# 第12章 関数ポインタ - 演習問題

# 演習問題

## 演習の目的

* 関数ポインタの基本概念を理解する
* コールバック関数の実装方法を習得する
* 関数ポインタ配列を活用した動的関数選択を学ぶ
* 実行時関数切り替えシステムの設計を理解する

## 演習問題

### 演習12-1: 関数ポインタの基本操作

関数ポインタを使って複数の関数を動的に呼び出すプログラムを作成してください。

**要件:** - 2つの整数を受け取り、結果を返す関数を複数定義する - 関数ポインタを使ってこれらの関数を呼び出す - 関数ポインタ配列を使った実装も含める

**実装すべき関数:** - int maximum(int a, int b) - 大きい方の値を返す - int minimum(int a, int b) - 小さい方の値を返す  
- int power(int a, int b) - aのb乗を返す（簡単な実装で可）

**ファイル名:** ex12\_1\_basic\_function\_pointer.c

### 演習12-2: 関数選択システム

文字に基づいて関数を選択し実行するシステムを実装してください。

**要件:** - 文字（‘+’, ‘-’, ’\*‘,’/’など）に基づいて関数を選択 - 選択された関数を実行して結果を表示 - 無効な文字が指定された場合のエラーハンドリング - 構造体を使った関数ポインタ管理

**ファイル名:** ex12\_2\_function\_selector.c

### 演習12-3: 配列処理のコールバック

コールバック関数を使って配列の各要素に異なる処理を適用するプログラムを作成してください。

**要件:** - 整数配列を処理する関数を複数定義 - コールバック関数として配列処理関数に渡す - 処理前後の配列の状態を表示 - 動的な処理選択機能

**実装すべき処理:** - 各要素を倍にする - 各要素から1を引く - 各要素の符号を反転する

**ファイル名:** ex12\_3\_array\_callback.c

### 演習12-4: 関数ポインタ配列を使った計算機

関数ポインタ配列を使用した計算機プログラムを作成してください。

**要件:** - 関数ポインタの配列を定義 - インデックスを指定して演算を選択 - 複数の演算を連続で実行可能 - 演算履歴を表示する機能 - 統計情報の収集と表示

**実装すべき演算:** - 加算、減算、乗算、除算 - ゼロ除算のエラーハンドリング

**ファイル名:** ex12\_4\_calculator\_function\_array.c

## チャレンジ問題

### チャレンジ12-5: ソートアルゴリズム選択システム

異なるソートアルゴリズムを関数ポインタで切り替えるシステムを作成してください。

**要件:** - バブルソート、選択ソート、挿入ソートを実装 - 比較関数も関数ポインタで指定（昇順/降順） - ソート前後の配列状態を表示 - アルゴリズムの性能比較機能

**ファイル名:** ex12\_5\_sort\_algorithms.c

### チャレンジ12-6: イベント駆動システム

コールバック関数を使ったイベント駆動システムを実装してください。

**要件:** - 複数種類のイベント（開始、停止、エラー、警告） - イベントタイプごとに異なるハンドラーを登録 - イベント発生時に適切なハンドラーを呼び出し - イベントの優先度機能

**ファイル名:** ex12\_6\_event\_system.c

## 提出方法

1. 各演習問題ごとに独立したCファイルを作成
2. ファイル名は ex12\_1.c, ex12\_2.c のような形式で
3. 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
5. C99版も作成し、\_c99.c サフィックスを付ける

## コンパイル例

### C90準拠

gcc -std=c90 -Wall -Wextra -pedantic ex12\_1\_basic\_function\_pointer.c -o ex12\_1\_basic\_function\_pointer

### C99準拠

gcc -std=c99 -Wall -Wextra -pedantic ex12\_1\_basic\_function\_pointer\_c99.c -o ex12\_1\_basic\_function\_pointer\_c99

## 学習のポイント

### 関数ポインタの基本

1. **宣言**: int (\*func\_ptr)(int, int);
2. **初期化**: func\_ptr = function\_name;
3. **呼び出し**: result = func\_ptr(a, b);
4. **配列**: int (\*operations[])(int, int) = {add, sub, mul};

### コールバック関数

1. **概念**: 関数を引数として渡す仕組み
2. **活用**: 動的な処理選択、カスタマイズ可能な処理
3. **設計**: 関数ポインタを引数に取る関数の実装

### 実用的なテクニック

1. **関数テーブル**: 構造体による関数ポインタ管理
2. **エラーハンドリング**: 無効な関数ポインタの検出
3. **状態管理**: 関数ポインタによる状態遷移
4. **性能**: 関数ポインタ使用時のオーバーヘッド考慮

## 注意事項

* 関数ポインタは初期化前に使用しない
* 型の一致を厳密に確認する
* NULL ポインタチェックを忘れずに
* デバッグ時は関数名の確認が困難な場合がある

## 実用的な応用

これらの技術は以下のような場面で使用されます：

* **GUI フレームワーク**: イベントハンドラ
* **組み込みシステム**: 割り込みハンドラ
* **ゲーム開発**: 状態管理、AI 行動パターン
* **システムプログラミング**: プラグインシステム

## 次のステップ

この章をマスターしたら、次の章に進みましょう： - [第13章: 高度なプログラミング技法](../advanced/) - 関数ポインタを活用したより複雑なシステム設計 - マルチスレッドプログラミングでの関数ポインタ活用

# 解答例

このディレクトリには、第12章の演習問題の解答例が含まれています。各解答例はC90標準準拠版とC99版の両方を提供し、すべて非対話型のデモ版として実装されています。

## 解答例一覧

### 演習12-1: 関数ポインタの基本操作

* **C90版**: <ex12_1_basic_function_pointer.c>
* **C99版**: <ex12_1_basic_function_pointer_c99.c>
* **内容**: 関数ポインタを使った動的関数呼び出しのデモ（maximum, minimum, power関数）
* **学習ポイント**: 関数ポインタの宣言、初期化、呼び出し、配列での管理

### 演習12-2: 関数選択システム

* **C90版**: <ex12_2_function_selector.c>
* **C99版**: <ex12_2_function_selector_c99.c>
* **内容**: 文字に基づく関数選択システム（構造体による関数ポインタ管理）
* **学習ポイント**: 構造体と関数ポインタの組み合わせ、エラーハンドリング

### 演習12-3: 配列処理のコールバック

* **C90版**: <ex12_3_array_callback.c>
* **C99版**: <ex12_3_array_callback_c99.c>
* **内容**: コールバック関数を使った配列要素処理（倍加、減算、符号反転）
* **学習ポイント**: コールバック関数の概念、関数ポインタを引数として渡す方法

### 演習12-4: 関数ポインタ配列を使った計算機

* **C90版**: <ex12_4_calculator_function_array.c>
* **C99版**: <ex12_4_calculator_function_array_c99.c>
* **内容**: 関数ポインタ配列による計算機（履歴管理、統計機能付き）
* **学習ポイント**: 関数ポインタ配列、履歴管理、連続計算

## 実装上の特徴

### 非対話型デモ版について

すべての解答例は、クラッシュを避けるため scanf() による対話型入力を排除し、予め定義された値を使用するデモ版として実装されています。

### 教育的配慮

* 各ファイルには詳細なコメントと学習ポイントを記載
* C90とC99の違いを明確に示すコード例
* 実用的な応用例を含むデモンストレーション

### セキュリティ対策

* ゼロ除算エラーのハンドリング
* 配列境界の適切なチェック
* 無効な関数ポインタの検出

## C90版とC99版の主な違い

### C90版の特徴

* 変数宣言は関数やブロックの先頭で行う
* /\* \*/ 形式のコメントのみ使用
* for文の初期化部で変数宣言不可
* bool型は独自定義または整数型で代替

### C99版の特徴

* 変数宣言を使用箇所で可能
* // 形式のコメントも使用可能
* for文内での変数宣言が可能
* <stdbool.h>のbool型を使用
* 指定初期化子による構造体初期化
* 可変長配列（VLA）のサポート

## コンパイル方法

### C90版のコンパイル

gcc -Wall -Wextra -pedantic -std=c90 -o program\_name source\_file.c

### C99版のコンパイル

gcc -Wall -Wextra -pedantic -std=c99 -o program\_name source\_file\_c99.c

## 学習のポイント

1. **関数ポインターの基本**
   * 関数ポインターの宣言方法
   * 関数アドレスの取得と呼び出し
   * 関数ポインターの型定義（typedef）
2. **関数ポインターの応用**
   * コールバック関数の実装
   * 関数ポインター配列の活用
   * 動的な関数選択
3. **設計パターン**
   * イベント駆動プログラミング
   * プラグインアーキテクチャ
   * 状態マシンパターン
4. **実践的な活用**
   * エラーハンドリング
   * 拡張性の高い設計
   * 保守性を考慮したコード構造

## 注意事項

* これらの解答例は一つの実装方法を示したものです
* 実際の開発では、要件に応じて適切な実装を選択してください
* エラーハンドリングは基本的なレベルに留めています
* メモリ管理が必要な場合は、適切な解放処理を追加してください ## ex12\_1\_basic\_function\_pointer.c

/\*  
 \* ファイル名: ex12\_1\_basic\_function\_pointer.c  
 \* 演習12-1: 関数ポインタの基本操作  
 \* 説明: 関数ポインタを使って複数の関数を動的に呼び出すデモプログラム  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
/\* 演算関数群 \*/  
int maximum(int a, int b)  
{  
 return (a > b) ? a : b;  
}  
  
int minimum(int a, int b)  
{  
 return (a < b) ? a : b;  
}  
  
int power(int a, int b)  
{  
 /\* 簡単な累乗実装（正の整数のみ） \*/  
 int result = 1;  
 int i;  
   
 if (b < 0) {  
 return 0; /\* 負の指数は0を返す \*/  
 }  
   
 for (i = 0; i < b; i++) {  
 result \*= a;  
 }  
   
 return result;  
}  
  
int main(void)  
{  
 /\* 変数の宣言（C90では先頭で宣言） \*/  
 int (\*operation)(int, int);  
 int a, b, result;  
   
 printf("=== 関数ポインタの基本操作デモ ===\n");  
   
 /\* デモ用の値（実際のプログラムでは入力を受け取る） \*/  
 a = 8;  
 b = 3;  
   
 printf("対象の値: a = %d, b = %d\n\n", a, b);  
   
 /\* 1. maximum関数のテスト \*/  
 operation = maximum;  
 result = operation(a, b);  
 printf("maximum(%d, %d) = %d\n", a, b, result);  
   
 /\* 2. minimum関数のテスト \*/  
 operation = minimum;  
 result = operation(a, b);  
 printf("minimum(%d, %d) = %d\n", a, b, result);  
   
 /\* 3. power関数のテスト \*/  
 operation = power;  
 result = operation(a, b);  
 printf("power(%d, %d) = %d\n", a, b, result);  
   
 /\* 関数ポインタ配列のデモ \*/  
 {  
 int (\*operations[])(int, int) = {maximum, minimum, power};  
 char \*names[] = {"maximum", "minimum", "power"};  
 int i;  
   
 printf("\n=== 関数ポインタ配列デモ ===\n");  
 for (i = 0; i < 3; i++) {  
 result = operations[i](a, b);  
 printf("%s(%d, %d) = %d\n", names[i], a, b, result);  
 }  
 }  
   
 /\* 異なる値での演算デモ \*/  
 printf("\n=== 異なる値での演算例 ===\n");  
 {  
 int test\_values[][2] = {{10, 5}, {3, 7}, {2, 4}, {15, 15}};  
 int num\_tests = sizeof(test\_values) / sizeof(test\_values[0]);  
 int i;  
   
 for (i = 0; i < num\_tests; i++) {  
 int x = test\_values[i][0];  
 int y = test\_values[i][1];  
   
 printf("値: %d, %d\n", x, y);  
 printf(" max: %d, min: %d, pow: %d\n",   
 maximum(x, y), minimum(x, y), power(x, y));  
 }  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 関数ポインタの宣言と初期化  
2. 関数ポインタを通じた間接関数呼び出し  
3. 関数ポインタ配列の活用  
4. 実行時の動的な関数選択  
  
実装のポイント:  
- C90準拠のため、すべての変数を関数先頭で宣言  
- 関数ポインタの型宣言: int (\*operation)(int, int)  
- 関数名は関数のアドレスを表す  
- デモ版として固定値を使用（実際は入力を受け取る）  
- 配列を使った複数の関数ポインタ管理  
\*/

## ex12\_2\_function\_selector.c

/\*  
 \* ファイル名: ex12\_2\_function\_selector.c  
 \* 演習12-2: 関数選択システム  
 \* 説明: 文字に基づいて関数を選択し実行するデモプログラム  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
/\* 演算関数群 \*/  
int add(int a, int b)  
{  
 return a + b;  
}  
  
int subtract(int a, int b)  
{  
 return a - b;  
}  
  
int multiply(int a, int b)  
{  
 return a \* b;  
}  
  
int divide\_safe(int a, int b)  
{  
 if (b == 0) {  
 printf("エラー: ゼロ除算を検出\n");  
 return 0;  
 }  
 return a / b;  
}  
  
/\* 関数ポインタと対応する文字を管理する構造体 \*/  
typedef struct {  
 char op\_char;  
 int (\*func)(int, int);  
 char \*name;  
} OperationMapping;  
  
int main(void)  
{  
 /\* 変数の宣言（C90では先頭で宣言） \*/  
 OperationMapping operations[] = {  
 {'+', add, "加算"},  
 {'-', subtract, "減算"},   
 {'\*', multiply, "乗算"},  
 {'/', divide\_safe, "除算"}  
 };  
 int num\_operations = sizeof(operations) / sizeof(operations[0]);  
 int a, b, result;  
 int i, j;  
 char demo\_ops[] = {'+', '-', '\*', '/', 'X'}; /\* X は無効な演算子 \*/  
 int num\_demos = sizeof(demo\_ops) / sizeof(demo\_ops[0]);  
   
 printf("=== 関数選択システムデモ ===\n");  
   
 /\* デモ用の値 \*/  
 a = 12;  
 b = 4;  
   
 printf("対象の値: a = %d, b = %d\n\n", a, b);  
   
 /\* 各演算子のデモ \*/  
 for (i = 0; i < num\_demos; i++) {  
 char op = demo\_ops[i];  
 int found = 0;  
   
 printf("演算子 '%c' を選択:\n", op);  
   
 /\* 対応する関数を検索 \*/  
 for (j = 0; j < num\_operations; j++) {  
 if (operations[j].op\_char == op) {  
 result = operations[j].func(a, b);  
 printf(" %s: %d %c %d = %d\n",   
 operations[j].name, a, op, b, result);  
 found = 1;  
 break;  
 }  
 }  
   
 if (!found) {  
 printf(" エラー: 無効な演算子です\n");  
 }  
 printf("\n");  
 }  
   
 /\* 関数ポインタ配列を使った方法 \*/  
 printf("=== 関数ポインタ配列を使った実装 ===\n");  
 {  
 int (\*operation\_funcs[])(int, int) = {add, subtract, multiply, divide\_safe};  
 char operators[] = {'+', '-', '\*', '/'};  
 char \*names[] = {"加算", "減算", "乗算", "除算"};  
   
 for (i = 0; i < 4; i++) {  
 result = operation\_funcs[i](a, b);  
 printf("%s (%c): %d %c %d = %d\n",   
 names[i], operators[i], a, operators[i], b, result);  
 }  
 }  
   
 /\* 異なる値での演算例 \*/  
 printf("\n=== 異なる値での演算例 ===\n");  
 {  
 int test\_values[][2] = {{20, 5}, {7, 3}, {15, 0}};  
 int num\_tests = sizeof(test\_values) / sizeof(test\_values[0]);  
   
 for (i = 0; i < num\_tests; i++) {  
 int x = test\_values[i][0];  
 int y = test\_values[i][1];  
   
 printf("値: %d, %d\n", x, y);  
 printf(" 加算: %d, 減算: %d, 乗算: %d, 除算: %d\n",  
 add(x, y), subtract(x, y), multiply(x, y), divide\_safe(x, y));  
 }  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 文字による関数選択システム  
2. 構造体を使った関数ポインタ管理  
3. エラーハンドリングの実装  
4. 動的な関数呼び出し  
  
実装のポイント:  
- C90準拠のため、すべての変数を関数先頭で宣言  
- 構造体配列による関数ポインタの管理  
- ループによる関数検索と実行  
- デモ版として固定値を使用（実際は入力を受け取る）  
- 無効な演算子のエラーハンドリング  
\*/

## ex12\_3\_array\_callback.c

/\*  
 \* ファイル名: ex12\_3\_array\_callback.c  
 \* 演習12-3: 配列処理のコールバック  
 \* 説明: コールバック関数を使って配列の各要素に異なる処理を適用するプログラム  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
#define ARRAY\_SIZE 6  
  
/\* 配列処理用のコールバック関数群 \*/  
void double\_element(int \*element)  
{  
 \*element \*= 2;  
}  
  
void decrement\_element(int \*element)  
{  
 \*element -= 1;  
}  
  
void negate\_element(int \*element)  
{  
 \*element = -(\*element);  
}  
  
/\* 配列を表示する関数 \*/  
void print\_array(int \*arr, int size, char \*description)  
{  
 int i;  
   
 printf("%s: [", description);  
 for (i = 0; i < size; i++) {  
 printf("%d", arr[i]);  
 if (i < size - 1) {  
 printf(", ");  
 }  
 }  
 printf("]\n");  
}  
  
/\* 配列の各要素にコールバック関数を適用する関数 \*/  
void apply\_to\_array(int \*arr, int size, void (\*callback)(int \*), char \*operation\_name)  
{  
 int i;  
   
 printf("\n=== %s を適用 ===\n", operation\_name);  
 print\_array(arr, size, "処理前");  
   
 for (i = 0; i < size; i++) {  
 callback(&arr[i]);  
 }  
   
 print\_array(arr, size, "処理後");  
}  
  
/\* 配列を初期値にリセットする関数 \*/  
void reset\_array(int \*arr, int \*original, int size)  
{  
 int i;  
 for (i = 0; i < size; i++) {  
 arr[i] = original[i];  
 }  
}  
  
int main(void)  
{  
 /\* 変数の宣言（C90では先頭で宣言） \*/  
 int original\_array[ARRAY\_SIZE] = {1, 2, 3, 4, 5, 6};  
 int work\_array[ARRAY\_SIZE];  
 int i;  
   
 /\* コールバック関数ポインタ配列 \*/  
 void (\*operations[])(int \*) = {double\_element, decrement\_element, negate\_element};  
 char \*operation\_names[] = {"各要素を倍にする", "各要素から1を引く", "各要素の符号を反転"};  
 int num\_operations = sizeof(operations) / sizeof(operations[0]);  
   
 printf("=== 配列処理のコールバックデモ ===\n");  
   
 /\* 初期配列の表示 \*/  
 print\_array(original\_array, ARRAY\_SIZE, "初期配列");  
   
 /\* 各操作を順次実行 \*/  
 for (i = 0; i < num\_operations; i++) {  
 /\* 作業用配列に初期値をコピー \*/  
 reset\_array(work\_array, original\_array, ARRAY\_SIZE);  
   
 /\* コールバック関数を適用 \*/  
 apply\_to\_array(work\_array, ARRAY\_SIZE, operations[i], operation\_names[i]);  
 }  
   
 /\* 複数の操作を連続適用するデモ \*/  
 printf("\n=== 複数操作の連続適用デモ ===\n");  
 reset\_array(work\_array, original\_array, ARRAY\_SIZE);  
 print\_array(work\_array, ARRAY\_SIZE, "開始");  
   
 /\* 1. 倍にする \*/  
 for (i = 0; i < ARRAY\_SIZE; i++) {  
 double\_element(&work\_array[i]);  
 }  
 print\_array(work\_array, ARRAY\_SIZE, "倍にした後");  
   
 /\* 2. 1を引く \*/  
 for (i = 0; i < ARRAY\_SIZE; i++) {  
 decrement\_element(&work\_array[i]);  
 }  
 print\_array(work\_array, ARRAY\_SIZE, "1を引いた後");  
   
 /\* 3. 符号を反転 \*/  
 for (i = 0; i < ARRAY\_SIZE; i++) {  
 negate\_element(&work\_array[i]);  
 }  
 print\_array(work\_array, ARRAY\_SIZE, "符号反転後");  
   
 /\* 関数ポインタを使った動的選択のデモ \*/  
 printf("\n=== 動的関数選択デモ ===\n");  
 {  
 int selection\_pattern[] = {0, 1, 2, 0, 1, 2}; /\* 操作の選択パターン \*/  
   
 reset\_array(work\_array, original\_array, ARRAY\_SIZE);  
 print\_array(work\_array, ARRAY\_SIZE, "開始");  
   
 for (i = 0; i < ARRAY\_SIZE; i++) {  
 int op\_index = selection\_pattern[i];  
 printf("要素[%d]: %s を適用\n", i, operation\_names[op\_index]);  
 operations[op\_index](&work\_array[i]);  
 }  
   
 print\_array(work\_array, ARRAY\_SIZE, "最終結果");  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. コールバック関数の概念と実装  
2. 関数ポインタを引数として渡す方法  
3. 配列処理での関数ポインタ活用  
4. 動的な処理選択システム  
  
実装のポイント:  
- C90準拠のため、すべての変数を関数先頭で宣言  
- void (\*callback)(int \*) 型の関数ポインタ  
- コールバック関数は配列要素を直接変更  
- 関数ポインタ配列による複数操作の管理  
- デモ版として予め定義された処理パターンを使用  
\*/

## ex12\_4\_calculator\_function\_array.c

/\*  
 \* ファイル名: ex12\_4\_calculator\_function\_array.c  
 \* 演習12-4: 関数ポインタ配列を使った計算機  
 \* 説明: 関数ポインタ配列を使用した計算機プログラム（デモ版）  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
#define MAX\_OPERATIONS 4  
#define MAX\_HISTORY 10  
  
/\* 計算履歴を記録する構造体 \*/  
typedef struct {  
 int operand1;  
 int operand2;  
 char operation;  
 int result;  
} CalculationHistory;  
  
/\* 演算関数群 \*/  
int add(int a, int b)  
{  
 return a + b;  
}  
  
int subtract(int a, int b)  
{  
 return a - b;  
}  
  
int multiply(int a, int b)  
{  
 return a \* b;  
}  
  
int divide\_safe(int a, int b)  
{  
 if (b == 0) {  
 printf("エラー: ゼロ除算\n");  
 return 0;  
 }  
 return a / b;  
}  
  
/\* 履歴を表示する関数 \*/  
void print\_history(CalculationHistory \*history, int count)  
{  
 int i;  
   
 printf("\n=== 計算履歴 ===\n");  
 if (count == 0) {  
 printf("履歴はありません\n");  
 return;  
 }  
   
 for (i = 0; i < count; i++) {  
 printf("%d. %d %c %d = %d\n",   
 i + 1,   
 history[i].operand1,   
 history[i].operation,  
 history[i].operand2,  
 history[i].result);  
 }  
}  
  
/\* 統計情報を表示する関数 \*/  
void print\_statistics(CalculationHistory \*history, int count)  
{  
 int i;  
 int operation\_count[MAX\_OPERATIONS] = {0}; /\* 各演算の回数 \*/  
 char operations[] = {'+', '-', '\*', '/'};  
 char \*operation\_names[] = {"加算", "減算", "乗算", "除算"};  
   
 printf("\n=== 統計情報 ===\n");  
 printf("総計算回数: %d回\n", count);  
   
 /\* 各演算の回数をカウント \*/  
 for (i = 0; i < count; i++) {  
 int j;  
 for (j = 0; j < MAX\_OPERATIONS; j++) {  
 if (history[i].operation == operations[j]) {  
 operation\_count[j]++;  
 break;  
 }  
 }  
 }  
   
 /\* 演算別統計を表示 \*/  
 for (i = 0; i < MAX\_OPERATIONS; i++) {  
 printf("%s (%c): %d回\n",   
 operation\_names[i], operations[i], operation\_count[i]);  
 }  
}  
  
int main(void)  
{  
 /\* 変数の宣言（C90では先頭で宣言） \*/  
 int (\*operations[])(int, int) = {add, subtract, multiply, divide\_safe};  
 char op\_symbols[] = {'+', '-', '\*', '/'};  
 char \*op\_names[] = {"加算", "減算", "乗算", "除算"};  
   
 CalculationHistory history[MAX\_HISTORY];  
 int history\_count = 0;  
   
 /\* デモ用の計算データ \*/  
 int demo\_data[][3] = {  
 {10, 5, 0}, /\* 10 + 5 \*/  
 {15, 3, 1}, /\* 15 - 3 \*/  
 {4, 6, 2}, /\* 4 \* 6 \*/  
 {20, 4, 3}, /\* 20 / 4 \*/  
 {8, 0, 3}, /\* 8 / 0 (ゼロ除算テスト) \*/  
 {7, 2, 2}, /\* 7 \* 2 \*/  
 {100, 25, 1}, /\* 100 - 25 \*/  
 };  
 int num\_demos = sizeof(demo\_data) / sizeof(demo\_data[0]);  
 int i;  
   
 printf("=== 関数ポインタ配列計算機デモ ===\n");  
   
 /\* デモ計算の実行 \*/  
 for (i = 0; i < num\_demos && history\_count < MAX\_HISTORY; i++) {  
 int a = demo\_data[i][0];  
 int b = demo\_data[i][1];  
 int op\_index = demo\_data[i][2];  
   
 if (op\_index >= 0 && op\_index < MAX\_OPERATIONS) {  
 int result = operations[op\_index](a, b);  
 char op\_char = op\_symbols[op\_index];  
   
 printf("計算 %d: %d %c %d = %d (%s)\n",   
 i + 1, a, op\_char, b, result, op\_names[op\_index]);  
   
 /\* 履歴に記録 \*/  
 history[history\_count].operand1 = a;  
 history[history\_count].operand2 = b;  
 history[history\_count].operation = op\_char;  
 history[history\_count].result = result;  
 history\_count++;  
 }  
 }  
   
 /\* 履歴表示 \*/  
 print\_history(history, history\_count);  
   
 /\* 統計情報表示 \*/  
 print\_statistics(history, history\_count);  
   
 /\* 関数ポインタ配列の活用例 \*/  
 printf("\n=== 関数ポインタ配列の特徴デモ ===\n");  
 {  
 int test\_a = 12;  
 int test\_b = 3;  
   
 printf("値 %d と %d での全演算結果:\n", test\_a, test\_b);  
 for (i = 0; i < MAX\_OPERATIONS; i++) {  
 int result = operations[i](test\_a, test\_b);  
 printf(" %s: %d %c %d = %d\n",   
 op\_names[i], test\_a, op\_symbols[i], test\_b, result);  
 }  
 }  
   
 /\* 連続計算のデモ \*/  
 printf("\n=== 連続計算デモ ===\n");  
 {  
 int value = 10;  
 int operands[] = {2, 3, 4, 2};  
 int ops[] = {0, 2, 1, 3}; /\* +, \*, -, / \*/  
   
 printf("初期値: %d\n", value);  
 for (i = 0; i < 4; i++) {  
 int old\_value = value;  
 value = operations[ops[i]](value, operands[i]);  
 printf("ステップ %d: %d %c %d = %d\n",   
 i + 1, old\_value, op\_symbols[ops[i]], operands[i], value);  
 }  
 printf("最終結果: %d\n", value);  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 関数ポインタ配列による動的関数選択  
2. 構造体を使った履歴管理  
3. 統計情報の収集と表示  
4. 連続計算での関数ポインタ活用  
  
実装のポイント:  
- C90準拠のため、すべての変数を関数先頭で宣言  
- 関数ポインタ配列の初期化と活用  
- 構造体配列による履歴管理  
- デモ版として予め定義された計算パターンを使用  
- ゼロ除算エラーのハンドリング  
\*/