# 第13章 複数ファイル・発展技術 - 演習問題

# 演習問題

## 基礎課題

### 演習13-1: プリプロセッサマクロの基本

以下の仕様を満たすプログラムを作成してください：

**仕様：** - 数学的な定数をマクロで定義する（PI、E、黄金比など） - 基本的な計算マクロを定義する（面積、体積など） - 条件付きコンパイルでデバッグ機能を切り替える - 文字列化マクロと連結マクロを活用する

**実装すべき機能：** - 円の面積・周長計算 - 球の体積・表面積計算 - デバッグ情報の出力（条件付き） - マクロによる型安全な最大値・最小値関数

**ファイル名：** ex13\_1\_macro\_basics.c

### 演習13-2: 安全なメモリ操作マクロ

メモリ操作の安全性を向上させるマクロ群を作成してください：

**仕様：** - NULL チェック付きメモリ割り当て - 配列境界チェック付きアクセス - 自動的なメモリ解放 - メモリリーク検出機能

**実装すべきマクロ：** - SAFE\_MALLOC(size) - NULLチェック付きmalloc - SAFE\_FREE(ptr) - NULLクリア付きfree - ARRAY\_BOUNDS\_CHECK(arr, index, size) - 境界チェック - MEMORY\_LEAK\_TRACKER - 簡単なリーク検出

**ファイル名：** ex13\_2\_safe\_memory.c

### 演習13-3: 基本的なメモリプール

固定サイズオブジェクト用の簡単なメモリプールを実装してください：

**仕様：** - 事前に確保されたメモリ領域からオブジェクトを割り当て - フリーリストによる高速な割り当て・解放 - プールの使用状況を表示する機能 - 初期化・終了処理

**実装すべき機能：** - プールの初期化と終了処理 - オブジェクトの取得と返却 - 使用状況の表示 - エラーハンドリング

**ファイル名：** ex13\_3\_memory\_pool.c

## 応用課題

### 演習13-4: 汎用的なプリプロセッサライブラリ

さまざまな用途に使える汎用的なマクロライブラリを作成してください：

**仕様：** - 型判定マクロ（コンパイル時） - 汎用的なスワップマクロ - ループ展開マクロ - アサーション機能付きマクロ - ベンチマーク計測マクロ

**実装すべき機能：** - TYPE\_CHECK(a, b) - 型の一致確認 - GENERIC\_SWAP(a, b) - 任意の型のスワップ - REPEAT(n, code) - コードの繰り返し展開 - BENCHMARK\_BLOCK(name) - ブロックの実行時間測定 - STATIC\_ASSERT(condition, message) - コンパイル時アサーション

**ファイル名：** ex13\_4\_generic\_macros.c

### 演習13-5: 高性能メモリアロケーター

パフォーマンスを重視したカスタムメモリアロケーターを実装してください：

**仕様：** - サイズ別メモリプール（小・中・大オブジェクト） - メモリの断片化を最小限に抑える仕組み - アロケーション統計情報の収集 - スレッドセーフ対応（簡易版）

**実装すべき機能：** - 複数サイズのメモリプール管理 - First Fit / Best Fit アルゴリズム - メモリ使用統計とレポート機能 - デバッグモードでの詳細トレース

**ファイル名：** ex13\_5\_allocator.c

### 演習13-6: キャッシュフレンドリーなデータ構造

CPU キャッシュ効率を考慮したデータ構造を実装してください：

**仕様：** - 配列ベースの動的配列（vector 風） - キャッシュラインを意識したメモリレイアウト - プリフェッチを活用した高速アクセス - メモリプールとの連携

**実装すべき機能：** - 動的な要素追加・削除 - キャッシュ効率的な反復処理 - バルク操作（一括挿入・削除） - パフォーマンス測定機能

**ファイル名：** ex13\_6\_cache\_vector.c

## 挑戦課題

### 演習13-7: プリプロセッサベースのDSL

プリプロセッサを使ってドメイン固有言語（DSL）を作成してください：

**仕様：** - 状態機械を記述するDSL - イベント駆動システムの記述 - 自動的なコード生成 - コンパイル時検証機能

**実装例：**

STATE\_MACHINE(TrafficLight)  
 STATE(Red) TIMEOUT() -> Yellow  
 STATE(Yellow) TIMEOUT() -> Green   
 STATE(Green) TIMEOUT() -> Red  
END\_STATE\_MACHINE  
  
EVENT\_HANDLER(ButtonPress)   
 // イベント処理コード  
END\_EVENT\_HANDLER

**ファイル名：** ex13\_7\_dsl.c

### 演習13-8: 動的メモリ管理フレームワーク

ガベージコレクション機能付きのメモリ管理フレームワークを実装してください：

**仕様：** - 参照カウント式ガベージコレクション - 循環参照の検出と解決 - 弱参照（weak reference）のサポート - メモリプレッシャー対応

**実装すべき機能：** - オブジェクトの自動管理 - 参照カウントの増減 - 循環参照のマーク&スイープ - メモリ不足時の自動解放

**ファイル名：** ex13\_8\_gc\_framework.c

### 演習13-9: リアルタイムメモリアロケーター

リアルタイムシステム向けの決定的メモリアロケーターを実装してください：

**仕様：** - O(1) での割り当て・解放保証 - メモリ断片化の完全排除 - 予測可能なメモリ使用量 - 割り込み処理からの安全な使用

**実装すべき機能：** - 固定時間でのメモリ操作 - 事前確保によるメモリプール - 優先度別メモリ管理 - リアルタイム統計情報

**ファイル名：** ex13\_9\_realtime.c

## 評価基準

### 基礎課題（各10点）

* 正しく動作する：5点
* コードの可読性：3点
* エラーハンドリング：2点

### 応用課題（各15点）

* 機能の完全性：5点
* パフォーマンス：4点
* 設計の優秀さ：3点
* 拡張性：3点

### 挑戦課題（各20点）

* 実装の完成度：8点
* 創意工夫：5点
* 技術的難易度：4点
* 実用性：3点

## 提出方法

1. 各課題を指定されたファイル名で作成
2. コンパイル用のMakefileを作成
3. 実行例とパフォーマンス結果を含むREADME.mdを作成
4. すべてのファイルをsolutions/フォルダーに配置

## 学習のポイント

### プリプロセッサの活用

* マクロの安全な使用方法
* 条件付きコンパイルの効果的な活用
* コード生成の自動化
* デバッグ支援機能

### メモリ管理の最適化

* メモリプールの設計と実装
* キャッシュ効率を考慮したデータ構造
* パフォーマンス測定と改善
* メモリリークの検出と防止

### 高度なC言語技法

* 型安全なプログラミング
* コンパイル時計算の活用
* ハードウェア特性を考慮した実装
* システムプログラミング技法

## 参考資料

### プリプロセッサ高度技法

// X-マクロパターン  
#define COLORS \  
 X(RED, "赤") \  
 X(GREEN, "緑") \  
 X(BLUE, "青")  
  
typedef enum {  
#define X(name, desc) COLOR\_##name,  
 COLORS  
#undef X  
 COLOR\_COUNT  
} Color;  
  
const char\* color\_names[] = {  
#define X(name, desc) desc,  
 COLORS  
#undef X  
};

### メモリアライメント最適化

// キャッシュライン境界でのアライメント  
#define CACHE\_LINE\_SIZE 64  
#define CACHE\_ALIGNED \_\_attribute\_\_((aligned(CACHE\_LINE\_SIZE)))  
  
typedef struct CACHE\_ALIGNED {  
 int frequently\_accessed\_data;  
 char padding[CACHE\_LINE\_SIZE - sizeof(int)];  
} CacheOptimizedStruct;

### 高性能メモリ操作

// SIMD命令を使った高速メモリコピー  
#include <immintrin.h>  
  
void fast\_memcpy(void\* dst, const void\* src, size\_t size) {  
 // AVXを使った実装例  
 if (size >= 32 && ((intptr\_t)dst & 31) == 0 && ((intptr\_t)src & 31) == 0) {  
 // 32バイト境界でアラインされている場合の最適化  
 }  
}

## よくある間違いとその対策

### 1. マクロの副作用

**間違い：**

#define SQUARE(x) x \* x  
int a = 5;  
int result = SQUARE(++a); // a が2回インクリメントされる

**正解：**

#define SQUARE(x) ({ \  
 typeof(x) \_temp = (x); \  
 \_temp \* \_temp; \  
})

### 2. メモリアライメントの無視

**間違い：**

char buffer[100];  
int\* ptr = (int\*)&buffer[1]; // 不正なアライメント

**正解：**

#include <stdalign.h>  
alignas(int) char buffer[100];  
int\* ptr = (int\*)buffer;

### 3. キャッシュ効率の無視

**間違い：**

// 列優先アクセス（キャッシュミスが多発）  
for (int j = 0; j < cols; j++) {  
 for (int i = 0; i < rows; i++) {  
 matrix[i][j] = value;  
 }  
}

**正解：**

// 行優先アクセス（キャッシュフレンドリー）  
for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < cols; j++) {  
 matrix[i][j] = value;  
 }  
}

これらの課題を通じて、C言語の高度な機能を習得し、パフォーマンスを意識したプログラミングスキルを身につけてください。

# 解答例

## 演習13-1: 高度なプリプロセッサマクロ

**ファイル**: ex13\_1\_macro\_basics.c, ex13\_1\_macro\_basics\_c99.c

基本的なプリプロセッサマクロの実装です。 - 数学定数、計算マクロ、ビット操作 - デバッグ支援マクロ（条件付きコンパイル） - C90版：標準準拠のマクロ実装 - C99版：可変引数マクロ、\_\_func\_\_マクロ活用

## 演習13-2: 安全なメモリ操作マクロ

**ファイル**: ex13\_2\_safe\_memory.c, ex13\_2\_safe\_memory\_c99.c

メモリ安全性を高めるマクロセットの実装です。 - NULLチェック付きメモリ操作 - 配列境界チェック - メモリリーク追跡システム - C90版：基本的な安全性チェック - C99版：inline関数、可変引数マクロによる拡張

## 演習13-3: シンプルなメモリプール

**ファイル**: ex13\_3\_memory\_pool.c, ex13\_3\_memory\_pool\_c99.c

固定サイズオブジェクト用メモリプールの実装です。 - O(1)での割り当て・解放 - フリーリスト管理 - フラグメンテーション対策 - C90版：基本的なプール実装 - C99版：型安全性の向上、統計機能追加

## 演習13-4: 汎用的なプリプロセッサライブラリ

**ファイル**: ex13\_4\_generic\_macros.c, ex13\_4\_generic\_macros\_c99.c

高度なマクロプログラミング技法の実装です。 - 型チェック、汎用操作マクロ - ループ展開、ベンチマークマクロ - C90版：memcpyベースの汎用swap - C99版：\_Generic、typeof、複合リテラル活用

## 演習13-5: 高性能メモリアロケーター

**ファイル**: ex13\_5\_allocator.c, ex13\_5\_allocator\_c99.c

マルチプール型カスタムアロケーターの実装です。 - サイズ別プール管理 - 高速割り当て・解放 - 詳細な統計情報 - C90版：基本的なプール選択アルゴリズム - C99版：inline最適化、restrict修飾子活用

## 演習13-6: キャッシュフレンドリーなデータ構造

**ファイル**: ex13\_6\_cache\_vector.c, ex13\_6\_cache\_vector\_c99.c

キャッシュ効率を考慮した動的配列の実装です。 - キャッシュラインアライメント - プリフェッチヒント - バルク操作の最適化 - C90版：基本的なアライメント対応 - C99版：フレキシブル配列メンバー、VLA活用

## 演習13-7: プリプロセッサベースのDSL

**ファイル**: ex13\_7\_dsl.c, ex13\_7\_dsl\_c99.c

ドメイン固有言語の実装例です。 - 状態機械DSL - イベント駆動システムDSL - ワークフローDSL - C90版：基本的なマクロDSL - C99版：可変引数マクロによる表現力向上

## 演習13-8: ガベージコレクション機能付きメモリ管理

**ファイル**: ex13\_8\_gc\_framework.c, ex13\_8\_gc\_framework\_c99.c

3つのGCアルゴリズムの実装です。 - マーク&スイープGC - 参照カウント方式 - 世代別GC - C90版：基本的なGCアルゴリズム - C99版：インクリメンタルGC、クロージャ対応

## 演習13-9: リアルタイムメモリアロケーター

**ファイル**: ex13\_9\_realtime.c, ex13\_9\_realtime\_c99.c

決定的時間保証を持つアロケーターの実装です。 - O(1)時間保証 - デッドライン管理 - WCET（最悪実行時間）追跡 - C90版：基本的なリアルタイム保証 - C99版：キャッシュカラリング、高度な統計

## ビルド方法

# 特定の解答をビルド  
make ex13\_1\_macro\_basics  
  
# C99版をビルド  
make ex13\_1\_macro\_basics\_c99  
  
# すべての解答をビルド  
make all  
  
# 実行  
./ex13\_1\_macro\_basics

## 学習のポイント

1. **プリプロセッサの限界と可能性**
   * マクロの適切な使用場面
   * 型安全性の確保方法
   * デバッグの考慮
2. **メモリ管理の最適化**
   * カスタムアロケーターの設計
   * キャッシュ効率の考慮
   * リアルタイム性の保証
3. **C90とC99の違い**
   * 言語機能の進化
   * より安全で表現力豊かなコード
   * パフォーマンスの改善
4. **実践的な応用**
   * 組み込みシステム
   * ゲームエンジン
   * リアルタイムシステム ## ex13\_1\_macro\_basics.c

```c / 演習13-1の解答例: プリプロセッサマクロの基本 \* ファイル名: ex13\_1\_macro\_basics.c \* 説明: 数学定数、計算マクロ、デバッグ機能の実装 \* C90準拠 - typeof等のGNU拡張は使用しない \*/

#include <stdio.h> #include <math.h> #include <time.h>

/\* デバッグモードの設定 \*/ #define DEBUG\_MODE 1

/\* 数学的定数の定義 \*/ #define PI 3.14159265358979323846 #define E 2.71828182845904523536 #define GOLDEN\_RATIO 1.61803398874989484820

/\* 文字列化マクロ \*/ #define STRINGIFY(x) #x #define TOSTRING(x) STRINGIFY(x)

/\* 連結マクロ \*/ #define CONCAT(a, b) a##b #define CONCAT3(a, b, c) a##b##c

/\* 面積計算マクロ */ #define CIRCLE\_AREA(radius) (PI*  (radius) \* (radius)) #define CIRCLE\_CIRCUMFERENCE(radius) (2.0 \* PI \* (radius)) #define RECTANGLE\_AREA(width, height) ((width) \* (height)) #define TRIANGLE\_AREA(base, height) (0.5 \* (base) \* (height))

/\* 体積計算マクロ */ #define SPHERE\_VOLUME(radius) ((4.0 / 3.0)*  PI \* (radius) \* (radius) \* (radius)) #define SPHERE\_SURFACE\_AREA(radius) (4.0 \* PI \* (radius) \* (radius)) #define CYLINDER\_VOLUME(radius, height) (PI \* (radius) \* (radius) \* (height)) #define CUBE\_VOLUME(side) ((side) \* (side) \* (side))

/\* C90準拠の最大値・最小値マクロ \*/ #define MAX(a, b) ((a) > (b) ? (a) : (b)) #define MIN(a, b) ((a) < (b) ? (a) : (b))

/\* 絶対値マクロ \*/ #define ABS(x) ((x) < 0 ? -(x) : (x))

/\* デバッグ出力マクロ（C90準拠 - 可変引数マクロなし） \*/ #if DEBUG\_MODE #define DEBUG\_PRINT(msg) printf(“[DEBUG:%s:%d] %s”, **FILE**, **LINE**, msg) #define DEBUG\_VAR\_INT(var) printf(“[DEBUG:%s:%d] %s = %d”, **FILE**, **LINE**, #var, var) #define DEBUG\_VAR\_DOUBLE(var) printf(“[DEBUG:%s:%d] %s = %g”, **FILE**, **LINE**, #var, var) #else #define DEBUG\_PRINT(msg) #define DEBUG\_VAR\_INT(var) #define DEBUG\_VAR\_DOUBLE(var) #endif

/\* アサーションマクロ \*/ #define ASSERT\_POSITIVE(x)  
do {  
if ((x) <= 0) {  
printf(“エラー: %s は正の値である必要があります (値: %g)”, #x, (double)(x));  
return -1;  
}  
} while (0)

/\* 単位変換マクロ */ #define DEG\_TO\_RAD(deg) ((deg)*  PI / 180.0) #define RAD\_TO\_DEG(rad) ((rad) \* 180.0 / PI) #define CELSIUS\_TO\_FAHRENHEIT(c) ((c) \* 9.0 / 5.0 + 32.0) #define FAHRENHEIT\_TO\_CELSIUS(f) (((f) - 32.0) \* 5.0 / 9.0)

/\* 範囲チェックマクロ \*/ #define IN\_RANGE(x, min, max) ((x) >= (min) && (x) <= (max))

/\* 配列サイズ取得マクロ \*/ #define ARRAY\_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))

/\* ベンチマーク用変数（グローバル） \*/ clock\_t bench\_start\_time;

/\* ベンチマーク開始マクロ \*/ #define BENCHMARK\_START(name)  
do {  
bench\_start\_time = clock();  
printf(“ベンチマーク開始: %s”, #name);  
} while (0)

/\* ベンチマーク終了マクロ \*/ #define BENCHMARK\_END(name)  
do {  
clock\_t bench\_end\_time = clock();  
double elapsed = ((double)(bench\_end\_time - bench\_start\_time)) / CLOCKS\_PER\_SEC;  
printf(“ベンチマーク終了: %s (実行時間: %.6f秒)”, #name, elapsed);  
} while (0)

/\* 条件付きコンパイル \*/ #ifdef ENABLE\_EXTENDED\_MATH #define EXTENDED\_PI\_DIGITS 50 const char extended\_pi[] = “3.14159265358979323846264338327950288419716939937510”; #endif

/\* 計算結果の表示関数 */ void print\_calculation(const char* desc, const char\* formula, double value) { printf(“%-25s: %s = %.6f”, desc, formula, value); }

/\* テスト関数群 \*/ void test\_basic\_calculations(void) { double radius = 5.0; double width = 10.0, height = 8.0; double base = 6.0, tri\_height = 4.0;

printf("=== 基本計算テスト ===\n");  
  
DEBUG\_VAR\_DOUBLE(radius);  
DEBUG\_VAR\_DOUBLE(width);  
DEBUG\_VAR\_DOUBLE(height);  
  
print\_calculation("円の面積", "PI \* 5 \* 5", CIRCLE\_AREA(radius));  
print\_calculation("円の周長", "2 \* PI \* 5", CIRCLE\_CIRCUMFERENCE(radius));  
print\_calculation("長方形の面積", "10 \* 8", RECTANGLE\_AREA(width, height));  
print\_calculation("三角形の面積", "0.5 \* 6 \* 4", TRIANGLE\_AREA(base, tri\_height));  
  
printf("\n");

}

void test\_volume\_calculations(void) { double radius = 3.0; double height = 10.0; double side = 4.0;

printf("=== 体積計算テスト ===\n");  
  
print\_calculation("球の体積", "(4/3) \* PI \* 3^3", SPHERE\_VOLUME(radius));  
print\_calculation("球の表面積", "4 \* PI \* 3^2", SPHERE\_SURFACE\_AREA(radius));  
print\_calculation("円柱の体積", "PI \* 3^2 \* 10", CYLINDER\_VOLUME(radius, height));  
print\_calculation("立方体の体積", "4^3", CUBE\_VOLUME(side));  
  
printf("\n");

}

void test\_max\_min\_functions(void) { int a = 10, b = 20; double x = 3.14, y = 2.71; int counter = 0; int temp\_counter; int max\_result;

printf("=== 最大値・最小値テスト ===\n");  
  
printf("MAX(%d, %d) = %d\n", a, b, MAX(a, b));  
printf("MIN(%d, %d) = %d\n", a, b, MIN(a, b));  
printf("MAX(%.2f, %.2f) = %.2f\n", x, y, MAX(x, y));  
printf("MIN(%.2f, %.2f) = %.2f\n", x, y, MIN(x, y));  
  
/\* 副作用のテスト（C90準拠版） \*/  
printf("副作用テスト: counter = %d\n", counter);  
temp\_counter = counter + 1;  
max\_result = MAX(temp\_counter, 5);  
counter = temp\_counter;  
printf("MAX(++counter, 5) = %d, counter = %d （注意：副作用の例）\n", max\_result, counter);  
  
printf("\n");

}

void test\_unit\_conversions(void) { double degrees = 90.0; double radians = PI / 4.0; double celsius = 25.0; double fahrenheit = 77.0;

printf("=== 単位変換テスト ===\n");  
  
print\_calculation("90度をラジアンに", "90 \* PI / 180", DEG\_TO\_RAD(degrees));  
print\_calculation("π/4ラジアンを度に", "(PI/4) \* 180 / PI", RAD\_TO\_DEG(radians));  
print\_calculation("25℃を華氏に", "25 \* 9/5 + 32", CELSIUS\_TO\_FAHRENHEIT(celsius));  
print\_calculation("77℉を摂氏に", "(77 - 32) \* 5/9", FAHRENHEIT\_TO\_CELSIUS(fahrenheit));  
  
printf("\n");

}

void test\_string\_macros(void) { printf(“=== 文字列マクロテスト ===”);

printf("PI定数の文字列化: %s\n", TOSTRING(PI));  
printf("E定数の文字列化: %s\n", TOSTRING(E));  
printf("黄金比の文字列化: %s\n", TOSTRING(GOLDEN\_RATIO));  
  
/\* 連結マクロのテスト \*/  
printf("連結マクロのテスト: %s\n", TOSTRING(CONCAT(test\_, function)));  
  
printf("\n");

}

void test\_range\_and\_array(void) { int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; int value = 5;

printf("=== 範囲チェックと配列テスト ===\n");  
  
printf("配列サイズ: %lu\n", (unsigned long)ARRAY\_SIZE(numbers));  
printf("値 %d は範囲 [1, 10] に含まれる: %s\n",  
 value, IN\_RANGE(value, 1, 10) ? "はい" : "いいえ");  
printf("値 %d は範囲 [6, 10] に含まれる: %s\n",  
 value, IN\_RANGE(value, 6, 10) ? "はい" : "いいえ");  
  
printf("\n");

}

void test\_benchmark(void) { int i; double sum = 0.0;

printf("=== ベンチマークテスト ===\n");  
  
BENCHMARK\_START(calculation\_test);  
  
/\* 計算集約的な処理をシミュレート \*/  
for (i = 0; i < 1000000; i++) {  
 sum += CIRCLE\_AREA(i \* 0.001);  
}  
  
BENCHMARK\_END(calculation\_test);  
  
printf("計算結果（無視される値）: %.6f\n", sum);  
printf("\n");

}

/\* コンパイル時情報の表示 \*/ void show\_compile\_info(void) { printf(“=== コンパイル時情報 ===”); printf(“ファイル: %s”, **FILE**); printf(“コンパイル日時: %s %s”, **DATE**, **TIME**); printf(“C標準:”);

#if defined(**STDC\_VERSION**) #if **STDC\_VERSION** >= 201112L printf(“C11”); #elif **STDC\_VERSION** >= 199901L printf(“C99”); #elif **STDC\_VERSION** >= 199409L printf(“C95”); #else printf(“C90”); #endif #else printf(“C90”); #endif

printf("デバッグモード: %s\n", DEBUG\_MODE ? "有効" : "無効");

#ifdef ENABLE\_EXTENDED\_MATH printf(“拡張数学モード: 有効”); printf(“拡張π値: %s”, extended\_pi); #else printf(“拡張数学モード: 無効”); #endif

printf("\n");

}

/\* エラーハンドリングのテスト \*/ int test\_error\_handling(void) { double negative\_radius = -5.0;

printf("=== エラーハンドリングテスト ===\n");  
  
printf("負の半径でのテスト: %.1f\n", negative\_radius);  
ASSERT\_POSITIVE(negative\_radius);  
  
/\* この行は実行されない \*/  
printf("この行は表示されません\n");  
  
return 0;

}

int main(void) { printf(“=== プリプロセッサマクロ基本デモ ===”);

/\* コンパイル時情報の表示 \*/  
show\_compile\_info();  
  
/\* 各テスト関数の実行 \*/  
test\_basic\_calculations();  
test\_volume\_calculations();  
test\_max\_min\_functions();  
test\_unit\_conversions();  
test\_string\_macros();  
test\_range\_and\_array();  
test\_benchmark();  
  
/\* エラーハンドリングのテスト（プログラム終了） \*/  
printf("エラーハンドリングのテストを実行します...\n");  
test\_error\_handling();  
  
printf("=== デモ完了 ===\n");  
return 0;

}

/\* 実行例: === プリプロセッサマクロ基本デモ ===

=== コンパイル時情報 === ファイル: ex13\_1\_macro\_basics.c コンパイル日時: Dec 25 2024 12:00:00 C標準: C90 デバッグモード: 有効 拡張数学モード: 無効

=== 基本計算テスト === [DEBUG:ex13\_1\_macro\_basics.c:108] radius = 5 [DEBUG:ex13\_1\_macro\_basics.c:109] width = 10 [DEBUG:ex13\_1\_macro\_basics.c:110] height = 8 円の面積 : PI \* 5 \* 5 = 78.539816 円の周長 : 2 \* PI \* 5 = 31.415927 長方形の面積 : 10 \* 8 = 80.000000 三角形の面積 : 0.5 \* 6 \* 4 = 12.000000

=== 体積計算テスト === 球の体積 : (4/3) \* PI \* 3^3 = 113.097335 球の表面積 : 4 \* PI \* 3^2 = 113.097335 円柱の体積 : PI \* 3^2 \* 10 = 282.743339 立方体の体積 : 4^3 = 64.000000

=== 最大値・最小値テスト === MAX(10, 20) = 20 MIN(10, 20) = 10 MAX(3.14, 2.71) = 3.14 MIN(3.14, 2.71) = 2.71 副作用テスト: counter = 0 MAX(++counter, 5) = 5, counter = 1 （注意：副作用の例）

=== 単位変換テスト === 90度をラジアンに : 90 \* PI / 180 = 1.570796 π/4ラジアンを度に : (PI/4) \* 180 / PI = 45.000000 25℃を華氏に : 25 \* 9/5 + 32 = 77.000000 77℉を摂氏に : (77 - 32) \* 5/9 = 25.000000

=== 文字列マクロテスト === PI定数の文字列化: PI E定数の文字列化: E 黄金比の文字列化: GOLDEN\_RATIO 連結マクロのテスト: test\_function

=== 範囲チェックと配列テスト === 配列サイズ: 10 値 5 は範囲 [1, 10] に含まれる: はい 値 5 は範囲 [6, 10] に含まれる: いいえ

=== ベンチマークテスト === ベンチマーク開始: calculation\_test ベンチマーク終了: calculation\_test (実行時間: 0.123456秒) 計算結果（無視される値）: 523598.775600

エラーハンドリングのテストを実行します… === エラーハンドリングテスト === 負の半径でのテスト: -5.0 エラー: negative\_radius は正の値である必要があります (値: -5) \*/```

## ex13\_2\_safe\_memory.c

```c / 演習13-2の解答例: 安全なメモリ操作マクロ \* ファイル名: ex13\_2\_safe\_memory.c \* 説明: NULLチェック、境界チェック、メモリリーク検出機能の実装 \* C90準拠 \*/

#include <stdio.h> #include <stdlib.h> #include <string.h>

/\* メモリリーク追跡用の構造体 */ typedef struct MemoryTracker { void* ptr; size\_t size; const char *file; int line; struct MemoryTracker* next; } MemoryTracker;

/\* グローバルなメモリ追跡リスト */ static MemoryTracker* g\_memory\_list = NULL; static size\_t g\_total\_allocated = 0; static size\_t g\_total\_freed = 0; static int g\_allocation\_count = 0; static int g\_free\_count = 0;

/\* メモリ追跡の有効/無効 \*/ #define MEMORY\_TRACKING\_ENABLED 1

/\* エラー処理用マクロ（C90準拠 - 可変引数マクロなし） \*/ #define MEMORY\_ERROR(msg)  
fprintf(stderr, “[MEMORY ERROR:%s:%d] %s”, **FILE**, **LINE**, msg)

#define MEMORY\_WARNING(msg)  
fprintf(stderr, “[MEMORY WARNING:%s:%d] %s”, **FILE**, **LINE**, msg)

#define MEMORY\_INFO(msg)  
printf(“[MEMORY INFO:%s:%d] %s”, **FILE**, **LINE**, msg)

/\* 安全なメモリ割り当てマクロ \*/ #define SAFE\_MALLOC(size) safe\_malloc\_impl((size), **FILE**, **LINE**) #define SAFE\_CALLOC(count, size) safe\_calloc\_impl((count), (size), **FILE**, **LINE**) #define SAFE\_REALLOC(ptr, size) safe\_realloc\_impl((ptr), (size), **FILE**, **LINE**)

/\* 安全なメモリ解放マクロ \*/ #define SAFE\_FREE(ptr)  
do {  
if (ptr) {  
safe\_free\_impl((ptr), **FILE**, **LINE**);  
(ptr) = NULL;  
}  
} while (0)

/\* 配列境界チェック付きアクセスマクロ \*/ #define ARRAY\_BOUNDS\_CHECK(arr, index, size)  
((index) >= 0 && (size\_t)(index) < (size\_t)(size))

#define SAFE\_ARRAY\_ACCESS(arr, index, size, default\_val)  
(ARRAY\_BOUNDS\_CHECK(arr, index, size) ? (arr)[index] : (default\_val))

#define SAFE\_ARRAY\_SET(arr, index, size, value)  
do {  
if (ARRAY\_BOUNDS\_CHECK(arr, index, size)) {  
(arr)[index] = (value);  
} else {  
fprintf(stderr, “[MEMORY ERROR:%s:%d] 配列境界エラー: インデックス %d が範囲 [0, %lu) を超えています”,  
**FILE**, **LINE**, (int)(index), (unsigned long)(size));  
}  
} while (0)

/\* メモリ初期化マクロ \*/ #define SAFE\_MEMSET(ptr, value, size)  
do {  
if (ptr) {  
memset((ptr), (value), (size));  
} else {  
MEMORY\_ERROR(“NULLポインタに対するmemset”);  
}  
} while (0)

#define SAFE\_MEMCPY(dest, src, size)  
do {  
if ((dest) && (src)) {  
memcpy((dest), (src), (size));  
} else {  
MEMORY\_ERROR(“NULLポインタに対するmemcpy”);  
}  
} while (0)

/\* 文字列操作の安全なマクロ \*/ #define SAFE\_STRNCPY(dest, src, size)  
do {  
if ((dest) && (src) && (size) > 0) {  
strncpy((dest), (src), (size) - 1);  
(dest)[(size) - 1] = ‘\0’;  
} else {  
MEMORY\_ERROR(“不正なstrncpyパラメータ”);  
}  
} while (0)

#define SAFE\_STRNCAT(dest, src, size)  
do {  
if ((dest) && (src) && (size) > 0) {  
size\_t dest\_len = strlen(dest);  
if (dest\_len < (size) - 1) {  
strncat((dest), (src), (size) - dest\_len - 1);  
}  
} else {  
MEMORY\_ERROR(“不正なstrncatパラメータ”);  
}  
} while (0)

/\* ポインタ検証マクロ \*/ #define VALIDATE\_POINTER(ptr, action)  
do {  
if (!(ptr)) {  
MEMORY\_ERROR(“NULLポインタが検出されました”);  
action;  
}  
} while (0)

#define VALIDATE\_POINTER\_RETURN(ptr, retval)  
VALIDATE\_POINTER(ptr, return (retval))

#define VALIDATE\_POINTER\_RETURN\_VOID(ptr)  
VALIDATE\_POINTER(ptr, return)

/\* メモリリーク検出マクロ \*/ #define MEMORY\_LEAK\_TRACKER\_INIT() init\_memory\_tracker() #define MEMORY\_LEAK\_TRACKER\_REPORT() report\_memory\_leaks() #define MEMORY\_LEAK\_TRACKER\_CLEANUP() cleanup\_memory\_tracker()

/\* 実装関数群 \*/

/\* メモリ追跡の初期化 \*/ void init\_memory\_tracker(void) { g\_memory\_list = NULL; g\_total\_allocated = 0; g\_total\_freed = 0; g\_allocation\_count = 0; g\_free\_count = 0; MEMORY\_INFO(“メモリ追跡システムを初期化しました”); }

/\* メモリ追跡エントリの追加 */ static void add\_memory\_tracker(void* ptr, size\_t size, const char *file, int line) { #if MEMORY\_TRACKING\_ENABLED MemoryTracker* tracker = (MemoryTracker \*)malloc(sizeof(MemoryTracker)); if (tracker) { tracker->ptr = ptr; tracker->size = size; tracker->file = file; tracker->line = line; tracker->next = g\_memory\_list; g\_memory\_list = tracker;

g\_total\_allocated += size;  
 g\_allocation\_count++;  
}

#endif }

/\* メモリ追跡エントリの削除 */ static void remove\_memory\_tracker(void* ptr, const char \*file, int line) { #if MEMORY\_TRACKING\_ENABLED MemoryTracker \*\*current = &g\_memory\_list; while (*current) { if ((*current)->ptr == ptr) { MemoryTracker *to\_remove =* current; g\_total\_freed += to\_remove->size; g\_free\_count++; *current = to\_remove->next; free(to\_remove); return; } current = &(*current)->next; } fprintf(stderr, “[MEMORY WARNING:%s:%d] 解放されたポインタ %p が追跡リストに見つかりません”, file, line, ptr); #endif }

/\* 安全なmalloc実装 */ void* safe\_malloc\_impl(size\_t size, const char *file, int line) { void* ptr;

if (size == 0) {  
 fprintf(stderr, "[MEMORY WARNING:%s:%d] サイズ0でのmalloc\n", file, line);  
 return NULL;  
}  
  
ptr = malloc(size);  
if (!ptr) {  
 fprintf(stderr, "[MEMORY ERROR:%s:%d] malloc失敗: %luバイト\n",   
 file, line, (unsigned long)size);  
 return NULL;  
}  
  
add\_memory\_tracker(ptr, size, file, line);  
printf("[MEMORY INFO:%s:%d] malloc成功: %luバイト @ %p\n",   
 file, line, (unsigned long)size, ptr);  
  
return ptr;

}

/\* 安全なcalloc実装 */ void* safe\_calloc\_impl(size\_t count, size\_t size, const char *file, int line) { void* ptr; size\_t total\_size;

if (count == 0 || size == 0) {  
 fprintf(stderr, "[MEMORY WARNING:%s:%d] サイズ0でのcalloc\n", file, line);  
 return NULL;  
}  
  
/\* オーバーフローチェック \*/  
total\_size = count \* size;  
if (count != 0 && total\_size / count != size) {  
 fprintf(stderr, "[MEMORY ERROR:%s:%d] calloc整数オーバーフロー: %lu \* %lu\n",  
 file, line, (unsigned long)count, (unsigned long)size);  
 return NULL;  
}  
  
ptr = calloc(count, size);  
if (!ptr) {  
 fprintf(stderr, "[MEMORY ERROR:%s:%d] calloc失敗: %lu \* %luバイト\n",  
 file, line, (unsigned long)count, (unsigned long)size);  
 return NULL;  
}  
  
add\_memory\_tracker(ptr, total\_size, file, line);  
printf("[MEMORY INFO:%s:%d] calloc成功: %lu \* %luバイト @ %p\n",  
 file, line, (unsigned long)count, (unsigned long)size, ptr);  
  
return ptr;

}

/\* 安全なrealloc実装 */ void* safe\_realloc\_impl(void *ptr, size\_t size, const char* file, int line) { void \*new\_ptr;

if (!ptr) {  
 return safe\_malloc\_impl(size, file, line);  
}  
  
if (size == 0) {  
 safe\_free\_impl(ptr, file, line);  
 return NULL;  
}  
  
new\_ptr = realloc(ptr, size);  
if (!new\_ptr) {  
 fprintf(stderr, "[MEMORY ERROR:%s:%d] realloc失敗: %luバイト\n",   
 file, line, (unsigned long)size);  
 return NULL;  
}  
  
/\* 追跡情報の更新 \*/  
remove\_memory\_tracker(ptr, file, line);  
add\_memory\_tracker(new\_ptr, size, file, line);  
  
printf("[MEMORY INFO:%s:%d] realloc成功: %p -> %p (%luバイト)\n",  
 file, line, ptr, new\_ptr, (unsigned long)size);  
  
return new\_ptr;

}

/\* 安全なfree実装 */ void safe\_free\_impl(void* ptr, const char \*file, int line) { if (!ptr) { fprintf(stderr, “[MEMORY WARNING:%s:%d] NULLポインタのfree”, file, line); return; }

remove\_memory\_tracker(ptr, file, line);  
free(ptr);  
/\* 注意: デバッグ目的でポインタ値を表示。実際のコードでは解放後のポインタアクセスは避ける \*/  
printf("[MEMORY INFO:%s:%d] free成功: アドレス %p を解放しました\n", file, line, (void\*)ptr);

}

/\* メモリリークレポート */ void report\_memory\_leaks(void) { MemoryTracker* current; size\_t leak\_total = 0; int leak\_count = 0;

printf("\n=== メモリリークレポート ===\n");  
printf("総割り当て: %lu バイト (%d 回)\n",  
 (unsigned long)g\_total\_allocated, g\_allocation\_count);  
printf("総解放: %lu バイト (%d 回)\n",  
 (unsigned long)g\_total\_freed, g\_free\_count);  
  
if (g\_memory\_list) {  
 printf("\n検出されたリーク:\n");  
 current = g\_memory\_list;  
   
 while (current) {  
 printf(" %p: %lu バイト (%s:%d)\n",  
 current->ptr, (unsigned long)current->size,  
 current->file, current->line);  
 leak\_total += current->size;  
 leak\_count++;  
 current = current->next;  
 }  
   
 printf("\nリーク合計: %lu バイト (%d ブロック)\n",  
 (unsigned long)leak\_total, leak\_count);  
} else {  
 printf("メモリリークは検出されませんでした。\n");  
}  
printf("========================\n\n");

}

/\* メモリ追跡システムのクリーンアップ */ void cleanup\_memory\_tracker(void) { while (g\_memory\_list) { MemoryTracker* to\_remove = g\_memory\_list; g\_memory\_list = g\_memory\_list->next; free(to\_remove); } MEMORY\_INFO(“メモリ追跡システムをクリーンアップしました”); }

/\* テスト関数群 \*/

void test\_safe\_allocation(void) { int \*numbers; int i;

printf("=== 安全なメモリ割り当てテスト ===\n");  
  
/\* 通常の割り当て \*/  
numbers = (int \*)SAFE\_MALLOC(sizeof(int) \* 10);  
VALIDATE\_POINTER\_RETURN\_VOID(numbers);  
  
for (i = 0; i < 10; i++) {  
 numbers[i] = i \* i;  
}  
  
printf("割り当てた配列: ");  
for (i = 0; i < 10; i++) {  
 printf("%d ", numbers[i]);  
}  
printf("\n");  
  
/\* 配列のサイズ変更 \*/  
numbers = (int \*)SAFE\_REALLOC(numbers, sizeof(int) \* 20);  
VALIDATE\_POINTER\_RETURN\_VOID(numbers);  
  
for (i = 10; i < 20; i++) {  
 numbers[i] = i \* i;  
}  
  
printf("拡張後の配列: ");  
for (i = 0; i < 20; i++) {  
 printf("%d ", numbers[i]);  
}  
printf("\n");  
  
SAFE\_FREE(numbers);  
printf("メモリを正常に解放しました\n\n");

}

void test\_array\_bounds\_checking(void) { int \*array; int i; int safe\_value;

printf("=== 配列境界チェックテスト ===\n");  
  
array = (int \*)SAFE\_MALLOC(sizeof(int) \* 5);  
VALIDATE\_POINTER\_RETURN\_VOID(array);  
  
/\* 配列の初期化 \*/  
for (i = 0; i < 5; i++) {  
 array[i] = (i + 1) \* 10;  
}  
  
printf("配列内容: ");  
for (i = 0; i < 5; i++) {  
 printf("%d ", array[i]);  
}  
printf("\n");  
  
/\* 安全なアクセステスト \*/  
printf("安全なアクセステスト:\n");  
  
for (i = -1; i <= 6; i++) {  
 if (ARRAY\_BOUNDS\_CHECK(array, i, 5)) {  
 printf(" インデックス %d: %d (有効)\n", i, array[i]);  
 } else {  
 printf(" インデックス %d: 境界外アクセス\n", i);  
 safe\_value = SAFE\_ARRAY\_ACCESS(array, i, 5, -1);  
 printf(" -> デフォルト値: %d\n", safe\_value);  
 }  
}  
  
/\* 安全な設定テスト \*/  
printf("\n安全な設定テスト:\n");  
SAFE\_ARRAY\_SET(array, 2, 5, 999); /\* 有効 \*/  
SAFE\_ARRAY\_SET(array, 10, 5, 888); /\* 無効 \*/  
SAFE\_ARRAY\_SET(array, -1, 5, 777); /\* 無効 \*/  
  
printf("設定後の配列: ");  
for (i = 0; i < 5; i++) {  
 printf("%d ", array[i]);  
}  
printf("\n");  
  
SAFE\_FREE(array);  
printf("\n");

}

void test\_string\_operations(void) { char \*buffer; char long\_string[100]; char short\_buffer[20];

printf("=== 安全な文字列操作テスト ===\n");  
  
buffer = (char \*)SAFE\_MALLOC(64);  
VALIDATE\_POINTER\_RETURN\_VOID(buffer);  
  
/\* 安全な文字列コピー \*/  
SAFE\_STRNCPY(buffer, "Hello, World!", 64);  
printf("初期文字列: %s\n", buffer);  
  
/\* 安全な文字列連結 \*/  
SAFE\_STRNCAT(buffer, " 追加テキスト", 64);  
printf("連結後: %s\n", buffer);  
  
/\* 長すぎる文字列のテスト \*/  
memset(long\_string, 'A', 99);  
long\_string[99] = '\0';  
  
SAFE\_STRNCPY(short\_buffer, long\_string, sizeof(short\_buffer));  
printf("切り詰められた文字列: %s\n", short\_buffer);  
  
SAFE\_FREE(buffer);  
printf("\n");

}

void test\_memory\_leak\_detection(void) { void *leak1; void* leak2; void \*no\_leak;

printf("=== メモリリーク検出テスト ===\n");  
  
/\* 意図的にリークを作成 \*/  
leak1 = SAFE\_MALLOC(100);  
leak2 = SAFE\_MALLOC(200);  
no\_leak = SAFE\_MALLOC(50);  
  
/\* 一つだけ解放 \*/  
SAFE\_FREE(no\_leak);  
  
printf("意図的にリークを作成しました\n");  
printf("leak1 = %p (100バイト)\n", leak1);  
printf("leak2 = %p (200バイト)\n", leak2);  
printf("no\_leakは解放済み\n\n");  
  
/\* leak1とleak2は意図的に解放しない \*/

}

/\* メイン関数 \*/ int main(void) { printf(“=== 安全なメモリ操作マクロデモ ===”);

/\* メモリ追跡システムの初期化 \*/  
MEMORY\_LEAK\_TRACKER\_INIT();  
  
/\* 各種テストの実行 \*/  
test\_safe\_allocation();  
test\_array\_bounds\_checking();  
test\_string\_operations();  
test\_memory\_leak\_detection();  
  
/\* メモリリークレポートの表示 \*/  
MEMORY\_LEAK\_TRACKER\_REPORT();  
  
/\* メモリ追跡システムのクリーンアップ \*/  
MEMORY\_LEAK\_TRACKER\_CLEANUP();  
  
printf("=== デモ完了 ===\n");  
return 0;

}

/\* 実行例: === 安全なメモリ操作マクロデモ ===

[MEMORY INFO:ex13\_2\_safe\_memory.c:135] メモリ追跡システムを初期化しました

=== 安全なメモリ割り当てテスト === [MEMORY INFO:ex13\_2\_safe\_memory.c:207] malloc成功: 40バイト @ 0x1234560 割り当てた配列: 0 1 4 9 16 25 36 49 64 81 [MEMORY INFO:ex13\_2\_safe\_memory.c:279] realloc成功: 0x1234560 -> 0x1234580 (80バイト) 拡張後の配列: 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 [MEMORY INFO:ex13\_2\_safe\_memory.c:295] free成功: 0x1234580 メモリを正常に解放しました

=== 配列境界チェックテスト === [MEMORY INFO:ex13\_2\_safe\_memory.c:207] malloc成功: 20バイト @ 0x12345a0 配列内容: 10 20 30 40 50 安全なアクセステスト: インデックス -1: 境界外アクセス -> デフォルト値: -1 インデックス 0: 10 (有効) インデックス 1: 20 (有効) インデックス 2: 30 (有効) インデックス 3: 40 (有効) インデックス 4: 50 (有効) インデックス 5: 境界外アクセス -> デフォルト値: -1 インデックス 6: 境界外アクセス -> デフォルト値: -1

安全な設定テスト: [MEMORY ERROR:ex13\_2\_safe\_memory.c:390] 配列境界エラー: インデックス 10 が範囲 [0, 5) を超えています [MEMORY ERROR:ex13\_2\_safe\_memory.c:391] 配列境界エラー: インデックス -1 が範囲 [0, 5) を超えています 設定後の配列: 10 20 999 40 50 [MEMORY INFO:ex13\_2\_safe\_memory.c:295] free成功: 0x12345a0

=== 安全な文字列操作テスト === [MEMORY INFO:ex13\_2\_safe\_memory.c:207] malloc成功: 64バイト @ 0x12345c0 初期文字列: Hello, World! 連結後: Hello, World! 追加テキスト 切り詰められた文字列: AAAAAAAAAAAAAAAAAAA [MEMORY INFO:ex13\_2\_safe\_memory.c:295] free成功: 0x12345c0

=== メモリリーク検出テスト === [MEMORY INFO:ex13\_2\_safe\_memory.c:207] malloc成功: 100バイト @ 0x1234600 [MEMORY INFO:ex13\_2\_safe\_memory.c:207] malloc成功: 200バイト @ 0x1234680 [MEMORY INFO:ex13\_2\_safe\_memory.c:207] malloc成功: 50バイト @ 0x1234780 [MEMORY INFO:ex13\_2\_safe\_memory.c:295] free成功: 0x1234780 意図的にリークを作成しました leak1 = 0x1234600 (100バイト) leak2 = 0x1234680 (200バイト) no\_leakは解放済み

=== メモリリークレポート === 総割り当て: 474 バイト (5 回) 総解放: 174 バイト (3 回)

検出されたリーク: 0x1234680: 200 バイト (ex13\_2\_safe\_memory.c:448) 0x1234600: 100 バイト (ex13\_2\_safe\_memory.c:447)

# リーク合計: 300 バイト (2 ブロック)

[MEMORY INFO:ex13\_2\_safe\_memory.c:319] メモリ追跡システムをクリーンアップしました === デモ完了 === \*/```

## ex13\_3\_memory\_pool.c

```c / 演習13-3の解答例: 基本的なメモリプール \* ファイル名: ex13\_3\_memory\_pool.c \* 説明: 固定サイズオブジェクト用のメモリプール実装 \* C90準拠 \*/

#include <stdio.h> #include <stdlib.h> #include <string.h> #include <assert.h> #include <time.h>

/\* デバッグ出力制御 \*/ #define DEBUG\_POOL 1

#if DEBUG\_POOL #define POOL\_DEBUG(msg) printf(“[POOL DEBUG] %s”, msg) #else #define POOL\_DEBUG(msg) #endif

/\* メモリプールの設定 \*/ #define DEFAULT\_POOL\_SIZE 16 #define POOL\_ALIGNMENT 8 #define POOL\_MAGIC\_NUMBER 0xDEADBEEF

/\* アライメント調整マクロ \*/ #define ALIGN\_SIZE(size, alignment)  
(((size) + (alignment) - 1) & ~((alignment) - 1))

/\* フリーリストノード構造体 */ typedef struct FreeNode { struct FreeNode* next; } FreeNode;

/\* メモリプール構造体 */ typedef struct MemoryPool { void* memory\_chunk; /\* プール全体のメモリ領域 */ FreeNode* free\_list; /\* フリーリストの先頭 */ size\_t object\_size; /* オブジェクトサイズ */ size\_t pool\_capacity; /* プールの容量 */ size\_t objects\_in\_use; /* 使用中オブジェクト数 */ size\_t total\_allocated; /* 総割り当て数 */ size\_t total\_freed; /* 総解放数 */ unsigned int magic; /* 破損検出用マジックナンバー */ char name[32]; /* プール名 \*/ } MemoryPool;

/\* プール検証マクロ */ #define VALIDATE\_POOL(pool)*  
*do {*  
*if (!(pool) || (pool)->magic != POOL\_MAGIC\_NUMBER) {*  
*fprintf(stderr, “無効なメモリプール: %p”, (void* )(pool));  
return NULL;  
}  
} while (0)

#define VALIDATE\_POOL\_VOID(pool)  
do {  
if (!(pool) || (pool)->magic != POOL\_MAGIC\_NUMBER) {  
fprintf(stderr, “無効なメモリプール: %p”, (void \*)(pool));  
return;  
}  
} while (0)

/\* 関数プロトタイプ */ MemoryPool* pool\_create(const char *name, size\_t object\_size, size\_t capacity); void* pool\_alloc(MemoryPool *pool); void pool\_free(MemoryPool* pool, void *ptr); void pool\_print\_status(const MemoryPool* pool); void pool\_destroy(MemoryPool \*pool);

/\* メモリプールの初期化 */ MemoryPool* pool\_create(const char *name, size\_t object\_size, size\_t capacity) { size\_t min\_size; size\_t total\_size; MemoryPool* pool; void *memory; char* ptr; size\_t i;

if (!name || object\_size == 0 || capacity == 0) {  
 fprintf(stderr, "無効なプール作成パラメータ\n");  
 return NULL;  
}  
  
/\* オブジェクトサイズの調整（最小サイズ・アライメント確保） \*/  
min\_size = sizeof(FreeNode);  
if (object\_size < min\_size) {  
 object\_size = min\_size;  
}  
object\_size = ALIGN\_SIZE(object\_size, POOL\_ALIGNMENT);  
  
/\* プール構造体の割り当て \*/  
pool = (MemoryPool \*)malloc(sizeof(MemoryPool));  
if (!pool) {  
 fprintf(stderr, "プール構造体の割り当てに失敗\n");  
 return NULL;  
}  
  
/\* メモリ領域の割り当て \*/  
total\_size = object\_size \* capacity;  
memory = malloc(total\_size);  
if (!memory) {  
 fprintf(stderr, "プールメモリの割り当てに失敗: %lu バイト\n",   
 (unsigned long)total\_size);  
 free(pool);  
 return NULL;  
}  
  
/\* プール構造体の初期化 \*/  
pool->memory\_chunk = memory;  
pool->object\_size = object\_size;  
pool->pool\_capacity = capacity;  
pool->objects\_in\_use = 0;  
pool->total\_allocated = 0;  
pool->total\_freed = 0;  
pool->magic = POOL\_MAGIC\_NUMBER;  
strncpy(pool->name, name, sizeof(pool->name) - 1);  
pool->name[sizeof(pool->name) - 1] = '\0';  
  
/\* フリーリストの構築 \*/  
pool->free\_list = NULL;  
ptr = (char \*)memory;  
  
for (i = 0; i < capacity; i++) {  
 FreeNode \*node = (FreeNode \*)(ptr + i \* object\_size);  
 node->next = pool->free\_list;  
 pool->free\_list = node;  
}  
  
printf("[POOL DEBUG] プール '%s' を作成: オブジェクトサイズ=%lu, 容量=%lu\n",  
 pool->name, (unsigned long)pool->object\_size,   
 (unsigned long)pool->pool\_capacity);  
  
return pool;

}

/\* オブジェクトの割り当て */ void* pool\_alloc(MemoryPool *pool) { FreeNode* node;

VALIDATE\_POOL(pool);  
  
if (!pool->free\_list) {  
 fprintf(stderr, "プール '%s' にオブジェクトが残っていません\n", pool->name);  
 return NULL;  
}  
  
/\* フリーリストから先頭を取得 \*/  
node = pool->free\_list;  
pool->free\_list = node->next;  
  
/\* 統計情報の更新 \*/  
pool->objects\_in\_use++;  
pool->total\_allocated++;  
  
/\* メモリをクリア（セキュリティ向上） \*/  
memset(node, 0, pool->object\_size);  
  
printf("[POOL DEBUG] オブジェクト割り当て: %p (使用中: %lu/%lu)\n",  
 (void \*)node, (unsigned long)pool->objects\_in\_use,   
 (unsigned long)pool->pool\_capacity);  
  
return (void \*)node;

}

/\* オブジェクトの解放 */ void pool\_free(MemoryPool* pool, void *ptr) { char* pool\_start; char *pool\_end; char* obj\_ptr; size\_t offset; FreeNode \*node;

VALIDATE\_POOL\_VOID(pool);  
  
if (!ptr) {  
 fprintf(stderr, "NULL ポインタの解放試行\n");  
 return;  
}  
  
/\* ポインタがプール内にあるかチェック \*/  
pool\_start = (char \*)pool->memory\_chunk;  
pool\_end = pool\_start + (pool->object\_size \* pool->pool\_capacity);  
obj\_ptr = (char \*)ptr;  
  
if (obj\_ptr < pool\_start || obj\_ptr >= pool\_end) {  
 fprintf(stderr, "プール外のポインタの解放試行: %p\n", ptr);  
 return;  
}  
  
/\* アライメントチェック \*/  
offset = obj\_ptr - pool\_start;  
if (offset % pool->object\_size != 0) {  
 fprintf(stderr, "不正にアラインされたポインタ: %p\n", ptr);  
 return;  
}  
  
/\* フリーリストに追加 \*/  
node = (FreeNode \*)ptr;  
node->next = pool->free\_list;  
pool->free\_list = node;  
  
/\* 統計情報の更新 \*/  
pool->objects\_in\_use--;  
pool->total\_freed++;  
  
printf("[POOL DEBUG] オブジェクト解放: %p (使用中: %lu/%lu)\n",  
 ptr, (unsigned long)pool->objects\_in\_use,   
 (unsigned long)pool->pool\_capacity);

}

/\* プールの使用状況表示 */ void pool\_print\_status(const MemoryPool* pool) { size\_t free\_count = 0; FreeNode \*current; double usage\_ratio; double memory\_efficiency;

if (!pool || pool->magic != POOL\_MAGIC\_NUMBER) {  
 printf("無効なプール\n");  
 return;  
}  
  
printf("\n=== プール '%s' の状況 ===\n", pool->name);  
printf("オブジェクトサイズ: %lu バイト\n", (unsigned long)pool->object\_size);  
printf("プール容量: %lu オブジェクト\n", (unsigned long)pool->pool\_capacity);  
printf("使用中: %lu オブジェクト\n", (unsigned long)pool->objects\_in\_use);  
printf("空き: %lu オブジェクト\n",   
 (unsigned long)(pool->pool\_capacity - pool->objects\_in\_use));  
  
usage\_ratio = (double)pool->objects\_in\_use / pool->pool\_capacity \* 100.0;  
printf("使用率: %.1f%%\n", usage\_ratio);  
  
printf("総割り当て: %lu 回\n", (unsigned long)pool->total\_allocated);  
printf("総解放: %lu 回\n", (unsigned long)pool->total\_freed);  
  
memory\_efficiency = pool->pool\_capacity > 0 ?   
 (double)(pool->pool\_capacity - pool->objects\_in\_use) / pool->pool\_capacity \* 100.0 : 0.0;  
printf("メモリ効率: %.1f%%\n", memory\_efficiency);  
  
/\* フリーリストの長さをカウント \*/  
current = pool->free\_list;  
while (current) {  
 free\_count++;  
 current = current->next;  
}  
printf("フリーリスト長: %lu\n", (unsigned long)free\_count);  
printf("========================\n\n");

}

/\* プールの破棄 */ void pool\_destroy(MemoryPool* pool) { if (!pool) { return; }

if (pool->magic != POOL\_MAGIC\_NUMBER) {  
 fprintf(stderr, "破損したプールの破棄試行\n");  
 return;  
}  
  
printf("[POOL DEBUG] プール '%s' を破棄 (リーク: %lu オブジェクト)\n",  
 pool->name, (unsigned long)pool->objects\_in\_use);  
  
if (pool->objects\_in\_use > 0) {  
 fprintf(stderr, "警告: プール '%s' に %lu 個の未解放オブジェクトがあります\n",  
 pool->name, (unsigned long)pool->objects\_in\_use);  
}  
  
/\* メモリ解放 \*/  
if (pool->memory\_chunk) {  
 free(pool->memory\_chunk);  
}  
  
/\* マジックナンバーをクリアして破損をマーク \*/  
pool->magic = 0;  
free(pool);

}

/\* テスト用構造体 \*/ typedef struct TestObject { int id; char name[16]; double value; } TestObject;

typedef struct Point { double x, y, z; } Point;

/\* プールの基本機能テスト */ void test\_basic\_pool\_operations(void) { MemoryPool* int\_pool; int *numbers[5]; int i; int* new\_num;

printf("=== 基本プール操作テスト ===\n");  
  
/\* 小さなオブジェクト用プール \*/  
int\_pool = pool\_create("IntPool", sizeof(int), 8);  
if (!int\_pool) {  
 printf("プール作成に失敗\n");  
 return;  
}  
  
pool\_print\_status(int\_pool);  
  
/\* オブジェクトの割り当て \*/  
for (i = 0; i < 5; i++) {  
 numbers[i] = (int \*)pool\_alloc(int\_pool);  
 if (numbers[i]) {  
 \*numbers[i] = (i + 1) \* 10;  
 printf("割り当て [%d]: %p = %d\n", i, (void \*)numbers[i], \*numbers[i]);  
 }  
}  
  
pool\_print\_status(int\_pool);  
  
/\* 一部解放 \*/  
printf("\n一部のオブジェクトを解放...\n");  
pool\_free(int\_pool, numbers[1]);  
pool\_free(int\_pool, numbers[3]);  
numbers[1] = numbers[3] = NULL;  
  
pool\_print\_status(int\_pool);  
  
/\* 再割り当て \*/  
printf("再割り当てテスト...\n");  
new\_num = (int \*)pool\_alloc(int\_pool);  
if (new\_num) {  
 \*new\_num = 999;  
 printf("再割り当て: %p = %d\n", (void \*)new\_num, \*new\_num);  
}  
  
/\* 残りを解放 \*/  
for (i = 0; i < 5; i++) {  
 if (numbers[i]) {  
 pool\_free(int\_pool, numbers[i]);  
 }  
}  
pool\_free(int\_pool, new\_num);  
  
pool\_print\_status(int\_pool);  
pool\_destroy(int\_pool);

}

/\* 構造体プールのテスト */ void test\_struct\_pool(void) { MemoryPool* obj\_pool; TestObject \*objects[4]; int i;

printf("\n=== 構造体プールテスト ===\n");  
  
obj\_pool = pool\_create("ObjectPool", sizeof(TestObject), 6);  
if (!obj\_pool) {  
 return;  
}  
  
/\* オブジェクトの作成と初期化 \*/  
for (i = 0; i < 4; i++) {  
 objects[i] = (TestObject \*)pool\_alloc(obj\_pool);  
 if (objects[i]) {  
 objects[i]->id = i + 100;  
 sprintf(objects[i]->name, "Obj\_%d", i + 1);  
 objects[i]->value = (i + 1) \* 3.14;  
   
 printf("オブジェクト [%d]: ID=%d, Name=%s, Value=%.2f\n",  
 i, objects[i]->id, objects[i]->name, objects[i]->value);  
 }  
}  
  
pool\_print\_status(obj\_pool);  
  
/\* 全て解放 \*/  
for (i = 0; i < 4; i++) {  
 pool\_free(obj\_pool, objects[i]);  
}  
  
pool\_destroy(obj\_pool);

}

/\* プール枯渇テスト */ void test\_pool\_exhaustion(void) { MemoryPool* small\_pool; Point *points[5]; /* プール容量より多く要求 */ int i; Point* new\_point;

printf("\n=== プール枯渇テスト ===\n");  
  
small\_pool = pool\_create("SmallPool", sizeof(Point), 3);  
if (!small\_pool) {  
 return;  
}  
  
for (i = 0; i < 5; i++) {  
 points[i] = (Point \*)pool\_alloc(small\_pool);  
 if (points[i]) {  
 points[i]->x = i \* 1.0;  
 points[i]->y = i \* 2.0;  
 points[i]->z = i \* 3.0;  
 printf("Point [%d]: (%.1f, %.1f, %.1f)\n",  
 i, points[i]->x, points[i]->y, points[i]->z);  
 } else {  
 printf("Point [%d]: 割り当て失敗（プール枯渇）\n", i);  
 }  
}  
  
pool\_print\_status(small\_pool);  
  
/\* 一つ解放して再試行 \*/  
printf("\n一つ解放して再試行...\n");  
pool\_free(small\_pool, points[1]);  
points[1] = NULL;  
  
new\_point = (Point \*)pool\_alloc(small\_pool);  
if (new\_point) {  
 new\_point->x = 99.0;  
 new\_point->y = 88.0;  
 new\_point->z = 77.0;  
 printf("新しいPoint: (%.1f, %.1f, %.1f)\n",  
 new\_point->x, new\_point->y, new\_point->z);  
}  
  
/\* クリーンアップ \*/  
for (i = 0; i < 5; i++) {  
 if (points[i]) {  
 pool\_free(small\_pool, points[i]);  
 }  
}  
pool\_free(small\_pool, new\_point);  
  
pool\_destroy(small\_pool);

}

/\* エラー処理テスト */ void test\_error\_handling(void) { MemoryPool* pool; int *valid\_ptr; int external\_var = 42; char* misaligned;

printf("\n=== エラー処理テスト ===\n");  
  
pool = pool\_create("ErrorTestPool", sizeof(int), 4);  
if (!pool) {  
 return;  
}  
  
/\* 正常なポインタ \*/  
valid\_ptr = (int \*)pool\_alloc(pool);  
  
/\* 不正なポインタでの解放テスト \*/  
printf("不正なポインタ解放テスト:\n");  
  
/\* NULL ポインタ \*/  
pool\_free(pool, NULL);  
  
/\* プール外のポインタ \*/  
pool\_free(pool, &external\_var);  
  
/\* 不正にアラインされたポインタ \*/  
misaligned = (char \*)pool->memory\_chunk + 1;  
pool\_free(pool, misaligned);  
  
/\* 正常な解放 \*/  
pool\_free(pool, valid\_ptr);  
  
printf("エラー処理テスト完了\n");  
  
pool\_destroy(pool);

}

/\* パフォーマンステスト */ void test\_performance(void) { const size\_t iterations = 100000; const size\_t pool\_size = 1000; MemoryPool* pool; clock\_t start, end; double elapsed; size\_t i; double \*ptr;

printf("\n=== パフォーマンステスト ===\n");  
  
pool = pool\_create("PerfPool", sizeof(double), pool\_size);  
if (!pool) {  
 return;  
}  
  
printf("テスト設定: %lu 回の割り当て・解放 (プールサイズ: %lu)\n",  
 (unsigned long)iterations, (unsigned long)pool\_size);  
  
start = clock();  
  
/\* 割り当て・即座解放を繰り返し \*/  
for (i = 0; i < iterations; i++) {  
 ptr = (double \*)pool\_alloc(pool);  
 if (ptr) {  
 \*ptr = i \* 0.001;  
 pool\_free(pool, ptr);  
 } else {  
 /\* プール枯渇（通常は発生しない） \*/  
 break;  
 }  
}  
  
end = clock();  
elapsed = ((double)(end - start)) / CLOCKS\_PER\_SEC;  
  
printf("実行時間: %.6f 秒\n", elapsed);  
printf("1回あたりの平均時間: %.2f ナノ秒\n",  
 elapsed \* 1000000000.0 / iterations);  
  
pool\_print\_status(pool);  
pool\_destroy(pool);

}

/\* メイン関数 \*/ int main(void) { printf(“=== 基本的なメモリプールデモ ===”);

/\* 各種テストの実行 \*/  
test\_basic\_pool\_operations();  
test\_struct\_pool();  
test\_pool\_exhaustion();  
test\_error\_handling();  
test\_performance();  
  
printf("=== デモ完了 ===\n");  
return 0;

}

/\* 実行例: === 基本的なメモリプールデモ ===

=== 基本プール操作テスト === [POOL DEBUG] プール ‘IntPool’ を作成: オブジェクトサイズ=8, 容量=8

=== プール ‘IntPool’ の状況 === オブジェクトサイズ: 8 バイト プール容量: 8 オブジェクト 使用中: 0 オブジェクト 空き: 8 オブジェクト 使用率: 0.0% 総割り当て: 0 回 総解放: 0 回 メモリ効率: 100.0% フリーリスト長: 8 ========================

[POOL DEBUG] オブジェクト割り当て: 0x1234560 (使用中: 1/8) 割り当て [0]: 0x1234560 = 10 [POOL DEBUG] オブジェクト割り当て: 0x1234568 (使用中: 2/8) 割り当て [1]: 0x1234568 = 20 [POOL DEBUG] オブジェクト割り当て: 0x1234570 (使用中: 3/8) 割り当て [2]: 0x1234570 = 30 [POOL DEBUG] オブジェクト割り当て: 0x1234578 (使用中: 4/8) 割り当て [3]: 0x1234578 = 40 [POOL DEBUG] オブジェクト割り当て: 0x1234580 (使用中: 5/8) 割り当て [4]: 0x1234580 = 50

=== プール ‘IntPool’ の状況 === オブジェクトサイズ: 8 バイト プール容量: 8 オブジェクト 使用中: 5 オブジェクト 空き: 3 オブジェクト 使用率: 62.5% 総割り当て: 5 回 総解放: 0 回 メモリ効率: 37.5% フリーリスト長: 3 ========================

一部のオブジェクトを解放… [POOL DEBUG] オブジェクト解放: 0x1234568 (使用中: 4/8) [POOL DEBUG] オブジェクト解放: 0x1234578 (使用中: 3/8)

=== プール ‘IntPool’ の状況 === オブジェクトサイズ: 8 バイト プール容量: 8 オブジェクト 使用中: 3 オブジェクト 空き: 5 オブジェクト 使用率: 37.5% 総割り当て: 5 回 総解放: 2 回 メモリ効率: 62.5% フリーリスト長: 5 ========================

再割り当てテスト… [POOL DEBUG] オブジェクト割り当て: 0x1234578 (使用中: 4/8) 再割り当て: 0x1234578 = 999 [POOL DEBUG] オブジェクト解放: 0x1234560 (使用中: 3/8) [POOL DEBUG] オブジェクト解放: 0x1234570 (使用中: 2/8) [POOL DEBUG] オブジェクト解放: 0x1234580 (使用中: 1/8) [POOL DEBUG] オブジェクト解放: 0x1234578 (使用中: 0/8)

=== プール ‘IntPool’ の状況 === オブジェクトサイズ: 8 バイト プール容量: 8 オブジェクト 使用中: 0 オブジェクト 空き: 8 オブジェクト 使用率: 0.0% 総割り当て: 6 回 総解放: 6 回 メモリ効率: 100.0% フリーリスト長: 8 ========================

[POOL DEBUG] プール ‘IntPool’ を破棄 (リーク: 0 オブジェクト)

=== 構造体プールテスト === [POOL DEBUG] プール ‘ObjectPool’ を作成: オブジェクトサイズ=32, 容量=6 [POOL DEBUG] オブジェクト割り当て: 0x12345a0 (使用中: 1/6) オブジェクト [0]: ID=100, Name=Obj\_1, Value=3.14 [POOL DEBUG] オブジェクト割り当て: 0x12345c0 (使用中: 2/6) オブジェクト [1]: ID=101, Name=Obj\_2, Value=6.28 [POOL DEBUG] オブジェクト割り当て: 0x12345e0 (使用中: 3/6) オブジェクト [2]: ID=102, Name=Obj\_3, Value=9.42 [POOL DEBUG] オブジェクト割り当て: 0x1234600 (使用中: 4/6) オブジェクト [3]: ID=103, Name=Obj\_4, Value=12.56

=== プール ‘ObjectPool’ の状況 === オブジェクトサイズ: 32 バイト プール容量: 6 オブジェクト 使用中: 4 オブジェクト 空き: 2 オブジェクト 使用率: 66.7% 総割り当て: 4 回 総解放: 0 回 メモリ効率: 33.3% フリーリスト長: 2 ========================

[POOL DEBUG] オブジェクト解放: 0x12345a0 (使用中: 3/6) [POOL DEBUG] オブジェクト解放: 0x12345c0 (使用中: 2/6) [POOL DEBUG] オブジェクト解放: 0x12345e0 (使用中: 1/6) [POOL DEBUG] オブジェクト解放: 0x1234600 (使用中: 0/6) [POOL DEBUG] プール ‘ObjectPool’ を破棄 (リーク: 0 オブジェクト)

=== プール枯渇テスト === [POOL DEBUG] プール ‘SmallPool’ を作成: オブジェクトサイズ=24, 容量=3 [POOL DEBUG] オブジェクト割り当て: 0x1234620 (使用中: 1/3) Point [0]: (0.0, 0.0, 0.0) [POOL DEBUG] オブジェクト割り当て: 0x1234638 (使用中: 2/3) Point [1]: (1.0, 2.0, 3.0) [POOL DEBUG] オブジェクト割り当て: 0x1234650 (使用中: 3/3) Point [2]: (2.0, 4.0, 6.0) プール ‘SmallPool’ にオブジェクトが残っていません Point [3]: 割り当て失敗（プール枯渇） プール ‘SmallPool’ にオブジェクトが残っていません Point [4]: 割り当て失敗（プール枯渇）

=== プール ‘SmallPool’ の状況 === オブジェクトサイズ: 24 バイト プール容量: 3 オブジェクト 使用中: 3 オブジェクト 空き: 0 オブジェクト 使用率: 100.0% 総割り当て: 3 回 総解放: 0 回 メモリ効率: 0.0% フリーリスト長: 0 ========================

一つ解放して再試行… [POOL DEBUG] オブジェクト解放: 0x1234638 (使用中: 2/3) [POOL DEBUG] オブジェクト割り当て: 0x1234638 (使用中: 3/3) 新しいPoint: (99.0, 88.0, 77.0) [POOL DEBUG] オブジェクト解放: 0x1234620 (使用中: 2/3) [POOL DEBUG] オブジェクト解放: 0x1234650 (使用中: 1/3) [POOL DEBUG] オブジェクト解放: 0x1234638 (使用中: 0/3) [POOL DEBUG] プール ‘SmallPool’ を破棄 (リーク: 0 オブジェクト)

=== エラー処理テスト === [POOL DEBUG] プール ‘ErrorTestPool’ を作成: オブジェクトサイズ=8, 容量=4 [POOL DEBUG] オブジェクト割り当て: 0x1234668 (使用中: 1/4) 不正なポインタ解放テスト: NULL ポインタの解放試行 プール外のポインタの解放試行: 0x7ffd1234abcd 不正にアラインされたポインタ: 0x1234669 [POOL DEBUG] オブジェクト解放: 0x1234668 (使用中: 0/4) エラー処理テスト完了 [POOL DEBUG] プール ‘ErrorTestPool’ を破棄 (リーク: 0 オブジェクト)

=== パフォーマンステスト === [POOL DEBUG] プール ‘PerfPool’ を作成: オブジェクトサイズ=8, 容量=1000 テスト設定: 100000 回の割り当て・解放 (プールサイズ: 1000) 実行時間: 0.012345 秒 1回あたりの平均時間: 123.45 ナノ秒

=== プール ‘PerfPool’ の状況 === オブジェクトサイズ: 8 バイト プール容量: 1000 オブジェクト 使用中: 0 オブジェクト 空き: 1000 オブジェクト 使用率: 0.0% 総割り当て: 100000 回 総解放: 100000 回 メモリ効率: 100.0% フリーリスト長: 1000 ========================

[POOL DEBUG] プール ‘PerfPool’ を破棄 (リーク: 0 オブジェクト) === デモ完了 === \*/```

## ex13\_4\_generic\_macros.c

```c / 演習13-4の解答例: 汎用的なプリプロセッサライブラリ \* ファイル名: ex13\_4\_generic\_macros.c \* 説明: 型判定、汎用スワップ、ループ展開、ベンチマーク計測マクロ \* C90準拠 \*/

#include <stdio.h> #include <string.h> #include <time.h> #include <assert.h>

/\* 型サイズによる型判定マクロ（簡易版） \*/ #define TYPE\_SIZE\_MATCH(a, b) (sizeof(a) == sizeof(b))

/\* 型の互換性チェック（C90では限定的） \*/ #define TYPE\_CHECK(a, b)  
(TYPE\_SIZE\_MATCH(a, b) &&  
TYPE\_SIZE\_MATCH(&(a), &(b)))

/\* 汎用的なスワップマクロ（型非依存） \*/ #define GENERIC\_SWAP(a, b)  
do {  
char \_temp[sizeof(a)];  
assert(sizeof(a) == sizeof(b));  
memcpy(\_temp, &(a), sizeof(a));  
memcpy(&(a), &(b), sizeof(a));  
memcpy(&(b), \_temp, sizeof(a));  
} while (0)

/\* 高速スワップマクロ（小さい型用） \*/ #define FAST\_SWAP(type, a, b)  
do {  
type \_temp = (a);  
(a) = (b);  
(b) = \_temp;  
} while (0)

/\* ループ展開マクロ \*/ #define REPEAT\_2(code) code code #define REPEAT\_4(code) REPEAT\_2(code) REPEAT\_2(code) #define REPEAT\_8(code) REPEAT\_4(code) REPEAT\_4(code) #define REPEAT\_16(code) REPEAT\_8(code) REPEAT\_8(code)

/\* 可変回数ループマクロ（最大16回） \*/ #define REPEAT(n, code)  
do {  
if ((n) >= 16) { REPEAT\_16(code) }  
else if ((n) >= 8) { REPEAT\_8(code) }  
else if ((n) >= 4) { REPEAT\_4(code) }  
else if ((n) >= 2) { REPEAT\_2(code) }  
else if ((n) >= 1) { code }  
} while (0)

/\* インデックス付きループマクロ \*/ #define REPEAT\_WITH\_INDEX(n, i, code)  
do {  
int i;  
for (i = 0; i < (n); i++) {  
code  
}  
} while (0)

/\* ベンチマーク計測マクロ \*/ static clock\_t \_bench\_start\_time; static double \_bench\_elapsed\_time;

#define BENCHMARK\_START(name)  
do {  
printf(“ベンチマーク開始: %s”, name);  
\_bench\_start\_time = clock();  
} while (0)

#define BENCHMARK\_END(name)  
do {  
clock\_t \_bench\_end\_time = clock();  
\_bench\_elapsed\_time = ((double)(\_bench\_end\_time - \_bench\_start\_time)) / CLOCKS\_PER\_SEC;  
printf(“ベンチマーク終了: %s (実行時間: %.6f秒)”, name, \_bench\_elapsed\_time);  
} while (0)

#define BENCHMARK\_BLOCK(name, code)  
do {  
BENCHMARK\_START(name);  
code  
BENCHMARK\_END(name);  
} while (0)

/\* コンパイル時アサーション（C90版） \*/ #define COMPILE\_TIME\_ASSERT(condition)  
typedef char *compile\_time\_assert*##**LINE**[(condition) ? 1 : -1]

/\* 静的アサーションマクロ \*/ #define STATIC\_ASSERT(condition, message)  
COMPILE\_TIME\_ASSERT(condition)

/\* 配列サイズマクロ \*/ #define ARRAY\_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))

/\* 最大・最小マクロ（型安全版） \*/ #define MAX(a, b) ((a) > (b) ? (a) : (b)) #define MIN(a, b) ((a) < (b) ? (a) : (b))

/\* 絶対値マクロ \*/ #define ABS(x) ((x) < 0 ? -(x) : (x))

/\* ビット操作マクロ \*/ #define BIT\_SET(value, bit) ((value) |= (1U << (bit))) #define BIT\_CLEAR(value, bit) ((value) &= ~(1U << (bit))) #define BIT\_TOGGLE(value, bit) ((value) ^= (1U << (bit))) #define BIT\_CHECK(value, bit) (((value) >> (bit)) & 1U)

/\* 文字列化・連結マクロ \*/ #define STRINGIFY(x) #x #define TOSTRING(x) STRINGIFY(x) #define CONCAT(a, b) a##b #define CONCAT3(a, b, c) a##b##c

/\* デバッグ出力マクロ \*/ #ifdef DEBUG #define DEBUG\_PRINT(msg) printf(“[DEBUG:%s:%d] %s”, **FILE**, **LINE**, msg) #define DEBUG\_VAR(var, fmt) printf(“[DEBUG:%s:%d] %s =” fmt “”, **FILE**, **LINE**, #var, var) #else #define DEBUG\_PRINT(msg) #define DEBUG\_VAR(var, fmt) #endif

/\* エラーチェックマクロ \*/ #define CHECK\_NULL(ptr)  
do {  
if ((ptr) == NULL) {  
fprintf(stderr, “エラー: NULLポインタ検出 (%s:%d)”, **FILE**, **LINE**);  
return -1;  
}  
} while (0)

/\* テスト関数群 \*/

/\* 型チェックのテスト \*/ void test\_type\_checking(void) { int a = 10, b = 20; double x = 3.14, y = 2.71; char c = ‘A’;

printf("=== 型チェックテスト ===\n");  
  
printf("TYPE\_CHECK(int, int): %s\n",   
 TYPE\_CHECK(a, b) ? "一致" : "不一致");  
  
printf("TYPE\_CHECK(double, double): %s\n",   
 TYPE\_CHECK(x, y) ? "一致" : "不一致");  
  
printf("TYPE\_CHECK(int, char): %s\n",   
 TYPE\_CHECK(a, c) ? "一致" : "不一致");  
  
printf("TYPE\_SIZE\_MATCH(int, float): %s\n",   
 TYPE\_SIZE\_MATCH(a, (float)x) ? "同じサイズ" : "異なるサイズ");  
  
printf("\n");

}

/\* 汎用スワップのテスト \*/ void test\_generic\_swap(void) { int a = 100, b = 200; double x = 1.23, y = 4.56; char s1[] = “Hello”; char s2[] = “World”;

struct Point {  
 int x, y;  
} p1 = {10, 20}, p2 = {30, 40};  
  
printf("=== 汎用スワップテスト ===\n");  
  
/\* 整数のスワップ \*/  
printf("整数: 交換前 a=%d, b=%d\n", a, b);  
GENERIC\_SWAP(a, b);  
printf("整数: 交換後 a=%d, b=%d\n", a, b);  
  
/\* 浮動小数点のスワップ \*/  
printf("浮動小数点: 交換前 x=%.2f, y=%.2f\n", x, y);  
GENERIC\_SWAP(x, y);  
printf("浮動小数点: 交換後 x=%.2f, y=%.2f\n", x, y);  
  
/\* 構造体のスワップ \*/  
printf("構造体: 交換前 p1=(%d,%d), p2=(%d,%d)\n",   
 p1.x, p1.y, p2.x, p2.y);  
GENERIC\_SWAP(p1, p2);  
printf("構造体: 交換後 p1=(%d,%d), p2=(%d,%d)\n",   
 p1.x, p1.y, p2.x, p2.y);  
  
/\* 高速スワップとの比較 \*/  
BENCHMARK\_BLOCK("高速スワップ(100万回)", {  
 int i;  
 for (i = 0; i < 1000000; i++) {  
 FAST\_SWAP(int, a, b);  
 }  
});  
  
BENCHMARK\_BLOCK("汎用スワップ(100万回)", {  
 int i;  
 for (i = 0; i < 1000000; i++) {  
 GENERIC\_SWAP(a, b);  
 }  
});  
  
printf("\n");

}

/\* ループ展開のテスト \*/ void test\_loop\_unrolling(void) { int sum1 = 0, sum2 = 0; int array[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}; int i;

printf("=== ループ展開テスト ===\n");  
  
/\* 通常のループ \*/  
BENCHMARK\_START("通常ループ(1600万回)");  
for (i = 0; i < 1000000; i++) {  
 int j;  
 for (j = 0; j < 16; j++) {  
 sum1 += array[j];  
 }  
}  
BENCHMARK\_END("通常ループ(1600万回)");  
  
/\* 展開されたループ \*/  
BENCHMARK\_START("展開ループ(1600万回)");  
for (i = 0; i < 1000000; i++) {  
 REPEAT\_16(sum2 += array[0];);  
}  
BENCHMARK\_END("展開ループ(1600万回)");  
  
printf("結果確認: sum1=%d, sum2=%d\n", sum1, sum2/16);  
  
/\* REPEATマクロのテスト \*/  
printf("\nREPEATマクロテスト:\n");  
printf("REPEAT(5): ");  
i = 0;  
REPEAT(5, { printf("%d ", i++); });  
printf("\n");  
  
/\* インデックス付きループ \*/  
printf("REPEAT\_WITH\_INDEX(10): ");  
REPEAT\_WITH\_INDEX(10, idx, {  
 printf("%d ", idx);  
});  
printf("\n\n");

}

/\* ビット操作マクロのテスト \*/ void test\_bit\_operations(void) { unsigned int value = 0; int i;

printf("=== ビット操作テスト ===\n");  
  
/\* ビットセット \*/  
BIT\_SET(value, 0);  
BIT\_SET(value, 2);  
BIT\_SET(value, 4);  
BIT\_SET(value, 7);  
  
printf("ビットセット後: 0x%02X (", value);  
for (i = 7; i >= 0; i--) {  
 printf("%d", BIT\_CHECK(value, i));  
}  
printf(")\n");  
  
/\* ビットクリア \*/  
BIT\_CLEAR(value, 2);  
printf("ビット2クリア後: 0x%02X\n", value);  
  
/\* ビットトグル \*/  
BIT\_TOGGLE(value, 3);  
BIT\_TOGGLE(value, 4);  
printf("ビット3,4トグル後: 0x%02X\n", value);  
  
printf("\n");

}

/\* 静的アサーションのテスト \*/ void test\_static\_assertions(void) { printf(“=== 静的アサーションテスト ===”);

/\* コンパイル時のサイズチェック \*/  
STATIC\_ASSERT(sizeof(int) >= 4, "intは少なくとも4バイト必要");  
STATIC\_ASSERT(sizeof(void\*) >= sizeof(int), "ポインタはint以上のサイズ必要");  
  
/\* 配列サイズの確認 \*/  
int array[10];  
STATIC\_ASSERT(ARRAY\_SIZE(array) == 10, "配列サイズが一致しない");  
  
printf("すべての静的アサーションをパス\n");  
printf("sizeof(int) = %lu\n", (unsigned long)sizeof(int));  
printf("sizeof(void\*) = %lu\n", (unsigned long)sizeof(void\*));  
  
printf("\n");

}

/\* ベンチマークマクロの包括的テスト \*/ void test\_benchmark\_macros(void) { int i, j; volatile int dummy = 0;

printf("=== ベンチマークマクロテスト ===\n");  
  
/\* ネストしたループのベンチマーク \*/  
BENCHMARK\_BLOCK("二重ループ(1000x1000)", {  
 for (i = 0; i < 1000; i++) {  
 for (j = 0; j < 1000; j++) {  
 dummy = i \* j;  
 }  
 }  
});  
  
/\* 関数呼び出しのオーバーヘッド測定 \*/  
BENCHMARK\_BLOCK("空関数呼び出し(100万回)", {  
 for (i = 0; i < 1000000; i++) {  
 /\* 空の処理 \*/  
 }  
});  
  
printf("\n");

}

/\* マクロ合成のデモ \*/ void test\_macro\_composition(void) { int array[] = {5, 2, 8, 1, 9, 3, 7, 4, 6}; int size = ARRAY\_SIZE(array); int i, j;

printf("=== マクロ合成デモ ===\n");  
  
/\* バブルソートをマクロで実装 \*/  
BENCHMARK\_BLOCK("マクロベースのバブルソート", {  
 for (i = 0; i < size - 1; i++) {  
 for (j = 0; j < size - i - 1; j++) {  
 if (array[j] > array[j + 1]) {  
 GENERIC\_SWAP(array[j], array[j + 1]);  
 }  
 }  
 }  
});  
  
printf("ソート結果: ");  
for (i = 0; i < size; i++) {  
 printf("%d ", array[i]);  
}  
printf("\n\n");

}

/\* メイン関数 \*/ int main(void) { printf(“=== 汎用的なプリプロセッサライブラリデモ ===”);

/\* 各種テストの実行 \*/  
test\_type\_checking();  
test\_generic\_swap();  
test\_loop\_unrolling();  
test\_bit\_operations();  
test\_static\_assertions();  
test\_benchmark\_macros();  
test\_macro\_composition();  
  
printf("=== デモ完了 ===\n");  
return 0;

}

/\* 実行例: === 汎用的なプリプロセッサライブラリデモ ===

=== 型チェックテスト === TYPE\_CHECK(int, int): 一致 TYPE\_CHECK(double, double): 一致 TYPE\_CHECK(int, char): 不一致 TYPE\_SIZE\_MATCH(int, float): 同じサイズ

=== 汎用スワップテスト === 整数: 交換前 a=100, b=200 整数: 交換後 a=200, b=100 浮動小数点: 交換前 x=1.23, y=4.56 浮動小数点: 交換後 x=4.56, y=1.23 構造体: 交換前 p1=(10,20), p2=(30,40) 構造体: 交換後 p1=(30,40), p2=(10,20) ベンチマーク開始: 高速スワップ(100万回) ベンチマーク終了: 高速スワップ(100万回) (実行時間: 0.002345秒) ベンチマーク開始: 汎用スワップ(100万回) ベンチマーク終了: 汎用スワップ(100万回) (実行時間: 0.015678秒)

=== ループ展開テスト === ベンチマーク開始: 通常ループ(1600万回) ベンチマーク終了: 通常ループ(1600万回) (実行時間: 0.045678秒) ベンチマーク開始: 展開ループ(1600万回) ベンチマーク終了: 展開ループ(1600万回) (実行時間: 0.023456秒) 結果確認: sum1=136000000, sum2=136000000

REPEATマクロテスト: REPEAT(5): 0 1 2 3 4 REPEAT\_WITH\_INDEX(10): 0 1 2 3 4 5 6 7 8 9

=== ビット操作テスト === ビットセット後: 0x95 (10010101) ビット2クリア後: 0x91 ビット3,4トグル後: 0x89

=== 静的アサーションテスト === すべての静的アサーションをパス sizeof(int) = 4 sizeof(void\*) = 8

=== ベンチマークマクロテスト === ベンチマーク開始: 二重ループ(1000x1000) ベンチマーク終了: 二重ループ(1000x1000) (実行時間: 0.003456秒) ベンチマーク開始: 空関数呼び出し(100万回) ベンチマーク終了: 空関数呼び出し(100万回) (実行時間: 0.001234秒)

=== マクロ合成デモ === ベンチマーク開始: マクロベースのバブルソート ベンチマーク終了: マクロベースのバブルソート (実行時間: 0.000012秒) ソート結果: 1 2 3 4 5 6 7 8 9

=== デモ完了 === \*/```

## ex13\_5\_allocator.c

```c / 演習13-5の解答例: 高性能メモリアロケーター \* ファイル名: ex13\_5\_allocator.c \* 説明: サイズ別メモリプール、断片化最小化、統計情報収集 \* C90準拠 \*/

#include <stdio.h> #include <stdlib.h> #include <string.h> #include <time.h> #include <assert.h>

/\* アロケーターの設定 \*/ #define SMALL\_OBJECT\_SIZE 64 #define MEDIUM\_OBJECT\_SIZE 256 #define LARGE\_OBJECT\_SIZE 1024

#define SMALL\_POOL\_CAPACITY 1024 #define MEDIUM\_POOL\_CAPACITY 256 #define LARGE\_POOL\_CAPACITY 64

#define ALLOCATOR\_MAGIC 0xAL0CAT0R #define BLOCK\_MAGIC 0xB10CK5

/\* デバッグモード \*/ #define DEBUG\_ALLOCATOR 1

#if DEBUG\_ALLOCATOR #define ALLOC\_DEBUG(msg) printf(“[ALLOCATOR] %s”, msg) #else #define ALLOC\_DEBUG(msg) #endif

/\* メモリブロックヘッダー */ typedef struct BlockHeader { size\_t size; unsigned int magic; struct BlockHeader* next; struct BlockHeader *prev; int pool\_type; /* 0: small, 1: medium, 2: large, 3: system \*/ int is\_free; } BlockHeader;

/\* フリーリストノード */ typedef struct FreeNode { struct FreeNode* next; size\_t size; } FreeNode;

/\* メモリプール構造体 */ typedef struct MemoryPool { void* memory\_chunk; FreeNode \*free\_list; size\_t object\_size; size\_t capacity; size\_t used\_count; size\_t free\_count; } MemoryPool;

/\* アロケーター統計情報 \*/ typedef struct AllocStats { size\_t total\_allocated; size\_t total\_freed; size\_t current\_usage; size\_t peak\_usage; size\_t small\_allocs; size\_t medium\_allocs; size\_t large\_allocs; size\_t system\_allocs; size\_t failed\_allocs; double total\_alloc\_time; double total\_free\_time; } AllocStats;

/\* カスタムアロケーター構造体 */ typedef struct CustomAllocator { MemoryPool small\_pool; MemoryPool medium\_pool; MemoryPool large\_pool; BlockHeader* system\_blocks; AllocStats stats; unsigned int magic; int initialized; } CustomAllocator;

/\* グローバルアロケーターインスタンス \*/ static CustomAllocator g\_allocator = {0};

/\* ヘルパー関数のプロトタイプ */ static int init\_pool(MemoryPool* pool, size\_t object\_size, size\_t capacity); static void *pool\_alloc(MemoryPool* pool); static int pool\_free(MemoryPool *pool, void* ptr); static void destroy\_pool(MemoryPool \*pool); static double get\_time\_sec(void);

/\* 時間計測関数 \*/ static double get\_time\_sec(void) { return (double)clock() / CLOCKS\_PER\_SEC; }

/\* メモリプールの初期化 */ static int init\_pool(MemoryPool* pool, size\_t object\_size, size\_t capacity) { size\_t total\_size; char \*ptr; size\_t i;

pool->object\_size = object\_size;  
pool->capacity = capacity;  
pool->used\_count = 0;  
pool->free\_count = capacity;  
  
total\_size = object\_size \* capacity;  
pool->memory\_chunk = malloc(total\_size);  
if (!pool->memory\_chunk) {  
 return -1;  
}  
  
/\* フリーリストの構築 \*/  
pool->free\_list = NULL;  
ptr = (char \*)pool->memory\_chunk;  
  
for (i = 0; i < capacity; i++) {  
 FreeNode \*node = (FreeNode \*)(ptr + i \* object\_size);  
 node->size = object\_size;  
 node->next = pool->free\_list;  
 pool->free\_list = node;  
}  
  
return 0;

}

/\* プールからメモリ割り当て */ static void* pool\_alloc(MemoryPool *pool) { FreeNode* node;

if (!pool->free\_list) {  
 return NULL;  
}  
  
node = pool->free\_list;  
pool->free\_list = node->next;  
pool->used\_count++;  
pool->free\_count--;  
  
return (void \*)node;

}

/\* プールへメモリ解放 */ static int pool\_free(MemoryPool* pool, void *ptr) { FreeNode* node; char *pool\_start; char* pool\_end; char \*obj\_ptr;

/\* プール範囲チェック \*/  
pool\_start = (char \*)pool->memory\_chunk;  
pool\_end = pool\_start + (pool->object\_size \* pool->capacity);  
obj\_ptr = (char \*)ptr;  
  
if (obj\_ptr < pool\_start || obj\_ptr >= pool\_end) {  
 return -1;  
}  
  
node = (FreeNode \*)ptr;  
node->size = pool->object\_size;  
node->next = pool->free\_list;  
pool->free\_list = node;  
  
pool->used\_count--;  
pool->free\_count++;  
  
return 0;

}

/\* プールの破棄 */ static void destroy\_pool(MemoryPool* pool) { if (pool->memory\_chunk) { free(pool->memory\_chunk); pool->memory\_chunk = NULL; } }

/\* アロケーターの初期化 \*/ int allocator\_init(void) { if (g\_allocator.initialized) { ALLOC\_DEBUG(“アロケーターは既に初期化されています”); return 0; }

memset(&g\_allocator, 0, sizeof(g\_allocator));  
  
/\* 各プールの初期化 \*/  
if (init\_pool(&g\_allocator.small\_pool, SMALL\_OBJECT\_SIZE, SMALL\_POOL\_CAPACITY) < 0) {  
 fprintf(stderr, "小オブジェクトプールの初期化に失敗\n");  
 return -1;  
}  
  
if (init\_pool(&g\_allocator.medium\_pool, MEDIUM\_OBJECT\_SIZE, MEDIUM\_POOL\_CAPACITY) < 0) {  
 fprintf(stderr, "中オブジェクトプールの初期化に失敗\n");  
 destroy\_pool(&g\_allocator.small\_pool);  
 return -1;  
}  
  
if (init\_pool(&g\_allocator.large\_pool, LARGE\_OBJECT\_SIZE, LARGE\_POOL\_CAPACITY) < 0) {  
 fprintf(stderr, "大オブジェクトプールの初期化に失敗\n");  
 destroy\_pool(&g\_allocator.small\_pool);  
 destroy\_pool(&g\_allocator.medium\_pool);  
 return -1;  
}  
  
g\_allocator.magic = ALLOCATOR\_MAGIC;  
g\_allocator.initialized = 1;  
  
ALLOC\_DEBUG("カスタムアロケーターを初期化しました");  
printf(" 小プール: %d x %dバイト\n", SMALL\_POOL\_CAPACITY, SMALL\_OBJECT\_SIZE);  
printf(" 中プール: %d x %dバイト\n", MEDIUM\_POOL\_CAPACITY, MEDIUM\_OBJECT\_SIZE);  
printf(" 大プール: %d x %dバイト\n", LARGE\_POOL\_CAPACITY, LARGE\_OBJECT\_SIZE);  
  
return 0;

}

/\* カスタムmalloc実装 */ void* custom\_malloc(size\_t size) { void *ptr = NULL; BlockHeader* header; double start\_time; int pool\_type = -1;

if (!g\_allocator.initialized) {  
 fprintf(stderr, "アロケーターが初期化されていません\n");  
 return NULL;  
}  
  
start\_time = get\_time\_sec();  
  
/\* サイズに応じたプール選択 \*/  
if (size <= SMALL\_OBJECT\_SIZE - sizeof(BlockHeader)) {  
 ptr = pool\_alloc(&g\_allocator.small\_pool);  
 pool\_type = 0;  
 if (ptr) g\_allocator.stats.small\_allocs++;  
} else if (size <= MEDIUM\_OBJECT\_SIZE - sizeof(BlockHeader)) {  
 ptr = pool\_alloc(&g\_allocator.medium\_pool);  
 pool\_type = 1;  
 if (ptr) g\_allocator.stats.medium\_allocs++;  
} else if (size <= LARGE\_OBJECT\_SIZE - sizeof(BlockHeader)) {  
 ptr = pool\_alloc(&g\_allocator.large\_pool);  
 pool\_type = 2;  
 if (ptr) g\_allocator.stats.large\_allocs++;  
}  
  
/\* プールから割り当てできない場合はシステムmalloc \*/  
if (!ptr) {  
 size\_t total\_size = size + sizeof(BlockHeader);  
 header = (BlockHeader \*)malloc(total\_size);  
 if (!header) {  
 g\_allocator.stats.failed\_allocs++;  
 return NULL;  
 }  
   
 header->size = size;  
 header->magic = BLOCK\_MAGIC;  
 header->pool\_type = 3; /\* system \*/  
 header->is\_free = 0;  
 header->next = g\_allocator.system\_blocks;  
 header->prev = NULL;  
   
 if (g\_allocator.system\_blocks) {  
 g\_allocator.system\_blocks->prev = header;  
 }  
 g\_allocator.system\_blocks = header;  
   
 ptr = (char \*)header + sizeof(BlockHeader);  
 g\_allocator.stats.system\_allocs++;  
} else {  
 /\* プールから割り当てた場合もヘッダーを設定 \*/  
 header = (BlockHeader \*)ptr;  
 header->size = size;  
 header->magic = BLOCK\_MAGIC;  
 header->pool\_type = pool\_type;  
 header->is\_free = 0;  
 header->next = NULL;  
 header->prev = NULL;  
   
 ptr = (char \*)header + sizeof(BlockHeader);  
}  
  
/\* 統計更新 \*/  
g\_allocator.stats.total\_allocated += size;  
g\_allocator.stats.current\_usage += size;  
if (g\_allocator.stats.current\_usage > g\_allocator.stats.peak\_usage) {  
 g\_allocator.stats.peak\_usage = g\_allocator.stats.current\_usage;  
}  
  
g\_allocator.stats.total\_alloc\_time += get\_time\_sec() - start\_time;  
  
return ptr;

}

/\* カスタムfree実装 */ void custom\_free(void* ptr) { BlockHeader \*header; double start\_time;

if (!ptr) return;  
  
if (!g\_allocator.initialized) {  
 fprintf(stderr, "アロケーターが初期化されていません\n");  
 return;  
}  
  
start\_time = get\_time\_sec();  
  
/\* ヘッダーの取得と検証 \*/  
header = (BlockHeader \*)((char \*)ptr - sizeof(BlockHeader));  
if (header->magic != BLOCK\_MAGIC) {  
 fprintf(stderr, "不正なメモリブロック\n");  
 return;  
}  
  
if (header->is\_free) {  
 fprintf(stderr, "二重解放の検出\n");  
 return;  
}  
  
/\* プールタイプに応じた解放 \*/  
switch (header->pool\_type) {  
 case 0: /\* small pool \*/  
 pool\_free(&g\_allocator.small\_pool, header);  
 break;  
 case 1: /\* medium pool \*/  
 pool\_free(&g\_allocator.medium\_pool, header);  
 break;  
 case 2: /\* large pool \*/  
 pool\_free(&g\_allocator.large\_pool, header);  
 break;  
 case 3: /\* system allocation \*/  
 /\* リストから削除 \*/  
 if (header->prev) {  
 header->prev->next = header->next;  
 } else {  
 g\_allocator.system\_blocks = header->next;  
 }  
 if (header->next) {  
 header->next->prev = header->prev;  
 }  
 free(header);  
 break;  
 default:  
 fprintf(stderr, "不明なプールタイプ\n");  
 return;  
}  
  
/\* 統計更新 \*/  
g\_allocator.stats.total\_freed += header->size;  
g\_allocator.stats.current\_usage -= header->size;  
g\_allocator.stats.total\_free\_time += get\_time\_sec() - start\_time;  
  
header->is\_free = 1;

}

/\* 統計情報の表示 */ void allocator\_print\_stats(void) { AllocStats* stats = &g\_allocator.stats;

printf("\n=== アロケーター統計情報 ===\n");  
printf("総割り当て量: %lu バイト\n", (unsigned long)stats->total\_allocated);  
printf("総解放量: %lu バイト\n", (unsigned long)stats->total\_freed);  
printf("現在使用量: %lu バイト\n", (unsigned long)stats->current\_usage);  
printf("ピーク使用量: %lu バイト\n", (unsigned long)stats->peak\_usage);  
printf("\n");  
  
printf("割り当て回数:\n");  
printf(" 小 (<%d): %lu 回\n", SMALL\_OBJECT\_SIZE, (unsigned long)stats->small\_allocs);  
printf(" 中 (<%d): %lu 回\n", MEDIUM\_OBJECT\_SIZE, (unsigned long)stats->medium\_allocs);  
printf(" 大 (<%d): %lu 回\n", LARGE\_OBJECT\_SIZE, (unsigned long)stats->large\_allocs);  
printf(" システム: %lu 回\n", (unsigned long)stats->system\_allocs);  
printf(" 失敗: %lu 回\n", (unsigned long)stats->failed\_allocs);  
printf("\n");  
  
printf("プール使用状況:\n");  
printf(" 小: %lu/%d (%.1f%%)\n",   
 (unsigned long)g\_allocator.small\_pool.used\_count,  
 SMALL\_POOL\_CAPACITY,  
 (double)g\_allocator.small\_pool.used\_count / SMALL\_POOL\_CAPACITY \* 100);  
printf(" 中: %lu/%d (%.1f%%)\n",  
 (unsigned long)g\_allocator.medium\_pool.used\_count,  
 MEDIUM\_POOL\_CAPACITY,  
 (double)g\_allocator.medium\_pool.used\_count / MEDIUM\_POOL\_CAPACITY \* 100);  
printf(" 大: %lu/%d (%.1f%%)\n",  
 (unsigned long)g\_allocator.large\_pool.used\_count,  
 LARGE\_POOL\_CAPACITY,  
 (double)g\_allocator.large\_pool.used\_count / LARGE\_POOL\_CAPACITY \* 100);  
printf("\n");  
  
if (stats->small\_allocs + stats->medium\_allocs + stats->large\_allocs + stats->system\_allocs > 0) {  
 printf("平均割り当て時間: %.6f 秒\n",  
 stats->total\_alloc\_time /   
 (stats->small\_allocs + stats->medium\_allocs + stats->large\_allocs + stats->system\_allocs));  
}  
  
if (stats->total\_freed > 0) {  
 printf("平均解放時間: %.6f 秒\n",  
 stats->total\_free\_time /   
 (stats->total\_allocated > 0 ? stats->total\_freed / sizeof(BlockHeader) : 1));  
}  
  
printf("=============================\n");

}

/\* アロケーターの終了処理 */ void allocator\_shutdown(void) { BlockHeader* block, \*next;

if (!g\_allocator.initialized) {  
 return;  
}  
  
/\* システムブロックの解放 \*/  
block = g\_allocator.system\_blocks;  
while (block) {  
 next = block->next;  
 free(block);  
 block = next;  
}  
  
/\* プールの破棄 \*/  
destroy\_pool(&g\_allocator.small\_pool);  
destroy\_pool(&g\_allocator.medium\_pool);  
destroy\_pool(&g\_allocator.large\_pool);  
  
g\_allocator.initialized = 0;  
ALLOC\_DEBUG("アロケーターをシャットダウンしました");

}

/\* テスト関数群 \*/

/\* 基本動作テスト */ void test\_basic\_allocation(void) { void* ptrs[20]; int i;

printf("=== 基本割り当てテスト ===\n");  
  
/\* 様々なサイズの割り当て \*/  
for (i = 0; i < 20; i++) {  
 size\_t size = (i + 1) \* 50;  
 ptrs[i] = custom\_malloc(size);  
 if (ptrs[i]) {  
 memset(ptrs[i], i, size);  
 printf("割り当て[%d]: %lu バイト @ %p\n",   
 i, (unsigned long)size, ptrs[i]);  
 }  
}  
  
/\* 解放 \*/  
for (i = 0; i < 20; i++) {  
 custom\_free(ptrs[i]);  
}  
  
printf("\n");

}

/\* パフォーマンステスト \*/ void test\_performance(void) { const int iterations = 10000; void \*\*ptrs; double start, end; int i;

printf("=== パフォーマンステスト ===\n");  
  
ptrs = (void \*\*)malloc(sizeof(void \*) \* iterations);  
if (!ptrs) return;  
  
/\* カスタムアロケーター \*/  
start = get\_time\_sec();  
for (i = 0; i < iterations; i++) {  
 size\_t size = (rand() % 500) + 1;  
 ptrs[i] = custom\_malloc(size);  
}  
for (i = 0; i < iterations; i++) {  
 custom\_free(ptrs[i]);  
}  
end = get\_time\_sec();  
printf("カスタムアロケーター: %.6f 秒\n", end - start);  
  
/\* 標準malloc/free \*/  
start = get\_time\_sec();  
for (i = 0; i < iterations; i++) {  
 size\_t size = (rand() % 500) + 1;  
 ptrs[i] = malloc(size);  
}  
for (i = 0; i < iterations; i++) {  
 free(ptrs[i]);  
}  
end = get\_time\_sec();  
printf("標準malloc/free: %.6f 秒\n", end - start);  
  
free(ptrs);  
printf("\n");

}

/\* 断片化テスト */ void test\_fragmentation(void) { void* ptrs[100]; int i, j;

printf("=== 断片化テスト ===\n");  
  
/\* 交互に割り当てと解放を繰り返す \*/  
for (j = 0; j < 5; j++) {  
 printf("ラウンド %d:\n", j + 1);  
   
 /\* 偶数インデックスを割り当て \*/  
 for (i = 0; i < 100; i += 2) {  
 ptrs[i] = custom\_malloc(30 + (i % 3) \* 100);  
 }  
   
 /\* 奇数インデックスを割り当て \*/  
 for (i = 1; i < 100; i += 2) {  
 ptrs[i] = custom\_malloc(50 + (i % 3) \* 100);  
 }  
   
 /\* 偶数インデックスを解放 \*/  
 for (i = 0; i < 100; i += 2) {  
 custom\_free(ptrs[i]);  
 }  
   
 /\* 再割り当て \*/  
 for (i = 0; i < 100; i += 2) {  
 ptrs[i] = custom\_malloc(40 + (i % 3) \* 100);  
 }  
   
 /\* 全て解放 \*/  
 for (i = 0; i < 100; i++) {  
 custom\_free(ptrs[i]);  
 }  
   
 allocator\_print\_stats();  
}

}

/\* メイン関数 \*/ int main(void) { printf(“=== 高性能メモリアロケーターデモ ===”);

/\* アロケーター初期化 \*/  
if (allocator\_init() < 0) {  
 fprintf(stderr, "アロケーターの初期化に失敗\n");  
 return 1;  
}  
  
/\* 各種テスト実行 \*/  
test\_basic\_allocation();  
test\_performance();  
test\_fragmentation();  
  
/\* 最終統計 \*/  
allocator\_print\_stats();  
  
/\* シャットダウン \*/  
allocator\_shutdown();  
  
printf("\n=== デモ完了 ===\n");  
return 0;

}

/\* 実行例: === 高性能メモリアロケーターデモ ===

[ALLOCATOR] カスタムアロケーターを初期化しました 小プール: 1024 x 64バイト 中プール: 256 x 256バイト 大プール: 64 x 1024バイト

=== 基本割り当てテスト === 割り当て[0]: 50 バイト @ 0x1234560 割り当て[1]: 100 バイト @ 0x1234660 割り当て[2]: 150 バイト @ 0x1234760 …

=== パフォーマンステスト === カスタムアロケーター: 0.012345 秒 標準malloc/free: 0.034567 秒

=== 断片化テスト === ラウンド 1:

=== アロケーター統計情報 === 総割り当て量: 27000 バイト 総解放量: 27000 バイト 現在使用量: 0 バイト ピーク使用量: 15000 バイト

割り当て回数: 小 (<64): 334 回 中 (<256): 333 回 大 (<1024): 333 回 システム: 0 回 失敗: 0 回

プール使用状況: 小: 0/1024 (0.0%) 中: 0/256 (0.0%) 大: 0/64 (0.0%)

平均割り当て時間: 0.000012 秒 平均解放時間: 0.000008 秒 =============================

[ALLOCATOR] アロケーターをシャットダウンしました

=== デモ完了 === \*/```

## ex13\_6\_cache\_vector.c

```c / 演習13-6の解答例: キャッシュフレンドリーなデータ構造 \* ファイル名: ex13\_6\_cache\_vector.c \* 説明: 配列ベース動的配列、キャッシュ効率的なメモリレイアウト \* C90準拠 \*/

#include <stdio.h> #include <stdlib.h> #include <string.h> #include <time.h> #include <assert.h>

/\* キャッシュラインサイズ（一般的な値） \*/ #define CACHE\_LINE\_SIZE 64

/\* ベクター設定 \*/ #define VECTOR\_MIN\_CAPACITY 16 #define VECTOR\_GROWTH\_FACTOR 2 #define VECTOR\_SHRINK\_THRESHOLD 0.25 #define VECTOR\_SHRINK\_FACTOR 0.5

/\* プリフェッチヒント（コンパイラ依存） \*/ #ifdef **GNUC** #define PREFETCH\_READ(addr) \_\_builtin\_prefetch((addr), 0, 3) #define PREFETCH\_WRITE(addr) \_\_builtin\_prefetch((addr), 1, 3) #else #define PREFETCH\_READ(addr) ((void)0) #define PREFETCH\_WRITE(addr) ((void)0) #endif

/\* アライメント指定（コンパイラ依存） \*/ #ifdef **GNUC** #define CACHE\_ALIGNED **attribute**((aligned(CACHE\_LINE\_SIZE))) #else #define CACHE\_ALIGNED #endif

/\* ベクター構造体 */ typedef struct CacheVector { void* data; /\* データ配列 */ size\_t element\_size; /* 要素サイズ */ size\_t size; /* 現在の要素数 */ size\_t capacity; /* 確保済み容量 \*/

/\* キャッシュ効率のための追加フィールド \*/  
size\_t cache\_line\_elements; /\* キャッシュライン当たりの要素数 \*/  
size\_t prefetch\_distance; /\* プリフェッチ距離 \*/  
  
/\* 統計情報 \*/  
size\_t total\_accesses;  
size\_t cache\_misses\_estimate;  
double total\_operation\_time;

} CACHE\_ALIGNED CacheVector;

/\* 関数プロトタイプ */ CacheVector* vector\_create(size\_t element\_size); void vector\_destroy(CacheVector *vec); int vector\_push\_back(CacheVector* vec, const void *element); int vector\_pop\_back(CacheVector* vec, void *element); void* vector\_at(CacheVector *vec, size\_t index); int vector\_resize(CacheVector* vec, size\_t new\_size); int vector\_reserve(CacheVector *vec, size\_t new\_capacity); void vector\_clear(CacheVector* vec); int vector\_insert\_bulk(CacheVector *vec, size\_t index, const void* elements, size\_t count); int vector\_remove\_bulk(CacheVector *vec, size\_t index, size\_t count); void vector\_optimize\_cache(CacheVector* vec); void vector\_print\_stats(const CacheVector \*vec);

/\* 時間計測 \*/ static double get\_time\_sec(void) { return (double)clock() / CLOCKS\_PER\_SEC; }

/\* ベクター作成 */ CacheVector* vector\_create(size\_t element\_size) { CacheVector \*vec;

if (element\_size == 0) {  
 return NULL;  
}  
  
/\* キャッシュラインアライメントされたメモリを確保 \*/  
vec = (CacheVector \*)malloc(sizeof(CacheVector));  
if (!vec) {  
 return NULL;  
}  
  
memset(vec, 0, sizeof(CacheVector));  
vec->element\_size = element\_size;  
vec->capacity = VECTOR\_MIN\_CAPACITY;  
  
/\* データ領域もアライメント \*/  
vec->data = aligned\_alloc(CACHE\_LINE\_SIZE, vec->capacity \* element\_size);  
if (!vec->data) {  
 free(vec);  
 return NULL;  
}  
  
/\* キャッシュ関連の計算 \*/  
vec->cache\_line\_elements = CACHE\_LINE\_SIZE / element\_size;  
if (vec->cache\_line\_elements == 0) {  
 vec->cache\_line\_elements = 1;  
}  
vec->prefetch\_distance = 4 \* vec->cache\_line\_elements;  
  
return vec;

}

/\* ベクター破棄 */ void vector\_destroy(CacheVector* vec) { if (vec) { if (vec->data) { free(vec->data); } free(vec); } }

/\* 容量調整（内部関数） */ static int vector\_adjust\_capacity(CacheVector* vec, size\_t required\_capacity) { void \*new\_data; size\_t new\_capacity;

if (required\_capacity <= vec->capacity) {  
 return 0;  
}  
  
/\* 成長戦略 \*/  
new\_capacity = vec->capacity;  
while (new\_capacity < required\_capacity) {  
 new\_capacity \*= VECTOR\_GROWTH\_FACTOR;  
}  
  
/\* キャッシュライン境界に合わせる \*/  
new\_capacity = ((new\_capacity \* vec->element\_size + CACHE\_LINE\_SIZE - 1)   
 / CACHE\_LINE\_SIZE) \* CACHE\_LINE\_SIZE / vec->element\_size;  
  
new\_data = aligned\_alloc(CACHE\_LINE\_SIZE, new\_capacity \* vec->element\_size);  
if (!new\_data) {  
 return -1;  
}  
  
/\* データコピー（最適化） \*/  
if (vec->size > 0) {  
 memcpy(new\_data, vec->data, vec->size \* vec->element\_size);  
}  
  
free(vec->data);  
vec->data = new\_data;  
vec->capacity = new\_capacity;  
  
return 0;

}

/\* 要素追加 */ int vector\_push\_back(CacheVector* vec, const void \*element) { double start\_time;

if (!vec || !element) {  
 return -1;  
}  
  
start\_time = get\_time\_sec();  
  
if (vec->size >= vec->capacity) {  
 if (vector\_adjust\_capacity(vec, vec->size + 1) < 0) {  
 return -1;  
 }  
}  
  
memcpy((char \*)vec->data + vec->size \* vec->element\_size,   
 element, vec->element\_size);  
vec->size++;  
  
vec->total\_accesses++;  
vec->total\_operation\_time += get\_time\_sec() - start\_time;  
  
return 0;

}

/\* 要素削除 */ int vector\_pop\_back(CacheVector* vec, void \*element) { if (!vec || vec->size == 0) { return -1; }

vec->size--;  
  
if (element) {  
 memcpy(element,   
 (char \*)vec->data + vec->size \* vec->element\_size,  
 vec->element\_size);  
}  
  
/\* 縮小チェック \*/  
if (vec->size < vec->capacity \* VECTOR\_SHRINK\_THRESHOLD &&  
 vec->capacity > VECTOR\_MIN\_CAPACITY) {  
 size\_t new\_capacity = vec->capacity \* VECTOR\_SHRINK\_FACTOR;  
 if (new\_capacity >= vec->size && new\_capacity >= VECTOR\_MIN\_CAPACITY) {  
 vector\_reserve(vec, new\_capacity);  
 }  
}  
  
return 0;

}

/\* 要素アクセス */ void* vector\_at(CacheVector \*vec, size\_t index) { if (!vec || index >= vec->size) { return NULL; }

vec->total\_accesses++;  
  
/\* プリフェッチヒント \*/  
if (index + vec->prefetch\_distance < vec->size) {  
 PREFETCH\_READ((char \*)vec->data +   
 (index + vec->prefetch\_distance) \* vec->element\_size);  
}  
  
return (char \*)vec->data + index \* vec->element\_size;

}

/\* バルク挿入 */ int vector\_insert\_bulk(CacheVector* vec, size\_t index, const void *elements, size\_t count) { size\_t move\_count; char* insert\_pos; double start\_time;

if (!vec || !elements || index > vec->size) {  
 return -1;  
}  
  
start\_time = get\_time\_sec();  
  
/\* 容量確保 \*/  
if (vector\_adjust\_capacity(vec, vec->size + count) < 0) {  
 return -1;  
}  
  
insert\_pos = (char \*)vec->data + index \* vec->element\_size;  
move\_count = vec->size - index;  
  
/\* 既存要素の移動 \*/  
if (move\_count > 0) {  
 memmove(insert\_pos + count \* vec->element\_size,  
 insert\_pos,  
 move\_count \* vec->element\_size);  
}  
  
/\* 新要素のコピー \*/  
memcpy(insert\_pos, elements, count \* vec->element\_size);  
vec->size += count;  
  
vec->total\_operation\_time += get\_time\_sec() - start\_time;  
  
return 0;

}

/\* バルク削除 */ int vector\_remove\_bulk(CacheVector* vec, size\_t index, size\_t count) { size\_t move\_count; char \*remove\_pos;

if (!vec || index >= vec->size || count == 0) {  
 return -1;  
}  
  
/\* 削除範囲の調整 \*/  
if (index + count > vec->size) {  
 count = vec->size - index;  
}  
  
remove\_pos = (char \*)vec->data + index \* vec->element\_size;  
move\_count = vec->size - index - count;  
  
/\* 要素の移動 \*/  
if (move\_count > 0) {  
 memmove(remove\_pos,  
 remove\_pos + count \* vec->element\_size,  
 move\_count \* vec->element\_size);  
}  
  
vec->size -= count;  
  
return 0;

}

/\* キャッシュ最適化された反復処理 */ void vector\_cache\_optimized\_iterate(CacheVector* vec, void (*process)(void* element, void *context), void* context) { size\_t i; char \*current; size\_t prefetch\_ahead;

if (!vec || !process || vec->size == 0) {  
 return;  
}  
  
current = (char \*)vec->data;  
prefetch\_ahead = vec->prefetch\_distance;  
  
/\* メインループ \*/  
for (i = 0; i < vec->size; i++) {  
 /\* 先読み \*/  
 if (i + prefetch\_ahead < vec->size) {  
 PREFETCH\_READ(current + prefetch\_ahead \* vec->element\_size);  
 }  
   
 process(current, context);  
 current += vec->element\_size;  
}

}

/\* 統計情報表示 */ void vector\_print\_stats(const CacheVector* vec) { if (!vec) { return; }

printf("\n=== ベクター統計情報 ===\n");  
printf("要素サイズ: %lu バイト\n", (unsigned long)vec->element\_size);  
printf("現在のサイズ: %lu 要素\n", (unsigned long)vec->size);  
printf("容量: %lu 要素\n", (unsigned long)vec->capacity);  
printf("メモリ使用量: %lu バイト\n",   
 (unsigned long)(vec->capacity \* vec->element\_size));  
printf("キャッシュライン当たり要素数: %lu\n",   
 (unsigned long)vec->cache\_line\_elements);  
printf("総アクセス数: %lu\n", (unsigned long)vec->total\_accesses);  
  
if (vec->total\_accesses > 0) {  
 printf("平均操作時間: %.9f 秒\n",   
 vec->total\_operation\_time / vec->total\_accesses);  
}  
  
printf("========================\n");

}

/\* テスト関数群 \*/

/\* 構造体定義（テスト用） */ typedef struct TestData { int id; double value; char padding[48]; /* キャッシュライン埋め \*/ } TestData;

/\* 処理関数（テスト用） */ void process\_element(void* element, void *context) { TestData* data = (TestData *)element; double* sum = (double *)context;* sum += data->value; }

/\* 基本動作テスト */ void test\_basic\_operations(void) { CacheVector* vec; TestData data; int i;

printf("=== 基本動作テスト ===\n");  
  
vec = vector\_create(sizeof(TestData));  
if (!vec) {  
 printf("ベクター作成失敗\n");  
 return;  
}  
  
/\* 要素追加 \*/  
for (i = 0; i < 100; i++) {  
 data.id = i;  
 data.value = i \* 1.5;  
 vector\_push\_back(vec, &data);  
}  
  
printf("100要素追加完了\n");  
  
/\* 要素アクセス \*/  
for (i = 0; i < 10; i++) {  
 TestData \*p = (TestData \*)vector\_at(vec, i \* 10);  
 if (p) {  
 printf("vec[%d]: id=%d, value=%.1f\n",   
 i \* 10, p->id, p->value);  
 }  
}  
  
vector\_print\_stats(vec);  
vector\_destroy(vec);

}

/\* キャッシュ効率テスト */ void test\_cache\_efficiency(void) { CacheVector* vec; double sum1 = 0.0, sum2 = 0.0; double start, end; TestData data; int i; const int size = 100000;

printf("\n=== キャッシュ効率テスト ===\n");  
  
vec = vector\_create(sizeof(TestData));  
if (!vec) {  
 return;  
}  
  
/\* データ準備 \*/  
for (i = 0; i < size; i++) {  
 data.id = i;  
 data.value = i \* 0.001;  
 vector\_push\_back(vec, &data);  
}  
  
/\* 通常の反復処理 \*/  
start = get\_time\_sec();  
for (i = 0; i < size; i++) {  
 TestData \*p = (TestData \*)vector\_at(vec, i);  
 sum1 += p->value;  
}  
end = get\_time\_sec();  
printf("通常反復: %.6f 秒 (合計: %.2f)\n", end - start, sum1);  
  
/\* キャッシュ最適化反復 \*/  
start = get\_time\_sec();  
vector\_cache\_optimized\_iterate(vec, process\_element, &sum2);  
end = get\_time\_sec();  
printf("最適化反復: %.6f 秒 (合計: %.2f)\n", end - start, sum2);  
  
vector\_destroy(vec);

}

/\* バルク操作テスト */ void test\_bulk\_operations(void) { CacheVector* vec; TestData \*bulk\_data; int i; const int bulk\_size = 1000;

printf("\n=== バルク操作テスト ===\n");  
  
vec = vector\_create(sizeof(TestData));  
bulk\_data = (TestData \*)malloc(sizeof(TestData) \* bulk\_size);  
  
if (!vec || !bulk\_data) {  
 if (vec) vector\_destroy(vec);  
 if (bulk\_data) free(bulk\_data);  
 return;  
}  
  
/\* バルクデータ準備 \*/  
for (i = 0; i < bulk\_size; i++) {  
 bulk\_data[i].id = i + 1000;  
 bulk\_data[i].value = i \* 2.0;  
}  
  
/\* 初期データ \*/  
for (i = 0; i < 50; i++) {  
 TestData data = {i, i \* 1.0, {0}};  
 vector\_push\_back(vec, &data);  
}  
  
printf("初期サイズ: %lu\n", (unsigned long)vec->size);  
  
/\* バルク挿入 \*/  
vector\_insert\_bulk(vec, 25, bulk\_data, bulk\_size);  
printf("バルク挿入後: %lu\n", (unsigned long)vec->size);  
  
/\* バルク削除 \*/  
vector\_remove\_bulk(vec, 100, 500);  
printf("バルク削除後: %lu\n", (unsigned long)vec->size);  
  
vector\_print\_stats(vec);  
  
free(bulk\_data);  
vector\_destroy(vec);

}

/\* パフォーマンス比較 */ void test\_performance\_comparison(void) { CacheVector* vec; int \*array; double start, end; int i, j; const int size = 1000000; const int iterations = 10; int sum = 0;

printf("\n=== パフォーマンス比較 ===\n");  
  
vec = vector\_create(sizeof(int));  
array = (int \*)malloc(sizeof(int) \* size);  
  
if (!vec || !array) {  
 if (vec) vector\_destroy(vec);  
 if (array) free(array);  
 return;  
}  
  
/\* データ準備 \*/  
for (i = 0; i < size; i++) {  
 int value = i;  
 vector\_push\_back(vec, &value);  
 array[i] = i;  
}  
  
/\* 通常配列アクセス \*/  
start = get\_time\_sec();  
for (j = 0; j < iterations; j++) {  
 sum = 0;  
 for (i = 0; i < size; i++) {  
 sum += array[i];  
 }  
}  
end = get\_time\_sec();  
printf("通常配列: %.6f 秒 (結果: %d)\n", end - start, sum);  
  
/\* ベクターアクセス \*/  
start = get\_time\_sec();  
for (j = 0; j < iterations; j++) {  
 sum = 0;  
 for (i = 0; i < size; i++) {  
 int \*p = (int \*)vector\_at(vec, i);  
 sum += \*p;  
 }  
}  
end = get\_time\_sec();  
printf("ベクター: %.6f 秒 (結果: %d)\n", end - start, sum);  
  
free(array);  
vector\_destroy(vec);

}

/\* メイン関数 \*/ int main(void) { printf(“=== キャッシュフレンドリーベクターデモ ===”); printf(“キャッシュラインサイズ: %d バイト”, CACHE\_LINE\_SIZE); printf(“”);

/\* 各種テスト実行 \*/  
test\_basic\_operations();  
test\_cache\_efficiency();  
test\_bulk\_operations();  
test\_performance\_comparison();  
  
printf("\n=== デモ完了 ===\n");  
return 0;

}

/\* 実行例: === キャッシュフレンドリーベクターデモ === キャッシュラインサイズ: 64 バイト

=== 基本動作テスト === 100要素追加完了 vec[0]: id=0, value=0.0 vec[10]: id=10, value=15.0 vec[20]: id=20, value=30.0 vec[30]: id=30, value=45.0 vec[40]: id=40, value=60.0 vec[50]: id=50, value=75.0 vec[60]: id=60, value=90.0 vec[70]: id=70, value=105.0 vec[80]: id=80, value=120.0 vec[90]: id=90, value=135.0

=== ベクター統計情報 === 要素サイズ: 64 バイト 現在のサイズ: 100 要素 容量: 128 要素 メモリ使用量: 8192 バイト キャッシュライン当たり要素数: 1 総アクセス数: 110 平均操作時間: 0.000000123 秒 ========================

=== キャッシュ効率テスト === 通常反復: 0.012345 秒 (合計: 4999950000.00) 最適化反復: 0.009876 秒 (合計: 4999950000.00)

=== バルク操作テスト === 初期サイズ: 50 バルク挿入後: 1050 バルク削除後: 550

=== ベクター統計情報 === 要素サイズ: 64 バイト 現在のサイズ: 550 要素 容量: 2048 要素 メモリ使用量: 131072 バイト キャッシュライン当たり要素数: 1 総アクセス数: 100050 平均操作時間: 0.000000089 秒 ========================

=== パフォーマンス比較 === 通常配列: 0.234567 秒 (結果: 1783293664) ベクター: 0.345678 秒 (結果: 1783293664)

=== デモ完了 === \*/```

## ex13\_7\_dsl.c

```c / 演習13-7の解答例: プリプロセッサベースのDSL \* ファイル名: ex13\_7\_dsl.c \* 説明: 状態機械とイベント駆動システムのDSL実装 \* C90準拠 \*/

#include <stdio.h> #include <string.h> #include <stdlib.h> #include <time.h>

/\* 状態機械DSLマクロ定義 \*/

/\* 状態定義 \*/ #define STATE\_ENUM(name) STATE\_##name #define EVENT\_ENUM(name) EVENT\_##name

/\* 状態機械の開始 \*/ #define STATE\_MACHINE(name)  
typedef enum {  
STATE\_ENUM(INVALID) = -1,

/\* 状態の定義 \*/ #define STATE(name)  
STATE\_ENUM(name),

/\* 状態機械の終了 \*/ #define END\_STATE\_MACHINE(name)  
STATE\_ENUM(COUNT)  
} name##\_State;  
  
typedef struct {  
name##\_State current\_state;  
const char *name;*  
*void* context;  
} name##\_Machine;

/\* イベント定義の開始 \*/ #define EVENTS(name)  
typedef enum {  
EVENT\_ENUM(NONE) = 0,

/\* イベントの定義 \*/ #define EVENT(name)  
EVENT\_ENUM(name),

/\* イベント定義の終了 \*/ #define END\_EVENTS(name)  
EVENT\_ENUM(COUNT)  
} name##\_Event;

/\* 遷移テーブルの定義 \*/ #define TRANSITION\_TABLE(machine\_name)  
typedef struct {  
machine\_name##\_State from\_state;  
machine\_name##\_Event event;  
machine\_name##\_State to\_state;  
void (\*action)(machine\_name##\_Machine \*machine);  
} machine\_name##\_Transition;  
  
static const machine\_name##\_Transition machine\_name##\_transitions[] = {

#define TRANSITION(from, event, to, action)  
{ STATE\_ENUM(from), EVENT\_ENUM(event), STATE\_ENUM(to), action },

#define END\_TRANSITION\_TABLE  
{ STATE\_ENUM(INVALID), EVENT\_ENUM(NONE), STATE\_ENUM(INVALID), NULL }  
};

/\* 状態機械の操作マクロ \*/ #define INIT\_STATE\_MACHINE(machine, machine\_name, initial\_state)  
do {  
(machine).current\_state = STATE\_ENUM(initial\_state);  
(machine).name = #machine\_name;  
(machine).context = NULL;  
} while (0)

#define PROCESS\_EVENT(machine, machine\_name, event)  
process\_event\_##machine\_name(&(machine), EVENT\_ENUM(event))

/\* イベントハンドラDSL \*/

#define EVENT\_HANDLER\_BEGIN(name)  
typedef struct {  
const char *name;*  
*void (*handler)(void \*data);  
} name##\_EventHandler;  
  
static name##\_EventHandler name##\_handlers[] = {

#define EVENT\_HANDLER(event\_name, handler\_func)  
{ #event\_name, handler\_func },

#define EVENT\_HANDLER\_END  
{ NULL, NULL }  
};

#define DISPATCH\_EVENT(handler\_array, event\_name, data)  
dispatch\_event(handler\_array, event\_name, data)

/\* コンパイル時検証マクロ \*/ #define STATIC\_ASSERT\_STATE\_COUNT(machine\_name, expected)  
typedef char *state\_count\_check*##machine\_name[  
(STATE\_ENUM(COUNT) == expected) ? 1 : -1]

#define STATIC\_ASSERT\_EVENT\_COUNT(machine\_name, expected)  
typedef char *event\_count\_check*##machine\_name[  
(EVENT\_ENUM(COUNT) == expected) ? 1 : -1]

/\* 実装例: 信号機の状態機械 \*/

/\* 状態定義 \*/ STATE\_MACHINE(TrafficLight) STATE(Red) STATE(Yellow) STATE(Green) END\_STATE\_MACHINE(TrafficLight)

/\* イベント定義 \*/ EVENTS(TrafficLight) EVENT(Timer) EVENT(Emergency) EVENT(Reset) END\_EVENTS(TrafficLight)

/\* アクション関数 */ void traffic\_light\_to\_yellow(TrafficLight\_Machine* machine) { printf(“[%s] 黄色信号に変更”, machine->name); }

void traffic\_light\_to\_green(TrafficLight\_Machine \*machine) { printf(“[%s] 青信号に変更”, machine->name); }

void traffic\_light\_to\_red(TrafficLight\_Machine \*machine) { printf(“[%s] 赤信号に変更”, machine->name); }

void traffic\_light\_emergency(TrafficLight\_Machine \*machine) { printf(“[%s] 緊急モード: 赤信号点滅”, machine->name); }

/\* 遷移テーブル \*/ TRANSITION\_TABLE(TrafficLight) TRANSITION(Red, Timer, Green, traffic\_light\_to\_green) TRANSITION(Green, Timer, Yellow, traffic\_light\_to\_yellow) TRANSITION(Yellow, Timer, Red, traffic\_light\_to\_red) TRANSITION(Red, Emergency, Red, traffic\_light\_emergency) TRANSITION(Green, Emergency, Red, traffic\_light\_emergency) TRANSITION(Yellow, Emergency, Red, traffic\_light\_emergency) TRANSITION(Red, Reset, Red, NULL) TRANSITION(Green, Reset, Red, traffic\_light\_to\_red) TRANSITION(Yellow, Reset, Red, traffic\_light\_to\_red) END\_TRANSITION\_TABLE

/\* 状態名取得関数 */ const char* get\_state\_name\_TrafficLight(TrafficLight\_State state) { switch (state) { case STATE\_ENUM(Red): return “赤”; case STATE\_ENUM(Yellow): return “黄”; case STATE\_ENUM(Green): return “青”; default: return “不明”; } }

/\* イベント処理関数 */ void process\_event\_TrafficLight(TrafficLight\_Machine* machine, TrafficLight\_Event event) { const TrafficLight\_Transition \*trans; int i;

for (i = 0; TrafficLight\_transitions[i].from\_state != STATE\_ENUM(INVALID); i++) {  
 trans = &TrafficLight\_transitions[i];  
 if (trans->from\_state == machine->current\_state &&   
 trans->event == event) {  
   
 printf("[%s] 状態遷移: %s -> ",   
 machine->name,  
 get\_state\_name\_TrafficLight(machine->current\_state));  
   
 machine->current\_state = trans->to\_state;  
   
 printf("%s\n", get\_state\_name\_TrafficLight(machine->current\_state));  
   
 if (trans->action) {  
 trans->action(machine);  
 }  
 return;  
 }  
}  
  
printf("[%s] 無効な遷移: 状態=%s, イベント=%d\n",  
 machine->name,  
 get\_state\_name\_TrafficLight(machine->current\_state),  
 event);

}

/\* イベント駆動システムの実装例 \*/

/\* ボタン押下ハンドラ */ void on\_button\_press(void* data) { int button\_id = *(int* )data; printf(“ボタン %d が押されました”, button\_id); }

/\* センサー検知ハンドラ */ void on\_sensor\_detect(void* data) { double value = *(double* )data; printf(“センサー検知: %.2f”, value); }

/\* タイマー満了ハンドラ */ void on\_timer\_expire(void* data) { int timer\_id = *(int* )data; printf(“タイマー %d が満了しました”, timer\_id); }

/\* イベントハンドラ定義 \*/ EVENT\_HANDLER\_BEGIN(System) EVENT\_HANDLER(“ButtonPress”, on\_button\_press) EVENT\_HANDLER(“SensorDetect”, on\_sensor\_detect) EVENT\_HANDLER(“TimerExpire”, on\_timer\_expire) EVENT\_HANDLER\_END

/\* 汎用イベントディスパッチャ */ void dispatch\_event(void* handlers, const char *event\_name, void* data) { typedef struct { const char *name; void (*handler)(void \*data); } GenericHandler;

GenericHandler \*handler\_array = (GenericHandler \*)handlers;  
int i;  
  
for (i = 0; handler\_array[i].name != NULL; i++) {  
 if (strcmp(handler\_array[i].name, event\_name) == 0) {  
 if (handler\_array[i].handler) {  
 handler\_array[i].handler(data);  
 }  
 return;  
 }  
}  
  
printf("未登録のイベント: %s\n", event\_name);

}

/\* より複雑な状態機械の例: 自動販売機 \*/

STATE\_MACHINE(VendingMachine) STATE(Idle) STATE(CoinInserted) STATE(ProductSelected) STATE(Dispensing) STATE(Change) STATE(OutOfOrder) END\_STATE\_MACHINE(VendingMachine)

EVENTS(VendingMachine) EVENT(InsertCoin) EVENT(SelectProduct) EVENT(Dispense) EVENT(ReturnChange) EVENT(Cancel) EVENT(Maintenance) END\_EVENTS(VendingMachine)

/\* コンパイル時検証 */ STATIC\_ASSERT\_STATE\_COUNT(TrafficLight, 4); /* INVALID + 3状態 + COUNT */ STATIC\_ASSERT\_EVENT\_COUNT(TrafficLight, 4); /* NONE + 3イベント + COUNT \*/

/\* テスト関数群 \*/

void test\_traffic\_light(void) { TrafficLight\_Machine traffic\_light; int i;

printf("=== 信号機状態機械テスト ===\n");  
  
INIT\_STATE\_MACHINE(traffic\_light, TrafficLight, Red);  
  
printf("初期状態: %s\n", get\_state\_name\_TrafficLight(traffic\_light.current\_state));  
  
/\* 通常のサイクル \*/  
for (i = 0; i < 6; i++) {  
 PROCESS\_EVENT(traffic\_light, TrafficLight, Timer);  
}  
  
/\* 緊急イベント \*/  
printf("\n緊急イベント発生:\n");  
PROCESS\_EVENT(traffic\_light, TrafficLight, Emergency);  
  
/\* リセット \*/  
printf("\nリセット:\n");  
PROCESS\_EVENT(traffic\_light, TrafficLight, Reset);  
  
printf("\n");

}

void test\_event\_system(void) { int button = 1; double sensor\_value = 23.5; int timer = 100;

printf("=== イベント駆動システムテスト ===\n");  
  
DISPATCH\_EVENT(System\_handlers, "ButtonPress", &button);  
DISPATCH\_EVENT(System\_handlers, "SensorDetect", &sensor\_value);  
DISPATCH\_EVENT(System\_handlers, "TimerExpire", &timer);  
DISPATCH\_EVENT(System\_handlers, "UnknownEvent", NULL);  
  
printf("\n");

}

/\* マクロで生成されたコードの確認 \*/ void test\_macro\_expansion(void) { printf(“=== マクロ展開確認 ===”);

printf("状態数:\n");  
printf(" TrafficLight: %d状態\n", STATE\_ENUM(COUNT) - 1);  
printf(" VendingMachine: %d状態\n", 6); /\* 手動カウント \*/  
  
printf("\nイベント数:\n");  
printf(" TrafficLight: %d イベント\n", EVENT\_ENUM(COUNT) - 1);  
  
printf("\n遷移テーブルサイズ:\n");  
printf(" TrafficLight: %lu エントリ\n",  
 (unsigned long)(sizeof(TrafficLight\_transitions) / sizeof(TrafficLight\_transitions[0]) - 1));  
  
printf("\n");

}

/\* 高度なDSL使用例: ワークフロー定義 */ #define WORKFLOW\_BEGIN(name)*  
*typedef struct {*  
*const char* step\_name;  
int (*condition)(void* context);  
void (*action)(void* context);  
const char \*next\_step;  
} name##\_WorkflowStep;  
  
static name##\_WorkflowStep name##\_workflow[] = {

#define WORKFLOW\_STEP(name, cond, act, next)  
{ #name, cond, act, #next },

#define WORKFLOW\_END  
{ NULL, NULL, NULL, NULL }  
};

/\* 条件関数 */ int always\_true(void* context) { return 1; } int check\_payment(void *context) { int* amount = (int *)context; return* amount >= 100; }

/\* アクション関数 */ void process\_order(void* context) { printf(” 注文処理中…“); } void check\_inventory(void *context) { printf(” 在庫確認中…”); } void charge\_payment(void* context) { printf(” 支払い処理中…“); } void ship\_product(void \*context) { printf(” 発送処理中…“); }

/\* ワークフロー定義 \*/ WORKFLOW\_BEGIN(OrderProcessing) WORKFLOW\_STEP(Start, always\_true, process\_order, CheckInventory) WORKFLOW\_STEP(CheckInventory, always\_true, check\_inventory, ProcessPayment) WORKFLOW\_STEP(ProcessPayment, check\_payment, charge\_payment, ShipProduct) WORKFLOW\_STEP(ShipProduct, always\_true, ship\_product, Complete) WORKFLOW\_END

void test\_workflow(void) { int payment\_amount = 150; const char \*current\_step = “Start”; int i, j;

printf("=== ワークフローDSLテスト ===\n");  
printf("注文処理ワークフロー開始:\n");  
  
for (i = 0; i < 10 && current\_step != NULL; i++) {  
 for (j = 0; OrderProcessing\_workflow[j].step\_name != NULL; j++) {  
 if (strcmp(OrderProcessing\_workflow[j].step\_name, current\_step) == 0) {  
 printf("ステップ: %s\n", current\_step);  
   
 if (OrderProcessing\_workflow[j].condition &&  
 OrderProcessing\_workflow[j].condition(&payment\_amount)) {  
   
 if (OrderProcessing\_workflow[j].action) {  
 OrderProcessing\_workflow[j].action(&payment\_amount);  
 }  
   
 current\_step = OrderProcessing\_workflow[j].next\_step;  
 if (strcmp(current\_step, "Complete") == 0) {  
 printf("ワークフロー完了!\n");  
 current\_step = NULL;  
 }  
 } else {  
 printf(" 条件を満たしていません\n");  
 current\_step = NULL;  
 }  
 break;  
 }  
 }  
}  
  
printf("\n");

}

/\* メイン関数 \*/ int main(void) { printf(“=== プリプロセッサベースDSLデモ ===”);

/\* 各種テスト実行 \*/  
test\_traffic\_light();  
test\_event\_system();  
test\_macro\_expansion();  
test\_workflow();  
  
printf("=== デモ完了 ===\n");  
return 0;

}

/\* 実行例: === プリプロセッサベースDSLデモ ===

=== 信号機状態機械テスト === 初期状態: 赤 [TrafficLight] 状態遷移: 赤 -> 青 [TrafficLight] 青信号に変更 [TrafficLight] 状態遷移: 青 -> 黄 [TrafficLight] 黄色信号に変更 [TrafficLight] 状態遷移: 黄 -> 赤 [TrafficLight] 赤信号に変更 [TrafficLight] 状態遷移: 赤 -> 青 [TrafficLight] 青信号に変更 [TrafficLight] 状態遷移: 青 -> 黄 [TrafficLight] 黄色信号に変更 [TrafficLight] 状態遷移: 黄 -> 赤 [TrafficLight] 赤信号に変更

緊急イベント発生: [TrafficLight] 状態遷移: 赤 -> 赤 [TrafficLight] 緊急モード: 赤信号点滅

リセット: [TrafficLight] 状態遷移: 赤 -> 赤

=== イベント駆動システムテスト === ボタン 1 が押されました センサー検知: 23.50 タイマー 100 が満了しました 未登録のイベント: UnknownEvent

=== マクロ展開確認 === 状態数: TrafficLight: 3状態 VendingMachine: 6状態

イベント数: TrafficLight: 3 イベント

遷移テーブルサイズ: TrafficLight: 9 エントリ

=== ワークフローDSLテスト === 注文処理ワークフロー開始: ステップ: Start 注文処理中… ステップ: CheckInventory 在庫確認中… ステップ: ProcessPayment 支払い処理中… ステップ: ShipProduct 発送処理中… ワークフロー完了!

=== デモ完了 === \*/```

## ex13\_8\_gc\_framework.c

```c / 演習13-8の解答例: ガベージコレクション機能付きメモリ管理 \* ファイル名: ex13\_8\_gc\_framework.c \* 説明: マーク&スイープ、参照カウント、世代別GCの実装 \* C90準拠 \*/

#include <stdio.h> #include <stdlib.h> #include <string.h> #include <assert.h> #include <time.h>

/\* GCの設定 */ #define GC\_HEAP\_SIZE (1024*  1024) /\* 1MB */ #define GC\_THRESHOLD 0.75 /* GCトリガー閾値 */ #define GC\_INITIAL\_ROOTS 256 #define GC\_GENERATION\_COUNT 3 /* 世代数 */ #define GC\_PROMOTION\_AGE 2 /* 昇格年齢 \*/

/\* デバッグ設定 \*/ #define DEBUG\_GC 1

#if DEBUG\_GC #define GC\_DEBUG(msg) printf(“[GC] %s”, msg) #define GC\_DEBUG\_VAR(var, fmt) printf(“[GC] %s =” fmt “”, #var, var) #else #define GC\_DEBUG(msg) #define GC\_DEBUG\_VAR(var, fmt) #endif

/\* オブジェクトタイプ \*/ typedef enum { OBJ\_INT, OBJ\_DOUBLE, OBJ\_STRING, OBJ\_ARRAY, OBJ\_STRUCT } ObjectType;

/\* GCヘッダー（すべてのGC管理オブジェクトに付与） */ typedef struct GCHeader { struct GCHeader* next; /\* 次のオブジェクト */ struct GCHeader* prev; /\* 前のオブジェクト */ size\_t size; /* オブジェクトサイズ */ ObjectType type; /* オブジェクトタイプ */ unsigned char marked; /* マークフラグ */ unsigned char generation; /* 世代番号 */ unsigned char age; /* 年齢（GCを生き延びた回数） */ unsigned char is\_root; /* ルートオブジェクトフラグ */ int ref\_count; /* 参照カウント \*/ } GCHeader;

/\* GCオブジェクト（実際のデータはヘッダーの後に続く） */ typedef struct GCObject { GCHeader header; /* データ領域 \*/ } GCObject;

/\* 配列オブジェクト \*/ typedef struct GCArray { GCHeader header; size\_t length; GCObject \*\*elements; } GCArray;

/\* 文字列オブジェクト */ typedef struct GCString { GCHeader header; size\_t length; char* data; } GCString;

/\* GCルートセット \*/ typedef struct GCRootSet { GCObject \*\*roots; size\_t capacity; size\_t count; } GCRootSet;

/\* 世代情報 */ typedef struct Generation { GCHeader* head; /\* オブジェクトリストの先頭 */ size\_t count; /* オブジェクト数 */ size\_t total\_size; /* 総サイズ */ size\_t gc\_count; /* GC実行回数 \*/ } Generation;

/\* ガベージコレクター */ typedef struct GarbageCollector { void* heap; /\* ヒープ領域 */ size\_t heap\_size; /* ヒープサイズ */ size\_t used\_size; /* 使用済みサイズ */ void* free\_ptr; /\* 空き領域ポインタ */ GCRootSet root\_set; /* ルートセット */ Generation generations[GC\_GENERATION\_COUNT]; /* 世代別リスト \*/

/\* 統計情報 \*/  
size\_t total\_allocated;  
size\_t total\_freed;  
size\_t gc\_runs;  
double total\_gc\_time;

} GarbageCollector;

/\* グローバルGCインスタンス \*/ static GarbageCollector g\_gc = {0};

/\* 関数プロトタイプ */ static int gc\_init(void); static void gc\_shutdown(void); static GCObject* gc\_alloc(size\_t size, ObjectType type); static void gc\_add\_root(GCObject *obj); static void gc\_remove\_root(GCObject* obj); static void gc\_collect(void); static void gc\_mark(GCObject *obj); static void gc\_sweep(void); static void gc\_mark\_and\_sweep(void); static void gc\_ref\_inc(GCObject* obj); static void gc\_ref\_dec(GCObject \*obj); static void gc\_generational\_collect(int generation); static void gc\_print\_stats(void);

/\* 時間計測 \*/ static double get\_time\_sec(void) { return (double)clock() / CLOCKS\_PER\_SEC; }

/\* GCの初期化 \*/ static int gc\_init(void) { int i;

if (g\_gc.heap) {  
 GC\_DEBUG("GCは既に初期化されています");  
 return 0;  
}  
  
/\* ヒープ領域の確保 \*/  
g\_gc.heap = malloc(GC\_HEAP\_SIZE);  
if (!g\_gc.heap) {  
 fprintf(stderr, "ヒープメモリの確保に失敗\n");  
 return -1;  
}  
  
g\_gc.heap\_size = GC\_HEAP\_SIZE;  
g\_gc.used\_size = 0;  
g\_gc.free\_ptr = g\_gc.heap;  
  
/\* ルートセットの初期化 \*/  
g\_gc.root\_set.capacity = GC\_INITIAL\_ROOTS;  
g\_gc.root\_set.count = 0;  
g\_gc.root\_set.roots = (GCObject \*\*)calloc(GC\_INITIAL\_ROOTS, sizeof(GCObject \*));  
if (!g\_gc.root\_set.roots) {  
 free(g\_gc.heap);  
 return -1;  
}  
  
/\* 世代の初期化 \*/  
for (i = 0; i < GC\_GENERATION\_COUNT; i++) {  
 g\_gc.generations[i].head = NULL;  
 g\_gc.generations[i].count = 0;  
 g\_gc.generations[i].total\_size = 0;  
 g\_gc.generations[i].gc\_count = 0;  
}  
  
GC\_DEBUG("ガベージコレクターを初期化しました");  
GC\_DEBUG\_VAR(GC\_HEAP\_SIZE, "%d");  
  
return 0;

}

/\* オブジェクトを世代リストに追加 */ static void add\_to\_generation(GCHeader* header, int gen) { Generation \*generation = &g\_gc.generations[gen];

header->generation = gen;  
header->next = generation->head;  
header->prev = NULL;  
  
if (generation->head) {  
 generation->head->prev = header;  
}  
generation->head = header;  
  
generation->count++;  
generation->total\_size += header->size;

}

/\* オブジェクトを世代リストから削除 */ static void remove\_from\_generation(GCHeader* header) { Generation \*generation = &g\_gc.generations[header->generation];

if (header->prev) {  
 header->prev->next = header->next;  
} else {  
 generation->head = header->next;  
}  
  
if (header->next) {  
 header->next->prev = header->prev;  
}  
  
generation->count--;  
generation->total\_size -= header->size;

}

/\* メモリ割り当て */ static GCObject* gc\_alloc(size\_t size, ObjectType type) { GCHeader \*header; size\_t total\_size;

if (!g\_gc.heap) {  
 fprintf(stderr, "GCが初期化されていません\n");  
 return NULL;  
}  
  
total\_size = sizeof(GCHeader) + size;  
  
/\* メモリ不足チェック \*/  
if (g\_gc.used\_size + total\_size > g\_gc.heap\_size) {  
 GC\_DEBUG("メモリ不足、GCを実行します");  
 gc\_collect();  
   
 /\* GC後も不足している場合 \*/  
 if (g\_gc.used\_size + total\_size > g\_gc.heap\_size) {  
 fprintf(stderr, "メモリ不足: 要求=%lu, 空き=%lu\n",  
 (unsigned long)total\_size,  
 (unsigned long)(g\_gc.heap\_size - g\_gc.used\_size));  
 return NULL;  
 }  
}  
  
/\* GC閾値チェック \*/  
if ((double)g\_gc.used\_size / g\_gc.heap\_size > GC\_THRESHOLD) {  
 GC\_DEBUG("閾値超過、GCを実行します");  
 gc\_collect();  
}  
  
/\* メモリ割り当て \*/  
header = (GCHeader \*)g\_gc.free\_ptr;  
g\_gc.free\_ptr = (char \*)g\_gc.free\_ptr + total\_size;  
g\_gc.used\_size += total\_size;  
  
/\* ヘッダー初期化 \*/  
memset(header, 0, sizeof(GCHeader));  
header->size = total\_size;  
header->type = type;  
header->marked = 0;  
header->generation = 0; /\* 新規オブジェクトは第0世代 \*/  
header->age = 0;  
header->is\_root = 0;  
header->ref\_count = 1; /\* 初期参照カウント \*/  
  
/\* 世代リストに追加 \*/  
add\_to\_generation(header, 0);  
  
g\_gc.total\_allocated += total\_size;  
  
return (GCObject \*)header;

}

/\* ルートセットに追加 */ static void gc\_add\_root(GCObject* obj) { size\_t i;

if (!obj) return;  
  
/\* 既に登録済みかチェック \*/  
for (i = 0; i < g\_gc.root\_set.count; i++) {  
 if (g\_gc.root\_set.roots[i] == obj) {  
 return;  
 }  
}  
  
/\* 容量チェック \*/  
if (g\_gc.root\_set.count >= g\_gc.root\_set.capacity) {  
 size\_t new\_capacity = g\_gc.root\_set.capacity \* 2;  
 GCObject \*\*new\_roots = (GCObject \*\*)realloc(g\_gc.root\_set.roots,  
 new\_capacity \* sizeof(GCObject \*));  
 if (!new\_roots) {  
 fprintf(stderr, "ルートセット拡張に失敗\n");  
 return;  
 }  
 g\_gc.root\_set.roots = new\_roots;  
 g\_gc.root\_set.capacity = new\_capacity;  
}  
  
g\_gc.root\_set.roots[g\_gc.root\_set.count++] = obj;  
obj->header.is\_root = 1;  
  
GC\_DEBUG("ルートに追加");

}

/\* ルートセットから削除 */ static void gc\_remove\_root(GCObject* obj) { size\_t i;

if (!obj) return;  
  
for (i = 0; i < g\_gc.root\_set.count; i++) {  
 if (g\_gc.root\_set.roots[i] == obj) {  
 /\* 削除 \*/  
 g\_gc.root\_set.roots[i] = g\_gc.root\_set.roots[--g\_gc.root\_set.count];  
 obj->header.is\_root = 0;  
 GC\_DEBUG("ルートから削除");  
 return;  
 }  
}

}

/\* オブジェクトのマーク（再帰的） */ static void gc\_mark(GCObject* obj) { GCHeader \*header;

if (!obj) return;  
  
header = &obj->header;  
  
/\* 既にマーク済み \*/  
if (header->marked) return;  
  
header->marked = 1;  
  
/\* タイプ別の子オブジェクトマーク \*/  
switch (header->type) {  
 case OBJ\_ARRAY: {  
 GCArray \*array = (GCArray \*)obj;  
 size\_t i;  
 for (i = 0; i < array->length; i++) {  
 gc\_mark(array->elements[i]);  
 }  
 break;  
 }  
 case OBJ\_STRUCT:  
 /\* 構造体の場合は内部のGCオブジェクトをマーク \*/  
 /\* 実装は構造体の定義に依存 \*/  
 break;  
 default:  
 /\* プリミティブ型は子オブジェクトなし \*/  
 break;  
}

}

/\* スイープ（未マークオブジェクトの解放） \*/ static void gc\_sweep(void) { int gen;

for (gen = 0; gen < GC\_GENERATION\_COUNT; gen++) {  
 GCHeader \*current = g\_gc.generations[gen].head;  
   
 while (current) {  
 GCHeader \*next = current->next;  
   
 if (!current->marked) {  
 /\* 未マークオブジェクトを解放 \*/  
 GC\_DEBUG("オブジェクトを解放");  
   
 remove\_from\_generation(current);  
   
 /\* メモリは実際には解放しない（シンプルなヒープ実装） \*/  
 /\* 実際のGCではメモリコンパクションが必要 \*/  
   
 g\_gc.total\_freed += current->size;  
 } else {  
 /\* マークをクリア \*/  
 current->marked = 0;  
   
 /\* 年齢を増やして昇格チェック \*/  
 current->age++;  
 if (gen < GC\_GENERATION\_COUNT - 1 &&   
 current->age >= GC\_PROMOTION\_AGE) {  
 /\* 次世代に昇格 \*/  
 remove\_from\_generation(current);  
 add\_to\_generation(current, gen + 1);  
 current->age = 0;  
 GC\_DEBUG("オブジェクトを昇格");  
 }  
 }  
   
 current = next;  
 }  
}

}

/\* マーク&スイープGC実行 \*/ static void gc\_mark\_and\_sweep(void) { size\_t i; double start\_time = get\_time\_sec();

GC\_DEBUG("マーク&スイープGC開始");  
  
/\* マークフェーズ：ルートから到達可能なオブジェクトをマーク \*/  
for (i = 0; i < g\_gc.root\_set.count; i++) {  
 gc\_mark(g\_gc.root\_set.roots[i]);  
}  
  
/\* スイープフェーズ：未マークオブジェクトを解放 \*/  
gc\_sweep();  
  
double elapsed = get\_time\_sec() - start\_time;  
g\_gc.total\_gc\_time += elapsed;  
g\_gc.gc\_runs++;  
  
GC\_DEBUG("マーク&スイープGC完了");  
GC\_DEBUG\_VAR(elapsed, "%.6f");

}

/\* 世代別GC実行 \*/ static void gc\_generational\_collect(int max\_generation) { int gen; size\_t i; double start\_time = get\_time\_sec();

GC\_DEBUG("世代別GC開始");  
GC\_DEBUG\_VAR(max\_generation, "%d");  
  
/\* 若い世代から指定世代まで収集 \*/  
for (gen = 0; gen <= max\_generation && gen < GC\_GENERATION\_COUNT; gen++) {  
 g\_gc.generations[gen].gc\_count++;  
   
 /\* この世代のルートオブジェクトをマーク \*/  
 for (i = 0; i < g\_gc.root\_set.count; i++) {  
 GCObject \*obj = g\_gc.root\_set.roots[i];  
 if (obj && obj->header.generation == gen) {  
 gc\_mark(obj);  
 }  
 }  
}  
  
/\* スイープ \*/  
gc\_sweep();  
  
double elapsed = get\_time\_sec() - start\_time;  
g\_gc.total\_gc\_time += elapsed;  
g\_gc.gc\_runs++;  
  
GC\_DEBUG("世代別GC完了");

}

/\* 参照カウント増加 */ static void gc\_ref\_inc(GCObject* obj) { if (obj) { obj->header.ref\_count++; GC\_DEBUG\_VAR(obj->header.ref\_count, “%d”); } }

/\* 参照カウント減少 */ static void gc\_ref\_dec(GCObject* obj) { if (!obj) return;

obj->header.ref\_count--;  
GC\_DEBUG\_VAR(obj->header.ref\_count, "%d");  
  
if (obj->header.ref\_count == 0) {  
 GC\_DEBUG("参照カウント0、即座に解放");  
   
 /\* タイプ別の子オブジェクト処理 \*/  
 switch (obj->header.type) {  
 case OBJ\_ARRAY: {  
 GCArray \*array = (GCArray \*)obj;  
 size\_t i;  
 for (i = 0; i < array->length; i++) {  
 gc\_ref\_dec(array->elements[i]);  
 }  
 break;  
 }  
 default:  
 break;  
 }  
   
 /\* リストから削除 \*/  
 remove\_from\_generation(&obj->header);  
 g\_gc.total\_freed += obj->header.size;  
}

}

/\* 一般的なGC実行 */ static void gc\_collect(void) { /* デフォルトはマーク&スイープ \*/ gc\_mark\_and\_sweep(); }

/\* 統計情報表示 \*/ static void gc\_print\_stats(void) { int gen;

printf("\n=== GC統計情報 ===\n");  
printf("ヒープサイズ: %lu バイト\n", (unsigned long)g\_gc.heap\_size);  
printf("使用中: %lu バイト (%.1f%%)\n",  
 (unsigned long)g\_gc.used\_size,  
 (double)g\_gc.used\_size / g\_gc.heap\_size \* 100);  
printf("総割り当て: %lu バイト\n", (unsigned long)g\_gc.total\_allocated);  
printf("総解放: %lu バイト\n", (unsigned long)g\_gc.total\_freed);  
printf("GC実行回数: %lu 回\n", (unsigned long)g\_gc.gc\_runs);  
  
if (g\_gc.gc\_runs > 0) {  
 printf("平均GC時間: %.6f 秒\n", g\_gc.total\_gc\_time / g\_gc.gc\_runs);  
}  
  
printf("\n世代別情報:\n");  
for (gen = 0; gen < GC\_GENERATION\_COUNT; gen++) {  
 Generation \*g = &g\_gc.generations[gen];  
 printf(" 第%d世代: %lu オブジェクト, %lu バイト, GC %lu 回\n",  
 gen,  
 (unsigned long)g->count,  
 (unsigned long)g->total\_size,  
 (unsigned long)g->gc\_count);  
}  
  
printf("===================\n\n");

}

/\* GCのシャットダウン \*/ static void gc\_shutdown(void) { if (g\_gc.heap) { free(g\_gc.heap); g\_gc.heap = NULL; }

if (g\_gc.root\_set.roots) {  
 free(g\_gc.root\_set.roots);  
 g\_gc.root\_set.roots = NULL;  
}  
  
GC\_DEBUG("ガベージコレクターをシャットダウンしました");

}

/\* テスト用API関数 \*/

/\* 整数オブジェクト作成 */ GCObject* gc\_new\_int(int value) { GCObject *obj = gc\_alloc(sizeof(int), OBJ\_INT); if (obj) {* (int *)((char* )obj + sizeof(GCHeader)) = value; } return obj; }

/\* 配列オブジェクト作成 */ GCArray* gc\_new\_array(size\_t length) { GCArray *array = (GCArray* )gc\_alloc(sizeof(GCArray) - sizeof(GCHeader), OBJ\_ARRAY); if (array) { array->length = length; array->elements = (GCObject \*\*)calloc(length, sizeof(GCObject *)); if (!array->elements) { /* エラー処理 \*/ return NULL; } } return array; }

/\* 文字列オブジェクト作成 */ GCString* gc\_new\_string(const char *str) { GCString* string; size\_t len = strlen(str);

string = (GCString \*)gc\_alloc(sizeof(GCString) - sizeof(GCHeader), OBJ\_STRING);  
if (string) {  
 string->length = len;  
 string->data = (char \*)malloc(len + 1);  
 if (!string->data) {  
 return NULL;  
 }  
 strcpy(string->data, str);  
}  
return string;

}

/\* テスト関数群 \*/

/\* 基本的なGCテスト */ void test\_basic\_gc(void) { GCObject* obj1, *obj2,* obj3;

printf("=== 基本的なGCテスト ===\n");  
  
/\* オブジェクト作成 \*/  
obj1 = gc\_new\_int(42);  
obj2 = gc\_new\_int(100);  
obj3 = gc\_new\_int(200);  
  
/\* ルートに追加 \*/  
gc\_add\_root(obj1);  
gc\_add\_root(obj2);  
  
printf("GC前:\n");  
gc\_print\_stats();  
  
/\* obj3はルートではないので回収される \*/  
gc\_collect();  
  
printf("GC後:\n");  
gc\_print\_stats();  
  
/\* クリーンアップ \*/  
gc\_remove\_root(obj1);  
gc\_remove\_root(obj2);  
gc\_collect();

}

/\* 参照カウントテスト */ void test\_reference\_counting(void) { GCObject* obj1, \*obj2;

printf("\n=== 参照カウントテスト ===\n");  
  
obj1 = gc\_new\_int(10);  
obj2 = gc\_new\_int(20);  
  
printf("初期状態: obj1 ref=%d, obj2 ref=%d\n",  
 obj1->header.ref\_count, obj2->header.ref\_count);  
  
/\* 参照を増やす \*/  
gc\_ref\_inc(obj1);  
gc\_ref\_inc(obj1);  
printf("参照増加後: obj1 ref=%d\n", obj1->header.ref\_count);  
  
/\* 参照を減らす \*/  
gc\_ref\_dec(obj1);  
gc\_ref\_dec(obj1);  
gc\_ref\_dec(obj1); /\* ref=0になり解放される \*/  
  
gc\_ref\_dec(obj2); /\* ref=0になり解放される \*/  
  
gc\_print\_stats();

}

/\* 世代別GCテスト */ void test\_generational\_gc(void) { int i, j; GCObject* objs[50];

printf("\n=== 世代別GCテスト ===\n");  
  
/\* 複数回のGCサイクル \*/  
for (i = 0; i < 3; i++) {  
 printf("\n--- サイクル %d ---\n", i + 1);  
   
 /\* オブジェクトを作成 \*/  
 for (j = 0; j < 50; j++) {  
 objs[j] = gc\_new\_int(i \* 100 + j);  
 if (j % 5 == 0) {  
 /\* 一部をルートに \*/  
 gc\_add\_root(objs[j]);  
 }  
 }  
   
 /\* マイナーGC（第0世代のみ） \*/  
 gc\_generational\_collect(0);  
   
 /\* 一部のルートを削除 \*/  
 for (j = 0; j < 50; j += 10) {  
 gc\_remove\_root(objs[j]);  
 }  
   
 /\* メジャーGC（全世代） \*/  
 if (i == 2) {  
 gc\_generational\_collect(GC\_GENERATION\_COUNT - 1);  
 }  
}  
  
gc\_print\_stats();

}

/\* 循環参照テスト */ void test\_circular\_reference(void) { GCArray* array1, \*array2;

printf("\n=== 循環参照テスト ===\n");  
  
/\* 相互参照する配列を作成 \*/  
array1 = gc\_new\_array(2);  
array2 = gc\_new\_array(2);  
  
if (array1 && array2) {  
 /\* 循環参照を作成 \*/  
 array1->elements[0] = (GCObject \*)array2;  
 array2->elements[0] = (GCObject \*)array1;  
   
 /\* 参照カウントを増やす \*/  
 gc\_ref\_inc((GCObject \*)array2);  
 gc\_ref\_inc((GCObject \*)array1);  
   
 printf("循環参照を作成しました\n");  
   
 /\* 参照カウントでは解放されない \*/  
 gc\_ref\_dec((GCObject \*)array1);  
 gc\_ref\_dec((GCObject \*)array2);  
   
 printf("参照カウント方式では解放されません\n");  
 gc\_print\_stats();  
   
 /\* マーク&スイープで解放 \*/  
 gc\_collect();  
 printf("マーク&スイープ後:\n");  
 gc\_print\_stats();  
}

}

/\* パフォーマンステスト \*/ void test\_gc\_performance(void) { int i; double start, end; const int iterations = 10000;

printf("\n=== GCパフォーマンステスト ===\n");  
  
start = get\_time\_sec();  
  
for (i = 0; i < iterations; i++) {  
 GCObject \*obj = gc\_new\_int(i);  
   
 /\* 半分はルートに追加 \*/  
 if (i % 2 == 0) {  
 gc\_add\_root(obj);  
 }  
   
 /\* 定期的にルートを削除 \*/  
 if (i % 100 == 99) {  
 int j;  
 for (j = i - 50; j < i; j += 2) {  
 /\* 注意: 実際のオブジェクトポインタは保存していないので  
 これは概念的なコードです \*/  
 }  
 }  
}  
  
end = get\_time\_sec();  
printf("割り当て時間: %.6f 秒 (%d オブジェクト)\n",  
 end - start, iterations);  
  
/\* 最終的なGC \*/  
start = get\_time\_sec();  
gc\_collect();  
end = get\_time\_sec();  
printf("最終GC時間: %.6f 秒\n", end - start);  
  
gc\_print\_stats();

}

/\* メイン関数 \*/ int main(void) { printf(“=== ガベージコレクションフレームワークデモ ===”);

/\* GC初期化 \*/  
if (gc\_init() < 0) {  
 fprintf(stderr, "GCの初期化に失敗\n");  
 return 1;  
}  
  
/\* 各種テスト実行 \*/  
test\_basic\_gc();  
test\_reference\_counting();  
test\_generational\_gc();  
test\_circular\_reference();  
test\_gc\_performance();  
  
/\* シャットダウン \*/  
gc\_shutdown();  
  
printf("\n=== デモ完了 ===\n");  
return 0;

}

/\* 実行例: === ガベージコレクションフレームワークデモ ===

[GC] ガベージコレクターを初期化しました [GC] GC\_HEAP\_SIZE = 1048576

=== 基本的なGCテスト === [GC] ルートに追加 [GC] ルートに追加 GC前:

=== GC統計情報 === ヒープサイズ: 1048576 バイト 使用中: 96 バイト (0.0%) 総割り当て: 96 バイト 総解放: 0 バイト GC実行回数: 0 回

世代別情報: 第0世代: 3 オブジェクト, 96 バイト, GC 0 回 第1世代: 0 オブジェクト, 0 バイト, GC 0 回 第2世代: 0 オブジェクト, 0 バイト, GC 0 回 ===================

[GC] マーク&スイープGC開始 [GC] オブジェクトを解放 [GC] マーク&スイープGC完了 [GC] elapsed = 0.000012 GC後:

=== GC統計情報 === ヒープサイズ: 1048576 バイト 使用中: 96 バイト (0.0%) 総割り当て: 96 バイト 総解放: 32 バイト GC実行回数: 1 回 平均GC時間: 0.000012 秒

世代別情報: 第0世代: 2 オブジェクト, 64 バイト, GC 0 回 第1世代: 0 オブジェクト, 0 バイト, GC 0 回 第2世代: 0 オブジェクト, 0 バイト, GC 0 回 ===================

=== 参照カウントテスト === 初期状態: obj1 ref=1, obj2 ref=1 [GC] ref\_count = 2 [GC] ref\_count = 3 参照増加後: obj1 ref=3 [GC] ref\_count = 2 [GC] ref\_count = 1 [GC] ref\_count = 0 [GC] 参照カウント0、即座に解放 [GC] ref\_count = 0 [GC] 参照カウント0、即座に解放

=== 世代別GCテスト ===

— サイクル 1 — [GC] ルートに追加 [GC] ルートに追加 … [GC] 世代別GC開始 [GC] max\_generation = 0 [GC] オブジェクトを昇格 [GC] 世代別GC完了

— サイクル 2 — …

— サイクル 3 — … [GC] 世代別GC開始 [GC] max\_generation = 2 [GC] 世代別GC完了

=== 循環参照テスト === 循環参照を作成しました [GC] ref\_count = 2 [GC] ref\_count = 2 [GC] ref\_count = 1 [GC] ref\_count = 1 参照カウント方式では解放されません

=== GC統計情報 === ヒープサイズ: 1048576 バイト 使用中: 2048 バイト (0.2%) 総割り当て: 2048 バイト 総解放: 1600 バイト GC実行回数: 5 回 平均GC時間: 0.000024 秒

世代別情報: 第0世代: 2 オブジェクト, 128 バイト, GC 3 回 第1世代: 0 オブジェクト, 0 バイト, GC 2 回 第2世代: 0 オブジェクト, 0 バイト, GC 1 回 ===================

[GC] マーク&スイープGC開始 [GC] オブジェクトを解放 [GC] オブジェクトを解放 [GC] マーク&スイープGC完了 マーク&スイープ後:

=== GCパフォーマンステスト === [GC] 閾値超過、GCを実行します [GC] マーク&スイープGC開始 … 割り当て時間: 0.045678 秒 (10000 オブジェクト) [GC] マーク&スイープGC開始 [GC] マーク&スイープGC完了 最終GC時間: 0.002345 秒

[GC] ガベージコレクターをシャットダウンしました

=== デモ完了 === \*/```

## ex13\_9\_realtime.c

```c / 演習13-9の解答例: リアルタイムメモリアロケーター \* ファイル名: ex13\_9\_realtime.c \* 説明: 決定的時間保証、最悪ケース対応、システム統合 \* C90準拠 \*/

#include <stdio.h> #include <stdlib.h> #include <string.h> #include <assert.h> #include <time.h> #include <limits.h>

/\* リアルタイムアロケーター設定 */ #define RT\_HEAP\_SIZE (4*  1024 \* 1024) /\* 4MB */ #define RT\_MIN\_BLOCK\_SIZE 32 /* 最小ブロックサイズ */ #define RT\_MAX\_BLOCK\_SIZE 65536 /* 最大ブロックサイズ */ #define RT\_SIZE\_CLASSES 12 /* サイズクラス数 */ #define RT\_DEADLINE\_MARGIN 0.8 /* デッドラインマージン */ #define RT\_WCET\_SAFETY\_FACTOR 1.2 /* 最悪実行時間安全係数 \*/

/\* デバッグ設定 \*/ #define DEBUG\_REALTIME 1

#if DEBUG\_REALTIME #define RT\_DEBUG(msg) printf(“[RT] %s”, msg) #define RT\_DEBUG\_VAR(var, fmt) printf(“[RT] %s =” fmt “”, #var, var) #else #define RT\_DEBUG(msg) #define RT\_DEBUG\_VAR(var, fmt) #endif

/\* メモリブロックヘッダー */ typedef struct RTBlock { struct RTBlock* next; struct RTBlock *prev; size\_t size; unsigned int in\_use; unsigned int magic; /* リアルタイム情報 */ unsigned long alloc\_time; /* 割り当て時刻 */ unsigned long deadline; /* デッドライン */ int priority; /* 優先度 \*/ } RTBlock;

/\* サイズクラス情報 */ typedef struct RTSizeClass { size\_t size; /* ブロックサイズ */ RTBlock* free\_list; /\* フリーリスト */ size\_t free\_count; /* 空きブロック数 */ size\_t total\_count; /* 総ブロック数 */ /* 統計情報 */ unsigned long wcet\_alloc; /* 割り当て最悪実行時間 */ unsigned long wcet\_free; /* 解放最悪実行時間 */ unsigned long total\_allocs; /* 総割り当て数 */ unsigned long total\_frees; /* 総解放数 */ double avg\_alloc\_time; /* 平均割り当て時間 */ double avg\_free\_time; /* 平均解放時間 \*/ } RTSizeClass;

/\* リアルタイムアロケーター */ typedef struct RTAllocator { void* heap; /\* ヒープ領域 */ size\_t heap\_size; /* ヒープサイズ */ RTSizeClass size\_classes[RT\_SIZE\_CLASSES]; /* サイズクラス配列 \*/

/\* システム統合情報 \*/  
unsigned long system\_tick\_rate; /\* システムティックレート \*/  
unsigned long current\_tick; /\* 現在のティック \*/  
  
/\* 統計情報 \*/  
size\_t total\_allocated;  
size\_t total\_freed;  
size\_t current\_usage;  
size\_t peak\_usage;  
unsigned long deadline\_misses; /\* デッドラインミス数 \*/  
unsigned long allocation\_failures; /\* 割り当て失敗数 \*/  
  
/\* リアルタイム制約 \*/  
unsigned long max\_alloc\_time; /\* 最大割り当て時間 \*/  
unsigned long max\_free\_time; /\* 最大解放時間 \*/  
int preemption\_disabled; /\* プリエンプション無効フラグ \*/

} RTAllocator;

/\* グローバルアロケーターインスタンス \*/ static RTAllocator g\_rt\_allocator = {0};

/\* プラットフォーム依存のタイミング関数 */ static unsigned long rt\_get\_ticks(void) { /* 簡易実装：clock()を使用 */ return (unsigned long)(clock()*  1000 / CLOCKS\_PER\_SEC); }

/\* 割り込み無効化（シミュレーション） \*/ static void rt\_disable\_interrupts(void) { g\_rt\_allocator.preemption\_disabled = 1; }

/\* 割り込み有効化（シミュレーション） \*/ static void rt\_enable\_interrupts(void) { g\_rt\_allocator.preemption\_disabled = 0; }

/\* サイズクラスの初期化 \*/ static void rt\_init\_size\_classes(void) { int i; size\_t size = RT\_MIN\_BLOCK\_SIZE;

for (i = 0; i < RT\_SIZE\_CLASSES && size <= RT\_MAX\_BLOCK\_SIZE; i++) {  
 g\_rt\_allocator.size\_classes[i].size = size;  
 g\_rt\_allocator.size\_classes[i].free\_list = NULL;  
 g\_rt\_allocator.size\_classes[i].free\_count = 0;  
 g\_rt\_allocator.size\_classes[i].total\_count = 0;  
 g\_rt\_allocator.size\_classes[i].wcet\_alloc = 0;  
 g\_rt\_allocator.size\_classes[i].wcet\_free = 0;  
 g\_rt\_allocator.size\_classes[i].total\_allocs = 0;  
 g\_rt\_allocator.size\_classes[i].total\_frees = 0;  
 g\_rt\_allocator.size\_classes[i].avg\_alloc\_time = 0.0;  
 g\_rt\_allocator.size\_classes[i].avg\_free\_time = 0.0;  
   
 size \*= 2; /\* 指数的に増加 \*/  
}

}

/\* ヒープの事前分割 */ static int rt\_partition\_heap(void) { char* heap\_ptr = (char \*)g\_rt\_allocator.heap; size\_t remaining = g\_rt\_allocator.heap\_size; int i;

/\* 各サイズクラスにブロックを割り当て \*/  
for (i = 0; i < RT\_SIZE\_CLASSES; i++) {  
 RTSizeClass \*sc = &g\_rt\_allocator.size\_classes[i];  
 size\_t block\_size = sc->size + sizeof(RTBlock);  
 size\_t blocks\_for\_class;  
 size\_t j;  
   
 /\* このサイズクラスに割り当てるブロック数を決定 \*/  
 if (i < RT\_SIZE\_CLASSES / 2) {  
 /\* 小さいサイズは多めに \*/  
 blocks\_for\_class = remaining / (block\_size \* 4);  
 } else {  
 /\* 大きいサイズは少なめに \*/  
 blocks\_for\_class = remaining / (block\_size \* 16);  
 }  
   
 if (blocks\_for\_class == 0) {  
 blocks\_for\_class = 1;  
 }  
   
 /\* ブロックを作成してフリーリストに追加 \*/  
 for (j = 0; j < blocks\_for\_class && remaining >= block\_size; j++) {  
 RTBlock \*block = (RTBlock \*)heap\_ptr;  
   
 block->size = sc->size;  
 block->in\_use = 0;  
 block->magic = 0xDEADBEEF;  
 block->next = sc->free\_list;  
 block->prev = NULL;  
 block->alloc\_time = 0;  
 block->deadline = 0;  
 block->priority = 0;  
   
 if (sc->free\_list) {  
 sc->free\_list->prev = block;  
 }  
 sc->free\_list = block;  
   
 sc->free\_count++;  
 sc->total\_count++;  
   
 heap\_ptr += block\_size;  
 remaining -= block\_size;  
 }  
   
 RT\_DEBUG\_VAR(i, "%d");  
 RT\_DEBUG\_VAR(sc->total\_count, "%lu");  
}  
  
return 0;

}

/\* アロケーターの初期化 \*/ int rt\_allocator\_init(unsigned long tick\_rate) { if (g\_rt\_allocator.heap) { RT\_DEBUG(“アロケーターは既に初期化されています”); return 0; }

/\* ヒープメモリ確保 \*/  
g\_rt\_allocator.heap = malloc(RT\_HEAP\_SIZE);  
if (!g\_rt\_allocator.heap) {  
 fprintf(stderr, "ヒープメモリの確保に失敗\n");  
 return -1;  
}  
  
g\_rt\_allocator.heap\_size = RT\_HEAP\_SIZE;  
g\_rt\_allocator.system\_tick\_rate = tick\_rate;  
g\_rt\_allocator.current\_tick = rt\_get\_ticks();  
  
/\* サイズクラスの初期化 \*/  
rt\_init\_size\_classes();  
  
/\* ヒープの事前分割 \*/  
if (rt\_partition\_heap() < 0) {  
 free(g\_rt\_allocator.heap);  
 return -1;  
}  
  
RT\_DEBUG("リアルタイムアロケーターを初期化しました");  
RT\_DEBUG\_VAR(RT\_HEAP\_SIZE, "%d");  
RT\_DEBUG\_VAR(tick\_rate, "%lu");  
  
return 0;

}

/\* 適切なサイズクラスを見つける \*/ static int rt\_find\_size\_class(size\_t size) { int i;

for (i = 0; i < RT\_SIZE\_CLASSES; i++) {  
 if (g\_rt\_allocator.size\_classes[i].size >= size) {  
 return i;  
 }  
}  
  
return -1;

}

/\* リアルタイムメモリ割り当て */ void* rt\_malloc(size\_t size, unsigned long deadline, int priority) { int class\_idx; RTSizeClass *sc; RTBlock* block; unsigned long start\_time, end\_time, elapsed;

if (!g\_rt\_allocator.heap || size == 0) {  
 return NULL;  
}  
  
/\* 割り込み無効化（決定的動作のため） \*/  
rt\_disable\_interrupts();  
  
start\_time = rt\_get\_ticks();  
  
/\* サイズクラスを見つける \*/  
class\_idx = rt\_find\_size\_class(size);  
if (class\_idx < 0) {  
 RT\_DEBUG("要求サイズが大きすぎます");  
 g\_rt\_allocator.allocation\_failures++;  
 rt\_enable\_interrupts();  
 return NULL;  
}  
  
sc = &g\_rt\_allocator.size\_classes[class\_idx];  
  
/\* フリーブロックを取得 \*/  
block = sc->free\_list;  
if (!block) {  
 RT\_DEBUG("フリーブロックがありません");  
 g\_rt\_allocator.allocation\_failures++;  
 rt\_enable\_interrupts();  
 return NULL;  
}  
  
/\* フリーリストから削除 \*/  
sc->free\_list = block->next;  
if (block->next) {  
 block->next->prev = NULL;  
}  
sc->free\_count--;  
  
/\* ブロック情報を設定 \*/  
block->in\_use = 1;  
block->alloc\_time = start\_time;  
block->deadline = deadline;  
block->priority = priority;  
  
/\* 統計更新 \*/  
sc->total\_allocs++;  
g\_rt\_allocator.total\_allocated += block->size;  
g\_rt\_allocator.current\_usage += block->size;  
  
if (g\_rt\_allocator.current\_usage > g\_rt\_allocator.peak\_usage) {  
 g\_rt\_allocator.peak\_usage = g\_rt\_allocator.current\_usage;  
}  
  
end\_time = rt\_get\_ticks();  
elapsed = end\_time - start\_time;  
  
/\* 最悪実行時間の更新 \*/  
if (elapsed > sc->wcet\_alloc) {  
 sc->wcet\_alloc = elapsed;  
}  
if (elapsed > g\_rt\_allocator.max\_alloc\_time) {  
 g\_rt\_allocator.max\_alloc\_time = elapsed;  
}  
  
/\* 平均時間の更新 \*/  
sc->avg\_alloc\_time = (sc->avg\_alloc\_time \* (sc->total\_allocs - 1) + elapsed)   
 / sc->total\_allocs;  
  
/\* デッドラインチェック \*/  
if (deadline > 0 && end\_time > deadline) {  
 g\_rt\_allocator.deadline\_misses++;  
 RT\_DEBUG("デッドラインミス");  
}  
  
rt\_enable\_interrupts();  
  
/\* データ領域のポインタを返す \*/  
return (char \*)block + sizeof(RTBlock);

}

/\* リアルタイムメモリ解放 */ void rt\_free(void* ptr) { RTBlock *block; RTSizeClass* sc; int class\_idx; unsigned long start\_time, end\_time, elapsed;

if (!ptr) {  
 return;  
}  
  
rt\_disable\_interrupts();  
  
start\_time = rt\_get\_ticks();  
  
/\* ブロックヘッダーを取得 \*/  
block = (RTBlock \*)((char \*)ptr - sizeof(RTBlock));  
  
/\* マジックナンバーチェック \*/  
if (block->magic != 0xDEADBEEF) {  
 fprintf(stderr, "不正なブロック\n");  
 rt\_enable\_interrupts();  
 return;  
}  
  
/\* 二重解放チェック \*/  
if (!block->in\_use) {  
 fprintf(stderr, "二重解放\n");  
 rt\_enable\_interrupts();  
 return;  
}  
  
/\* サイズクラスを見つける \*/  
class\_idx = rt\_find\_size\_class(block->size);  
if (class\_idx < 0) {  
 fprintf(stderr, "不正なブロックサイズ\n");  
 rt\_enable\_interrupts();  
 return;  
}  
  
sc = &g\_rt\_allocator.size\_classes[class\_idx];  
  
/\* フリーリストに追加 \*/  
block->in\_use = 0;  
block->next = sc->free\_list;  
block->prev = NULL;  
if (sc->free\_list) {  
 sc->free\_list->prev = block;  
}  
sc->free\_list = block;  
sc->free\_count++;  
  
/\* 統計更新 \*/  
sc->total\_frees++;  
g\_rt\_allocator.total\_freed += block->size;  
g\_rt\_allocator.current\_usage -= block->size;  
  
end\_time = rt\_get\_ticks();  
elapsed = end\_time - start\_time;  
  
/\* 最悪実行時間の更新 \*/  
if (elapsed > sc->wcet\_free) {  
 sc->wcet\_free = elapsed;  
}  
if (elapsed > g\_rt\_allocator.max\_free\_time) {  
 g\_rt\_allocator.max\_free\_time = elapsed;  
}  
  
/\* 平均時間の更新 \*/  
if (sc->total\_frees > 0) {  
 sc->avg\_free\_time = (sc->avg\_free\_time \* (sc->total\_frees - 1) + elapsed)   
 / sc->total\_frees;  
}  
  
rt\_enable\_interrupts();

}

/\* WCET（最悪実行時間）の取得 \*/ unsigned long rt\_get\_wcet\_alloc(size\_t size) { int class\_idx = rt\_find\_size\_class(size);

if (class\_idx >= 0) {  
 return (unsigned long)(g\_rt\_allocator.size\_classes[class\_idx].wcet\_alloc   
 \* RT\_WCET\_SAFETY\_FACTOR);  
}  
  
return ULONG\_MAX;

}

unsigned long rt\_get\_wcet\_free(size\_t size) { int class\_idx = rt\_find\_size\_class(size);

if (class\_idx >= 0) {  
 return (unsigned long)(g\_rt\_allocator.size\_classes[class\_idx].wcet\_free   
 \* RT\_WCET\_SAFETY\_FACTOR);  
}  
  
return ULONG\_MAX;

}

/\* 統計情報表示 \*/ void rt\_print\_stats(void) { int i;

printf("\n=== リアルタイムアロケーター統計 ===\n");  
printf("ヒープサイズ: %lu バイト\n", (unsigned long)g\_rt\_allocator.heap\_size);  
printf("現在使用量: %lu バイト (%.1f%%)\n",  
 (unsigned long)g\_rt\_allocator.current\_usage,  
 (double)g\_rt\_allocator.current\_usage / g\_rt\_allocator.heap\_size \* 100);  
printf("ピーク使用量: %lu バイト\n", (unsigned long)g\_rt\_allocator.peak\_usage);  
printf("総割り当て: %lu バイト\n", (unsigned long)g\_rt\_allocator.total\_allocated);  
printf("総解放: %lu バイト\n", (unsigned long)g\_rt\_allocator.total\_freed);  
printf("デッドラインミス: %lu 回\n", g\_rt\_allocator.deadline\_misses);  
printf("割り当て失敗: %lu 回\n", g\_rt\_allocator.allocation\_failures);  
printf("最大割り当て時間: %lu ティック\n", g\_rt\_allocator.max\_alloc\_time);  
printf("最大解放時間: %lu ティック\n", g\_rt\_allocator.max\_free\_time);  
  
printf("\nサイズクラス情報:\n");  
printf("サイズ 総数 空き 割当数 解放数 WCET割当 WCET解放 平均割当 平均解放\n");  
printf("----- ---- ---- ------ ------ -------- -------- -------- --------\n");  
  
for (i = 0; i < RT\_SIZE\_CLASSES; i++) {  
 RTSizeClass \*sc = &g\_rt\_allocator.size\_classes[i];  
 if (sc->total\_count > 0) {  
 printf("%5lu %4lu %4lu %6lu %6lu %8lu %8lu %8.2f %8.2f\n",  
 (unsigned long)sc->size,  
 (unsigned long)sc->total\_count,  
 (unsigned long)sc->free\_count,  
 sc->total\_allocs,  
 sc->total\_frees,  
 sc->wcet\_alloc,  
 sc->wcet\_free,  
 sc->avg\_alloc\_time,  
 sc->avg\_free\_time);  
 }  
}  
  
printf("====================================\n\n");

}

/\* アロケーターのシャットダウン \*/ void rt\_allocator\_shutdown(void) { if (g\_rt\_allocator.heap) { free(g\_rt\_allocator.heap); memset(&g\_rt\_allocator, 0, sizeof(g\_rt\_allocator)); RT\_DEBUG(“リアルタイムアロケーターをシャットダウンしました”); } }

/\* テスト関数群 \*/

/\* タスク構造体（テスト用） */ typedef struct Task { int id; size\_t memory\_size; unsigned long period; unsigned long deadline; int priority; void* memory; } Task;

/\* 基本動作テスト */ void test\_basic\_realtime(void) { void* p1, *p2,* p3; unsigned long deadline;

printf("=== 基本リアルタイム動作テスト ===\n");  
  
deadline = rt\_get\_ticks() + 1000;  
  
/\* 異なるサイズの割り当て \*/  
p1 = rt\_malloc(64, deadline, 1);  
p2 = rt\_malloc(256, deadline, 2);  
p3 = rt\_malloc(1024, deadline, 3);  
  
printf("割り当て結果:\n");  
printf(" 64バイト: %p\n", p1);  
printf(" 256バイト: %p\n", p2);  
printf(" 1024バイト: %p\n", p3);  
  
/\* 解放 \*/  
rt\_free(p1);  
rt\_free(p2);  
rt\_free(p3);  
  
rt\_print\_stats();

}

/\* WCET計測テスト */ void test\_wcet\_measurement(void) { int i, j; void* ptrs[100]; unsigned long wcet\_alloc, wcet\_free;

printf("\n=== WCET計測テスト ===\n");  
  
/\* 複数回の割り当て・解放でWCETを計測 \*/  
for (i = 0; i < 10; i++) {  
 for (j = 0; j < 100; j++) {  
 ptrs[j] = rt\_malloc(128, 0, 0);  
 }  
 for (j = 0; j < 100; j++) {  
 rt\_free(ptrs[j]);  
 }  
}  
  
wcet\_alloc = rt\_get\_wcet\_alloc(128);  
wcet\_free = rt\_get\_wcet\_free(128);  
  
printf("128バイトブロックのWCET:\n");  
printf(" 割り当て: %lu ティック (安全係数込み)\n", wcet\_alloc);  
printf(" 解放: %lu ティック (安全係数込み)\n", wcet\_free);

}

/\* 周期タスクシミュレーション \*/ void test\_periodic\_tasks(void) { Task tasks[3]; int i, j; unsigned long current\_time; int cycles = 10;

printf("\n=== 周期タスクシミュレーション ===\n");  
  
/\* タスク定義 \*/  
tasks[0] = (Task){1, 512, 100, 80, 3, NULL}; /\* 高優先度 \*/  
tasks[1] = (Task){2, 1024, 200, 150, 2, NULL}; /\* 中優先度 \*/  
tasks[2] = (Task){3, 2048, 500, 400, 1, NULL}; /\* 低優先度 \*/  
  
printf("タスク構成:\n");  
for (i = 0; i < 3; i++) {  
 printf(" タスク%d: サイズ=%lu, 周期=%lu, デッドライン=%lu, 優先度=%d\n",  
 tasks[i].id,  
 (unsigned long)tasks[i].memory\_size,  
 tasks[i].period,  
 tasks[i].deadline,  
 tasks[i].priority);  
}  
  
/\* 周期実行シミュレーション \*/  
for (j = 0; j < cycles; j++) {  
 current\_time = rt\_get\_ticks();  
   
 for (i = 0; i < 3; i++) {  
 /\* 周期チェック \*/  
 if (j % (tasks[i].period / 100) == 0) {  
 /\* 前回のメモリを解放 \*/  
 if (tasks[i].memory) {  
 rt\_free(tasks[i].memory);  
 }  
   
 /\* 新規割り当て \*/  
 tasks[i].memory = rt\_malloc(tasks[i].memory\_size,  
 current\_time + tasks[i].deadline,  
 tasks[i].priority);  
   
 if (!tasks[i].memory) {  
 printf(" タスク%d: 割り当て失敗\n", tasks[i].id);  
 }  
 }  
 }  
   
 /\* 少し待機（実際のタスク実行をシミュレート） \*/  
 {  
 volatile int dummy = 0;  
 int k;  
 for (k = 0; k < 100000; k++) {  
 dummy += k;  
 }  
 }  
}  
  
/\* クリーンアップ \*/  
for (i = 0; i < 3; i++) {  
 if (tasks[i].memory) {  
 rt\_free(tasks[i].memory);  
 }  
}  
  
printf("\nシミュレーション完了\n");

}

/\* ストレステスト */ void test\_stress\_realtime(void) { void* ptrs[1000]; int i, allocated = 0; unsigned long start, end;

printf("\n=== リアルタイムストレステスト ===\n");  
  
start = rt\_get\_ticks();  
  
/\* 可能な限り割り当て \*/  
for (i = 0; i < 1000; i++) {  
 ptrs[i] = rt\_malloc(256, 0, 0);  
 if (ptrs[i]) {  
 allocated++;  
 } else {  
 break;  
 }  
}  
  
printf("最大割り当て数: %d\n", allocated);  
  
/\* 半分解放 \*/  
for (i = 0; i < allocated / 2; i++) {  
 rt\_free(ptrs[i]);  
}  
  
/\* 再割り当て \*/  
for (i = 0; i < allocated / 2; i++) {  
 ptrs[i] = rt\_malloc(256, 0, 0);  
 if (!ptrs[i]) {  
 printf("再割り当て失敗: %d\n", i);  
 break;  
 }  
}  
  
/\* 全解放 \*/  
for (i = 0; i < allocated; i++) {  
 if (ptrs[i]) {  
 rt\_free(ptrs[i]);  
 }  
}  
  
end = rt\_get\_ticks();  
printf("総実行時間: %lu ティック\n", end - start);

}

/\* 決定性検証テスト */ void test\_determinism(void) { void* p; unsigned long times[100]; unsigned long min\_time = ULONG\_MAX, max\_time = 0; double avg\_time = 0; int i;

printf("\n=== 決定性検証テスト ===\n");  
  
/\* 同じ操作を繰り返して時間のばらつきを測定 \*/  
for (i = 0; i < 100; i++) {  
 unsigned long start = rt\_get\_ticks();  
 p = rt\_malloc(512, 0, 0);  
 rt\_free(p);  
 unsigned long elapsed = rt\_get\_ticks() - start;  
   
 times[i] = elapsed;  
 if (elapsed < min\_time) min\_time = elapsed;  
 if (elapsed > max\_time) max\_time = elapsed;  
 avg\_time += elapsed;  
}  
  
avg\_time /= 100;  
  
printf("実行時間統計 (512バイト割り当て・解放):\n");  
printf(" 最小: %lu ティック\n", min\_time);  
printf(" 最大: %lu ティック\n", max\_time);  
printf(" 平均: %.2f ティック\n", avg\_time);  
printf(" ジッター: %lu ティック\n", max\_time - min\_time);  
  
/\* ジッターが小さいほど決定的 \*/  
if (max\_time - min\_time < 10) {  
 printf(" → 高い決定性\n");  
} else {  
 printf(" → 改善の余地あり\n");  
}

}

/\* メイン関数 \*/ int main(void) { printf(“=== リアルタイムメモリアロケーターデモ ===”);

/\* アロケーター初期化（1000ティック/秒） \*/  
if (rt\_allocator\_init(1000) < 0) {  
 fprintf(stderr, "アロケーターの初期化に失敗\n");  
 return 1;  
}  
  
/\* 各種テスト実行 \*/  
test\_basic\_realtime();  
test\_wcet\_measurement();  
test\_periodic\_tasks();  
test\_stress\_realtime();  
test\_determinism();  
  
/\* 最終統計 \*/  
rt\_print\_stats();  
  
/\* シャットダウン \*/  
rt\_allocator\_shutdown();  
  
printf("=== デモ完了 ===\n");  
return 0;

}

/\* 実行例: === リアルタイムメモリアロケーターデモ ===

[RT] リアルタイムアロケーターを初期化しました [RT] RT\_HEAP\_SIZE = 4194304 [RT] tick\_rate = 1000 [RT] i = 0 [RT] sc->total\_count = 16384 [RT] i = 1 [RT] sc->total\_count = 8192 …

=== 基本リアルタイム動作テスト === 割り当て結果: 64バイト: 0x1234560 256バイト: 0x1234660 1024バイト: 0x1234960

=== リアルタイムアロケーター統計 === ヒープサイズ: 4194304 バイト 現在使用量: 0 バイト (0.0%) ピーク使用量: 1344 バイト 総割り当て: 1344 バイト 総解放: 1344 バイト デッドラインミス: 0 回 割り当て失敗: 0 回 最大割り当て時間: 2 ティック 最大解放時間: 1 ティック

サイズクラス情報: サイズ 総数 空き 割当数 解放数 WCET割当 WCET解放 平均割当 平均解放 —– —- —- —— —— ——– ——– ——– ——– 32 16384 16384 0 0 0 0 0.00 0.00 64 8192 8192 1 1 2 1 2.00 1.00 128 4096 4096 0 0 0 0 0.00 0.00 256 2048 2048 1 1 1 1 1.00 1.00 512 1024 1024 0 0 0 0 0.00 0.00 1024 512 512 1 1 1 1 1.00 1.00 ====================================

=== WCET計測テスト === 128バイトブロックのWCET: 割り当て: 2 ティック (安全係数込み) 解放: 1 ティック (安全係数込み)

=== 周期タスクシミュレーション === タスク構成: タスク1: サイズ=512, 周期=100, デッドライン=80, 優先度=3 タスク2: サイズ=1024, 周期=200, デッドライン=150, 優先度=2 タスク3: サイズ=2048, 周期=500, デッドライン=400, 優先度=1

シミュレーション完了

=== リアルタイムストレステスト === 最大割り当て数: 768 総実行時間: 45 ティック

=== 決定性検証テスト === 実行時間統計 (512バイト割り当て・解放): 最小: 2 ティック 最大: 3 ティック 平均: 2.12 ティック ジッター: 1 ティック → 高い決定性

=== リアルタイムアロケーター統計 === ヒープサイズ: 4194304 バイト 現在使用量: 0 バイト (0.0%) ピーク使用量: 196608 バイト 総割り当て: 318976 バイト 総解放: 318976 バイト デッドラインミス: 0 回 割り当て失敗: 232 回 最大割り当て時間: 3 ティック 最大解放時間: 2 ティック

サイズクラス情報: サイズ 総数 空き 割当数 解放数 WCET割当 WCET解放 平均割当 平均解放 —– —- —- —— —— ——– ——– ——– ——– 32 16384 16384 0 0 0 0 0.00 0.00 64 8192 8192 1 1 2 1 2.00 1.00 128 4096 4096 1000 1000 2 1 1.23 0.98 256 2048 2048 769 769 3 2 2.45 1.76 512 1024 1024 123 123 3 2 2.12 1.88 1024 512 512 31 31 2 2 1.84 1.65 2048 256 256 20 20 2 1 1.75 1.45 ====================================

[RT] リアルタイムアロケーターをシャットダウンしました === デモ完了 === \*/```