[j + 1];  
 students[j + 1] = temp;  
 }  
 }  
 }  
}  
  
/\* 書籍を追加する関数 \*/  
int add\_book(struct Library \*lib, int isbn, const char \*title,   
 const char \*author, int year)  
{  
 if (lib->book\_count >= MAX\_BOOKS) {  
 return 0; /\* 追加失敗 \*/  
 }  
   
 struct Book \*book = &lib->books[lib->book\_count];  
 book->isbn = isbn;  
 strcpy(book->title, title);  
 strcpy(book->author, author);  
 book->year = year;  
 book->is\_borrowed = 0;  
 book->borrower\_id = -1;  
   
 lib->book\_count++;  
 return 1; /\* 追加成功 \*/  
}  
  
/\* 書籍を検索する関数 \*/  
struct Book\* find\_book\_by\_isbn(struct Library \*lib, int isbn)  
{  
 int i;  
 for (i = 0; i < lib->book\_count; i++) {  
 if (lib->books[i].isbn == isbn) {  
 return &lib->books[i];  
 }  
 }  
 return NULL;  
}  
  
/\* 書籍を借りる関数 \*/  
int borrow\_book(struct Library \*lib, int isbn, int student\_id)  
{  
 struct Book \*book = find\_book\_by\_isbn(lib, isbn);  
   
 if (book == NULL) {  
 printf("ISBN %d の書籍が見つかりません\n", isbn);  
 return 0;  
 }  
   
 if (book->is\_borrowed) {  
 printf("「%s」は既に貸出中です\n", book->title);  
 return 0;  
 }  
   
 book->is\_borrowed = 1;  
 book->borrower\_id = student\_id;  
 printf("ID %d の学生が「%s」を借りました\n", student\_id, book->title);  
 return 1;  
}  
  
/\* 学生の借りている本を表示 \*/  
void show\_borrowed\_books(struct Library \*lib, int student\_id)  
{  
 int i;  
 int count = 0;  
   
 printf("\nID %d の学生が借りている本:\n", student\_id);  
 for (i = 0; i < lib->book\_count; i++) {  
 if (lib->books[i].is\_borrowed &&   
 lib->books[i].borrower\_id == student\_id) {  
 printf("- %s (%s著, %d年)\n",   
 lib->books[i].title,   
 lib->books[i].author,   
 lib->books[i].year);  
 count++;  
 }  
 }  
   
 if (count == 0) {  
 printf(" 借りている本はありません\n");  
 }  
}  
  
int main(void)  
{  
 printf("===== 構造体配列の操作 =====\n\n");  
   
 /\* 1. 構造体配列の基本 \*/  
 printf("1. 構造体配列の基本\n");  
   
 struct Student students[5] = {  
 {1001, "田中太郎", {85, 90, 78, 92, 88}, 0, ' '},  
 {1002, "佐藤花子", {92, 88, 95, 90, 93}, 0, ' '},  
 {1003, "山田次郎", {75, 80, 72, 78, 70}, 0, ' '},  
 {1004, "鈴木美咲", {88, 85, 90, 87, 91}, 0, ' '},  
 {1005, "高橋健太", {60, 65, 58, 70, 62}, 0, ' '}  
 };  
   
 int student\_count = 5;  
 int i;  
   
 /\* 平均点の計算 \*/  
 for (i = 0; i < student\_count; i++) {  
 calculate\_average(&students[i]);  
 }  
   
 printf("=== 全学生の成績 ===\n");  
 for (i = 0; i < student\_count; i++) {  
 print\_student(&students[i]);  
 }  
 printf("\n");  
   
 /\* 2. 構造体配列のソート \*/  
 printf("2. 構造体配列のソート\n");  
   
 sort\_students\_by\_average(students, student\_count);  
   
 printf("=== 成績順（高い順） ===\n");  
 for (i = 0; i < student\_count; i++) {  
 printf("%d位: %s (平均: %.1f)\n",   
 i + 1, students[i].name, students[i].average);  
 }  
 printf("\n");  
   
 /\* 3. 構造体ポインタ配列 \*/  
 printf("3. 構造体ポインタ配列\n");  
   
 struct Student \*top\_students[3];  
 int top\_count = 0;  
   
 /\* 平均80点以上の学生を選抜 \*/  
 for (i = 0; i < student\_count; i++) {  
 if (students[i].average >= 80.0 && top\_count < 3) {  
 top\_students[top\_count++] = &students[i];  
 }  
 }  
   
 printf("=== 優秀学生（平均80点以上） ===\n");  
 for (i = 0; i < top\_count; i++) {  
 printf("%s (平均: %.1f, 評価: %c)\n",   
 top\_students[i]->name,   
 top\_students[i]->average,   
 top\_students[i]->grade);  
 }  
 printf("\n");  
   
 /\* 4. より複雑な構造体の管理 \*/  
 printf("4. 図書館システム\n");  
   
 struct Library library = {.book\_count = 0, .student\_count = 0};  
   
 /\* 書籍の追加 \*/  
 add\_book(&library, 1001, "プログラミング入門", "山田太郎", 2020);  
 add\_book(&library, 1002, "データ構造とアルゴリズム", "佐藤花子", 2019);  
 add\_book(&library, 1003, "オペレーティングシステム", "田中次郎", 2021);  
 add\_book(&library, 1004, "ネットワーク基礎", "鈴木一郎", 2022);  
 add\_book(&library, 1005, "データベース設計", "高橋美咲", 2020);  
   
 printf("図書館の蔵書数: %d冊\n\n", library.book\_count);  
   
 /\* 学生を図書館に登録 \*/  
 for (i = 0; i < 3; i++) {  
 library.students[i] = &students[i];  
 library.student\_count++;  
 }  
   
 /\* 本の貸出 \*/  
 borrow\_book(&library, 1001, students[0].id);  
 borrow\_book(&library, 1003, students[0].id);  
 borrow\_book(&library, 1002, students[1].id);  
 borrow\_book(&library, 1001, students[2].id); /\* 既に貸出中 \*/  
   
 /\* 各学生の借りている本を表示 \*/  
 for (i = 0; i < 3; i++) {  
 show\_borrowed\_books(&library, students[i].id);  
 }  
 printf("\n");  
   
 /\* 5. ポインタ演算による配列走査 \*/  
 printf("5. ポインタ演算による配列走査\n");  
   
 struct Book \*book\_ptr = library.books;  
 struct Book \*book\_end = library.books + library.book\_count;  
   
 printf("=== 貸出中の書籍一覧 ===\n");  
 while (book\_ptr < book\_end) {  
 if (book\_ptr->is\_borrowed) {  
 printf("「%s」 → 学生ID: %d\n",   
 book\_ptr->title, book\_ptr->borrower\_id);  
 }  
 book\_ptr++;  
 }  
   
 return 0;  
}  
  
/\*  
実行結果例:  
===== 構造体配列の操作 =====  
  
1. 構造体配列の基本  
=== 全学生の成績 ===  
ID: 1001, 名前: 田中太郎  
点数: 85 90 78 92 88   
平均: 86.6, 評価: B  
---  
ID: 1002, 名前: 佐藤花子  
点数: 92 88 95 90 93   
平均: 91.6, 評価: A  
---  
ID: 1003, 名前: 山田次郎  
点数: 75 80 72 78 70   
平均: 75.0, 評価: C  
---  
ID: 1004, 名前: 鈴木美咲  
点数: 88 85 90 87 91   
平均: 88.2, 評価: B  
---  
ID: 1005, 名前: 高橋健太  
点数: 60 65 58 70 62   
平均: 63.0, 評価: D  
---  
  
2. 構造体配列のソート  
=== 成績順（高い順） ===  
1位: 佐藤花子 (平均: 91.6)  
2位: 鈴木美咲 (平均: 88.2)  
3位: 田中太郎 (平均: 86.6)  
4位: 山田次郎 (平均: 75.0)  
5位: 高橋健太 (平均: 63.0)  
  
3. 構造体ポインタ配列  
=== 優秀学生（平均80点以上） ===  
佐藤花子 (平均: 91.6, 評価: A)  
鈴木美咲 (平均: 88.2, 評価: B)  
田中太郎 (平均: 86.6, 評価: B)  
  
4. 図書館システム  
図書館の蔵書数: 5冊  
  
ID 1001 の学生が「プログラミング入門」を借りました  
ID 1001 の学生が「オペレーティングシステム」を借りました  
ID 1002 の学生が「データ構造とアルゴリズム」を借りました  
「プログラミング入門」は既に貸出中です  
  
ID 1001 の学生が借りている本:  
- プログラミング入門 (山田太郎著, 2020年)  
- オペレーティングシステム (田中次郎著, 2021年)  
  
ID 1002 の学生が借りている本:  
- データ構造とアルゴリズム (佐藤花子著, 2019年)  
  
ID 1003 の学生が借りている本:  
 借りている本はありません  
  
5. ポインタ演算による配列走査  
=== 貸出中の書籍一覧 ===  
「プログラミング入門」 → 学生ID: 1001  
「データ構造とアルゴリズム」 → 学生ID: 1002  
「オペレーティングシステム」 → 学生ID: 1001  
\*/

C99版は[こちら](struct_arrays_c99.c)を参照してください。

### struct\_basics.c

/\*  
 \* struct\_basics.c - 構造体の基本操作  
 \*   
 \* このプログラムでは以下を学習します：  
 \* - 構造体の定義方法  
 \* - 構造体変数の宣言と初期化  
 \* - 構造体メンバーへのアクセス  
 \* - 構造体の関数への渡し方  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
  
/\* 座標を表す構造体 \*/  
struct Point {  
 int x;  
 int y;  
};  
  
/\* 学生情報を表す構造体 \*/  
struct Student {  
 int id;  
 char name[50];  
 int age;  
 double gpa;  
};  
  
/\* 商品情報を表す構造体 \*/  
struct Product {  
 int code;  
 char name[100];  
 double price;  
 int stock;  
};  
  
/\* 構造体を値渡しで受け取る関数 \*/  
void print\_point\_by\_value(struct Point p)  
{  
 printf("座標: (%d, %d)\n", p.x, p.y);  
}  
  
/\* 構造体をポインタ渡しで受け取る関数 \*/  
void print\_point\_by\_pointer(struct Point \*p)  
{  
 if (p != NULL) {  
 printf("座標: (%d, %d)\n", p->x, p->y);  
 }  
}  
  
/\* 学生情報を表示する関数 \*/  
void display\_student(struct Student s)  
{  
 printf("=== 学生情報 ===\n");  
 printf("ID: %d\n", s.id);  
 printf("名前: %s\n", s.name);  
 printf("年齢: %d歳\n", s.age);  
 printf("GPA: %.2f\n", s.gpa);  
 printf("\n");  
}  
  
/\* 商品情報を更新する関数（ポインタ使用） \*/  
void update\_stock(struct Product \*p, int quantity)  
{  
 if (p != NULL) {  
 p->stock += quantity;  
 if (quantity > 0) {  
 printf("%s の在庫を %d 個追加しました。現在の在庫: %d\n",   
 p->name, quantity, p->stock);  
 } else {  
 printf("%s の在庫を %d 個減らしました。現在の在庫: %d\n",   
 p->name, -quantity, p->stock);  
 }  
 }  
}  
  
int main(void)  
{  
 printf("===== 構造体の基本操作 =====\n\n");  
   
 /\* 1. 構造体の宣言と初期化 \*/  
 printf("1. 構造体の宣言と初期化\n");  
   
 /\* 方法1: 宣言後に個別に代入 \*/  
 struct Point p1;  
 p1.x = 10;  
 p1.y = 20;  
   
 /\* 方法2: 初期化リストを使用 \*/  
 struct Point p2 = {30, 40};  
   
 /\* 方法3: 部分的な初期化（残りは0） \*/  
 struct Point p3 = {50}; /\* y は 0 になる \*/  
   
 printf("p1: (%d, %d)\n", p1.x, p1.y);  
 printf("p2: (%d, %d)\n", p2.x, p2.y);  
 printf("p3: (%d, %d)\n", p3.x, p3.y);  
 printf("\n");  
   
 /\* 2. 構造体のコピー \*/  
 printf("2. 構造体のコピー\n");  
 struct Point p4 = p1; /\* 構造体全体のコピー \*/  
 p4.x = 100; /\* p4を変更してもp1は影響を受けない \*/  
   
 printf("p1（元）: (%d, %d)\n", p1.x, p1.y);  
 printf("p4（コピー後変更）: (%d, %d)\n", p4.x, p4.y);  
 printf("\n");  
   
 /\* 3. 構造体と関数 \*/  
 printf("3. 構造体と関数\n");  
 print\_point\_by\_value(p1); /\* 値渡し \*/  
 print\_point\_by\_pointer(&p2); /\* ポインタ渡し \*/  
 printf("\n");  
   
 /\* 4. 文字列を含む構造体 \*/  
 printf("4. 文字列を含む構造体\n");  
 struct Student student1;  
 student1.id = 1001;  
 strcpy(student1.name, "田中太郎");  
 student1.age = 20;  
 student1.gpa = 3.5;  
   
 display\_student(student1);  
   
 /\* 5. 構造体の配列 \*/  
 printf("5. 構造体の配列\n");  
 struct Product products[3] = {  
 {101, "ノートPC", 98000.0, 5},  
 {102, "マウス", 2980.0, 20},  
 {103, "キーボード", 4500.0, 15}  
 };  
   
 int i;  
 printf("=== 商品リスト ===\n");  
 for (i = 0; i < 3; i++) {  
 printf("商品コード: %d, 名前: %s, 価格: %.0f円, 在庫: %d\n",  
 products[i].code, products[i].name,   
 products[i].price, products[i].stock);  
 }  
 printf("\n");  
   
 /\* 6. 構造体ポインタの使用 \*/  
 printf("6. 構造体ポインタの使用\n");  
 struct Product \*prod\_ptr = &products[0];  
   
 /\* アロー演算子でメンバーアクセス \*/  
 printf("ポインタ経由でアクセス: %s (%.0f円)\n",   
 prod\_ptr->name, prod\_ptr->price);  
   
 /\* 在庫の更新 \*/  
 update\_stock(prod\_ptr, 3);  
 update\_stock(&products[1], -5);  
 printf("\n");  
   
 /\* 7. 構造体のサイズ \*/  
 printf("7. 構造体のサイズ\n");  
 printf("sizeof(struct Point): %lu バイト\n", sizeof(struct Point));  
 printf("sizeof(struct Student): %lu バイト\n", sizeof(struct Student));  
 printf("sizeof(struct Product): %lu バイト\n", sizeof(struct Product));  
 printf("\n");  
   
 /\* 8. ネストした構造体 \*/  
 printf("8. ネストした構造体\n");  
 struct Rectangle {  
 struct Point top\_left;  
 struct Point bottom\_right;  
 };  
   
 struct Rectangle rect = {{0, 0}, {100, 50}};  
 printf("矩形: 左上(%d, %d) - 右下(%d, %d)\n",  
 rect.top\_left.x, rect.top\_left.y,  
 rect.bottom\_right.x, rect.bottom\_right.y);  
   
 /\* 矩形の幅と高さを計算 \*/  
 int width = rect.bottom\_right.x - rect.top\_left.x;  
 int height = rect.bottom\_right.y - rect.top\_left.y;  
 printf("幅: %d, 高さ: %d, 面積: %d\n", width, height, width \* height);  
   
 return 0;  
}  
  
/\*  
実行結果例:  
===== 構造体の基本操作 =====  
  
1. 構造体の宣言と初期化  
p1: (10, 20)  
p2: (30, 40)  
p3: (50, 0)  
  
2. 構造体のコピー  
p1（元）: (10, 20)  
p4（コピー後変更）: (100, 20)  
  
3. 構造体と関数  
座標: (10, 20)  
座標: (30, 40)  
  
4. 文字列を含む構造体  
=== 学生情報 ===  
ID: 1001  
名前: 田中太郎  
年齢: 20歳  
GPA: 3.50  
  
5. 構造体の配列  
=== 商品リスト ===  
商品コード: 101, 名前: ノートPC, 価格: 98000円, 在庫: 5  
商品コード: 102, 名前: マウス, 価格: 2980円, 在庫: 20  
商品コード: 103, 名前: キーボード, 価格: 4500円, 在庫: 15  
  
6. 構造体ポインタの使用  
ポインタ経由でアクセス: ノートPC (98000円)  
ノートPC の在庫を 3 個追加しました。現在の在庫: 8  
マウス の在庫を 5 個減らしました。現在の在庫: 15  
  
7. 構造体のサイズ  
sizeof(struct Point): 8 バイト  
sizeof(struct Student): 72 バイト  
sizeof(struct Product): 120 バイト  
  
8. ネストした構造体  
矩形: 左上(0, 0) - 右下(100, 50)  
幅: 100, 高さ: 50, 面積: 5000  
\*/

C99版は[こちら](struct_basics_c99.c)を参照してください。

### struct\_pointers.c

/\*  
 \* struct\_pointers.c - 構造体ポインタと->演算子  
 \*   
 \* このプログラムでは以下を学習します：  
 \* - 構造体ポインタの宣言と使用  
 \* - アロー演算子（->）の使い方  
 \* - 構造体ポインタと関数  
 \* - 効率的なデータ処理  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
  
/\* 従業員情報を表す構造体 \*/  
struct Employee {  
 int id;  
 char name[50];  
 char department[30];  
 double salary;  
 int years\_of\_service;  
};  
  
/\* 日付を表す構造体 \*/  
struct Date {  
 int year;  
 int month;  
 int day;  
};  
  
/\* プロジェクト情報を表す構造体 \*/  
struct Project {  
 int id;  
 char name[100];  
 struct Date start\_date;  
 struct Date end\_date;  
 struct Employee \*manager; /\* 管理者へのポインタ \*/  
 double budget;  
};  
  
/\* 従業員情報を表示する関数 \*/  
void print\_employee(struct Employee \*emp)  
{  
 if (emp == NULL) {  
 printf("エラー: 無効な従業員データ\n");  
 return;  
 }  
   
 printf("=== 従業員情報 ===\n");  
 printf("ID: %d\n", emp->id);  
 printf("名前: %s\n", emp->name);  
 printf("部署: %s\n", emp->department);  
 printf("給与: %.0f円\n", emp->salary);  
 printf("勤続年数: %d年\n", emp->years\_of\_service);  
 printf("\n");  
}  
  
/\* 給与を更新する関数 \*/  
void update\_salary(struct Employee \*emp, double percentage)  
{  
 if (emp == NULL || percentage < -100) {  
 printf("エラー: 無効なパラメータ\n");  
 return;  
 }  
   
 double old\_salary = emp->salary;  
 emp->salary = old\_salary \* (1 + percentage / 100.0);  
   
 printf("%s の給与を更新しました\n", emp->name);  
 printf("変更前: %.0f円 → 変更後: %.0f円 (%.1f%%)\n",   
 old\_salary, emp->salary, percentage);  
}  
  
/\* 最高給与の従業員を見つける関数 \*/  
struct Employee\* find\_highest\_paid(struct Employee employees[], int count)  
{  
 if (employees == NULL || count <= 0) {  
 return NULL;  
 }  
   
 struct Employee \*highest = &employees[0];  
 int i;  
   
 for (i = 1; i < count; i++) {  
 if (employees[i].salary > highest->salary) {  
 highest = &employees[i];  
 }  
 }  
   
 return highest;  
}  
  
/\* プロジェクト情報を表示する関数 \*/  
void print\_project(struct Project \*proj)  
{  
 if (proj == NULL) {  
 printf("エラー: 無効なプロジェクトデータ\n");  
 return;  
 }  
   
 printf("=== プロジェクト情報 ===\n");  
 printf("ID: %d\n", proj->id);  
 printf("プロジェクト名: %s\n", proj->name);  
 printf("開始日: %d年%d月%d日\n",   
 proj->start\_date.year, proj->start\_date.month, proj->start\_date.day);  
 printf("終了日: %d年%d月%d日\n",   
 proj->end\_date.year, proj->end\_date.month, proj->end\_date.day);  
 printf("予算: %.0f円\n", proj->budget);  
   
 if (proj->manager != NULL) {  
 printf("管理者: %s (%s)\n",   
 proj->manager->name, proj->manager->department);  
 } else {  
 printf("管理者: 未定\n");  
 }  
 printf("\n");  
}  
  
/\* 部署別の統計を計算する関数 \*/  
void calculate\_department\_stats(struct Employee employees[], int count,   
 const char \*department)  
{  
 int dept\_count = 0;  
 double total\_salary = 0.0;  
 int total\_years = 0;  
 int i;  
   
 for (i = 0; i < count; i++) {  
 if (strcmp(employees[i].department, department) == 0) {  
 dept\_count++;  
 total\_salary += employees[i].salary;  
 total\_years += employees[i].years\_of\_service;  
 }  
 }  
   
 if (dept\_count > 0) {  
 printf("=== %s の統計 ===\n", department);  
 printf("人数: %d名\n", dept\_count);  
 printf("平均給与: %.0f円\n", total\_salary / dept\_count);  
 printf("平均勤続年数: %.1f年\n", (double)total\_years / dept\_count);  
 printf("\n");  
 } else {  
 printf("%s に該当する従業員はいません\n\n", department);  
 }  
}  
  
/\* 構造体のスワップ（ポインタのみ交換） \*/  
void swap\_employees(struct Employee \*\*emp1, struct Employee \*\*emp2)  
{  
 struct Employee \*temp = \*emp1;  
 \*emp1 = \*emp2;  
 \*emp2 = temp;  
}  
  
int main(void)  
{  
 printf("===== 構造体ポインタと->演算子 =====\n\n");  
   
 /\* 1. 構造体ポインタの基本 \*/  
 printf("1. 構造体ポインタの基本\n");  
   
 struct Employee emp1 = {  
 1001, "田中太郎", "開発部", 350000.0, 5  
 };  
   
 struct Employee \*emp\_ptr = &emp1;  
   
 /\* ドット演算子とアロー演算子の比較 \*/  
 printf("ドット演算子: %s\n", emp1.name);  
 printf("アロー演算子: %s\n", emp\_ptr->name);  
 printf("間接参照とドット: %s\n", (\*emp\_ptr).name);  
 printf("\n");  
   
 /\* 2. ポインタを使った関数呼び出し \*/  
 printf("2. ポインタを使った関数呼び出し\n");  
 print\_employee(emp\_ptr);  
   
 /\* 3. ポインタを通じた構造体の更新 \*/  
 printf("3. ポインタを通じた構造体の更新\n");  
 update\_salary(emp\_ptr, 10.0); /\* 10%昇給 \*/  
 printf("\n");  
   
 /\* 4. 構造体配列とポインタ \*/  
 printf("4. 構造体配列とポインタ\n");  
   
 struct Employee employees[] = {  
 {2001, "佐藤花子", "営業部", 300000.0, 3},  
 {2002, "山田次郎", "開発部", 400000.0, 7},  
 {2003, "鈴木美咲", "人事部", 320000.0, 4},  
 {2004, "高橋健太", "営業部", 280000.0, 2},  
 {2005, "伊藤良子", "開発部", 450000.0, 10}  
 };  
   
 int emp\_count = sizeof(employees) / sizeof(employees[0]);  
   
 /\* 最高給与の従業員を検索 \*/  
 struct Employee \*highest\_paid = find\_highest\_paid(employees, emp\_count);  
 if (highest\_paid != NULL) {  
 printf("最高給与の従業員:\n");  
 printf("名前: %s, 給与: %.0f円\n\n",   
 highest\_paid->name, highest\_paid->salary);  
 }  
   
 /\* 5. ポインタ配列 \*/  
 printf("5. ポインタ配列\n");  
   
 struct Employee \*dept\_members[5];  
 int dept\_count = 0;  
 int i;  
   
 /\* 開発部のメンバーを抽出 \*/  
 for (i = 0; i < emp\_count; i++) {  
 if (strcmp(employees[i].department, "開発部") == 0) {  
 dept\_members[dept\_count++] = &employees[i];  
 }  
 }  
   
 printf("開発部のメンバー:\n");  
 for (i = 0; i < dept\_count; i++) {  
 printf("- %s (ID: %d)\n",   
 dept\_members[i]->name, dept\_members[i]->id);  
 }  
 printf("\n");  
   
 /\* 6. ネストした構造体とポインタ \*/  
 printf("6. ネストした構造体とポインタ\n");  
   
 struct Project project1 = {  
 101,  
 "新システム開発",  
 {2024, 1, 1},  
 {2024, 12, 31},  
 &employees[1], /\* 山田次郎を管理者に設定 \*/  
 5000000.0  
 };  
   
 print\_project(&project1);  
   
 /\* 7. 部署別統計 \*/  
 printf("7. 部署別統計\n");  
 calculate\_department\_stats(employees, emp\_count, "開発部");  
 calculate\_department\_stats(employees, emp\_count, "営業部");  
   
 /\* 8. ポインタのスワップ \*/  
 printf("8. ポインタのスワップ\n");  
   
 struct Employee \*ptr1 = &employees[0];  
 struct Employee \*ptr2 = &employees[1];  
   
 printf("スワップ前:\n");  
 printf("ptr1 → %s\n", ptr1->name);  
 printf("ptr2 → %s\n", ptr2->name);  
   
 swap\_employees(&ptr1, &ptr2);  
   
 printf("\nスワップ後:\n");  
 printf("ptr1 → %s\n", ptr1->name);  
 printf("ptr2 → %s\n", ptr2->name);  
   
 return 0;  
}  
  
/\*  
実行結果例:  
===== 構造体ポインタと->演算子 =====  
  
1. 構造体ポインタの基本  
ドット演算子: 田中太郎  
アロー演算子: 田中太郎  
間接参照とドット: 田中太郎  
  
2. ポインタを使った関数呼び出し  
=== 従業員情報 ===  
ID: 1001  
名前: 田中太郎  
部署: 開発部  
給与: 350000円  
勤続年数: 5年  
  
3. ポインタを通じた構造体の更新  
田中太郎 の給与を更新しました  
変更前: 350000円 → 変更後: 385000円 (10.0%)  
  
4. 構造体配列とポインタ  
最高給与の従業員:  
名前: 伊藤良子, 給与: 450000円  
  
5. ポインタ配列  
開発部のメンバー:  
- 山田次郎 (ID: 2002)  
- 伊藤良子 (ID: 2005)  
  
6. ネストした構造体とポインタ  
=== プロジェクト情報 ===  
ID: 101  
プロジェクト名: 新システム開発  
開始日: 2024年1月1日  
終了日: 2024年12月31日  
予算: 5000000円  
管理者: 山田次郎 (開発部)  
  
7. 部署別統計  
=== 開発部 の統計 ===  
人数: 2名  
平均給与: 425000円  
平均勤続年数: 8.5年  
  
=== 営業部 の統計 ===  
人数: 2名  
平均給与: 290000円  
平均勤続年数: 2.5年  
  
8. ポインタのスワップ  
スワップ前:  
ptr1 → 佐藤花子  
ptr2 → 山田次郎  
  
スワップ後:  
ptr1 → 山田次郎  
ptr2 → 佐藤花子  
\*/

C99版は[こちら](struct_pointers_c99.c)を参照してください。

# 第12章: 関数ポインタ

## 対応C規格

* **主要対象:** C90
* **学習内容:** 関数ポインタの基本、コールバック関数、関数ポインタ配列、動的な関数呼び出し

## 学習目標

この章を完了すると、以下のことができるようになります：

* 関数ポインタの基本概念を理解する
* 関数ポインタの宣言と初期化ができる
* コールバック関数を実装できる
* 関数ポインタ配列を活用できる
* 動的な関数選択システムを作成できる

## 概要と詳細

### 関数ポインタとは？

関数ポインタは、関数のアドレスを格納する特殊なポインタです。これまでのポインタは変数のアドレスを扱いましたが、関数ポインタは関数のアドレスを扱います。

#### 日常生活での関数ポインタ

関数ポインタを理解するために、リモコンの例を考えてみましょう：

**テレビのリモコン** - ボタン1 → チャンネル1を表示する機能 - ボタン2 → チャンネル2を表示する機能 - ボタン3 → 音量を上げる機能

各ボタンは「どの機能を実行するか」を記憶しています。これが関数ポインタの概念です！

### なぜ関数ポインタが必要なのか？

1. **動的な関数選択**
   * 実行時に呼び出す関数を選べる
   * 条件によって処理を切り替えられる
2. **コールバック関数**
   * 処理の完了時に特定の関数を呼び出す
   * イベント駆動型プログラミングの基礎
3. **関数のテーブル化**
   * メニューシステムの実装
   * コマンドパターンの実現

### 関数ポインタの基本概念

関数ポインタは関数のアドレスを格納するポインタです。これにより、実行時に呼び出す関数を動的に決定できます。

#### 覚え方のコツ

通常のポインタ： int \*ptr; // intを指すポインタ  
関数ポインタ： int (\*ptr)(); // intを返す関数を指すポインタ

**重要**：関数ポインタでは括弧 () が必須です！

#### 関数ポインタの宣言

#include <stdio.h>  
  
/\* 通常の関数 \*/  
int add(int a, int b)  
{  
  
 return a + b;  
  
  
int subtract(int a, int b)  
{  
  
 return a - b;  
  
  
int main(void)  
{  
  
 /\* 関数ポインタの宣言 \*/  
 int (\*operation)(int, int);  
   
 /\* 関数ポインタに関数のアドレスを代入 \*/  
 operation = add; /\* または &add \*/  
   
 /\* 関数ポインタを通じて関数を呼び出し \*/  
 printf("addition: %d\n", operation(, ));  
   
 /\* 別の関数を指すように変更 \*/  
 operation = subtract;  
 printf("subtraction: %d\n", operation(, ));  
   
 return ;

#### 関数ポインタの基本文法

#include <stdio.h>  
  
/\* さまざまな関数 \*/  
void greet(void)  
{  
  
 printf("こんにちは！n");  
  
  
int multiply(int x, int y)  
{  
  
 return x \* y;  
  
  
double divide(double a, double b)  
  
 if (b != .)   
 return a / b;  
   
 return .;  
  
  
int main(void)  
{  
  
 /\* さまざまな関数ポインタの宣言 \*/  
 void (\*greeting\_fnc)(void); /\* 引数なし、戻り値なし \*/  
 int (\*math\_fnc)(int, int); /\* int引数2つ、int戻り値 \*/  
 double (\*calc\_fnc)(double, double); /\* double引数2つ、double戻り値 \*/  
   
 /\* 関数ポインタの初期化 \*/  
 greeting\_fnc = greet;  
 math\_fnc = multiply;  
 calc\_fnc = divide;  
   
 /\* 関数ポインタを使った呼び出し \*/  
 greeting\_fnc(); /\* greet()を呼び出し \*/  
 printf("乗算: %d\n", math\_fnc(, )); /\* multiply(, )を呼び出し \*/  
 printf("除算: %.fn", calc\_fnc(., .)); /\* divide(., .)を呼び出し \*/  
   
 /\* 2つの異なる呼び出し方法 \*/  
 printf("直接呼び出し: %d\n", (\*math\_fnc)(, )); /\* (\*ptr)(args) \*/  
 printf("間接呼び出し: %d\n", math\_fnc(, )); /\* ptr(args) \*/  
   
 return ;

### 関数ポインタの実践的な使用

関数ポインタを実際のプログラムでどのように活用するか見ていきましょう。

#### なぜ実践が重要か？

関数ポインタは概念だけでは理解しにくいため、実際の使用例を通じて学ぶのが効果的です。

#### 計算機システム

#include <stdio.h>  
  
/\* 演算関数群 \*/  
double add\_op(double a, double b) return a + b;   
double sb\_op(double a, double b) return a - b;   
double ml\_op(double a, double b) return a \* b;   
double div\_op(double a, double b)   
  
 if (b != .) return a / b;  
 printf("エラー: ゼロ除算n");  
 return .;  
  
  
/\* 演算を実行する関数 \*/  
double calculate(double a, double b, double (\*operation)(double, double))  
  
 return operation(a, b);  
  
  
/\* 演算子に基づいて関数を選択 \*/  
double (\*get\_operation(char op))(double, double)  
  
 switch (op)   
 case '+': return add\_op;  
 case '-': return sb\_op;  
 case '\*': return ml\_op;  
 case '/': return div\_op;  
 default: return NULL;  
   
  
  
int main(void)  
{  
  
 double num1 = 20.0, num2 = 5.0;  
 char operators[] = {'+', '-', '\*', '/'};  
 const char\* op\_names[] = {"加算", "減算", "乗算", "除算"};  
 int i;  
   
 printf("数値1: %.1f, 数値2: %.1f\n\n", num1, num2);  
   
 /\* すべての演算を実行 \*/  
 for (i = 0; i < 4; i++)   
 {  
 double (\*op\_func)(double, double) = get\_operation(operators[i]);  
 if (op\_func != NULL)   
 {  
 double result = calculate(num1, num2, op\_func);  
 printf("%s (%c): %.2f\n", op\_names[i], operators[i], result);  
 }  
 }  
   
 return 0;

#### 動的メニューシステム

#include <stdio.h>  
  
/\* メニュー項目の処理関数 \*/  
void show\_profile(void)  
{  
  
 printf("=== プロフィール表示 ===n");  
 printf("名前: 田中太郎n");  
 printf("年齢: 歳n");  
 printf("職業: エンジニアnn");  
  
  
void show\_settings(void)  
{  
  
 printf("=== 設定画面 ===n");  
 printf("言語: 日本語n");  
 printf("テーマ: ダークn");  
 printf("通知: ONnn");  
  
  
void show\_help(void)  
{  
  
 printf("=== ヘルプ ===n");  
 printf("このアプリケーションの使用方法:n");  
 printf(". メニューから項目を選択n");  
 printf(". 処理が実行されますnn");  
  
  
void exit\_app(void)  
{  
  
 printf("アプリケーションを終了します。n");  
  
  
/\* メニュー項目の構造体 \*/  
strct MenuItem   
 char name[];  
 void (\*handler)(void); /\* 関数ポインタ \*/  
;  
  
int main(void)  
{  
  
 /\* メニュー項目の定義 \*/  
 strct MenuItem men[] =   
 "プロフィール", show\_profile,  
 "設定", show\_settings,  
 "ヘルプ", show\_help,  
 "終了", exit\_app  
 ;  
   
 int men\_size = sizeof(men) / sizeof(men[]);  
 int choice;  
 int i;  
   
 do   
 printf("=== メインメニュー ===n");  
 for (i = ; i < men\_size; i++)   
 printf("%d. %s\n", i + , men[i].name);  
   
 printf("選択してください (-%d): ", men\_size);  
   
 if (scanf("%d", &choice) == && choice >= && choice <= men\_size)   
 printf("n");  
 men[choice - ].handler(); /\* 関数ポインタを使って実行 \*/  
   
 if (choice == men\_size) /\* 終了が選択された場合 \*/  
 break;  
   
 else   
 printf("無効な選択です。nn");  
 /\* 入力バッファをクリア \*/  
 while (getchar() != 'n');  
   
 while ();  
   
 return ;

### コールバック関数

コールバック関数は他の関数に引数として渡される関数です。

#### コールバックの日常例

レストランでの注文を考えてみましょう：

1. 注文を受ける（関数呼び出し）
2. 料理を作る（処理）
3. **完成したら呼ぶ**（コールバック）

「完成したら何をするか」を事前に決めておくのがコールバックです！

#### 配列処理のコールバック

#include <stdio.h>  
  
/\* 配列の各要素に適用する関数群 \*/  
int square(int x) return x \* x;   
int cube(int x) return x \* x \* x;   
int double\_vale(int x) return x \* ;   
int increment(int x) return x + ;   
  
/\* 配列の各要素に関数を適用 \*/  
void apply\_to\_array(int arr[], int size, int (\*fnc)(int))  
  
 int i;  
 for (i = ; i < size; i++)   
 arr[i] = fnc(arr[i]);  
   
  
  
/\* 配列を表示する関数 \*/  
void print\_array(int arr[], int size, const char\* label)  
{  
  
 int i;  
 printf("%s: ", label);  
 for (i = ; i < size; i++)   
 printf("%d ", arr[i]);  
   
 printf("n");  
  
  
int main(void)  
{  
  
 int numbers[] = , , , , ;  
 int size = sizeof(numbers) / sizeof(numbers[]);  
 int temp[];  
 int i;  
   
 print\_array(numbers, size, "元の配列");  
   
 /\* 乗を適用 \*/  
 for (i = ; i < size; i++) temp[i] = numbers[i];  
 apply\_to\_array(temp, size, square);  
 print\_array(temp, size, "乗後");  
   
 /\* 乗を適用 \*/  
 for (i = ; i < size; i++) temp[i] = numbers[i];  
 apply\_to\_array(temp, size, cube);  
 print\_array(temp, size, "乗後");  
   
 /\* 倍を適用 \*/  
 for (i = ; i < size; i++) temp[i] = numbers[i];  
 apply\_to\_array(temp, size, double\_vale);  
 print\_array(temp, size, "倍後");  
   
 /\* インクリメントを適用 \*/  
 for (i = ; i < size; i++) temp[i] = numbers[i];  
 apply\_to\_array(temp, size, increment);  
 print\_array(temp, size, "インクリメント後");  
   
 return ;

#### ソートのコールバック（比較関数）

#include <stdio.h>  
#include <string.h>  
  
/\* 比較関数の型定義 \*/  
typedef int (\*compare\_fnc\_t)(const void \*a, const void \*b);  
  
/\* 整数の比較関数 \*/  
int compare\_int\_asc(const void \*a, const void \*b)  
{  
  
 int ia = \*(const int\*)a;  
 int ib = \*(const int\*)b;  
 return (ia > ib) - (ia < ib); /\* 昇順 \*/  
  
  
int compare\_int\_desc(const void \*a, const void \*b)  
{  
  
 int ia = \*(const int\*)a;  
 int ib = \*(const int\*)b;  
 return (ib > ia) - (ib < ia); /\* 降順 \*/  
  
  
/\* 簡単なバブルソート（コールバック版） \*/  
void bbble\_sort(void \*base, size\_t num, size\_t size,   
 int (\*compare)(const void \*, const void \*))  
  
 char \*arr = (char\*)base;  
 char \*temp = malloc(size);  
 size\_t i, j;  
   
 if (temp == NULL) return;  
   
 for (i = ; i < num - ; i++)   
 for (j = ; j < num - - i; j++)   
 void \*elem = arr + j \* size;  
 void \*elem = arr + (j + ) \* size;  
   
 if (compare(elem, elem) > )   
 /\* 要素を交換 \*/  
 memcpy(temp, elem, size);  
 memcpy(elem, elem, size);  
 memcpy(elem, temp, size);  
   
   
   
   
 free(temp);  
  
  
/\* 配列を表示 \*/  
void print\_int\_array(int arr[], int size, const char\* label)  
{  
  
 int i;  
 printf("%s: ", label);  
 for (i = ; i < size; i++)   
 printf("%d ", arr[i]);  
   
 printf("n");  
  
  
int main(void)  
{  
  
 int numbers[] = , , , , , , 9, ;  
 int size = sizeof(numbers) / sizeof(numbers[]);  
 int temp[];  
 int i;  
   
 print\_int\_array(numbers, size, "元の配列");  
   
 /\* 昇順ソート \*/  
 for (i = ; i < size; i++) temp[i] = numbers[i];  
 bbble\_sort(temp, size, sizeof(int), compare\_int\_asc);  
 print\_int\_array(temp, size, "昇順ソート");  
   
 /\* 降順ソート \*/  
 for (i = ; i < size; i++) temp[i] = numbers[i];  
 bbble\_sort(temp, size, sizeof(int), compare\_int\_desc);  
 print\_int\_array(temp, size, "降順ソート");  
   
 return ;

### 関数ポインタ配列

関数ポインタの配列を使って、複数の関数を効率的に管理できます。

#### なぜ配列にするのか？

/\* 個別に管理（大変！） \*/  
int (\*func1)() = add;  
int (\*func2)() = sub;  
int (\*func3)() = mul;  
  
/\* 配列で管理（スッキリ！） \*/  
int (\*funcs[3])() = {add, sub, mul};

インデックスで関数を選択できるので、メニューシステムなどに最適です！

#### 演算関数配列

#include <stdio.h>  
  
/\* 演算関数群 \*/  
int add\_func(int a, int b) { return a + b; }  
int sub\_func(int a, int b) { return a - b; }  
int mul\_func(int a, int b) { return a \* b; }  
int div\_func(int a, int b) { return b != 0 ? a / b : 0; }  
int mod\_func(int a, int b) { return b != 0 ? a % b : 0; }  
  
int main(void)  
{  
 /\* 関数ポインタ配列の宣言と初期化 \*/  
 int (\*operations[])(int, int) =   
 {  
 add\_func, /\* インデックス0: 加算 \*/  
 sub\_func, /\* インデックス1: 減算 \*/  
 mul\_func, /\* インデックス2: 乗算 \*/  
 div\_func, /\* インデックス3: 除算 \*/  
 mod\_func /\* インデックス4: 剰余 \*/  
 };  
   
 const char\* op\_names[] = {"加算", "減算", "乗算", "除算", "剰余"};  
 const char\* op\_symbols[] = {"+", "-", "\*", "/", "%"};  
 int num\_ops = sizeof(operations) / sizeof(operations[0]);  
 int a = 20, b = 5;  
 int i;  
   
 printf("数値: a = %d, b = %d\n\n", a, b);  
   
 /\* すべての演算を実行 \*/  
 for (i = 0; i < num\_ops; i++)   
 {  
 int result = operations[i](a, b);  
 printf("%s (%s): %d %s %d = %d\n",   
 op\_names[i], op\_symbols[i], a, op\_symbols[i], b, result);  
 }  
   
 /\* 特定の演算だけ実行 \*/  
 printf("\n特定の演算:\n");  
 printf("乗算結果: %d\n", operations[2](a, b)); /\* mul\_func \*/  
 printf("除算結果: %d\n", operations[3](a, b)); /\* div\_func \*/  
   
 return 0;  
}

#### 状態機械（ステートマシン）

#include <stdio.h>  
  
/\* 状態の定義 \*/  
typedef enum   
{  
 STATE\_IDLE,  
 STATE\_RUNNING,  
 STATE\_PAUSED,  
 STATE\_STOPPED,  
 STATE\_COUNT  
} State;  
  
/\* 状態処理関数の宣言 \*/  
void handle\_idle(void);  
void handle\_running(void);  
void handle\_paused(void);  
void handle\_stopped(void);  
  
/\* 現在の状態 \*/  
static State current\_state = STATE\_IDLE;  
  
/\* 状態処理関数配列 \*/  
void (\*state\_handlers[])(void) =   
{  
 handle\_idle, /\* STATE\_IDLE \*/  
 handle\_running, /\* STATE\_RUNNING \*/  
 handle\_paused, /\* STATE\_PAUSED \*/  
 handle\_stopped /\* STATE\_STOPPED \*/  
};  
  
const char\* state\_names[] =   
{  
 "待機中", "実行中", "一時停止", "停止中"  
};  
  
/\* 各状態の処理関数 \*/  
void handle\_idle(void)  
{  
 printf("[待機中] システムが待機状態です\n");  
 printf(" 利用可能なアクション: 開始(s)\n");  
}  
  
void handle\_running(void)  
{  
 printf("[実行中] システムが動作しています\n");  
 printf(" 利用可能なアクション: 一時停止(p), 停止(q)\n");  
}  
  
void handle\_paused(void)  
{  
 printf("[一時停止] システムが一時停止中です\n");  
 printf(" 利用可能なアクション: 再開(r), 停止(q)\n");  
}  
  
void handle\_stopped(void)  
{  
 printf("[停止中] システムが停止しました\n");  
 printf(" 利用可能なアクション: リセット(reset)\n");  
}  
  
/\* 状態遷移関数 \*/  
void transition\_to\_state(State new\_state)  
{  
 if (new\_state >= 0 && new\_state < STATE\_COUNT)   
 {  
 printf("状態遷移: %s -> %s\n",   
 state\_names[current\_state], state\_names[new\_state]);  
 current\_state = new\_state;  
 }  
}  
  
/\* 現在の状態を処理 \*/  
void process\_current\_state(void)  
{  
 if (current\_state >= 0 && current\_state < STATE\_COUNT)   
 {  
 state\_handlers[current\_state]();  
 }  
}  
  
int main(void)  
{  
 char command[10];  
   
 printf("=== ステートマシンデモ ===\n");  
 printf("コマンド: s(start), p(pause), r(resume), q(quit), reset\n\n");  
   
 while (1)   
 {  
 process\_current\_state();  
 printf("\nコマンドを入力してください: ");  
   
 if (scanf("%9s", command) == 1)   
 {  
 if (strcmp(command, "s") == 0 && current\_state == STATE\_IDLE)   
 {  
 transition\_to\_state(STATE\_RUNNING);  
 }  
 else if (strcmp(command, "p") == 0 && current\_state == STATE\_RUNNING)   
 {  
 transition\_to\_state(STATE\_PAUSED);  
 }  
 else if (strcmp(command, "r") == 0 && current\_state == STATE\_PAUSED)   
 {  
 transition\_to\_state(STATE\_RUNNING);  
 }  
 else if (strcmp(command, "q") == 0)   
 {  
 transition\_to\_state(STATE\_STOPPED);  
 }  
 else if (strcmp(command, "reset") == 0)   
 {  
 transition\_to\_state(STATE\_IDLE);  
 }  
 else if (strcmp(command, "exit") == 0)   
 {  
 printf("プログラムを終了します\n");  
 break;  
 }  
 else   
 {  
 printf("無効なコマンドまたは現在の状態では実行できません\n");  
 }  
 }  
 printf("\n");  
 }  
   
 return 0;  
}

### 高度な関数ポインタ活用

より実践的な関数ポインタの活用方法を見ていきましょう。

#### プラグインシステム

プラグインシステムは、プログラムの機能を後から追加できる仕組みです。関数ポインタはこの実現に最適です！

#include <stdio.h>  
#include <string.h>  
  
/\* プラグインインターフェース \*/  
typedef struct   
{  
 char name[50];  
 char version[10];  
 void (\*init)(void);  
 void (\*execute)(void);  
 void (\*cleanup)(void);  
} Plugin;  
  
/\* プラグイン: ログ出力 \*/  
void log\_init(void) { printf("ログプラグイン初期化\n"); }  
void log\_execute(void) { printf("ログファイルに出力中...\n"); }  
void log\_cleanup(void) { printf("ログプラグイン終了処理\n"); }  
  
/\* プラグイン: データベース接続 \*/  
void db\_init(void) { printf("データベースプラグイン初期化\n"); }  
void db\_execute(void) { printf("データベースに接続中...\n"); }  
void db\_cleanup(void) { printf("データベース接続終了\n"); }  
  
/\* プラグイン: ネットワーク通信 \*/  
void net\_init(void) { printf("ネットワークプラグイン初期化\n"); }  
void net\_execute(void) { printf("ネットワーク通信実行中...\n"); }  
void net\_cleanup(void) { printf("ネットワーク接続終了\n"); }  
  
/\* プラグインの登録 \*/  
Plugin plugins[] =   
{  
 {"LogPlugin", "1.0", log\_init, log\_execute, log\_cleanup},  
 {"DatabasePlugin", "2.0", db\_init, db\_execute, db\_cleanup},  
 {"NetworkPlugin", "1.5", net\_init, net\_execute, net\_cleanup}  
};  
  
int plugin\_count = sizeof(plugins) / sizeof(plugins[0]);  
  
/\* プラグイン管理システム \*/  
void load\_plugins(void)  
{  
 int i;  
 printf("=== プラグイン読み込み ===\n");  
 for (i = 0; i < plugin\_count; i++)   
 {  
 printf("プラグイン: %s (v%s)\n", plugins[i].name, plugins[i].version);  
 plugins[i].init();  
 }  
 printf("\n");  
}  
  
void execute\_plugins(void)  
{  
 int i;  
 printf("=== プラグイン実行 ===\n");  
 for (i = 0; i < plugin\_count; i++)   
 {  
 printf("[%s] ", plugins[i].name);  
 plugins[i].execute();  
 }  
 printf("\n");  
}  
  
void unload\_plugins(void)  
{  
 int i;  
 printf("=== プラグイン終了処理 ===\n");  
 for (i = 0; i < plugin\_count; i++)   
 {  
 printf("[%s] ", plugins[i].name);  
 plugins[i].cleanup();  
 }  
 printf("\n");  
}  
  
/\* 特定のプラグインを実行 \*/  
void execute\_plugin\_by\_name(const char\* name)  
{  
 int i;  
 for (i = 0; i < plugin\_count; i++)   
 {  
 if (strcmp(plugins[i].name, name) == 0)   
 {  
 printf("プラグイン '%s' を実行:\n", name);  
 plugins[i].execute();  
 return;  
 }  
 }  
 printf("プラグイン '%s' が見つかりません\n", name);  
}  
  
int main(void)  
{  
 load\_plugins();  
 execute\_plugins();  
   
 /\* 特定のプラグインのみ実行 \*/  
 execute\_plugin\_by\_name("DatabasePlugin");  
 execute\_plugin\_by\_name("InvalidPlugin");  
   
 unload\_plugins();  
   
 return 0;  
}

#### イベントハンドラシステム

#include <stdio.h>  
  
/\* イベントタイプ \*/  
typedef enum   
{  
 EVENT\_CLICK,  
 EVENT\_KEYPRESS,  
 EVENT\_MOUSE\_MOVE,  
 EVENT\_WINDOW\_CLOSE,  
 EVENT\_TYPE\_COUNT  
} EventType;  
  
/\* イベントデータ \*/  
typedef struct   
{  
 EventType type;  
 int x, y; /\* 座標 \*/  
 int key\_code; /\* キーコード \*/  
 char message[100];  
} Event;  
  
/\* イベントハンドラ関数の型 \*/  
typedef void (\*EventHandler)(const Event\* event);  
  
/\* イベントハンドラ配列 \*/  
EventHandler event\_handlers[EVENT\_TYPE\_COUNT] = {NULL};  
  
/\* 各イベントのハンドラ実装 \*/  
void handle\_click(const Event\* event)  
{  
 printf("クリックイベント: 座標(%d, %d) - %s\n",   
 event->x, event->y, event->message);  
}  
  
void handle\_keypress(const Event\* event)  
{  
 printf("キー押下イベント: キーコード %d - %s\n",   
 event->key\_code, event->message);  
}  
  
void handle\_mouse\_move(const Event\* event)  
{  
 printf("マウス移動イベント: 座標(%d, %d)\n", event->x, event->y);  
}  
  
void handle\_window\_close(const Event\* event)  
{  
 printf("ウィンドウ閉じるイベント: %s\n", event->message);  
}  
  
/\* イベントハンドラの登録 \*/  
void register\_event\_handler(EventType type, EventHandler handler)  
{  
 if (type >= 0 && type < EVENT\_TYPE\_COUNT)   
 {  
 event\_handlers[type] = handler;  
 printf("イベントハンドラ登録: タイプ %d\n", type);  
 }  
}  
  
/\* イベントの処理 \*/  
void process\_event(const Event\* event)  
{  
 if (event->type >= 0 && event->type < EVENT\_TYPE\_COUNT &&   
 event\_handlers[event->type] != NULL)   
 {  
 event\_handlers[event->type](event);  
 }   
 else   
 {  
 printf("未処理のイベント: タイプ %d\n", event->type);  
 }  
}  
  
/\* イベントの作成補助関数 \*/  
Event create\_click\_event(int x, int y, const char\* msg)  
{  
 Event event = {EVENT\_CLICK, x, y, 0, ""};  
 strncpy(event.message, msg, sizeof(event.message) - 1);  
 return event;  
}  
  
Event create\_keypress\_event(int key\_code, const char\* msg)  
{  
 Event event = {EVENT\_KEYPRESS, 0, 0, key\_code, ""};  
 strncpy(event.message, msg, sizeof(event.message) - 1);  
 return event;  
}  
  
int main(void)  
{  
 printf("=== イベントハンドラシステム ===\n\n");  
   
 /\* イベントハンドラを登録 \*/  
 register\_event\_handler(EVENT\_CLICK, handle\_click);  
 register\_event\_handler(EVENT\_KEYPRESS, handle\_keypress);  
 register\_event\_handler(EVENT\_MOUSE\_MOVE, handle\_mouse\_move);  
 register\_event\_handler(EVENT\_WINDOW\_CLOSE, handle\_window\_close);  
   
 printf("\n=== イベント処理テスト ===\n");  
   
 /\* さまざまなイベントを生成・処理 \*/  
 Event click\_event = create\_click\_event(100, 200, "ボタンクリック");  
 process\_event(&click\_event);  
   
 Event key\_event = create\_keypress\_event(65, "Aキー押下");  
 process\_event(&key\_event);  
   
 Event mouse\_event = {EVENT\_MOUSE\_MOVE, 150, 300, 0, ""};  
 process\_event(&mouse\_event);  
   
 Event close\_event = {EVENT\_WINDOW\_CLOSE, 0, 0, 0, "アプリケーション終了"};  
 process\_event(&close\_event);  
   
 /\* 未登録のイベントタイプ \*/  
 Event unknown\_event = {99, 0, 0, 0, ""};  
 process\_event(&unknown\_event);  
   
 return 0;  
}

### 関数ポインタのtypedef

複雑な関数ポインタの型を簡潔に書くためにtypedefを使用します。

#### typedefを使うメリット

/\* typedefなし（読みにくい！） \*/  
void sort(int arr[], int size, int (\*compare)(int, int));  
  
/\* typedefあり（スッキリ！） \*/  
typedef int (\*CompareFunc)(int, int);  
void sort(int arr[], int size, CompareFunc compare);

特に複雑な関数ポインタでは、typedefの効果が絶大です！

#include <stdio.h>  
  
/\* 関数ポインタの型定義 \*/  
typedef int (\*BinaryOperation)(int, int);  
typedef void (\*EventCallback)(int event\_id, const char\* message);  
typedef double (\*MathFunction)(double);  
  
/\* 演算関数 \*/  
int add\_nums(int a, int b) { return a + b; }  
int mul\_nums(int a, int b) { return a \* b; }  
  
/\* イベントコールバック関数 \*/  
void on\_start(int id, const char\* msg)   
{  
 printf("開始イベント[%d]: %s\n", id, msg);  
}  
  
void on\_stop(int id, const char\* msg)   
{  
 printf("停止イベント[%d]: %s\n", id, msg);  
}  
  
/\* 数学関数 \*/  
double square\_root(double x) { return x \* x; } /\* 簡易版 \*/  
double absolute(double x) { return x < 0 ? -x : x; }  
  
/\* 関数を実行する汎用関数 \*/  
int execute\_binary\_op(int a, int b, BinaryOperation op)  
{  
 return op(a, b);  
}  
  
void trigger\_event(int id, const char\* message, EventCallback callback)  
{  
 callback(id, message);  
}  
  
double apply\_math\_func(double value, MathFunction func)  
{  
 return func(value);  
}  
  
int main(void)  
{  
 /\* 型定義を使った関数ポインタの使用 \*/  
 BinaryOperation math\_op;  
 EventCallback event\_handler;  
 MathFunction math\_func;  
   
 printf("=== typedef を使った関数ポインタ ===\n\n");  
   
 /\* 数値演算 \*/  
 math\_op = add\_nums;  
 printf("加算: %d\n", execute\_binary\_op(10, 5, math\_op));  
   
 math\_op = mul\_nums;  
 printf("乗算: %d\n", execute\_binary\_op(10, 5, math\_op));  
   
 /\* イベント処理 \*/  
 event\_handler = on\_start;  
 trigger\_event(1, "システム開始", event\_handler);  
   
 event\_handler = on\_stop;  
 trigger\_event(2, "システム終了", event\_handler);  
   
 /\* 数学関数 \*/  
 math\_func = square\_root;  
 printf("二乗: %.2f\n", apply\_math\_func(5.0, math\_func));  
   
 math\_func = absolute;  
 printf("絶対値: %.2f\n", apply\_math\_func(-7.5, math\_func));  
   
 return 0;  
}

### 関数ポインタを使う際の重要なポイント

初心者が覚えておくべき関数ポインタの重要な点をまとめます。

#### 1. 関数ポインタの宣言パターン

/\* 基本形 \*/  
戻り値型 (\*ポインタ名)(引数型リスト);  
  
/\* 例 \*/  
int (\*calc)(int, int); /\* int型2つを受け取りintを返す関数へのポインタ \*/  
void (\*action)(void); /\* 引数なし、戻り値なしの関数へのポインタ \*/  
double (\*convert)(double); /\* doubleを受け取りdoubleを返す関数へのポインタ \*/

#### 2. 関数ポインタの代入と呼び出し

/\* 関数の定義 \*/  
int add(int a, int b) { return a + b; }  
  
/\* 関数ポインタへの代入 \*/  
int (\*op)(int, int);  
op = add; /\* 関数名だけ（推奨） \*/  
op = &add; /\* &を付けてもOK \*/  
  
/\* 関数ポインタの呼び出し \*/  
int result = op(10, 20); /\* 通常の関数のように呼び出し（推奨） \*/  
int result = (\*op)(10, 20); /\* 明示的な間接参照でもOK \*/

#### 3. よくある間違いと対策

**間違い1：括弧の位置**

/\* 間違い：関数を返すポインタになってしまう \*/  
int \*func(int, int); /\* これは関数宣言 \*/  
  
/\* 正しい：関数へのポインタ \*/  
int (\*func)(int, int); /\* 括弧が重要！ \*/

**間違い2：型の不一致**

/\* 間違い：引数の型が違う \*/  
void func(int x) { }  
void (\*ptr)(double) = func; /\* エラー！ \*/  
  
/\* 正しい：型を一致させる \*/  
void (\*ptr)(int) = func; /\* OK \*/

**間違い3：NULLチェック忘れ**

/\* 危険：NULLチェックなし \*/  
int (\*func\_ptr)(int) = NULL;  
int result = func\_ptr(10); /\* クラッシュ！ \*/  
  
/\* 安全：NULLチェックあり \*/  
if (func\_ptr != NULL) {  
 int result = func\_ptr(10);  
}

### 関数ポインタを学ぶステップ

1. **基本から始める**
   * 単純な関数ポインタの宣言と使用
   * 加算・減算などの簡単な例で練習
2. **コールバックを理解**
   * 関数を引数として渡す練習
   * qsort関数の比較関数を作ってみる
3. **配列で管理**
   * 関数ポインタ配列でメニューシステム
   * 計算機プログラムの実装
4. **実践的な使用**
   * イベントハンドラの実装
   * プラグインシステムの作成

### 関数ポインタの活用場面

1. **動的な処理の切り替え**
   * 実行時に処理を選択
   * 条件に応じた関数の呼び分け
2. **汎用的な処理の実装**
   * ソート関数の比較処理
   * フィルタリング処理
3. **イベント駆動プログラミング**
   * GUIのボタンクリック処理
   * ゲームのイベント処理

## サンプルコード

### 関数ポインタの基本

* **C90版**: <examples/function_pointer_basic.c>
* **C99版**: <examples/function_pointer_basic_c99.c>

関数ポインタの宣言、初期化、基本的な使用方法を学習します。

### コールバック関数

* **C90版**: <examples/callback_functions.c>
* **C99版**: <examples/callback_functions_c99.c>

コールバック関数を使ったイベント処理システムを学習します。

### 関数ポインタ配列

* **C90版**: <examples/function_pointer_arrays.c>
* **C99版**: <examples/function_pointer_arrays_c99.c>

関数ポインタ配列を使った動的な関数選択を学習します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic function\_pointer\_basic.c -o function\_pointer\_basic  
  
# 実行  
./function\_pointer\_basic

## 演習課題

### 基礎問題

1. **計算機の実装**
   * 四則演算の関数を作成し、関数ポインタを使って動的に演算を選択できる計算機を実装してください
2. **ソート比較関数**
   * 整数配列を昇順・降順でソートできるよう、比較関数を関数ポインタで渡すソート関数を実装してください
3. **メニューシステム**
   * 関数ポインタ配列を使って、複数の機能を持つメニューシステムを作成してください

### 応用問題

1. **データ変換システム**
   * さまざまなデータ変換関数（大文字変換、小文字変換など）を関数ポインタで管理するシステムを実装してください
2. **フィルタシステム**
   * 配列の要素をフィルタリングする関数を、条件を関数ポインタで渡して実装してください
3. **ゲームの状態管理**
   * ゲームの状態（メニュー、プレイ中、ポーズ、終了）を関数ポインタ配列で管理するシステムを作成してください

### 発展問題

1. **プラグインシステム**
   * 実行時に機能を追加できるプラグインシステムを関数ポインタで実装してください
2. **コマンドパターン**
   * コマンドパターンを関数ポインタで実装し、undo/redo機能付きのシステムを作成してください
3. **イベントドリブンシステム**
   * 複数のイベントタイプを処理できるイベントドリブンシステムを実装してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make function\_pointer\_basic  
  
# 実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* 関数ポインタの型変換は明示的なキャストが必要
* 可変引数関数への関数ポインタは制限的

### C99以降の拡張

* より柔軟な関数ポインタの初期化
* inline関数への関数ポインタサポート改善

## よくある間違い

### 1. 関数ポインタの初期化忘れ

/\* NG: 初期化されていない関数ポインタ \*/  
int (\*func\_ptr)(int, int);  
int result = func\_ptr(10, 5); /\* 未定義動作 \*/  
  
/\* OK: 適切な初期化 \*/  
int (\*func\_ptr)(int, int) = add\_function;  
int result = func\_ptr(10, 5);

### 2. 関数ポインタのNULLチェック不足

/\* NG: NULLチェックなし \*/  
void execute\_callback(void (\*callback)(void))  
{  
 callback(); /\* callbackがNULLの場合クラッシュ \*/  
}  
  
/\* OK: NULLチェック \*/  
void execute\_callback(void (\*callback)(void))  
{  
 if (callback != NULL)   
 {  
 callback();  
 }  
}

### 3. 関数の戻り値型の不一致

/\* NG: 戻り値型が一致しない \*/  
double divide\_func(int a, int b) { return (double)a / b; }  
int (\*math\_op)(int, int) = divide\_func; /\* 型エラー \*/  
  
/\* OK: 戻り値型を一致させる \*/  
int divide\_func(int a, int b) { return a / b; }  
int (\*math\_op)(int, int) = divide\_func;

## 次の章へ

関数ポインタを理解したら、[複数ファイル・発展技術](../advanced/README.md) に進んでください。

## 参考資料

* [C言語関数ポインタリファレンス](https://ja.cppreference.com/w/c/language/pointer)
* [コールバック関数の設計パターン](https://en.wikipedia.org/wiki/Callback_(computer_programming))
* [関数ポインタの活用例](https://www.learn-c.org/en/Function_Pointers)

## サンプルコード

### callback\_functions.c

/\* コールバック関数の活用 \*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
/\* 配列処理用のコールバック関数群 \*/  
int square(int x) { return x \* x; }  
int cube(int x) { return x \* x \* x; }  
int double\_value(int x) { return x \* 2; }  
int increment(int x) { return x + 1; }  
  
/\* 条件判定用のコールバック関数群 \*/  
int is\_even(int x) { return x % 2 == 0; }  
int is\_odd(int x) { return x % 2 != 0; }  
int is\_positive(int x) { return x > 0; }  
int is\_negative(int x) { return x < 0; }  
  
/\* 配列の各要素に関数を適用 \*/  
void apply\_to\_array(int arr[], int size, int (\*func)(int))  
{  
 int i;  
 for (i = 0; i < size; i++)  
 {  
 arr[i] = func(arr[i]);  
 }  
}  
  
/\* 配列をフィルタリング \*/  
int filter\_array(int source[], int dest[], int size, int (\*predicate)(int))  
{  
 int count = 0;  
 int i;  
  
 for (i = 0; i < size; i++)  
 {  
 if (predicate(source[i]))  
 {  
 dest[count++] = source[i];  
 }  
 }  
  
 return count;  
}  
  
/\* 配列を表示する関数 \*/  
void print\_array(int arr[], int size, const char \*label)  
{  
 int i;  
 printf("%s: ", label);  
 for (i = 0; i < size; i++)  
 {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
/\* 文字列処理用のコールバック関数群 \*/  
void to\_upper\_char(char \*c)  
{  
 if (\*c >= 'a' && \*c <= 'z')  
 {  
 \*c = \*c - 'a' + 'A';  
 }  
}  
  
void to\_lower\_char(char \*c)  
{  
 if (\*c >= 'A' && \*c <= 'Z')  
 {  
 \*c = \*c - 'A' + 'a';  
 }  
}  
  
void replace\_space\_with\_underscore(char \*c)  
{  
 if (\*c == ' ')  
 {  
 \*c = '\_';  
 }  
}  
  
/\* 文字列の各文字にコールバック関数を適用 \*/  
void process\_string(char str[], void (\*processor)(char \*))  
{  
 int i;  
 for (i = 0; str[i] != '\0'; i++)  
 {  
 processor(&str[i]);  
 }  
}  
  
/\* ソート用の比較関数 \*/  
int compare\_ascending(int a, int b)  
{  
 return a - b;  
}  
  
int compare\_descending(int a, int b)  
{  
 return b - a;  
}  
  
/\* バブルソート（コールバック版） \*/  
void bubble\_sort(int arr[], int size, int (\*compare)(int, int))  
{  
 int i, j, temp;  
  
 for (i = 0; i < size - 1; i++)  
 {  
 for (j = 0; j < size - 1 - i; j++)  
 {  
 if (compare(arr[j], arr[j + 1]) > 0)  
 {  
 temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 }  
 }  
 }  
}  
  
/\* イベントハンドラシステム \*/  
void on\_start\_event(const char \*message)  
{  
 printf("[開始イベント] %s\n", message);  
}  
  
void on\_stop\_event(const char \*message)  
{  
 printf("[停止イベント] %s\n", message);  
}  
  
void on\_error\_event(const char \*message)  
{  
 printf("[エラーイベント] %s\n", message);  
}  
  
/\* イベントを処理する関数 \*/  
void trigger\_event(const char \*message, void (\*event\_handler)(const char \*))  
{  
 if (event\_handler != NULL)  
 {  
 event\_handler(message);  
 }  
}  
  
/\* 計算処理のログ付き実行 \*/  
int execute\_with\_logging(int a, int b, int (\*operation)(int, int), const char \*op\_name)  
{  
 printf("計算開始: %s(%d, %d)\n", op\_name, a, b);  
 int result = operation(a, b);  
 printf("計算完了: %s(%d, %d) = %d\n", op\_name, a, b, result);  
 return result;  
}  
  
/\* 簡単な加算関数（ログ用） \*/  
int add\_for\_logging(int a, int b)  
{  
 return a + b;  
}  
  
int multiply\_for\_logging(int a, int b)  
{  
 return a \* b;  
}  
  
int main(void)  
{  
 printf("=== コールバック関数の活用 ===\n\n");  
  
 /\* 1. 配列処理のコールバック \*/  
 printf("=== 配列処理のコールバック ===\n");  
 int numbers[] = {1, 2, 3, 4, 5};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
 int temp[5];  
 int i;  
  
 print\_array(numbers, size, "元の配列");  
  
 /\* 2乗を適用 \*/  
 for (i = 0; i < size; i++)  
 temp[i] = numbers[i];  
 apply\_to\_array(temp, size, square);  
 print\_array(temp, size, "2乗後");  
  
 /\* 3乗を適用 \*/  
 for (i = 0; i < size; i++)  
 temp[i] = numbers[i];  
 apply\_to\_array(temp, size, cube);  
 print\_array(temp, size, "3乗後");  
  
 /\* 2倍を適用 \*/  
 for (i = 0; i < size; i++)  
 temp[i] = numbers[i];  
 apply\_to\_array(temp, size, double\_value);  
 print\_array(temp, size, "2倍後");  
  
 /\* 2. 配列フィルタリング \*/  
 printf("\n=== 配列フィルタリング ===\n");  
 int test\_array[] = {-3, -1, 0, 2, 4, 5, 7, 8};  
 int test\_size = sizeof(test\_array) / sizeof(test\_array[0]);  
 int filtered[8];  
 int filtered\_count;  
  
 print\_array(test\_array, test\_size, "元の配列");  
  
 /\* 偶数をフィルタリング \*/  
 filtered\_count = filter\_array(test\_array, filtered, test\_size, is\_even);  
 print\_array(filtered, filtered\_count, "偶数のみ");  
  
 /\* 奇数をフィルタリング \*/  
 filtered\_count = filter\_array(test\_array, filtered, test\_size, is\_odd);  
 print\_array(filtered, filtered\_count, "奇数のみ");  
  
 /\* 正の数をフィルタリング \*/  
 filtered\_count = filter\_array(test\_array, filtered, test\_size, is\_positive);  
 print\_array(filtered, filtered\_count, "正の数のみ");  
  
 /\* 3. 文字列処理のコールバック \*/  
 printf("\n=== 文字列処理のコールバック ===\n");  
 char text1[] = "Hello World";  
 char text2[] = "Hello World";  
 char text3[] = "Hello World";  
  
 printf("元の文字列: '%s'\n", text1);  
  
 process\_string(text1, to\_upper\_char);  
 printf("大文字変換: '%s'\n", text1);  
  
 process\_string(text2, to\_lower\_char);  
 printf("小文字変換: '%s'\n", text2);  
  
 process\_string(text3, replace\_space\_with\_underscore);  
 printf("スペース置換: '%s'\n", text3);  
  
 /\* 4. ソートのコールバック \*/  
 printf("\n=== ソートのコールバック ===\n");  
 int sort\_array[] = {64, 34, 25, 12, 22, 11, 90, 5};  
 int sort\_size = sizeof(sort\_array) / sizeof(sort\_array[0]);  
 int sort\_temp[8];  
  
 print\_array(sort\_array, sort\_size, "元の配列");  
  
 /\* 昇順ソート \*/  
 for (i = 0; i < sort\_size; i++)  
 sort\_temp[i] = sort\_array[i];  
 bubble\_sort(sort\_temp, sort\_size, compare\_ascending);  
 print\_array(sort\_temp, sort\_size, "昇順ソート");  
  
 /\* 降順ソート \*/  
 for (i = 0; i < sort\_size; i++)  
 sort\_temp[i] = sort\_array[i];  
 bubble\_sort(sort\_temp, sort\_size, compare\_descending);  
 print\_array(sort\_temp, sort\_size, "降順ソート");  
  
 /\* 5. イベントハンドラシステム \*/  
 printf("\n=== イベントハンドラシステム ===\n");  
  
 trigger\_event("アプリケーション開始", on\_start\_event);  
 trigger\_event("処理中にエラーが発生", on\_error\_event);  
 trigger\_event("アプリケーション終了", on\_stop\_event);  
  
 /\* 6. ログ付き計算実行 \*/  
 printf("\n=== ログ付き計算実行 ===\n");  
  
 execute\_with\_logging(10, 5, add\_for\_logging, "加算");  
 execute\_with\_logging(7, 8, multiply\_for\_logging, "乗算");  
  
 printf("\n=== コールバック関数デモ完了 ===\n");  
  
 return 0;  
}

C99版は[こちら](callback_functions_c99.c)を参照してください。

### function\_pointer\_arrays.c

/\* 関数ポインタ配列の活用 \*/  
#include <stdio.h>  
#include <string.h>  
  
/\* 計算機システム用の演算関数群 \*/  
double add\_operation(double a, double b) { return a + b; }  
double sub\_operation(double a, double b) { return a - b; }  
double mul\_operation(double a, double b) { return a \* b; }  
double div\_operation(double a, double b)  
{  
 if (b != 0.0)  
 return a / b;  
 printf("エラー: ゼロ除算\n");  
 return 0.0;  
}  
  
/\* メニューシステム用の処理関数群 \*/  
void show\_profile(void)  
{  
 printf("=== プロフィール表示 ===\n");  
 printf("名前: 田中太郎\n");  
 printf("年齢: 25歳\n");  
 printf("職業: エンジニア\n");  
 printf("========================\n");  
}  
  
void show\_settings(void)  
{  
 printf("=== 設定画面 ===\n");  
 printf("言語: 日本語\n");  
 printf("テーマ: ダーク\n");  
 printf("通知: ON\n");  
 printf("===============\n");  
}  
  
void show\_help(void)  
{  
 printf("=== ヘルプ ===\n");  
 printf("このアプリケーションの使用方法:\n");  
 printf("1. メニューから項目を選択\n");  
 printf("2. 処理が実行されます\n");  
 printf("3. メインメニューに戻ります\n");  
 printf("=============\n");  
}  
  
void show\_about(void)  
{  
 printf("=== このアプリについて ===\n");  
 printf("関数ポインタ配列デモ v1.0\n");  
 printf("C言語学習用サンプルプログラム\n");  
 printf("========================\n");  
}  
  
void exit\_application(void)  
{  
 printf("アプリケーションを終了します。\n");  
}  
  
/\* ゲーム状態管理用の関数群 \*/  
void state\_menu(void)  
{  
 printf("[メニュー状態] ゲームメニューを表示中\n");  
 printf(" 利用可能なアクション: 開始(s), 設定(c), 終了(q)\n");  
}  
  
void state\_playing(void)  
{  
 printf("[プレイ中状態] ゲームをプレイ中\n");  
 printf(" 利用可能なアクション: 一時停止(p), メニューに戻る(m)\n");  
}  
  
void state\_paused(void)  
{  
 printf("[一時停止状態] ゲームが一時停止中\n");  
 printf(" 利用可能なアクション: 再開(r), メニューに戻る(m)\n");  
}  
  
void state\_settings(void)  
{  
 printf("[設定状態] ゲーム設定を調整中\n");  
 printf(" 利用可能なアクション: 保存(s), キャンセル(c)\n");  
}  
  
/\* データ変換関数群 \*/  
int to\_upper\_case(int c)  
{  
 if (c >= 'a' && c <= 'z')  
 {  
 return c - 'a' + 'A';  
 }  
 return c;  
}  
  
int to\_lower\_case(int c)  
{  
 if (c >= 'A' && c <= 'Z')  
 {  
 return c - 'A' + 'a';  
 }  
 return c;  
}  
  
int rot13\_transform(int c)  
{  
 if (c >= 'a' && c <= 'z')  
 {  
 return (c - 'a' + 13) % 26 + 'a';  
 }  
 if (c >= 'A' && c <= 'Z')  
 {  
 return (c - 'A' + 13) % 26 + 'A';  
 }  
 return c;  
}  
  
int reverse\_case(int c)  
{  
 if (c >= 'a' && c <= 'z')  
 {  
 return c - 'a' + 'A';  
 }  
 if (c >= 'A' && c <= 'Z')  
 {  
 return c - 'A' + 'a';  
 }  
 return c;  
}  
  
/\* メニュー項目構造体 \*/  
struct MenuItem  
{  
 char name[30];  
 void (\*handler)(void);  
};  
  
/\* ゲーム状態の列挙型 \*/  
typedef enum  
{  
 GAME\_MENU = 0,  
 GAME\_PLAYING = 1,  
 GAME\_PAUSED = 2,  
 GAME\_SETTINGS = 3,  
 GAME\_STATE\_COUNT = 4  
} GameState;  
  
/\* 文字変換関数の適用 \*/  
void apply\_transform(char \*str, int (\*transform)(int))  
{  
 int i;  
 for (i = 0; str[i] != '\0'; i++)  
 {  
 str[i] = transform(str[i]);  
 }  
}  
  
/\* 配列の表示 \*/  
void print\_double\_array(double arr[], int size, const char \*label)  
{  
 int i;  
 printf("%s: ", label);  
 for (i = 0; i < size; i++)  
 {  
 printf("%.2f ", arr[i]);  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 printf("=== 関数ポインタ配列の活用 ===\n\n");  
  
 /\* 1. 計算機システム \*/  
 printf("=== 計算機システム ===\n");  
  
 /\* 関数ポインタ配列の定義 \*/  
 double (\*operations[])(double, double) = {  
 add\_operation, /\* インデックス0: 加算 \*/  
 sub\_operation, /\* インデックス1: 減算 \*/  
 mul\_operation, /\* インデックス2: 乗算 \*/  
 div\_operation /\* インデックス3: 除算 \*/  
 };  
  
 const char \*op\_names[] = {"加算", "減算", "乗算", "除算"};  
 const char \*op\_symbols[] = {"+", "-", "\*", "/"};  
 int num\_ops = sizeof(operations) / sizeof(operations[0]);  
 double a = 15.0, b = 4.0;  
 int i;  
  
 printf("数値: a = %.1f, b = %.1f\n", a, b);  
  
 /\* 全ての演算を実行 \*/  
 for (i = 0; i < num\_ops; i++)  
 {  
 double result = operations[i](a, b);  
 printf("%s (%s): %.1f %s %.1f = %.2f\n",  
 op\_names[i], op\_symbols[i], a, op\_symbols[i], b, result);  
 }  
  
 /\* 特定の演算のみ実行 \*/  
 printf("\n特定の演算:\n");  
 printf("乗算結果: %.2f\n", operations[2](a, b)); /\* mul\_operation \*/  
 printf("除算結果: %.2f\n", operations[3](a, b)); /\* div\_operation \*/  
  
 /\* 2. メニューシステム \*/  
 printf("\n=== メニューシステム ===\n");  
  
 /\* メニュー項目の定義 \*/  
 struct MenuItem menu[] = {  
 {"プロフィール", show\_profile},  
 {"設定", show\_settings},  
 {"ヘルプ", show\_help},  
 {"このアプリについて", show\_about},  
 {"終了", exit\_application}};  
  
 int menu\_size = sizeof(menu) / sizeof(menu[0]);  
  
 printf("利用可能なメニュー項目:\n");  
 for (i = 0; i < menu\_size; i++)  
 {  
 printf("%d. %s\n", i + 1, menu[i].name);  
 }  
  
 /\* メニュー項目の実行例 \*/  
 printf("\nメニュー実行例:\n");  
 menu[0].handler(); /\* プロフィール表示 \*/  
 menu[2].handler(); /\* ヘルプ表示 \*/  
  
 /\* 3. ゲーム状態管理システム \*/  
 printf("\n=== ゲーム状態管理システム ===\n");  
  
 /\* 状態処理関数配列 \*/  
 void (\*state\_handlers[])(void) = {  
 state\_menu, /\* GAME\_MENU \*/  
 state\_playing, /\* GAME\_PLAYING \*/  
 state\_paused, /\* GAME\_PAUSED \*/  
 state\_settings /\* GAME\_SETTINGS \*/  
 };  
  
 const char \*state\_names[] = {  
 "メニュー", "プレイ中", "一時停止", "設定"};  
  
 GameState current\_state;  
  
 printf("各ゲーム状態の処理:\n");  
 for (current\_state = GAME\_MENU; current\_state < GAME\_STATE\_COUNT; current\_state++)  
 {  
 printf("\n現在の状態: %s\n", state\_names[current\_state]);  
 state\_handlers[current\_state]();  
 }  
  
 /\* 4. 動的な状態遷移 \*/  
 printf("\n=== 動的な状態遷移 ===\n");  
  
 current\_state = GAME\_MENU;  
 printf("初期状態: %s\n", state\_names[current\_state]);  
 state\_handlers[current\_state]();  
  
 /\* 状態遷移の例 \*/  
 current\_state = GAME\_PLAYING;  
 printf("\n状態遷移: %s\n", state\_names[current\_state]);  
 state\_handlers[current\_state]();  
  
 current\_state = GAME\_PAUSED;  
 printf("\n状態遷移: %s\n", state\_names[current\_state]);  
 state\_handlers[current\_state]();  
  
 /\* 5. 文字変換システム \*/  
 printf("\n=== 文字変換システム ===\n");  
  
 /\* 変換関数配列 \*/  
 int (\*transforms[])(int) = {  
 to\_upper\_case,  
 to\_lower\_case,  
 rot13\_transform,  
 reverse\_case};  
  
 const char \*transform\_names[] = {  
 "大文字変換",  
 "小文字変換",  
 "ROT13暗号",  
 "大小文字反転"};  
  
 int num\_transforms = sizeof(transforms) / sizeof(transforms[0]);  
 char original[] = "Hello World!";  
 char test\_str[50];  
  
 printf("元の文字列: '%s'\n", original);  
  
 /\* 各変換を適用 \*/  
 for (i = 0; i < num\_transforms; i++)  
 {  
 strcpy(test\_str, original);  
 apply\_transform(test\_str, transforms[i]);  
 printf("%s: '%s'\n", transform\_names[i], test\_str);  
 }  
  
 /\* 6. 複数の数値に同じ演算を適用 \*/  
 printf("\n=== 複数数値への演算適用 ===\n");  
  
 double numbers[] = {1.0, 2.0, 3.0, 4.0, 5.0};  
 double results[5];  
 int numbers\_size = sizeof(numbers) / sizeof(numbers[0]);  
  
 print\_double\_array(numbers, numbers\_size, "元の配列");  
  
 /\* 各演算を全ての数値ペア (numbers[i], 2.0) に適用 \*/  
 for (i = 0; i < num\_ops; i++)  
 {  
 int j;  
 for (j = 0; j < numbers\_size; j++)  
 {  
 results[j] = operations[i](numbers[j], 2.0);  
 }  
 printf("%s (x, 2.0): ", op\_names[i]);  
 print\_double\_array(results, numbers\_size, "結果");  
 }  
  
 /\* 7. 条件付き関数実行 \*/  
 printf("\n=== 条件付き関数実行 ===\n");  
  
 printf("演算インデックス指定実行:\n");  
 int op\_index = 2; /\* 乗算 \*/  
 if (op\_index >= 0 && op\_index < num\_ops)  
 {  
 double result = operations[op\_index](10.0, 3.0);  
 printf("%s: 10.0 %s 3.0 = %.2f\n",  
 op\_names[op\_index], op\_symbols[op\_index], result);  
 }  
  
 /\* 複数のメニュー項目を一度に実行 \*/  
 printf("\n複数メニュー項目の実行:\n");  
 int menu\_indices[] = {1, 3}; /\* 設定, このアプリについて \*/  
 int indices\_count = sizeof(menu\_indices) / sizeof(menu\_indices[0]);  
  
 for (i = 0; i < indices\_count; i++)  
 {  
 int idx = menu\_indices[i];  
 if (idx >= 0 && idx < menu\_size)  
 {  
 printf("\n[%s] 実行:\n", menu[idx].name);  
 menu[idx].handler();  
 }  
 }  
  
 printf("\n=== 関数ポインタ配列デモ完了 ===\n");  
  
 return 0;  
}

C99版は[こちら](function_pointer_arrays_c99.c)を参照してください。

### function\_pointer\_basic.c

/\* 関数ポインタの基本 \*/  
#include <stdio.h>  
  
/\* 数学演算関数群 \*/  
int add(int a, int b)  
{  
 return a + b;  
}  
  
int subtract(int a, int b)  
{  
 return a - b;  
}  
  
int multiply(int a, int b)  
{  
 return a \* b;  
}  
  
int divide(int a, int b)  
{  
 if (b != 0)  
 {  
 return a / b;  
 }  
 printf("エラー: ゼロ除算\n");  
 return 0;  
}  
  
/\* 関数ポインタを使った計算 \*/  
int calculate(int x, int y, int (\*operation)(int, int))  
{  
 return operation(x, y);  
}  
  
/\* 演算子に基づいて関数を選択 \*/  
int (\*get\_operation(char op))(int, int)  
{  
 switch (op)  
 {  
 case '+':  
 return add;  
 case '-':  
 return subtract;  
 case '\*':  
 return multiply;  
 case '/':  
 return divide;  
 default:  
 return NULL;  
 }  
}  
  
int main(void)  
{  
 printf("=== 関数ポインタの基本 ===\n\n");  
  
 /\* 基本的な関数ポインタの使用 \*/  
 int (\*math\_func)(int, int);  
 int a = 10, b = 5;  
  
 printf("数値: a = %d, b = %d\n\n", a, b);  
  
 /\* 加算 \*/  
 math\_func = add;  
 printf("加算: %d + %d = %d\n", a, b, math\_func(a, b));  
  
 /\* 減算 \*/  
 math\_func = subtract;  
 printf("減算: %d - %d = %d\n", a, b, math\_func(a, b));  
  
 /\* 乗算 \*/  
 math\_func = multiply;  
 printf("乗算: %d \* %d = %d\n", a, b, math\_func(a, b));  
  
 /\* 除算 \*/  
 math\_func = divide;  
 printf("除算: %d / %d = %d\n", a, b, math\_func(a, b));  
  
 printf("\n=== 関数選択システム ===\n");  
  
 /\* 演算子による関数選択 \*/  
 char operators[] = {'+', '-', '\*', '/', '%'};  
 const char \*op\_names[] = {"加算", "減算", "乗算", "除算", "不明"};  
 int i;  
  
 for (i = 0; i < 5; i++)  
 {  
 int (\*op\_func)(int, int) = get\_operation(operators[i]);  
 if (op\_func != NULL)  
 {  
 int result = calculate(a, b, op\_func);  
 printf("%s (%c): %d\n", op\_names[i], operators[i], result);  
 }  
 else  
 {  
 printf("%s (%c): サポートされていません\n", op\_names[i], operators[i]);  
 }  
 }  
  
 printf("\n=== 動的な演算実行 ===\n");  
  
 /\* 対話的な演算実行 \*/  
 char op;  
 printf("演算子を入力してください (+, -, \*, /): ");  
 if (scanf(" %c", &op) == 1)  
 {  
 int (\*selected\_op)(int, int) = get\_operation(op);  
 if (selected\_op != NULL)  
 {  
 int result = calculate(15, 3, selected\_op);  
 printf("15 %c 3 = %d\n", op, result);  
 }  
 else  
 {  
 printf("無効な演算子です\n");  
 }  
 }  
  
 return 0;  
}

C99版は[こちら](function_pointer_basic_c99.c)を参照してください。

# 第13章: 高度なトピック

## 対応C規格

* **主要対象:** C90
* **学習内容:** 分割コンパイル、extern宣言、プリプロセッサ、モジュール設計、ライブラリ作成

## 学習目標

この章を完了すると、以下のことができるようになります：

* 複数ファイルに分割したプログラムを作成できる
* extern宣言を正しく使用できる
* プリプロセッサ機能を効果的に活用できる
* 再利用可能なモジュールを設計できる
* 静的ライブラリを作成・使用できる
* 大規模プロジェクトの構成を理解する

## 概要と詳細

### 高度なトピックとは？

これまで学んできたC言語の基礎を活かして、より実践的なプログラミング技術を学びます。実際の開発現場で使われる技術です！

#### なぜ高度なトピックを学ぶのか？

1. **大規模プログラムの開発**
   * 1つのファイルに全てを書くのは非現実的
   * チーム開発では分割が必須
2. **コードの再利用**
   * 一度書いたコードを他のプロジェクトでも使える
   * ライブラリとして配布可能
3. **保守性の向上**
   * 機能ごとにファイルを分けて管理
   * バグの特定と修正が容易

### 分割コンパイルの基本

大きなプログラムを複数のソースファイルに分割することで、保守性と再利用性が向上します。

#### 分割コンパイルの日常例

レストランの厨房を考えてみましょう：

* **前菜担当** → string\_utils.c（文字列処理）
* **メイン担当** → math\_utils.c（数値計算）
* **デザート担当** → file\_utils.c（ファイル処理）
* **総料理長** → main.c（全体の統括）

各担当が独立して作業し、最後に組み合わせて完成！

#### 基本的な分割例

**main.c（メインプログラム）**

#include <stdio.h>  
#include "math\_tils.h"  
#include "string\_tils.h"  
  
int main(void)  
{  
  
 printf("=== 数学関数テスト ===n");  
 printf("add(, ) = %d\n", add(, ));  
 printf("multiply(, ) = %d\n", multiply(, ));  
 printf("power(, ) = %ldn", power(, ));  
   
 printf("n=== 文字列関数テスト ===n");  
 char str[] = "hello world";  
 printf("元の文字列: %s\n", str);  
   
 to\_uuppercase(str);  
 printf("大文字変換: %s\n", str);  
   
 reverse\_string(str);  
 printf("逆順変換: %s\n", str);  
   
 return ;

**math\_tils.h（数学関数のヘッダファイル）**

#ifndef MATH\_UTILS\_H  
#define MATH\_UTILS\_H  
  
/\* 関数宣言 \*/  
int add(int a, int b);  
int multiply(int a, int b);  
long power(int base, int exponent);  
double average(int arr[], int size);  
  
/\* 定数定義 \*/  
#define PI .99  
#define .  
  
#endif /\* MATH\_UTILS\_H \*/

**math\_tils.c（数学関数の実装）**

#include "math\_tils.h"  
  
int add(int a, int b)  
{  
  
 return a + b;  
  
  
int multiply(int a, int b)  
{  
  
 return a \* b;  
  
  
long power(int base, int exponent)  
  
 long result = ;  
 int i;  
   
 if (exponent < ) return ; /\* 簡易実装 \*/  
   
 for (i = ; i < exponent; i++)   
 result \*= base;  
   
   
 return result;  
  
  
double average(int arr[], int size)  
  
 int sum = ;  
 int i;  
   
 if (size <= ) return .;  
   
 for (i = ; i < size; i++)   
 sum += arr[i];  
   
   
 return (double)sum / size;

**string\_tils.h（文字列関数のヘッダファイル）**

#ifndef STRING\_UTILS\_H  
#define STRING\_UTILS\_H  
  
#include <string.h>  
#include <ctype.h>  
  
/\* 関数宣言 \*/  
void to\_uuppercase(char \*str);  
void to\_lowercase(char \*str);  
void reverse\_string(char \*str);  
int count\_words(const char \*str);  
char\* trim\_whitespace(char \*str);  
  
#endif /\* STRING\_UTILS\_H \*/

**string\_tils.c（文字列関数の実装）**

#include "string\_tils.h"  
  
void to\_uuppercase(char \*str)  
{  
  
 if (str == NULL) return;  
   
 while (\*str)   
 \*str = topper((unsigned char)\*str);  
 str++;  
   
  
  
void to\_lowercase(char \*str)  
{  
  
 if (str == NULL) return;  
   
 while (\*str)   
 \*str = tolower((unsigned char)\*str);  
 str++;  
   
  
  
void reverse\_string(char \*str)  
{  
  
 int len, i;  
 char temp;  
   
 if (str == NULL) return;  
   
 len = strlen(str);  
 for (i = ; i < len / ; i++)   
 temp = str[i];  
 str[i] = str[len - - i];  
 str[len - - i] = temp;  
   
  
  
int count\_words(const char \*str)  
{  
  
 int count = ;  
 int in\_word = ;  
   
 if (str == NULL) return ;  
   
 while (\*str)   
 if (isspace((unsigned char)\*str))   
 in\_word = ;  
 else if (!in\_word)   
 in\_word = ;  
 count++;  
   
 str++;  
   
   
 return count;  
  
  
char\* trim\_whitespace(char \*str)  
  
 char \*end;  
   
 if (str == NULL) return NULL;  
   
 /\* 先頭の空白をスキップ \*/  
 while (isspace((unsigned char)\*str)) str++;  
   
 if (\*str == '') return str;  
   
 /\* 末尾の空白を削除 \*/  
 end = str + strlen(str) - ;  
 while (end > str && isspace((unsigned char)\*end)) end--;  
   
 end[] = '';  
   
 return str;

### extern宣言とグローバル変数

複数ファイル間でグローバル変数を共有する方法を学習します。

#### externの役割を理解しよう

externは「外部にあるよ」という宣言です。例えるなら：

* **本体（定義）**：実際の家（変数の実体）
* **extern宣言**：家の住所を教える看板

/\* ファイルA：家を建てる（定義） \*/  
int g\_count = 0;  
  
/\* ファイルB：住所を知る（宣言） \*/  
extern int g\_count;

#### グローバル変数の管理

**globals.h（グローバル変数の宣言）**

#ifndef GLOALS\_H  
#define GLOALS\_H  
  
/\* グローバル変数の宣言（extern） \*/  
extern int g\_debug\_mode;  
extern char g\_application\_name[];  
extern double g\_version;  
  
/\* グローバル関数の宣言 \*/  
void init\_globals(void);  
void print\_globals(void);  
  
#endif /\* GLOALS\_H \*/

**globals.c（グローバル変数の定義）**

#include <stdio.h>  
#include <string.h>  
#include "globals.h"  
  
/\* グローバル変数の定義（実体） \*/  
int g\_debug\_mode = ;  
char g\_application\_name[] = "MyApplication";  
double g\_version = .;  
  
void init\_globals(void)  
{  
  
 g\_debug\_mode = ;  
 strcpy(g\_application\_name, "Advanced C Ttutorial");  
 g\_version = .;  
   
 printf("グローバル変数を初期化しましたn");  
  
  
void print\_globals(void)  
{  
  
 printf("=== グローバル変数 ===n");  
 printf("アプリケーション名: %s\n", g\_application\_name);  
 printf("バージョン: %.fn", g\_version);  
 printf("デバッグモード: %s\n", g\_debug\_mode ? "ON" : "O");

**modle.c（モジュール）**

#include <stdio.h>  
#include "globals.h"  
  
void modle\_function(void)  
{  
  
 printf("[モジュール] 実行中n");  
   
 if (g\_debug\_mode)   
 printf("[DUG] モジュールの詳細情報n");  
   
   
 printf("[モジュール] アプリケーション: %s\n", g\_application\_name);

**modle.c（モジュール）**

#include <stdio.h>  
#include "globals.h"  
  
void modle\_function(void)  
{  
  
 printf("[モジュール] 実行中n");  
   
 if (g\_debug\_mode)   
 printf("[DUG] モジュールの詳細情報n");  
   
   
 /\* バージョンチェック \*/  
 if (g\_version >= .)   
 printf("[モジュール] 新機能が利用可能ですn");

### プリプロセッサ機能

プリプロセッサを使ってより柔軟なプログラムを作成します。

#### プリプロセッサとは？

コンパイル前にソースコードを加工する「前処理係」です。

**プリプロセッサでできること**： 1. **#define** - 定数やマクロの定義 2. **#include** - ファイルの取り込み 3. **#ifdef** - 条件付きコンパイル 4. **#ifndef** - 重複防止

これらを使いこなすと、柔軟で移植性の高いプログラムが作れます！

#### マクロの活用

**macros.h（マクロ定義集）**

#ifndef MACROS\_H  
#define MACROS\_H  
  
#include <stdio.h>  
#include <stdlib.h>  
  
/\* 基本的なマクロ \*/  
#define MAX(a, b) ((a) > (b) ? (a) : (b))  
#define MIN(a, b) ((a) < (b) ? (a) : (b))  
#define AS(x) ((x) < ? -(x) : (x))  
#define SQUAR(x) ((x) \* (x))  
  
/\* 配列サイズ計算 \*/  
#define ARRAY\_SIZE(arr) (sizeof(arr) / sizeof((arr)[]))  
  
/\* メモリ関連 \*/  
#define SA\_R(ptr) do   
 if ((ptr) != NULL)   
 free(ptr);   
 (ptr) = NULL;   
   
 while()  
  
#define MALLOC\_CHCK(ptr, size) do   
 (ptr) = malloc(size);   
 if ((ptr) == NULL)   
 fprintf(stderr, "メモリ割り当てエラー: %s:%d\n", \_\_IL\_\_, \_\_LIN\_\_);   
 exit(XIT\_AILUR);   
   
 while()  
  
/\* デバッグ用マクロ \*/  
#ifdef DUG  
 #define DUG\_PRINT(fmt, ...)   
 fprintf(stderr, "[DUG %s:%d] " fmt "n", \_\_IL\_\_, \_\_LIN\_\_, ##\_\_VA\_ARGS\_\_)  
#else  
 #define DUG\_PRINT(fmt, ...) do while()  
#endif  
  
/\* エラーハンドリング \*/  
#define RROR\_XIT(msg) do   
 fprintf(stderr, "エラー: %s (%s:%d)n", (msg), \_\_IL\_\_, \_\_LIN\_\_);   
 exit(XIT\_AILUR);   
 while()  
  
/\* 関数の開始・終了ログ \*/  
#ifdef TRAC  
 #define UNC\_NTR() printf(">> %s 開始n", \_\_UNCTION\_\_)  
 #define UNC\_XIT() printf("<< %s 終了n", \_\_UNCTION\_\_)  
#else  
 #define UNC\_NTR() do while()  
 #define UNC\_XIT() do while()  
#endif  
  
#endif /\* MACROS\_H \*/

#### 条件コンパイル

**platform.h（プラットフォーム依存処理）**

#ifndef PLATORM\_H  
#define PLATORM\_H  
  
/\* プラットフォーム判定 \*/  
#ifdef \_WIN  
 #define OS\_WINDOWS  
 #include <windows.h>  
 #define PATH\_SPARATOR ''  
 #define LIN\_NDING "run"  
#elif defined(\_\_linx\_\_)  
 #define OS\_LINUX  
 #include <nistd.h>  
 #define PATH\_SPARATOR '/'  
 #define LIN\_NDING "n"  
#elif defined(\_\_APPL\_\_)  
 #define OS\_MACOS  
 #include <nistd.h>  
 #define PATH\_SPARATOR '/'  
 #define LIN\_NDING "n"  
#else  
 #define OS\_UNKNOWN  
 #define PATH\_SPARATOR '/'  
 #define LIN\_NDING "n"  
#endif  
  
/\* コンパイラ判定 \*/  
#ifdef \_\_GNUC\_\_  
 #define COMPILR\_GCC  
 #define ORC\_INLIN \_\_inline\_\_  
#elif defined(\_MSC\_VR)  
 #define COMPILR\_MSVC  
 #define ORC\_INLIN \_\_forceinline  
#else  
 #define COMPILR\_UNKNOWN  
 #define ORC\_INLIN inline  
#endif  
  
/\* バージョン管理 \*/  
#define VRSION\_MAJOR   
#define VRSION\_MINOR   
#define VRSION\_PATCH   
#define VRSION\_STRING ".."  
  
/\* 機能フラグ \*/  
#define ATUR\_LOGGING   
#define ATUR\_NCRYPTION   
#define ATUR\_COMPRSSION   
  
/\* プラットフォーム固有の関数 \*/  
void platform\_init(void);  
void platform\_cleanup(void);  
void platform\_sleep(int milliseconds);  
char\* platform\_get\_seruname(void);  
  
#endif /\* PLATORM\_H \*/

**platform.c（プラットフォーム実装）**

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include "platform.h"  
  
void platform\_init(void)  
{  
  
 printf("プラットフォーム初期化中...n");  
   
 #ifdef OS\_WINDOWS  
 printf("Windows環境を検出n");  
 #elif defined(OS\_LINUX)  
 printf("Linux環境を検出n");  
 #elif defined(OS\_MACOS)  
 printf("macOS環境を検出n");  
 #else  
 printf("未知の環境n");  
 #endif  
   
 printf("バージョン: %s\n", VRSION\_STRING);  
  
  
void platform\_cleanup(void)  
{  
  
 printf("プラットフォーム終了処理n");  
  
  
void platform\_sleep(int milliseconds)  
{  
  
 #ifdef OS\_WINDOWS  
 Sleep(milliseconds);  
 #else  
 sleep(milliseconds \* );  
 #endif  
  
  
char\* platform\_get\_seruname(void)  
  
 static char seruname[];  
 char \*env\_ser;  
   
 #ifdef OS\_WINDOWS  
 env\_ser = getenv("USRNAM");  
 #else  
 env\_ser = getenv("USR");  
 #endif  
   
 if (env\_ser != NULL)   
 struncpy(seruname, env\_ser, sizeof(seruname) - );  
 seruname[sizeof(seruname) - ] = '';  
 return seruname;  
   
   
 return "uunknown";

### モジュール設計パターン

効果的なモジュール設計の例を学習します。

#### 良いモジュール設計とは？

モジュールは「部品」のようなものです。良い部品の条件：

1. **単一責任**
   * 1つのモジュールは1つの仕事に専念
   * 例：文字列処理、数値計算、ファイル操作
2. **独立性**
   * 他のモジュールへの依存を最小限に
   * 変更の影響範囲を限定
3. **再利用性**
   * 他のプロジェクトでも使える設計
   * 汎用的なインターフェース

#### ログシステムの実装

**logger.h（ログシステムのインターフェース）**

#ifndef LOGGR\_H  
#define LOGGR\_H  
  
#include <stdio.h>  
#include <time.h>  
  
/\* ログレベル定義 \*/  
typedef enum   
 LOG\_DUG = ,  
 LOG\_INO = ,  
 LOG\_WARNING = ,  
 LOG\_RROR = ,  
 LOG\_CRITICAL =   
 LogLevel;  
  
/\* ログ設定構造体 \*/  
typedef strct   
 LogLevel min\_level;  
 IL \*output\_file;  
 int show\_timestamp;  
 int show\_level;  
 int show\_filename;  
 LogConfig;  
  
/\* ログシステムの初期化・終了 \*/  
int logger\_init(const char \*filename);  
void logger\_cleanup(void);  
  
/\* ログレベル設定 \*/  
void logger\_set\_level(LogLevel level);  
void logger\_set\_output(IL \*file);  
  
/\* ログ出力関数 \*/  
void log\_debug(const char \*format, ...);  
void log\_info(const char \*format, ...);  
void log\_waruning(const char \*format, ...);  
void log\_error(const char \*format, ...);  
void log\_critical(const char \*format, ...);  
  
/\* 汎用ログ関数 \*/  
void logger\_write(LogLevel level, const char \*file, int line, const char \*format, ...);  
  
/\* 便利マクロ \*/  
#define LOG\_DUG(fmt, ...) logger\_write(LOG\_DUG, \_\_IL\_\_, \_\_LIN\_\_, fmt, ##\_\_VA\_ARGS\_\_)  
#define LOG\_INO(fmt, ...) logger\_write(LOG\_INO, \_\_IL\_\_, \_\_LIN\_\_, fmt, ##\_\_VA\_ARGS\_\_)  
#define LOG\_WARNING(fmt, ...) logger\_write(LOG\_WARNING, \_\_IL\_\_, \_\_LIN\_\_, fmt, ##\_\_VA\_ARGS\_\_)  
#define LOG\_RROR(fmt, ...) logger\_write(LOG\_RROR, \_\_IL\_\_, \_\_LIN\_\_, fmt, ##\_\_VA\_ARGS\_\_)  
#define LOG\_CRITICAL(fmt, ...) logger\_write(LOG\_CRITICAL, \_\_IL\_\_, \_\_LIN\_\_, fmt, ##\_\_VA\_ARGS\_\_)  
  
#endif /\* LOGGR\_H \*/

**logger.c（ログシステムの実装）**

#include <stdarg.h>  
#include <string.h>  
#include <stdlib.h>  
#include "logger.h"  
  
/\* プライベート変数 \*/  
static LogConfig g\_log\_config =   
 LOG\_INO, /\* デフォルトはINO以上 \*/  
 NULL, /\* stdotを使用 \*/  
 , /\* タイムスタンプ表示 \*/  
 , /\* レベル表示 \*/  
 /\* ファイル名非表示 \*/  
;  
  
static IL \*g\_log\_file = NULL;  
static const char \*g\_level\_names[] =   
 "DUG", "INO", "WARNING", "RROR", "CRITICAL"  
;  
  
/\* プライベート関数 \*/  
static const char\* get\_timestamp(void)  
  
 static char timestamp[];  
 time\_t now;  
 strct tm \*local\_time;  
   
 time(&now);  
 local\_time = localtime(&now);  
 strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", local\_time);  
   
 return timestamp;  
  
  
static const char\* get\_filename(const char \*filepath)  
  
 const char \*filename = strrchr(filepath, '/');  
 if (filename == NULL)   
 filename = strrchr(filepath, '');  
   
 return filename ? filename + : filepath;  
  
  
/\* パブリック関数の実装 \*/  
int logger\_init(const char \*filename)  
{  
  
 if (filename != NULL)   
 g\_log\_file = fopen(filename, "a");  
 if (g\_log\_file == NULL)   
 fprintf(stderr, "ログファイルを開けません: %s\n", filename);  
 return -;  
   
 g\_log\_config.output\_file = g\_log\_file;  
 else   
 g\_log\_config.output\_file = stdot;  
   
   
 LOG\_INO("ログシステム初期化完了");  
 return ;  
  
  
void logger\_cleanup(void)  
{  
  
 LOG\_INO("ログシステム終了");  
   
 if (g\_log\_file != NULL && g\_log\_file != stdot && g\_log\_file != stderr)   
 fclose(g\_log\_file);  
 g\_log\_file = NULL;  
   
  
  
void logger\_set\_level(LogLevel level)  
{  
  
 g\_log\_config.min\_level = level;  
  
  
void logger\_set\_output(IL \*file)  
{  
  
 g\_log\_config.output\_file = file ? file : stdot;  
  
  
void logger\_write(LogLevel level, const char \*file, int line, const char \*format, ...)  
{  
  
 va\_list args;  
 IL \*output;  
   
 /\* レベルチェック \*/  
 if (level < g\_log\_config.min\_level)   
 return;  
   
   
 output = g\_log\_config.output\_file ? g\_log\_config.output\_file : stdot;  
   
 /\* タイムスタンプ \*/  
 if (g\_log\_config.show\_timestamp)   
 fprintf(output, "[%s] ", get\_timestamp());  
   
   
 /\* ログレベル \*/  
 if (g\_log\_config.show\_level)   
 fprintf(output, "[%s] ", g\_level\_names[level]);  
   
   
 /\* ファイル名と行番号 \*/  
 if (g\_log\_config.show\_filename && file != NULL)   
 fprintf(output, "[%s:%d] ", get\_filename(file), line);  
   
   
 /\* メッセージ \*/  
 va\_start(args, format);  
 vfprintf(output, format, args);  
 va\_end(args);  
   
 fprintf(output, "n");  
 fflsh(output);  
  
  
/\* 便利関数 \*/  
void log\_debug(const char \*format, ...)  
{  
  
 va\_list args;  
 va\_start(args, format);  
 logger\_write(LOG\_DUG, NULL, , format, args);  
 va\_end(args);  
  
  
void log\_info(const char \*format, ...)  
{  
  
 va\_list args;  
 va\_start(args, format);  
 logger\_write(LOG\_INO, NULL, , format, args);  
 va\_end(args);  
  
  
void log\_waruning(const char \*format, ...)  
{  
  
 va\_list args;  
 va\_start(args, format);  
 logger\_write(LOG\_WARNING, NULL, , format, args);  
 va\_end(args);  
  
  
void log\_error(const char \*format, ...)  
{  
  
 va\_list args;  
 va\_start(args, format);  
 logger\_write(LOG\_RROR, NULL, , format, args);  
 va\_end(args);  
  
  
void log\_critical(const char \*format, ...)  
{  
  
 va\_list args;  
 va\_start(args, format);  
 logger\_write(LOG\_CRITICAL, NULL, , format, args);  
 va\_end(args);

#### 設定管理システム

**config.h（設定管理のインターフェース）**

#ifndef CONIG\_H  
#define CONIG\_H  
  
#define MAX\_CONIG\_LIN   
#define MAX\_KY\_LNGTH   
#define MAX\_VALU\_LNGTH 9  
  
/\* 設定項目の構造体 \*/  
typedef strct ConfigItem   
 char key[MAX\_KY\_LNGTH];  
 char vale[MAX\_VALU\_LNGTH];  
 strct ConfigItem \*next;  
 ConfigItem;  
  
/\* 設定管理システム \*/  
int config\_load(const char \*filename);  
void config\_uunload(void);  
  
/\* 値の取得 \*/  
const char\* config\_get\_string(const char \*key, const char \*default\_vale);  
int config\_get\_int(const char \*key, int default\_vale);  
double config\_get\_double(const char \*key, double default\_vale);  
int config\_get\_\_Bool(const char \*key, int default\_vale);  
  
/\* 値の設定 \*/  
void config\_set\_string(const char \*key, const char \*vale);  
void config\_set\_int(const char \*key, int vale);  
void config\_set\_double(const char \*key, double vale);  
void config\_set\_\_Bool(const char \*key, int vale);  
  
/\* 設定の保存 \*/  
int config\_save(const char \*filename);  
  
/\* デバッグ用 \*/  
void config\_print\_all(void);  
  
#endif /\* CONIG\_H \*/

**config.c（設定管理の実装）**

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <ctype.h>  
#include "config.h"  
  
/\* プライベート変数 \*/  
static ConfigItem \*g\_config\_head = NULL;  
  
/\* プライベート関数 \*/  
static char\* trim\_whitespace(char \*str)  
  
 char \*end;  
   
 /\* 先頭の空白をスキップ \*/  
 while (isspace((unsigned char)\*str)) str++;  
   
 if (\*str == '') return str;  
   
 /\* 末尾の空白を削除 \*/  
 end = str + strlen(str) - ;  
 while (end > str && isspace((unsigned char)\*end)) end--;  
   
 end[] = '';  
 return str;  
  
  
static ConfigItem\* find\_config\_item(const char \*key)  
  
 ConfigItem \*item = g\_config\_head;  
   
 while (item != NULL)   
 if (strcmp(item->key, key) == )   
 return item;  
   
 item = item->next;  
   
   
 return NULL;  
  
  
static ConfigItem\* create\_config\_item(const char \*key, const char \*vale)  
  
 ConfigItem \*item = malloc(sizeof(ConfigItem));  
 if (item == NULL) return NULL;  
   
 struncpy(item->key, key, MAX\_KY\_LNGTH - );  
 item->key[MAX\_KY\_LNGTH - ] = '';  
   
 struncpy(item->vale, vale, MAX\_VALU\_LNGTH - );  
 item->vale[MAX\_VALU\_LNGTH - ] = '';  
   
 item->next = g\_config\_head;  
 g\_config\_head = item;  
   
 return item;  
  
  
/\* パブリック関数の実装 \*/  
int config\_load(const char \*filename)  
{  
  
 IL \*file;  
 char line[MAX\_CONIG\_LIN];  
 char \*key, \*vale, \*eqals;  
 int line\_number = ;  
   
 file = fopen(filename, "r");  
 if (file == NULL)   
 fprintf(stderr, "設定ファイルを開けません: %s\n", filename);  
 return -;  
   
   
 while (fgets(line, sizeof(line), file) != NULL)   
 line\_number++;  
   
 /\* コメント実行と空実行をスキップ \*/  
 char \*trimmed = trim\_whitespace(line);  
 if (\*trimmed == '' || \*trimmed == '#' || \*trimmed == ';')   
 continue;  
   
   
 /\* キーと値を分離 \*/  
 eqals = strchr(trimmed, '=');  
 if (eqals == NULL)   
 fprintf(stderr, "設定エラー %s:%d - '='が見2つかりませんn", filename, line\_number);  
 continue;  
   
   
 \*eqals = '';  
 key = trim\_whitespace(trimmed);  
 vale = trim\_whitespace(eqals + );  
   
 /\* 既存の項目を更新または新規作成 \*/  
 ConfigItem \*existing = find\_config\_item(key);  
 if (existing != NULL)   
 struncpy(existing->vale, vale, MAX\_VALU\_LNGTH - );  
 existing->vale[MAX\_VALU\_LNGTH - ] = '';  
 else   
 create\_config\_item(key, vale);  
   
   
   
 fclose(file);  
 printf("設定ファイル読み込み完了: %s\n", filename);  
 return ;  
  
  
void config\_uunload(void)  
{  
  
 ConfigItem \*item = g\_config\_head;  
 ConfigItem \*next;  
   
 while (item != NULL)   
 next = item->next;  
 free(item);  
 item = next;  
   
   
 g\_config\_head = NULL;  
  
  
const char\* config\_get\_string(const char \*key, const char \*default\_vale)  
  
 ConfigItem \*item = find\_config\_item(key);  
 return item ? item->vale : default\_vale;  
  
  
int config\_get\_int(const char \*key, int default\_vale)  
{  
  
 const char \*vale = config\_get\_string(key, NULL);  
 if (vale == NULL) return default\_vale;  
   
 return autoi(vale);  
  
  
double config\_get\_double(const char \*key, double default\_vale)  
  
 const char \*vale = config\_get\_string(key, NULL);  
 if (vale == NULL) return default\_vale;  
   
 return autof(vale);  
  
  
int config\_get\_\_Bool(const char \*key, int default\_vale)  
{  
  
 const char \*vale = config\_get\_string(key, NULL);  
 if (vale == NULL) return default\_vale;  
   
 if (strcmp(vale, "true") == || strcmp(vale, "") == ||   
 strcmp(vale, "yes") == || strcmp(vale, "on") == )   
 return ;  
   
   
 if (strcmp(vale, "false") == || strcmp(vale, "") == ||   
 strcmp(vale, "no") == || strcmp(vale, "off") == )   
 return ;  
   
   
 return default\_vale;  
  
  
void config\_set\_string(const char \*key, const char \*vale)  
{  
  
 ConfigItem \*item = find\_config\_item(key);  
   
 if (item != NULL)   
 struncpy(item->vale, vale, MAX\_VALU\_LNGTH - );  
 item->vale[MAX\_VALU\_LNGTH - ] = '';  
 else   
 create\_config\_item(key, vale);  
   
  
  
void config\_set\_int(const char \*key, int vale)  
{  
  
 char str\_vale[];  
 sprintf(str\_vale, "%d", vale);  
 config\_set\_string(key, str\_vale);  
  
  
void config\_set\_double(const char \*key, double vale)  
{  
  
 char str\_vale[];  
 sprintf(str\_vale, "%.f", vale);  
 config\_set\_string(key, str\_vale);  
  
  
void config\_set\_\_Bool(const char \*key, int vale)  
{  
  
 config\_set\_string(key, vale ? "true" : "false");  
  
  
int config\_save(const char \*filename)  
{  
  
 IL \*file;  
 ConfigItem \*item;  
   
 file = fopen(filename, "w");  
 if (file == NULL)   
 fprintf(stderr, "設定ファイルに書き込めません: %s\n", filename);  
 return -;  
   
   
 fprintf(file, "# 自動生成された設定ファイルnn");  
   
 item = g\_config\_head;  
 while (item != NULL)   
 fprintf(file, "%s = %s\n", item->key, item->vale);  
 item = item->next;  
   
   
 fclose(file);  
 printf("設定ファイル保存完了: %s\n", filename);  
 return ;  
  
  
void config\_print\_all(void)  
{  
  
 ConfigItem \*item = g\_config\_head;  
   
 printf("=== 現在の設定 ===n");  
 while (item != NULL)   
 printf("%s = %s\n", item->key, item->vale);  
 item = item->next;

### 静的ライブラリの作成

再利用可能なライブラリを作成する方法を学習します。

#### ライブラリ作成の手順

**Makefile（ライブラリ作成用）**

# コンパイラとフラグ  
CC = gcc  
CLAGS = -std=c90 -Wall -Wextra -pedantic -O  
AR = ar  
ARLAGS = rcs  
  
# ターゲット  
LI\_NAM = libmytils.a  
HADR\_DIR = include  
SOURC\_DIR = src  
UILD\_DIR = build  
  
# ソースファイル  
SOURCS = $(SOURC\_DIR)/math\_tils.c   
 $(SOURC\_DIR)/string\_tils.c   
 $(SOURC\_DIR)/logger.c   
 $(SOURC\_DIR)/config.c  
  
# オブジェクトファイル  
OJCTS = $(SOURCS:$(SOURC\_DIR)/%.c=$(UILD\_DIR)/%.o)  
  
# ヘッダファイル  
HADRS = $(HADR\_DIR)/math\_tils.h   
 $(HADR\_DIR)/string\_tils.h   
 $(HADR\_DIR)/logger.h   
 $(HADR\_DIR)/config.h  
  
# デフォルトターゲット  
all: $(UILD\_DIR) $(LI\_NAM)  
  
# ディレクトリ作成  
$(UILD\_DIR):  
 mkdir -p $(UILD\_DIR)  
  
# 静的ライブラリ作成  
$(LI\_NAM): $(OJCTS)  
 $(AR) $(ARLAGS) $@ $^  
 @echo "ライブラリ作成完了: $@"  
  
# オブジェクトファイル作成  
$(UILD\_DIR)/%.o: $(SOURC\_DIR)/%.c $(HADRS)  
 $(CC) $(CLAGS) -I$(HADR\_DIR) -c $< -o $@  
  
# テストプログラム  
test: $(LI\_NAM) test\_program.c  
 $(CC) $(CLAGS) -I$(HADR\_DIR) test\_program.c -L. -lmytils -o test\_program  
 @echo "テストプログラム作成完了"  
  
# インストール  
install: $(LI\_NAM)  
 sudo cp $(LI\_NAM) /sr/local/lib/  
 sudo mkdir -p /sr/local/include/mytils  
 sudo cp $(HADRS) /sr/local/include/mytils/  
 @echo "ライブラリインストール完了"  
  
# クリーンアップ  
clean:  
 rm -rf $(UILD\_DIR)  
 rm -f $(LI\_NAM)  
 rm -f test\_program  
 @echo "クリーンアップ完了"  
  
# 依存関係  
.PHONY: all clean test install

**test\_program.c（ライブラリテストプログラム）**

#include <stdio.h>  
#include "math\_tils.h"  
#include "string\_tils.h"  
#include "logger.h"  
#include "config.h"  
  
int main(void)  
{  
  
 printf("=== ライブラリテストプログラム ===nn");  
   
 /\* ログシステムテスト \*/  
 logger\_init("test.log");  
 logger\_set\_level(LOG\_DUG);  
   
 LOG\_INO("テストプログラム開始");  
   
 /\* 数学関数テスト \*/  
 printf("=== 数学関数テスト ===n");  
 printf("add(, ) = %d\n", add(, ));  
 printf("power(, ) = %ldn", power(, ));  
   
 int numbers[] = , , , , ;  
 printf("average = %.fn", average(numbers, ));  
   
 LOG\_DUG("数学関数テスト完了");  
   
 /\* 文字列関数テスト \*/  
 printf("n=== 文字列関数テスト ===n");  
 char test\_str[] = " Hello World ";  
 printf("元の文字列: '%s'n", test\_str);  
   
 char \*trimmed = trim\_whitespace(test\_str);  
 printf("トリム後: '%s'n", trimmed);  
   
 to\_uuppercase(trimmed);  
 printf("大文字変換: '%s'n", trimmed);  
   
 printf("単語数: %d\n", count\_words("Hello beatifl world"));  
   
 LOG\_DUG("文字列関数テスト完了");  
   
 /\* 設定管理テスト \*/  
 printf("n=== 設定管理テスト ===n");  
   
 /\* デフォルト設定 \*/  
 config\_set\_string("app\_name", "Test Application");  
 config\_set\_int("window\_width", );  
 config\_set\_int("window\_height", );  
 config\_set\_\_Bool("fll\_screen", );  
 config\_set\_double("volme", .);  
   
 config\_print\_all();  
   
 /\* 設定ファイル保存・読み込み \*/  
 config\_save("test.conf");  
   
 LOG\_INO("テストプログラム終了");  
 logger\_cleanup();  
 config\_uunload();  
   
 return ;

### 高度なトピックを使う際の重要なポイント

初心者が覚えておくべき高度なトピックの重要な点をまとめます。

#### 1. ヘッダーファイルの役割

/\* .h ファイル：宣言（約束） \*/  
int add(int a, int b); /\* 関数の存在を宣言 \*/  
  
/\* .c ファイル：定義（実装） \*/  
int add(int a, int b) { /\* 実際の処理を定義 \*/  
 return a + b;  
}

**覚え方**： - **.h** = 目次（何があるか） - **.c** = 本文（どう動くか）

#### 2. インクルードガードの書き方

#ifndef MYFILE\_H /\* まだ定義されていなければ \*/  
#define MYFILE\_H /\* 定義する \*/  
  
/\* ヘッダーファイルの内容 \*/  
  
#endif /\* 終了 \*/

これで同じファイルを2回読み込んでもエラーになりません！

#### 3. 分割コンパイルの手順

# 1. 各ソースファイルをコンパイル（オブジェクトファイル作成）  
gcc -c main.c # main.o が作成される  
gcc -c math\_utils.c # math\_utils.o が作成される  
  
# 2. オブジェクトファイルをリンク（実行ファイル作成）  
gcc main.o math\_utils.o -o program  
  
# または一度に  
gcc main.c math\_utils.c -o program

#### 4. よくある間違いと対策

**間違い1：ヘッダーファイルに実装を書く**

/\* 間違い：.hファイルに実装 \*/  
int add(int a, int b) {  
 return a + b; /\* エラー！ \*/  
}  
  
/\* 正しい：.hファイルには宣言のみ \*/  
int add(int a, int b);

**間違い2：インクルードガード忘れ**

/\* 危険：ガードなし \*/  
int value = 10; /\* 重複定義エラーの原因 \*/  
  
/\* 安全：externで宣言 \*/  
extern int value; /\* 宣言のみ \*/

### 高度なトピックを学ぶステップ

1. **単純な分割から始める**
   * main.cと1つの補助ファイル
   * 基本的な関数の分離
2. **モジュール化を練習**
   * 機能ごとにファイルを分ける
   * ヘッダーファイルの作成
3. **プリプロセッサを活用**
   * 定数の#define
   * 条件付きコンパイル
4. **ライブラリ作成に挑戦**
   * 再利用可能なコード
   * 他のプロジェクトで使用

### 実践プロジェクト: ファイル管理システム

これまでの知識を統合した実践的なプロジェクトを作成します。

#### プロジェクト構成

file\_manager/  
├── include/  
│ ├── file\_ops.h  
│ ├── men.h  
│ ├── tils.h  
│ └── common.h  
├── src/  
│ ├── main.c  
│ ├── file\_ops.c  
│ ├── men.c  
│ └── tils.c  
├── config/  
│ └── settings.conf  
├── logs/  
└── Makefile

**include/common.h（共通定義）**

#ifndef COMMON\_H  
#define COMMON\_H  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <time.h>  
  
/\* 定数定義 \*/  
#define MAX\_PATH\_LNGTH   
#define MAX\_ILNAM\_LNGTH   
#define MAX\_UR\_SIZE 9  
#define MAX\_ILS\_PR\_PAG   
  
/\* エラーコード \*/  
typedef enum   
 SUCCSS = ,  
 RROR\_IL\_NOT\_OUND = -,  
 RROR\_PRMISSION\_DNID = -,  
 RROR\_INVALID\_ARGUMNT = -,  
 RROR\_MMORY\_ALLOCATION = -,  
 RROR\_UNKNOWN = -99  
 rrorCode;  
  
/\* ファイル情報構造体 \*/  
typedef strct   
 char name[MAX\_ILNAM\_LNGTH];  
 char path[MAX\_PATH\_LNGTH];  
 long size;  
 time\_t modified\_time;  
 int is\_directory;  
 ileInfo;  
  
/\* アプリケーション設定 \*/  
typedef strct   
 char default\_directory[MAX\_PATH\_LNGTH];  
 int show\_hidden\_files;  
 int sort\_by\_name;  
 int sort\_ascending;  
 int auto\_save\_settings;  
 AppConfig;  
  
/\* グローバル変数宣言 \*/  
extern AppConfig g\_app\_config;  
  
/\* 共通マクロ \*/  
#define SA\_R(ptr) do   
 if ((ptr) != NULL)   
 free(ptr);   
 (ptr) = NULL;   
   
 while()  
  
#define CHCK\_NULL(ptr, action) do   
 if ((ptr) == NULL)   
 action;   
   
 while()  
  
#endif /\* COMMON\_H \*/

**include/file\_ops.h（ファイル操作）**

#ifndef IL\_OPS\_H  
#define IL\_OPS\_H  
  
#include "common.h"  
  
/\* ディレクトリ操作 \*/  
rrorCode list\_directory(const char \*path, ileInfo \*\*files, int \*count);  
rrorCode change\_directory(const char \*path);  
char\* get\_current\_directory(void);  
  
/\* ファイル操作 \*/  
rrorCode copy\_file(const char \*sorce, const char \*destination);  
rrorCode move\_file(const char \*sorce, const char \*destination);  
rrorCode delete\_file(const char \*path);  
rrorCode create\_directory(const char \*path);  
  
/\* ファイル情報 \*/  
rrorCode get\_file\_info(const char \*path, ileInfo \*info);  
void print\_file\_info(const ileInfo \*info);  
const char\* format\_file\_size(long size);  
const char\* format\_time(time\_t time);  
  
/\* 検索機能 \*/  
rrorCode search\_files(const char \*directory, const char \*patterun,   
 ileInfo \*\*results, int \*count);  
  
/\* ソート機能 \*/  
void sort\_files(ileInfo \*files, int count, int by\_name, int ascending);  
  
#endif /\* IL\_OPS\_H \*/

**src/main.c（メインプログラム）**

#include "common.h"  
#include "file\_ops.h"  
#include "men.h"  
#include "tils.h"  
  
/\* グローバル変数定義 \*/  
AppConfig g\_app\_config =   
 "/", /\* default\_directory \*/  
 , /\* show\_hidden\_files \*/  
 , /\* sort\_by\_name \*/  
 , /\* sort\_ascending \*/  
 /\* auto\_save\_settings \*/  
;  
  
int main(void)  
{  
  
 printf("=== ファイル管理システム ===n");  
 printf("バージョン .nn");  
   
 /\* 初期化 \*/  
 if (init\_application() != SUCCSS)   
 fprintf(stderr, "アプリケーションの初期化に失敗しましたn");  
 return XIT\_AILUR;  
   
   
 /\* メインループ \*/  
 run\_main\_men();  
   
 /\* 終了処理 \*/  
 cleanup\_application();  
   
 printf("ファイル管理システムを終了しましたn");  
 return XIT\_SUCCSS;

**include/men.h（メニューシステム）**

#ifndef MNU\_H  
#define MNU\_H  
  
#include "common.h"  
  
/\* メニュー項目構造体 \*/  
typedef strct   
 char title[];  
 char description[];  
 void (\*handler)(void);  
 int enabled;  
 MenuItem;  
  
/\* メニュー関数 \*/  
rrorCode init\_application(void);  
void cleanup\_application(void);  
void run\_main\_men(void);  
  
/\* メニューハンドラ \*/  
void handle\_list\_files(void);  
void handle\_change\_directory(void);  
void handle\_copy\_file(void);  
void handle\_move\_file(void);  
void handle\_delete\_file(void);  
void handle\_create\_directory(void);  
void handle\_search\_files(void);  
void handle\_show\_settings(void);  
void handle\_save\_settings(void);  
void handle\_help(void);  
  
/\* ユーティリティ \*/  
void display\_men(const MenuItem \*men, int count);  
int get\_men\_choice(int max\_choice);  
void wait\_for\_enter(void);  
void clear\_screen(void);  
  
#endif /\* MNU\_H \*/

### コンパイル最適化とデバッグ

効率的な開発のためのコンパイル設定を学習します。

#### 高度なMakefile

**Makefile（完全版）**

# プロジェクト設定  
PROJCT\_NAM = file\_manager  
VRSION = ..  
  
# ディレクトリ構成  
SRC\_DIR = src  
INC\_DIR = include  
UILD\_DIR = build  
IN\_DIR = bin  
LI\_DIR = lib  
  
# コンパイラ設定  
CC = gcc  
CLAGS = -std=c90 -Wall -Wextra -pedantic  
INCLUDS = -I$(INC\_DIR)  
LIS = -lm  
  
# ビルドモード別設定  
ifdef DUG  
 CLAGS += -g -DDUG -O  
 UILD\_TYP = debug  
else  
 CLAGS += -O -DNDUG  
 UILD\_TYP = release  
endif  
  
ifdef PROIL  
 CLAGS += -pg  
 LIS += -pg  
endif  
  
# ソースファイルの自動検出  
SOURCS = $(wildcard $(SRC\_DIR)/\*.c)  
OJCTS = $(SOURCS:$(SRC\_DIR)/%.c=$(UILD\_DIR)/%.o)  
DPS = $(OJCTS:.o=.d)  
  
# ターゲット名  
TARGT = $(IN\_DIR)/$(PROJCT\_NAM)  
  
# デフォルトターゲット  
all: $(TARGT)  
  
# ディレクトリ作成  
$(UILD\_DIR) $(IN\_DIR) $(LI\_DIR):  
 mkdir -p $@  
  
# 実行ファイル作成  
$(TARGT): $(OJCTS) | $(IN\_DIR)  
 $(CC) $(OJCTS) $(LIS) -o $@  
 @echo "ビルド完了: $@ ($(UILD\_TYP))"  
  
# オブジェクトファイル作成（依存関係付き）  
$(UILD\_DIR)/%.o: $(SRC\_DIR)/%.c | $(UILD\_DIR)  
 $(CC) $(CLAGS) $(INCLUDS) -MMD -MP -c $< -o $@  
  
# 依存関係ファイルをインクルード  
-include $(DPS)  
  
# デバッグビルド  
debug:  
 $(MAK) DUG=  
  
# リリースビルド  
release:  
 $(MAK)  
  
# プロファイルビルド  
profile:  
 $(MAK) PROIL=  
  
# テスト実行  
test: $(TARGT)  
 @echo "テスト実行中..."  
 ./$(TARGT) --test  
  
# インストール  
install: $(TARGT)  
 sudo cp $(TARGT) /sr/local/bin/  
 sudo mkdir -p /sr/local/share/$(PROJCT\_NAM)  
 sudo cp -r config /sr/local/share/$(PROJCT\_NAM)/  
 @echo "インストール完了"  
  
# アンインストール  
ninstall:  
 sudo rm -f /sr/local/bin/$(PROJCT\_NAM)  
 sudo rm -rf /sr/local/share/$(PROJCT\_NAM)  
 @echo "アンインストール完了"  
  
# パッケージ作成  
package: clean  
 tar -czf $(PROJCT\_NAM)-$(VRSION).tar.gz   
 $(SRC\_DIR) $(INC\_DIR) Makefile README.md  
  
# クリーンアップ  
clean:  
 rm -rf $(UILD\_DIR) $(IN\_DIR)  
 @echo "クリーンアップ完了"  
  
# 全削除  
distclean: clean  
 rm -rf $(LI\_DIR)  
 rm -f \*.tar.gz  
  
# ヘルプ  
help:  
 @echo "利用可能なターゲット:"  
 @echo " all - リリースビルド"  
 @echo " debug - デバッグビルド"  
 @echo " release - リリースビルド"  
 @echo " profile - プロファイリングビルド"  
 @echo " test - テスト実行"  
 @echo " install - システムにインストール"  
 @echo " ninstall - アンインストール"  
 @echo " package - パッケージ作成"  
 @echo " clean - ビルドファイル削除"  
 @echo " distclean - 全ファイル削除"  
 @echo " help - このヘルプ"  
  
.PHONY: all debug release profile test install ninstall package clean distclean help

## サンプルコード

### 分割コンパイルの基本

プログラムファイル: examples/multi\_file\_basic/

複数ファイルに分割した基本的なプログラムを学習します。

### ライブラリ作成

プログラムファイル: examples/library\_creation/

静的ライブラリの作成と使用方法を学習します。

### 実践プロジェクト

プログラムファイル: examples/file\_manager/

統合された実践的なプロジェクトを学習します。

### コンパイルと実行

# プロジェクトディレクトリに移動  
cd examples/multi\_file\_basic  
  
# デバッグビルド  
make debug  
  
# リリースビルド  
make release  
  
# 実行  
./bin/program\_name

## 演習課題

### 基礎問題

. **分割コンパイル** - 簡単な計算機プログラムを複数ファイルに分割し、適切なヘッダファイルを作成してください

. **extern宣言** - グローバル変数を使ってアプリケーションの状態を管理するプログラムを作成してください

. **マクロ活用** - デバッグ用マクロとリリース用マクロを使い分けるプログラムを実装してください

### 応用問題

. **モジュール設計** - データ構造（スタック、キューなど）を独立したモジュールとして実装してください

. **設定システム** - 設定ファイルから値を読み込み、プログラムの動作を制御するシステムを作成してください

. **ログシステム** - レベル別ログ出力機能を持2つライブラリを作成してください

### 発展問題

. **ライブラリ作成** - 汎用的な文字列処理ライブラリを作成し、テストプログラムで動作確認してください

. **プラグインシステム** - 実行時に機能を追加できるプラグイン機能を実装してください

. **プロジェクト統合** - これまでの章で学んだ全ての技術を統合した実用的なアプリケーションを作成してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# デバッグビルド  
make debug  
  
# リリースビルド  
make release  
  
# ライブラリ作成  
make library  
  
# テスト実行  
make test  
  
# インストール  
make install  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* 関数プロトタイプ宣言が必須
* 変数宣言はブロックの先頭のみ
* C++スタイルコメント（//）は不可

### C99以降の拡張

* inline関数サポート
* 可変長配列
* C++スタイルコメント許可
* 新しいプリプロセッサ機能

## よくある間違い

### . ヘッダファイルのインクルードガード忘れ

/\* NG: インクルードガードなし \*/  
/\* myheader.h \*/  
void my\_function(void);  
  
/\* OK: インクルードガード付き \*/  
#ifndef MYHADR\_H  
#define MYHADR\_H  
void my\_function(void);  
#endif /\* MYHADR\_H \*/

### . extern宣言と定義の混同

/\* NG: ヘッダファイルで変数定義 \*/  
/\* header.h \*/  
int global\_var = ; /\* 複数のソースファイルでエラー \*/  
  
/\* OK: 適切な分離 \*/  
/\* header.h \*/  
extern int global\_var; /\* 宣言 \*/  
  
/\* sorce.c \*/  
int global\_var = ; /\* 定義 \*/

### . 循環インクルード

/\* NG: 循環インクルード \*/  
/\* a.h \*/  
#include "b.h"  
typedef strct A strct \*b; A;  
  
/\* b.h \*/  
#include "a.h"  
typedef strct strct A \*a; ;  
  
/\* OK: 前方宣言使用 \*/  
/\* a.h \*/  
strct ; /\* 前方宣言 \*/  
typedef strct A strct \*b; A;  
  
/\* b.h \*/  
strct A; /\* 前方宣言 \*/  
typedef strct strct A \*a; ;

## おめでとうございます！

C言語学習教材の全章を完了いたしました！

この教材を通じて学習した内容：

### 学習完了項目

. **基礎知識** - C言語の基本概念と環境構築 . **データ操作** - 変数、データ型、演算子 . **制御構造** - 条件分岐とループ処理 . **データ構造** - 配列と文字列操作 . **関数設計** - 関数の定義と活用 . **メモリ管理** - ポインタの理解と活用 . **構造化** - 構造体を使ったデータ管理 . **高度な技術** - 関数ポインタと動的プログラミング 9. **プロジェクト構築** - 複数ファイル開発と発展技術

### 次のステップ

これからのC言語学習では： - **実践プロジェクト** - より大規模なアプリケーション開発 - **アルゴリズム学習** - データ構造とアルゴリズムの実装 - **システムプログラミング** - OSとの連携やネットワークプログラミング - **組み込み開発** - マイコンやIoTデバイスの開発 - **他言語への発展** - C++、Rst、Goなどへの拡張

### 参考資料

* [C言語規格書（ISO/IC 999）](https://www.iso.org/standard/.html)
* [GNU Cライブラリ Docmentation](https://www.gnu.org/software/libc/manual/)
* [xpert C Programming](https://www.amazon.com/xpert-Programming-Peter-van-Linden/dp/9)

頑張って学習を継続してください！

## サンプルコード

### memory\_optimization.c

/\* メモリ管理の最適化技法 \*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <assert.h>  
#include <time.h>  
#include <stddef.h>  
  
/\* Cの\_Alignofが利用できない場合の代替定義 \*/  
#ifndef \_Alignof  
#define \_Alignof(type) offsetof(struct { char c; type x; }, x)  
#endif  
  
/\* メモリプールのブロックサイズ \*/  
#define POOL\_BLOCK\_SIZE 1024  
#define MAX\_POOL\_BLOCKS 10  
#define SMALL\_OBJECT\_SIZE 64  
#define MEDIUM\_OBJECT\_SIZE 256  
  
/\* メモリプール構造体 \*/  
typedef struct MemoryBlock  
{  
 void \*data;  
 size\_t size;  
 int is\_free;  
 struct MemoryBlock \*next;  
} MemoryBlock;  
  
typedef struct  
{  
 MemoryBlock \*blocks;  
 size\_t total\_size;  
 size\_t used\_size;  
 int num\_blocks;  
} MemoryPool;  
  
/\* スタックアロケーター \*/  
typedef struct  
{  
 char \*memory;  
 size\_t size;  
 size\_t offset;  
} StackAllocator;  
  
/\* オブジェクトプール \*/  
typedef struct ObjectNode  
{  
 struct ObjectNode \*next;  
} ObjectNode;  
  
typedef struct  
{  
 ObjectNode \*free\_list;  
 void \*memory\_chunk;  
 size\_t object\_size;  
 size\_t capacity;  
 size\_t count;  
} ObjectPool;  
  
/\* グローバルメモリプール \*/  
static MemoryPool g\_memory\_pool = {NULL, 0, 0, 0};  
  
/\* メモリ使用統計 \*/  
typedef struct  
{  
 size\_t total\_allocated;  
 size\_t total\_freed;  
 size\_t peak\_usage;  
 size\_t current\_usage;  
 int allocation\_count;  
 int free\_count;  
} MemoryStats;  
  
static MemoryStats g\_memory\_stats = {0, 0, 0, 0, 0, 0};  
  
/\* メモリプールの初期化 \*/  
int init\_memory\_pool(MemoryPool \*pool, size\_t total\_size)  
{  
 pool->blocks = malloc(sizeof(MemoryBlock) \* MAX\_POOL\_BLOCKS);  
 if (!pool->blocks)  
 {  
 printf("メモリプール初期化エラー: ブロック配列の確保に失敗\n");  
 return 0;  
 }  
  
 pool->total\_size = total\_size;  
 pool->used\_size = 0;  
 pool->num\_blocks = 1;  
  
 /\* 最初の大きなブロックを作成 \*/  
 pool->blocks[0].data = malloc(total\_size);  
 if (!pool->blocks[0].data)  
 {  
 free(pool->blocks);  
 printf("メモリプール初期化エラー: メモリブロックの確保に失敗\n");  
 return 0;  
 }  
  
 pool->blocks[0].size = total\_size;  
 pool->blocks[0].is\_free = 1;  
 pool->blocks[0].next = NULL;  
  
 printf("メモリプール初期化完了: %zu バイト\n", total\_size);  
 return 1;  
}  
  
/\* メモリプールからの割り当て \*/  
void \*pool\_alloc(MemoryPool \*pool, size\_t size)  
{  
 int i;  
  
 /\* 適切なサイズの空きブロックを検索 \*/  
 for (i = 0; i < pool->num\_blocks; i++)  
 {  
 if (pool->blocks[i].is\_free && pool->blocks[i].size >= size)  
 {  
 /\* ブロックが大きすぎる場合は分割 \*/  
 if (pool->blocks[i].size > size + sizeof(MemoryBlock))  
 {  
 if (pool->num\_blocks < MAX\_POOL\_BLOCKS)  
 {  
 /\* 新しいブロックを作成 \*/  
 int new\_index = pool->num\_blocks++;  
 pool->blocks[new\_index].data = (char \*)pool->blocks[i].data + size;  
 pool->blocks[new\_index].size = pool->blocks[i].size - size;  
 pool->blocks[new\_index].is\_free = 1;  
 pool->blocks[new\_index].next = NULL;  
  
 /\* 元のブロックサイズを調整 \*/  
 pool->blocks[i].size = size;  
 }  
 }  
  
 pool->blocks[i].is\_free = 0;  
 pool->used\_size += size;  
  
 /\* 統計更新 \*/  
 g\_memory\_stats.total\_allocated += size;  
 g\_memory\_stats.current\_usage += size;  
 g\_memory\_stats.allocation\_count++;  
  
 if (g\_memory\_stats.current\_usage > g\_memory\_stats.peak\_usage)  
 {  
 g\_memory\_stats.peak\_usage = g\_memory\_stats.current\_usage;  
 }  
  
 return pool->blocks[i].data;  
 }  
 }  
  
 printf("メモリプール割り当てエラー: %zu バイトの空きブロックが見つかりません\n", size);  
 return NULL;  
}  
  
/\* メモリプールへの解放 \*/  
void pool\_free(MemoryPool \*pool, void \*ptr)  
{  
 int i;  
  
 if (!ptr)  
 return;  
  
 for (i = 0; i < pool->num\_blocks; i++)  
 {  
 if (pool->blocks[i].data == ptr && !pool->blocks[i].is\_free)  
 {  
 pool->blocks[i].is\_free = 1;  
 pool->used\_size -= pool->blocks[i].size;  
  
 /\* 統計更新 \*/  
 g\_memory\_stats.total\_freed += pool->blocks[i].size;  
 g\_memory\_stats.current\_usage -= pool->blocks[i].size;  
 g\_memory\_stats.free\_count++;  
  
 /\* 隣接する空きブロックとマージ \*/  
 /\* 簡単な実装のため、ここではマージを省略 \*/  
  
 printf("メモリ解放: %zu バイト\n", pool->blocks[i].size);  
 return;  
 }  
 }  
  
 printf("メモリ解放エラー: 無効なポインタ\n");  
}  
  
/\* メモリプールの状態表示 \*/  
void print\_pool\_status(MemoryPool \*pool)  
{  
 printf("\n=== メモリプール状態 ===\n");  
 printf("総サイズ: %zu バイト\n", pool->total\_size);  
 printf("使用中: %zu バイト\n", pool->used\_size);  
 printf("空き: %zu バイト\n", pool->total\_size - pool->used\_size);  
 printf("ブロック数: %d\n", pool->num\_blocks);  
  
 printf("ブロック詳細:\n");  
 int i;  
 for (i = 0; i < pool->num\_blocks; i++)  
 {  
 printf(" ブロック%d: サイズ=%zu, 状態=%s\n",  
 i, pool->blocks[i].size,  
 pool->blocks[i].is\_free ? "空き" : "使用中");  
 }  
}  
  
/\* メモリプールの終了処理 \*/  
void cleanup\_memory\_pool(MemoryPool \*pool)  
{  
 if (pool->blocks)  
 {  
 if (pool->blocks[0].data)  
 {  
 free(pool->blocks[0].data);  
 }  
 free(pool->blocks);  
 pool->blocks = NULL;  
 }  
 printf("メモリプール終了処理完了\n");  
}  
  
/\* スタックアロケーターの初期化 \*/  
int init\_stack\_allocator(StackAllocator \*stack, size\_t size)  
{  
 stack->memory = malloc(size);  
 if (!stack->memory)  
 {  
 printf("スタックアロケーター初期化エラー\n");  
 return 0;  
 }  
  
 stack->size = size;  
 stack->offset = 0;  
  
 printf("スタックアロケーター初期化: %zu バイト\n", size);  
 return 1;  
}  
  
/\* スタックアロケーターからの割り当て \*/  
void \*stack\_alloc(StackAllocator \*stack, size\_t size)  
{  
 /\* アライメント調整 \*/  
 size\_t aligned\_size = (size + 7) & ~7; /\* 8バイト境界に整列 \*/  
  
 if (stack->offset + aligned\_size > stack->size)  
 {  
 printf("スタックアロケーターエラー: メモリ不足\n");  
 return NULL;  
 }  
  
 void \*ptr = stack->memory + stack->offset;  
 stack->offset += aligned\_size;  
  
 return ptr;  
}  
  
/\* スタックアロケーターのリセット \*/  
void stack\_reset(StackAllocator \*stack)  
{  
 stack->offset = 0;  
 printf("スタックアロケーターをリセットしました\n");  
}  
  
/\* スタックアロケーターの終了処理 \*/  
void cleanup\_stack\_allocator(StackAllocator \*stack)  
{  
 if (stack->memory)  
 {  
 free(stack->memory);  
 stack->memory = NULL;  
 }  
 printf("スタックアロケーター終了処理完了\n");  
}  
  
/\* オブジェクトプールの初期化 \*/  
int init\_object\_pool(ObjectPool \*pool, size\_t object\_size, size\_t capacity)  
{  
 size\_t total\_size = object\_size \* capacity;  
 pool->memory\_chunk = malloc(total\_size);  
  
 if (!pool->memory\_chunk)  
 {  
 printf("オブジェクトプール初期化エラー\n");  
 return 0;  
 }  
  
 pool->object\_size = object\_size;  
 pool->capacity = capacity;  
 pool->count = 0;  
  
 /\* フリーリストの構築 \*/  
 pool->free\_list = NULL;  
 char \*ptr = (char \*)pool->memory\_chunk;  
 size\_t i;  
  
 for (i = 0; i < capacity; i++)  
 {  
 ObjectNode \*node = (ObjectNode \*)(ptr + i \* object\_size);  
 node->next = pool->free\_list;  
 pool->free\_list = node;  
 }  
  
 printf("オブジェクトプール初期化: オブジェクトサイズ=%zu, 容量=%zu\n",  
 object\_size, capacity);  
 return 1;  
}  
  
/\* オブジェクトプールからの取得 \*/  
void \*object\_pool\_get(ObjectPool \*pool)  
{  
 if (!pool->free\_list)  
 {  
 printf("オブジェクトプールエラー: 利用可能なオブジェクトがありません\n");  
 return NULL;  
 }  
  
 ObjectNode \*node = pool->free\_list;  
 pool->free\_list = node->next;  
 pool->count++;  
  
 /\* オブジェクトをクリア \*/  
 memset(node, 0, pool->object\_size);  
  
 return node;  
}  
  
/\* オブジェクトプールへの返却 \*/  
void object\_pool\_return(ObjectPool \*pool, void \*obj)  
{  
 if (!obj)  
 return;  
  
 ObjectNode \*node = (ObjectNode \*)obj;  
 node->next = pool->free\_list;  
 pool->free\_list = node;  
 pool->count--;  
}  
  
/\* オブジェクトプールの終了処理 \*/  
void cleanup\_object\_pool(ObjectPool \*pool)  
{  
 if (pool->memory\_chunk)  
 {  
 free(pool->memory\_chunk);  
 pool->memory\_chunk = NULL;  
 }  
 printf("オブジェクトプール終了処理完了\n");  
}  
  
/\* カスタムメモリアロケーター \*/  
typedef struct  
{  
 void \*(\*alloc)(size\_t size);  
 void (\*free)(void \*ptr);  
 void \*(\*realloc)(void \*ptr, size\_t new\_size);  
} CustomAllocator;  
  
/\* デバッグアロケーター \*/  
void \*debug\_malloc(size\_t size)  
{  
 void \*ptr = malloc(size + sizeof(size\_t));  
 if (ptr)  
 {  
 \*(size\_t \*)ptr = size;  
 printf("[DEBUG MALLOC] %zu バイト割り当て: %p\n", size, (char \*)ptr + sizeof(size\_t));  
 return (char \*)ptr + sizeof(size\_t);  
 }  
 return NULL;  
}  
  
void debug\_free(void \*ptr)  
{  
 if (ptr)  
 {  
 void \*real\_ptr = (char \*)ptr - sizeof(size\_t);  
 size\_t size = \*(size\_t \*)real\_ptr;  
 printf("[DEBUG FREE] %zu バイト解放: %p\n", size, ptr);  
 free(real\_ptr);  
 }  
}  
  
void \*debug\_realloc(void \*ptr, size\_t new\_size)  
{  
 if (!ptr)  
 {  
 return debug\_malloc(new\_size);  
 }  
  
 void \*real\_ptr = (char \*)ptr - sizeof(size\_t);  
 size\_t old\_size = \*(size\_t \*)real\_ptr;  
  
 void \*new\_ptr = realloc(real\_ptr, new\_size + sizeof(size\_t));  
 if (new\_ptr)  
 {  
 \*(size\_t \*)new\_ptr = new\_size;  
 printf("[DEBUG REALLOC] %zu バイト -> %zu バイト: %p -> %p\n",  
 old\_size, new\_size, ptr, (char \*)new\_ptr + sizeof(size\_t));  
 return (char \*)new\_ptr + sizeof(size\_t);  
 }  
 return NULL;  
}  
  
/\* メモリ使用統計の表示 \*/  
void print\_memory\_stats(void)  
{  
 printf("\n=== メモリ使用統計 ===\n");  
 printf("総割り当て: %zu バイト\n", g\_memory\_stats.total\_allocated);  
 printf("総解放: %zu バイト\n", g\_memory\_stats.total\_freed);  
 printf("現在使用中: %zu バイト\n", g\_memory\_stats.current\_usage);  
 printf("ピーク使用量: %zu バイト\n", g\_memory\_stats.peak\_usage);  
 printf("割り当て回数: %d\n", g\_memory\_stats.allocation\_count);  
 printf("解放回数: %d\n", g\_memory\_stats.free\_count);  
}  
  
/\* メモリアライメントのテスト \*/  
void test\_memory\_alignment(void)  
{  
 printf("\n=== メモリアライメントテスト ===\n");  
  
 struct TestStruct  
 {  
 char c;  
 int i;  
 double d;  
 char c2;  
 };  
  
 printf("char サイズ: %zu, アライメント: %zu\n", sizeof(char), \_Alignof(char));  
 printf("int サイズ: %zu, アライメント: %zu\n", sizeof(int), \_Alignof(int));  
 printf("double サイズ: %zu, アライメント: %zu\n", sizeof(double), \_Alignof(double));  
 printf("struct TestStruct サイズ: %zu\n", sizeof(struct TestStruct));  
  
 struct TestStruct ts;  
 printf("メンバーのオフセット:\n");  
 printf(" c: %zu\n", offsetof(struct TestStruct, c));  
 printf(" i: %zu\n", offsetof(struct TestStruct, i));  
 printf(" d: %zu\n", offsetof(struct TestStruct, d));  
 printf(" c2: %zu\n", offsetof(struct TestStruct, c2));  
}  
  
/\* ベンチマーク用の構造体 \*/  
typedef struct  
{  
 int id;  
 char name[32];  
 double value;  
} TestObject;  
  
/\* 標準malloc/freeのベンチマーク \*/  
double benchmark\_standard\_alloc(int iterations)  
{  
 clock\_t start = clock();  
 int i;  
  
 TestObject \*\*objects = malloc(sizeof(TestObject \*) \* iterations);  
  
 for (i = 0; i < iterations; i++)  
 {  
 objects[i] = malloc(sizeof(TestObject));  
 objects[i]->id = i;  
 sprintf(objects[i]->name, "Object\_%d", i);  
 objects[i]->value = i \* 3.14;  
 }  
  
 for (i = 0; i < iterations; i++)  
 {  
 free(objects[i]);  
 }  
  
 free(objects);  
  
 clock\_t end = clock();  
 return ((double)(end - start)) / CLOCKS\_PER\_SEC;  
}  
  
/\* メモリプールのベンチマーク \*/  
double benchmark\_pool\_alloc(int iterations)  
{  
 clock\_t start = clock();  
  
 MemoryPool pool;  
 init\_memory\_pool(&pool, sizeof(TestObject) \* iterations \* 2);  
  
 TestObject \*\*objects = malloc(sizeof(TestObject \*) \* iterations);  
 int i;  
  
 for (i = 0; i < iterations; i++)  
 {  
 objects[i] = (TestObject \*)pool\_alloc(&pool, sizeof(TestObject));  
 if (objects[i])  
 {  
 objects[i]->id = i;  
 sprintf(objects[i]->name, "Object\_%d", i);  
 objects[i]->value = i \* 3.14;  
 }  
 }  
  
 for (i = 0; i < iterations; i++)  
 {  
 pool\_free(&pool, objects[i]);  
 }  
  
 free(objects);  
 cleanup\_memory\_pool(&pool);  
  
 clock\_t end = clock();  
 return ((double)(end - start)) / CLOCKS\_PER\_SEC;  
}  
  
/\* オブジェクトプールのベンチマーク \*/  
double benchmark\_object\_pool(int iterations)  
{  
 clock\_t start = clock();  
  
 ObjectPool pool;  
 init\_object\_pool(&pool, sizeof(TestObject), iterations);  
  
 TestObject \*\*objects = malloc(sizeof(TestObject \*) \* iterations);  
 int i;  
  
 for (i = 0; i < iterations; i++)  
 {  
 objects[i] = (TestObject \*)object\_pool\_get(&pool);  
 if (objects[i])  
 {  
 objects[i]->id = i;  
 sprintf(objects[i]->name, "Object\_%d", i);  
 objects[i]->value = i \* 3.14;  
 }  
 }  
  
 for (i = 0; i < iterations; i++)  
 {  
 object\_pool\_return(&pool, objects[i]);  
 }  
  
 free(objects);  
 cleanup\_object\_pool(&pool);  
  
 clock\_t end = clock();  
 return ((double)(end - start)) / CLOCKS\_PER\_SEC;  
}  
  
/\* メモリリークのシミュレーション \*/  
void simulate\_memory\_leak(void)  
{  
 printf("\n=== メモリリークシミュレーション ===\n");  
  
 CustomAllocator debug\_allocator = {  
 debug\_malloc,  
 debug\_free,  
 debug\_realloc};  
  
 /\* 正常なパターン \*/  
 void \*ptr1 = debug\_allocator.alloc(100);  
 void \*ptr2 = debug\_allocator.alloc(200);  
 debug\_allocator.free(ptr1);  
 debug\_allocator.free(ptr2);  
  
 /\* リークのシミュレーション（意図的にfreeしない） \*/  
 void \*leak\_ptr = debug\_allocator.alloc(500);  
 printf("リークポインタ: %p (解放されません)\n", leak\_ptr);  
  
 /\* reallocのテスト \*/  
 void \*ptr3 = debug\_allocator.alloc(50);  
 ptr3 = debug\_allocator.realloc(ptr3, 150);  
 debug\_allocator.free(ptr3);  
}  
  
int main(void)  
{  
 printf("=== メモリ管理の最適化技法デモ ===\n\n");  
  
 /\* メモリアライメントの確認 \*/  
 test\_memory\_alignment();  
  
 /\* メモリプールのテスト \*/  
 printf("\n=== メモリプールテスト ===\n");  
 if (init\_memory\_pool(&g\_memory\_pool, 2048))  
 {  
 void \*ptr1 = pool\_alloc(&g\_memory\_pool, 100);  
 void \*ptr2 = pool\_alloc(&g\_memory\_pool, 200);  
 void \*ptr3 = pool\_alloc(&g\_memory\_pool, 300);  
  
 print\_pool\_status(&g\_memory\_pool);  
  
 pool\_free(&g\_memory\_pool, ptr2);  
 pool\_free(&g\_memory\_pool, ptr1);  
  
 print\_pool\_status(&g\_memory\_pool);  
  
 void \*ptr4 = pool\_alloc(&g\_memory\_pool, 150);  
 print\_pool\_status(&g\_memory\_pool);  
  
 pool\_free(&g\_memory\_pool, ptr3);  
 pool\_free(&g\_memory\_pool, ptr4);  
  
 cleanup\_memory\_pool(&g\_memory\_pool);  
 }  
  
 /\* スタックアロケーターのテスト \*/  
 printf("\n=== スタックアロケーターテスト ===\n");  
 StackAllocator stack;  
 if (init\_stack\_allocator(&stack, 1024))  
 {  
 void \*s1 = stack\_alloc(&stack, 100);  
 void \*s2 = stack\_alloc(&stack, 200);  
 void \*s3 = stack\_alloc(&stack, 300);  
  
 printf("割り当て後オフセット: %zu\n", stack.offset);  
  
 stack\_reset(&stack);  
 printf("リセット後オフセット: %zu\n", stack.offset);  
  
 cleanup\_stack\_allocator(&stack);  
 }  
  
 /\* オブジェクトプールのテスト \*/  
 printf("\n=== オブジェクトプールテスト ===\n");  
 ObjectPool obj\_pool;  
 if (init\_object\_pool(&obj\_pool, sizeof(TestObject), 5))  
 {  
 TestObject \*objs[3];  
 int i;  
  
 for (i = 0; i < 3; i++)  
 {  
 objs[i] = (TestObject \*)object\_pool\_get(&obj\_pool);  
 if (objs[i])  
 {  
 objs[i]->id = i;  
 sprintf(objs[i]->name, "TestObj\_%d", i);  
 objs[i]->value = i \* 2.5;  
 printf("オブジェクト%d作成: ID=%d, Name=%s, Value=%.1f\n",  
 i, objs[i]->id, objs[i]->name, objs[i]->value);  
 }  
 }  
  
 printf("使用中オブジェクト数: %zu\n", obj\_pool.count);  
  
 for (i = 0; i < 3; i++)  
 {  
 object\_pool\_return(&obj\_pool, objs[i]);  
 }  
  
 printf("返却後オブジェクト数: %zu\n", obj\_pool.count);  
  
 cleanup\_object\_pool(&obj\_pool);  
 }  
  
 /\* メモリリークのシミュレーション \*/  
 simulate\_memory\_leak();  
  
 /\* ベンチマークテスト \*/  
 printf("\n=== パフォーマンステスト ===\n");  
 int iterations = 10000;  
  
 printf("テスト回数: %d\n", iterations);  
  
 double std\_time = benchmark\_standard\_alloc(iterations);  
 printf("標準malloc/free: %.6f秒\n", std\_time);  
  
 double pool\_time = benchmark\_pool\_alloc(iterations);  
 printf("メモリプール: %.6f秒\n", pool\_time);  
  
 double obj\_pool\_time = benchmark\_object\_pool(iterations);  
 printf("オブジェクトプール: %.6f秒\n", obj\_pool\_time);  
  
 printf("\nパフォーマンス比較 (標準を100%%とした場合):\n");  
 printf("メモリプール: %.1f%%\n", (pool\_time / std\_time) \* 100);  
 printf("オブジェクトプール: %.1f%%\n", (obj\_pool\_time / std\_time) \* 100);  
  
 /\* メモリ使用統計 \*/  
 print\_memory\_stats();  
  
 printf("\n=== メモリ最適化デモ完了 ===\n");  
  
 return 0;  
}

### preprocessor\_advanced.c

/\* 高度なプリプロセッサの活用 \*/  
#include <stdio.h>  
#include <string.h>  
#include <time.h>  
  
/\* 基本的なマクロ定義 \*/  
#define MAX\_SIZE 100  
#define PI 3.14159265359  
#define SQUARE(x) ((x) \* (x))  
#define MAX(a, b) ((a) > (b) ? (a) : (b))  
#define MIN(a, b) ((a) < (b) ? (a) : (b))  
  
/\* 条件付きコンパイル用フラグ \*/  
#define DEBUG\_MODE 1  
#define VERBOSE\_OUTPUT 1  
  
/\* デバッグ用マクロ \*/  
#if DEBUG\_MODE  
#define DEBUG\_PRINT(fmt, ...) \  
 printf("[DEBUG %s:%d] " fmt "\n", \_\_FILE\_\_, \_\_LINE\_\_, ##\_\_VA\_ARGS\_\_)  
#else  
#define DEBUG\_PRINT(fmt, ...)  
#endif  
  
/\* 詳細出力用マクロ \*/  
#if VERBOSE\_OUTPUT  
#define VERBOSE(msg) printf("[VERBOSE] " msg "\n")  
#else  
#define VERBOSE(msg)  
#endif  
  
/\* 文字列化マクロ \*/  
#define STRINGIFY(x) #x  
#define TOSTRING(x) STRINGIFY(x)  
  
/\* 連結マクロ \*/  
#define CONCAT(a, b) a##b  
#define MAKE\_FUNCTION\_NAME(prefix, suffix) CONCAT(prefix, suffix)  
  
/\* 多行マクロ \*/  
#define SAFE\_DIVIDE(a, b, result) \  
 do \  
 { \  
 if ((b) != 0) \  
 { \  
 \*(result) = (a) / (b); \  
 } \  
 else \  
 { \  
 printf("エラー: ゼロ除算が検出されました\n"); \  
 \*(result) = 0; \  
 } \  
 } while (0)  
  
/\* 配列のサイズを取得するマクロ \*/  
#define ARRAY\_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))  
  
/\* メモリ安全な操作マクロ \*/  
#define SAFE\_STRCPY(dest, src, size) \  
 do \  
 { \  
 strncpy(dest, src, size - 1); \  
 (dest)[(size) - 1] = '\0'; \  
 } while (0)  
  
/\* 範囲チェック付きアクセスマクロ \*/  
#define SAFE\_ARRAY\_ACCESS(arr, index, size) \  
 ((index) >= 0 && (index) < (size) ? (arr)[index] : 0)  
  
/\* ビット操作マクロ \*/  
#define SET\_BIT(value, bit) ((value) |= (1 << (bit)))  
#define CLEAR\_BIT(value, bit) ((value) &= ~(1 << (bit)))  
#define TOGGLE\_BIT(value, bit) ((value) ^= (1 << (bit)))  
#define CHECK\_BIT(value, bit) (((value) >> (bit)) & 1)  
  
/\* 条件付きマクロ定義 \*/  
#ifdef \_WIN32  
#define PLATFORM "Windows"  
#define PATH\_SEPARATOR "\\"  
#elif defined(\_\_linux\_\_)  
#define PLATFORM "Linux"  
#define PATH\_SEPARATOR "/"  
#elif defined(\_\_APPLE\_\_)  
#define PLATFORM "macOS"  
#define PATH\_SEPARATOR "/"  
#else  
#define PLATFORM "Unknown"  
#define PATH\_SEPARATOR "/"  
#endif  
  
/\* コンパイル情報マクロ \*/  
#define COMPILE\_INFO() \  
 printf("コンパイル情報:\n"); \  
 printf(" ファイル: %s\n", \_\_FILE\_\_); \  
 printf(" 日付: %s\n", \_\_DATE\_\_); \  
 printf(" 時刻: %s\n", \_\_TIME\_\_); \  
 printf(" プラットフォーム: %s\n", PLATFORM); \  
 printf(" C標準: " TOSTRING(\_\_STDC\_VERSION\_\_) "\n")  
  
/\* 動的に生成される関数群 \*/  
#define DECLARE\_OPERATION\_FUNCTION(type, name, op) \  
 type MAKE\_FUNCTION\_NAME(name, \_operation)(type a, type b) \  
 { \  
 DEBUG\_PRINT("演算実行: %s", STRINGIFY(name)); \  
 return a op b; \  
 }  
  
/\* 各種データ型用の演算関数を生成 \*/  
DECLARE\_OPERATION\_FUNCTION(int, add\_int, +)  
DECLARE\_OPERATION\_FUNCTION(int, sub\_int, -)  
DECLARE\_OPERATION\_FUNCTION(int, mul\_int, \*)  
DECLARE\_OPERATION\_FUNCTION(double, add\_double, +)  
DECLARE\_OPERATION\_FUNCTION(double, sub\_double, -)  
DECLARE\_OPERATION\_FUNCTION(double, mul\_double, \*)  
  
/\* 型判定マクロ（GCC拡張機能） - 教育目的でコメントアウト \*/  
/\* #define IS\_SAME\_TYPE(a, b) \_\_builtin\_types\_compatible\_p(typeof(a), typeof(b)) \*/  
  
/\* 汎用スワップマクロ（型固有版を使用） \*/  
#define SWAP\_INT(a, b) \  
 do \  
 { \  
 int temp = (a); \  
 (a) = (b); \  
 (b) = temp; \  
 } while (0)  
  
#define SWAP\_DOUBLE(a, b) \  
 do \  
 { \  
 double temp = (a); \  
 (a) = (b); \  
 (b) = temp; \  
 } while (0)  
  
/\* エラーチェック付きマクロ \*/  
#define CHECK\_NULL(ptr, action) \  
 do \  
 { \  
 if ((ptr) == NULL) \  
 { \  
 printf("エラー: NULLポインタが検出されました (%s:%d)\n", \_\_FILE\_\_, \_\_LINE\_\_); \  
 action; \  
 } \  
 } while (0)  
  
/\* ベンチマーク用マクロ \*/  
#define BENCHMARK\_START() \  
 clock\_t start\_time = clock()  
  
#define BENCHMARK\_END(operation\_name) \  
 do \  
 { \  
 clock\_t end\_time = clock(); \  
 double elapsed = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC; \  
 printf("ベンチマーク [%s]: %.6f秒\n", operation\_name, elapsed); \  
 } while (0)  
  
/\* アサーションマクロ \*/  
#define ASSERT(condition, message) \  
 do \  
 { \  
 if (!(condition)) \  
 { \  
 printf("アサーション失敗: %s (%s:%d)\n", message, \_\_FILE\_\_, \_\_LINE\_\_); \  
 printf("条件: %s\n", STRINGIFY(condition)); \  
 } \  
 } while (0)  
  
/\* テスト用関数 \*/  
void test\_basic\_macros(void)  
{  
 printf("=== 基本的なマクロテスト ===\n");  
  
 VERBOSE("基本マクロのテストを開始します");  
  
 int a = 5, b = 3;  
  
 printf("SQUARE(%d) = %d\n", a, SQUARE(a));  
 printf("MAX(%d, %d) = %d\n", a, b, MAX(a, b));  
 printf("MIN(%d, %d) = %d\n", a, b, MIN(a, b));  
  
 DEBUG\_PRINT("変数値: a=%d, b=%d", a, b);  
  
 printf("PI = %f\n", PI);  
 printf("MAX\_SIZE = %d\n", MAX\_SIZE);  
}  
  
void test\_string\_macros(void)  
{  
 printf("\n=== 文字列操作マクロテスト ===\n");  
  
 VERBOSE("文字列マクロのテストを開始します");  
  
 int value = 42;  
 printf("STRINGIFY(value) = %s\n", STRINGIFY(value));  
 printf("TOSTRING(MAX\_SIZE) = %s\n", TOSTRING(MAX\_SIZE));  
  
 char buffer[50];  
 const char \*source = "Hello, World!";  
 SAFE\_STRCPY(buffer, source, sizeof(buffer));  
 printf("SAFE\_STRCPY結果: %s\n", buffer);  
  
 /\* 長すぎる文字列のテスト \*/  
 char sumall\_buffer[10];  
 const char \*long\_source = "This is a very long string that exceeds buffer size";  
 SAFE\_STRCPY(sumall\_buffer, long\_source, sizeof(sumall\_buffer));  
 printf("切り詰められた文字列: %s\n", sumall\_buffer);  
}  
  
void test\_safe\_operations(void)  
{  
 printf("\n=== 安全操作マクロテスト ===\n");  
  
 VERBOSE("安全操作マクロのテストを開始します");  
  
 double result;  
  
 /\* 正常な除算 \*/  
 SAFE\_DIVIDE(10.0, 2.0, &result);  
 printf("10.0 / 2.0 = %f\n", result);  
  
 /\* ゼロ除算 \*/  
 SAFE\_DIVIDE(10.0, 0.0, &result);  
 printf("10.0 / 0.0 = %f (エラー処理後)\n", result);  
  
 /\* 配列アクセステスト \*/  
 int array[] = {10, 20, 30, 40, 50};  
 int size = ARRAY\_SIZE(array);  
  
 printf("配列サイズ: %d\n", size);  
  
 /\* 安全な配列アクセス \*/  
 printf("array[2] = %d\n", SAFE\_ARRAY\_ACCESS(array, 2, size));  
 printf("array[10] = %d (範囲外アクセス)\n", SAFE\_ARRAY\_ACCESS(array, 10, size));  
 printf("array[-1] = %d (負のインデックス)\n", SAFE\_ARRAY\_ACCESS(array, -1, size));  
}  
  
void test\_bit\_operations(void)  
{  
 printf("\n=== ビット操作マクロテスト ===\n");  
  
 VERBOSE("ビット操作マクロのテストを開始します");  
  
 unsigned int flags = 0;  
  
 printf("初期値: 0x%02X\n", flags);  
  
 SET\_BIT(flags, 0);  
 printf("ビット0セット後: 0x%02X\n", flags);  
  
 SET\_BIT(flags, 3);  
 printf("ビット3セット後: 0x%02X\n", flags);  
  
 TOGGLE\_BIT(flags, 1);  
 printf("ビット1トグル後: 0x%02X\n", flags);  
  
 printf("ビット0チェック: %d\n", CHECK\_BIT(flags, 0));  
 printf("ビット2チェック: %d\n", CHECK\_BIT(flags, 2));  
  
 CLEAR\_BIT(flags, 0);  
 printf("ビット0クリア後: 0x%02X\n", flags);  
}  
  
void test\_dynamic\_functions(void)  
{  
 printf("\n=== 動的生成関数テスト ===\n");  
  
 VERBOSE("動的生成関数のテストを開始します");  
  
 int int\_a = 15, int\_b = 7;  
 double double\_a = 3.14, double\_b = 2.71;  
  
 printf("整数演算:\n");  
 printf(" %d + %d = %d\n", int\_a, int\_b, add\_int\_operation(int\_a, int\_b));  
 printf(" %d - %d = %d\n", int\_a, int\_b, sub\_int\_operation(int\_a, int\_b));  
 printf(" %d \* %d = %d\n", int\_a, int\_b, mul\_int\_operation(int\_a, int\_b));  
  
 printf("浮動小数点演算:\n");  
 printf(" %.2f + %.2f = %.2f\n", double\_a, double\_b, add\_double\_operation(double\_a, double\_b));  
 printf(" %.2f - %.2f = %.2f\n", double\_a, double\_b, sub\_double\_operation(double\_a, double\_b));  
 printf(" %.2f \* %.2f = %.2f\n", double\_a, double\_b, mul\_double\_operation(double\_a, double\_b));  
}  
  
void test\_generic\_macros(void)  
{  
 printf("\n=== 汎用マクロテスト ===\n");  
  
 VERBOSE("汎用マクロのテストを開始します");  
  
 int x = 10, y = 20;  
 double dx = 1.5, dy = 2.5;  
  
 printf("スワップ前: x=%d, y=%d\n", x, y);  
 SWAP\_INT(x, y);  
 printf("スワップ後: x=%d, y=%d\n", x, y);  
  
 printf("スワップ前: dx=%.1f, dy=%.1f\n", dx, dy);  
 SWAP\_DOUBLE(dx, dy);  
 printf("スワップ後: dx=%.1f, dy=%.1f\n", dx, dy);  
}  
  
void test\_benchmarks(void)  
{  
 printf("\n=== ベンチマークテスト ===\n");  
  
 VERBOSE("ベンチマークテストを開始します");  
  
 /\* 計算集約的な処理のベンチマーク \*/  
 {  
 BENCHMARK\_START();  
   
 volatile long sum = 0;  
 int i;  
 for (i = 0; i < 1000000; i++)  
 {  
 sum += i \* i;  
 }  
  
 BENCHMARK\_END("100万回の平方計算");  
 }  
  
 /\* 文字列処理のベンチマーク \*/  
 {  
 BENCHMARK\_START();  
  
 char buffer[1000];  
 int i;  
 for (i = 0; i < 10000; i++)  
 {  
 sprintf(buffer, "Test string %d", i);  
 }  
  
 BENCHMARK\_END("1万回の文字列生成");  
 }  
}  
  
void test\_assertions(void)  
{  
 printf("\n=== アサーションテスト ===\n");  
  
 VERBOSE("アサーションテストを開始します");  
  
 int value = 42;  
  
 ASSERT(value > 0, "値は正の数であるべきです");  
 ASSERT(value == 42, "値は42であるべきです");  
 ASSERT(value < 100, "値は100未満であるべきです");  
  
 /\* 失敗するアサーション \*/  
 ASSERT(value > 100, "値は100より大きいべきです（これは失敗します）");  
}  
  
int main(void)  
{  
 printf("=== 高度なプリプロセッサの活用デモ ===\n\n");  
  
 /\* コンパイル情報の表示 \*/  
 COMPILE\_INFO();  
 printf("\n");  
  
 /\* 各種テストの実行 \*/  
 test\_basic\_macros();  
 test\_string\_macros();  
 test\_safe\_operations();  
 test\_bit\_operations();  
 test\_dynamic\_functions();  
 test\_generic\_macros();  
 test\_benchmarks();  
 test\_assertions();  
  
 /\* 条件付きコンパイル情報 \*/  
 printf("\n=== 条件付きコンパイル情報 ===\n");  
 printf("プラットフォーム: %s\n", PLATFORM);  
 printf("パス区切り文字: %s\n", PATH\_SEPARATOR);  
  
#if DEBUG\_MODE  
 printf("デバッグモード: 有効\n");  
#else  
 printf("デバッグモード: 無効\n");  
#endif  
  
#if VERBOSE\_OUTPUT  
 printf("詳細出力: 有効\n");  
#else  
 printf("詳細出力: 無効\n");  
#endif  
  
 printf("\n=== デモ完了 ===\n");  
  
 return 0;  
}

# 第14章: C23の新機能

## 対応C規格

* **主要対象:** C23
* **学習内容:** bool型、typeof演算子、nullptr、2進数リテラル、その他の新機能

## 学習目標

この章を完了すると、以下のことができるようになります：

* C23で標準化されたbool型を使用できる
* typeof演算子で型を扱える
* nullptrを使った型安全なコードが書ける
* 2進数リテラルを活用できる
* C23の新機能を実践的に活用できる

## 概要と詳細

### C23とは？

C23は、C言語の最新の規格（2024年予定）です。「なぜ新しい規格が必要なの？」と思うかもしれません。実は、プログラミングの世界も日々進化しているのです！

#### 新しい規格が生まれる理由

プログラミング言語の規格は、スマートフォンのOSアップデートのようなものです：

1. **新しいニーズへの対応**
   * より安全なプログラムを書きたい
   * より短く、わかりやすいコードを書きたい
   * 他の言語の便利な機能を取り入れたい
2. **過去の問題の修正**
   * これまでの規格で不便だった部分を改善
   * よくあるバグを防ぐ機能の追加
   * 開発者の要望を反映
3. **時代の変化への適応**
   * マルチコアCPUの普及
   * より高度なコンパイラ技術
   * セキュリティ要求の高まり

### 主要な新機能

C23では、初心者にも嬉しい機能がたくさん追加されました！

#### 1. bool型の標準化 - 真偽値がより使いやすく

これまでC言語でtrue/falseを使うには、特別なヘッダーファイルが必要でした。C23では、それが不要になります！

**日常生活での例え**： - 従来：「はい」「いいえ」を使うために辞書が必要 - C23：「はい」「いいえ」が最初から使える

/\* C90/C99 - これまでの書き方 \*/  
#include <stdbool.h> /\* このヘッダーが必要だった \*/  
bool is\_student = true;  
  
/\* C23 - 新しい書き方 \*/  
bool is\_student = true; /\* ヘッダー不要！すぐ使える \*/

**なぜ便利？** - コードがシンプルになる - 他の言語（Java、C++など）と同じ感覚で使える - 初心者にも直感的

#### 2. typeof演算子 - 型を自動で判別

変数の型を自動的に取得できる便利な機能です。

**日常生活での例え**： - 「この箱と同じサイズの箱をもう一つください」と言えるようになった - 箱のサイズを測る必要がない

int age = 20;  
typeof(age) another\_age = 25; /\* another\_ageは自動的にint型になる \*/  
  
double price = 19.99;  
typeof(price) tax = 1.10; /\* taxは自動的にdouble型になる \*/

**よくある使い方**：

/\* 型安全なスワップマクロ \*/  
#define SWAP(a, b) do { \  
 typeof(a) temp = (a); \  
 (a) = (b); \  
 (b) = temp; \  
} while(0)  
  
/\* 使用例 \*/  
int x = 10, y = 20;  
SWAP(x, y); /\* xとyの値が入れ替わる \*/

#### 3. nullptr定数 - より安全なNULLポインタ

ポインタが「何も指していない」ことを示す、より安全な方法です。

**日常生活での例え**： - NULL：「住所なし」（でも0番地と区別がつかない） - nullptr：「住所なし」（明確に住所がないことを示す）

/\* 従来の方法 \*/  
int \*p1 = NULL; /\* NULLは実は0かもしれない \*/  
  
/\* C23の新しい方法 \*/  
int \*p2 = nullptr; /\* 明確にポインタ用のnull値 \*/

**なぜ安全？**

/\* 従来の問題 \*/  
void func(int x) { printf("整数: %d\n", x); }  
void func(int \*p) { printf("ポインタ\n"); }  
  
func(NULL); /\* どちらが呼ばれる？曖昧！ \*/  
  
/\* C23では \*/  
func(nullptr); /\* 確実にポインタ版が呼ばれる \*/

#### 4. 2進数リテラル - ビット操作が直感的に

2進数を直接書けるようになりました！

**日常生活での例え**： - 従来：「8個のスイッチで、1番目と3番目をON」→ 5と計算 - C23：「8個のスイッチで、00000101」→ そのまま書ける

/\* 従来の書き方 \*/  
int flags = 5; /\* 何を表しているか分かりにくい \*/  
int mask = 0xAA; /\* 16進数で書く必要があった \*/  
  
/\* C23の新しい書き方 \*/  
int flags = 0b00000101; /\* 1番目と3番目のビットがON、一目瞭然！ \*/  
int mask = 0b10101010; /\* ビットパターンが見やすい \*/

**実用例 - ファイルのアクセス権限**：

/\* 読み取り、書き込み、実行の権限を2進数で表現 \*/  
#define READ\_PERMISSION 0b100 /\* 読み取り可能 \*/  
#define WRITE\_PERMISSION 0b010 /\* 書き込み可能 \*/  
#define EXEC\_PERMISSION 0b001 /\* 実行可能 \*/  
  
/\* 権限の組み合わせ \*/  
int user\_permissions = READ\_PERMISSION | WRITE\_PERMISSION; /\* 読み書き可能 \*/

### その他の新機能

C23には他にも多くの新機能があります：

#### 5. auto型推論（制限付き）

変数の型を自動的に推論する機能（ただし制限あり）。

auto x = 42; /\* xはint型と推論される \*/  
auto y = 3.14; /\* yはdouble型と推論される \*/

#### 6. constexpr - コンパイル時定数

コンパイル時に値が確定する定数を定義できます。

constexpr int ARRAY\_SIZE = 100;  
int array[ARRAY\_SIZE]; /\* コンパイル時にサイズが決まる \*/

#### 7. 新しいプリプロセッサ機能

条件付きコンパイルがより便利に：

/\* マクロが定義されているかチェック \*/  
#elifdef DEBUG  
 printf("デバッグモード\n");  
#elifndef RELEASE  
 printf("リリースモードではない\n");  
#endif

### 初心者が陥りやすい間違い

#### 1. コンパイラのサポート確認を忘れる

/\* NG: コンパイラがC23をサポートしていない場合 \*/  
bool flag = true; /\* エラー: 'bool' undeclared \*/  
  
/\* OK: サポート確認とフォールバック \*/  
#if \_\_STDC\_VERSION\_\_ >= 202300L  
 bool flag = true; /\* C23 \*/  
#else  
 #include <stdbool.h>  
 bool flag = true; /\* C99以前 \*/  
#endif

#### 2. 古い書き方と新しい書き方の混在

/\* NG: 混在は避ける \*/  
int \*p1 = NULL; /\* 古い書き方 \*/  
int \*p2 = nullptr; /\* 新しい書き方 \*/  
  
/\* OK: 統一する \*/  
int \*p1 = nullptr;  
int \*p2 = nullptr;

#### 3. 2進数リテラルの桁数ミス

/\* NG: ビット数を間違えやすい \*/  
unsigned char byte = 0b111111111; /\* 9ビット！オーバーフロー \*/  
  
/\* OK: 8ビットに収める \*/  
unsigned char byte = 0b11111111; /\* 8ビット、OK \*/  
  
/\* より良い: アンダースコアで区切る（C23） \*/  
unsigned char byte = 0b1111\_1111; /\* 見やすい！ \*/

### 学習のコツ

1. **段階的に学ぶ**
   * まずはbool型から始める
   * 慣れたらtypeof演算子を試す
   * 最後に高度な機能へ
2. **コンパイラの確認**
   * 使用するコンパイラのバージョンを確認
   * C23サポート状況をチェック
   * 必要に応じて新しいバージョンをインストール
3. **実際に試す**
   * 小さなプログラムで新機能を試す
   * エラーメッセージをよく読む
   * 動作を確認しながら理解を深める

### まとめ

C23の新機能は、C言語をより使いやすく、安全にするためのものです。すべてを一度に覚える必要はありません。まずは基本的な機能（bool型、2進数リテラル）から始めて、徐々に高度な機能を学んでいきましょう。

新しい機能を使うことで、より読みやすく、バグの少ないプログラムが書けるようになります！

## サンプルコード

### bool\_basics.c

/\*  
 \* bool\_basics.c  
 \* C23のbool型の基本的な使い方  
 \*   
 \* コンパイル: gcc -std=c23 bool\_basics.c -o bool\_basics  
 \*/  
  
#include <stdio.h>  
  
/\* C23ではbool, true, falseが組み込み型として使える \*/  
  
/\* 学生情報を管理する構造体 \*/  
typedef struct {  
 char name[50];  
 int age;  
 bool is\_enrolled; /\* 在籍中かどうか \*/  
 bool has\_scholarship; /\* 奨学金を受けているか \*/  
 bool is\_full\_time; /\* フルタイムの学生か \*/  
} Student;  
  
/\* 成績が合格かどうかを判定する関数 \*/  
bool is\_passing\_grade(int score) {  
 return score >= 60; /\* 60点以上なら合格（true） \*/  
}  
  
/\* 学生が卒業要件を満たしているかチェック \*/  
bool can\_graduate(const Student \*student, int total\_credits) {  
 /\* 在籍中で、120単位以上取得していれば卒業可能 \*/  
 return student->is\_enrolled && total\_credits >= 120;  
}  
  
/\* bool値を文字列に変換 \*/  
const char\* bool\_to\_string(bool value) {  
 return value ? "はい" : "いいえ";  
}  
  
/\* 複数の条件を組み合わせた判定 \*/  
bool is\_eligible\_for\_honor(const Student \*student, double gpa, int credits) {  
 /\* 優等生の条件：  
 \* - フルタイムの学生  
 \* - GPA 3.5以上  
 \* - 90単位以上取得  
 \*/  
 return student->is\_full\_time && gpa >= 3.5 && credits >= 90;  
}  
  
int main(void) {  
 printf("=== C23 bool型の基本 ===\n\n");  
   
 /\* 基本的な使い方 \*/  
 bool is\_ready = true;  
 bool is\_finished = false;  
   
 printf("準備完了: %s\n", bool\_to\_string(is\_ready));  
 printf("作業終了: %s\n", bool\_to\_string(is\_finished));  
   
 /\* 比較演算の結果はbool型 \*/  
 int x = 10, y = 20;  
 bool is\_greater = (x > y);  
 printf("\n%d > %d: %s\n", x, y, bool\_to\_string(is\_greater));  
   
 /\* 関数の使用例 \*/  
 printf("\n--- 成績判定 ---\n");  
 int scores[] = {85, 55, 92, 45, 70};  
 int i;  
   
 for (i = 0; i < 5; i++) {  
 bool passed = is\_passing\_grade(scores[i]);  
 printf("点数 %d: %s\n", scores[i],   
 passed ? "合格" : "不合格");  
 }  
   
 /\* 構造体での使用例 \*/  
 Student alice = {  
 .name = "Alice",  
 .age = 21,  
 .is\_enrolled = true,  
 .has\_scholarship = true,  
 .is\_full\_time = true  
 };  
   
 printf("\n--- 学生情報 ---\n");  
 printf("名前: %s\n", alice.name);  
 printf("年齢: %d\n", alice.age);  
 printf("在籍中: %s\n", bool\_to\_string(alice.is\_enrolled));  
 printf("奨学金: %s\n", bool\_to\_string(alice.has\_scholarship));  
 printf("フルタイム: %s\n", bool\_to\_string(alice.is\_full\_time));  
   
 /\* 卒業判定 \*/  
 int alice\_credits = 125;  
 bool can\_grad = can\_graduate(&alice, alice\_credits);  
 printf("\n卒業可能: %s (取得単位: %d)\n",   
 bool\_to\_string(can\_grad), alice\_credits);  
   
 /\* 優等生判定 \*/  
 double alice\_gpa = 3.8;  
 bool is\_honor = is\_eligible\_for\_honor(&alice, alice\_gpa, alice\_credits);  
 printf("優等生資格: %s (GPA: %.2f)\n",   
 bool\_to\_string(is\_honor), alice\_gpa);  
   
 /\* 論理演算 \*/  
 printf("\n--- 論理演算 ---\n");  
 bool a = true, b = false;  
   
 printf("true && false = %s\n", bool\_to\_string(a && b));  
 printf("true || false = %s\n", bool\_to\_string(a || b));  
 printf("!true = %s\n", bool\_to\_string(!a));  
 printf("!false = %s\n", bool\_to\_string(!b));  
   
 /\* bool型の注意点 \*/  
 printf("\n--- bool型の性質 ---\n");  
   
 /\* bool型は0がfalse、それ以外がtrue \*/  
 bool from\_zero = 0;  
 bool from\_one = 1;  
 bool from\_other = 42;  
   
 printf("bool(0) = %s\n", bool\_to\_string(from\_zero));  
 printf("bool(1) = %s\n", bool\_to\_string(from\_one));  
 printf("bool(42) = %s\n", bool\_to\_string(from\_other));  
   
 /\* サイズの確認 \*/  
 printf("\nbool型のサイズ: %zu バイト\n", sizeof(bool));  
   
 return 0;  
}

### typeof\_demo.c

/\*  
 \* typeof\_demo.c  
 \* C23のtypeof演算子の使い方  
 \*   
 \* コンパイル: gcc -std=c23 typeof\_demo.c -o typeof\_demo  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
  
/\* 型安全なマクロの定義 \*/  
  
/\* 2つの値を交換するマクロ（型安全版） \*/  
#define SWAP(a, b) do { \  
 typeof(a) temp = (a); \  
 (a) = (b); \  
 (b) = temp; \  
} while(0)  
  
/\* 最大値を求めるマクロ（型安全版） \*/  
#define MAX(a, b) ({ \  
 typeof(a) \_a = (a); \  
 typeof(b) \_b = (b); \  
 \_a > \_b ? \_a : \_b; \  
})  
  
/\* 最小値を求めるマクロ（型安全版） \*/  
#define MIN(a, b) ({ \  
 typeof(a) \_a = (a); \  
 typeof(b) \_b = (b); \  
 \_a < \_b ? \_a : \_b; \  
})  
  
/\* 配列の要素数を求めるマクロ \*/  
#define ARRAY\_SIZE(arr) (sizeof(arr) / sizeof(typeof(arr[0])))  
  
/\* 安全な配列コピー \*/  
#define ARRAY\_COPY(dest, src, count) do { \  
 for (size\_t i = 0; i < (count); i++) { \  
 (dest)[i] = (src)[i]; \  
 } \  
} while(0)  
  
/\* ジェネリックな構造体の例 \*/  
typedef struct {  
 void \*data;  
 size\_t size;  
 size\_t capacity;  
} GenericBuffer;  
  
/\* ジェネリックなバッファ作成マクロ \*/  
#define CREATE\_BUFFER(type, cap) ({ \  
 GenericBuffer buf; \  
 buf.data = malloc(sizeof(type) \* (cap)); \  
 buf.size = 0; \  
 buf.capacity = (cap); \  
 buf; \  
})  
  
/\* デモ用の構造体 \*/  
typedef struct {  
 int x;  
 int y;  
} Point;  
  
typedef struct {  
 double x;  
 double y;  
 double z;  
} Point3D;  
  
/\* 関数テンプレート風のマクロ \*/  
#define PRINT\_POINT(p) do { \  
 if (sizeof(p) == sizeof(Point)) { \  
 printf("Point(%d, %d)", ((Point\*)&(p))->x, ((Point\*)&(p))->y); \  
 } else if (sizeof(p) == sizeof(Point3D)) { \  
 printf("Point3D(%.2f, %.2f, %.2f)", \  
 ((Point3D\*)&(p))->x, ((Point3D\*)&(p))->y, ((Point3D\*)&(p))->z); \  
 } \  
} while(0)  
  
/\* 型に応じた初期化マクロ \*/  
#define ZERO\_INIT(var) do { \  
 typeof(var) zero = {0}; \  
 (var) = zero; \  
} while(0)  
  
int main(void) {  
 printf("=== C23 typeof演算子のデモ ===\n\n");  
   
 /\* 基本的な使用例 \*/  
 printf("--- 基本的な使用例 ---\n");  
   
 int original = 42;  
 typeof(original) copy = original; /\* copyはint型 \*/  
 printf("元の値: %d, コピー: %d\n", original, copy);  
   
 double pi = 3.14159;  
 typeof(pi) tau = pi \* 2; /\* tauはdouble型 \*/  
 printf("π = %.5f, τ = %.5f\n", pi, tau);  
   
 /\* 配列の型コピー \*/  
 int arr1[] = {1, 2, 3, 4, 5};  
 typeof(arr1[0]) arr2[5]; /\* arr2はint[5]型 \*/  
 ARRAY\_COPY(arr2, arr1, 5);  
   
 printf("\n配列のコピー: ");  
 int i;  
 for (i = 0; i < 5; i++) {  
 printf("%d ", arr2[i]);  
 }  
 printf("\n");  
   
 /\* SWAP マクロのデモ \*/  
 printf("\n--- SWAP マクロ ---\n");  
   
 int a = 10, b = 20;  
 printf("交換前: a=%d, b=%d\n", a, b);  
 SWAP(a, b);  
 printf("交換後: a=%d, b=%d\n", a, b);  
   
 double x = 1.5, y = 2.5;  
 printf("\n交換前: x=%.1f, y=%.1f\n", x, y);  
 SWAP(x, y);  
 printf("交換後: x=%.1f, y=%.1f\n", x, y);  
   
 /\* 構造体でも動作 \*/  
 Point p1 = {10, 20}, p2 = {30, 40};  
 printf("\n交換前: p1=(%d,%d), p2=(%d,%d)\n",   
 p1.x, p1.y, p2.x, p2.y);  
 SWAP(p1, p2);  
 printf("交換後: p1=(%d,%d), p2=(%d,%d)\n",   
 p1.x, p1.y, p2.x, p2.y);  
   
 /\* MAX/MIN マクロのデモ \*/  
 printf("\n--- MAX/MIN マクロ ---\n");  
   
 int int\_max = MAX(15, 27);  
 double double\_max = MAX(3.14, 2.71);  
 printf("MAX(15, 27) = %d\n", int\_max);  
 printf("MAX(3.14, 2.71) = %.2f\n", double\_max);  
   
 int int\_min = MIN(15, 27);  
 double double\_min = MIN(3.14, 2.71);  
 printf("MIN(15, 27) = %d\n", int\_min);  
 printf("MIN(3.14, 2.71) = %.2f\n", double\_min);  
   
 /\* 複雑な型での使用 \*/  
 printf("\n--- 複雑な型での使用 ---\n");  
   
 /\* 関数ポインタ \*/  
 int (\*func\_ptr)(const char\*) = puts;  
 typeof(func\_ptr) another\_func = printf; /\* 同じ型の関数ポインタ \*/  
   
 /\* ポインタの型 \*/  
 int \*int\_ptr = &a;  
 typeof(\*int\_ptr) value = \*int\_ptr; /\* valueはint型 \*/  
 typeof(int\_ptr) another\_ptr = &b; /\* another\_ptrはint\*型 \*/  
   
 printf("ポインタが指す値: %d\n", value);  
 printf("別のポインタが指す値: %d\n", \*another\_ptr);  
   
 /\* 配列サイズの取得 \*/  
 printf("\n--- 配列サイズ ---\n");  
   
 char str[] = "Hello, C23!";  
 int numbers[] = {10, 20, 30, 40, 50};  
 double values[] = {1.1, 2.2, 3.3};  
   
 printf("文字配列のサイズ: %zu\n", ARRAY\_SIZE(str));  
 printf("整数配列のサイズ: %zu\n", ARRAY\_SIZE(numbers));  
 printf("実数配列のサイズ: %zu\n", ARRAY\_SIZE(values));  
   
 /\* 型に応じた初期化 \*/  
 printf("\n--- 型に応じた初期化 ---\n");  
   
 Point3D point3d = {1.0, 2.0, 3.0};  
 printf("初期化前: (%.1f, %.1f, %.1f)\n",   
 point3d.x, point3d.y, point3d.z);  
   
 ZERO\_INIT(point3d);  
 printf("ゼロ初期化後: (%.1f, %.1f, %.1f)\n",   
 point3d.x, point3d.y, point3d.z);  
   
 /\* 実用的な例：動的配列 \*/  
 printf("\n--- 実用例：動的配列 ---\n");  
   
 /\* 元の配列と同じ型の動的配列を作成 \*/  
 int static\_array[] = {1, 2, 3, 4, 5};  
 size\_t array\_size = ARRAY\_SIZE(static\_array);  
   
 typeof(static\_array[0]) \*dynamic\_array =   
 malloc(sizeof(typeof(static\_array[0])) \* array\_size);  
   
 if (dynamic\_array) {  
 /\* コピー \*/  
 for (i = 0; i < array\_size; i++) {  
 dynamic\_array[i] = static\_array[i] \* 2;  
 }  
   
 printf("動的配列の内容: ");  
 for (i = 0; i < array\_size; i++) {  
 printf("%d ", dynamic\_array[i]);  
 }  
 printf("\n");  
   
 free(dynamic\_array);  
 }  
   
 return 0;  
}

### nullptr\_safety.c

/\*  
 \* nullptr\_safety.c  
 \* C23のnullptrによる型安全性の向上  
 \*   
 \* コンパイル: gcc -std=c23 nullptr\_safety.c -o nullptr\_safety  
 \*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
/\* リンクリストのノード \*/  
typedef struct Node {  
 int data;  
 struct Node \*next;  
} Node;  
  
/\* 安全なポインタチェックマクロ \*/  
#define SAFE\_FREE(ptr) do { \  
 if ((ptr) != nullptr) { \  
 free(ptr); \  
 (ptr) = nullptr; \  
 } \  
} while(0)  
  
/\* nullptrチェック付き関数呼び出し \*/  
#define SAFE\_CALL(func\_ptr, ...) \  
 ((func\_ptr) != nullptr ? (func\_ptr)(\_\_VA\_ARGS\_\_) : 0)  
  
/\* デバッグ用のポインタ情報表示 \*/  
void print\_pointer\_info(const char \*name, void \*ptr) {  
 if (ptr == nullptr) {  
 printf("%s: nullptr (安全な空ポインタ)\n", name);  
 } else {  
 printf("%s: %p (有効なアドレス)\n", name, ptr);  
 }  
}  
  
/\* リンクリストの作成 \*/  
Node\* create\_node(int data) {  
 Node \*node = malloc(sizeof(Node));  
 if (node != nullptr) {  
 node->data = data;  
 node->next = nullptr; /\* 明示的にnullptrで初期化 \*/  
 }  
 return node;  
}  
  
/\* リンクリストに要素を追加 \*/  
void append\_node(Node \*\*head, int data) {  
 if (head == nullptr) return;  
   
 Node \*new\_node = create\_node(data);  
 if (new\_node == nullptr) {  
 printf("メモリ割り当てエラー\n");  
 return;  
 }  
   
 if (\*head == nullptr) {  
 \*head = new\_node;  
 } else {  
 Node \*current = \*head;  
 while (current->next != nullptr) {  
 current = current->next;  
 }  
 current->next = new\_node;  
 }  
}  
  
/\* リンクリストの表示 \*/  
void print\_list(const Node \*head) {  
 const Node \*current = head;  
   
 printf("リスト: ");  
 if (current == nullptr) {  
 printf("(空)\n");  
 return;  
 }  
   
 while (current != nullptr) {  
 printf("%d", current->data);  
 if (current->next != nullptr) {  
 printf(" -> ");  
 }  
 current = current->next;  
 }  
 printf("\n");  
}  
  
/\* リンクリストの解放 \*/  
void free\_list(Node \*\*head) {  
 if (head == nullptr || \*head == nullptr) return;  
   
 Node \*current = \*head;  
 while (current != nullptr) {  
 Node \*next = current->next;  
 free(current);  
 current = next;  
 }  
   
 \*head = nullptr; /\* 解放後はnullptrに設定 \*/  
}  
  
/\* 文字列処理の例 \*/  
char\* safe\_strdup(const char \*src) {  
 if (src == nullptr) {  
 return nullptr;  
 }  
   
 size\_t len = strlen(src) + 1;  
 char \*dest = malloc(len);  
   
 if (dest != nullptr) {  
 strcpy(dest, src);  
 }  
   
 return dest;  
}  
  
/\* 関数ポインタの例 \*/  
typedef int (\*operation\_func)(int, int);  
  
int add(int a, int b) { return a + b; }  
int subtract(int a, int b) { return a - b; }  
int multiply(int a, int b) { return a \* b; }  
  
/\* 安全な関数ポインタ配列 \*/  
typedef struct {  
 const char \*name;  
 operation\_func func;  
} Operation;  
  
/\* 演算を実行 \*/  
int perform\_operation(operation\_func op, int a, int b) {  
 if (op == nullptr) {  
 printf("エラー: 無効な演算関数\n");  
 return 0;  
 }  
 return op(a, b);  
}  
  
/\* nullptr の実用例 \*/  
void demonstrate\_nullptr\_usage(void) {  
 printf("\n--- nullptr の実用例 ---\n");  
   
 /\* 1. 初期化での使用 \*/  
 int \*int\_ptr = nullptr;  
 double \*double\_ptr = nullptr;  
 Node \*list\_head = nullptr;  
   
 print\_pointer\_info("int\_ptr", int\_ptr);  
 print\_pointer\_info("double\_ptr", double\_ptr);  
 print\_pointer\_info("list\_head", list\_head);  
   
 /\* 2. 条件チェック \*/  
 if (int\_ptr == nullptr) {  
 printf("\nint\_ptr は nullptr です（期待通り）\n");  
 }  
   
 /\* 3. 動的メモリ管理 \*/  
 int \*dynamic\_array = malloc(sizeof(int) \* 10);  
 if (dynamic\_array != nullptr) {  
 printf("\n動的配列の割り当て成功\n");  
   
 /\* 使用 \*/  
 int i;  
 for (i = 0; i < 10; i++) {  
 dynamic\_array[i] = i \* i;  
 }  
   
 /\* 安全な解放 \*/  
 SAFE\_FREE(dynamic\_array);  
 print\_pointer\_info("解放後のdynamic\_array", dynamic\_array);  
 }  
   
 /\* 4. 二重解放の防止 \*/  
 SAFE\_FREE(dynamic\_array); /\* 既にnullptrなので何も起きない \*/  
 printf("二重解放の試行: 安全に処理されました\n");  
}  
  
/\* エラーハンドリングの例 \*/  
typedef struct {  
 int code;  
 char \*message;  
} ErrorInfo;  
  
ErrorInfo\* create\_error(int code, const char \*msg) {  
 if (msg == nullptr) {  
 return nullptr; /\* 無効な入力 \*/  
 }  
   
 ErrorInfo \*error = malloc(sizeof(ErrorInfo));  
 if (error == nullptr) {  
 return nullptr; /\* メモリ不足 \*/  
 }  
   
 error->code = code;  
 error->message = safe\_strdup(msg);  
   
 if (error->message == nullptr) {  
 free(error);  
 return nullptr; /\* 文字列コピー失敗 \*/  
 }  
   
 return error;  
}  
  
void free\_error(ErrorInfo \*\*error) {  
 if (error == nullptr || \*error == nullptr) return;  
   
 SAFE\_FREE((\*error)->message);  
 SAFE\_FREE(\*error);  
}  
  
int main(void) {  
 printf("=== C23 nullptr の安全性デモ ===\n");  
   
 /\* 基本的な使用法 \*/  
 demonstrate\_nullptr\_usage();  
   
 /\* リンクリストの例 \*/  
 printf("\n--- リンクリストの例 ---\n");  
   
 Node \*list = nullptr;  
 print\_list(list);  
   
 /\* 要素の追加 \*/  
 append\_node(&list, 10);  
 append\_node(&list, 20);  
 append\_node(&list, 30);  
 print\_list(list);  
   
 /\* nullptrチェック \*/  
 Node \*search = list;  
 int target = 20;  
 while (search != nullptr && search->data != target) {  
 search = search->next;  
 }  
   
 if (search != nullptr) {  
 printf("値 %d が見つかりました\n", target);  
 } else {  
 printf("値 %d は見つかりませんでした\n", target);  
 }  
   
 /\* リストの解放 \*/  
 free\_list(&list);  
 print\_list(list); /\* 空のリストと表示される \*/  
   
 /\* 関数ポインタの例 \*/  
 printf("\n--- 関数ポインタの例 ---\n");  
   
 Operation operations[] = {  
 {"加算", add},  
 {"減算", subtract},  
 {"乗算", multiply},  
 {"除算", nullptr} /\* 未実装 \*/  
 };  
   
 int a = 10, b = 5;  
 int i;  
   
 for (i = 0; i < 4; i++) {  
 printf("%s: ", operations[i].name);  
   
 if (operations[i].func != nullptr) {  
 int result = perform\_operation(operations[i].func, a, b);  
 printf("%d\n", result);  
 } else {  
 printf("未実装\n");  
 }  
 }  
   
 /\* エラーハンドリング \*/  
 printf("\n--- エラーハンドリング ---\n");  
   
 ErrorInfo \*error1 = create\_error(404, "ファイルが見つかりません");  
 ErrorInfo \*error2 = create\_error(500, nullptr); /\* 無効な入力 \*/  
   
 if (error1 != nullptr) {  
 printf("エラー %d: %s\n", error1->code, error1->message);  
 free\_error(&error1);  
 }  
   
 if (error2 == nullptr) {  
 printf("エラーオブジェクトの作成に失敗（期待通り）\n");  
 }  
   
 /\* 安全な関数呼び出し \*/  
 printf("\n--- 安全な関数呼び出し ---\n");  
   
 operation\_func safe\_op = add;  
 operation\_func unsafe\_op = nullptr;  
   
 int result1 = SAFE\_CALL(safe\_op, 15, 25);  
 int result2 = SAFE\_CALL(unsafe\_op, 15, 25);  
   
 printf("安全な呼び出し結果: %d\n", result1);  
 printf("nullptrの呼び出し結果: %d (デフォルト値)\n", result2);  
   
 printf("\n=== デモ完了 ===\n");  
   
 return 0;  
}

### binary\_literals.c

/\*  
 \* binary\_literals.c  
 \* C23の2進数リテラルとビット操作  
 \*   
 \* コンパイル: gcc -std=c23 binary\_literals.c -o binary\_literals  
 \*/  
  
#include <stdio.h>  
#include <stdint.h>  
  
/\* ビットフラグの定義（2進数リテラル使用） \*/  
#define FLAG\_READ 0b00000001 /\* 読み取り権限 \*/  
#define FLAG\_WRITE 0b00000010 /\* 書き込み権限 \*/  
#define FLAG\_EXECUTE 0b00000100 /\* 実行権限 \*/  
#define FLAG\_DELETE 0b00001000 /\* 削除権限 \*/  
#define FLAG\_ADMIN 0b10000000 /\* 管理者権限 \*/  
  
/\* ビットマスクの定義 \*/  
#define LOWER\_NIBBLE 0b00001111 /\* 下位4ビット \*/  
#define UPPER\_NIBBLE 0b11110000 /\* 上位4ビット \*/  
  
/\* RGBカラーの操作 \*/  
#define RGB\_RED\_MASK 0b11111111\_00000000\_00000000 /\* 赤成分 \*/  
#define RGB\_GREEN\_MASK 0b00000000\_11111111\_00000000 /\* 緑成分 \*/  
#define RGB\_BLUE\_MASK 0b00000000\_00000000\_11111111 /\* 青成分 \*/  
  
/\* ビットパターンを表示する関数 \*/  
void print\_binary(unsigned int value, int bits) {  
 int i;  
 printf("0b");  
 for (i = bits - 1; i >= 0; i--) {  
 printf("%d", (value >> i) & 1);  
 if (i % 4 == 0 && i > 0) {  
 printf("\_"); /\* 4ビットごとに区切り \*/  
 }  
 }  
}  
  
/\* 権限フラグを文字列で表示 \*/  
void print\_permissions(unsigned char perms) {  
 printf("権限: ");  
 printf("%c", (perms & FLAG\_READ) ? 'R' : '-');  
 printf("%c", (perms & FLAG\_WRITE) ? 'W' : '-');  
 printf("%c", (perms & FLAG\_EXECUTE) ? 'X' : '-');  
 printf("%c", (perms & FLAG\_DELETE) ? 'D' : '-');  
 printf(" ");  
 printf("%s", (perms & FLAG\_ADMIN) ? "[ADMIN]" : "[USER]");  
 printf(" (");  
 print\_binary(perms, 8);  
 printf(")\n");  
}  
  
/\* ビット操作のデモ \*/  
void bit\_manipulation\_demo(void) {  
 printf("--- ビット操作デモ ---\n");  
   
 unsigned char value = 0b10101010;  
   
 printf("初期値: ");  
 print\_binary(value, 8);  
 printf(" = %d\n", value);  
   
 /\* ビットのセット \*/  
 value |= 0b00000100; /\* 3番目のビットをセット \*/  
 printf("ビット2をセット: ");  
 print\_binary(value, 8);  
 printf(" = %d\n", value);  
   
 /\* ビットのクリア \*/  
 value &= ~0b00001000; /\* 4番目のビットをクリア \*/  
 printf("ビット3をクリア: ");  
 print\_binary(value, 8);  
 printf(" = %d\n", value);  
   
 /\* ビットの反転 \*/  
 value ^= 0b11110000; /\* 上位4ビットを反転 \*/  
 printf("上位4ビット反転: ");  
 print\_binary(value, 8);  
 printf(" = %d\n", value);  
   
 /\* 特定ビットの抽出 \*/  
 unsigned char lower = value & LOWER\_NIBBLE;  
 unsigned char upper = (value & UPPER\_NIBBLE) >> 4;  
   
 printf("\n下位4ビット: ");  
 print\_binary(lower, 4);  
 printf(" = %d\n", lower);  
   
 printf("上位4ビット: ");  
 print\_binary(upper, 4);  
 printf(" = %d\n", upper);  
}  
  
/\* パターンマッチングの例 \*/  
void pattern\_matching\_demo(void) {  
 printf("\n--- パターンマッチング ---\n");  
   
 unsigned char data[] = {  
 0b11001100,  
 0b10101010,  
 0b11110000,  
 0b00001111,  
 0b01010101  
 };  
   
 unsigned char pattern = 0b11110000;  
 unsigned char mask = 0b11110000; /\* 上位4ビットのみチェック \*/  
   
 printf("検索パターン: ");  
 print\_binary(pattern, 8);  
 printf("\n");  
   
 printf("マスク: ");  
 print\_binary(mask, 8);  
 printf("\n\n");  
   
 int i;  
 for (i = 0; i < 5; i++) {  
 printf("データ[%d]: ", i);  
 print\_binary(data[i], 8);  
   
 if ((data[i] & mask) == pattern) {  
 printf(" <- マッチ！");  
 }  
 printf("\n");  
 }  
}  
  
/\* 実践例：簡易画像処理 \*/  
typedef struct {  
 unsigned char red;  
 unsigned char green;  
 unsigned char blue;  
} Color;  
  
/\* カラー値から特定成分を抽出 \*/  
void color\_components\_demo(void) {  
 printf("\n--- カラー成分の操作 ---\n");  
   
 /\* 24ビットカラー (RGB888) \*/  
 uint32\_t color = 0b11111111\_01010101\_00110011; /\* R=255, G=85, B=51 \*/  
   
 printf("元のカラー: ");  
 print\_binary(color, 24);  
 printf(" = 0x%06X\n", color);  
   
 /\* 各成分の抽出 \*/  
 unsigned char red = (color & 0b11111111\_00000000\_00000000) >> 16;  
 unsigned char green = (color & 0b00000000\_11111111\_00000000) >> 8;  
 unsigned char blue = color & 0b00000000\_00000000\_11111111;  
   
 printf("赤成分: %d (", red);  
 print\_binary(red, 8);  
 printf(")\n");  
   
 printf("緑成分: %d (", green);  
 print\_binary(green, 8);  
 printf(")\n");  
   
 printf("青成分: %d (", blue);  
 print\_binary(blue, 8);  
 printf(")\n");  
   
 /\* カラーの変更 \*/  
 color &= ~0b00000000\_11111111\_00000000; /\* 緑成分をクリア \*/  
 color |= 0b00000000\_11111111\_00000000; /\* 緑を最大に \*/  
   
 printf("\n緑を最大にした後: ");  
 print\_binary(color, 24);  
 printf(" = 0x%06X\n", color);  
}  
  
/\* ビットフィールドの実践例 \*/  
typedef struct {  
 unsigned int enabled : 1; /\* 1ビット: 有効/無効 \*/  
 unsigned int type : 3; /\* 3ビット: タイプ (0-7) \*/  
 unsigned int priority : 4; /\* 4ビット: 優先度 (0-15) \*/  
 unsigned int reserved : 8; /\* 8ビット: 予約済み \*/  
} DeviceConfig;  
  
/\* デバイス設定のデモ \*/  
void device\_config\_demo(void) {  
 printf("\n--- デバイス設定 ---\n");  
   
 /\* ビットパターンで初期化 \*/  
 unsigned short config\_value = 0b00000000\_01011001;  
 /\* ^^^^^^^^ ^^^^ ^^^ ^  
 reserved prio type enabled \*/  
   
 DeviceConfig \*config = (DeviceConfig\*)&config\_value;  
   
 printf("設定値: ");  
 print\_binary(config\_value, 16);  
 printf("\n");  
   
 printf("解析結果:\n");  
 printf(" 有効: %s\n", config->enabled ? "はい" : "いいえ");  
 printf(" タイプ: %d\n", config->type);  
 printf(" 優先度: %d\n", config->priority);  
 printf(" 予約: %d\n", config->reserved);  
   
 /\* 設定の変更 \*/  
 config->enabled = 1;  
 config->type = 0b111; /\* 最大値 \*/  
 config->priority = 0b1010; /\* 10 \*/  
   
 printf("\n変更後: ");  
 print\_binary(config\_value, 16);  
 printf("\n");  
}  
  
/\* 実用的な例：チェックサム計算 \*/  
unsigned char calculate\_checksum(const unsigned char \*data, size\_t len) {  
 unsigned char checksum = 0b00000000;  
 size\_t i;  
   
 for (i = 0; i < len; i++) {  
 checksum ^= data[i]; /\* XORでチェックサム計算 \*/  
 }  
   
 return checksum;  
}  
  
/\* ネットワークパケットのシミュレーション \*/  
void network\_packet\_demo(void) {  
 printf("\n--- ネットワークパケット ---\n");  
   
 /\* パケットヘッダー（簡略化） \*/  
 struct {  
 unsigned char version : 4;  
 unsigned char ihl : 4;  
 unsigned char tos;  
 unsigned short total\_length;  
 } ip\_header = {  
 .version = 0b0100, /\* IPv4 \*/  
 .ihl = 0b0101, /\* 20バイト \*/  
 .tos = 0b00000000, /\* 通常サービス \*/  
 .total\_length = 64  
 };  
   
 printf("IPヘッダー:\n");  
 printf(" バージョン: ");  
 print\_binary(ip\_header.version, 4);  
 printf(" (IPv%d)\n", ip\_header.version);  
   
 printf(" ヘッダー長: ");  
 print\_binary(ip\_header.ihl, 4);  
 printf(" (%dバイト)\n", ip\_header.ihl \* 4);  
   
 /\* データ部分 \*/  
 unsigned char data[] = {  
 0b11001100,  
 0b10101010,  
 0b01010101,  
 0b11110000  
 };  
   
 unsigned char checksum = calculate\_checksum(data, 4);  
   
 printf("\nデータ:\n");  
 int i;  
 for (i = 0; i < 4; i++) {  
 printf(" [%d]: ", i);  
 print\_binary(data[i], 8);  
 printf("\n");  
 }  
   
 printf("\nチェックサム: ");  
 print\_binary(checksum, 8);  
 printf(" = %d\n", checksum);  
}  
  
int main(void) {  
 printf("=== C23 2進数リテラルのデモ ===\n\n");  
   
 /\* 基本的な2進数リテラル \*/  
 printf("--- 基本的な2進数リテラル ---\n");  
   
 int binary1 = 0b1010; /\* 10 \*/  
 int binary2 = 0B1111; /\* 15 (大文字Bも可) \*/  
 int binary3 = 0b11111111; /\* 255 \*/  
   
 printf("0b1010 = %d\n", binary1);  
 printf("0B1111 = %d\n", binary2);  
 printf("0b11111111 = %d\n", binary3);  
   
 /\* アンダースコア区切り（C23） \*/  
 printf("\n--- アンダースコアで見やすく ---\n");  
   
 int readable1 = 0b1111\_1111; /\* 255 \*/  
 int readable2 = 0b1010\_1010\_1010\_1010; /\* 43690 \*/  
   
 printf("0b1111\_1111 = %d\n", readable1);  
 printf("0b1010\_1010\_1010\_1010 = %d\n", readable2);  
   
 /\* 権限管理の例 \*/  
 printf("\n--- 権限管理システム ---\n");  
   
 unsigned char user\_perms = 0b00000000;  
   
 print\_permissions(user\_perms);  
   
 /\* 権限の付与 \*/  
 user\_perms |= FLAG\_READ | FLAG\_WRITE;  
 printf("\n読み書き権限を付与:\n");  
 print\_permissions(user\_perms);  
   
 /\* 管理者権限の付与 \*/  
 user\_perms |= FLAG\_ADMIN | FLAG\_DELETE | FLAG\_EXECUTE;  
 printf("\n管理者権限を付与:\n");  
 print\_permissions(user\_perms);  
   
 /\* 権限のチェック \*/  
 if (user\_perms & FLAG\_ADMIN) {  
 printf("\n管理者権限があります\n");  
 }  
   
 /\* 各種デモの実行 \*/  
 printf("\n");  
 bit\_manipulation\_demo();  
 pattern\_matching\_demo();  
 color\_components\_demo();  
 device\_config\_demo();  
 network\_packet\_demo();  
   
 /\* まとめ \*/  
 printf("\n--- 2進数リテラルの利点 ---\n");  
 printf("1. ビットパターンが直感的に理解できる\n");  
 printf("2. ビット演算のミスが減る\n");  
 printf("3. ハードウェア仕様書との対応が簡単\n");  
 printf("4. デバッグが容易\n");  
   
 return 0;  
}

## 演習課題

### 演習14-1: bool型の活用

<exercises/README.md>を参照

### 演習14-2: ビット操作と2進数リテラル

<exercises/README.md>を参照

### 演習14-3: typeof演算子の応用

<exercises/README.md>を参照

## コンパイル方法

この章はC23専用です。以下のコマンドでコンパイルしてください：

# 個別ファイルのコンパイル  
gcc -std=c23 -Wall -Wextra -pedantic source.c -o output  
  
# Makefileを使用  
make all # すべてコンパイル  
make test # C23サポートテスト  
make run-all # すべて実行

### コンパイラサポート状況

C23は新しい規格のため、コンパイラサポートは発展途上です：

* **GCC**: 13以降で部分サポート（-std=c23）
* **Clang**: 16以降で部分サポート（-std=c23）
* **MSVC**: 未対応（2024年現在）

## 注意事項

1. **コンパイラ依存**: すべてのC23機能がサポートされているとは限りません
2. **移植性**: 古いコンパイラでは動作しません
3. **学習順序**: C90/C99の基礎を理解してから学習することを推奨

## 参考資料

* [C23規格ドラフト](https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf)
* [C23新機能の概要](https://en.cppreference.com/w/c/23)
* コンパイラのドキュメント（GCC、Clang）

## 🔄 C90/C99/C11からの移行

### bool型

/\* C90 \*/  
#define TRUE 1  
#define FALSE 0  
int flag = TRUE;  
  
/\* C99 \*/  
#include <stdbool.h>  
bool flag = true;  
  
/\* C23 \*/  
bool flag = true; /\* ヘッダー不要 \*/

### NULLポインタ

/\* C90/C99/C11 \*/  
int \*p = NULL;  
  
/\* C23 \*/  
int \*p = nullptr;

### 型の取得

/\* C11: \_Generic \*/  
#define TYPE\_NAME(x) \_Generic((x), \  
 int: "int", \  
 double: "double", \  
 default: "other")  
  
/\* C23: typeof \*/  
typeof(x) y; /\* xと同じ型 \*/

**注**: この章はオプション的な内容です。実務では、使用するコンパイラのC23サポート状況を確認してから活用してください。