# 第10章 ポインタ基礎

## 対応C規格

* **主要対象:** C90
* **学習内容:** ポインタの基本概念、アドレス演算子、間接参照演算子、ポインタと配列、ポインタ演算

## 学習目標

この章を完了すると、以下のことができるようになります：

* ポインタの基本概念を理解する
* アドレス演算子（&）と間接参照演算子（\*）を使える
* ポインタと配列の関係を理解する
* ポインタ演算ができる
* ポインタを関数の引数として使える

## 概要と詳細

### ポインタとは？

ポインタは、C言語で最も重要かつ難しいと言われる概念です。しかし、適切な例えを使えば理解できます！

#### 日常生活でのポインタ

ポインタを理解するために、「住所」の例えを使いましょう：

1. **家 = 変数**
   * 実際に人が住んでいる場所
   * 中に値（住人）が入っている
2. **住所 = ポインタ**
   * 家の場所を示す情報
   * 住所を知っていれば、その家に行ける
3. **住所録 = ポインタ変数**
   * 住所を書き留めておく手帳
   * 違う住所に書き換えることもできる

int house = 100; /\* 家（変数）に100という値が住んでいる \*/  
int \*address = &house; /\* 住所録にhouseの住所を記録 \*/

### ポインタの基本概念

ポインタは他の変数のメモリアドレスを格納する変数です。これにより間接的に他の変数にアクセスできます。

#### なぜポインタが必要？

1. **大きなデータを効率的に扱える**
   * データのコピーではなく、場所だけを教える
2. **関数で複数の値を返せる**
   * 通常の関数は1つしか値を返せないが、ポインタなら複数可能
3. **動的なメモリ管理ができる**
   * プログラム実行中に必要なメモリを確保

#### メモリとアドレス

#include <stdio.h>  
  
int main(void)  
{  
 int number = 42;  
   
 printf("変数numberの値: %d\n", number);  
 printf("変数numberのアドレス: %p\n", (void\*)&number);  
 printf("変数numberのサイズ: %lu バイト\n", (unsigned long)sizeof(number));  
   
 return 0;  
}

#### ポインタ変数の宣言

#include <stdio.h>  
  
int main(void)  
{  
 int value = 100; /\* 通常の整数変数 \*/  
 int \*ptr; /\* 整数を指すポインタ変数 \*/  
   
 ptr = &value; /\* valueのアドレスをptrに代入 \*/  
   
 printf("value = %d\n", value);  
 printf("&value = %p\n", (void\*)&value);  
 printf("ptr = %p\n", (void\*)ptr);  
 printf("\*ptr = %d\n", \*ptr); /\* ポインタが指す値 \*/  
   
 return 0;  
}

### アドレス演算子（&）と間接参照演算子（\*）

この2つの演算子は、ポインタを使う上で最も重要です。

#### 演算子の意味を覚える方法

* **& = “の住所”**（address of）
  + &house = “houseの住所”
* **\* = “が指す場所の中身”**（value at）
  + \*address = “addressが指す場所の中身”

#### アドレス演算子（&）

#include <stdio.h>  
  
int main(void)  
{  
 int a = 10;  
 double b = 3.14;  
 char c = 'A';  
   
 printf("変数のアドレス:\n");  
 printf("&a = %p\n", (void\*)&a);  
 printf("&b = %p\n", (void\*)&b);  
 printf("&c = %p\n", (void\*)&c);  
   
 /\* ポインタ変数の宣言と初期化 \*/  
 int \*ptr\_a = &a;  
 double \*ptr\_b = &b;  
 char \*ptr\_c = &c;  
   
 printf("\nポインタの値（アドレス）:\n");  
 printf("ptr\_a = %p\n", (void\*)ptr\_a);  
 printf("ptr\_b = %p\n", (void\*)ptr\_b);  
 printf("ptr\_c = %p\n", (void\*)ptr\_c);  
   
 return 0;  
}

#### 間接参照演算子（\*）

#include <stdio.h>  
  
int main(void)  
{  
 int original = 50;  
 int \*pointer = &original;  
   
 printf("=== 元の状態 ===\n");  
 printf("original = %d\n", original);  
 printf("\*pointer = %d\n", \*pointer);  
   
 /\* ポインタを通じて値を変更 \*/  
 \*pointer = 75;  
   
 printf("\n=== \*pointer = 75 実行後 ===\n");  
 printf("original = %d\n", original); /\* 75に変更される \*/  
 printf("\*pointer = %d\n", \*pointer); /\* 75 \*/  
   
 /\* 元の変数を直接変更 \*/  
 original = 99;  
   
 printf("\n=== original = 99 実行後 ===\n");  
 printf("original = %d\n", original); /\* 99 \*/  
 printf("\*pointer = %d\n", \*pointer); /\* 99 \*/  
   
 return 0;  
}

### ポインタのデータ型

#### さまざまなデータ型のポインタ

#include <stdio.h>  
  
int main(void)  
{  
 /\* 各データ型の変数 \*/  
 char char\_var = 'X';  
 int int\_var = 123;  
 float float\_var = 3.14f;  
 double double\_var = 2.718;  
   
 /\* 各データ型のポインタ \*/  
 char \*char\_ptr = &char\_var;  
 int \*int\_ptr = &int\_var;  
 float \*float\_ptr = &float\_var;  
 double \*double\_ptr = &double\_var;  
   
 printf("=== 値の表示 ===\n");  
 printf("char: %c\n", \*char\_ptr);  
 printf("int: %d\n", \*int\_ptr);  
 printf("float: %.2f\n", \*float\_ptr);  
 printf("double: %.3f\n", \*double\_ptr);  
   
 printf("\n=== ポインタのサイズ ===\n");  
 printf("char\*: %lu バイト\n", (unsigned long)sizeof(char\_ptr));  
 printf("int\*: %lu バイト\n", (unsigned long)sizeof(int\_ptr));  
 printf("float\*: %lu バイト\n", (unsigned long)sizeof(float\_ptr));  
 printf("double\*: %lu バイト\n", (unsigned long)sizeof(double\_ptr));  
   
 return 0;  
}

#### void\*ポインタ（汎用ポインタ）

#include <stdio.h>  
  
int main(void)  
{  
 int int\_value = 456;  
 double double\_value = 1.618;  
   
 void \*generic\_ptr; /\* 汎用ポインタ \*/  
   
 /\* intを指す \*/  
 generic\_ptr = &int\_value;  
 printf("int値: %d\n", \*(int\*)generic\_ptr); /\* キャストが必要 \*/  
   
 /\* doubleを指す \*/  
 generic\_ptr = &double\_value;  
 printf("double値: %.3f\n", \*(double\*)generic\_ptr); /\* キャストが必要 \*/  
   
 return 0;  
}

### ポインタと配列の関係

ポインタと配列は非常に密接な関係があります。実は、配列名そのものがポインタとして扱われることを理解すると、C言語がより深く理解できます。

#### 配列名はポインタ

配列名は「配列の最初の要素を指すポインタ」として扱われます。これは重要な概念です！

#include <stdio.h>  
  
int main(void)  
{  
 int arr[] = {10, 20, 30, 40, 50};  
 int \*ptr = arr; /\* arr は &arr[0] と同じ \*/  
 int i;  
   
 printf("=== 配列とポインタの関係 ===\n");  
 printf("arr = %p\n", (void\*)arr);  
 printf("&arr[0] = %p\n", (void\*)&arr[0]);  
 printf("ptr = %p\n", (void\*)ptr);  
   
 printf("\n=== 配列要素へのアクセス ===\n");  
 for (i = 0; i < 5; i++) {  
 printf("arr[%d] = %d, \*(arr + %d) = %d, \*(ptr + %d) = %d\n",  
 i, arr[i], i, \*(arr + i), i, \*(ptr + i));  
 }  
   
 return 0;  
}

#### ポインタを使った配列操作

#include <stdio.h>  
  
void print\_array\_index(int arr[], int size)  
{  
 int i;  
   
 printf("インデックス記法: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
void print\_array\_pointer(int \*ptr, int size)  
{  
 int i;  
   
 printf("ポインタ記法: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*(ptr + i));  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 int numbers[] = {11, 22, 33, 44, 55};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
   
 print\_array\_index(numbers, size);  
 print\_array\_pointer(numbers, size);  
   
 return 0;  
}

### ポインタ演算

ポインタには通常の数値とは異なる特殊な演算規則があります。これを理解することで、配列やメモリを効率的に操作できるようになります。

#### なぜポインタ演算が特殊なのか？

ポインタに1を足しても、アドレスが1増えるわけではありません。代わりに「次の要素」に移動します。

int配列の場合：  
ptr + 1 → 次のint（通常4バイト先）へ移動  
  
char配列の場合：  
ptr + 1 → 次のchar（1バイト先）へ移動

#### ポインタの加算・減算

#include <stdio.h>  
  
int main(void)  
{  
 int arr[] = {5, 15, 25, 35, 45};  
 int \*ptr = arr;  
 int i;  
   
 printf("=== ポインタ演算の例 ===\n");  
 for (i = 0; i < 5; i++) {  
 printf("ptr + %d = %p, \*(ptr + %d) = %d\n",  
 i, (void\*)(ptr + i), i, \*(ptr + i));  
 }  
   
 printf("\n=== ポインタのインクリメント ===\n");  
 ptr = arr; /\* 先頭に戻す \*/  
 for (i = 0; i < 5; i++) {  
 printf("\*ptr = %d (アドレス: %p)\n", \*ptr, (void\*)ptr);  
 ptr++; /\* 次の要素に移動 \*/  
 }  
   
 return 0;  
}

#### ポインタ同士の差

#include <stdio.h>  
  
int main(void)  
{  
 int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
 int \*start = &arr[2]; /\* arr[2]を指す \*/  
 int \*end = &arr[7]; /\* arr[7]を指す \*/  
   
 printf("start が指す値: %d (インデックス2)\n", \*start);  
 printf("end が指す値: %d (インデックス7)\n", \*end);  
 printf("ポインタの差: %ld\n", end - start); /\* 5 \*/  
 printf("バイト差: %ld\n", (char\*)end - (char\*)start);  
   
 return 0;  
}

### ポインタと関数

関数とポインタを組み合わせることで、C言語の真の力を発揮できます。特に、関数で複数の値を変更したり、大きなデータを効率的に扱えるようになります。

#### なぜ関数でポインタを使うのか？

1. **複数の値を変更できる**
   * 通常の関数は1つの値しか返せない
   * ポインタなら複数の値を同時に変更可能
2. **効率的なデータ処理**
   * 大きな配列や構造体をコピーせずに処理
   * メモリと処理時間を節約
3. **実際の変数を変更できる**
   * 値渡しでは元の変数は変わらない
   * ポインタ渡しなら元の変数を直接変更

#### ポインタを引数とする関数

#include <stdio.h>  
  
/\* 値を交換する関数 \*/  
void swap(int \*a, int \*b)  
{  
 int temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
}  
  
/\* 値を倍にする関数 \*/  
void double\_value(int \*value)  
{  
 \*value = (\*value) \* 2;  
}  
  
/\* 配列の要素を変更する関数 \*/  
void modify\_array(int arr[], int size)  
{  
 int i;  
   
 for (i = 0; i < size; i++) {  
 arr[i] \*= 3; /\* 各要素を3倍 \*/  
 }  
}  
  
int main(void)  
{  
 int x = 8, y = 12;  
 int numbers[] = {1, 2, 3, 4, 5};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
 int i;  
   
 printf("=== swap関数のテスト ===\n");  
 printf("交換前: x = %d, y = %d\n", x, y);  
 swap(&x, &y);  
 printf("交換後: x = %d, y = %d\n", x, y);  
   
 printf("\n=== double\_value関数のテスト ===\n");  
 printf("2倍前: x = %d\n", x);  
 double\_value(&x);  
 printf("2倍後: x = %d\n", x);  
   
 printf("\n=== modify\_array関数のテスト ===\n");  
 printf("変更前: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 modify\_array(numbers, size);  
   
 printf("変更後: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", numbers[i]);  
 }  
 printf("\n");  
   
 return 0;  
}

#### ポインタを戻り値とする関数

#include <stdio.h>  
  
/\* 配列から最大値のアドレスを返す \*/  
int\* find\_max\_ptr(int arr[], int size)  
{  
 int \*max\_ptr = &arr[0];  
 int i;  
   
 for (i = 1; i < size; i++) {  
 if (arr[i] > \*max\_ptr) {  
 max\_ptr = &arr[i];  
 }  
 }  
   
 return max\_ptr;  
}  
  
/\* 配列から指定値を検索してアドレスを返す \*/  
int\* search\_value(int arr[], int size, int target)  
{  
 int i;  
   
 for (i = 0; i < size; i++) {  
 if (arr[i] == target) {  
 return &arr[i]; /\* 見つかった要素のアドレス \*/  
 }  
 }  
   
 return NULL; /\* 見つからない場合 \*/  
}  
  
int main(void)  
{  
 int data[] = {3, 7, 2, 9, 5, 1, 8};  
 int size = sizeof(data) / sizeof(data[0]);  
 int \*result\_ptr;  
 int search\_target = 9;  
 int i;  
   
 /\* 配列の表示 \*/  
 printf("配列: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", data[i]);  
 }  
 printf("\n");  
   
 /\* 最大値の検索 \*/  
 result\_ptr = find\_max\_ptr(data, size);  
 printf("最大値: %d (アドレス: %p)\n", \*result\_ptr, (void\*)result\_ptr);  
   
 /\* 特定値の検索 \*/  
 result\_ptr = search\_value(data, size, search\_target);  
 if (result\_ptr != NULL) {  
 printf("%d が見つかりました (アドレス: %p)\n",   
 \*result\_ptr, (void\*)result\_ptr);  
 } else {  
 printf("%d は見つかりませんでした\n", search\_target);  
 }  
   
 return 0;  
}

## サンプルコード

### 基本的なポインタ操作

プログラムファイル: examples/pointer\_basic.c

ポインタの宣言、初期化、参照を学習します。

### ポインタと配列

プログラムファイル: examples/pointer\_array.c

配列とポインタの関係、ポインタ演算を学習します。

### ポインタと関数

プログラムファイル: examples/pointer\_function.c

関数でのポインタの使い方を学習します。

### コンパイルと実行

# examples ディレクトリに移動  
cd examples  
  
# C90準拠でコンパイル  
gcc -std=c90 -Wall -Wextra -pedantic pointer\_basic.c -o pointer\_basic  
  
# 実行  
./pointer\_basic

## 演習課題

### 基礎問題

1. **ポインタの基本操作**
   * 2つの変数の値をポインタを使って交換するプログラムを作成してください
2. **配列とポインタ**
   * ポインタ演算を使って配列の要素を逆順に表示するプログラムを作成してください
3. **文字列操作**
   * ポインタを使って文字列の長さを計算し、文字列を逆順にするプログラムを作成してください

### 応用問題

1. **配列操作関数**
   * ポインタを使って配列の最大値、最小値、平均値を計算する関数を作成してください
2. **文字列処理**
   * ポインタを使って文字列の検索、置換、分割を実行する関数群を作成してください
3. **データ変換**
   * ポインタを使って配列のデータ型変換を実行するプログラムを作成してください

### 発展問題

1. **メモリ操作**
   * ポインタを使って任意のデータ型の配列をコピーする汎用関数を作成してください
2. **アルゴリズム実装**
   * ポインタを使って各種ソートアルゴリズムを実装してください

## コンパイル方法

この章では以下のMakefileを使用してコンパイルができます：

# 全ての例題をコンパイル  
make all  
  
# 特定のプログラムをコンパイル  
make pointer\_basic  
  
# 実行  
make run  
  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* void\*からの暗黙的な型変換は制限的
* ポインタ演算は整数型との組み合わせのみ
* 関数ポインタの表記法は限定的

### C99以降の拡張

* restrict修飾子の追加
* 可変長配列とポインタの関係改善
* より厳密なポインタ型チェック

## よくある間違い

### 1. 未初期化ポインタの使用

/\* NG: 初期化されていないポインタ \*/  
int \*ptr;  
\*ptr = 10; /\* 危険: 不明なメモリへの書き込み \*/  
  
/\* OK: 適切に初期化 \*/  
int value = 0;  
int \*ptr = &value;  
\*ptr = 10;

### 2. NULLポインタの参照

/\* NG: NULLポインタの参照 \*/  
int \*ptr = NULL;  
printf("%d\n", \*ptr); /\* セグメンテーション違反 \*/  
  
/\* OK: NULLチェック \*/  
if (ptr != NULL) {  
 printf("%d\n", \*ptr);  
}

### 3. スコープ外変数へのポインタ

/\* NG: ローカル変数のアドレス返し \*/  
int\* bad\_function(void)  
{  
 int local = 100;  
 return &local; /\* 危険: ローカル変数のアドレス \*/  
}  
  
/\* OK: 静的変数または引数で渡された変数を使用 \*/  
int\* good\_function(int \*param)  
{  
 return param; /\* 引数で渡されたアドレスを返す \*/  
}

### 4. 配列境界の越えた参照

/\* NG: 配列の範囲外アクセス \*/  
int arr[] = {1, 2, 3, 4, 5};  
int \*ptr = arr;  
printf("%d\n", \*(ptr + 10)); /\* 範囲外アクセス \*/  
  
/\* OK: 範囲チェック \*/  
int index = 2;  
if (index >= 0 && index < 5) {  
 printf("%d\n", \*(ptr + index));  
}

## 次の章へ

ポインタの基礎を理解したら、[構造体](../structures/README.md) に進んでください。

## 参考資料

* [C言語ポインタリファレンス](https://ja.cppreference.com/w/c/language/pointer)
* [ポインタ演算の詳細](https://ja.cppreference.com/w/c/language/operator_arithmetic)
* [メモリモデルとポインタ](https://ja.cppreference.com/w/c/language/memory_model)

# サンプルコード

## pointer\_arrays.c

/\*  
 \* pointer\_arrays.c  
 \* ポインタと配列の関係、ポインタ演算を学習  
 \* C90準拠  
 \*/  
  
#include <stdio.h>  
  
void print\_array\_with\_index(int arr[], int size)  
{  
 int i;  
 printf("インデックス記法: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
void print\_array\_with\_pointer(int \*ptr, int size)  
{  
 int i;  
 printf("ポインタ記法: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*(ptr + i));  
 }  
 printf("\n");  
}  
  
void print\_array\_with\_increment(int \*ptr, int size)  
{  
 int i;  
 printf("インクリメント: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*ptr);  
 ptr++; /\* ポインタを次の要素に進める \*/  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 int numbers[6] = {10, 20, 30, 40, 50, 60};  
 int \*ptr = numbers; /\* 配列名は先頭要素のアドレス \*/  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
 int i;  
 int \*start; /\* C90: ポインタ変数の宣言を先頭に配置 \*/  
 int \*end;  
 char message[] = "Hello, World!";  
 char \*str\_ptr;  
 int str\_len = 0;  
 int matrix[3][4] = {  
 {1, 2, 3, 4},  
 {5, 6, 7, 8},  
 {9, 10, 11, 12}  
 };  
 int \*matrix\_ptr;  
 int total\_elements = 3 \* 4;  
 int a = 100, b = 200, c = 300;  
 int \*ptr\_array[3]; /\* ポインタの配列 \*/  
   
 printf("===== 配列とポインタの関係 =====\n");  
 printf("配列名numbers: %p\n", (void\*)numbers);  
 printf("&numbers[0]: %p\n", (void\*)&numbers[0]);  
 printf("ポインタptr: %p\n", (void\*)ptr);  
 printf("これらはすべて同じアドレスです\n");  
   
 printf("\n===== 配列要素へのアクセス方法 =====\n");  
 print\_array\_with\_index(numbers, size);  
 print\_array\_with\_pointer(ptr, size);  
   
 /\* ポインタをリセット（print\_array\_with\_incrementでptrが変更されるため） \*/  
 ptr = numbers;  
 print\_array\_with\_increment(ptr, size);  
   
 printf("\n===== ポインタ演算の詳細 =====\n");  
 ptr = numbers; /\* 先頭に戻す \*/  
   
 for (i = 0; i < size; i++) {  
 printf("ptr + %d = %p, \*(ptr + %d) = %d\n",  
 i, (void\*)(ptr + i), i, \*(ptr + i));  
 }  
   
 printf("\n===== アドレスの差による要素間隔の確認 =====\n");  
 printf("numbers[0]のアドレス: %p\n", (void\*)&numbers[0]);  
 printf("numbers[1]のアドレス: %p\n", (void\*)&numbers[1]);  
 printf("アドレス差: %ld バイト\n",   
 (char\*)&numbers[1] - (char\*)&numbers[0]);  
 printf("int型のサイズ: %lu バイト\n", (unsigned long)sizeof(int));  
   
 printf("\n===== ポインタによる配列の逆順表示 =====\n");  
 ptr = numbers + size - 1; /\* 最後の要素を指す \*/  
 printf("逆順: ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", \*ptr);  
 ptr--; /\* ポインタを前の要素に戻す \*/  
 }  
 printf("\n");  
   
 printf("\n===== ポインタ同士の演算 =====\n");  
 start = &numbers[1]; /\* numbers[1]を指す \*/  
 end = &numbers[4]; /\* numbers[4]を指す \*/  
   
 printf("start が指す値: %d (numbers[1])\n", \*start);  
 printf("end が指す値: %d (numbers[4])\n", \*end);  
 printf("ポインタの差: %ld (要素数)\n", end - start);  
 printf("バイト差: %ld バイト\n", (char\*)end - (char\*)start);  
   
 printf("\n===== 文字列（文字配列）とポインタ =====\n");  
 str\_ptr = message;  
   
 printf("文字列: %s\n", message);  
 printf("文字ごとの表示: ");  
   
 /\* 文字列の終端（'\0'）まで表示 \*/  
 while (\*str\_ptr != '\0') {  
 printf("%c", \*str\_ptr);  
 str\_ptr++;  
 str\_len++;  
 }  
 printf("\n");  
 printf("文字列の長さ: %d文字\n", str\_len);  
   
 printf("\n===== 2次元配列とポインタ =====\n");  
 matrix\_ptr = (int\*)matrix; /\* 1次元ポインタとして扱う \*/  
   
 printf("2次元配列を1次元ポインタで表示:\n");  
 for (i = 0; i < total\_elements; i++) {  
 printf("%2d ", \*(matrix\_ptr + i));  
 if ((i + 1) % 4 == 0) {  
 printf("\n");  
 }  
 }  
   
 printf("\n===== ポインタ配列の例 =====\n");  
   
 ptr\_array[0] = &a;  
 ptr\_array[1] = &b;  
 ptr\_array[2] = &c;  
   
 printf("ポインタ配列の内容:\n");  
 for (i = 0; i < 3; i++) {  
 printf("ptr\_array[%d] = %p, \*ptr\_array[%d] = %d\n",  
 i, (void\*)ptr\_array[i], i, \*ptr\_array[i]);  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 配列名は先頭要素のポインタと同等  
2. ポインタ演算: ptr + n は n個後の要素を指す  
3. \*(ptr + i) と ptr[i] は同等  
4. ポインタのインクリメント/デクリメント  
5. ポインタ同士の差は要素数を表す  
6. 文字列もポインタで操作可能  
7. 2次元配列も1次元ポインタで扱える  
8. ポインタの配列の活用  
  
重要な概念:  
- 配列の記法とポインタ記法の等価性  
- ポインタ演算の自動的な型サイズ調整  
- メモリレイアウトの理解  
- 安全な境界チェックの重要性  
\*/

## pointer\_basic.c

/\*  
 \* pointer\_basic.c  
 \* ポインタの宣言、初期化、基本的な操作を学習  
 \* C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void)  
{  
 int value = 42; /\* 通常の整数変数 \*/  
 int \*ptr; /\* 整数ポインタ変数の宣言 \*/  
 int another\_value = 100;  
 char char\_var = 'A'; /\* C90: 全変数宣言を先頭に配置 \*/  
 float float\_var = 3.14f;  
 double double\_var = 2.718;  
 char \*char\_ptr;  
 float \*float\_ptr;  
 double \*double\_ptr;  
 int \*null\_ptr; /\* C90: NULLポインタ用変数 \*/  
 void \*generic\_ptr; /\* C90: void\*ポインタ用変数 \*/  
 int int\_val = 123;  
 double double\_val = 45.67;  
   
 printf("===== ポインタの基本操作 =====\n");  
   
 /\* ポインタの初期化 \*/  
 ptr = &value; /\* valueのアドレスをptrに代入 \*/  
   
 printf("変数valueの値: %d\n", value);  
 printf("変数valueのアドレス: %p\n", (void\*)&value);  
 printf("ポインタptrの値（アドレス）: %p\n", (void\*)ptr);  
 printf("ポインタptrが指す値: %d\n", \*ptr);  
   
 printf("\n===== 間接参照による値の変更 =====\n");  
 printf("変更前: value = %d, \*ptr = %d\n", value, \*ptr);  
   
 \*ptr = 75; /\* ポインタを通じて値を変更 \*/  
 printf("\*ptr = 75 実行後:\n");  
 printf("変更後: value = %d, \*ptr = %d\n", value, \*ptr);  
   
 printf("\n===== ポインタの再代入 =====\n");  
 printf("another\_value = %d\n", another\_value);  
   
 ptr = &another\_value; /\* ptrを別の変数に向ける \*/  
 printf("ptr = &another\_value 実行後:\n");  
 printf("ポインタptrが指す値: %d\n", \*ptr);  
 printf("ポインタptrのアドレス: %p\n", (void\*)ptr);  
   
 printf("\n===== 様々なデータ型のポインタ =====\n");  
   
 /\* C90: ポインタの初期化は宣言後に \*/  
 char\_ptr = &char\_var;  
 float\_ptr = &float\_var;  
 double\_ptr = &double\_var;  
   
 printf("char値: %c (アドレス: %p)\n", \*char\_ptr, (void\*)char\_ptr);  
 printf("float値: %.2f (アドレス: %p)\n", \*float\_ptr, (void\*)float\_ptr);  
 printf("double値: %.3f (アドレス: %p)\n", \*double\_ptr, (void\*)double\_ptr);  
   
 printf("\n===== ポインタのサイズ =====\n");  
 printf("int\*のサイズ: %lu バイト\n", (unsigned long)sizeof(int\*));  
 printf("char\*のサイズ: %lu バイト\n", (unsigned long)sizeof(char\*));  
 printf("float\*のサイズ: %lu バイト\n", (unsigned long)sizeof(float\*));  
 printf("double\*のサイズ: %lu バイト\n", (unsigned long)sizeof(double\*));  
   
 printf("\n===== NULLポインタ =====\n");  
 null\_ptr = NULL; /\* C90: 初期化は宣言後に \*/  
 printf("NULLポインタの値: %p\n", (void\*)null\_ptr);  
   
 if (null\_ptr == NULL) {  
 printf("null\_ptrはNULLです（安全チェック）\n");  
 }  
   
 /\* NULLポインタの参照は危険なのでコメントアウト \*/  
 /\* printf("NULL参照: %d\n", \*null\_ptr); // 危険！ \*/  
   
 printf("\n===== void\*ポインタ（汎用ポインタ） =====\n");  
   
 generic\_ptr = &int\_val;  
 printf("intを指すvoid\*: %d\n", \*(int\*)generic\_ptr);  
   
 generic\_ptr = &double\_val;  
 printf("doubleを指すvoid\*: %.2f\n", \*(double\*)generic\_ptr);  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. ポインタの宣言: int \*ptr;  
2. アドレス演算子: &variable  
3. 間接参照演算子: \*pointer  
4. ポインタの初期化と再代入  
5. NULLポインタの安全な使用  
6. void\*ポインタと型キャスト  
7. ポインタのサイズは型に関係なく一定  
8. 安全なプログラミングのためのNULLチェック  
  
注意点:  
- 未初期化ポインタの使用は危険  
- NULLポインタの参照は避ける  
- void\*ポインタは使用時に適切な型キャストが必要  
\*/

## pointer\_functions.c

/\*  
 \* pointer\_functions.c  
 \* 関数引数、戻り値としてのポインタの使用方法を学習  
 \* C90準拠  
 \*/  
  
#include <stdio.h>  
#include <string.h>  
  
/\* 2つの値を交換する関数 \*/  
void swap\_values(int \*a, int \*b)  
{  
 int temp;  
   
 printf(" swap関数内: 交換前 \*a = %d, \*b = %d\n", \*a, \*b);  
   
 temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
   
 printf(" swap関数内: 交換後 \*a = %d, \*b = %d\n", \*a, \*b);  
}  
  
/\* 値を2倍にする関数 \*/  
void double\_value(int \*value)  
{  
 printf(" double\_value関数内: 変更前 \*value = %d\n", \*value);  
 \*value = (\*value) \* 2;  
 printf(" double\_value関数内: 変更後 \*value = %d\n", \*value);  
}  
  
/\* 配列の合計を計算する関数 \*/  
int sum\_array(int \*arr, int size)  
{  
 int sum = 0;  
 int i;  
   
 for (i = 0; i < size; i++) {  
 sum += \*(arr + i); /\* ポインタ記法 \*/  
 }  
   
 return sum;  
}  
  
/\* 配列の要素を変更する関数 \*/  
void modify\_array(int \*arr, int size, int multiplier)  
{  
 int i;  
   
 printf(" modify\_array関数内: 変更前\n ");  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
   
 for (i = 0; i < size; i++) {  
 arr[i] \*= multiplier;  
 }  
   
 printf(" modify\_array関数内: 変更後（%d倍）\n ", multiplier);  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
/\* 配列から最大値のポインタを返す関数 \*/  
int\* find\_max\_pointer(int \*arr, int size)  
{  
 int \*max\_ptr = arr; /\* 最初の要素を最大とする \*/  
 int i;  
   
 for (i = 1; i < size; i++) {  
 if (\*(arr + i) > \*max\_ptr) {  
 max\_ptr = arr + i;  
 }  
 }  
   
 return max\_ptr;  
}  
  
/\* 配列から指定値を検索してポインタを返す関数 \*/  
int\* search\_value(int \*arr, int size, int target)  
{  
 int i;  
   
 for (i = 0; i < size; i++) {  
 if (\*(arr + i) == target) {  
 return arr + i; /\* 見つかった要素のポインタを返す \*/  
 }  
 }  
   
 return NULL; /\* 見つからない場合はNULLを返す \*/  
}  
  
/\* 文字列の長さを計算する関数（ポインタ版） \*/  
int string\_length\_ptr(char \*str)  
{  
 int length = 0;  
   
 while (\*str != '\0') {  
 length++;  
 str++;  
 }  
   
 return length;  
}  
  
/\* 文字列をコピーする関数（ポインタ版） \*/  
void string\_copy\_ptr(char \*dest, char \*src)  
{  
 while (\*src != '\0') {  
 \*dest = \*src;  
 dest++;  
 src++;  
 }  
 \*dest = '\0'; /\* null終端文字を追加 \*/  
}  
  
/\* 配列を初期化する関数 \*/  
void init\_array(int \*arr, int size, int init\_value)  
{  
 int i;  
   
 for (i = 0; i < size; i++) {  
 \*(arr + i) = init\_value + i; /\* 初期値 + インデックス \*/  
 }  
}  
  
/\* 配列を表示する関数 \*/  
void print\_array(char \*label, int \*arr, int size)  
{  
 int i;  
   
 printf("%s: ", label);  
 for (i = 0; i < size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
}  
  
int main(void)  
{  
 /\* C90: 全変数宣言を先頭に配置 \*/  
 int x = 10, y = 20;  
 int numbers[6];  
 int size;  
 int total;  
 int data[] = {45, 67, 23, 89, 12, 56, 78};  
 int data\_size;  
 int \*result\_ptr;  
 int search\_target = 56;  
 char original[] = "Hello, World!";  
 char copy\_buffer[50];  
 char \*fruits[] = {"Apple", "Banana", "Cherry", "Date"};  
 int fruit\_count;  
 int i;  
 int \*arrays[2]; /\* C90では可変初期化ができないため配列サイズを固定 \*/  
 int sizes[2];  
 char \*labels[] = {"numbers配列", "data配列"};  
 int array\_sum;  
   
 printf("===== ポインタを引数とする関数 =====\n");  
   
 /\* 変数の初期化 \*/  
 size = sizeof(numbers) / sizeof(numbers[0]);  
 data\_size = sizeof(data) / sizeof(data[0]);  
 fruit\_count = sizeof(fruits) / sizeof(fruits[0]);  
   
 /\* swap関数のテスト \*/  
 printf("\nswap関数のテスト:\n");  
 printf("交換前: x = %d, y = %d\n", x, y);  
 swap\_values(&x, &y);  
 printf("交換後: x = %d, y = %d\n", x, y);  
   
 /\* double\_value関数のテスト \*/  
 printf("\ndouble\_value関数のテスト:\n");  
 printf("2倍前: x = %d\n", x);  
 double\_value(&x);  
 printf("2倍後: x = %d\n", x);  
   
 printf("\n===== 配列を操作する関数 =====\n");  
   
 /\* 配列の初期化 \*/  
 init\_array(numbers, size, 10);  
 print\_array("初期化後", numbers, size);  
   
 /\* 配列の合計計算 \*/  
 total = sum\_array(numbers, size);  
 printf("配列の合計: %d\n", total);  
   
 /\* 配列の変更 \*/  
 printf("\n配列変更のテスト:\n");  
 modify\_array(numbers, size, 3);  
 print\_array("main関数内での変更後", numbers, size);  
   
 printf("\n===== ポインタを戻り値とする関数 =====\n");  
   
 print\_array("検索対象配列", data, data\_size);  
   
 /\* 最大値の検索 \*/  
 result\_ptr = find\_max\_pointer(data, data\_size);  
 if (result\_ptr != NULL) {  
 printf("最大値: %d (アドレス: %p)\n", \*result\_ptr, (void\*)result\_ptr);  
 printf("最大値のインデックス: %ld\n", result\_ptr - data);  
 }  
   
 /\* 特定値の検索 \*/  
 result\_ptr = search\_value(data, data\_size, search\_target);  
 if (result\_ptr != NULL) {  
 printf("%d が見つかりました (アドレス: %p)\n",   
 \*result\_ptr, (void\*)result\_ptr);  
 printf("見つかった位置のインデックス: %ld\n", result\_ptr - data);  
   
 /\* 見つかった値を変更 \*/  
 \*result\_ptr = 999;  
 print\_array("値変更後", data, data\_size);  
 } else {  
 printf("%d は見つかりませんでした\n", search\_target);  
 }  
   
 printf("\n===== 文字列操作関数 =====\n");  
   
 printf("元の文字列: \"%s\"\n", original);  
 printf("文字列の長さ: %d文字\n", string\_length\_ptr(original));  
   
 /\* 文字列のコピー \*/  
 string\_copy\_ptr(copy\_buffer, original);  
 printf("コピーした文字列: \"%s\"\n", copy\_buffer);  
   
 printf("\n===== ポインタによる高度な操作 =====\n");  
   
 /\* ポインタの配列を使った複数の文字列 \*/  
   
 printf("果物リスト:\n");  
 for (i = 0; i < fruit\_count; i++) {  
 printf(" %d. %s (長さ: %d文字)\n",   
 i + 1, fruits[i], string\_length\_ptr(fruits[i]));  
 }  
   
 /\* 複数の配列を処理 \*/  
 /\* C90: 配列初期化は宣言時のみ可能なため、手動で代入 \*/  
 arrays[0] = numbers;  
 arrays[1] = data;  
 sizes[0] = size;  
 sizes[1] = data\_size;  
   
 printf("\n複数配列の合計:\n");  
 for (i = 0; i < 2; i++) {  
 array\_sum = sum\_array(arrays[i], sizes[i]);  
 printf(" %s の合計: %d\n", labels[i], array\_sum);  
 }  
   
 return 0;  
}  
  
/\*  
学習ポイント:  
1. 関数引数としてのポインタ:  
 - 値の変更が可能（参照渡し）  
 - 配列の受け渡し  
 - NULLチェックの重要性  
  
2. 戻り値としてのポインタ:  
 - 配列要素のアドレス返し  
 - NULLによるエラー表現  
 - ローカル変数のアドレス返しは危険  
  
3. 実用的なポインタ活用:  
 - 配列操作の効率化  
 - 文字列処理  
 - データ検索と変更  
 - 複数データの一括処理  
  
4. メモリ安全性:  
 - 境界チェック  
 - NULLポインタチェック  
 - 適切な初期化  
  
注意点:  
- ローカル変数のアドレスを返してはいけない  
- 配列の範囲外アクセスに注意  
- NULLポインタの参照は避ける  
- ポインタ演算の境界を意識する  
\*/