C言語プログラミング教材

メインテキスト

C言語学習教材プロジェクト

2025年06月30日

Table of Contents

# 目次

* 第1章 導入・環境構築
* 第2章 基本文法・Hello World
* 第3章 データ型と変数
* 第4章 演算子
* 第5章 制御構造（条件分岐）
* 第6章 制御構造（ループ）
* 第7章 配列
* 第8章 文字列処理
* 第9章 関数
* 第10章 ポインタ基礎
* 第11章 構造体とポインタ
* 第12章 関数ポインタ
* 第13章 複数ファイル・発展技術
* 第14章 C23の新機能（オプション）

# 第1章 導入・環境構築

## 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

## 理論解説

### 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。以下の特徴があります：

#### 主な特徴

* **システムプログラミング向け**: OS開発やハードウェア制御に適している
* **高い移植性**: 異なるプラットフォーム間でのコード再利用が容易
* **効率性**: メモリとCPUの使用効率が良い
* **シンプルな文法**: 基本的な機能に絞られた明確な仕様

#### 使用分野

* オペレーティングシステム（Linux、Windows等）
* 組込みシステム（マイコン、IoTデバイス）
* データベースシステム
* ゲームエンジン
* 科学技術計算

### C言語の規格

C言語は時代とともに進化し、複数の規格が策定されています：

| 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 |
| **C23** | 2024年 | ISO/IEC 9899:2024 | bool型標準化、typeof演算子、nullptr、2進数リテラル |

#### C23の新機能について

C23（ISO/IEC 9899:2024）は、C99以来の大規模な改訂となり、現代的なプログラミングのニーズに対応した多くの新機能が追加されました。主な新機能として：

* **bool型の標準化**: <stdbool.h>なしでbool、true、falseが使用可能
* **typeof演算子**: 式の型を取得できる演算子で、型安全なマクロ作成に有用
* **nullptr定数**: 型安全なヌルポインタ定数
* **2進数リテラル**: 0bプレフィックスで2進数表記が可能
* その他多数の改良と新機能

これらの詳細については、[第14章「C23の新機能」](../c23-features/README.md)で詳しく解説します。

#### 規格選択の指針

* **学習目的**: C90から始めて段階的に新機能を学習
* **業務開発**: C99以降を推奨（実用的な機能が多い）
* **組込み**: C90またはC99（コンパイラ対応状況による）
* **最新機能**: C23対応コンパイラ（gcc 13以降等）で最新機能を活用

### 開発環境

C言語プログラムの開発には以下のツールが必要です：

#### 必須ツール

1. **テキストエディター**: コードを記述するツール
2. **コンパイラ**: ソースコードを実行ファイルに変換
3. **リンカー**: 複数のオブジェクトファイルを結合
4. **デバッガー**: プログラムの動作を詳細に調査

#### 推奨開発環境

**Windows:**

# MinGW-w64のインストール（推奨）  
# または Microsoft Visual Studio Community

**macOS:**

# Xcode Command Line Toolsのインストール  
xcode-select --install

**Linux（Ubuntu/Debian）:**

sudo apt update  
sudo apt install build-essential

### コンパイル手順

C言語プログラムの実行までの流れ：

1. **ソースコード作成**（.cファイル）
2. **プリプロセッサ処理**（#includeの展開等）
3. **コンパイル**（アセンブリコードに変換）
4. **アセンブル**（オブジェクトファイルに変換）
5. **リンク**（実行ファイルの生成）
6. **実行**

#### 基本的なコンパイルコマンド

# 一段階でコンパイル  
gcc program.c -o program  
  
# 段階的なコンパイル  
gcc -c program.c # オブジェクトファイル作成  
gcc program.o -o program # 実行ファイル作成

# 第2章 基本文法・Hello World

## 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

## 理論解説

### プログラムの基本構造

C言語のプログラムは以下の基本構造を持ちます：

#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 \*/

#### 各部分の説明

1. **プリプロセッサ指令** (#include)
   * ヘッダーファイルを取り込む指令
   * stdio.hは標準入出力関数の宣言を含む
2. **main関数**
   * プログラムの実行開始点
   * すべてのC言語プログラムに必須
3. **関数本体**（{ と } で囲まれた部分）
   * 実際の処理を記述する場所
4. **return文**
   * 関数の終了と戻り値の指定

### 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。

#### main関数の形式

**標準的な形式:**

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

**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;

#### 戻り値の意味

* **0**: プログラムが正常終了
* **0以外**: エラーで終了（エラーコード）

### printf関数

printf関数は**フォーマット付き出力**を実行う標準ライブラリ関数です。

#### 基本的な使い方

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

#### フォーマット指定子

| 指定子 | データ型 | 説明 | 例 |
| --- | --- | --- | --- |
| %d | int | 10進整数 | printf("%d", 42); |
| %c | char | 文字 | printf("%c", 'A'); |
| %s | char\* | 文字列 | printf("%s", "Hello"); |
| %f | double | 浮動小数点数 | printf("%f", 3.14); |

#### エスケープシーケンス

特殊文字を表現するための記号です：

| シーケンス | 意味 | 説明 |
| --- | --- | --- |
| \n | 改行 | 次の実行に移る |
| \t | タブ | タブ文字を挿入 |
| \" | ダブルクォート | 文字列内で"を表示 |
| \\ | バックスラッシュ | \文字を表示 |

### 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;

# 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

## 理論解説

### C言語の基本データ型

C言語では、さまざまなデータ型が用意されています。

#### 整数型

| 型名 | サイズ | 範囲 | 説明 |
| --- | --- | --- | --- |
| char | 1バイト | -128 〜 127 | 文字、小さな整数 |
| unsigned char | 1バイト | 0 〜 255 | 符号なし文字 |
| short | 2バイト | -32,768 〜 32,767 | 短整数 |
| unsigned short | 2バイト | 0 〜 65,535 | 符号なし短整数 |
| int | 4バイト | -2,147,483,648 〜 2,147,483,647 | 整数 |
| unsigned int | 4バイト | 0 〜 4,294,967,295 | 符号なし整数 |
| long | 4/8バイト | システム依存 | 長整数 |
| unsigned long | 4/8バイト | システム依存 | 符号なし長整数 |

#### 浮動小数点数型

| 型名 | サイズ | 精度 | 説明 |
| --- | --- | --- | --- |
| float | 4バイト | 約7桁 | 単精度浮動小数点 |
| double | 8バイト | 約15桁 | 倍精度浮動小数点 |
| long double | 12/16バイト | システム依存 | 拡張精度浮動小数点 |

#### その他の型

| 型名 | 説明 |
| --- | --- |
| void | 「型なし」を表す（関数の戻り値等で使用） |

### 変数の宣言と初期化

#### 基本的な変数宣言

int age; /\* 整数型変数の宣言 \*/  
double height; /\* 倍精度浮動小数点型変数の宣言 \*/  
char grade; /\* 文字型変数の宣言 \*/

#### 初期化付き宣言

int count = 10; /\* 宣言と同時に初期化 \*/  
double pi = 3.14159; /\* 浮動小数点数の初期化 |  
char letter = 'A'; /\* 文字の初期化 \*/

#### 複数変数の同時宣言

int x, y, z; /\* 同じ型の複数変数 \*/  
int a = 1, b = 2, c; /\* 一部だけ初期化 \*/

### 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;

# 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

## 理論解説

### 算術演算子

基本的な数値計算を実行する演算子です。

| 演算子 | 意味 | 例 | 結果 |
| --- | --- | --- | --- |
| + | 加算 | 5 + 3 | 8 |
| - | 減算 | 5 - 3 | 2 |
| \* | 乗算 | 5 \* 3 | 15 |
| / | 除算 | 7 / 3 | 2 (整数除算) |
| % | 剰余 | 7 % 3 | 1 |

#### 除算の注意点

int a = 7, b = 3;  
int result = a / b; /\* 結果: 2 (整数除算) \*/  
double result = a / b; /\* 結果: 2.0 (整数除算後に型変換) \*/  
double result = (double)a / b; /\* 結果: 2.333... (実数除算) \*/

### 代入演算子

値を変数に代入する演算子です。

| 演算子 | 意味 | 例 | 等価な記述 |
| --- | --- | --- | --- |
| = | 代入 | a = 5 | - |
| += | 加算代入 | a += 3 | a = a + 3 |
| -= | 減算代入 | a -= 2 | a = a - 2 |
| \*= | 乗算代入 | a \*= 2 | a = a \* 2 |
| /= | 除算代入 | a /= 3 | a = a / 3 |
| %= | 剰余代入 | a %= 4 | a = a % 4 |

int count = 10;  
count += 5; /\* count は 15 になる \*/  
count \*= 2; /\* count は 30 になる \*/

### インクリメント・デクリメント演算子

変数の値を増減させる演算子です。

| 演算子 | 意味 | 前置 | 後置 |
| --- | --- | --- | --- |
| ++ | インクリメント | ++a | a++ |
| -- | デクリメント | --a | a-- |

#### 前置と後置の違い

int a = 5, b = 5;  
int x, y;  
  
x = ++a; /\* a を先にインクリメント, x = 6, a = 6 \*/  
y = b++; /\* b の値を先に使用, y = 5, b = 6 \*/

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

### 関係演算子

2つの値を比較する演算子です。結果は真（非0）または偽（0）になります。

| 演算子 | 意味 | 例 | 結果 |
| --- | --- | --- | --- |
| == | 等しい | 5 == 3 | 0 (偽) |
| != | 等しくない | 5 != 3 | 1 (真) |
| < | より小さい | 5 < 3 | 0 (偽) |
| <= | 以下 | 5 <= 3 | 0 (偽) |
| > | より大きい | 5 > 3 | 1 (真) |
| >= | 以上 | 5 >= 3 | 1 (真) |

### 論理演算子

論理値（真偽）を操作する演算子です。

| 演算子 | 意味 | 例 | 説明 |
| --- | --- | --- | --- |
| && | 論理積（AND） | a && b | a も b も真の場合のみ真 |
| || | 論理和（OR） | a || b | a または b が真なら真 |
| ! | 論理否定（NOT） | !a | a が偽なら真、真なら偽 |

#### 短絡評価

int a = 0, b = 10;  
  
if (a != 0 && b / a > 5) /\* a が 0 なので、b / a は評価されない \*/  
 /\* 実行されない \*/

### 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)

# 第5章 制御構造（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

## 理論解説

### if文の基本 🤔

if文は条件に基づいてプログラムの実行を分岐させます。

#### 基本構文

if (条件式)   
 /\* 条件が真の場合に実行される文 \*/

#### 単純なif文の例

#include <stdio.h>  
  
int main(void)  
{  
  
 int score = ;  
   
 if (score >= )   
 printf("優秀です！n");  
   
   
 return ;

### if-else文

条件が偽の場合の処理も指定できます。

#### 基本構文

if (条件式)   
 /\* 条件が真の場合の処理 \*/  
 else   
 /\* 条件が偽の場合の処理 \*/

#### if-else文の例

int age = ;  
  
if (age >= )   
 printf("成人ですn");  
 else   
 printf("未成年ですn");

### if-else if文

複数の条件を順次チェックできます。

#### 基本構文

if (条件)   
 /\* 条件が真の場合 \*/  
 else if (条件)   
 /\* 条件が真の場合 \*/  
 else if (条件)   
 /\* 条件が真の場合 \*/  
 else   
 /\* すべての条件が偽の場合 \*/

#### 成績判定の例

int score = ;  
  
if (score >= 9)   
 printf("成績: An");  
 else if (score >= )   
 printf("成績: n");  
 else if (score >= )   
 printf("成績: Cn");  
 else if (score >= )   
 printf("成績: Dn");  
 else   
 printf("成績: n");

### ネストしたif文

if文の中にさらにif文を書くことができます。

int temperatrue = ;  
int hmidity = ;  
  
if (temperatrue >= )   
 if (hmidity <= )   
 printf("快適な天気ですn");  
 else   
 printf("暖かいですが湿度が高いですn");  
   
 else   
 if (hmidity <= )   
 printf("涼しく乾燥していますn");  
 else   
 printf("寒くて湿気があります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。

#### 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）：

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：

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

#### 代入と比較の混同

int x = ;  
  
/\* NG: 代入になってしまう \*/  
if (x = )   
 printf("常に実行されるn"); /\* x に が代入され、常に真 \*/  
  
  
/\* OK: 比較演算子を使用 \*/  
if (x == )   
 printf("x が の場合のみ実行n");

#### 浮動小数点数の比較

double d = . + .;  
  
/\* NG: 浮動小数点の誤差で期待通りにならない可能性 \*/  
if (d == .)   
 printf("等しいn");  
  
  
/\* OK: 誤差を考慮した比較 \*/  
if (d >= .99999 && d <= .)   
 printf("ほぼ等しいn");

# 第6章 制御構造（ループ）

## 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

## 理論解説

### for文

最も一般的なループ文で、初期化・条件・更新を実行で記述できます。

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

#### さまざまなfor文のパターン

/\* 逆順ループ \*/  
for (i = 10; i >= 1; i--) {  
 printf("%d ", i);  
}  
  
  
/\* 2つずつ増加 \*/  
for (i = 0; i <= 20; i += 2) {  
 printf("%d ", i);  
}  
  
  
/\* 複数変数の制御 \*/  
for (i = 0, j = 10; i < j; i++, j--) {  
 printf("i=%d, j=%d\n", i, j);  
}

### while文

条件が真である間、繰り返し処理を実行します。

#### 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 = 1;  
   
 while (count <= 5) {  
 printf("count = %d\n", count);  
 count++;  
 }  
   
   
 return 0;

#### while文の実践例

/\* 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");

### 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。

#### 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("1. オプション1\n");  
 printf("2. オプション2\n");  
 printf("3. 終了\n");  
 printf("選択してください: ");  
 scanf("%d", &choice);  
   
 switch (choice) {  
 case 1:  
 printf("オプション1が選択されました\n");  
 break;  
 case 2:  
 printf("オプション2が選択されました\n");  
 break;  
 case 3:  
 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 != 3);  
   
 return 0;

### break文とcontinue文 ⏭

ループの流れを制御する特別な文です。

#### 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 = 1; i <= 10; i++) {  
 if (i == 5) {  
 break; /\* i が 5 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 0;  
  
/\* 出力: 1 2 3 4 \*/

#### 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 = 1; i <= 10; i++) {  
 if (i % 2 == 0) {  
 continue; /\* 偶数の場合はスキップ \*/  
 }  
 printf("%d ", i);  
 }  
 printf("\n");  
   
 return 0;  
  
/\* 出力: 1 3 5 7 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。

#### 二重ループの例

#include <stdio.h>  
  
int main(void)  
{  
  
 int i, j;  
   
 /\* 九九表の作成 \*/  
 for (i = 1; i <= 9; i++) {  
 for (j = 1; j <= 9; j++) {  
 printf("%3d ", i \* j);  
 }  
 printf("\n");  
 }  
   
 return 0;

#### 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 = 1; i <= 5; i++) {  
 for (j = 1; j <= i; j++) {  
 printf("\* ");  
 }  
 printf("\n");  
 }  
   
 return 0;  
  
/\*  
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 found = 0;  
   
 for (i = 1; i <= 10 && !found; i++) {  
 for (j = 1; j <= 10; j++) {  
 if (i \* j == 36) {  
 printf("発見: %d \* %d = 36\n", i, j);  
 found = 1;  
 break; /\* 内側のループを抜ける \*/  
 }  
 }  
 }  
   
 return 0;

#### ラベル付きbreak（goto文を使用）

#include <stdio.h>  
  
int main(void)  
{  
  
 int i, j;  
   
 for (i = 1; i <= 10; i++) {  
 for (j = 1; j <= 10; j++) {  
 if (i \* j == 36) {  
 printf("発見: %d \* %d = 36\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 0;

### 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。

#### 無限ループの作成方法

/\* 方法1: for文 \*/  
for (;;) {  
 /\* 無限ループ \*/  
 if (条件) {  
 break;  
 }  
}  
  
  
/\* 方法2: while文 \*/  
while (1) {  
 /\* 無限ループ \*/  
 if (条件) {  
 break;  
 }  
}  
  
  
/\* 方法3: do-while文 \*/  
do {  
 /\* 無限ループ \*/  
 if (条件) {  
 break;  
 }  
} while (1);

#### 無限ループの実用例

#include <stdio.h>  
  
int main(void)  
{  
  
 int choice;  
   
 while (1) { /\* 無限ループ \*/  
 printf("\n=== 計算機 ===\n");  
 printf("1. 足し算\n");  
 printf("2. 引き算\n");  
 printf("3. 終了\n");  
 printf("選択: ");  
   
 if (scanf("%d", &choice) != 1) {  
 printf("入力エラー\n");  
 break;  
 }  
   
 if (choice == 3) {  
 printf("終了します\n");  
 break;  
 }  
   
 /\* 計算処理... \*/  
 }  
   
 return 0;

### ループの最適化とベストプラクティス

#### 効率的なループ

/\* NG: 毎回strlen()を呼び出し \*/  
for (i = 0; i < strlen(str); i++) {  
 /\* 処理 \*/  
}  
  
  
/\* OK: 長さを事前に計算 \*/  
len = strlen(str);  
for (i = 0; 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 = 0; i < 10; i++) {  
 /\* iはループ外でも有効 \*/  
 }  
   
 printf("最終的なi = %d\n", i); /\* 10が出力 \*/  
   
 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

## 理論解説

### 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

#### 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）

#### 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

これがユーザーの重要な要望の一2つです。

#### 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。

#### 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

#### 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。

#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

# 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

## 理論解説

### 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、nll文字（’’）で終端されます。

#### 文字列の表現方法

#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 ;

#### 文字列の長さとサイズ

#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

#### 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

これがユーザーの重要な要望の2つです。

#### 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

#### 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

#### 文字列の分割（トークン化）

#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 ;

# 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

## 理論解説

### 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

#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;  
}

### 関数の定義と宣言

#### 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

#### 値渡し（Call by Value）

/\* 値渡しの例 \*/  
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 \*/  
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

#### 戻り値のない関数（void関数）

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。

/\* 階乗を計算する再帰関数 \*/  
int factorial(int n)  
{  
 /\* 基底条件 \*/  
 if (n <= 1)  
 {  
 return 1;  
 }  
   
 /\* 再帰呼び出し \*/  
 return n \* factorial(n - 1);  
}  
  
/\* 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

#### ローカル変数とグローバル変数

#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。

#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

# 第10章 ポインタ基礎

## 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

## 理論解説

### 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。

#### 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（\*）

#### 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

#### 配列名はポインタ

#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

#### 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 ;

#### 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 ;

# 第11章 構造体とポインタ

## 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

## 理論解説

### 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。

#### 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。

#### ->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。

#### 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;  
}

### 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;  
}

# 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

## 理論解説

### 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。

#### 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

#### 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。

#### 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。

#### 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

#### プラグインシステム

#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。

#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;  
}

# 第13章 複数ファイル・発展技術

## 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

## 理論解説

### 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。

#### 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。

#### 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。

#### 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。

#### 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 ;

### 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

# 第14章 C23の新機能（オプション）

## 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

## 理論解説

### C23とは

C23（正式にはISO/IEC 9899:2024）は、C言語の最新規格です。C99以来の大規模な改訂となり、現代的なプログラミングのニーズに対応した多くの新機能が追加されました。

### 主要な新機能

#### 1. bool型の標準化

C23では、<stdbool.h>をインクルードすることなく、bool、true、falseが使用可能になりました。

/\* C90/C99 \*/  
#include <stdbool.h> /\* 必要 \*/  
bool flag = true;  
  
/\* C23 \*/  
bool flag = true; /\* ヘッダー不要 \*/

#### 2. typeof演算子

式の型を取得できる演算子です。マクロでの型安全性向上に役立ちます。

int x = 42;  
typeof(x) y = 100; /\* yはint型 \*/  
  
/\* 型安全なマクロ \*/  
#define SWAP(a, b) do { \  
 typeof(a) temp = (a); \  
 (a) = (b); \  
 (b) = temp; \  
} while(0)

#### 3. nullptr定数

型安全なNULLポインタ定数です。

/\* 従来 \*/  
int \*p1 = NULL; /\* NULLは0やvoid\*として定義 \*/  
  
/\* C23 \*/  
int \*p2 = nullptr; /\* 明確にポインタ型 \*/

#### 4. 2進数リテラル

0bまたは0Bプレフィックスで2進数を直接記述できます。

int binary = 0b1010; /\* 10進数の10 \*/  
int flags = 0b11111111; /\* 10進数の255 \*/  
  
/\* ビット演算での活用 \*/  
#define READ\_BIT 0b100  
#define WRITE\_BIT 0b010  
#define EXEC\_BIT 0b001

### その他の新機能（一部）

* auto型推論（制限付き）
* constexpr（コンパイル時定数）
* \_BitInt(N)（任意ビット幅整数）
* 属性構文（[[deprecated]]など）
* プリプロセッサの拡張（#elifdef、#elifndef）