# 第10章 ポインタ基礎 - 演習問題

# 演習問題

## 基礎問題

### 演習10-1: ポインタの基本操作

2つの整数変数の値をポインタを使って交換するプログラムを作成してください。

**要件:** - 2つの整数変数を宣言・初期化 - ポインタを使った値の交換関数を実装 - 交換前後の値とアドレスを表示 - 直接的な値の交換とポインタ経由の交換を比較

**期待される出力例:**

交換前: a = (アドレス: xffffbffac), b = (アドレス: xffffbffa)  
ポインタを使った交換実行...  
交換後: a = (アドレス: xffffbffac), b = (アドレス: xffffbffa)

### 演習10-2: 配列とポインタ

ポインタ演算を使って配列の要素を逆順に表示するプログラムを作成してください。

**要件:** - 整数配列を宣言・初期化 - ポインタ演算で配列の最後から最初に向かってアクセス - インデックス記法とポインタ記法の両方で表示 - 配列のサイズを動的に計算

### 演習10-3: 文字列操作

ポインタを使って文字列の長さを計算し、文字列を逆順にするプログラムを作成してください。

**要件:** - 文字列の長さ計算関数（strlen相当）をポインタで実装 - 文字列を逆順にする関数をポインタで実装 - 元の文字列と逆順文字列を表示 - 文字列リテラルと文字配列の違いを考慮

## 応用問題

### 演習10-4: 配列操作関数

ポインタを使って配列の最大値、最小値、平均値を計算する関数群を作成してください。

**要件:** - 最大値を見つけてそのポインタを返す関数 - 最小値を見つけてそのポインタを返す関数 - 平均値を計算する関数（戻り値はdouble） - 各関数の結果を使ってレポートを作成

**実装する関数:**

int\* find\_max(int \*arr, int size);  
int\* find\_min(int \*arr, int size);  
double calculate\_average(int \*arr, int size);  
void print\_statistics(int \*arr, int size);

### 演習10-5: 文字列処理

ポインタを使って文字列の検索、置換、分割を実行する関数群を作成してください。

**要件:** - 文字列内で特定の文字を検索する関数 - 文字列内の文字を置換する関数 - 文字列を特定の文字で分割する関数 - 大文字・小文字を無視した比較機能

### 演習10-6: データ変換

ポインタを使って配列のデータ型変換を実行するプログラムを作成してください。

**要件:** - int配列をfloat配列に変換 - 文字列を数値配列に変換 - バイト配列を整数として解釈 - エラーハンドリングの実装

## 発展問題

### 演習10-7: メモリ操作

ポインタを使って任意のデータ型の配列をコピーする汎用関数を作成してください。

**要件:** - void\*ポインタを使った汎用コピー関数 - バイト単位でのメモリコピー - 型安全性の考慮 - コピー対象の境界チェック

**関数プロトタイプ:**

void\* generic\_copy(void \*dest, const void \*src, size\_t size);  
int compare\_memory(const void \*ptr, const void \*ptr, size\_t size);

### 演習10-8: アルゴリズム実装

ポインタを使って各種ソートアルゴリズムを実装してください。

**要件:** - バブルソート（ポインタ版） - 選択ソート（ポインタ版） - 挿入ソート（ポインタ版） - 汎用的な比較関数の使用

### 演習10-9: データ構造

ポインタを使って簡単なリンクリスト構造を実装してください。

**要件:** - ノード構造体の定義 - リストへの要素追加・削除 - リストの走査・検索 - メモリリークの防止

**構造体例:**

typedef struct Node {  
 int data;  
 struct Node \*next;  
} Node;

## チャレンジ問題

### 演習10-10: 動的配列シミュレータ

malloc/freeを使わずに、静的配列とポインタを使って動的配列のような動作を実現してください。

**要件:** - 固定サイズの大きな配列をメモリプールとして使用 - 要素の追加・削除・挿入機能 - メモリの断片化管理 - ガベージコレクション機能

## 提出方法

1. 各問題に対して別々のCファイルを作成
2. ファイル名は ex\_pointer\_swap.c, ex\_array\_reverse.c のように命名
3. 各プログラムの冒頭にコメントで問題番号と簡単な説明を記載
4. C90規格でコンパイルできることを確認

## コンパイル例

gcc -std=c90 -Wall -Wextra -pedantic ex\_pointer\_swap.c -o ex\_pointer\_swap  
./ex\_pointer\_swap

## ヒント

### ポインタの基本

* 宣言: int \*ptr;
* アドレス取得: ptr = &variable;
* 値の取得: value = \*ptr;
* 演算: ptr + 1 は次の要素を指す

### 安全なプログラミング

* NULLポインタのチェック: if (ptr != NULL)
* 配列の境界チェック
* 初期化されていないポインタの使用を避ける

### デバッグのコツ

* アドレスと値を分けて表示
* ポインタ演算の結果を段階的に確認
* コンパイラの警告に注意を払う

### よくある間違い

* \*ptr++ と (\*ptr)++ の違い
* ローカル変数のアドレスを返すこと
* 配列の範囲外アクセス
* NULLポインタの参照

# 解答例

この章の演習問題の解答例です。各解答にはC90版とC99版の両方を用意しています。

## 演習問題の解答

### 演習10-1: ポインタの基本操作

2つの整数変数の値をポインタを使って交換する演習です。

* <ex10_1_pointer_swap.c> - C90準拠版
* <ex10_1_pointer_swap_c99.c> - C99準拠版（//コメント、bool型、インライン関数、restrict修飾子）

**学習ポイント:** - ポインタによる参照渡しの仕組み - 値渡しとの違いの理解 - アドレス演算子（&）と間接参照演算子（\*）の使用 - ポインタの安全な使用方法

### 演習10-2: 配列とポインタ

ポインタ演算を使って配列の要素を逆順に表示する演習です。

* <ex10_2_array_reverse.c> - C90準拠版
* <ex10_2_array_reverse_c99.c> - C99準拠版（可変長配列、複合リテラル、指定初期化子）

**学習ポイント:** - 配列とポインタの等価性 - ポインタ演算による配列要素アクセス - メモリレイアウトの理解 - 安全な境界チェック

### 演習10-3: 文字列操作

ポインタを使った文字列の長さ計算、複写、連結を行う演習です。

* <ex10_3_string_manipulation.c> - C90準拠版
* <ex10_3_string_manipulation_c99.c> - C99準拠版（//コメント、restrict修飾子、bool型）

**学習ポイント:** - 文字列とポインタの関係 - 文字列の終端（ヌル文字）の処理 - ポインタを使った文字列操作関数の実装 - バッファオーバーフローの防止

### 演習10-4: 配列操作関数

ポインタを使って配列の最大値、最小値、平均値を求める演習です。

* <ex10_4_array_functions.c> - C90準拠版
* <ex10_4_array_functions_c99.c> - C99準拠版（//コメント、bool型、restrict修飾子）

**学習ポイント:** - 配列をポインタとして関数に渡す方法 - 配列サイズの適切な管理 - ポインタを使った効率的な配列操作 - 関数の戻り値としてのポインタの活用

## C90とC99の主な違い

### 1. コメントスタイル

* C90: /\* \*/ のみ
* C99: // 単一行コメントも使用可能

### 2. 変数宣言

* C90: ブロックの先頭でのみ宣言可能
* C99: 使用箇所で宣言可能（forループ内での宣言など）

### 3. bool型

* C90: 存在しない（intで代用）
* C99: <stdbool.h>でbool、true、falseが使用可能

### 4. インライン関数

* C90: 存在しない
* C99: inlineキーワードで関数のインライン化が可能

### 5. 可変長配列（VLA）

* C90: 存在しない
* C99: 実行時にサイズが決まる配列を作成可能

### 6. 複合リテラル

* C90: 存在しない
* C99: (Type){初期化子}の形式で一時オブジェクトを作成可能

### 7. 指定初期化子

* C90: 存在しない
* C99: {[index] = value}の形式で配列の特定要素を初期化可能

### 8. restrict修飾子

* C90: 存在しない
* C99: ポインタが重複しないことをコンパイラに伝える最適化ヒント

### 9. 固定幅整数型

* C90: 存在しない
* C99: <stdint.h>でint8\_t、int16\_t、int32\_tなどが使用可能

## 実行方法

# C90版のコンパイルと実行  
gcc -std=c90 -Wall -Wextra -pedantic ex10\_1\_pointer\_swap.c -o ex10\_1\_pointer\_swap  
./ex10\_1\_pointer\_swap  
  
# C99版のコンパイルと実行  
gcc -std=c99 -Wall -Wextra -pedantic ex10\_1\_pointer\_swap\_c99.c -o ex10\_1\_pointer\_swap\_c99  
./ex10\_1\_pointer\_swap\_c99  
  
# その他の演習例  
gcc -std=c90 -Wall -Wextra -pedantic ex10\_2\_array\_reverse.c -o ex10\_2\_array\_reverse  
gcc -std=c90 -Wall -Wextra -pedantic ex10\_3\_string\_manipulation.c -o ex10\_3\_string\_manipulation  
gcc -std=c90 -Wall -Wextra -pedantic ex10\_4\_array\_functions.c -o ex10\_4\_array\_functions  
  
# Makefileを使用した場合  
make run-ex10\_1\_pointer\_swap # C90版  
make run-ex10\_1\_pointer\_swap\_c99 # C99版  
make run-ex10\_2\_array\_reverse # C90版  
make run-ex10\_2\_array\_reverse\_c99 # C99版  
make run-ex10\_3\_string\_manipulation # C90版  
make run-ex10\_3\_string\_manipulation\_c99 # C99版  
make run-ex10\_4\_array\_functions # C90版  
make run-ex10\_4\_array\_functions\_c99 # C99版

## ポインタ使用時の注意事項

### 1. 初期化の重要性

// NG: 未初期化ポインタの使用  
int \*ptr;  
\*ptr = 10; // 危険！未初期化ポインタの参照  
  
// OK: 適切な初期化  
int value = 0;  
int \*ptr = &value;  
\*ptr = 10; // 安全

### 2. NULLポインタのチェック

// 関数でポインタを受け取る際は必ずチェック  
void process\_data(int \*ptr)  
{  
 if (ptr == NULL) {  
 printf("エラー: NULLポインタが渡されました\n");  
 return;  
 }  
 // 処理を続行  
}

### 3. 配列の境界チェック

// 配列のサイズを超えたアクセスは未定義動作  
int arr[5] = {1, 2, 3, 4, 5};  
int \*ptr = arr;  
  
// NG: 境界を超えたアクセス  
printf("%d\n", \*(ptr + 10)); // 危険！  
  
// OK: 境界内でのアクセス  
for (int i = 0; i < 5; i++) {  
 printf("%d ", \*(ptr + i));  
}

### 4. ローカル変数のアドレスを返さない

// NG: ローカル変数のアドレスを返す  
int\* dangerous\_function(void)  
{  
 int local\_var = 100;  
 return &local\_var; // 危険！関数終了後に無効になる  
}  
  
// OK: 静的変数または呼び出し元から受け取ったポインタを返す  
int\* safe\_function(int \*input)  
{  
 static int static\_var = 100;  
 return &static\_var; // 安全（静的変数）  
 // または return input; （引数として受け取ったポインタ）  
}

## デバッグのヒント

1. **アドレスの確認**: printf("%p\n", (void\*)ptr) でポインタの値を確認
2. **値の確認**: printf("%d\n", \*ptr) でポインタが指す値を確認
3. **境界チェック**: 配列アクセス前に必ず範囲をチェック
4. **コンパイラ警告**: -Wall -Wextra -pedantic を使用して警告を有効化

これらの解答例を参考に、ポインタの安全で効果的な使用方法を身につけてください。 ## ex10\_1\_pointer\_swap.c

/\*  
 \* ファイル名: ex10\_1\_pointer\_swap.c  
 \* 演習10-1: ポインタの基本操作  
 \* 説明: 2つの整数変数の値をポインタを使って交換  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
  
/\* ポインタを使った値の交換関数 \*/  
void swap\_with\_pointers(int \*ptr\_a, int \*ptr\_b)  
{  
 int temp;  
   
 printf(" swap関数内部:\n");  
 printf(" 交換前: \*ptr\_a = %d, \*ptr\_b = %d\n", \*ptr\_a, \*ptr\_b);  
 printf(" ポインタ: ptr\_a = %p, ptr\_b = %p\n", (void\*)ptr\_a, (void\*)ptr\_b);  
   
 temp = \*ptr\_a;  
 \*ptr\_a = \*ptr\_b;  
 \*ptr\_b = temp;  
   
 printf(" 交換後: \*ptr\_a = %d, \*ptr\_b = %d\n", \*ptr\_a, \*ptr\_b);  
}  
  
/\* 参考：値渡しでは交換できない例 \*/  
void swap\_by\_value(int a, int b)  
{  
 int temp;  
   
 printf(" 値渡し関数内部:\n");  
 printf(" 交換前: a = %d, b = %d\n", a, b);  
   
 temp = a;  
 a = b;  
 b = temp;  
   
 printf(" 交換後: a = %d, b = %d\n", a, b);  
 printf(" (この変更は呼び出し元には反映されません)\n");  
}  
  
int main(void)  
{  
 int x = 10, y = 20;  
 int \*ptr\_x, \*ptr\_y;  
 int numbers[4] = {100, 200, 300, 400};  
 int \*start, \*end;  
 int i; /\* C90: ループ変数をブロックの先頭で宣言 \*/  
   
 printf("===== ポインタによる値の交換デモ =====\n\n");  
   
 /\* 初期状態の表示 \*/  
 printf("1. 初期状態:\n");  
 printf(" x = %d (アドレス: %p)\n", x, (void\*)&x);  
 printf(" y = %d (アドレス: %p)\n", y, (void\*)&y);  
   
 /\* ポインタの初期化 \*/  
 ptr\_x = &x;  
 ptr\_y = &y;  
   
 printf("\n2. ポインタの初期化後:\n");  
 printf(" ptr\_x = %p (指す値: %d)\n", (void\*)ptr\_x, \*ptr\_x);  
 printf(" ptr\_y = %p (指す値: %d)\n", (void\*)ptr\_y, \*ptr\_y);  
   
 /\* ポインタを使った交換の実行 \*/  
 printf("\n3. ポインタを使った交換の実行:\n");  
 swap\_with\_pointers(ptr\_x, ptr\_y);  
   
 printf("\n4. 交換後の状態（main関数内）:\n");  
 printf(" x = %d (アドレス: %p)\n", x, (void\*)&x);  
 printf(" y = %d (アドレス: %p)\n", y, (void\*)&y);  
 printf(" \*ptr\_x = %d, \*ptr\_y = %d\n", \*ptr\_x, \*ptr\_y);  
   
 /\* 元に戻して値渡しとの比較 \*/  
 printf("\n===== 参考：値渡しとの比較 =====\n");  
   
 /\* 元の値に戻す \*/  
 x = 10; y = 20;  
 printf("\n5. 値を元に戻しました: x = %d, y = %d\n", x, y);  
   
 printf("\n6. 値渡しによる交換試行（効果なし）:\n");  
 swap\_by\_value(x, y);  
   
 printf("\n7. 値渡し後の状態（main関数内）:\n");  
 printf(" x = %d, y = %d (変更されていない)\n", x, y);  
   
 /\* アドレス演算子の使用例 \*/  
 printf("\n===== アドレス演算子の直接使用 =====\n");  
   
 printf("\n8. アドレス演算子を直接使った交換:\n");  
 printf(" 交換前: x = %d, y = %d\n", x, y);  
 swap\_with\_pointers(&x, &y);  
 printf(" 交換後: x = %d, y = %d\n", x, y);  
   
 /\* ポインタ演算の応用例 \*/  
 printf("\n===== ポインタ演算の応用 =====\n");  
   
 start = numbers;  
 end = numbers + 3;  
   
 printf("\n9. 配列の両端要素の交換:\n");  
 printf(" 配列: ");  
 for (i = 0; i < 4; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 printf(" start = %p (\*start = %d)\n", (void\*)start, \*start);  
 printf(" end = %p (\*end = %d)\n", (void\*)end, \*end);  
   
 swap\_with\_pointers(start, end);  
   
 printf(" 交換後の配列: ");  
 for (i = 0; i < 4; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. ポインタによる参照渡し:  
 - 関数で元の変数の値を変更可能  
 - アドレスを渡すことで間接アクセス  
   
2. 値渡しとの違い:  
 - 値渡しでは元の変数は変更されない  
 - ポインタ渡しでは元の変数が変更される  
   
3. アドレス演算子(&)の使用:  
 - &variable で変数のアドレスを取得  
 - 関数呼び出し時に直接使用可能  
   
4. 間接参照演算子(\*)の使用:  
 - \*pointer でポインタが指す値にアクセス  
 - 左辺値として使用すると値の変更が可能  
   
5. メモリアドレスの理解:  
 - 変数のアドレスは実行ごとに変わる可能性  
 - ポインタはアドレスを格納する変数  
   
6. 実用的な応用:  
 - 配列要素の交換  
 - 複数の値を同時に変更する関数  
 - データ構造の操作  
  
注意点:  
- ポインタは必ず有効なアドレスを指すよう初期化  
- NULLポインタの参照は避ける  
- ポインタが指すメモリが有効であることを確認  
\*/

## ex10\_2\_array\_reverse.c

/\*  
 \* ファイル名: ex10\_2\_array\_reverse.c  
 \* 演習10-2: 配列とポインタ  
 \* 説明: ポインタ演算を使って配列の要素を逆順に表示  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
  
/\* インデックス記法で配列を表示 \*/  
void print\_array\_index(int arr[], int size, char \*label)  
{  
 int i;  
   
 printf("%s (インデックス記法): ", label);  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
/\* ポインタ記法で配列を表示 \*/  
void print\_array\_pointer(int \*ptr, int size, char \*label)  
{  
 int i;  
   
 printf("%s (ポインタ記法): ", label);  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*(ptr + i));  
 }  
 printf("\n");  
}  
  
/\* ポインタ演算で配列を逆順表示 \*/  
void print\_array\_reverse\_pointer(int \*ptr, int size)  
{  
 int i;  
 int \*last\_ptr = ptr + size - 1; /\* 最後の要素を指す \*/  
   
 printf("逆順表示 (ポインタ演算): ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*last\_ptr);  
 last\_ptr--; /\* ポインタを前の要素に移動 \*/  
 }  
 printf("\n");  
}  
  
/\* ポインタインクリメントで配列を逆順表示 \*/  
void print\_array\_reverse\_increment(int \*ptr, int size)  
{  
 int \*reverse\_ptr = ptr + size - 1; /\* 最後の要素から開始 \*/  
   
 printf("逆順表示 (デクリメント): ");  
 while (reverse\_ptr >= ptr) {  
 printf("%d ", \*reverse\_ptr);  
 reverse\_ptr--;  
 }  
 printf("\n");  
}  
  
/\* 配列のアドレス情報を表示 \*/  
void show\_address\_info(int \*arr, int size)  
{  
 int i;  
   
 printf("\n=== アドレス情報 ===\n");  
 printf("配列名のアドレス: %p\n", (void\*)arr);  
 printf("&arr[0]のアドレス: %p\n", (void\*)&arr[0]);  
 printf("同じアドレス? %s\n", (arr == &arr[0]) ? "はい" : "いいえ");  
   
 printf("\n各要素のアドレスと値:\n");  
 for (i = 0; i < size; i++) {  
 printf(" arr[%d]: アドレス=%p, 値=%d, ポインタ記法=\*(arr+%d)=%d\n",  
 i, (void\*)&arr[i], arr[i], i, \*(arr + i));  
 }  
   
 printf("\nアドレス間隔の確認:\n");  
 for (i = 0; i < size - 1; i++) {  
 long byte\_diff = (char\*)&arr[i+1] - (char\*)&arr[i];  
 printf(" arr[%d]とarr[%d]の差: %ld バイト\n", i, i+1, byte\_diff);  
 }  
 printf(" int型のサイズ: %lu バイト\n", (unsigned long)sizeof(int));  
}  
  
/\* ポインタ演算の詳細確認 \*/  
void demonstrate\_pointer\_arithmetic(int \*arr, int size)  
{  
 int i;  
 int \*ptr = arr;  
   
 printf("\n=== ポインタ演算の詳細 ===\n");  
 printf("基準ポインタ ptr = %p\n", (void\*)ptr);  
   
 for (i = 0; i < size; i++) {  
 printf("ptr + %d = %p, \*(ptr + %d) = %d\n",  
 i, (void\*)(ptr + i), i, \*(ptr + i));  
 }  
   
 printf("\n負のオフセットでのアクセス:\n");  
 ptr = arr + size - 1; /\* 最後の要素を指す \*/  
 printf("基準ポインタ ptr = %p (最後の要素)\n", (void\*)ptr);  
   
 for (i = 0; i < size; i++) {  
 printf("ptr - %d = %p, \*(ptr - %d) = %d\n",  
 i, (void\*)(ptr - i), i, \*(ptr - i));  
 }  
}  
  
/\* 配列を実際に逆順に並び替える関数 \*/  
void reverse\_array\_inplace(int \*arr, int size)  
{  
 int \*start = arr;  
 int \*end = arr + size - 1;  
 int temp;  
   
 printf("\n=== 配列の実際の逆順並び替え ===\n");  
 printf("並び替え前: ");  
 print\_array\_pointer(arr, size, "");  
   
 while (start < end) {  
 printf("交換: \*%p(%d) ↔ \*%p(%d)\n",   
 (void\*)start, \*start, (void\*)end, \*end);  
   
 temp = \*start;  
 \*start = \*end;  
 \*end = temp;  
   
 start++;  
 end--;  
 }  
   
 printf("並び替え後: ");  
 print\_array\_pointer(arr, size, "");  
}  
  
int main(void)  
{  
 int numbers[] = {10, 20, 30, 40, 50, 60};  
 int backup[] = {10, 20, 30, 40, 50, 60}; /\* 元の配列のバックアップ \*/  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
 int \*ptr = numbers;  
   
 printf("===== ポインタ演算による配列の逆順表示 =====\n\n");  
   
 /\* 配列サイズの動的計算 \*/  
 printf("配列サイズの動的計算:\n");  
 printf(" sizeof(numbers) = %lu バイト\n", (unsigned long)sizeof(numbers));  
 printf(" sizeof(numbers[0]) = %lu バイト\n", (unsigned long)sizeof(numbers[0]));  
 printf(" 要素数 = %d\n", size);  
   
 printf("\n1. 通常の表示:\n");  
 print\_array\_index(numbers, size, "元の配列");  
 print\_array\_pointer(ptr, size, "元の配列");  
   
 printf("\n2. 逆順表示の方法:\n");  
 print\_array\_reverse\_pointer(ptr, size);  
 print\_array\_reverse\_increment(ptr, size);  
   
 /\* アドレス情報の表示 \*/  
 show\_address\_info(numbers, size);  
   
 /\* ポインタ演算の詳細 \*/  
 demonstrate\_pointer\_arithmetic(numbers, size);  
   
 /\* 実際の逆順並び替え \*/  
 reverse\_array\_inplace(numbers, size);  
   
 printf("\n=== 元の配列との比較 ===\n");  
 print\_array\_pointer(backup, size, "元の配列");  
 print\_array\_pointer(numbers, size, "逆順配列");  
   
 /\* 配列を元に戻す \*/  
 reverse\_array\_inplace(numbers, size);  
 printf("\n再度逆順にして元に戻しました:\n");  
 print\_array\_pointer(numbers, size, "復元された配列");  
   
 /\* 部分配列の逆順表示 \*/  
 printf("\n=== 部分配列の逆順表示 ===\n");  
 {  
 int start\_index = 1;  
 int end\_index = 4;  
 int partial\_size = end\_index - start\_index + 1;  
 int i; /\* C90: ループ変数をブロックの先頭で宣言 \*/  
 int \*partial\_end;  
   
 printf("部分配列 [%d-%d] の逆順表示:\n", start\_index, end\_index);  
 printf("通常: ");  
 for (i = start\_index; i <= end\_index; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 printf("逆順: ");  
 for (i = end\_index; i >= start\_index; i--) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 printf("ポインタ演算: ");  
 partial\_end = numbers + end\_index;  
 for (i = 0; i < partial\_size; i++) {  
 printf("%d ", \*(partial\_end - i));  
 }  
 printf("\n");  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 配列とポインタの等価性:  
 - 配列名は先頭要素のポインタと同等  
 - arr[i] と \*(arr + i) は同じ意味  
   
2. ポインタ演算:  
 - ptr + n は n個後の要素を指す  
 - ptr - n は n個前の要素を指す  
 - ポインタ同士の差は要素数  
   
3. 逆順アクセスの方法:  
 - 最後の要素から順次デクリメント  
 - 負のオフセットを使用  
 - ポインタの境界チェック  
   
4. メモリレイアウト:  
 - 配列要素は連続したメモリに配置  
 - アドレス差 = 要素サイズ × 要素数の差  
   
5. 実用的な応用:  
 - 配列の逆順並び替え  
 - 部分配列の処理  
 - 動的なサイズ計算  
   
6. 安全なプログラミング:  
 - 配列の境界チェック (start < end)  
 - ポインタの有効性確認  
 - オーバーフロー・アンダーフローの防止  
  
注意点:  
- 配列の範囲外アクセスは未定義動作  
- ポインタ演算は型サイズを自動考慮  
- sizeof演算子は配列全体のサイズを返す  
\*/

## ex10\_3\_string\_manipulation.c

/\*  
 \* ex10\_3\_string\_manipulation.c  
 \*   
 \* 演習10-3: 文字列操作  
 \*   
 \* ポインタを使った文字列操作の実装  
 \* - カスタムstrlen関数  
 \* - 文字列逆順関数  
 \* - 文字列リテラルと文字配列の違い  
 \*   
 \* Author: Claude  
 \* Date: 2025-06-29  
 \*/  
  
#include <stdio.h>  
  
/\* カスタムstrlen関数：ポインタを使って文字列の長さを計算 \*/  
size\_t my\_strlen(const char \*str) {  
 const char \*p = str; /\* ポインタで文字列の先頭を記憶 \*/  
   
 /\* ヌル文字に到達するまでポインタを進める \*/  
 while (\*p != '\0') {  
 p++;  
 }  
   
 /\* ポインタの差分が文字列の長さ \*/  
 return p - str;  
}  
  
/\* 文字列を逆順にする関数：ポインタを使って実装 \*/  
void reverse\_string(char \*str) {  
 char \*start = str; /\* 文字列の先頭ポインタ \*/  
 char \*end = str + my\_strlen(str) - 1; /\* 文字列の末尾ポインタ \*/  
 char temp;  
   
 /\* 先頭と末尾のポインタが交差するまで文字を交換 \*/  
 while (start < end) {  
 /\* ポインタを使って文字を交換 \*/  
 temp = \*start;  
 \*start = \*end;  
 \*end = temp;  
   
 /\* ポインタを内側に移動 \*/  
 start++;  
 end--;  
 }  
}  
  
/\* 文字列情報を表示する関数 \*/  
void print\_string\_info(const char \*label, const char \*str) {  
 printf("%s: \"%s\"\n", label, str);  
 printf(" アドレス: %p\n", (void\*)str);  
 printf(" 長さ: %zu文字\n", my\_strlen(str));  
}  
  
int main(void) {  
 /\* 文字列リテラル（読み取り専用） \*/  
 const char \*literal = "Hello, World!";  
   
 /\* 文字配列（変更可能） \*/  
 char array1[] = "C Programming";  
 char array2[] = "Pointer Practice";  
 char array3[] = "12345";  
   
 printf("=== 文字列操作のデモンストレーション ===\n\n");  
   
 /\* 1. カスタムstrlen関数のテスト \*/  
 printf("1. 文字列の長さ計算（my\_strlen関数）\n");  
 printf("----------------------------------------\n");  
 print\_string\_info("文字列リテラル", literal);  
 print\_string\_info("文字配列1", array1);  
 print\_string\_info("文字配列2", array2);  
 print\_string\_info("文字配列3", array3);  
 printf("\n");  
   
 /\* 2. 文字列の逆順処理 \*/  
 printf("2. 文字列の逆順処理\n");  
 printf("----------------------------------------\n");  
   
 /\* 文字列リテラルは変更できないことを説明 \*/  
 printf("文字列リテラル \"%s\" は読み取り専用のため、\n", literal);  
 printf("逆順にすることはできません。\n\n");  
   
 /\* 文字配列は変更可能 \*/  
 printf("文字配列1の逆順処理:\n");  
 printf(" 元の文字列: \"%s\"\n", array1);  
 reverse\_string(array1);  
 printf(" 逆順文字列: \"%s\"\n\n", array1);  
   
 printf("文字配列2の逆順処理:\n");  
 printf(" 元の文字列: \"%s\"\n", array2);  
 reverse\_string(array2);  
 printf(" 逆順文字列: \"%s\"\n\n", array2);  
   
 printf("文字配列3の逆順処理:\n");  
 printf(" 元の文字列: \"%s\"\n", array3);  
 reverse\_string(array3);  
 printf(" 逆順文字列: \"%s\"\n\n", array3);  
   
 /\* 3. ポインタ操作の詳細説明 \*/  
 printf("3. ポインタ操作の仕組み\n");  
 printf("----------------------------------------\n");  
   
 /\* 文字列をポインタで走査する例 \*/  
 {  
 const char \*p;  
 int count = 0;  
   
 printf("文字列 \"%s\" の各文字:\n", array3);  
 for (p = array3; \*p != '\0'; p++) {  
 printf(" array3[%d] = '%c' (アドレス: %p)\n",   
 count, \*p, (void\*)p);  
 count++;  
 }  
 }  
   
 printf("\n");  
   
 /\* 4. 文字列リテラルと文字配列の違い \*/  
 printf("4. 文字列リテラルと文字配列の違い\n");  
 printf("----------------------------------------\n");  
 printf("文字列リテラル:\n");  
 printf(" - const char \*型で宣言\n");  
 printf(" - 読み取り専用メモリに配置\n");  
 printf(" - 変更しようとするとエラー\n\n");  
   
 printf("文字配列:\n");  
 printf(" - char array[]型で宣言\n");  
 printf(" - スタック上に配置\n");  
 printf(" - 変更可能\n");  
   
 return 0;  
}

## ex10\_4\_array\_functions.c

/\*  
 \* 演習10-4: 配列操作関数  
 \*   
 \* 配列を操作する関数をポインタを使って実装する演習です。  
 \* - 最大値・最小値を見つけてポインタを返す  
 \* - 平均値を計算する  
 \* - 統計情報をレポート形式で表示する  
 \*   
 \* 作成者: C言語学習者  
 \* 作成日: 2024年  
 \*/  
  
#include <stdio.h>  
  
/\* 関数プロトタイプ宣言 \*/  
int\* find\_max(int \*arr, int size);  
int\* find\_min(int \*arr, int size);  
double calculate\_average(int \*arr, int size);  
void print\_statistics(int \*arr, int size);  
  
/\*  
 \* 配列の最大値を見つけてそのポインタを返す  
 \* 引数: arr - 配列の先頭ポインタ, size - 配列のサイズ  
 \* 戻り値: 最大値へのポインタ  
 \*/  
int\* find\_max(int \*arr, int size) {  
 int \*max\_ptr = arr; /\* 最初は先頭要素を最大値と仮定 \*/  
 int i;  
   
 /\* 配列の全要素を走査 \*/  
 for (i = 1; i < size; i++) {  
 /\*  
 \* ポインタ演算を使った要素アクセス  
 \* arr + i は i番目の要素のアドレス  
 \* \*(arr + i) でその値を取得  
 \*/  
 if (\*(arr + i) > \*max\_ptr) {  
 max\_ptr = arr + i; /\* 新しい最大値のアドレスを記録 \*/  
 }  
 }  
   
 return max\_ptr; /\* 最大値のアドレスを返す \*/  
}  
  
/\*  
 \* 配列の最小値を見つけてそのポインタを返す  
 \* 引数: arr - 配列の先頭ポインタ, size - 配列のサイズ  
 \* 戻り値: 最小値へのポインタ  
 \*/  
int\* find\_min(int \*arr, int size) {  
 int \*min\_ptr = arr; /\* 最初は先頭要素を最小値と仮定 \*/  
 int i;  
   
 /\* 配列の全要素を走査 \*/  
 for (i = 1; i < size; i++) {  
 /\* ポインタを使った比較 \*/  
 if (\*(arr + i) < \*min\_ptr) {  
 min\_ptr = arr + i; /\* 新しい最小値のアドレスを記録 \*/  
 }  
 }  
   
 return min\_ptr; /\* 最小値のアドレスを返す \*/  
}  
  
/\*  
 \* 配列の平均値を計算する  
 \* 引数: arr - 配列の先頭ポインタ, size - 配列のサイズ  
 \* 戻り値: 平均値（double型）  
 \*/  
double calculate\_average(int \*arr, int size) {  
 int sum = 0;  
 int i;  
   
 /\* ポインタを使った要素の合計計算 \*/  
 for (i = 0; i < size; i++) {  
 sum += \*(arr + i); /\* ポインタ演算で各要素にアクセス \*/  
 }  
   
 /\* 平均値を計算して返す（double型にキャスト） \*/  
 return (double)sum / size;  
}  
  
/\*  
 \* 配列の統計情報を表示する  
 \* 最大値、最小値、平均値とそれらのポインタ情報を含む  
 \*/  
void print\_statistics(int \*arr, int size) {  
 int \*max\_ptr, \*min\_ptr;  
 double average;  
 int i;  
   
 printf("========== 配列統計レポート ==========\n");  
 printf("配列のサイズ: %d要素\n\n", size);  
   
 /\* 配列の内容を表示 \*/  
 printf("配列の内容:\n");  
 for (i = 0; i < size; i++) {  
 printf(" arr[%d] = %d (アドレス: %p)\n",   
 i, \*(arr + i), (void\*)(arr + i));  
 }  
 printf("\n");  
   
 /\* 最大値の情報 \*/  
 max\_ptr = find\_max(arr, size);  
 printf("最大値:\n");  
 printf(" 値: %d\n", \*max\_ptr);  
 printf(" アドレス: %p\n", (void\*)max\_ptr);  
 printf(" インデックス: %ld\n", (long)(max\_ptr - arr));  
 printf("\n");  
   
 /\* 最小値の情報 \*/  
 min\_ptr = find\_min(arr, size);  
 printf("最小値:\n");  
 printf(" 値: %d\n", \*min\_ptr);  
 printf(" アドレス: %p\n", (void\*)min\_ptr);  
 printf(" インデックス: %ld\n", (long)(min\_ptr - arr));  
 printf("\n");  
   
 /\* 平均値の情報 \*/  
 average = calculate\_average(arr, size);  
 printf("平均値: %.2f\n\n", average);  
   
 /\* ポインタ解析情報 \*/  
 printf("ポインタ解析:\n");  
 printf(" 配列の先頭アドレス: %p\n", (void\*)arr);  
 printf(" 最大値と先頭の差: %ld要素\n", (long)(max\_ptr - arr));  
 printf(" 最小値と先頭の差: %ld要素\n", (long)(min\_ptr - arr));  
 printf(" 最大値と最小値の差: %ld要素\n", (long)(max\_ptr - min\_ptr));  
   
 printf("======================================\n");  
}  
  
int main(void) {  
 /\* デモ用配列データ \*/  
 int numbers[] = {45, 23, 67, 89, 12, 34, 56, 78, 90, 1};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
   
 printf("配列操作関数のデモンストレーション\n");  
 printf("=====================================\n\n");  
   
 /\* 統計情報を表示 \*/  
 print\_statistics(numbers, size);  
   
 /\* 個別の関数呼び出しデモ \*/  
 printf("\n個別関数の使用例:\n");  
 printf("------------------\n");  
   
 /\* find\_max関数の使用例 \*/  
 {  
 int \*max = find\_max(numbers, size);  
 printf("find\_max()の結果: 値=%d, アドレス=%p\n",   
 \*max, (void\*)max);  
 }  
   
 /\* find\_min関数の使用例 \*/  
 {  
 int \*min = find\_min(numbers, size);  
 printf("find\_min()の結果: 値=%d, アドレス=%p\n",   
 \*min, (void\*)min);  
 }  
   
 /\* calculate\_average関数の使用例 \*/  
 {  
 double avg = calculate\_average(numbers, size);  
 printf("calculate\_average()の結果: %.2f\n", avg);  
 }  
   
 printf("\n");  
   
 /\* ポインタ概念の説明 \*/  
 printf("ポインタの重要な概念:\n");  
 printf("--------------------\n");  
 printf("1. 配列名は配列の先頭要素へのポインタとして扱われる\n");  
 printf("2. 関数に配列を渡すと、実際にはポインタが渡される\n");  
 printf("3. ポインタ演算により配列要素にアクセスできる\n");  
 printf("4. 関数からポインタを返すことで、配列内の特定要素を指示できる\n");  
   
 return 0;  
}