# 第3章 データ型と変数

## 対応C規格

* **主要対象:** C90
* **学習内容:** 基本データ型、変数宣言、初期化、スコープ、変数の生存期間

## 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：

* C言語の基本データ型を理解する
* 変数の宣言と初期化ができる
* 変数のスコープ（有効範囲）を理解する
* データ型のサイズと範囲を把握する
* 適切な型を選択してプログラムを作成できる

## 概要と詳細

### C言語の基本データ型

C言語では、さまざまなデータ型が用意されています。データ型とは、コンピュータに「どんな種類のデータを扱うか」を伝えるためのものです。

#### なぜデータ型が必要なのか？

現実世界で例えると： - **コップ**には水を入れるが、熱いコーヒーは入れられない - **財布**にはお金を入れるが、水は入れられない - **本棚**には本を置くが、液体は置けない

同じように、コンピュータでも「整数を入れる場所」「小数を入れる場所」「文字を入れる場所」を区別する必要があります。

#### 整数型（整数を扱う型）

整数型は、小数点を含まない数（1、2、-5など）を扱うための型です。

| 型名 | サイズ | 範囲 | 使いどころ |
| --- | --- | --- | --- |
| char | 1バイト | -128 〜 127 | 1文字、小さな数値 |
| unsigned char | 1バイト | 0 〜 255 | 負の数が不要な場合 |
| short | 2バイト | -32,768 〜 32,767 | 節約が必要な場合 |
| unsigned short | 2バイト | 0 〜 65,535 | 負の数が不要で節約したい場合 |
| int | 4バイト | 約-21億 〜 約21億 | 一般的な整数（最もよく使う） |
| unsigned int | 4バイト | 0 〜 約42億 | 負の数が不要な一般的な整数 |
| long | 4/8バイト | システム依存 | より大きな整数 |
| unsigned long | 4/8バイト | システム依存 | より大きな正の整数 |

**初心者のための型選択ガイド：** - 普通の整数なら int を使う - 文字を扱うなら char を使う - 負の数が不要なら unsigned を付ける - より大きな数が必要なら long を使う

#### 浮動小数点数型（小数を扱う型）

小数点を含む数（3.14、-0.5など）を扱うための型です。

| 型名 | サイズ | 精度 | 使用例 |
| --- | --- | --- | --- |
| float | 4バイト | 約7桁 | ゲームの座標、簡単な計算 |
| double | 8バイト | 約15桁 | 科学計算、金額計算 |
| long double | 12/16バイト | システム依存 | 超高精度計算 |

**初心者のための型選択ガイド：** - 通常は double を使う（精度が高い） - メモリを節約したい場合のみ float を使う - long double は特殊な用途のみ

#### その他の型

| 型名 | 説明 | 使用例 |
| --- | --- | --- |
| void | 「型なし」を表す | 値を返さない関数、汎用ポインタ |

#### 型のサイズについて

「バイト」はコンピュータのメモリの単位です： - 1バイト = 8ビット = 8個の0または1 - 1バイトで表現できる数：256通り（2の8乗） - 4バイトで表現できる数：約42億通り（2の32乗）

サイズが大きいほど、より大きな数や、より精密な小数を扱えます。

### 変数の宣言と初期化

変数は、データを入れる「箱」のようなものです。箱を使う前に、「どんな種類の箱か」「何という名前の箱か」を決める必要があります。これが変数の宣言です。

#### 変数とは何か？

変数を現実世界に例えると： - **引き出し**：それぞれに名前（ラベル）が付いていて、中に物を入れられる - **貯金箱**：お金（数値）を入れたり、取り出したりできる - **メモ帳**：情報を書いたり、書き換えたりできる

プログラムの中で、変数は値を記憶し、必要なときに取り出せる仕組みです。

#### 基本的な変数宣言

変数を使うには、まず「宣言」が必要です：

int age; /\* 年齢を入れる整数型の箱を用意 \*/  
double height; /\* 身長を入れる小数型の箱を用意 \*/  
char grade; /\* 成績（A、B、Cなど）を入れる文字型の箱を用意 \*/

宣言の構造：

データ型 変数名;

**変数名のルール：** 1. 英字（a-z、A-Z）、数字（0-9）、アンダースコア（\_）が使える 2. 数字から始めることはできない 3. C言語の予約語（int、if、forなど）は使えない 4. 大文字と小文字は区別される（ageとAgeは別の変数）

**良い変数名の例：**

int studentAge; /\* 学生の年齢 \*/  
double roomTemperature; /\* 室温 \*/  
char firstInitial; /\* 名前のイニシャル \*/

**悪い変数名の例：**

int 2ndPlace; /\* NG: 数字で始まっている \*/  
double my-score; /\* NG: ハイフンは使えない \*/  
char int; /\* NG: 予約語は使えない \*/

#### 初期化付き宣言

変数は宣言と同時に初期値を設定できます。これを「初期化」といいます：

int count = 10; /\* 最初から10が入った箱を用意 \*/  
double pi = 3.14159; /\* 円周率の値で初期化 \*/  
char letter = 'A'; /\* 文字'A'で初期化 \*/

**なぜ初期化が重要なのか？**

初期化しない変数には「ゴミ値」（予測できない値）が入っています：

int x; /\* 初期化なし：何が入っているか分からない \*/  
printf("%d\n", x); /\* 予測不能な値が表示される \*/  
  
int y = 0; /\* 初期化あり：確実に0 \*/  
printf("%d\n", y); /\* 0が表示される \*/

**推奨：変数は必ず初期化しましょう！**

#### 複数変数の同時宣言

同じ型の変数は、まとめて宣言できます：

/\* 3つの整数変数を一度に宣言 \*/  
int x, y, z;  
  
/\* 宣言と初期化を混ぜることも可能 \*/  
int a = 1, b = 2, c; /\* aは1、bは2、cは未初期化 \*/  
  
/\* より読みやすい書き方（推奨） \*/  
int width = 100;  
int height = 200;  
int depth = 50;

#### 変数の命名規則（ベストプラクティス）

良いプログラムは、変数名を見ただけで何を表すか分かります：

**1. 意味のある名前を使う**

/\* 悪い例 \*/  
int a, b, c;  
  
/\* 良い例 \*/  
int width, height, area;

**2. 命名スタイルを統一する**

/\* キャメルケース（推奨） \*/  
int studentAge;  
double averageScore;  
  
/\* スネークケース \*/  
int student\_age;  
double average\_score;

**3. 適切な長さにする**

/\* 短すぎる \*/  
int s; /\* sが何を表すか不明 \*/  
  
/\* ちょうど良い \*/  
int score;  
  
/\* 長すぎる \*/  
int theScoreOfTheStudentInTheMathematicsExam;

### 変数の代入と演算

#### 代入演算子

int number;  
number = 10; /\* 代入 \*/  
number += 5; /\* number = number + 5 と同じ \*/  
number -= 3; /\* number = number - 3 と同じ \*/  
number \*= 2; /\* number = number \* 2 と同じ \*/  
number /= 4; /\* number = number / 4 と同じ \*/

#### 基本演算子

int a = 10, b = 3;  
int sum = a + b; /\* 加算: 13 \*/  
int diff = a - b; /\* 減算: 7 \*/  
int product = a \* b; /\* 乗算: 30 \*/  
int quotient = a / b; /\* 除算: 3 (整数除算) \*/  
int remainder = a % b; /\* 剰余: 1 \*/

### 型変換（キャスト）

#### 暗黙の型変換

int i = 42;  
double d;  
d = i; /\* int から double への自動変換 \*/

#### 明示的な型変換

double pi = 3.14159;  
int rounded;  
rounded = (int)pi; /\* 明示的にintに変換（3になる） \*/

#### 型変換の注意点

int a = 5, b = 2;  
double result;  
  
result = a / b; /\* 結果: 2.0 (整数除算後に変換) \*/  
result = (double)a / b; /\* 結果: 2.5 (実数除算) \*/

### 変数のスコープ（有効範囲）

スコープとは、変数がプログラム中のどの部分でアクセス可能かを示す概念です。

#### 1. ファイルスコープ（グローバルスコープ）

ファイル全体でアクセス可能な変数です。

#include <stdio.h>  
  
int global\_count = 100; /\* ファイルスコープ変数 \*/  
double global\_rate; /\* 初期化されていない場合は0になる \*/  
  
void function(void)  
{  
 global\_count = 200; /\* 関数内からアクセス可能 \*/  
}  
  
int main(void)  
{  
 global\_count = 150; /\* main関数からもアクセス可能 \*/  
 function();  
 printf("global\_count = %d\n", global\_count); /\* 200が出力 \*/  
 return 0;  
}

#### 2. 関数スコープ

関数内で宣言された変数は、その関数内でのみアクセス可能です。

void function\_a(void)  
{  
 int local\_a = 10; /\* function\_a内でのみ有効 \*/  
 printf("local\_a = %d\n", local\_a);  
}  
  
void function\_b(void)  
{  
 int local\_a = 20; /\* 異なる関数なので同じ名前でも別変数 \*/  
 printf("local\_a = %d\n", local\_a);  
 /\* function\_aのlocal\_aにはアクセス不可 \*/  
}

#### 3. ブロックスコープ

{ と } で囲まれたブロック内でのみアクセス可能な変数です。

int main(void)  
{  
 int main\_var = 5;  
   
 if (main\_var > 0) {  
 int block\_var = 10; /\* このifブロック内でのみ有効 \*/  
 printf("block\_var = %d\n", block\_var);  
   
 {  
 int nested\_var = 20; /\* さらに内側のブロック \*/  
 printf("nested\_var = %d\n", nested\_var);  
 /\* block\_varとmain\_varにもアクセス可能 \*/  
 }  
 /\* nested\_varはここではアクセス不可 \*/  
 }  
 /\* block\_varはここではアクセス不可 \*/  
   
 return 0;  
}

#### 4. スコープの隠蔽（シャドウイング）

内側のスコープで同じ名前の変数を宣言すると、外側の変数が隠蔽されます。

int x = 100; /\* グローバル変数 \*/  
  
int main(void)  
{  
 printf("グローバルx = %d\n", x); /\* 100が出力 \*/  
   
 {  
 int x = 200; /\* ローカル変数がグローバル変数を隠蔽 \*/  
 printf("ローカルx = %d\n", x); /\* 200が出力 \*/  
   
 {  
 int x = 300; /\* さらに内側で隠蔽 \*/  
 printf("内側x = %d\n", x); /\* 300が出力 \*/  
 }  
   
 printf("ローカルx = %d\n", x); /\* 200が出力（復活） \*/  
 }  
   
 printf("グローバルx = %d\n", x); /\* 100が出力（復活） \*/  
 return 0;  
}

### 変数の生存期間（Storage Duration）

変数がメモリ上に存在する期間を生存期間といいます。

#### 1. 自動変数（Automatic Variables）

関数やブロックに入るときに作成され、出るときに破棄される変数です。

void demo\_function(void)  
{  
 int auto\_var = 5; /\* 自動変数（autoは省略可能） \*/  
 auto int explicit\_auto = 10; /\* 明示的にauto指定 \*/  
   
 printf("auto\_var = %d\n", auto\_var);  
 /\* 関数終了時にauto\_varは破棄される \*/  
}  
  
int main(void)  
{  
 demo\_function();  
 demo\_function(); /\* 毎回新しくauto\_varが作成される \*/  
 return 0;  
}

#### 2. 静的変数（Static Variables）

プログラム開始時に作成され、プログラム終了まで存在し続ける変数です。

void counter\_function(void)  
{  
 static int count = 0; /\* 静的ローカル変数、初期化は11回のみ \*/  
   
 count++;  
 printf("関数呼出回数: %d\n", count);  
}  
  
int main(void)  
{  
 counter\_function(); /\* 1が出力 \*/  
 counter\_function(); /\* 2が出力 \*/  
 counter\_function(); /\* 3が出力 \*/  
 return 0;  
}

#### 3. 静的グローバル変数

ファイル内でのみアクセス可能なグローバル変数です。

static int file\_private = 10; /\* このファイル内でのみアクセス可能 \*/  
int file\_public = 20; /\* 他のファイルからもアクセス可能 \*/  
  
static void private\_function(void) /\* このファイル内でのみ呼出可能 \*/  
{  
 printf("file\_private = %d\n", file\_private);  
}  
  
int main(void)  
{  
 private\_function();  
 return 0;  
}

#### 4. レジスタ変数（Register Variables）

可能であればレジスタに格納される変数です（C90では推奨のみ）。

int main(void)  
{  
 register int fast\_var = 100; /\* レジスタ格納を要求 \*/  
   
 /\* レジスタ変数のアドレスは取得できない \*/  
 /\* int \*ptr = &fast\_var; <- エラー \*/  
   
 return 0;  
}

### ストレージクラス指定子まとめ

| 指定子 | スコープ | 生存期間 | 初期化 | 説明 |
| --- | --- | --- | --- | --- |
| auto | ブロック | 自動 | 未定義値 | デフォルト（省略可能） |
| static（ローカル） | ブロック | 静的 | 0 | 関数呼出間で値保持 |
| static（グローバル） | ファイル | 静的 | 0 | ファイル内限定 |
| extern | グローバル | 静的 | 他で定義 | 他ファイルの変数参照 |
| register | ブロック | 自動 | 未定義値 | レジスタ格納要求 |

### 変数の初期化規則

#### 自動変数の初期化

int main(void)  
{  
 int uninitialized; /\* 未定義値（危険） \*/  
 int initialized = 0; /\* 明示的初期化（推奨） \*/  
   
 printf("uninitialized = %d\n", uninitialized); /\* 予測不能 \*/  
 printf("initialized = %d\n", initialized); /\* 0 \*/  
   
 return 0;  
}

#### 静的変数の初期化

int global\_int; /\* 自動的に0で初期化 \*/  
static int static\_int; /\* 自動的に0で初期化 \*/  
  
int main(void)  
{  
 static int local\_static; /\* 自動的に0で初期化 \*/  
   
 printf("global\_int = %d\n", global\_int); /\* 0 \*/  
 printf("static\_int = %d\n", static\_int); /\* 0 \*/  
 printf("local\_static = %d\n", local\_static); /\* 0 \*/  
   
 return 0;  
}

### 実践的なスコープ活用例

#### カウンター関数の実装

int get\_next\_id(void)  
{  
 static int id\_counter = 1000; /\* 初期値 \*/  
   
 return id\_counter++; /\* 現在値を返してからインクリメント \*/  
}  
  
int main(void)  
{  
 printf("ID: %d\n", get\_next\_id()); /\* 1000 \*/  
 printf("ID: %d\n", get\_next\_id()); /\* 1001 \*/  
 printf("ID: %d\n", get\_next\_id()); /\* 1002 \*/  
   
 return 0;  
}

#### モジュール内状態管理

/\* 設定管理モジュールの例 \*/  
static int debug\_mode = 0; /\* モジュール内部状態 \*/  
static int max\_connections = 10;  
  
void set\_debug\_mode(int mode)  
{  
 debug\_mode = mode;  
}  
  
int is\_debug\_enabled(void)  
{  
 return debug\_mode;  
}  
  
void set\_max\_connections(int max)  
{  
 if (max > 0 && max <= 100) {  
 max\_connections = max;  
 }  
}  
  
int get\_max\_connections(void)  
{  
 return max\_connections;  
}

### 定数の定義

#### constキーワード

const int MAX\_SIZE = 100; /\* 定数の定義 \*/  
const double PI = 3.14159; /\* 浮動小数点定数 \*/

#### #defineプリプロセッサ

#define MAX\_STUDENTS 50 /\* マクロ定数 \*/  
#define TITLE "C Language Tutorial"

### データ型のサイズ確認

sizeof演算子を使ってデータ型のサイズを確認できます：

#include <stdio.h>  
  
int main(void)  
{  
 printf("char: %lu バイト\n", sizeof(char));  
 printf("int: %lu バイト\n", sizeof(int));  
 printf("double: %lu バイト\n", sizeof(double));  
 return 0;

## サンプルコード

### 基本データ型の使用例

**プログラムファイル:** <examples/data_types_demo.c> | [C99版](examples/data_types_demo_c99.c)

各データ型の宣言、初期化、出力を学習します。

/\*  
 \* ファイル名: data\_types\_demo.c  
 \* 説明: C言語の基本データ型の使用例  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void)  
{  
 /\* 整数型の変数宣言と初期化 \*/  
 char letter = 'A';  
 short small\_num = 1000;  
 int age = 25;  
 long big\_num = 1234567L;  
 unsigned int positive = 100U;  
  
 /\* 浮動小数点型の変数宣言と初期化 \*/  
 float pi\_f = 3.14f;  
 double pi\_d = 3.141592653589793;  
  
 printf("=== データ型のサイズと値 ===\n");  
  
 /\* 各データ型のサイズを表示 \*/  
 printf("char : %lu バイト, 値: %c (%d)\n",  
 sizeof(char), letter, letter);  
 printf("short : %lu バイト, 値: %d\n",  
 sizeof(short), small\_num);  
 printf("int : %lu バイト, 値: %d\n",  
 sizeof(int), age);  
 printf("long : %lu バイト, 値: %ld\n",  
 sizeof(long), big\_num);  
 printf("unsigned : %lu バイト, 値: %u\n",  
 sizeof(unsigned int), positive);  
 printf("float : %lu バイト, 値: %.6f\n",  
 sizeof(float), pi\_f);  
 printf("double : %lu バイト, 値: %.15f\n",  
 sizeof(double), pi\_d);  
  
 /\* 演算の例 \*/  
 printf("\n=== 基本演算 ===\n");  
 printf("10 + 3 = %d\n", 10 + 3);  
 printf("10 - 3 = %d\n", 10 - 3);  
 printf("10 \* 3 = %d\n", 10 \* 3);  
 printf("10 / 3 = %d (整数除算)\n", 10 / 3);  
 printf("10 %% 3 = %d (剰余)\n", 10 % 3);  
 printf("10.0 / 3.0 = %.2f (実数除算)\n", 10.0 / 3.0);  
  
 return 0;  
}

### 数値計算の応用例

**プログラムファイル:** <solutions/numerical_calc.c> | [C99版](solutions/numerical_calc_c99.c)

math.hライブラリを使用した高度な数値計算を学習します。

### 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 data\_types\_demo.c -o data\_types\_demo  
  
# 実行  
./data\_types\_demo

## 演習課題

### 基礎問題

1. **変数宣言と初期化**
   * さまざまなデータ型の変数を宣言し、初期化してください
   * 各変数の値をprintf関数で出力してください
2. **四則演算計算機**
   * 2つの数値を変数に格納し、四則演算の結果を表示するプログラムを作成してください
3. **データ型サイズの確認**
   * sizeof演算子を使って、各データ型のサイズを表示するプログラムを作成してください

### 応用問題

1. **型変換の理解**
   * 整数除算と実数除算の違いを確認するプログラムを作成してください
2. **スコープの実験**
   * グローバル変数とローカル変数を使い分けるプログラムを作成してください
3. **定数の活用**
   * constと#defineを使って定数を定義し、計算に使用してください

## 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 data\_types\_demo  
  
# 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

* 変数宣言は関数やブロックの先頭で行う必要がある
* long long型は使用できない
* \_Bool型は使用できない

### C99以降の拡張

* long long型の追加（64ビット整数）
* \_Bool型の追加（論理型）
* 複素数型（\_Complex）の追加
* 変数宣言位置の制限緩和

### 型サイズの注意点

データ型のサイズはシステムに依存するため、portable（移植可能）なプログラムを書く際は注意が必要です：

/\* 推奨: 具体的なサイズが必要な場合 \*/  
#include <stdint.h>  
int32\_t exact\_32bit; /\* C99以降 \*/

## よくある間違い

### 1. 初期化忘れ

/\* NG: 初期化せずに使用 \*/  
int count;  
printf("%d\n", count); /\* 未定義値が出力される \*/  
  
/\* OK: 初期化してから使用 \*/  
int count = 0;  
printf("%d\n", count);

### 2. 整数除算の結果

/\* NG: 期待した結果にならない \*/  
int result = 5 / 2; /\* 結果: 2 \*/  
  
/\* OK: 実数除算を使用 \*/  
double result = 5.0 / 2.0; /\* 結果: 2.5 \*/

### 3. 型の範囲超過

/\* NG: charの範囲を超える \*/  
char big\_num = 300; /\* オーバーフロー \*/  
  
/\* OK: 適切な型を使用 \*/  
int big\_num = 300;

## 次の章へ

データ型と変数を理解したら、[演算子と式](../operators/README.md) に進んでください。

## 参考資料

* [C言語データ型リファレンス](https://ja.cppreference.com/w/c/language/type)
* [sizeof演算子](https://ja.cppreference.com/w/c/language/sizeof)
* [型変換](https://ja.cppreference.com/w/c/language/conversion)

# サンプルコード

## data\_types\_demo.c

/\*  
 \* ファイル名: data\_types\_demo.c  
 \* 説明: C言語の基本データ型の使用例  
 \* 規格: C90準拠  
 \*/  
  
#include <stdio.h>  
  
int main(void)  
{  
 /\* 整数型の変数宣言と初期化 \*/  
 char letter = 'A';  
 short small\_num = 1000;  
 int age = 25;  
 long big\_num = 1234567L;  
 unsigned int positive = 100U;  
  
 /\* 浮動小数点型の変数宣言と初期化 \*/  
 float pi\_f = 3.14f;  
 double pi\_d = 3.141592653589793;  
  
 printf("=== データ型のサイズと値 ===\n");  
  
 /\* 各データ型のサイズを表示 \*/  
 printf("char : %lu バイト, 値: %c (%d)\n",  
 sizeof(char), letter, letter);  
 printf("short : %lu バイト, 値: %d\n",  
 sizeof(short), small\_num);  
 printf("int : %lu バイト, 値: %d\n",  
 sizeof(int), age);  
 printf("long : %lu バイト, 値: %ld\n",  
 sizeof(long), big\_num);  
 printf("unsigned : %lu バイト, 値: %u\n",  
 sizeof(unsigned int), positive);  
 printf("float : %lu バイト, 値: %.6f\n",  
 sizeof(float), pi\_f);  
 printf("double : %lu バイト, 値: %.15f\n",  
 sizeof(double), pi\_d);  
  
 /\* 演算の例 \*/  
 printf("\n=== 基本演算 ===\n");  
 printf("10 + 3 = %d\n", 10 + 3);  
 printf("10 - 3 = %d\n", 10 - 3);  
 printf("10 \* 3 = %d\n", 10 \* 3);  
 printf("10 / 3 = %d (整数除算)\n", 10 / 3);  
 printf("10 %% 3 = %d (剰余)\n", 10 % 3);  
 printf("10.0 / 3.0 = %.2f (実数除算)\n", 10.0 / 3.0);  
  
 return 0;  
}