# 第14章 C23の新機能（オプション） - 演習問題

# 演習問題

C23の新機能を活用した演習問題です。

## 演習14-1: bool型の活用

bool型を使用して、簡単な状態管理システムを作成してください。

### 要件：

* システムの各種状態をbool型で管理
* 状態の組み合わせによる判定ロジック
* 状態表示関数の実装

### ヒント：

typedef struct {  
 bool is\_connected;  
 bool is\_authenticated;  
 bool has\_permission;  
 bool is\_busy;  
} SystemStatus;

## 演習14-2: ビット操作と2進数リテラル

2進数リテラルを使用して、8ビットのフラグ管理システムを作成してください。

### 要件：

* 各ビットが異なる機能のON/OFFを表す
* ビットの設定、クリア、トグル、チェック関数
* 現在の状態を2進数で表示

### 例：

#define FLAG\_READ 0b00000001  
#define FLAG\_WRITE 0b00000010  
#define FLAG\_EXECUTE 0b00000100  
// ... 続く

## 演習14-3: typeof演算子の応用

typeof演算子を使用して、汎用的なデータ構造操作マクロを作成してください。

### 要件：

* 配列の最大値・最小値を求めるマクロ
* 2つの変数を安全に交換するマクロ
* 配列要素の合計を計算するマクロ

### 例：

#define ARRAY\_MAX(arr, size) ({ \  
 typeof(arr[0]) max = arr[0]; \  
 /\* 実装を完成させる \*/ \  
 max; \  
})

## 演習14-4: nullptr安全プログラミング

nullptrを活用して、安全なリンクリスト操作関数を作成してください。

### 要件：

* ノードの追加、削除、検索
* すべてのポインタ操作でnullptrチェック
* エラーハンドリングの実装

## 演習14-5: C23総合演習

C23の複数の新機能を組み合わせた小規模なプログラムを作成してください。

### テーマ例：

1. **設定管理システム**
   * bool型で各種設定のON/OFF
   * 2進数リテラルでフラグ管理
   * nullptrで安全なポインタ操作
2. **簡易デバッガ**
   * typeof演算子で変数情報取得
   * 2進数表示機能
   * メモリダンプ機能
3. **ビット演算計算機**
   * 2進数リテラルで入力
   * ビット演算の結果表示
   * 教育的な説明付き

## 提出方法

1. 各演習の解答をsolutions/ディレクトリに作成
2. ファイル名: ex14\_1\_status\_system.cのような形式
3. 十分なコメントを含めること

## 評価基準

* C23機能の適切な使用
* コードの可読性
* エラーハンドリング
* 実行時の安定性

# 解答例

各演習問題の解答例と解説です。

## 演習14-1: bool型の活用

**ファイル**: ex14\_1\_status\_system.c

システム状態管理の実装例。bool型を活用して、複数の状態を管理し、それらの組み合わせによる判定を行います。

### 学習ポイント：

* bool型の基本的な使用法
* 構造体でのbool型メンバー
* 論理演算の活用

## 演習14-2: ビット操作と2進数リテラル

**ファイル**: ex14\_2\_bit\_flags.c

2進数リテラルを使った8ビットフラグシステムの実装例。

### 学習ポイント：

* 2進数リテラルの実用的な使用
* ビット演算の基本操作
* マクロでのビット操作

## 演習14-3: typeof演算子の応用

**ファイル**: ex14\_3\_generic\_macros.c

typeof演算子を使った汎用マクロの実装例。

### 学習ポイント：

* typeof演算子による型の取得
* 型安全なマクロの作成
* Statement Expression（GCC拡張）の活用

## 演習14-4: nullptr安全プログラミング

**ファイル**: ex14\_4\_safe\_linkedlist.c

nullptrを使った安全なリンクリスト実装例。

### 学習ポイント：

* nullptrによる明示的なポインタチェック
* エラーハンドリングのベストプラクティス
* メモリ管理の安全性

## 演習14-5: C23総合演習

**ファイル**: ex14\_5\_config\_manager.c

C23の複数機能を組み合わせた設定管理システムの実装例。

### 学習ポイント：

* 複数のC23機能の統合
* 実用的なプログラム設計
* 可読性と保守性の両立

## 解答例の実行方法

# 個別の解答例をコンパイル・実行  
make ex14\_1\_status\_system  
./solutions/ex14\_1\_status\_system  
  
# すべての解答例を実行  
make run-solutions

## 注意事項

* これらは解答例の一つです。他にも様々な実装方法があります
* C23のサポート状況により、一部の機能が動作しない可能性があります
* エラーハンドリングは教育目的で詳細に実装しています ## ex14\_1\_status\_system.c

/\*\*  
 \* 演習14-1: bool型の活用  
 \* システム状態管理プログラム  
 \*   
 \* C23のbool型を使用して、システムの各種状態を管理し、  
 \* 状態の組み合わせによる判定ロジックを実装します。  
 \*/  
  
#include <stdio.h>  
  
/\* システム状態を管理する構造体 \*/  
typedef struct {  
 bool is\_connected; /\* ネットワーク接続状態 \*/  
 bool is\_authenticated; /\* 認証状態 \*/  
 bool has\_permission; /\* アクセス権限 \*/  
 bool is\_busy; /\* ビジー状態 \*/  
} SystemStatus;  
  
/\* 状態を初期化 \*/  
void init\_status(SystemStatus \*status)  
{  
 status->is\_connected = false;  
 status->is\_authenticated = false;  
 status->has\_permission = false;  
 status->is\_busy = false;  
}  
  
/\* 状態を表示 \*/  
void display\_status(const SystemStatus \*status)  
{  
 printf("=== システム状態 ===\n");  
 printf("接続状態: %s\n", status->is\_connected ? "接続中" : "未接続");  
 printf("認証状態: %s\n", status->is\_authenticated ? "認証済み" : "未認証");  
 printf("権限状態: %s\n", status->has\_permission ? "権限あり" : "権限なし");  
 printf("動作状態: %s\n", status->is\_busy ? "ビジー" : "アイドル");  
 printf("==================\n");  
}  
  
/\* システムが利用可能かチェック \*/  
bool is\_system\_available(const SystemStatus \*status)  
{  
 /\* 接続済み、認証済み、権限あり、かつビジーでない場合に利用可能 \*/  
 return status->is\_connected &&   
 status->is\_authenticated &&   
 status->has\_permission &&   
 !status->is\_busy;  
}  
  
/\* セキュリティチェック \*/  
bool check\_security(const SystemStatus \*status)  
{  
 /\* 接続済みかつ認証済みの場合のみセキュア \*/  
 return status->is\_connected && status->is\_authenticated;  
}  
  
/\* 接続を試行 \*/  
bool connect\_system(SystemStatus \*status)  
{  
 if (status->is\_connected) {  
 printf("既に接続されています\n");  
 return false;  
 }  
   
 printf("システムに接続中...\n");  
 status->is\_connected = true;  
 printf("接続成功\n");  
 return true;  
}  
  
/\* 認証を試行 \*/  
bool authenticate\_system(SystemStatus \*status, const char \*username)  
{  
 if (!status->is\_connected) {  
 printf("エラー: 先に接続してください\n");  
 return false;  
 }  
   
 if (status->is\_authenticated) {  
 printf("既に認証済みです\n");  
 return false;  
 }  
   
 printf("ユーザー '%s' で認証中...\n", username);  
 status->is\_authenticated = true;  
   
 /\* 特定のユーザーには自動的に権限を付与 \*/  
 if (username[0] == 'a' && username[1] == 'd' &&   
 username[2] == 'm' && username[3] == 'i' &&   
 username[4] == 'n' && username[5] == '\0') {  
 status->has\_permission = true;  
 printf("管理者権限を付与しました\n");  
 }  
   
 printf("認証成功\n");  
 return true;  
}  
  
/\* タスクを実行 \*/  
bool execute\_task(SystemStatus \*status, const char \*task\_name)  
{  
 if (!is\_system\_available(status)) {  
 printf("エラー: システムが利用できません\n");  
 display\_status(status);  
 return false;  
 }  
   
 status->is\_busy = true;  
 printf("タスク '%s' を実行中...\n", task\_name);  
   
 /\* タスク実行のシミュレーション \*/  
 for (int i = 0; i < 3; i++) {  
 printf(".");  
 }  
 printf(" 完了\n");  
   
 status->is\_busy = false;  
 return true;  
}  
  
/\* 切断 \*/  
void disconnect\_system(SystemStatus \*status)  
{  
 printf("システムから切断中...\n");  
 init\_status(status); /\* すべての状態をリセット \*/  
 printf("切断完了\n");  
}  
  
int main(void)  
{  
 SystemStatus system;  
   
 /\* システム初期化 \*/  
 init\_status(&system);  
   
 printf("=== bool型を使ったシステム状態管理 ===\n\n");  
   
 /\* 初期状態を表示 \*/  
 display\_status(&system);  
   
 /\* セキュリティチェック（失敗するはず） \*/  
 printf("\nセキュリティチェック: %s\n",   
 check\_security(&system) ? "OK" : "NG");  
   
 /\* 接続試行 \*/  
 printf("\n");  
 connect\_system(&system);  
   
 /\* 再度セキュリティチェック（まだ失敗） \*/  
 printf("\nセキュリティチェック: %s\n",   
 check\_security(&system) ? "OK" : "NG");  
   
 /\* 一般ユーザーで認証 \*/  
 printf("\n");  
 authenticate\_system(&system, "user");  
   
 /\* セキュリティチェック（成功） \*/  
 printf("\nセキュリティチェック: %s\n",   
 check\_security(&system) ? "OK" : "NG");  
   
 /\* タスク実行（権限なしで失敗） \*/  
 printf("\n");  
 execute\_task(&system, "重要なタスク");  
   
 /\* 切断して再接続 \*/  
 printf("\n");  
 disconnect\_system(&system);  
   
 /\* 管理者で再接続 \*/  
 printf("\n");  
 connect\_system(&system);  
 authenticate\_system(&system, "admin");  
   
 /\* 現在の状態 \*/  
 printf("\n");  
 display\_status(&system);  
   
 /\* タスク実行（成功） \*/  
 printf("\n");  
 execute\_task(&system, "重要なタスク");  
   
 /\* 別のタスクも実行 \*/  
 execute\_task(&system, "データベース更新");  
 execute\_task(&system, "ログファイル圧縮");  
   
 /\* 最終的な切断 \*/  
 printf("\n");  
 disconnect\_system(&system);  
   
 return 0;  
}

## ex14\_2\_bit\_flags.c

/\*\*  
 \* 演習14-2: ビット操作と2進数リテラル  
 \* 8ビットフラグ管理システム  
 \*   
 \* C23の2進数リテラルを使用して、ビットフラグの管理システムを実装します。  
 \*/  
  
#include <stdio.h>  
  
/\* ビットフラグの定義（2進数リテラル使用） \*/  
#define FLAG\_READ 0b00000001 /\* 読み取り権限 \*/  
#define FLAG\_WRITE 0b00000010 /\* 書き込み権限 \*/  
#define FLAG\_EXECUTE 0b00000100 /\* 実行権限 \*/  
#define FLAG\_DELETE 0b00001000 /\* 削除権限 \*/  
#define FLAG\_SHARE 0b00010000 /\* 共有権限 \*/  
#define FLAG\_ARCHIVE 0b00100000 /\* アーカイブフラグ \*/  
#define FLAG\_HIDDEN 0b01000000 /\* 隠しファイルフラグ \*/  
#define FLAG\_SYSTEM 0b10000000 /\* システムファイルフラグ \*/  
  
/\* ビット操作マクロ \*/  
#define SET\_BIT(flags, bit) ((flags) |= (bit))  
#define CLEAR\_BIT(flags, bit) ((flags) &= ~(bit))  
#define TOGGLE\_BIT(flags, bit) ((flags) ^= (bit))  
#define CHECK\_BIT(flags, bit) ((flags) & (bit))  
  
/\* フラグの型定義 \*/  
typedef unsigned char FileFlags;  
  
/\* フラグを2進数形式で表示 \*/  
void display\_binary(FileFlags flags)  
{  
 printf("0b");  
 for (int i = 7; i >= 0; i--) {  
 printf("%d", (flags >> i) & 1);  
 }  
}  
  
/\* フラグの状態を詳細表示 \*/  
void display\_flags(FileFlags flags)  
{  
 printf("現在のフラグ: ");  
 display\_binary(flags);  
 printf(" (0x%02X)\n", flags);  
   
 printf("権限設定:\n");  
 printf(" 読み取り: %s\n", CHECK\_BIT(flags, FLAG\_READ) ? "○" : "×");  
 printf(" 書き込み: %s\n", CHECK\_BIT(flags, FLAG\_WRITE) ? "○" : "×");  
 printf(" 実行: %s\n", CHECK\_BIT(flags, FLAG\_EXECUTE) ? "○" : "×");  
 printf(" 削除: %s\n", CHECK\_BIT(flags, FLAG\_DELETE) ? "○" : "×");  
 printf(" 共有: %s\n", CHECK\_BIT(flags, FLAG\_SHARE) ? "○" : "×");  
   
 printf("属性:\n");  
 printf(" アーカイブ: %s\n", CHECK\_BIT(flags, FLAG\_ARCHIVE) ? "○" : "×");  
 printf(" 隠し: %s\n", CHECK\_BIT(flags, FLAG\_HIDDEN) ? "○" : "×");  
 printf(" システム: %s\n", CHECK\_BIT(flags, FLAG\_SYSTEM) ? "○" : "×");  
}  
  
/\* 基本権限の設定（読み取り、書き込み、実行） \*/  
void set\_basic\_permissions(FileFlags \*flags)  
{  
 \*flags = FLAG\_READ | FLAG\_WRITE | FLAG\_EXECUTE;  
 printf("基本権限を設定しました: ");  
 display\_binary(\*flags);  
 printf("\n");  
}  
  
/\* フルアクセス権限の設定 \*/  
void set\_full\_access(FileFlags \*flags)  
{  
 \*flags = FLAG\_READ | FLAG\_WRITE | FLAG\_EXECUTE | FLAG\_DELETE | FLAG\_SHARE;  
 printf("フルアクセス権限を設定しました: ");  
 display\_binary(\*flags);  
 printf("\n");  
}  
  
/\* 読み取り専用に設定 \*/  
void set\_readonly(FileFlags \*flags)  
{  
 \*flags = FLAG\_READ;  
 printf("読み取り専用に設定しました: ");  
 display\_binary(\*flags);  
 printf("\n");  
}  
  
/\* アクセスレベルをチェック \*/  
void check\_access\_level(FileFlags flags)  
{  
 printf("\nアクセスレベル解析:\n");  
   
 /\* 基本的なアクセスパターンをチェック \*/  
 if ((flags & (FLAG\_READ | FLAG\_WRITE | FLAG\_EXECUTE)) ==   
 (FLAG\_READ | FLAG\_WRITE | FLAG\_EXECUTE)) {  
 printf("- フルコントロール可能\n");  
 } else if ((flags & (FLAG\_READ | FLAG\_WRITE)) == (FLAG\_READ | FLAG\_WRITE)) {  
 printf("- 読み書き可能\n");  
 } else if (flags & FLAG\_READ) {  
 printf("- 読み取り専用\n");  
 } else {  
 printf("- アクセス権限なし\n");  
 }  
   
 /\* 特殊な組み合わせをチェック \*/  
 if ((flags & (FLAG\_SYSTEM | FLAG\_HIDDEN)) == (FLAG\_SYSTEM | FLAG\_HIDDEN)) {  
 printf("- 保護されたシステムファイル\n");  
 }  
   
 if (!(flags & FLAG\_SHARE) && (flags & FLAG\_WRITE)) {  
 printf("- 排他的書き込みモード\n");  
 }  
}  
  
/\* ビット演算のデモンストレーション \*/  
void demonstrate\_bit\_operations(void)  
{  
 FileFlags flags = 0;  
   
 printf("\n=== ビット演算デモ ===\n");  
   
 /\* 個別ビットの設定 \*/  
 printf("\n1. ビットを1つずつ設定:\n");  
 SET\_BIT(flags, FLAG\_READ);  
 printf(" READ設定後: ");  
 display\_binary(flags);  
 printf("\n");  
   
 SET\_BIT(flags, FLAG\_WRITE);  
 printf(" WRITE設定後: ");  
 display\_binary(flags);  
 printf("\n");  
   
 SET\_BIT(flags, FLAG\_HIDDEN);  
 printf(" HIDDEN設定後: ");  
 display\_binary(flags);  
 printf("\n");  
   
 /\* ビットのクリア \*/  
 printf("\n2. ビットをクリア:\n");  
 CLEAR\_BIT(flags, FLAG\_WRITE);  
 printf(" WRITEクリア後: ");  
 display\_binary(flags);  
 printf("\n");  
   
 /\* ビットのトグル \*/  
 printf("\n3. ビットをトグル:\n");  
 printf(" EXECUTE トグル前: ");  
 display\_binary(flags);  
 printf("\n");  
 TOGGLE\_BIT(flags, FLAG\_EXECUTE);  
 printf(" EXECUTE トグル後: ");  
 display\_binary(flags);  
 printf("\n");  
 TOGGLE\_BIT(flags, FLAG\_EXECUTE);  
 printf(" 再度トグル後: ");  
 display\_binary(flags);  
 printf("\n");  
   
 /\* 複数ビットの一括操作 \*/  
 printf("\n4. 複数ビットの一括設定:\n");  
 flags |= (FLAG\_ARCHIVE | FLAG\_SYSTEM);  
 printf(" ARCHIVE|SYSTEM: ");  
 display\_binary(flags);  
 printf("\n");  
}  
  
int main(void)  
{  
 FileFlags file\_flags = 0;  
   
 printf("=== 2進数リテラルを使ったビットフラグ管理 ===\n\n");  
   
 /\* 初期状態 \*/  
 printf("初期状態:\n");  
 display\_flags(file\_flags);  
   
 /\* 基本権限の設定 \*/  
 printf("\n--- 基本権限の設定 ---\n");  
 set\_basic\_permissions(&file\_flags);  
 display\_flags(file\_flags);  
 check\_access\_level(file\_flags);  
   
 /\* 属性の追加 \*/  
 printf("\n--- 属性の追加 ---\n");  
 SET\_BIT(file\_flags, FLAG\_ARCHIVE);  
 printf("アーカイブ属性を追加\n");  
 display\_flags(file\_flags);  
   
 /\* フルアクセスに変更 \*/  
 printf("\n--- フルアクセスに変更 ---\n");  
 set\_full\_access(&file\_flags);  
 display\_flags(file\_flags);  
 check\_access\_level(file\_flags);  
   
 /\* システムファイルとして設定 \*/  
 printf("\n--- システムファイル設定 ---\n");  
 SET\_BIT(file\_flags, FLAG\_SYSTEM);  
 SET\_BIT(file\_flags, FLAG\_HIDDEN);  
 CLEAR\_BIT(file\_flags, FLAG\_DELETE); /\* 削除不可に \*/  
 display\_flags(file\_flags);  
 check\_access\_level(file\_flags);  
   
 /\* 読み取り専用に変更 \*/  
 printf("\n--- 読み取り専用に変更 ---\n");  
 set\_readonly(&file\_flags);  
 display\_flags(file\_flags);  
 check\_access\_level(file\_flags);  
   
 /\* ビット演算のデモ \*/  
 demonstrate\_bit\_operations();  
   
 /\* 実用例: 権限のマスク処理 \*/  
 printf("\n\n=== 権限マスクの例 ===\n");  
 FileFlags user\_request = FLAG\_READ | FLAG\_WRITE | FLAG\_DELETE;  
 FileFlags allowed\_mask = FLAG\_READ | FLAG\_WRITE; /\* 削除は許可しない \*/  
 FileFlags granted = user\_request & allowed\_mask;  
   
 printf("要求された権限: ");  
 display\_binary(user\_request);  
 printf("\n許可マスク: ");  
 display\_binary(allowed\_mask);  
 printf("\n付与された権限: ");  
 display\_binary(granted);  
 printf("\n");  
   
 return 0;  
}

## ex14\_3\_generic\_macros.c

/\*\*  
 \* 演習14-3: typeof演算子の応用  
 \* 汎用マクロの実装  
 \*   
 \* C23のtypeof演算子を使用して、型安全な汎用マクロを実装します。  
 \*/  
  
#include <stdio.h>  
  
/\* 配列の要素数を取得 \*/  
#define ARRAY\_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))  
  
/\* 型安全な最大値マクロ \*/  
#define ARRAY\_MAX(arr, size) ({ \  
 typeof(arr[0]) \_max = arr[0]; \  
 for (size\_t \_i = 1; \_i < (size); \_i++) { \  
 if (arr[\_i] > \_max) { \  
 \_max = arr[\_i]; \  
 } \  
 } \  
 \_max; \  
})  
  
/\* 型安全な最小値マクロ \*/  
#define ARRAY\_MIN(arr, size) ({ \  
 typeof(arr[0]) \_min = arr[0]; \  
 for (size\_t \_i = 1; \_i < (size); \_i++) { \  
 if (arr[\_i] < \_min) { \  
 \_min = arr[\_i]; \  
 } \  
 } \  
 \_min; \  
})  
  
/\* 型安全な配列要素の合計マクロ \*/  
#define ARRAY\_SUM(arr, size) ({ \  
 typeof(arr[0]) \_sum = 0; \  
 for (size\_t \_i = 0; \_i < (size); \_i++) { \  
 \_sum += arr[\_i]; \  
 } \  
 \_sum; \  
})  
  
/\* 型安全な変数交換マクロ \*/  
#define SWAP\_SAFE(a, b) do { \  
 typeof(a) \_temp = (a); \  
 (a) = (b); \  
 (b) = \_temp; \  
} while(0)  
  
/\* 型安全な絶対値マクロ \*/  
#define ABS\_SAFE(x) ({ \  
 typeof(x) \_x = (x); \  
 \_x < 0 ? -\_x : \_x; \  
})  
  
/\* 型安全なクランプ（範囲制限）マクロ \*/  
#define CLAMP(value, min, max) ({ \  
 typeof(value) \_val = (value); \  
 typeof(value) \_min = (min); \  
 typeof(value) \_max = (max); \  
 \_val < \_min ? \_min : (\_val > \_max ? \_max : \_val); \  
})  
  
/\* 配列の平均を計算（整数版と浮動小数点版を自動選択） \*/  
#define ARRAY\_AVG(arr, size) ({ \  
 typeof(arr[0]) \_sum = ARRAY\_SUM(arr, size); \  
 \_sum / (typeof(arr[0]))(size); \  
})  
  
/\* 配列要素を表示する汎用関数マクロ \*/  
#define PRINT\_ARRAY(arr, size, format) do { \  
 printf("[ "); \  
 for (size\_t \_i = 0; \_i < (size); \_i++) { \  
 printf(format, arr[\_i]); \  
 if (\_i < (size) - 1) printf(", "); \  
 } \  
 printf(" ]\n"); \  
} while(0)  
  
/\* デモンストレーション関数 \*/  
void demonstrate\_int\_arrays(void)  
{  
 printf("=== 整数配列での動作 ===\n");  
   
 int numbers[] = {45, 12, 78, 23, 56, 89, 34, 67};  
 size\_t size = ARRAY\_SIZE(numbers);  
   
 printf("配列: ");  
 PRINT\_ARRAY(numbers, size, "%d");  
   
 int max = ARRAY\_MAX(numbers, size);  
 int min = ARRAY\_MIN(numbers, size);  
 int sum = ARRAY\_SUM(numbers, size);  
 int avg = ARRAY\_AVG(numbers, size);  
   
 printf("最大値: %d\n", max);  
 printf("最小値: %d\n", min);  
 printf("合計: %d\n", sum);  
 printf("平均: %d\n", avg);  
   
 /\* 変数の交換 \*/  
 printf("\n変数の交換:\n");  
 int a = 100, b = 200;  
 printf("交換前: a = %d, b = %d\n", a, b);  
 SWAP\_SAFE(a, b);  
 printf("交換後: a = %d, b = %d\n", a, b);  
   
 /\* 絶対値 \*/  
 int negative = -42;  
 printf("\n絶対値: ABS(%d) = %d\n", negative, ABS\_SAFE(negative));  
   
 /\* クランプ \*/  
 printf("\nクランプ（範囲制限）:\n");  
 for (int val = -5; val <= 15; val += 5) {  
 printf("CLAMP(%d, 0, 10) = %d\n", val, CLAMP(val, 0, 10));  
 }  
}  
  
void demonstrate\_double\_arrays(void)  
{  
 printf("\n=== 浮動小数点配列での動作 ===\n");  
   
 double values[] = {3.14, 2.71, 1.41, 1.73, 2.23, 0.57};  
 size\_t size = ARRAY\_SIZE(values);  
   
 printf("配列: ");  
 PRINT\_ARRAY(values, size, "%.2f");  
   
 double max = ARRAY\_MAX(values, size);  
 double min = ARRAY\_MIN(values, size);  
 double sum = ARRAY\_SUM(values, size);  
 double avg = ARRAY\_AVG(values, size);  
   
 printf("最大値: %.2f\n", max);  
 printf("最小値: %.2f\n", min);  
 printf("合計: %.2f\n", sum);  
 printf("平均: %.2f\n", avg);  
   
 /\* 変数の交換 \*/  
 printf("\n変数の交換:\n");  
 double x = 3.14, y = 2.71;  
 printf("交換前: x = %.2f, y = %.2f\n", x, y);  
 SWAP\_SAFE(x, y);  
 printf("交換後: x = %.2f, y = %.2f\n", x, y);  
   
 /\* 絶対値 \*/  
 double neg\_val = -3.14159;  
 printf("\n絶対値: ABS(%.5f) = %.5f\n", neg\_val, ABS\_SAFE(neg\_val));  
}  
  
/\* 複雑な型での動作確認 \*/  
struct Point {  
 int x;  
 int y;  
};  
  
void demonstrate\_complex\_types(void)  
{  
 printf("\n=== 複雑な型での動作 ===\n");  
   
 /\* ポインタの交換 \*/  
 int val1 = 100, val2 = 200;  
 int \*ptr1 = &val1;  
 int \*ptr2 = &val2;  
   
 printf("ポインタの交換:\n");  
 printf("交換前: \*ptr1 = %d, \*ptr2 = %d\n", \*ptr1, \*ptr2);  
 SWAP\_SAFE(ptr1, ptr2);  
 printf("交換後: \*ptr1 = %d, \*ptr2 = %d\n", \*ptr1, \*ptr2);  
   
 /\* 構造体の交換 \*/  
 struct Point p1 = {10, 20};  
 struct Point p2 = {30, 40};  
   
 printf("\n構造体の交換:\n");  
 printf("交換前: p1=(%d,%d), p2=(%d,%d)\n", p1.x, p1.y, p2.x, p2.y);  
 SWAP\_SAFE(p1, p2);  
 printf("交換後: p1=(%d,%d), p2=(%d,%d)\n", p1.x, p1.y, p2.x, p2.y);  
}  
  
/\* 実用的な例: ソート関数での活用 \*/  
#define BUBBLE\_SORT(arr, size) do { \  
 for (size\_t i = 0; i < (size) - 1; i++) { \  
 for (size\_t j = 0; j < (size) - i - 1; j++) { \  
 if (arr[j] > arr[j + 1]) { \  
 SWAP\_SAFE(arr[j], arr[j + 1]); \  
 } \  
 } \  
 } \  
} while(0)  
  
void demonstrate\_sorting(void)  
{  
 printf("\n=== 型安全なソート ===\n");  
   
 /\* 整数配列のソート \*/  
 int int\_arr[] = {64, 34, 25, 12, 22, 11, 90};  
 size\_t int\_size = ARRAY\_SIZE(int\_arr);  
   
 printf("整数配列（ソート前）: ");  
 PRINT\_ARRAY(int\_arr, int\_size, "%d");  
   
 BUBBLE\_SORT(int\_arr, int\_size);  
   
 printf("整数配列（ソート後）: ");  
 PRINT\_ARRAY(int\_arr, int\_size, "%d");  
   
 /\* 浮動小数点配列のソート \*/  
 double double\_arr[] = {64.5, 34.2, 25.8, 12.1, 22.7};  
 size\_t double\_size = ARRAY\_SIZE(double\_arr);  
   
 printf("\n浮動小数点配列（ソート前）: ");  
 PRINT\_ARRAY(double\_arr, double\_size, "%.1f");  
   
 BUBBLE\_SORT(double\_arr, double\_size);  
   
 printf("浮動小数点配列（ソート後）: ");  
 PRINT\_ARRAY(double\_arr, double\_size, "%.1f");  
}  
  
int main(void)  
{  
 printf("=== typeof演算子を使った汎用マクロ ===\n\n");  
   
 /\* 整数配列でのデモ \*/  
 demonstrate\_int\_arrays();  
   
 /\* 浮動小数点配列でのデモ \*/  
 demonstrate\_double\_arrays();  
   
 /\* 複雑な型でのデモ \*/  
 demonstrate\_complex\_types();  
   
 /\* ソートのデモ \*/  
 demonstrate\_sorting();  
   
 return 0;  
}

## ex14\_4\_safe\_linkedlist.c

/\*\*  
 \* 演習14-4: nullptr安全プログラミング  
 \* 安全なリンクリスト実装  
 \*   
 \* C23のnullptrを使用して、NULL安全性を重視したリンクリストを実装します。  
 \*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
/\* ノード構造体 \*/  
typedef struct Node {  
 int data;  
 struct Node \*next;  
} Node;  
  
/\* リンクリスト構造体 \*/  
typedef struct {  
 Node \*head;  
 size\_t size;  
} LinkedList;  
  
/\* エラーコード \*/  
typedef enum {  
 LIST\_SUCCESS = 0,  
 LIST\_ERROR\_NULL\_POINTER,  
 LIST\_ERROR\_MEMORY\_ALLOCATION,  
 LIST\_ERROR\_NOT\_FOUND,  
 LIST\_ERROR\_EMPTY\_LIST,  
 LIST\_ERROR\_INVALID\_INDEX  
} ListError;  
  
/\* エラーメッセージを取得 \*/  
const char\* get\_error\_message(ListError error)  
{  
 switch (error) {  
 case LIST\_SUCCESS:  
 return "操作成功";  
 case LIST\_ERROR\_NULL\_POINTER:  
 return "NULLポインタエラー";  
 case LIST\_ERROR\_MEMORY\_ALLOCATION:  
 return "メモリ割り当てエラー";  
 case LIST\_ERROR\_NOT\_FOUND:  
 return "要素が見つかりません";  
 case LIST\_ERROR\_EMPTY\_LIST:  
 return "リストが空です";  
 case LIST\_ERROR\_INVALID\_INDEX:  
 return "無効なインデックス";  
 default:  
 return "未知のエラー";  
 }  
}  
  
/\* リストを初期化 \*/  
ListError list\_init(LinkedList \*\*list)  
{  
 if (list == nullptr) {  
 return LIST\_ERROR\_NULL\_POINTER;  
 }  
   
 \*list = (LinkedList \*)malloc(sizeof(LinkedList));  
 if (\*list == nullptr) {  
 return LIST\_ERROR\_MEMORY\_ALLOCATION;  
 }  
   
 (\*list)->head = nullptr;  
 (\*list)->size = 0;  
   
 return LIST\_SUCCESS;  
}  
  
/\* ノードを作成 \*/  
static Node\* create\_node(int data)  
{  
 Node \*new\_node = (Node \*)malloc(sizeof(Node));  
 if (new\_node != nullptr) {  
 new\_node->data = data;  
 new\_node->next = nullptr;  
 }  
 return new\_node;  
}  
  
/\* リストの先頭に追加 \*/  
ListError list\_push\_front(LinkedList \*list, int data)  
{  
 if (list == nullptr) {  
 return LIST\_ERROR\_NULL\_POINTER;  
 }  
   
 Node \*new\_node = create\_node(data);  
 if (new\_node == nullptr) {  
 return LIST\_ERROR\_MEMORY\_ALLOCATION;  
 }  
   
 new\_node->next = list->head;  
 list->head = new\_node;  
 list->size++;  
   
 return LIST\_SUCCESS;  
}  
  
/\* リストの末尾に追加 \*/  
ListError list\_push\_back(LinkedList \*list, int data)  
{  
 if (list == nullptr) {  
 return LIST\_ERROR\_NULL\_POINTER;  
 }  
   
 Node \*new\_node = create\_node(data);  
 if (new\_node == nullptr) {  
 return LIST\_ERROR\_MEMORY\_ALLOCATION;  
 }  
   
 if (list->head == nullptr) {  
 list->head = new\_node;  
 } else {  
 Node \*current = list->head;  
 while (current->next != nullptr) {  
 current = current->next;  
 }  
 current->next = new\_node;  
 }  
   
 list->size++;  
 return LIST\_SUCCESS;  
}  
  
/\* 指定位置に挿入 \*/  
ListError list\_insert(LinkedList \*list, size\_t index, int data)  
{  
 if (list == nullptr) {  
 return LIST\_ERROR\_NULL\_POINTER;  
 }  
   
 if (index > list->size) {  
 return LIST\_ERROR\_INVALID\_INDEX;  
 }  
   
 if (index == 0) {  
 return list\_push\_front(list, data);  
 }  
   
 Node \*new\_node = create\_node(data);  
 if (new\_node == nullptr) {  
 return LIST\_ERROR\_MEMORY\_ALLOCATION;  
 }  
   
 Node \*current = list->head;  
 for (size\_t i = 0; i < index - 1; i++) {  
 current = current->next;  
 }  
   
 new\_node->next = current->next;  
 current->next = new\_node;  
 list->size++;  
   
 return LIST\_SUCCESS;  
}  
  
/\* 値を検索 \*/  
ListError list\_find(const LinkedList \*list, int data, size\_t \*index)  
{  
 if (list == nullptr || index == nullptr) {  
 return LIST\_ERROR\_NULL\_POINTER;  
 }  
   
 if (list->head == nullptr) {  
 return LIST\_ERROR\_EMPTY\_LIST;  
 }  
   
 Node \*current = list->head;  
 size\_t pos = 0;  
   
 while (current != nullptr) {  
 if (current->data == data) {  
 \*index = pos;  
 return LIST\_SUCCESS;  
 }  
 current = current->next;  
 pos++;  
 }  
   
 return LIST\_ERROR\_NOT\_FOUND;  
}  
  
/\* 先頭から削除 \*/  
ListError list\_pop\_front(LinkedList \*list, int \*data)  
{  
 if (list == nullptr) {  
 return LIST\_ERROR\_NULL\_POINTER;  
 }  
   
 if (list->head == nullptr) {  
 return LIST\_ERROR\_EMPTY\_LIST;  
 }  
   
 Node \*temp = list->head;  
 if (data != nullptr) {  
 \*data = temp->data;  
 }  
   
 list->head = list->head->next;  
 free(temp);  
 list->size--;  
   
 return LIST\_SUCCESS;  
}  
  
/\* 値で削除 \*/  
ListError list\_remove\_value(LinkedList \*list, int data)  
{  
 if (list == nullptr) {  
 return LIST\_ERROR\_NULL\_POINTER;  
 }  
   
 if (list->head == nullptr) {  
 return LIST\_ERROR\_EMPTY\_LIST;  
 }  
   
 /\* 先頭要素の場合 \*/  
 if (list->head->data == data) {  
 return list\_pop\_front(list, nullptr);  
 }  
   
 /\* それ以外の要素 \*/  
 Node \*current = list->head;  
 while (current->next != nullptr && current->next->data != data) {  
 current = current->next;  
 }  
   
 if (current->next == nullptr) {  
 return LIST\_ERROR\_NOT\_FOUND;  
 }  
   
 Node \*temp = current->next;  
 current->next = temp->next;  
 free(temp);  
 list->size--;  
   
 return LIST\_SUCCESS;  
}  
  
/\* リストを表示 \*/  
void list\_display(const LinkedList \*list)  
{  
 if (list == nullptr) {  
 printf("Error: リストがnullptrです\n");  
 return;  
 }  
   
 printf("リスト（要素数: %zu）: ", list->size);  
   
 if (list->head == nullptr) {  
 printf("空\n");  
 return;  
 }  
   
 printf("[ ");  
 Node \*current = list->head;  
 while (current != nullptr) {  
 printf("%d", current->data);  
 if (current->next != nullptr) {  
 printf(" -> ");  
 }  
 current = current->next;  
 }  
 printf(" ]\n");  
}  
  
/\* リストを解放 \*/  
ListError list\_destroy(LinkedList \*\*list)  
{  
 if (list == nullptr || \*list == nullptr) {  
 return LIST\_ERROR\_NULL\_POINTER;  
 }  
   
 Node \*current = (\*list)->head;  
 while (current != nullptr) {  
 Node \*temp = current;  
 current = current->next;  
 free(temp);  
 }  
   
 free(\*list);  
 \*list = nullptr;  
   
 return LIST\_SUCCESS;  
}  
  
/\* エラーハンドリングのデモ \*/  
void demonstrate\_error\_handling(void)  
{  
 printf("=== エラーハンドリングのデモ ===\n");  
   
 LinkedList \*list = nullptr;  
 ListError error;  
   
 /\* NULLポインタでの操作を試行 \*/  
 error = list\_push\_front(nullptr, 10);  
 printf("NULLリストへの追加: %s\n", get\_error\_message(error));  
   
 /\* リストを初期化 \*/  
 error = list\_init(&list);  
 printf("リスト初期化: %s\n", get\_error\_message(error));  
   
 /\* 空のリストから削除を試行 \*/  
 int value;  
 error = list\_pop\_front(list, &value);  
 printf("空リストからの削除: %s\n", get\_error\_message(error));  
   
 /\* 存在しない要素の検索 \*/  
 list\_push\_back(list, 10);  
 size\_t index;  
 error = list\_find(list, 20, &index);  
 printf("存在しない要素の検索: %s\n", get\_error\_message(error));  
   
 /\* 無効なインデックスへの挿入 \*/  
 error = list\_insert(list, 10, 30);  
 printf("無効なインデックスへの挿入: %s\n", get\_error\_message(error));  
   
 list\_destroy(&list);  
}  
  
/\* 通常操作のデモ \*/  
void demonstrate\_normal\_operations(void)  
{  
 printf("\n=== 通常操作のデモ ===\n");  
   
 LinkedList \*list = nullptr;  
 ListError error;  
   
 /\* リストの作成と初期化 \*/  
 error = list\_init(&list);  
 if (error != LIST\_SUCCESS) {  
 printf("初期化エラー: %s\n", get\_error\_message(error));  
 return;  
 }  
   
 printf("空のリスト: ");  
 list\_display(list);  
   
 /\* 要素の追加 \*/  
 printf("\n要素を追加:\n");  
 list\_push\_front(list, 10);  
 list\_display(list);  
   
 list\_push\_back(list, 20);  
 list\_display(list);  
   
 list\_push\_front(list, 5);  
 list\_display(list);  
   
 list\_insert(list, 2, 15);  
 list\_display(list);  
   
 /\* 検索 \*/  
 printf("\n検索テスト:\n");  
 size\_t index;  
 error = list\_find(list, 15, &index);  
 if (error == LIST\_SUCCESS) {  
 printf("値15はインデックス%zuにあります\n", index);  
 }  
   
 /\* 削除 \*/  
 printf("\n削除テスト:\n");  
 int removed;  
 list\_pop\_front(list, &removed);  
 printf("先頭から%dを削除\n", removed);  
 list\_display(list);  
   
 list\_remove\_value(list, 15);  
 printf("値15を削除\n");  
 list\_display(list);  
   
 /\* メモリ解放 \*/  
 list\_destroy(&list);  
   
 /\* 解放後のポインタチェック \*/  
 if (list == nullptr) {  
 printf("\nリストは正常に解放されました\n");  
 }  
}  
  
/\* ストレステスト \*/  
void stress\_test(void)  
{  
 printf("\n=== ストレステスト ===\n");  
   
 LinkedList \*list = nullptr;  
 list\_init(&list);  
   
 /\* 大量のデータを追加 \*/  
 printf("1000個の要素を追加...\n");  
 for (int i = 0; i < 1000; i++) {  
 if (i % 2 == 0) {  
 list\_push\_front(list, i);  
 } else {  
 list\_push\_back(list, i);  
 }  
 }  
 printf("要素数: %zu\n", list->size);  
   
 /\* ランダムに削除 \*/  
 printf("500個の要素を削除...\n");  
 for (int i = 0; i < 500; i++) {  
 int value;  
 list\_pop\_front(list, &value);  
 }  
 printf("残り要素数: %zu\n", list->size);  
   
 /\* メモリリークチェック \*/  
 list\_destroy(&list);  
 printf("メモリを解放しました\n");  
}  
  
int main(void)  
{  
 printf("=== nullptr を使った安全なリンクリスト ===\n\n");  
   
 /\* エラーハンドリングのデモ \*/  
 demonstrate\_error\_handling();  
   
 /\* 通常操作のデモ \*/  
 demonstrate\_normal\_operations();  
   
 /\* ストレステスト \*/  
 stress\_test();  
   
 printf("\nすべてのテストが完了しました\n");  
   
 return 0;  
}

## ex14\_5\_config\_manager.c

/\*\*  
 \* 演習14-5: C23総合演習  
 \* 設定管理システム  
 \*   
 \* C23の複数の新機能を組み合わせた設定管理システムを実装します。  
 \* - bool型で各種設定のON/OFF  
 \* - 2進数リテラルでフラグ管理  
 \* - typeof演算子で型安全な操作  
 \* - nullptrで安全なポインタ操作  
 \*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
/\* 設定フラグ（2進数リテラル使用） \*/  
#define CONFIG\_DEBUG 0b00000001  
#define CONFIG\_VERBOSE 0b00000010  
#define CONFIG\_LOGGING 0b00000100  
#define CONFIG\_NETWORKING 0b00001000  
#define CONFIG\_ENCRYPTION 0b00010000  
#define CONFIG\_COMPRESSION 0b00100000  
#define CONFIG\_BACKUP 0b01000000  
#define CONFIG\_READONLY 0b10000000  
  
/\* 設定値の型定義 \*/  
typedef enum {  
 TYPE\_BOOL,  
 TYPE\_INT,  
 TYPE\_DOUBLE,  
 TYPE\_STRING  
} ConfigType;  
  
/\* 設定項目構造体 \*/  
typedef struct ConfigItem {  
 char key[64];  
 ConfigType type;  
 union {  
 bool bool\_value;  
 int int\_value;  
 double double\_value;  
 char \*string\_value;  
 } value;  
 struct ConfigItem \*next;  
} ConfigItem;  
  
/\* 設定マネージャー構造体 \*/  
typedef struct {  
 ConfigItem \*items;  
 unsigned char flags;  
 bool is\_modified;  
 char \*filename;  
} ConfigManager;  
  
/\* 型安全な値の設定マクロ \*/  
#define SET\_CONFIG\_VALUE(item, val) do { \  
 typeof(val) \_val = (val); \  
 if (item != nullptr) { \  
 item->value.int\_value = (int)\_val; \  
 } \  
} while(0)  
  
/\* 型安全な最大値取得マクロ \*/  
#define GET\_MAX\_CONFIG(a, b) ({ \  
 typeof(a) \_a = (a); \  
 typeof(b) \_b = (b); \  
 \_a > \_b ? \_a : \_b; \  
})  
  
/\* 設定マネージャーを作成 \*/  
ConfigManager\* config\_create(const char \*filename)  
{  
 ConfigManager \*config = (ConfigManager \*)malloc(sizeof(ConfigManager));  
 if (config == nullptr) {  
 return nullptr;  
 }  
   
 config->items = nullptr;  
 config->flags = 0;  
 config->is\_modified = false;  
   
 if (filename != nullptr) {  
 config->filename = (char \*)malloc(strlen(filename) + 1);  
 if (config->filename != nullptr) {  
 strcpy(config->filename, filename);  
 }  
 } else {  
 config->filename = nullptr;  
 }  
   
 return config;  
}  
  
/\* 設定項目を追加（bool型） \*/  
bool config\_add\_bool(ConfigManager \*config, const char \*key, bool value)  
{  
 if (config == nullptr || key == nullptr) {  
 return false;  
 }  
   
 ConfigItem \*item = (ConfigItem \*)malloc(sizeof(ConfigItem));  
 if (item == nullptr) {  
 return false;  
 }  
   
 strncpy(item->key, key, sizeof(item->key) - 1);  
 item->key[sizeof(item->key) - 1] = '\0';  
 item->type = TYPE\_BOOL;  
 item->value.bool\_value = value;  
 item->next = config->items;  
 config->items = item;  
 config->is\_modified = true;  
   
 return true;  
}  
  
/\* 設定項目を追加（int型） \*/  
bool config\_add\_int(ConfigManager \*config, const char \*key, int value)  
{  
 if (config == nullptr || key == nullptr) {  
 return false;  
 }  
   
 ConfigItem \*item = (ConfigItem \*)malloc(sizeof(ConfigItem));  
 if (item == nullptr) {  
 return false;  
 }  
   
 strncpy(item->key, key, sizeof(item->key) - 1);  
 item->key[sizeof(item->key) - 1] = '\0';  
 item->type = TYPE\_INT;  
 item->value.int\_value = value;  
 item->next = config->items;  
 config->items = item;  
 config->is\_modified = true;  
   
 return true;  
}  
  
/\* 設定項目を追加（double型） \*/  
bool config\_add\_double(ConfigManager \*config, const char \*key, double value)  
{  
 if (config == nullptr || key == nullptr) {  
 return false;  
 }  
   
 ConfigItem \*item = (ConfigItem \*)malloc(sizeof(ConfigItem));  
 if (item == nullptr) {  
 return false;  
 }  
   
 strncpy(item->key, key, sizeof(item->key) - 1);  
 item->key[sizeof(item->key) - 1] = '\0';  
 item->type = TYPE\_DOUBLE;  
 item->value.double\_value = value;  
 item->next = config->items;  
 config->items = item;  
 config->is\_modified = true;  
   
 return true;  
}  
  
/\* 設定項目を追加（文字列） \*/  
bool config\_add\_string(ConfigManager \*config, const char \*key, const char \*value)  
{  
 if (config == nullptr || key == nullptr || value == nullptr) {  
 return false;  
 }  
   
 ConfigItem \*item = (ConfigItem \*)malloc(sizeof(ConfigItem));  
 if (item == nullptr) {  
 return false;  
 }  
   
 strncpy(item->key, key, sizeof(item->key) - 1);  
 item->key[sizeof(item->key) - 1] = '\0';  
 item->type = TYPE\_STRING;  
 item->value.string\_value = (char \*)malloc(strlen(value) + 1);  
 if (item->value.string\_value == nullptr) {  
 free(item);  
 return false;  
 }  
 strcpy(item->value.string\_value, value);  
 item->next = config->items;  
 config->items = item;  
 config->is\_modified = true;  
   
 return true;  
}  
  
/\* 設定項目を検索 \*/  
ConfigItem\* config\_find(ConfigManager \*config, const char \*key)  
{  
 if (config == nullptr || key == nullptr) {  
 return nullptr;  
 }  
   
 ConfigItem \*current = config->items;  
 while (current != nullptr) {  
 if (strcmp(current->key, key) == 0) {  
 return current;  
 }  
 current = current->next;  
 }  
   
 return nullptr;  
}  
  
/\* フラグを設定 \*/  
void config\_set\_flag(ConfigManager \*config, unsigned char flag, bool enable)  
{  
 if (config == nullptr) {  
 return;  
 }  
   
 if (enable) {  
 config->flags |= flag;  
 } else {  
 config->flags &= ~flag;  
 }  
 config->is\_modified = true;  
}  
  
/\* フラグをチェック \*/  
bool config\_check\_flag(ConfigManager \*config, unsigned char flag)  
{  
 if (config == nullptr) {  
 return false;  
 }  
   
 return (config->flags & flag) != 0;  
}  
  
/\* 設定を表示 \*/  
void config\_display(ConfigManager \*config)  
{  
 if (config == nullptr) {  
 printf("Error: 設定マネージャーがnullptrです\n");  
 return;  
 }  
   
 printf("=== 設定情報 ===\n");  
 printf("ファイル: %s\n", config->filename != nullptr ? config->filename : "(なし)");  
 printf("変更状態: %s\n", config->is\_modified ? "変更あり" : "変更なし");  
   
 /\* フラグ状態を2進数で表示 \*/  
 printf("フラグ: 0b");  
 for (int i = 7; i >= 0; i--) {  
 printf("%d", (config->flags >> i) & 1);  
 }  
 printf("\n");  
   
 /\* 個別フラグの状態 \*/  
 printf(" デバッグ: %s\n", config\_check\_flag(config, CONFIG\_DEBUG) ? "ON" : "OFF");  
 printf(" 詳細出力: %s\n", config\_check\_flag(config, CONFIG\_VERBOSE) ? "ON" : "OFF");  
 printf(" ログ記録: %s\n", config\_check\_flag(config, CONFIG\_LOGGING) ? "ON" : "OFF");  
 printf(" ネットワーク: %s\n", config\_check\_flag(config, CONFIG\_NETWORKING) ? "ON" : "OFF");  
 printf(" 暗号化: %s\n", config\_check\_flag(config, CONFIG\_ENCRYPTION) ? "ON" : "OFF");  
 printf(" 圧縮: %s\n", config\_check\_flag(config, CONFIG\_COMPRESSION) ? "ON" : "OFF");  
 printf(" バックアップ: %s\n", config\_check\_flag(config, CONFIG\_BACKUP) ? "ON" : "OFF");  
 printf(" 読み取り専用: %s\n", config\_check\_flag(config, CONFIG\_READONLY) ? "ON" : "OFF");  
   
 /\* 設定項目 \*/  
 printf("\n設定項目:\n");  
 ConfigItem \*current = config->items;  
 while (current != nullptr) {  
 printf(" %s = ", current->key);  
 switch (current->type) {  
 case TYPE\_BOOL:  
 printf("%s", current->value.bool\_value ? "true" : "false");  
 break;  
 case TYPE\_INT:  
 printf("%d", current->value.int\_value);  
 break;  
 case TYPE\_DOUBLE:  
 printf("%.2f", current->value.double\_value);  
 break;  
 case TYPE\_STRING:  
 printf("\"%s\"", current->value.string\_value);  
 break;  
 }  
 printf(" (%s型)\n",   
 current->type == TYPE\_BOOL ? "bool" :  
 current->type == TYPE\_INT ? "int" :  
 current->type == TYPE\_DOUBLE ? "double" : "string");  
 current = current->next;  
 }  
}  
  
/\* 設定を破棄 \*/  
void config\_destroy(ConfigManager \*\*config)  
{  
 if (config == nullptr || \*config == nullptr) {  
 return;  
 }  
   
 /\* 設定項目を解放 \*/  
 ConfigItem \*current = (\*config)->items;  
 while (current != nullptr) {  
 ConfigItem \*temp = current;  
 current = current->next;  
   
 if (temp->type == TYPE\_STRING && temp->value.string\_value != nullptr) {  
 free(temp->value.string\_value);  
 }  
 free(temp);  
 }  
   
 /\* ファイル名を解放 \*/  
 if ((\*config)->filename != nullptr) {  
 free((\*config)->filename);  
 }  
   
 /\* 設定マネージャーを解放 \*/  
 free(\*config);  
 \*config = nullptr;  
}  
  
/\* 実用的な使用例 \*/  
void demonstrate\_practical\_use(void)  
{  
 printf("=== 実用的な設定管理の例 ===\n\n");  
   
 /\* 設定マネージャーを作成 \*/  
 ConfigManager \*app\_config = config\_create("app\_config.ini");  
 if (app\_config == nullptr) {  
 printf("設定マネージャーの作成に失敗\n");  
 return;  
 }  
   
 /\* アプリケーション設定を追加 \*/  
 config\_add\_string(app\_config, "app\_name", "C23 Demo Application");  
 config\_add\_string(app\_config, "version", "1.0.0");  
 config\_add\_int(app\_config, "max\_connections", 100);  
 config\_add\_int(app\_config, "timeout", 30);  
 config\_add\_double(app\_config, "cache\_size\_mb", 256.5);  
 config\_add\_bool(app\_config, "auto\_save", true);  
 config\_add\_bool(app\_config, "show\_tips", false);  
   
 /\* フラグを設定 \*/  
 config\_set\_flag(app\_config, CONFIG\_DEBUG | CONFIG\_VERBOSE, true);  
 config\_set\_flag(app\_config, CONFIG\_LOGGING | CONFIG\_BACKUP, true);  
 config\_set\_flag(app\_config, CONFIG\_ENCRYPTION, false);  
   
 /\* 設定を表示 \*/  
 config\_display(app\_config);  
   
 /\* 設定の変更 \*/  
 printf("\n\n--- 設定を変更 ---\n");  
 ConfigItem \*item = config\_find(app\_config, "max\_connections");  
 if (item != nullptr) {  
 item->value.int\_value = 200;  
 printf("max\_connectionsを200に変更しました\n");  
 }  
   
 /\* セキュリティモードを有効化 \*/  
 printf("\nセキュリティモードを有効化...\n");  
 config\_set\_flag(app\_config, CONFIG\_ENCRYPTION, true);  
 config\_set\_flag(app\_config, CONFIG\_READONLY, true);  
   
 /\* 最終状態を表示 \*/  
 printf("\n");  
 config\_display(app\_config);  
   
 /\* クリーンアップ \*/  
 config\_destroy(&app\_config);  
   
 if (app\_config == nullptr) {  
 printf("\n\n設定マネージャーは正常に破棄されました\n");  
 }  
}  
  
/\* デバッグモードのシミュレーション \*/  
void simulate\_debug\_mode(void)  
{  
 printf("\n\n=== デバッグモードのシミュレーション ===\n");  
   
 ConfigManager \*debug\_config = config\_create("debug.conf");  
   
 /\* デバッグ設定 \*/  
 config\_add\_bool(debug\_config, "enable\_assertions", true);  
 config\_add\_int(debug\_config, "log\_level", 3); /\* 0:ERROR, 1:WARN, 2:INFO, 3:DEBUG \*/  
 config\_add\_string(debug\_config, "log\_file", "/tmp/app\_debug.log");  
   
 /\* すべてのデバッグフラグをON \*/  
 config\_set\_flag(debug\_config, CONFIG\_DEBUG | CONFIG\_VERBOSE | CONFIG\_LOGGING, true);  
   
 /\* デバッグ情報を表示 \*/  
 if (config\_check\_flag(debug\_config, CONFIG\_DEBUG)) {  
 printf("デバッグモードが有効です\n");  
   
 ConfigItem \*log\_level = config\_find(debug\_config, "log\_level");  
 if (log\_level != nullptr) {  
 printf("ログレベル: %d\n", log\_level->value.int\_value);  
 }  
   
 ConfigItem \*log\_file = config\_find(debug\_config, "log\_file");  
 if (log\_file != nullptr) {  
 printf("ログファイル: %s\n", log\_file->value.string\_value);  
 }  
 }  
   
 config\_destroy(&debug\_config);  
}  
  
int main(void)  
{  
 printf("=== C23総合演習: 設定管理システム ===\n\n");  
   
 /\* 実用的な使用例 \*/  
 demonstrate\_practical\_use();  
   
 /\* デバッグモードのシミュレーション \*/  
 simulate\_debug\_mode();  
   
 printf("\n\nプログラムを終了します\n");  
   
 return 0;  
}