# 第8章 文字列処理 - 演習問題

# 演習問題

この章では、C言語における文字列操作の基本から応用まで、段階的に学習できる演習問題を提供しています。

## 演習の概要

文字列処理は、プログラミングにおいて最も基本的かつ重要な技術の一つです。この演習では、以下の3つの段階で文字列操作を習得します。

### 学習目標

1. **基本的な文字列操作の理解**
   * null終端文字列の概念
   * ポインタと配列の関係
   * メモリ安全な文字列操作
2. **文字列配列の管理**
   * 二次元配列の操作
   * 動的な要素の追加・削除
   * ソートと検索アルゴリズム
3. **高度な文字列処理**
   * パターンマッチング
   * アルゴリズムの最適化
   * 実用的な応用例

## 演習問題一覧

### 演習8-1: 基本的な文字列操作

**難易度**: 初級  
**所要時間**: 2-3時間

標準ライブラリの関数を使わずに、基本的な文字列操作関数を実装します。

**実装する関数**: 1. my\_strlen() - 文字列の長さを計算 2. my\_strcpy() - 文字列をコピー 3. my\_strcmp() - 文字列を比較 4. my\_strcat() - 文字列を連結 5. my\_strchr() - 文字を検索 6. my\_strrchr() - 文字を後方検索 7. count\_char() - 特定文字をカウント

**学習ポイント**: - null終端文字列の仕組み - ポインタ演算の基礎 - バッファオーバーフロー対策 - エラーハンドリング

### 演習8-2: 文字列配列の管理

**難易度**: 中級  
**所要時間**: 3-4時間

学生名を格納する文字列配列の管理システムを実装します。

**実装する機能**: 1. **配列管理機能** - 学生名の追加・削除 - 重複チェック - 配列の動的管理

1. **検索とソート機能**
   * 名前による検索
   * アルファベット順ソート
   * 一覧表示機能

**学習ポイント**: - 二次元配列の操作 - 配列要素の挿入・削除 - バブルソートアルゴリズム - 線形検索の実装

### 演習8-3: 高度な文字列検索

**難易度**: 上級  
**所要時間**: 4-6時間

標準ライブラリを使わずに、高度な文字列検索機能を実装します。

**実装する機能**: 1. **検索機能** - 大小文字を区別しない検索 - 単語境界での検索 - 部分文字列検索

1. **パターンマッチング**
   * ワイルドカード検索（\*、?）
   * 複数パターンの同時検索
   * 検索結果の詳細情報

**学習ポイント**: - 高度な文字列マッチングアルゴリズム - メモリ効率的な実装 - 動的プログラミング - パフォーマンス最適化

## 提出要件

各演習について、以下の要件を満たしてください：

### 必須要件

1. **ソースコード**
   * 適切なコメント（日本語）
   * 関数の詳細な説明
   * エラーハンドリング
2. **テストプログラム**
   * 各関数の動作確認
   * 境界値テスト
   * エラーケースのテスト
3. **実行例**
   * 正常ケースの実行結果
   * エラーケースの実行結果

### 推奨要件

1. **コーディングスタイル**
   * 一貫したインデント
   * 意味のある変数名
   * 適切な関数分割
2. **安全性**
   * バッファオーバーフロー対策
   * NULLポインタチェック
   * メモリリークの防止

## 評価基準

### 正確性 (40%)

* 仕様通りに動作するか
* エラーケースが適切に処理されるか
* テストケースが網羅的か

### 安全性 (30%)

* バッファオーバーフローがないか
* メモリ管理が適切か
* 入力検証が十分か

### 効率性 (20%)

* アルゴリズムの計算量が適切か
* 不要な処理がないか
* メモリ使用量が最適か

### 可読性 (10%)

* コードが理解しやすいか
* コメントが適切か
* 命名規則が一貫しているか

## 学習の進め方

### 推奨手順

1. **基礎の理解**
   * 文字列の基本概念を復習
   * ポインタと配列の関係を確認
   * null終端文字列の仕組みを理解
2. **段階的実装**
   * 演習1から順番に取り組む
   * 小さな関数から実装開始
   * テストケースを先に作成
3. **デバッグとテスト**
   * コンパイル時の警告を確認
   * 実行時エラーをチェック
   * メモリ使用状況を監視
4. **最適化と改善**
   * パフォーマンスを測定
   * コードの可読性を向上
   * エラーハンドリングを強化

### 学習のヒント

1. **安全第一**
   * 配列の境界を常に確認
   * NULLポインタをチェック
   * バッファサイズを適切に管理
2. **テスト駆動**
   * 関数作成前にテストケースを考える
   * エッジケースを忘れずに
   * 段階的に機能を拡張
3. **効率性の考慮**
   * 不要な文字列コピーを避ける
   * ポインタ操作を活用
   * メモリ使用量を最適化

## トラブルシューティング

### よくある問題

1. **セグメンテーション違反**
   * 配列の境界越え
   * null終端の忘れ
   * 未初期化ポインタ
2. **予期しない出力**
   * null終端文字の扱い
   * 文字コードの問題
   * バッファの不足
3. **メモリリーク**
   * 動的メモリの解放忘れ
   * 重複した確保
   * 例外時の処理不備

### デバッグ方法

1. **printfデバッグ**
   * 変数の値を確認
   * 関数の呼び出し順序を追跡
   * 配列の内容を表示
2. **gdbの使用**
   * ブレークポイントの設定
   * ステップ実行
   * 変数の監視
3. **静的解析ツール**
   * コンパイラの警告
   * lintツールの使用
   * メモリチェッカー

## 参考資料

### 必読資料

* [文字列操作の基礎](../README.md)
* [実装例とサンプルコード](../examples/)
* [解答例](../solutions/)

### 推奨図書

* 『プログラミング言語C 第2版』（K&R）
* 『Cプログラミング専門課程』
* 『セキュアプログラミング講座』

### オンラインリソース

* C言語標準ライブラリリファレンス
* 文字列処理アルゴリズム
* セキュアコーディング標準

これらの演習を通じて、C言語における文字列操作の基礎から応用まで、体系的に学習することができます。各演習に丁寧に取り組み、安全で効率的なコードの作成技術を身につけてください。

# 解答例

このディレクトリには文字列操作に関する演習問題の解答例が含まれています。

## 解答ファイル一覧

### 演習8-1: 基本的な文字列操作関数の実装

標準ライブラリの関数を使わずに、基本的な文字列操作関数を実装する問題です。

* **C90版**: <ex8_1_string_operations.c>
* **C99版**: <ex8_1_string_operations_c99.c>

**実装する関数**: - my\_strlen() - 文字列の長さを計算 - my\_strcpy() - 文字列をコピー - my\_strcmp() - 文字列を比較 - my\_strcat() - 文字列を連結 - my\_strchr() - 文字を検索

**学習ポイント**: - 文字列がnull終端であることの理解 - ポインタと配列の操作 - 文字列処理の基本アルゴリズム

### 演習8-2: 文字列配列の管理システム

学生名を格納する文字列配列の管理システムを実装する問題です。

* **C90版**: <ex8_2_string_arrays.c>
* **C99版**: <ex8_2_string_arrays_c99.c>

**実装する機能**: - 学生名の追加・削除 - 名前による検索 - アルファベット順ソート - 一覧表示

**学習ポイント**: - 二次元配列の操作 - 文字列の動的管理 - 配列要素の挿入・削除 - 文字列ソートアルゴリズム

### 演習8-3: 高度な文字列検索関数

標準ライブラリを使わずに、高度な文字列検索機能を実装する問題です。

* **C90版**: <ex8_3_string_search.c>
* **C99版**: <ex8_3_string_search_c99.c>

**実装する機能**: - 大小文字を区別しない検索 - 単語境界での検索 - ワイルドカード検索（\*、?） - 複数パターンの同時検索

**学習ポイント**: - 高度な文字列マッチングアルゴリズム - パターンマッチング - 検索結果の詳細情報管理 - メモリ効率的な実装

## C90版とC99版の違い

### C90版の特徴

* ANSI C（C90）に準拠
* 従来の配列とポインタ操作
* 関数の先頭での変数宣言
* int型による真偽値表現

### C99版の特徴

* C99の新機能を活用
* **bool型**: 明確な真偽値表現
* **可変長配列（VLA）**: 実行時サイズ決定
* **複合リテラル**: 一時的なデータ構造
* **指定初期化子**: 特定要素の初期化
* **restrict修飾子**: 最適化ヒント
* **inline関数**: 関数呼び出しコスト削減
* **固定幅整数型**: 明確なデータサイズ

## コンパイルと実行

### 基本的なコンパイル

# C90版  
gcc -Wall -Wextra -pedantic -std=c90 ex8\_1\_string\_operations.c -o ex8\_1  
  
# C99版  
gcc -Wall -Wextra -pedantic -std=c99 ex8\_1\_string\_operations\_c99.c -o ex8\_1\_c99

### Makefileを使用

# 全ての解答をコンパイル  
make solutions  
  
# 特定の解答を実行  
make run-ex8-1  
make run-ex8-1-c99

## 学習の進め方

1. **基礎から応用へ**: 演習8-1から順番に取り組む
2. **C90版を先に学習**: 基本的な実装を理解
3. **C99版で機能拡張**: 新しい言語機能を活用
4. **コードの比較**: 両バージョンの違いを理解
5. **実行とテスト**: 実際に動かして動作を確認

## 各解答の詳細説明

### 演習8-1: 基本的な文字列操作

**C90版の重要ポイント**: - 基本的なポインタ操作 - null終端文字列の理解 - エラーハンドリングの基礎

**C99版の追加機能**: - restrict修飾子による最適化 - bool型による明確な戻り値 - inline関数による性能向上 - 詳細な統計情報と分析機能

### 演習8-2: 文字列配列管理

**C90版の重要ポイント**: - 二次元配列の操作 - 配列要素の動的管理 - 基本的なソートアルゴリズム

**C99版の追加機能**: - 可変長配列による柔軟な処理 - 構造体による詳細情報管理 - 複数の操作モード（検索・削除） - 統計分析機能

### 演習8-3: 高度な文字列検索

**C90版の重要ポイント**: - 基本的なパターンマッチング - 文字列検索アルゴリズム - メモリ効率的な実装

**C99版の追加機能**: - 動的プログラミングによるワイルドカード検索 - 複数パターン同時検索 - 詳細な検索結果情報 - 可変長配列によるメモリ最適化

## 実践的な応用例

これらの解答では、以下のような実践的な応用例も示しています：

1. **ログ解析システム** - 文字列検索を使用
2. **学生管理システム** - 配列操作を使用
3. **文字列統計分析** - 基本操作を組み合わせ
4. **パフォーマンステスト** - 大量データでの動作確認

## トラブルシューティング

### よくある問題

1. **セグメンテーション違反**
   * null終端の確認
   * 配列の境界チェック
   * ポインタの初期化
2. **コンパイルエラー**
   * 標準の指定（-std=c90 または -std=c99）
   * 必要なヘッダファイルのインクルード
3. **予期しない動作**
   * デバッグ情報の有効化（-gフラグ）
   * テストケースでの段階的確認

## 参考資料

* [C言語の文字列操作基礎](../README.md)
* [演習問題の詳細](../exercises/README.md)
* [実行例とサンプルコード](../examples/)

これらの解答例を通じて、C言語における文字列操作の基礎から応用まで、体系的に学習することができます。C90版で基本を固め、C99版で新しい機能と最適化テクニックを身につけてください。 ## ex8\_1\_string\_operations.c

/\*  
 \* ファイル名: ex8\_1\_string\_operations.c  
 \* 演習8-1: 基本的な文字列操作  
 \* 説明: 標準ライブラリを使わずに文字列操作関数を実装  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
#include <stddef.h>  
  
/\* 文字列の長さを計算する関数 \*/  
int my\_strlen(const char str[])  
{  
 int length = 0;  
   
 if (str == NULL) {  
 return 0;  
 }  
   
 while (str[length] != '\0') {  
 length++;  
 }  
   
 return length;  
}  
  
/\* 文字列をコピーする関数 \*/  
void my\_strcpy(char dest[], const char src[])  
{  
 int i = 0;  
   
 if (dest == NULL || src == NULL) {  
 return;  
 }  
   
 while (src[i] != '\0') {  
 dest[i] = src[i];  
 i++;  
 }  
 dest[i] = '\0'; /\* 終端文字をコピー \*/  
}  
  
/\* 文字列を比較する関数 \*/  
int my\_strcmp(const char str1[], const char str2[])  
{  
 int i = 0;  
   
 if (str1 == NULL && str2 == NULL) {  
 return 0;  
 }  
 if (str1 == NULL) {  
 return -1;  
 }  
 if (str2 == NULL) {  
 return 1;  
 }  
   
 while (str1[i] != '\0' && str2[i] != '\0') {  
 if (str1[i] < str2[i]) {  
 return -1;  
 } else if (str1[i] > str2[i]) {  
 return 1;  
 }  
 i++;  
 }  
   
 /\* どちらかが終了した場合 \*/  
 if (str1[i] == '\0' && str2[i] == '\0') {  
 return 0; /\* 等しい \*/  
 } else if (str1[i] == '\0') {  
 return -1; /\* str1が短い \*/  
 } else {  
 return 1; /\* str2が短い \*/  
 }  
}  
  
/\* 文字列を連結する関数 \*/  
void my\_strcat(char dest[], const char src[])  
{  
 int dest\_len = 0;  
 int i = 0;  
   
 if (dest == NULL || src == NULL) {  
 return;  
 }  
   
 /\* 連結先の終端文字を見つける \*/  
 while (dest[dest\_len] != '\0') {  
 dest\_len++;  
 }  
   
 /\* 連結元をコピー \*/  
 while (src[i] != '\0') {  
 dest[dest\_len + i] = src[i];  
 i++;  
 }  
   
 /\* 終端文字を追加 \*/  
 dest[dest\_len + i] = '\0';  
}  
  
/\* 文字列内で指定した文字を検索する関数 \*/  
char \*my\_strchr(const char str[], int ch)  
{  
 int i = 0;  
   
 if (str == NULL) {  
 return NULL;  
 }  
   
 while (str[i] != '\0') {  
 if (str[i] == ch) {  
 return (char \*)&str[i]; /\* 見つかった位置を返す \*/  
 }  
 i++;  
 }  
   
 /\* 終端文字を検索する場合 \*/  
 if (ch == '\0') {  
 return (char \*)&str[i];  
 }  
   
 return NULL; /\* 見つからなかった \*/  
}  
  
/\* 文字列内で指定した文字を最後から検索する関数 \*/  
char \*my\_strrchr(const char str[], int ch)  
{  
 int i;  
 int length;  
   
 if (str == NULL) {  
 return NULL;  
 }  
   
 length = my\_strlen(str);  
   
 /\* 後ろから検索 \*/  
 for (i = length; i >= 0; i--) {  
 if (str[i] == ch) {  
 return (char \*)&str[i];  
 }  
 }  
   
 return NULL; /\* 見つからなかった \*/  
}  
  
/\* 文字列内の特定文字をカウントする関数 \*/  
int count\_char(const char str[], char ch)  
{  
 int count = 0;  
 int i = 0;  
   
 if (str == NULL) {  
 return 0;  
 }  
   
 while (str[i] != '\0') {  
 if (str[i] == ch) {  
 count++;  
 }  
 i++;  
 }  
   
 return count;  
}  
  
/\* テスト関数 \*/  
void test\_string\_functions(void)  
{  
 char buffer[100];  
 char \*result;  
   
 printf("=== 自作文字列関数のテスト ===\n");  
   
 /\* my\_strlen のテスト \*/  
 printf("\n1. my\_strlen のテスト:\n");  
 printf("my\_strlen(\"Hello\"): %d\n", my\_strlen("Hello"));  
 printf("my\_strlen(\"\"): %d\n", my\_strlen(""));  
 printf("my\_strlen(\"Hello World\"): %d\n", my\_strlen("Hello World"));  
   
 /\* my\_strcpy のテスト \*/  
 printf("\n2. my\_strcpy のテスト:\n");  
 my\_strcpy(buffer, "Hello");  
 printf("コピー結果: \"%s\"\n", buffer);  
   
 /\* my\_strcmp のテスト \*/  
 printf("\n3. my\_strcmp のテスト:\n");  
 printf("my\_strcmp(\"apple\", \"apple\"): %d\n", my\_strcmp("apple", "apple"));  
 printf("my\_strcmp(\"apple\", \"banana\"): %d\n", my\_strcmp("apple", "banana"));  
 printf("my\_strcmp(\"banana\", \"apple\"): %d\n", my\_strcmp("banana", "apple"));  
   
 /\* my\_strcat のテスト \*/  
 printf("\n4. my\_strcat のテスト:\n");  
 my\_strcpy(buffer, "Hello");  
 my\_strcat(buffer, " World");  
 printf("連結結果: \"%s\"\n", buffer);  
   
 /\* my\_strchr のテスト \*/  
 printf("\n5. my\_strchr のテスト:\n");  
 result = my\_strchr("Hello World", 'W');  
 if (result != NULL) {  
 printf("'W' found at position: %ld\n", result - "Hello World");  
 }  
   
 result = my\_strchr("Hello World", 'x');  
 if (result == NULL) {  
 printf("'x' not found\n");  
 }  
   
 /\* my\_strrchr のテスト \*/  
 printf("\n6. my\_strrchr のテスト:\n");  
 result = my\_strrchr("Hello World", 'l');  
 if (result != NULL) {  
 printf("Last 'l' found at position: %ld\n", result - "Hello World");  
 }  
   
 /\* count\_char のテスト \*/  
 printf("\n7. count\_char のテスト:\n");  
 printf("count\_char(\"Hello World\", 'l'): %d\n", count\_char("Hello World", 'l'));  
 printf("count\_char(\"Hello World\", 'o'): %d\n", count\_char("Hello World", 'o'));  
 printf("count\_char(\"Hello World\", 'x'): %d\n", count\_char("Hello World", 'x'));  
}  
  
int main(void)  
{  
 printf("===== 基本的な文字列操作の実装 =====\n");  
   
 test\_string\_functions();  
   
 printf("\n=== 実用例 ===\n");  
   
 /\* 実用例: 文字列の解析 \*/  
 char text[] = "C programming is fun and challenging!";  
 printf("文字列: \"%s\"\n", text);  
 printf("長さ: %d文字\n", my\_strlen(text));  
 printf("'n'の数: %d個\n", count\_char(text, 'n'));  
 printf("スペースの数: %d個\n", count\_char(text, ' '));  
   
 /\* 'a'の最初と最後の位置 \*/  
 char \*first\_a = my\_strchr(text, 'a');  
 char \*last\_a = my\_strrchr(text, 'a');  
   
 if (first\_a != NULL) {  
 printf("最初の'a'の位置: %ld\n", first\_a - text);  
 }  
 if (last\_a != NULL) {  
 printf("最後の'a'の位置: %ld\n", last\_a - text);  
 }  
   
 /\* 文字列の組み立て \*/  
 char result[200] = "";  
 my\_strcpy(result, "Hello");  
 my\_strcat(result, ", ");  
 my\_strcat(result, "World");  
 my\_strcat(result, "!");  
 printf("\n組み立てた文字列: \"%s\"\n", result);  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 標準ライブラリの再実装:  
 - 関数の内部動作の理解  
 - ポインタ操作の練習  
 - エッジケースの処理  
  
2. エラーハンドリング:  
 - NULLポインタのチェック  
 - 境界条件の処理  
 - 適切な戻り値の設定  
  
3. メモリ安全性:  
 - バッファオーバーフローの防止  
 - 終端文字の適切な処理  
 - ポインタの有効性確認  
  
4. 実用的な応用:  
 - 文字列解析  
 - テキスト処理  
 - データ検索  
  
注意点:  
- ポインタ演算時の型安全性  
- 文字列の不変性の維持  
- 効率的なアルゴリズムの選択  
\*/

## ex8\_2\_string\_arrays.c

/\*  
 \* ファイル名: ex8\_2\_string\_arrays.c  
 \* 演習8-2: 文字列配列の管理  
 \* 説明: 学生名を格納する文字列配列の追加・削除・表示機能  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
  
#define MAX\_STUDENTS 10  
#define MAX\_NAME\_LENGTH 50  
  
/\* 学生名配列の表示 \*/  
void display\_students(char students[][MAX\_NAME\_LENGTH], int count)  
{  
 int i;  
   
 if (count == 0) {  
 printf("登録されている学生はいません。\n");  
 return;  
 }  
   
 printf("=== 学生一覧 ===\n");  
 for (i = 0; i < count; i++) {  
 printf("%d. %s\n", i + 1, students[i]);  
 }  
 printf("合計: %d名\n", count);  
}  
  
/\* 学生の追加 \*/  
int add\_student(char students[][MAX\_NAME\_LENGTH], int \*count, const char \*name)  
{  
 if (\*count >= MAX\_STUDENTS) {  
 printf("エラー: これ以上学生を追加できません（最大%d名）\n", MAX\_STUDENTS);  
 return 0; /\* 失敗 \*/  
 }  
   
 if (strlen(name) >= MAX\_NAME\_LENGTH) {  
 printf("エラー: 名前が長すぎます（最大%d文字）\n", MAX\_NAME\_LENGTH - 1);  
 return 0; /\* 失敗 \*/  
 }  
   
 /\* 重複チェック \*/  
 {  
 int i;  
 for (i = 0; i < \*count; i++) {  
 if (strcmp(students[i], name) == 0) {  
 printf("エラー: \"%s\" は既に登録されています\n", name);  
 return 0; /\* 失敗 \*/  
 }  
 }  
 }  
   
 strcpy(students[\*count], name);  
 (\*count)++;  
 printf("\"%s\" を追加しました\n", name);  
 return 1; /\* 成功 \*/  
}  
  
/\* 学生の削除（名前で検索） \*/  
int remove\_student\_by\_name(char students[][MAX\_NAME\_LENGTH], int \*count, const char \*name)  
{  
 int i, found\_index = -1;  
   
 /\* 削除対象を検索 \*/  
 for (i = 0; i < \*count; i++) {  
 if (strcmp(students[i], name) == 0) {  
 found\_index = i;  
 break;  
 }  
 }  
   
 if (found\_index == -1) {  
 printf("エラー: \"%s\" は見つかりませんでした\n", name);  
 return 0; /\* 失敗 \*/  
 }  
   
 /\* 要素を前に詰める \*/  
 for (i = found\_index; i < \*count - 1; i++) {  
 strcpy(students[i], students[i + 1]);  
 }  
   
 (\*count)--;  
 printf("\"%s\" を削除しました\n", name);  
 return 1; /\* 成功 \*/  
}  
  
/\* 学生の削除（インデックス指定） \*/  
int remove\_student\_by\_index(char students[][MAX\_NAME\_LENGTH], int \*count, int index)  
{  
 int i;  
 char name[MAX\_NAME\_LENGTH];  
   
 if (index < 0 || index >= \*count) {  
 printf("エラー: 無効なインデックスです（1-%d の範囲で指定してください）\n", \*count);  
 return 0; /\* 失敗 \*/  
 }  
   
 /\* 削除前に名前を保存 \*/  
 strcpy(name, students[index]);  
   
 /\* 要素を前に詰める \*/  
 for (i = index; i < \*count - 1; i++) {  
 strcpy(students[i], students[i + 1]);  
 }  
   
 (\*count)--;  
 printf("%d番目の学生 \"%s\" を削除しました\n", index + 1, name);  
 return 1; /\* 成功 \*/  
}  
  
/\* 学生の検索 \*/  
int search\_student(char students[][MAX\_NAME\_LENGTH], int count, const char \*name)  
{  
 int i;  
   
 for (i = 0; i < count; i++) {  
 if (strcmp(students[i], name) == 0) {  
 return i; /\* 見つかった場合のインデックス \*/  
 }  
 }  
   
 return -1; /\* 見つからない場合 \*/  
}  
  
/\* 学生名のソート（アルファベット順） \*/  
void sort\_students(char students[][MAX\_NAME\_LENGTH], int count)  
{  
 int i, j;  
 char temp[MAX\_NAME\_LENGTH];  
   
 /\* バブルソート \*/  
 for (i = 0; i < count - 1; i++) {  
 for (j = 0; j < count - 1 - i; j++) {  
 if (strcmp(students[j], students[j + 1]) > 0) {  
 strcpy(temp, students[j]);  
 strcpy(students[j], students[j + 1]);  
 strcpy(students[j + 1], temp);  
 }  
 }  
 }  
   
 printf("学生名をアルファベット順にソートしました\n");  
}  
  
/\* 部分一致検索 \*/  
void search\_partial(char students[][MAX\_NAME\_LENGTH], int count, const char \*partial)  
{  
 int i;  
 int found\_count = 0;  
   
 printf("「%s」を含む学生名:\n", partial);  
   
 for (i = 0; i < count; i++) {  
 if (strstr(students[i], partial) != NULL) {  
 printf(" %d. %s\n", i + 1, students[i]);  
 found\_count++;  
 }  
 }  
   
 if (found\_count == 0) {  
 printf(" 該当する学生はいません\n");  
 } else {  
 printf(" %d名が見つかりました\n", found\_count);  
 }  
}  
  
int main(void)  
{  
 char students[MAX\_STUDENTS][MAX\_NAME\_LENGTH];  
 int student\_count = 0;  
 int search\_index;  
   
 printf("===== 学生名管理システム =====\n\n");  
   
 /\* 初期データの追加 \*/  
 printf("1. 学生の追加:\n");  
 add\_student(students, &student\_count, "田中太郎");  
 add\_student(students, &student\_count, "佐藤花子");  
 add\_student(students, &student\_count, "鈴木一郎");  
 add\_student(students, &student\_count, "高橋二郎");  
 add\_student(students, &student\_count, "伊藤三子");  
   
 printf("\n");  
 display\_students(students, student\_count);  
   
 /\* 重複追加のテスト \*/  
 printf("\n2. 重複追加のテスト:\n");  
 add\_student(students, &student\_count, "田中太郎"); /\* 既に存在 \*/  
   
 /\* 検索のテスト \*/  
 printf("\n3. 学生の検索:\n");  
 search\_index = search\_student(students, student\_count, "佐藤花子");  
 if (search\_index != -1) {  
 printf("\"佐藤花子\" は %d番目にいます\n", search\_index + 1);  
 }  
   
 search\_index = search\_student(students, student\_count, "山田太郎");  
 if (search\_index == -1) {  
 printf("\"山田太郎\" は見つかりませんでした\n");  
 }  
   
 /\* 部分一致検索 \*/  
 printf("\n4. 部分一致検索:\n");  
 search\_partial(students, student\_count, "太郎");  
 search\_partial(students, student\_count, "田");  
   
 /\* ソート \*/  
 printf("\n5. ソート:\n");  
 sort\_students(students, student\_count);  
 display\_students(students, student\_count);  
   
 /\* 削除のテスト \*/  
 printf("\n6. 学生の削除:\n");  
 remove\_student\_by\_name(students, &student\_count, "佐藤花子");  
 display\_students(students, student\_count);  
   
 printf("\n");  
 remove\_student\_by\_index(students, &student\_count, 0); /\* 最初の学生を削除 \*/  
 display\_students(students, student\_count);  
   
 /\* エラーケースのテスト \*/  
 printf("\n7. エラーケースのテスト:\n");  
 remove\_student\_by\_name(students, &student\_count, "存在しない学生");  
 remove\_student\_by\_index(students, &student\_count, 100); /\* 無効なインデックス \*/  
   
 /\* 最終状態 \*/  
 printf("\n=== 最終状態 ===\n");  
 display\_students(students, student\_count);  
   
 /\* 統計情報 \*/  
 printf("\n=== 統計情報 ===\n");  
 printf("現在の学生数: %d名\n", student\_count);  
 printf("最大収容数: %d名\n", MAX\_STUDENTS);  
 printf("残り収容可能数: %d名\n", MAX\_STUDENTS - student\_count);  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 配列の動的管理:  
 - 要素の追加・削除  
 - 配列の圧縮（詰める処理）  
 - 容量管理  
  
2. 文字列配列の操作:  
 - strcpy()による安全なコピー  
 - strcmp()による比較  
 - strstr()による部分一致検索  
  
3. エラーハンドリング:  
 - 境界チェック  
 - 重複チェック  
 - 戻り値による成功/失敗の通知  
  
4. 実用的な機能:  
 - 検索機能（完全一致・部分一致）  
 - ソート機能  
 - 統計情報の表示  
  
5. データ構造の設計:  
 - 配列サイズの管理  
 - インデックスの有効性確認  
 - 一貫性のあるAPI設計  
  
注意点:  
- バッファオーバーフローの防止  
- 配列の境界チェック  
- メモリ効率的な削除処理  
- ユーザビリティの向上  
\*/

## ex8\_3\_string\_search.c

/\*  
 \* ファイル名: ex8\_3\_string\_search.c  
 \* 演習8-3: 高度な文字列検索  
 \* 説明: 文字列配列から特定の文字列を検索する高度な機能  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
#include <ctype.h>  
  
#define MAX\_STRINGS 20  
#define MAX\_LENGTH 100  
  
/\* 完全一致検索 \*/  
int exact\_search(char strings[][MAX\_LENGTH], int count, const char \*target)  
{  
 int i;  
   
 for (i = 0; i < count; i++) {  
 if (strcmp(strings[i], target) == 0) {  
 return i; /\* 見つかった位置 \*/  
 }  
 }  
   
 return -1; /\* 見つからない \*/  
}  
  
/\* 大文字小文字を区別しない検索 \*/  
int case\_insensitive\_search(char strings[][MAX\_LENGTH], int count, const char \*target)  
{  
 int i, j;  
 char str\_lower[MAX\_LENGTH];  
 char target\_lower[MAX\_LENGTH];  
   
 /\* ターゲットを小文字に変換 \*/  
 for (j = 0; target[j] != '\0' && j < MAX\_LENGTH - 1; j++) {  
 target\_lower[j] = tolower((unsigned char)target[j]);  
 }  
 target\_lower[j] = '\0';  
   
 for (i = 0; i < count; i++) {  
 /\* 比較対象の文字列を小文字に変換 \*/  
 for (j = 0; strings[i][j] != '\0' && j < MAX\_LENGTH - 1; j++) {  
 str\_lower[j] = tolower((unsigned char)strings[i][j]);  
 }  
 str\_lower[j] = '\0';  
   
 if (strcmp(str\_lower, target\_lower) == 0) {  
 return i;  
 }  
 }  
   
 return -1;  
}  
  
/\* 部分一致検索（すべての該当項目を表示） \*/  
int partial\_search(char strings[][MAX\_LENGTH], int count, const char \*partial,   
 int results[], int max\_results)  
{  
 int i;  
 int found\_count = 0;  
   
 for (i = 0; i < count && found\_count < max\_results; i++) {  
 if (strstr(strings[i], partial) != NULL) {  
 results[found\_count] = i;  
 found\_count++;  
 }  
 }  
   
 return found\_count;  
}  
  
/\* 前方一致検索 \*/  
int prefix\_search(char strings[][MAX\_LENGTH], int count, const char \*prefix,  
 int results[], int max\_results)  
{  
 int i;  
 int found\_count = 0;  
 int prefix\_len = strlen(prefix);  
   
 for (i = 0; i < count && found\_count < max\_results; i++) {  
 if (strncmp(strings[i], prefix, prefix\_len) == 0) {  
 results[found\_count] = i;  
 found\_count++;  
 }  
 }  
   
 return found\_count;  
}  
  
/\* 後方一致検索 \*/  
int suffix\_search(char strings[][MAX\_LENGTH], int count, const char \*suffix,  
 int results[], int max\_results)  
{  
 int i;  
 int found\_count = 0;  
 int suffix\_len = strlen(suffix);  
   
 for (i = 0; i < count && found\_count < max\_results; i++) {  
 int str\_len = strlen(strings[i]);  
   
 if (str\_len >= suffix\_len) {  
 if (strcmp(strings[i] + str\_len - suffix\_len, suffix) == 0) {  
 results[found\_count] = i;  
 found\_count++;  
 }  
 }  
 }  
   
 return found\_count;  
}  
  
/\* 正規表現風のワイルドカード検索（簡易版） \*/  
int wildcard\_search(const char \*str, const char \*pattern)  
{  
 const char \*s = str;  
 const char \*p = pattern;  
   
 while (\*p != '\0') {  
 if (\*p == '\*') {  
 /\* ワイルドカード：0文字以上の任意の文字 \*/  
 p++; /\* '\*'をスキップ \*/  
   
 if (\*p == '\0') {  
 return 1; /\* パターンの最後が'\*'なら一致 \*/  
 }  
   
 /\* 次の文字が一致する位置を探す \*/  
 while (\*s != '\0') {  
 if (wildcard\_search(s, p)) {  
 return 1;  
 }  
 s++;  
 }  
 return 0;  
 } else if (\*p == '?') {  
 /\* 任意の1文字 \*/  
 if (\*s == '\0') {  
 return 0; /\* 文字列が足りない \*/  
 }  
 s++;  
 p++;  
 } else {  
 /\* 通常文字 \*/  
 if (\*s != \*p) {  
 return 0;  
 }  
 s++;  
 p++;  
 }  
 }  
   
 return (\*s == '\0'); /\* 両方とも終端に達していれば一致 \*/  
}  
  
/\* ワイルドカード検索（配列用） \*/  
int wildcard\_search\_array(char strings[][MAX\_LENGTH], int count, const char \*pattern,  
 int results[], int max\_results)  
{  
 int i;  
 int found\_count = 0;  
   
 for (i = 0; i < count && found\_count < max\_results; i++) {  
 if (wildcard\_search(strings[i], pattern)) {  
 results[found\_count] = i;  
 found\_count++;  
 }  
 }  
   
 return found\_count;  
}  
  
/\* 検索結果の表示 \*/  
void display\_search\_results(char strings[][MAX\_LENGTH], int results[], int count,  
 const char \*search\_type, const char \*query)  
{  
 int i;  
   
 printf("\n=== %s検索結果: \"%s\" ===\n", search\_type, query);  
   
 if (count == 0) {  
 printf("該当する項目はありません。\n");  
 return;  
 }  
   
 printf("%d件が見つかりました:\n", count);  
 for (i = 0; i < count; i++) {  
 printf(" %d. [%d] %s\n", i + 1, results[i] + 1, strings[results[i]]);  
 }  
}  
  
/\* 文字列の統計情報を表示 \*/  
void display\_statistics(char strings[][MAX\_LENGTH], int count, const char \*query)  
{  
 int i;  
 int total\_length = 0;  
 int min\_length = MAX\_LENGTH;  
 int max\_length = 0;  
 int query\_count = 0;  
   
 printf("\n=== 統計情報 ===\n");  
   
 for (i = 0; i < count; i++) {  
 int len = strlen(strings[i]);  
 total\_length += len;  
   
 if (len < min\_length) min\_length = len;  
 if (len > max\_length) max\_length = len;  
   
 if (strstr(strings[i], query) != NULL) {  
 query\_count++;  
 }  
 }  
   
 printf("総項目数: %d\n", count);  
 printf("平均文字数: %.1f\n", (double)total\_length / count);  
 printf("最短文字数: %d\n", min\_length);  
 printf("最長文字数: %d\n", max\_length);  
 printf("\"%s\"を含む項目数: %d (%.1f%%)\n",   
 query, query\_count, (double)query\_count \* 100 / count);  
}  
  
int main(void)  
{  
 char data[MAX\_STRINGS][MAX\_LENGTH] = {  
 "Apple", "Banana", "Cherry", "Date", "Elderberry",  
 "Fig", "Grape", "apple", "BANANA", "cherry pie",  
 "Apple juice", "grape fruit", "Date palm", "Elder tree",  
 "Fig leaf", "Cherry blossom", "Banana split", "Apple pie",  
 "Grape wine", "Date sugar"  
 };  
 int data\_count = 20;  
 int results[MAX\_STRINGS];  
 int found\_count;  
 int exact\_index;  
   
 printf("===== 高度な文字列検索システム =====\n");  
   
 /\* データ一覧の表示 \*/  
 printf("\n=== データ一覧 ===\n");  
 {  
 int i;  
 for (i = 0; i < data\_count; i++) {  
 printf("%2d. %s\n", i + 1, data[i]);  
 }  
 }  
   
 /\* 1. 完全一致検索 \*/  
 printf("\n1. 完全一致検索:\n");  
 exact\_index = exact\_search(data, data\_count, "Apple");  
 if (exact\_index != -1) {  
 printf("\"Apple\" は %d番目にあります: %s\n",   
 exact\_index + 1, data[exact\_index]);  
 }  
   
 exact\_index = exact\_search(data, data\_count, "apple");  
 if (exact\_index != -1) {  
 printf("\"apple\" は %d番目にあります: %s\n",   
 exact\_index + 1, data[exact\_index]);  
 }  
   
 /\* 2. 大文字小文字を区別しない検索 \*/  
 printf("\n2. 大文字小文字を区別しない検索:\n");  
 exact\_index = case\_insensitive\_search(data, data\_count, "APPLE");  
 if (exact\_index != -1) {  
 printf("\"APPLE\"（大小文字無視）は %d番目にあります: %s\n",   
 exact\_index + 1, data[exact\_index]);  
 }  
   
 /\* 3. 部分一致検索 \*/  
 found\_count = partial\_search(data, data\_count, "Apple", results, MAX\_STRINGS);  
 display\_search\_results(data, results, found\_count, "部分一致", "Apple");  
   
 /\* 4. 前方一致検索 \*/  
 found\_count = prefix\_search(data, data\_count, "Apple", results, MAX\_STRINGS);  
 display\_search\_results(data, results, found\_count, "前方一致", "Apple");  
   
 /\* 5. 後方一致検索 \*/  
 found\_count = suffix\_search(data, data\_count, "pie", results, MAX\_STRINGS);  
 display\_search\_results(data, results, found\_count, "後方一致", "pie");  
   
 /\* 6. ワイルドカード検索 \*/  
 printf("\n=== ワイルドカード検索のテスト ===\n");  
   
 /\* \*ple パターン \*/  
 found\_count = wildcard\_search\_array(data, data\_count, "\*ple", results, MAX\_STRINGS);  
 display\_search\_results(data, results, found\_count, "ワイルドカード", "\*ple");  
   
 /\* ?ate パターン \*/  
 found\_count = wildcard\_search\_array(data, data\_count, "?ate", results, MAX\_STRINGS);  
 display\_search\_results(data, results, found\_count, "ワイルドカード", "?ate");  
   
 /\* \*e\* パターン \*/  
 found\_count = wildcard\_search\_array(data, data\_count, "\*e\*", results, MAX\_STRINGS);  
 display\_search\_results(data, results, found\_count, "ワイルドカード", "\*e\*");  
   
 /\* 7. 複合検索の例 \*/  
 printf("\n=== 複合検索の例 ===\n");  
 printf("「Apple」で始まる項目:\n");  
 found\_count = prefix\_search(data, data\_count, "Apple", results, MAX\_STRINGS);  
 {  
 int i;  
 for (i = 0; i < found\_count; i++) {  
 printf(" %s\n", data[results[i]]);  
 }  
 }  
   
 /\* 8. 統計情報 \*/  
 display\_statistics(data, data\_count, "e");  
   
 /\* 9. 対話的検索の例 \*/  
 printf("\n=== 対話的検索のデモ ===\n");  
 {  
 char queries[][20] = {"Banana", "tree", "ju\*", "????"};  
 int i;  
   
 for (i = 0; i < 4; i++) {  
 printf("\nクエリ: \"%s\"\n", queries[i]);  
   
 /\* ワイルドカード文字が含まれているかチェック \*/  
 if (strchr(queries[i], '\*') != NULL || strchr(queries[i], '?') != NULL) {  
 found\_count = wildcard\_search\_array(data, data\_count, queries[i],   
 results, MAX\_STRINGS);  
 printf("ワイルドカード検索: %d件\n", found\_count);  
 } else {  
 found\_count = partial\_search(data, data\_count, queries[i],   
 results, MAX\_STRINGS);  
 printf("部分一致検索: %d件\n", found\_count);  
 }  
   
 if (found\_count > 0 && found\_count <= 3) {  
 int j;  
 for (j = 0; j < found\_count; j++) {  
 printf(" - %s\n", data[results[j]]);  
 }  
 }  
 }  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 多様な検索アルゴリズム:  
 - 完全一致・部分一致・前方一致・後方一致  
 - 大文字小文字を区別しない検索  
 - ワイルドカード検索  
  
2. 効率的な実装:  
 - 結果配列による複数結果の処理  
 - 早期終了による最適化  
 - メモリ効率的な検索  
  
3. 実用的な機能:  
 - 統計情報の表示  
 - 検索結果の整理  
 - エラーハンドリング  
  
4. アルゴリズムの応用:  
 - パターンマッチング  
 - 文字列処理の最適化  
 - 再帰的処理  
  
5. ユーザビリティ:  
 - 分かりやすい結果表示  
 - 複数の検索オプション  
 - 対話的な機能  
  
注意点:  
- バッファオーバーフローの防止  
- パターンマッチングの効率性  
- 文字コードの適切な処理  
- エッジケースの考慮  
\*/