C言語プログラミング教材 - メインテキスト（応用編）

第7章〜第11章：実践的なプログラミング技術

C言語学習教材プロジェクト

2024年

Table of Contents

# はじめに

本書は、C言語プログラミング教材の応用編です。基礎編を修了した学習者を対象に、より実践的なプログラミング技術を習得できるよう構成されています。

## 前提知識

応用編を学習する前に、以下の内容を理解していることが必要です：

* 基本的なC言語の文法
* 変数とデータ型
* 演算子の使い方
* 条件分岐とループ

## 本書の構成

応用編では以下の内容を扱います：

* 第7章: 配列（基本編）
* 第8章: ポインタ基礎
* 第9章: 配列とポインタ
* 第10章: 文字列処理
* 第11章: 関数

## 学習のポイント

応用編では、C言語の最も重要な概念であるポインタを中心に学習します。ポインタは初学者にとって難しい概念ですが、メモリの仕組みを理解しながら段階的に学習することで、必ず習得できます。

# 配列（基本編）

## 対応C規格

* **主要対象:** C90
* **学習内容:** 一次元配列、多次元配列、配列の初期化、基本的な配列操作

## 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。 - 一次元配列の宣言・初期化・使用ができる - 多次元配列を理解して活用できる - 数値配列の基本操作（検索、統計計算）ができる - 配列を使った実践的なプログラムを作成できる

## 概要と詳細

### 配列とは？

配列（はいれつ）は、プログラミングで最も重要なデータ構造の一つです。

#### 日常生活での配列

配列を理解するために、身近な例を考えてみましょう。

1. **教室の座席**
   * 5行6列の座席 = 2次元配列
   * 各座席には生徒が座る = 配列の要素
   * 「3行目の4番目の席」= 配列の[3][4]
2. **アパートの部屋番号**
   * 101号室、102号室… = 1次元配列
   * 各部屋に住人がいる = 配列の要素
3. **カレンダー**
   * 曜日×週 = 2次元配列
   * 各マスに日付 = 配列の要素

#### なぜ配列が必要なの？

配列がないと、たくさんのデータを扱うのが大変になります。例えば5人の点数を管理する場合、配列なしでは変数を5つ作る必要がありますが、配列を使えば1つの変数で管理でき、100人でも1000人でも簡単に管理できます。

### 配列の基本概念

配列は同じデータ型の要素を連続したメモリ領域に格納するデータ構造です。

#### 配列の3つの重要な特徴

1. **同じ型の集まり**
   * すべての要素が同じデータ型でなければならない
   * int型の配列にはint型の値のみ格納可能
2. **連続したメモリ領域**
   * 要素は隣り合ったメモリ上に配置される
   * このため高速なアクセスが可能
3. **固定サイズ**
   * 配列のサイズは宣言時に決まり、後から変更できない
   * 動的なサイズ変更が必要な場合は動的メモリ確保を使用

#### 配列のメモリ上の配置

配列 int numbers[5] のメモリ配置：  
  
インデックス: [0] [1] [2] [3] [4]  
 +------+------+------+------+------+  
値: | 10 | 20 | 30 | 40 | 50 |  
 +------+------+------+------+------+

#### 1次元配列の宣言

データ型 配列名[要素数];

例えば、5つの整数を格納する配列は以下のように宣言します。

int numbers[5]; /\* 5つのint型要素を持つ配列 \*/

### 配列の初期化

配列を使用する前に適切な値で初期化することが重要です。

#### 宣言時の初期化

int numbers[5] = {10, 20, 30, 40, 50};

#### 部分的な初期化

int numbers[5] = {10, 20}; /\* 残りの3つは0で初期化される \*/

#### サイズを省略した初期化

int numbers[] = {10, 20, 30, 40, 50}; /\* コンパイラが自動的にサイズを決定 \*/

### 配列の操作

配列の要素にアクセスするには、添字（インデックス）を使用します。

#### 要素へのアクセス

numbers[0] = 10; /\* 最初の要素に値を代入 \*/  
int value = numbers[2]; /\* 3番目の要素の値を取得 \*/

**重要な注意点：** - 配列の添字は0から始まります - 要素数がnの配列の有効な添字は0からn-1まで - 範囲外アクセスは未定義動作を引き起こします

### 配列の基本操作

数値配列では以下のような操作が頻繁に行われます。

#### 配列内の値を検索する

#include <stdio.h>  
  
int main(void) {  
 int numbers[10] = {3, 7, 1, 9, 4, 6, 8, 2, 5, 0};  
 int target = 6; /\* 探したい値 \*/  
 int found = -1; /\* 見つかった位置（-1は見つからない） \*/  
 int i;  
   
 /\* 配列を検索 \*/  
 for (i = 0; i < 10; i++) {  
 if (numbers[i] == target) {  
 found = i; /\* 見つかった位置を記録 \*/  
 break; /\* 見つかったらループを抜ける \*/  
 }  
 }  
   
 /\* 結果を表示 \*/  
 if (found != -1) {  
 printf("%d は位置 %d にあります\n", target, found);  
 } else {  
 printf("%d は配列内にありません\n", target);  
 }  
   
 return 0;  
}

#### 配列の統計計算（最大値・最小値・平均）

#include <stdio.h>  
  
int main(void) {  
 int scores[5] = {85, 92, 78, 96, 88};  
 int max, min, sum;  
 double average;  
 int i;  
   
 /\* 初期値の設定 \*/  
 max = scores[0];  
 min = scores[0];  
 sum = scores[0];  
   
 /\* 2番目の要素から比較開始 \*/  
 for (i = 1; i < 5; i++) {  
 /\* 最大値の更新 \*/  
 if (scores[i] > max) {  
 max = scores[i];  
 }  
   
 /\* 最小値の更新 \*/  
 if (scores[i] < min) {  
 min = scores[i];  
 }  
   
 /\* 合計に加算 \*/  
 sum += scores[i];  
 }  
   
 /\* 平均値の計算 \*/  
 average = (double)sum / 5;  
   
 /\* 結果を表示 \*/  
 printf("最大値: %d\n", max);  
 printf("最小値: %d\n", min);  
 printf("合計: %d\n", sum);  
 printf("平均値: %.1f\n", average);  
   
 return 0;  
}

#### 配列の要素を逆順にする

#include <stdio.h>  
  
int main(void) {  
 int arr[6] = {1, 2, 3, 4, 5, 6};  
 int temp;  
 int i;  
   
 /\* 元の配列を表示 \*/  
 printf("元の配列: ");  
 for (i = 0; i < 6; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
   
 /\* 配列を逆順にする \*/  
 for (i = 0; i < 6 / 2; i++) {  
 /\* 前後の要素を交換 \*/  
 temp = arr[i];  
 arr[i] = arr[5 - i];  
 arr[5 - i] = temp;  
 }  
   
 /\* 逆順にした配列を表示 \*/  
 printf("逆順配列: ");  
 for (i = 0; i < 6; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
   
 return 0;  
}

### 多次元配列

配列の配列を作ることで、表やマトリックスのような構造を表現できます。

#### 2次元配列の宣言

int matrix[3][4]; /\* 3行4列の配列 \*/

#### 2次元配列の初期化

int matrix[2][3] = {  
 {1, 2, 3},  
 {4, 5, 6}  
};

#### 2次元配列のアクセス

#include <stdio.h>  
  
int main(void) {  
 int matrix[3][3] = {  
 {1, 2, 3},  
 {4, 5, 6},  
 {7, 8, 9}  
 };  
 int i, j;  
   
 /\* 2次元配列の全要素を表示 \*/  
 printf("行列の内容:\n");  
 for (i = 0; i < 3; i++) {  
 for (j = 0; j < 3; j++) {  
 printf("%2d ", matrix[i][j]);  
 }  
 printf("\n");  
 }  
   
 return 0;  
}

#### 3次元配列

int cube[2][3][4]; /\* 2×3×4の3次元配列 \*/

### 配列の実践的応用

配列は様々な場面で活用できます。

1. **統計計算** - 数値の集合から平均、最大値、最小値を計算
2. **データ検索** - 特定の値を配列から探す
3. **ソート** - データを昇順や降順に並べ替える
4. **行列演算** - 数学的な行列計算
5. **画像処理** - 画素データの管理

## 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。

### 基本的な配列操作

* [array\_basics.c](examples/array_basics.c) - C90準拠版
* [array\_basics\_c99.c](examples/array_basics_c99.c) - C99準拠版

### 多次元配列の活用

* [multidimensional\_arrays.c](examples/multidimensional_arrays.c) - C90準拠版
* [multidimensional\_arrays\_c99.c](examples/multidimensional_arrays_c99.c) - C99準拠版

## 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

## 学習フローとコンパイル方法

### 推奨学習順序

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

## C90とC99の違い

### C90の特徴（このチュートリアルの基準）

* **変数宣言**: 関数の先頭でまとめて宣言する必要がある
* **配列サイズ**: コンパイル時に決定される固定サイズのみ
* **初期化**: 宣言と同時に行う必要がある
* **コメント**: /\* \*/ 形式のみ使用可能

### C99の拡張機能

* **変数宣言**: 使用する場所で宣言可能
* **可変長配列（VLA）**: 実行時にサイズを決定可能
* **配列の指定初期化**: arr[5] = {[2] = 10, [4] = 20}
* **行コメント**: // 形式のコメントが使用可能

## よくある間違いとデバッグ方法

### 1. 配列の境界外アクセス

**問題:** 配列のサイズを超えた添字でアクセス

int arr[5] = {1, 2, 3, 4, 5};  
int value = arr[5]; /\* エラー：添字は0-4が有効 \*/

**対策:** - 常に配列のサイズを意識する - ループの条件を正しく設定する - デバッガーやランタイムチェックツールを使用

### 2. 配列の初期化忘れ

**問題:** 配列が適切に初期化されていない

int arr[10];  
/\* arr[0]から arr[9]の値は不定 \*/

**対策:** - 使用前に必ず初期化する - 宣言時に初期化を同時に行う - ループで初期値を設定する

### 3. 配列代入の間違い

**問題:** 配列全体を直接代入しようとする

int arr1[5] = {1, 2, 3, 4, 5};  
int arr2[5];  
arr2 = arr1; /\* エラー：配列は直接代入不可 \*/

**対策:** - ループを使用して要素を1つずつコピー - 標準ライブラリの関数を使用（後の章で学習）

## 次の章へ

配列の基本を理解したら、[第8章: ポインタ基礎](../pointers/README.md) に進んでください。ポインタを学ぶことで、配列のより深い理解が得られます。

## 参考資料

* [C90規格書](https://www.iso.org/standard/17782.html)
* [C99規格書](https://www.iso.org/standard/29237.html)

# 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、ポインタと配列、ポインタと関数、メモリ管理の基礎

## 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完全に理解する - ポインタを使った変数の操作ができる - ポインタと配列の関係を理解する - 関数でポインタを適切に使える - 動的メモリ割り当ての基礎を理解する

## 概要と詳細

### ポインタとは？

ポインタは、他の変数のメモリアドレス（住所）を格納する変数です。C言語の最も強力で重要な機能の一つです。

#### 日常生活でのポインタ

ポインタを理解するために、身近な例を考えてみましょう。

1. **住所録**
   * 友人の名前 = 変数名
   * 友人の住所 = ポインタ（メモリアドレス）
   * 友人の家 = 実際のデータ
2. **図書館の目録**
   * 本のタイトル = 変数名
   * 本の所在（A棚3段目） = ポインタ
   * 実際の本 = データ
3. **リンク集**
   * サイト名 = 変数名
   * URL = ポインタ
   * 実際のWebページ = データ

#### なぜポインタが必要なの？

1. **効率的なデータ操作** - 大きなデータをコピーせず、アドレスだけを渡す
2. **動的メモリ管理** - 実行時にメモリを確保・解放
3. **関数間でのデータ共有** - 元の変数を直接変更
4. **複雑なデータ構造** - リスト、ツリーなどの実装
5. **配列の効率的な処理** - 配列要素への高速アクセス

### ポインタの基本概念

#### ポインタの宣言

int \*ptr; /\* int型へのポインタ \*/  
char \*str\_ptr; /\* char型へのポインタ \*/  
double \*d\_ptr; /\* double型へのポインタ \*/

#### アドレス演算子（&）

変数のアドレスを取得します。

int num = 42;  
int \*ptr = &num; /\* numのアドレスをptrに格納 \*/

#### 間接参照演算子（\*）

ポインタが指す場所の値にアクセスします。

int value = \*ptr; /\* ptrが指す場所の値を取得 \*/  
\*ptr = 100; /\* ptrが指す場所に100を代入 \*/

### ポインタの基本操作

#### 基本的な使い方

int num = 42;  
int \*ptr;  
  
ptr = &num; /\* numのアドレスをptrに代入 \*/  
printf("numの値: %d\n", num);  
printf("numのアドレス: %p\n", &num);  
printf("ptrの値（アドレス）: %p\n", ptr);  
printf("ptrが指す値: %d\n", \*ptr);

#### ポインタを使った値の変更

int num = 10;  
int \*ptr = &num;  
  
printf("変更前: %d\n", num); /\* 10 \*/  
\*ptr = 20;  
printf("変更後: %d\n", num); /\* 20 \*/

### ポインタと配列の関係

#### 配列名はポインタ

C言語では、配列名は最初の要素へのポインタと同じです。

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

#### ポインタ演算

ポインタには加算・減算ができます。

int arr[5] = {10, 20, 30, 40, 50};  
int \*ptr = arr;  
  
printf("%d\n", \*ptr); /\* 10 (arr[0]) \*/  
printf("%d\n", \*(ptr+1)); /\* 20 (arr[1]) \*/  
printf("%d\n", \*(ptr+2)); /\* 30 (arr[2]) \*/

#### 配列のポインタによる操作

int arr[5] = {1, 2, 3, 4, 5};  
int \*ptr = arr;  
int i;  
  
/\* ポインタを使った配列要素のアクセス \*/  
for (i = 0; i < 5; i++) {  
 printf("arr[%d] = %d\n", i, \*(ptr + i));  
}  
  
/\* ポインタのインクリメント \*/  
for (ptr = arr; ptr < arr + 5; ptr++) {  
 printf("値: %d\n", \*ptr);  
}

### ポインタと関数

#### 関数の引数としてのポインタ

関数で元の変数を変更するためにポインタを使用します。

/\* 値を交換する関数 \*/  
void swap(int \*a, int \*b)  
{  
 int temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
}  
  
int main(void)  
{  
 int x = 10, y = 20;  
   
 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);  
   
 return 0;  
}

#### 配列を関数に渡す

/\* 配列の合計を計算する関数 \*/  
int array\_sum(int \*arr, int size)  
{  
 int sum = 0;  
 int i;  
   
 for (i = 0; i < size; i++) {  
 sum += arr[i]; /\* または sum += \*(arr + i); \*/  
 }  
   
 return sum;  
}  
  
int main(void)  
{  
 int numbers[5] = {1, 2, 3, 4, 5};  
 int total = array\_sum(numbers, 5);  
 printf("合計: %d\n", total);  
 return 0;  
}

#### 関数からポインタを返す

/\* 配列の最大値を指すポインタを返す \*/  
int\* find\_max(int \*arr, int size)  
{  
 int \*max\_ptr = arr;  
 int i;  
   
 for (i = 1; i < size; i++) {  
 if (\*(arr + i) > \*max\_ptr) {  
 max\_ptr = arr + i;  
 }  
 }  
   
 return max\_ptr;  
}

### 文字列とポインタ

#### 文字列の表現方法

char str1[] = "Hello"; /\* 文字配列 \*/  
char \*str2 = "World"; /\* 文字列リテラルへのポインタ \*/

#### 文字列の操作

char \*str = "Hello";  
char \*ptr = str;  
  
/\* 文字列を1文字ずつ表示 \*/  
while (\*ptr != '\0') {  
 printf("%c", \*ptr);  
 ptr++;  
}  
printf("\n");

### 多次元配列とポインタ

#### 2次元配列へのポインタ

int matrix[3][4] = {  
 {1, 2, 3, 4},  
 {5, 6, 7, 8},  
 {9, 10, 11, 12}  
};  
  
int (\*ptr)[4] = matrix; /\* 4要素のint配列へのポインタ \*/

#### 2次元配列の要素アクセス

/\* 以下はすべて同じ要素にアクセス \*/  
matrix[1][2]; /\* 通常のアクセス \*/  
(\*(matrix + 1))[2]; /\* ポインタ演算1 \*/  
\*(\*(matrix + 1) + 2); /\* ポインタ演算2 \*/

### 動的メモリ割り当て

#### malloc()とfree()

#include <stdlib.h>  
  
int \*ptr;  
int size = 5;  
  
/\* メモリの動的確保 \*/  
ptr = (int\*)malloc(size \* sizeof(int));  
  
if (ptr == NULL) {  
 printf("メモリ確保に失敗しました\n");  
 return 1;  
}  
  
/\* メモリの使用 \*/  
for (i = 0; i < size; i++) {  
 ptr[i] = i \* 2;  
}  
  
/\* メモリの解放 \*/  
free(ptr);  
ptr = NULL; /\* ダングリングポインタの防止 \*/

#### calloc()とrealloc()

/\* calloc: ゼロ初期化されたメモリを確保 \*/  
int \*arr = (int\*)calloc(5, sizeof(int));  
  
/\* realloc: メモリサイズを変更 \*/  
arr = (int\*)realloc(arr, 10 \* sizeof(int));  
  
free(arr);

### ポインタのポインタ

#### 2重ポインタの概念

int num = 42;  
int \*ptr = &num; /\* numへのポインタ \*/  
int \*\*dptr = &ptr; /\* ptrへのポインタ \*/  
  
printf("num = %d\n", num);  
printf("\*ptr = %d\n", \*ptr);  
printf("\*\*dptr = %d\n", \*\*dptr);

#### 2重ポインタの用途

* 関数でポインタ自体を変更する
* 動的な2次元配列の作成
* 複雑なデータ構造の実装

### よくある間違いとデバッグ

#### 1. 未初期化ポインタ

**問題:** ポインタを初期化せずに使用

int \*ptr;  
\*ptr = 10; /\* 危険！未初期化ポインタの使用 \*/

**対策:**

int \*ptr = NULL; /\* NULLで初期化 \*/  
int num;  
ptr = &num; /\* 適切なアドレスを代入してから使用 \*/  
\*ptr = 10; /\* 安全 \*/

#### 2. ダングリングポインタ

**問題:** 解放されたメモリを指すポインタ

int \*ptr = (int\*)malloc(sizeof(int));  
free(ptr);  
\*ptr = 10; /\* 危険！解放済みメモリにアクセス \*/

**対策:**

int \*ptr = (int\*)malloc(sizeof(int));  
free(ptr);  
ptr = NULL; /\* NULLを代入 \*/

#### 3. 配列の境界外アクセス

**問題:** ポインタ演算での範囲外アクセス

int arr[5] = {1, 2, 3, 4, 5};  
int \*ptr = arr;  
printf("%d\n", \*(ptr + 10)); /\* 危険！境界外アクセス \*/

**対策:**

int arr[5] = {1, 2, 3, 4, 5};  
int \*ptr = arr;  
if (ptr + 2 < arr + 5) { /\* 境界チェック \*/  
 printf("%d\n", \*(ptr + 2));  
}

#### 4. メモリリーク

**問題:** 確保したメモリを解放し忘れ

void function()  
{  
 int \*ptr = (int\*)malloc(100 \* sizeof(int));  
 /\* free(ptr); を忘れる \*/  
} /\* メモリリーク発生 \*/

**対策:**

void function()  
{  
 int \*ptr = (int\*)malloc(100 \* sizeof(int));  
 if (ptr == NULL) return;  
   
 /\* 処理 \*/  
   
 free(ptr); /\* 必ず解放 \*/  
 ptr = NULL;  
}

### ポインタの高度な使用法

#### 関数ポインタ

/\* 関数の定義 \*/  
int add(int a, int b) { return a + b; }  
int multiply(int a, int b) { return a \* b; }  
  
/\* 関数ポインタの使用 \*/  
int (\*operation)(int, int);  
operation = add;  
printf("5 + 3 = %d\n", operation(5, 3));  
  
operation = multiply;  
printf("5 \* 3 = %d\n", operation(5, 3));

#### 構造体とポインタ

struct point {  
 int x;  
 int y;  
};  
  
struct point p = {10, 20};  
struct point \*ptr = &p;  
  
/\* 構造体メンバへのアクセス \*/  
printf("x = %d, y = %d\n", ptr->x, ptr->y);  
/\* または \*/  
printf("x = %d, y = %d\n", (\*ptr).x, (\*ptr).y);

## 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。

### ポインタの基本操作

* [pointer\_basics.c](examples/pointer_basics.c) - C90準拠版
* [pointer\_basics\_c99.c](examples/pointer_basics_c99.c) - C99準拠版

### ポインタと配列

* [pointer\_arrays.c](examples/pointer_arrays.c) - C90準拠版
* [pointer\_arrays\_c99.c](examples/pointer_arrays_c99.c) - C99準拠版

### 動的メモリ管理

* [dynamic\_memory.c](examples/dynamic_memory.c) - C90準拠版
* [dynamic\_memory\_c99.c](examples/dynamic_memory_c99.c) - C99準拠版

## 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 pointer\_basics.c -o pointer\_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 pointer\_basics  
# C99版をコンパイル  
make pointer\_basics\_c99  
# プログラムを実行  
make run-all  
# クリーンアップ  
make clean

## 学習フローとコンパイル方法

### 推奨学習順序

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

### 学習のコツ

1. **視覚化**: メモリ図を描いてポインタの動作を理解
2. **段階的学習**: 基本→配列→関数→動的メモリの順で学習
3. **デバッグ技術**: printfでアドレスと値を出力して確認
4. **実践練習**: 小さなプログラムから始めて徐々に複雑に

## 注意事項とベストプラクティス

### セキュリティの考慮

1. **NULL チェック**: ポインタ使用前の NULL チェック
2. **境界チェック**: 配列アクセス時の範囲確認
3. **メモリ管理**: malloc/free のペア、ダブルフリーの防止
4. **初期化**: ポインタの適切な初期化

### パフォーマンスの最適化

1. **ポインタ演算**: 配列アクセスの高速化
2. **参照渡し**: 大きなデータのコピー回避
3. **キャッシュ効率**: メモリアクセスパターンの最適化

## 次の章へ

[配列とポインタ](../arrays-pointers/README.md)

## 参考資料

* [C標準ライブラリリファレンス](https://en.cppreference.com/w/c)
* [セキュアコーディング標準](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177)

# 配列とポインタ

## 対応C規格

* **主要対象:** C90
* **学習内容:** 配列とポインタの関係、ポインタを使った配列操作、関数への配列渡し

## 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。 - 配列名がポインタとして扱われることを理解する - ポインタ演算を使った配列操作ができる - 配列を関数に渡す仕組みを理解する - より効率的な配列処理ができる

## 概要と詳細

### 前提知識

この章を学習する前に、以下の章を完了していることを前提とします： - 第7章: 配列（基本編） - 第8章: ポインタ基礎

### 配列とポインタの関係

C言語において、配列とポインタには密接な関係があります。この関係を理解することは、C言語をマスターする上で非常に重要です。

#### 配列名の本質

配列名は、配列の最初の要素へのポインタとして扱われます。

int arr[5] = {10, 20, 30, 40, 50};  
  
/\* 以下の3つは同じアドレスを示す \*/  
printf("arr = %p\n", arr); /\* 配列名 \*/  
printf("&arr[0] = %p\n", &arr[0]); /\* 最初の要素のアドレス \*/  
printf("&arr = %p\n", &arr); /\* 配列全体のアドレス（値は同じ） \*/

#### 視覚的な理解

配列 int arr[5] = {10, 20, 30, 40, 50} のメモリ配置：  
  
アドレス: 1000 1004 1008 1012 1016  
 +-------+-------+-------+-------+-------+  
 | 10 | 20 | 30 | 40 | 50 |  
 +-------+-------+-------+-------+-------+  
 ↑  
 arr（配列名）は、この最初の要素のアドレス（1000）を表す

### 配列要素へのアクセス方法の等価性

#### 2つの記法

C言語では、配列要素にアクセスする方法が2つあります。

int arr[5] = {10, 20, 30, 40, 50};  
int i;  
  
/\* 以下の2つの方法は完全に等価 \*/  
for (i = 0; i < 5; i++) {  
 printf("arr[%d] = %d\n", i, arr[i]); /\* 配列記法 \*/  
 printf("\*(arr+%d) = %d\n", i, \*(arr+i)); /\* ポインタ記法 \*/  
}  
  
/\* つまり、arr[i] は \*(arr + i) の糖衣構文（シンタックスシュガー） \*/

#### なぜこれが等価なのか？

arr[i] の仕組み：  
  
1. arr は配列の先頭アドレス（例: 1000）  
2. i は配列のインデックス（例: 2）  
3. arr + i は、arr から i要素分進んだアドレス  
 - 実際のアドレス = 1000 + (2 × 4) = 1008  
 - （int型は4バイトなので）  
4. \*(arr + i) は、そのアドレスの値を取得  
5. これは arr[2] と同じ！

### 配列とポインタの違い

配列名はポインタのように振る舞いますが、重要な違いがあります。

#### 配列名は定数

int arr[5] = {1, 2, 3, 4, 5};  
int \*ptr = arr; /\* OK: 配列の先頭アドレスをポインタに代入 \*/  
  
/\* 配列名は定数（変更不可） \*/  
arr = ptr; /\* エラー: 配列名への代入はできない \*/  
arr++; /\* エラー: 配列名の変更はできない \*/  
  
/\* ポインタは変数（変更可能） \*/  
ptr++; /\* OK: ポインタを次の要素に移動 \*/  
ptr = &arr[3]; /\* OK: ポインタに別のアドレスを代入 \*/

#### サイズの違い

int arr[5] = {1, 2, 3, 4, 5};  
int \*ptr = arr;  
  
printf("sizeof(arr) = %lu\n", sizeof(arr)); /\* 20（5×4バイト） \*/  
printf("sizeof(ptr) = %lu\n", sizeof(ptr)); /\* 8（64ビット環境）or 4（32ビット環境） \*/

### ポインタ演算による配列操作

#### ポインタの移動

#include <stdio.h>  
  
int main(void) {  
 int arr[5] = {10, 20, 30, 40, 50};  
 int \*ptr = arr;  
   
 printf("ptr が指す値: %d\n", \*ptr); /\* 10 \*/  
   
 ptr++; /\* 次の要素へ移動 \*/  
 printf("ptr が指す値: %d\n", \*ptr); /\* 20 \*/  
   
 ptr += 2; /\* 2つ先の要素へ移動 \*/  
 printf("ptr が指す値: %d\n", \*ptr); /\* 40 \*/  
   
 return 0;  
}

#### ポインタを使った配列の走査

#include <stdio.h>  
  
int main(void) {  
 int arr[5] = {1, 2, 3, 4, 5};  
 int \*ptr;  
   
 /\* 方法1: インデックスを使用 \*/  
 printf("インデックスを使用:\n");  
 for (ptr = arr; ptr < arr + 5; ptr++) {  
 printf("%d ", \*ptr);  
 }  
 printf("\n");  
   
 /\* 方法2: ポインタの比較 \*/  
 printf("ポインタの比較:\n");  
 ptr = arr;  
 while (ptr <= &arr[4]) {  
 printf("%d ", \*ptr);  
 ptr++;  
 }  
 printf("\n");  
   
 return 0;  
}

### 関数への配列の渡し方

配列を関数に渡すとき、実際には先頭要素へのポインタが渡されます。

#### 関数の宣言方法

/\* これらの宣言は同じ意味 \*/  
void process\_array(int arr[]); /\* 配列として宣言 \*/  
void process\_array(int \*arr); /\* ポインタとして宣言 \*/  
void process\_array(int arr[10]); /\* サイズ指定（無視される） \*/

#### 配列を受け取る関数の例

#include <stdio.h>  
  
/\* 配列の合計を計算する関数 \*/  
int array\_sum(int \*arr, int size) {  
 int sum = 0;  
 int i;  
   
 for (i = 0; i < size; i++) {  
 sum += arr[i]; /\* または sum += \*(arr + i); \*/  
 }  
   
 return sum;  
}  
  
/\* 配列の要素をすべて2倍にする関数 \*/  
void double\_array(int arr[], int size) {  
 int i;  
   
 for (i = 0; i < size; i++) {  
 arr[i] \*= 2; /\* 元の配列が変更される \*/  
 }  
}  
  
int main(void) {  
 int numbers[5] = {1, 2, 3, 4, 5};  
 int total;  
 int i;  
   
 /\* 合計を計算 \*/  
 total = array\_sum(numbers, 5);  
 printf("合計: %d\n", total);  
   
 /\* 配列を2倍に \*/  
 printf("元の配列: ");  
 for (i = 0; i < 5; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 double\_array(numbers, 5);  
   
 printf("2倍後: ");  
 for (i = 0; i < 5; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 return 0;  
}

### 多次元配列とポインタ

#### 2次元配列のメモリ配置

int matrix[3][4] = {  
 {1, 2, 3, 4},  
 {5, 6, 7, 8},  
 {9, 10, 11, 12}  
};  
  
/\* メモリ上では一次元的に配置される \*/  
/\* [1][2][3][4][5][6][7][8][9][10][11][12] \*/

#### 2次元配列へのポインタ

int matrix[3][4];  
int (\*ptr)[4] = matrix; /\* 4要素のint配列へのポインタ \*/  
  
/\* 以下はすべて同じ要素にアクセス \*/  
matrix[1][2]; /\* 通常のアクセス \*/  
\*(\*(matrix + 1) + 2); /\* ポインタ演算 \*/  
ptr[1][2]; /\* ポインタ経由のアクセス \*/

### 実践的な応用例

#### 配列の最大値を見つける（ポインタ版）

#include <stdio.h>  
  
int\* find\_max(int \*arr, int size) {  
 int \*max\_ptr = arr;  
 int i;  
   
 for (i = 1; i < size; i++) {  
 if (\*(arr + i) > \*max\_ptr) {  
 max\_ptr = arr + i;  
 }  
 }  
   
 return max\_ptr; /\* 最大値へのポインタを返す \*/  
}  
  
int main(void) {  
 int numbers[5] = {34, 67, 12, 89, 45};  
 int \*max\_ptr;  
   
 max\_ptr = find\_max(numbers, 5);  
   
 printf("最大値: %d\n", \*max\_ptr);  
 printf("最大値の位置: %ld\n", max\_ptr - numbers);  
   
 return 0;  
}

#### 文字列操作の基礎

#include <stdio.h>  
  
/\* 文字列の長さを計算（ポインタ版） \*/  
int string\_length(char \*str) {  
 char \*start = str;  
   
 while (\*str != '\0') {  
 str++;  
 }  
   
 return str - start; /\* ポインタの差が文字数 \*/  
}  
  
int main(void) {  
 char message[] = "Hello, World!";  
 int len;  
   
 len = string\_length(message);  
 printf("文字列の長さ: %d\n", len);  
   
 return 0;  
}

### よくある間違いと注意点

#### 1. 配列の境界を超えたアクセス

int arr[5] = {1, 2, 3, 4, 5};  
int \*ptr = arr;  
  
ptr += 10; /\* 危険！配列の範囲外 \*/  
\*ptr = 100; /\* 未定義動作 \*/

#### 2. ポインタと配列の混同

int arr[5];  
int \*ptr;  
  
sizeof(arr); /\* 20: 配列全体のサイズ \*/  
sizeof(ptr); /\* 4 or 8: ポインタのサイズ \*/

#### 3. 関数での配列サイズ

void process(int arr[]) {  
 /\* sizeof(arr) はポインタのサイズを返す！ \*/  
 /\* 配列のサイズは別途引数で渡す必要がある \*/  
}

## 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。

### 配列とポインタの基本

* [array\_pointer\_basics.c](examples/array_pointer_basics.c) - C90準拠版
* [array\_pointer\_basics\_c99.c](examples/array_pointer_basics_c99.c) - C99準拠版

### 高度な配列操作

* [advanced\_array\_ops.c](examples/advanced_array_ops.c) - C90準拠版
* [advanced\_array\_ops\_c99.c](examples/advanced_array_ops_c99.c) - C99準拠版

## 学習のポイント

1. **配列名 = ポインタ定数**: 配列名は変更できないポインタ
2. \*\*arr[i] = \*(arr + i)\*\*: 2つの記法は完全に等価
3. **関数への渡し方**: 配列は常にポインタとして渡される
4. **サイズ情報**: 関数では配列のサイズが分からないので別途渡す

## 次の章へ

配列とポインタの関係を理解したら、[文字列処理](../strings/README.md) に進んでください。文字列は文字の配列として扱われるため、この知識が活かされます。

## 参考資料

* [C90規格書](https://www.iso.org/standard/17782.html)
* [ポインタと配列の詳細](https://en.cppreference.com/w/c/language/array)

# 文字列処理

## 対応C規格

* **主要対象:** C90
* **学習内容:** 文字列の基本、文字列操作関数、文字列配列の詳細操作、文字列の比較・検索・変換

## 学習目標

この章を完了すると、以下のことができるようになります。 - 文字列の基本概念を完全に理解する - 標準ライブラリの文字列操作関数を使いこなせる - 文字列配列の詳細な操作ができる - 次元文字配列と文字列ポインタ配列を使い分けられる - 実践的な文字列処理プログラムを作成できる

## 概要と詳細

### 文字列処理とは？

文字列処理は、プログラミングで最も頻繁に行う作業の一つです。名前、メッセージ、ファイル名など、私たちが扱うデータの多くは文字列です。

#### 日常生活での文字列処理

身近な例を考えてみましょう。

1. **メールアドレスの検証**
   * “@”が含まれているか？
   * 適切な形式か？
2. **パスワードのチェック**
   * 8文字以上か？
   * 大文字・小文字・数字が含まれているか？
3. **名前の整形**
   * 姓と名を分ける
   * 大文字・小文字を整える

これらはすべて文字列処理の例です！

### 文字列の基本概念

C言語では文字列は文字の配列として表現され、null文字（‘\0’）で終端されます。この仕組みを理解することが、文字列処理をマスターする第一歩です。

#### 文字列とは何か？

C言語における文字列は、実は「文字の配列」に過ぎません。ただし、特別なルールがあります。

文字列 "Hello" のメモリ上の配置：  
  
インデックス: [0] [1] [2] [3] [4] [5]  
内容: 'H' 'e' 'l' 'l' 'o' '\0'  
 ↑ ↑  
 最初の文字 null終端文字（文字列の終わり）

#### 文字列の4つの表現方法

/\* 1. 文字配列として明示的に初期化 \*/  
char str1[] = {'H', 'e', 'l', 'l', 'o', '\0'};   
/\* 各文字を個別に指定。最後に必ず'\0'が必要 \*/  
  
/\* 2. 文字列リテラルで初期化（最も一般的） \*/  
char str2[] = "Hello";  
/\* コンパイラが自動的に'\0'を追加。サイズは6バイト \*/  
  
/\* 3. 固定サイズ配列に文字列を格納 \*/  
char str3[10] = "Hello";  
/\* 10バイト確保。"Hello\0"の後の4バイトは'\0'で初期化 \*/  
  
/\* 4. 文字列リテラルへのポインタ \*/  
char \*str4 = "Hello";  
/\* 文字列リテラルは読み取り専用領域に配置される \*/

#### null終端文字（‘\0’）の重要性

null終端文字は、文字列の終わりを示す特別な文字です。

char str[] = "ABC";  
/\* メモリ上では: 'A' 'B' 'C' '\0' と格納される \*/  
  
/\* null終端がない場合の問題 \*/  
char bad\_str[3] = {'A', 'B', 'C'}; /\* '\0'がない！ \*/  
printf("%s", bad\_str); /\* どこまで表示されるか不明（危険） \*/

#### 文字と文字列の違い

初心者が混同しやすい重要な違いです。

/\* 文字（1バイト） \*/  
char ch = 'A'; /\* シングルクォート \*/  
  
/\* 文字列（2バイト以上） \*/  
char str[] = "A"; /\* ダブルクォート \*/  
/\* 実際は 'A' と '\0' の2バイト \*/  
  
/\* 間違いやすい例 \*/  
char wrong1 = "A"; /\* エラー：文字列を文字型に代入 \*/  
char wrong2[] = 'A'; /\* エラー：文字で配列を初期化 \*/

#### 文字列のサイズと長さ

char str[10] = "Hello";  
  
/\* サイズ：配列として確保されたメモリ量 \*/  
sizeof(str) /\* 結果：10（バイト） \*/  
  
/\* 長さ：実際の文字数（'\0'を除く） \*/  
strlen(str) /\* 結果：5（文字） \*/  
  
/\* メモリ配置 \*/  
/\* [H][e][l][l][o][\0][\0][\0][\0][\0] \*/  
/\* 0 1 2 3 4 5 6 7 8 9 \*/

#### 文字列リテラルの特殊性

/\* 文字列リテラルは読み取り専用 \*/  
char \*ptr = "Hello";  
ptr[0] = 'h'; /\* 実行時エラー（セグメンテーションフォルト） \*/  
  
/\* 配列にコピーすれば変更可能 \*/  
char arr[] = "Hello";  
arr[0] = 'h'; /\* OK: "hello"になる \*/  
  
/\* 同じ文字列リテラルは共有される可能性 \*/  
char \*p1 = "Hello";  
char \*p2 = "Hello";  
/\* p1とp2は同じアドレスを指す可能性がある \*/

### 基本的な文字列操作

#### 文字列の長さを取得

#include <string.h>  
char str[] = "Hello";  
int len = strlen(str); /\* 結果: 5 \*/

#### 文字列のコピー

char src[] = "Hello";  
char dest[20];  
strcpy(dest, src); /\* srcをdestにコピー \*/

#### 文字列の比較

char str1[] = "Hello";  
char str2[] = "World";  
int result = strcmp(str1, str2); /\* 0でない値が返される \*/

#### 文字列の連結

char str1[20] = "Hello";  
char str2[] = " World";  
strcat(str1, str2); /\* str1が "Hello World" になる \*/

### 標準ライブラリの文字列関数

#### <string.h>の主要関数

1. **strlen()** - 文字列の長さを取得

* size\_t strlen(const char \*s);

1. **strcpy()** - 文字列をコピー

* char \*strcpy(char \*dest, const char \*src);

1. **strncpy()** - 指定長でコピー（安全）

* char \*strncpy(char \*dest, const char \*src, size\_t n);

1. **strcmp()** - 文字列を比較

* int strcmp(const char \*s1, const char \*s2);

1. **strncmp()** - 指定長で比較

* int strncmp(const char \*s1, const char \*s2, size\_t n);

1. **strcat()** - 文字列を連結

* char \*strcat(char \*dest, const char \*src);

1. **strncat()** - 指定長で連結（安全）

* char \*strncat(char \*dest, const char \*src, size\_t n);

1. **strchr()** - 文字を検索

* char \*strchr(const char \*s, int c);

1. **strstr()** - 文字列を検索

* char \*strstr(const char \*haystack, const char \*needle);

### 文字列配列の詳細操作

#### 2次元文字配列（char配列）

char names[5][20]; /\* 5つの名前、各最大20文字 \*/  
char fruits[5][20] = {  
 "Apple",  
 "Banana",   
 "Orange",  
 "Grape",  
 "Strawberry"  
};

**特徴:** - 固定サイズのメモリ領域を確保 - 各文字列は固定長で格納される - メモリ使用量が予測可能 - 文字列の変更が可能 - サイズ = 行数 × 列数 × sizeof(char)

#### 文字列ポインタ配列（char\*配列）

char \*fruits[] = {"Apple", "Banana", "Orange"};  
char \*months[] = {  
 "January", "February", "March", "April",  
 "May", "June", "July", "August",  
 "September", "October", "November", "December"  
};

**特徴:** - 文字列リテラルへのポインタを格納 - メモリ効率が良い（無駄な領域なし） - 文字列の長さは可変 - 文字列リテラルの変更は不可 - サイズ = ポインタ数 × sizeof(char\*)

#### char配列 vs char\*配列の使い分け

**char配列を選ぶべき場合:** - 文字列を変更する必要がある - メモリ使用量を予測したい - 全ての文字列が同程度の長さ

\*\*char\*配列を選ぶべき場合:\*\* - 文字列を変更しない（読み取り専用） - メモリ効率を重視する - 文字列の長さがバラバラ

#### 文字列配列の操作例

##### 配列の検索

int search\_string(char arr[][20], int size, char \*target) {  
 int i;  
 for (i = 0; i < size; i++) {  
 if (strcmp(arr[i], target) == 0) {  
 return i; /\* 見つかった位置 \*/  
 }  
 }  
 return -1; /\* 見つからなかった \*/  
}

##### 文字列配列のソート（選択ソート）

void sort\_strings(char arr[][20], int size) {  
 int i, j, min\_idx;  
 char temp[20];  
   
 for (i = 0; i < size - 1; i++) {  
 min\_idx = i;  
 for (j = i + 1; j < size; j++) {  
 if (strcmp(arr[j], arr[min\_idx]) < 0) {  
 min\_idx = j;  
 }  
 }  
 if (min\_idx != i) {  
 strcpy(temp, arr[i]);  
 strcpy(arr[i], arr[min\_idx]);  
 strcpy(arr[min\_idx], temp);  
 }  
 }  
}

### 文字列の検索と置換

#### 文字の検索

* **strchr()** - 最初に見つかった文字の位置
* **strrchr()** - 最後に見つかった文字の位置

#### 文字列の検索

* **strstr()** - 部分文字列の検索
* **大文字小文字を無視した検索**は標準関数では提供されていない

#### 文字列の置換

C標準ライブラリには置換関数はないため、自分で実装する必要があります。

### 文字列の変換

#### 大文字・小文字変換

#include <ctype.h>  
char c = 'a';  
char upper = toupper(c); /\* 'A' \*/  
char lower = tolower('B'); /\* 'b' \*/

#### 数値との変換

#include <stdlib.h>  
int num = atoi("123"); /\* 文字列から整数へ \*/  
double d = atof("3.14"); /\* 文字列から浮動小数点へ \*/

### 文字列の入力と出力

#### 標準入力からの文字列読み込み

char buffer[100];  
printf("名前を入力: ");  
scanf("%s", buffer); /\* 空白まで読み込み \*/  
fgets(buffer, sizeof(buffer), stdin); /\* 改行まで読み込み（推奨） \*/

#### 安全な文字列入力

* **scanf()** - 空白で区切られる、バッファオーバーフローの危険
* **fgets()** - 改行まで読み込み、サイズ制限あり（推奨）

### 文字列処理のベストプラクティス

#### 安全性の確保

1. **バッファオーバーフローの防止**
   * strncpy(), strncat(), snprintf()を使用
   * 常にバッファサイズを意識
2. **null終端の確保**
   * strncpy()後は明示的にnull文字を設定
   * メモリ初期化を適切に行う
3. **戻り値のチェック**
   * 文字列関数の戻り値を適切にチェック
   * NULLポインタの可能性を考慮

#### 効率性の向上

1. **適切なデータ構造の選択**
   * 固定長vs可変長の選択
   * メモリ使用量の最適化
2. **文字列操作の最小化**
   * 不必要なコピーを避ける
   * インプレース操作の活用

### よくある間違いとデバッグ

#### 1. バッファオーバーフロー

**問題:** 文字列が配列のサイズを超える

char buffer[5];  
strcpy(buffer, "Hello World"); /\* 危険！ \*/

**対策:** - strncpy()やsnprintf()を使用 - 常にバッファサイズを確認

#### 2. null終端の忘れ

**問題:** null文字がない文字列

char str[5] = {'H', 'e', 'l', 'l', 'o'}; /\* null終端なし \*/  
printf("%s", str); /\* 未定義動作 \*/

**対策:** - 文字列リテラルを使用 - 明示的にnull文字を設定

#### 3. 文字列リテラルの変更

**問題:** 読み取り専用の文字列を変更

char \*str = "Hello";  
str[0] = 'h'; /\* 未定義動作 \*/

**対策:** - 変更が必要な場合は配列を使用 - const修飾子を適切に使用

#### 4. ポインタの誤用

**問題:** 無効なポインタの使用

char \*str;  
strcpy(str, "Hello"); /\* 未初期化ポインタ \*/

**対策:** - ポインタを適切に初期化 - メモリ確保を忘れずに

## 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。

### 基本的な文字列操作

* [string\_basics.c](examples/string_basics.c) - C90準拠版
* [string\_basics\_c99.c](examples/string_basics_c99.c) - C99準拠版

### 文字列操作関数の活用

* [string\_functions.c](examples/string_functions.c) - C90準拠版
* [string\_functions\_c99.c](examples/string_functions_c99.c) - C99準拠版

### 文字列配列の実践と詳細操作

* [string\_array\_operations.c](examples/string_array_operations.c) - C90準拠版
* [string\_array\_operations\_c99.c](examples/string_array_operations_c99.c) - C99準拠版

### 文字列配列の完全活用例

* [string\_arrays.c](examples/string_arrays.c) - C90準拠版（配列章から移行）
* [string\_arrays\_c99.c](examples/string_arrays_c99.c) - C99準拠版（配列章から移行）

## 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 string\_basics.c -o string\_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 string\_basics  
# C99版をコンパイル  
make string\_basics\_c99  
# プログラムを実行  
make run-all  
# クリーンアップ  
make clean

## 学習フローとコンパイル方法

### 推奨学習順序

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

### 実践的な学習方法

1. **基本関数の理解** - strlen, strcpy, strcmp等の基本動作を確認
2. **安全な関数の活用** - strncpy, strncat等の安全版を使用
3. **実際の問題解決** - 具体的な文字列処理問題に取り組む
4. **エラー処理の実装** - 例外的なケースへの対応を学習

## C90とC99の違い

### 文字列処理での主な違い

* **C90**: 変数宣言は関数の先頭でまとめて行う
* **C99**: 変数を使用する場所で宣言可能
* **コメント**: C90は /\* \*/のみ、C99は // も使用可能
* **新しい関数**: C99以降で追加された関数の有無

## 注意事項とベストプラクティス

### セキュリティの考慮

1. **入力検証の徹底**
   * ユーザー入力は常に検証
   * バッファサイズを超える入力の制御
2. **安全な関数の使用**
   * 古い関数（strcpy, strcat等）より安全版を選択
   * 可能な限りstrncpy, strncat, snprintf等を使用
3. **メモリ管理**
   * 動的に確保したメモリの適切な解放
   * ダングリングポインタの回避

## 次の章へ

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

## 参考資料

* [C標準ライブラリリファレンス](https://en.cppreference.com/w/c/string/byte)
* [セキュアコーディング標準](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177)

# 関数

## 対応C規格

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

## 学習目標

この章を完了すると、以下のことができるようになります。 - 関数の定義と呼び出しができる - 引数と戻り値を適切に使える - 関数のスコープを理解する - 再帰関数を作成できる - 関数プロトタイプの重要性を理解する

## 概要と詳細

### 関数とは？

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

#### 日常生活での関数

関数を理解するために、料理を例に考えてみましょう。

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

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

### 関数の基本概念

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

#### 関数を使う理由

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

#### 関数の構成要素

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

### 関数の定義と宣言

関数を使うには、「宣言」と「定義」を理解する必要があります。

#### なぜ関数プロトタイプが必要？

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

#### 関数プロトタイプ（前方宣言）

/\* 関数プロトタイプ \*/  
int add(int a, int b);  
void print\_message(void);

#### 関数の定義

/\* 関数の実装 \*/  
int add(int a, int b)  
{  
 return a + b;  
}  
  
void print\_message(void)  
{  
 printf("Hello, World!\n");  
}

### 引数の渡し方

関数に値を渡す方法は2つあります。この違いを理解することは非常に重要です！

#### 値渡し（Call by Value）

C言語では、通常の変数を渡すと「値のコピー」が渡されます。これはC言語の基本的な動作です。

##### 値渡しのメカニズム

関数呼び出し時のメモリの様子：  
  
main関数のメモリ: modify\_value関数のメモリ:  
+---------------+ +---------------+  
| num = 10 | ---値をコピー--> | x = 10 |  
+---------------+ +---------------+  
 ↓ x = 100に変更  
+---------------+ +---------------+  
| num = 10 | | x = 100 |  
+---------------+ +---------------+  
※numは変更されない

##### 値渡しの実例

void modify\_value(int x)  
{  
 printf("関数内（変更前）: x = %d\n", x);  
 x = 100; /\* 元の変数には影響しない \*/  
 printf("関数内（変更後）: x = %d\n", x);  
}  
  
int main(void)  
{  
 int num = 10;  
 printf("呼び出し前: num = %d\n", num);  
   
 modify\_value(num);  
   
 printf("呼び出し後: num = %d\n", num); /\* 10のまま \*/  
 return 0;  
}

##### 値渡しの利点と欠点

**利点：** - 元の変数が保護される（予期しない変更を防げる） - 理解しやすく、バグが起きにくい - 関数の独立性が高い

**欠点：** - 大きなデータ構造では非効率（コピーに時間がかかる） - 複数の値を返すことができない - 関数内で元の変数を変更できない

#### 参照渡し（Call by Reference）

ポインタを使用して、変数のアドレスを渡すことで元の変数を変更できます。

##### 参照渡しのメカニズム

関数呼び出し時のメモリの様子：  
  
main関数のメモリ: modify\_value関数のメモリ:  
+---------------+ +------------------+  
| num = 10 |<--アドレスを指す-- | x = &num |  
| アドレス:1000 | +------------------+  
+---------------+ ↓ \*x = 100でnumを変更  
   
+---------------+   
| num = 100 |   
| アドレス:1000 |   
+---------------+   
※numが直接変更される

##### 参照渡しの実例

void modify\_value(int \*x)  
{  
 printf("関数内（変更前）: \*x = %d\n", \*x);  
 \*x = 100; /\* 元の変数を変更 \*/  
 printf("関数内（変更後）: \*x = %d\n", \*x);  
}  
  
int main(void)  
{  
 int num = 10;  
 printf("呼び出し前: num = %d\n", num);  
   
 modify\_value(&num); /\* &でアドレスを渡す \*/  
   
 printf("呼び出し後: num = %d\n", num); /\* 100に変更されている \*/  
 return 0;  
}

##### 参照渡しの利点と欠点

**利点：** - 関数内で元の変数を変更できる - 大きなデータ構造でも効率的（アドレスだけ渡す） - 複数の値を返すことができる

**欠点：** - ポインタの理解が必要 - 意図しない変更の可能性がある - NULLポインタのチェックが必要

#### 配列の引数渡し

配列を関数に渡す場合、特別な動作をします。

##### 配列は常に「参照渡し」になる

void modify\_array(int arr[], int size)  
{  
 int i;  
 for (i = 0; i < size; i++) {  
 arr[i] \*= 2; /\* 元の配列を変更 \*/  
 }  
}  
  
int main(void)  
{  
 int numbers[5] = {1, 2, 3, 4, 5};  
 int i;  
   
 printf("変更前: ");  
 for (i = 0; i < 5; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 modify\_array(numbers, 5);  
   
 printf("変更後: ");  
 for (i = 0; i < 5; i++) {  
 printf("%d ", numbers[i]); /\* 値が2倍になっている \*/  
 }  
 printf("\n");  
   
 return 0;  
}

##### なぜ配列は参照渡しになるのか？

配列名は配列の先頭要素へのポインタとして扱われる：  
  
numbers[5] = {1, 2, 3, 4, 5}  
  
numbers → [1][2][3][4][5]  
 ↑  
 先頭要素のアドレス  
  
関数に渡されるのは、この先頭要素のアドレス

#### 構造体の引数渡し

構造体は値渡しと参照渡しの両方が可能です。

##### 構造体の値渡し

struct Point {  
 int x;  
 int y;  
};  
  
void move\_point(struct Point p)  
{  
 p.x += 10; /\* コピーを変更（元の構造体は変わらない） \*/  
 p.y += 10;  
}

##### 構造体の参照渡し

void move\_point\_ref(struct Point \*p)  
{  
 p->x += 10; /\* 元の構造体を変更 \*/  
 p->y += 10;  
}

#### 引数渡しの実践的な使い分け

プログラミングでは、状況に応じて値渡しと参照渡しを使い分けることが重要です。ここでは、実際によく使われるパターンを詳しく見ていきましょう。

##### 1. swap関数（値の交換）

2つの変数の値を交換する関数は、参照渡しの必要性を理解する最も分かりやすい例です。

/\* 値渡しでは交換できない（間違い例） \*/  
void swap\_wrong(int a, int b)  
{  
 int temp = a;  
 a = b;  
 b = temp;  
 /\* 関数内でコピーを交換しただけ \*/  
}  
  
/\* 参照渡しで正しく交換 \*/  
void swap\_correct(int \*a, int \*b)  
{  
 int temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
}  
  
int main(void)  
{  
 int x = 5, y = 10;  
   
 printf("交換前: x=%d, y=%d\n", x, y);  
   
 swap\_wrong(x, y);  
 printf("swap\_wrong後: x=%d, y=%d\n", x, y); /\* 5, 10のまま \*/  
   
 swap\_correct(&x, &y);  
 printf("swap\_correct後: x=%d, y=%d\n", x, y); /\* 10, 5に交換 \*/  
   
 return 0;  
}

**なぜswap\_wrongは動作しないのか？**

swap\_wrong呼び出し時：  
main: x=5, y=10 →コピー→ swap\_wrong: a=5, b=10  
 ↓交換  
main: x=5, y=10 ←影響なし swap\_wrong: a=10, b=5（関数終了で消滅）

##### 2. 複数の値を返す関数

C言語の関数は通常1つの値しか返せませんが、参照渡しを使うと複数の値を返すことができます。

/\* 商と余りを同時に返す \*/  
void divide\_with\_remainder(int dividend, int divisor,   
 int \*quotient, int \*remainder)  
{  
 if (divisor == 0) {  
 /\* エラー処理 \*/  
 if (quotient) \*quotient = 0;  
 if (remainder) \*remainder = 0;  
 return;  
 }  
   
 \*quotient = dividend / divisor;  
 \*remainder = dividend % divisor;  
}  
  
/\* 統計情報を一度に計算する \*/  
void calculate\_stats(int arr[], int size,   
 int \*min, int \*max, double \*average)  
{  
 int i, sum = 0;  
   
 if (size <= 0 || arr == NULL) {  
 return; /\* エラー処理 \*/  
 }  
   
 \*min = \*max = arr[0];  
   
 for (i = 0; i < size; i++) {  
 if (arr[i] < \*min) \*min = arr[i];  
 if (arr[i] > \*max) \*max = arr[i];  
 sum += arr[i];  
 }  
   
 \*average = (double)sum / size;  
}  
  
int main(void)  
{  
 int q, r;  
 int numbers[] = {23, 67, 12, 89, 45};  
 int minimum, maximum;  
 double avg;  
   
 /\* 除算の例 \*/  
 divide\_with\_remainder(17, 5, &q, &r);  
 printf("17 ÷ 5 = %d 余り %d\n", q, r); /\* 3 余り 2 \*/  
   
 /\* 統計計算の例 \*/  
 calculate\_stats(numbers, 5, &minimum, &maximum, &avg);  
 printf("最小値: %d, 最大値: %d, 平均: %.1f\n",   
 minimum, maximum, avg);  
   
 return 0;  
}

##### 3. 安全な参照渡し（エラー処理）

ポインタを受け取る関数では、必ずNULLチェックを行うべきです。

/\* 基本的なNULLチェック \*/  
void safe\_increment(int \*ptr)  
{  
 if (ptr != NULL) { /\* NULLポインタチェック \*/  
 (\*ptr)++;  
 }  
}  
  
/\* より実践的な例：文字列の安全なコピー \*/  
int safe\_string\_copy(char \*dest, const char \*src, int dest\_size)  
{  
 int i;  
   
 /\* 引数の検証 \*/  
 if (dest == NULL || src == NULL || dest\_size <= 0) {  
 return -1; /\* エラー \*/  
 }  
   
 /\* 安全にコピー \*/  
 for (i = 0; i < dest\_size - 1 && src[i] != '\0'; i++) {  
 dest[i] = src[i];  
 }  
 dest[i] = '\0'; /\* null終端を保証 \*/  
   
 return i; /\* コピーした文字数を返す \*/  
}  
  
/\* エラーコードを返す関数の例 \*/  
int safe\_divide(int a, int b, int \*result)  
{  
 if (result == NULL) {  
 return -1; /\* 引数エラー \*/  
 }  
   
 if (b == 0) {  
 return -2; /\* ゼロ除算エラー \*/  
 }  
   
 \*result = a / b;  
 return 0; /\* 成功 \*/  
}

##### 4. 大きなデータ構造の効率的な処理

構造体が大きくなると、値渡しのコピーコストが無視できなくなります。

/\* 大きな構造体の例 \*/  
struct StudentRecord {  
 char name[50];  
 char id[20];  
 int scores[10];  
 double gpa;  
 char address[100];  
};  
  
/\* 値渡し（非効率） - 約200バイトをコピー \*/  
void print\_student\_by\_value(struct StudentRecord student)  
{  
 printf("名前: %s\n", student.name);  
 printf("GPA: %.2f\n", student.gpa);  
}  
  
/\* 参照渡し（効率的） - ポインタ（4-8バイト）のみコピー \*/  
void print\_student\_by\_reference(const struct StudentRecord \*student)  
{  
 if (student == NULL) return;  
   
 printf("名前: %s\n", student->name);  
 printf("GPA: %.2f\n", student->gpa);  
}  
  
/\* 構造体を変更する関数 \*/  
void update\_gpa(struct StudentRecord \*student, double new\_gpa)  
{  
 if (student == NULL) return;  
   
 student->gpa = new\_gpa;  
 printf("%sのGPAを%.2fに更新しました\n",   
 student->name, new\_gpa);  
}

##### 5. 配列操作の実践例

配列は自動的に参照渡しになりますが、安全に扱うためのテクニックがあります。

/\* 配列の要素をすべて特定の値に設定 \*/  
void fill\_array(int arr[], int size, int value)  
{  
 int i;  
   
 if (arr == NULL || size <= 0) return;  
   
 for (i = 0; i < size; i++) {  
 arr[i] = value;  
 }  
}  
  
/\* 2つの配列を比較 \*/  
int compare\_arrays(const int arr1[], const int arr2[], int size)  
{  
 int i;  
   
 if (arr1 == NULL || arr2 == NULL || size <= 0) {  
 return -1; /\* エラー \*/  
 }  
   
 for (i = 0; i < size; i++) {  
 if (arr1[i] != arr2[i]) {  
 return 0; /\* 異なる \*/  
 }  
 }  
   
 return 1; /\* 同じ \*/  
}  
  
/\* 配列の一部を別の配列にコピー \*/  
void copy\_array\_range(int dest[], const int src[],   
 int start, int count, int dest\_size)  
{  
 int i;  
   
 if (dest == NULL || src == NULL) return;  
 if (start < 0 || count <= 0) return;  
 if (count > dest\_size) count = dest\_size; /\* 安全性確保 \*/  
   
 for (i = 0; i < count; i++) {  
 dest[i] = src[start + i];  
 }  
}

##### 6. 関数の戻り値と参照渡しの組み合わせ

戻り値でステータスを返し、参照渡しで実際の値を返すパターンはよく使われます。

/\* 文字列を整数に変換（エラーチェック付き） \*/  
int string\_to\_int(const char \*str, int \*result)  
{  
 int value = 0;  
 int sign = 1;  
 int i = 0;  
   
 if (str == NULL || result == NULL) {  
 return -1; /\* 引数エラー \*/  
 }  
   
 /\* 空白をスキップ \*/  
 while (str[i] == ' ') i++;  
   
 /\* 符号の処理 \*/  
 if (str[i] == '-') {  
 sign = -1;  
 i++;  
 } else if (str[i] == '+') {  
 i++;  
 }  
   
 /\* 数字でない場合はエラー \*/  
 if (str[i] < '0' || str[i] > '9') {  
 return -2; /\* フォーマットエラー \*/  
 }  
   
 /\* 数値に変換 \*/  
 while (str[i] >= '0' && str[i] <= '9') {  
 value = value \* 10 + (str[i] - '0');  
 i++;  
 }  
   
 \*result = value \* sign;  
 return 0; /\* 成功 \*/  
}  
  
/\* 使用例 \*/  
int main(void)  
{  
 int num;  
 int status;  
   
 status = string\_to\_int("123", &num);  
 if (status == 0) {  
 printf("変換成功: %d\n", num);  
 } else {  
 printf("変換失敗: エラーコード %d\n", status);  
 }  
   
 return 0;  
}

##### 7. 実践的な選択基準

**値渡しを選ぶべき場合：** - 基本データ型（int, char, float, double） - 小さな構造体（数十バイト以下） - 元の値を保護したい場合 - 関数の独立性を高めたい場合

**参照渡しを選ぶべき場合：** - 大きな構造体や配列 - 複数の値を返したい場合 - 元の値を変更する必要がある場合 - パフォーマンスが重要な場合

**constを使った参照渡しを選ぶべき場合：** - 大きなデータを読むだけの場合 - 誤った変更を防ぎたい場合 - APIの意図を明確にしたい場合

#### const修飾子による保護

参照渡しでも、値を変更したくない場合はconstを使います。

/\* 配列の内容を変更しない関数 \*/  
int array\_sum(const int arr[], int size)  
{  
 int i, sum = 0;  
   
 /\* arr[i] = 0; コンパイルエラー（constで保護） \*/  
   
 for (i = 0; i < size; i++) {  
 sum += arr[i]; /\* 読み取りはOK \*/  
 }  
   
 return sum;  
}

#### 引数渡しのベストプラクティス

1. **基本データ型（int, char, floatなど）**
   * 通常は値渡しを使用
   * 変更が必要な場合のみ参照渡し
2. **配列**
   * 自動的に参照渡しになる
   * 変更しない場合はconstを付ける
3. **構造体**
   * 小さい構造体：値渡しでOK
   * 大きい構造体：参照渡しで効率化
   * 変更しない場合：const付き参照渡し
4. **文字列**
   * char配列またはchar\*で渡す
   * 変更しない場合はconstを付ける

### 様々な関数の種類

関数にはいろいろな種類があります。用途に応じて使い分けましょう。

#### 戻り値のない関数（void関数）

処理だけ行って、結果を返さない関数です。

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

#### 配列を扱う関数

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

### 再帰関数

再帰関数は自分自身を呼び出す関数です。

#### 再帰を理解する日常例

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

これが再帰の考え方です！

#### 再帰関数の例

/\* 階乗を計算する再帰関数 \*/  
int factorial(int n)  
{  
 if (n <= 1) {  
 return 1; /\* 基底条件 \*/  
 }  
 return n \* factorial(n - 1); /\* 再帰呼び出し \*/  
}  
  
/\* フィボナッチ数列を計算する再帰関数 \*/  
int fibonacci(int n)  
{  
 if (n <= 0) {  
 return 0;  
 }  
 if (n == 1) {  
 return 1;  
 }  
 return fibonacci(n - 1) + fibonacci(n - 2);  
}

#### 再帰関数の構成要素

1. **基底条件** - 再帰を終了する条件
2. **再帰呼び出し** - 自分自身を呼び出す部分
3. **問題の分割** - 元の問題をより小さな問題に分割

### 関数のスコープと生存期間

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

#### ローカル変数とグローバル変数

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

この違いを理解することは、バグの少ない、保守しやすいプログラムを書くために非常に重要です。

#### ローカル変数の詳細

ローカル変数は関数内で宣言され、その関数内でのみ使用できる変数です。

##### ローカル変数の特徴

void function\_example(void)  
{  
 int local\_var = 10; /\* この関数内でのみ有効 \*/  
   
 { /\* ブロックスコープ \*/  
 int block\_var = 20; /\* このブロック内でのみ有効 \*/  
 printf("block\_var = %d\n", block\_var);  
 } /\* block\_varはここで破棄される \*/  
   
 /\* printf("%d\n", block\_var); エラー：block\_varは見えない \*/  
   
} /\* local\_varはここで破棄される \*/

##### ローカル変数のメモリ配置

スタックメモリの様子：  
  
関数呼び出し時: 関数終了時:  
+----------------+ +----------------+  
| main()の変数 | | main()の変数 |  
+----------------+ +----------------+  
| function()の | | （空） |  
| local\_var = 10 | | |  
+----------------+ +----------------+

##### 同名のローカル変数

void func1(void)  
{  
 int count = 0; /\* func1のcount \*/  
 count++;  
 printf("func1: count = %d\n", count); /\* 1 \*/  
}  
  
void func2(void)  
{  
 int count = 100; /\* func2のcount（func1とは別物） \*/  
 count++;  
 printf("func2: count = %d\n", count); /\* 101 \*/  
}  
  
int main(void)  
{  
 int count = 50; /\* mainのcount \*/  
   
 func1(); /\* func1のcountを使用 \*/  
 func2(); /\* func2のcountを使用 \*/  
   
 printf("main: count = %d\n", count); /\* 50（変更されない） \*/  
 return 0;  
}

#### グローバル変数の詳細

グローバル変数は関数の外で宣言され、プログラム全体で使用できる変数です。

##### グローバル変数の宣言と使用

/\* グローバル変数の宣言 \*/  
int global\_count = 0; /\* 初期化あり \*/  
double global\_rate; /\* 初期化なし（0.0になる） \*/  
char global\_message[100]; /\* 初期化なし（全て0になる） \*/  
  
void increment\_counter(void)  
{  
 global\_count++; /\* どの関数からでもアクセス可能 \*/  
}  
  
void print\_counter(void)  
{  
 printf("カウンター: %d\n", global\_count);  
}  
  
int main(void)  
{  
 print\_counter(); /\* カウンター: 0 \*/  
 increment\_counter();  
 increment\_counter();  
 print\_counter(); /\* カウンター: 2 \*/  
   
 global\_count = 10; /\* main関数からも変更可能 \*/  
 print\_counter(); /\* カウンター: 10 \*/  
   
 return 0;  
}

##### グローバル変数の初期化

/\* グローバル変数は自動的に0で初期化される \*/  
int g\_int; /\* 0 \*/  
double g\_double; /\* 0.0 \*/  
char g\_char; /\* '\0' \*/  
int g\_array[10]; /\* すべて0 \*/  
  
/\* 明示的な初期化も可能 \*/  
int g\_initialized = 100;  
char g\_string[] = "Hello, Global!";  
  
/\* 関数呼び出しによる初期化はできない \*/  
/\* int g\_value = get\_value(); エラー！ \*/

#### 変数の可視性（スコープ）と名前の隠蔽

同じ名前の変数が複数ある場合、最も内側のスコープの変数が優先されます。

int value = 100; /\* グローバル変数 \*/  
  
void test\_scope(void)  
{  
 printf("1: value = %d\n", value); /\* 100（グローバル） \*/  
   
 int value = 200; /\* ローカル変数（グローバルを隠蔽） \*/  
 printf("2: value = %d\n", value); /\* 200（ローカル） \*/  
   
 {  
 int value = 300; /\* ブロック内変数（ローカルを隠蔽） \*/  
 printf("3: value = %d\n", value); /\* 300（ブロック内） \*/  
 }  
   
 printf("4: value = %d\n", value); /\* 200（ローカル） \*/  
}  
  
int main(void)  
{  
 test\_scope();  
 printf("5: value = %d\n", value); /\* 100（グローバル） \*/  
 return 0;  
}

#### グローバル変数の問題点と対策

##### 1. 予期しない変更

int total\_score = 0; /\* グローバル変数 \*/  
  
void add\_score(int points)  
{  
 total\_score += points;  
}  
  
void reset\_score(void)  
{  
 total\_score = 0; /\* 他の場所で使用中かもしれない！ \*/  
}  
  
/\* より良い設計：引数と戻り値を使用 \*/  
int add\_score\_safe(int current\_score, int points)  
{  
 return current\_score + points;  
}

##### 2. デバッグの困難さ

/\* 悪い例：グローバル変数を多用 \*/  
int g\_width, g\_height, g\_area;  
  
void calculate\_area(void)  
{  
 g\_area = g\_width \* g\_height;  
}  
  
/\* 良い例：引数と戻り値を使用 \*/  
int calculate\_area\_safe(int width, int height)  
{  
 return width \* height;  
}

##### 3. 名前の衝突

/\* file1.c \*/  
int counter = 0; /\* グローバル変数 \*/  
  
/\* file2.c \*/  
int counter = 0; /\* 同じ名前！リンクエラーになる \*/  
  
/\* 対策：staticを使用してファイルスコープに限定 \*/  
static int counter = 0; /\* このファイル内でのみ有効 \*/

#### ローカル変数とグローバル変数の使い分け

##### ローカル変数を使うべき場合（推奨）

/\* 関数内で完結する処理 \*/  
double calculate\_average(int arr[], int size)  
{  
 int i;  
 int sum = 0; /\* ローカル変数 \*/  
   
 for (i = 0; i < size; i++) {  
 sum += arr[i];  
 }  
   
 return (double)sum / size;  
}  
  
/\* 一時的な計算 \*/  
void print\_multiplication\_table(int n)  
{  
 int i, j; /\* ループカウンタはローカル変数 \*/  
   
 for (i = 1; i <= n; i++) {  
 for (j = 1; j <= n; j++) {  
 printf("%4d", i \* j);  
 }  
 printf("\n");  
 }  
}

##### グローバル変数が適切な場合（限定的）

/\* 設定値やフラグ \*/  
int debug\_mode = 0; /\* デバッグモードのフラグ \*/  
const double PI = 3.14159265359; /\* 定数 \*/  
  
/\* プログラム全体の状態 \*/  
char program\_name[256] = "MyApplication";  
int error\_count = 0;  
  
void enable\_debug(void)  
{  
 debug\_mode = 1;  
}  
  
void log\_message(const char \*msg)  
{  
 if (debug\_mode) {  
 printf("[DEBUG] %s\n", msg);  
 }  
}

#### グローバル変数の安全な使用方法

##### 1. constを使用して読み取り専用にする

const int MAX\_USERS = 100;  
const double TAX\_RATE = 0.08;  
const char VERSION[] = "1.0.0";

##### 2. staticでファイルスコープに限定

/\* このファイル内でのみ使用可能 \*/  
static int file\_local\_counter = 0;  
  
static void increment\_counter(void)  
{  
 file\_local\_counter++;  
}

##### 3. アクセス関数を提供する

/\* グローバル変数を直接公開しない \*/  
static int score = 0; /\* ファイルスコープ \*/  
  
/\* アクセス関数を通じて操作 \*/  
int get\_score(void)  
{  
 return score;  
}  
  
void set\_score(int new\_score)  
{  
 if (new\_score >= 0) { /\* 検証も可能 \*/  
 score = new\_score;  
 }  
}  
  
void add\_score(int points)  
{  
 if (points > 0) {  
 score += points;  
 }  
}

#### まとめ：変数スコープのベストプラクティス

1. **可能な限りローカル変数を使用する**
   * バグが少なく、理解しやすいコードになる
2. **グローバル変数は最小限に**
   * 本当に必要な場合のみ使用
   * constやstaticで保護する
3. **明確な命名規則**
   * グローバル変数にはプレフィックスを付ける（例：g\_count）
   * 意味のある名前を使用する
4. **初期化を忘れない**
   * ローカル変数は必ず初期化する
   * グローバル変数の初期値も明示する

#### static変数

void counter\_function(void)  
{  
 static int count = 0; /\* 初回のみ初期化 \*/  
 count++;  
 printf("呼び出し回数: %d\n", count);  
}

### 関数ポインタの基礎

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

#### 関数ポインタのイメージ

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

#### 基本的な関数ポインタ

/\* 関数の定義 \*/  
int add(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);  
   
 /\* 関数ポインタに関数を代入 \*/  
 operation = add;  
 printf("加算: %d\n", operation(5, 3));  
   
 operation = multiply;  
 printf("乗算: %d\n", operation(5, 3));  
   
 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

## 注意事項

初心者が関数で間違えやすいポイント。

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

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

1. **配列の扱い**: 配列を関数に渡すときは、サイズ情報も一緒に渡す

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

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

## 次のステップ

関数の基本を理解したら、以下のトピックに進みましょう。

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

## 学習フローとコンパイル方法

### 推奨学習順序

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

### Makefileを使用したコンパイル

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

## 次の章へ

[構造体とポインタ](../structures/README.md)

## 参考資料

* [C90規格書](https://www.iso.org/standard/17782.html)
* [C99規格書](https://www.iso.org/standard/29237.html)

# 終わりに

応用編の学習お疲れさまでした。ここまでで、C言語の中核となる概念を学びました。

## 学習の成果

応用編を通じて、以下のスキルを習得しました：

* 配列を使った効率的なデータ管理
* ポインタによるメモリの直接操作
* 文字列の柔軟な処理
* 関数による構造化プログラミング

## 次のステップ

応用編を修了したら、上級編に進んでください。上級編では以下の内容を学習します：

* ビット操作とビットフィールド
* 構造体とポインタ
* 関数ポインタ
* 複数ファイルプログラミング

これらの技術を習得することで、より大規模で実践的なプログラムを作成できるようになります。