C言語プログラミング教材 - メインテキスト（基礎編）

第1章〜第6章：プログラミングの基礎を学ぶ

C言語学習教材プロジェクト

2024年

Table of Contents

[はじめに 7](#_Toc202361692)

[本書の構成 7](#_Toc202361693)

[学習の進め方 7](#_Toc202361694)

[C言語入門 8](#_Toc202361695)

[対応C規格 8](#_Toc202361696)

[学習目標 8](#_Toc202361697)

[概要と詳細 8](#_Toc202361698)

[n C言語とは 8](#_Toc202361699)

[n C言語の規格 10](#_Toc202361700)

[n 開発環境 12](#_Toc202361701)

[n コンパイル手順 14](#_Toc202361702)

[コンパイル方法 15](#_Toc202361703)

[規格による違い 15](#_Toc202361704)

[次の章へ 16](#_Toc202361705)

[参考資料 16](#_Toc202361706)

[基本構文 17](#_Toc202361707)

[対応C規格 17](#_Toc202361708)

[学習目標 17](#_Toc202361709)

[概要と詳細 17](#_Toc202361710)

[n はじめてのC言語プログラム 17](#_Toc202361711)

[n プログラムの基本構造 17](#_Toc202361712)

[n main関数の詳細 19](#_Toc202361713)

[n printf関数 20](#_Toc202361714)

[n コメント 23](#_Toc202361715)

[n 基本的なプログラム例 24](#_Toc202361716)

[コンパイル方法 26](#_Toc202361717)

[規格による違い 27](#_Toc202361718)

[n C90での制限事項 27](#_Toc202361719)

[n C99以降の拡張 27](#_Toc202361720)

[次の章へ 27](#_Toc202361721)

[参考資料 27](#_Toc202361722)

[データ型と変数 28](#_Toc202361723)

[対応C規格 28](#_Toc202361724)

[学習目標 28](#_Toc202361725)

[概要と詳細 28](#_Toc202361726)

[n データ型と変数の世界へようこそ 28](#_Toc202361727)

[n C言語の基本データ型 28](#_Toc202361728)

[n 変数の宣言と初期化 30](#_Toc202361729)

[n 変数の代入と演算 33](#_Toc202361730)

[n 型変換（キャスト） 33](#_Toc202361731)

[n 変数のスコープ（有効範囲） 34](#_Toc202361732)

[n 変数の生存期間（Storage Duration） 36](#_Toc202361733)

[n ストレージクラス指定子まとめ 39](#_Toc202361734)

[n 変数の初期化規則 40](#_Toc202361735)

[n 実践的なスコープ活用例 40](#_Toc202361736)

[n 定数の定義 41](#_Toc202361737)

[n データ型のサイズ確認 44](#_Toc202361738)

[コンパイル方法 45](#_Toc202361739)

[規格による違い 45](#_Toc202361740)

[n C90での制限事項 45](#_Toc202361741)

[n C99以降の拡張 45](#_Toc202361742)

[n 型サイズの注意点 45](#_Toc202361743)

[よくある間違い 46](#_Toc202361744)

[n 1. 初期化忘れ 46](#_Toc202361745)

[n 2. 整数除算の結果 46](#_Toc202361746)

[n 3. 型の範囲超過 46](#_Toc202361747)

[次の章へ 46](#_Toc202361748)

[参考資料 46](#_Toc202361749)

[演算子 47](#_Toc202361750)

[対応C規格 47](#_Toc202361751)

[学習目標 47](#_Toc202361752)

[概要と詳細 47](#_Toc202361753)

[n 演算子の世界へようこそ 47](#_Toc202361754)

[n 算術演算子（計算の基本） 47](#_Toc202361755)

[n 代入演算子（値を保存する） 49](#_Toc202361756)

[n インクリメント・デクリメント演算子（1つずつ増減） 50](#_Toc202361757)

[n 関係演算子（比較する） 53](#_Toc202361758)

[n 論理演算子（条件を組み合わせる） 54](#_Toc202361759)

[n 条件演算子（三項演算子） 61](#_Toc202361760)

[n sizeof演算子 61](#_Toc202361761)

[n 演算子の優先順位 61](#_Toc202361762)

[コンパイル方法 62](#_Toc202361763)

[規格による違い 62](#_Toc202361764)

[n C90での制限事項 62](#_Toc202361765)

[n C99以降の拡張 63](#_Toc202361766)

[よくある間違い 63](#_Toc202361767)

[n 1. 代入と比較の混同 63](#_Toc202361768)

[n 2. 整数除算の結果 63](#_Toc202361769)

[n 3. 演算子の優先順位 63](#_Toc202361770)

[n 4. ビット演算の優先順位 63](#_Toc202361771)

[次の章へ 64](#_Toc202361772)

[参考資料 64](#_Toc202361773)

[条件分岐 65](#_Toc202361774)

[対応C規格 65](#_Toc202361775)

[学習目標 65](#_Toc202361776)

[概要と詳細 65](#_Toc202361777)

[n 条件分岐の世界へようこそ 65](#_Toc202361778)

[n なぜ条件分岐が必要なのか？ 65](#_Toc202361779)

[n if文の基本（もし〜なら） 66](#_Toc202361780)

[n if-else文（そうでなければ） 66](#_Toc202361781)

[n if-else if文（複数の選択肢） 68](#_Toc202361782)

[n ネストしたif文（入れ子構造） 69](#_Toc202361783)

[n 条件式の詳細 70](#_Toc202361784)

[n switch文 71](#_Toc202361785)

[n 条件演算子（三項演算子）の復習 74](#_Toc202361786)

[n 条件式での注意点 74](#_Toc202361787)

[コンパイル方法 75](#_Toc202361788)

[規格による違い 75](#_Toc202361789)

[n C90での制限事項 75](#_Toc202361790)

[n C99以降の拡張 75](#_Toc202361791)

[よくある間違い 76](#_Toc202361792)

[n 1. セミコロンの位置 76](#_Toc202361793)

[n 1. switch文でのbreak忘れ 76](#_Toc202361794)

[n 1. 複雑な条件式での括弧不足 76](#_Toc202361795)

[次の章へ 77](#_Toc202361796)

[参考資料 77](#_Toc202361797)

[繰り返し処理 78](#_Toc202361798)

[対応C規格 78](#_Toc202361799)

[学習目標 78](#_Toc202361800)

[概要と詳細 78](#_Toc202361801)

[n 繰り返し処理とは？ 78](#_Toc202361802)

[n for文 78](#_Toc202361803)

[n while文 81](#_Toc202361804)

[n do-while文 82](#_Toc202361805)

[n break文とcontinue文 83](#_Toc202361806)

[n ネストしたループ 84](#_Toc202361807)

[n ループでのbreak・continueの応用 85](#_Toc202361808)

[n 無限ループ ♾ 86](#_Toc202361809)

[n ループの最適化とベストプラクティス 88](#_Toc202361810)

[コンパイル方法 88](#_Toc202361811)

[規格による違い 89](#_Toc202361812)

[n C90での制限事項 89](#_Toc202361813)

[n C99以降の拡張 89](#_Toc202361814)

[よくある間違い 89](#_Toc202361815)

[n 1. 無限ループの作成 89](#_Toc202361816)

[n 2. セミコロンの位置 89](#_Toc202361817)

[n 3. ループ変数の範囲外アクセス 89](#_Toc202361818)

[次の章へ 90](#_Toc202361819)

[参考資料 90](#_Toc202361820)

[終わりに 90](#_Toc202361821)

[次のステップ 90](#_Toc202361822)

# はじめに

本書は、C言語プログラミング教材の基礎編です。プログラミング初心者を対象に、C言語の基本的な概念と文法を段階的に学習できるよう構成されています。

## 本書の構成

基礎編では以下の内容を扱います：

* 第1章: 導入・環境構築
* 第2章: 基本文法・Hello World
* 第3章: データ型と変数
* 第4章: 演算子
* 第5章: 制御構造（条件分岐）
* 第6章: 制御構造（ループ）

## 学習の進め方

各章は以下の構成になっています：

1. **概念の説明**: 学習する内容の理論的な説明
2. **サンプルコード**: 実際に動作するコード例
3. **演習問題**: 理解を深めるための練習問題

サンプルコードは必ず自分で入力して実行し、動作を確認しながら学習を進めてください。

# C言語入門

## 対応C規格

* **主要対象:** 全規格共通
* **学習内容:** C言語の基礎知識、開発環境構築、規格解説

## 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言語の特徴と歴史を理解する
* 開発環境を構築し、基本的なコンパイル手順を覚える
* C言語の規格（C90, C99, C11, C17）の違いを理解する
* 簡単なプログラムをコンパイル・実行できる

## 概要と詳細

### C言語とは

C言語は1972年にデニス・リッチーがベル研究所で開発したプログラミング言語です。現在でも世界中で広く使われている、とても重要なプログラミング言語の一つです。

#### なぜC言語を学ぶのか？

初めてプログラミングを学ぶ方は「なぜC言語から始めるの？」と思うかもしれません。実は、C言語を学ぶことには大きなメリットがあります。

1. **コンピュータの仕組みが理解できる**
   * C言語はコンピュータのハードウェアに近い言語です
   * メモリやCPUがどのように動作するか実感できます
   * 他の言語を学ぶときの基礎知識になります
2. **多くのプログラミング言語の元になっている**
   * C++、Java、C#などはC言語の影響を受けています
   * C言語の文法を知っていれば、他の言語も学びやすくなります
3. **今でも現役で使われている**
   * 50年以上の歴史がありますが、今でも重要な場面で使われています
   * 特に高速性や効率性が求められる場面では欠かせません

#### 主な特徴

C言語には以下のような特徴があります。初心者には少し難しく感じるかもしれませんが、一つずつ理解していけば大丈夫です。

* **システムプログラミング向け**:
  + OS（オペレーティングシステム）の開発に使われます
  + ハードウェアを直接制御できます
  + 例：LinuxカーネルはC言語で書かれています
* **高い移植性**:
  + 一度書いたプログラムが様々なコンピュータで動きます
  + Windows、Mac、Linuxなど、異なるOSでも同じコードが使えます
  + これを「Write Once, Run Anywhere（一度書けば、どこでも動く）」と言います
* **効率性**:
  + 実行速度が非常に速い
  + メモリの使用量を最小限に抑えられる
  + 限られたリソースで動作させる必要がある場面で重宝されます
* **シンプルな文法**:
  + 言語の機能が基本的なものに絞られています
  + 覚えることが比較的少ない
  + ただし、その分プログラマーが考えることは多くなります

#### 使用分野

C言語は以下のような分野で活躍しています。

* **オペレーティングシステム（OS）**
  + Linux：世界中のサーバーで使われているOS
  + Windows：一部のコア部分
  + macOS：カーネル部分
* **組込みシステム**
  + 家電製品（エアコン、洗濯機、電子レンジなど）
  + 自動車の制御システム
  + IoTデバイス（スマートウォッチ、センサーなど）
* **データベースシステム**
  + MySQL、PostgreSQLなどの有名なデータベース
  + 高速なデータ処理が必要なため
* **ゲームエンジン**
  + ゲームの基盤となる部分
  + 3Dグラフィックスの処理など
* **科学技術計算**
  + 物理シミュレーション
  + 数値解析
  + 機械学習の基礎ライブラリ

### C言語の規格

C言語は時代とともに進化し、複数の規格（バージョン）が策定されています。プログラミング言語にも「バージョン」があることを不思議に思うかもしれませんが、これは言語をより使いやすく、より強力にするための改良の歴史です。

#### C言語規格の歴史

| 規格名 | 発表年 | 正式名称 | 主な特徴 |
| --- | --- | --- | --- |
| **C90** | 1990年 | ISO/IEC 9899:1990 | 初の国際標準、ANSI C |
| **C99** | 1999年 | ISO/IEC 9899:1999 | \_Bool型、可変長配列、inline関数 |
| **C11** | 2011年 | ISO/IEC 9899:2011 | 匿名構造体、\_Generic、マルチスレッド |
| **C17** | 2018年 | ISO/IEC 9899:2018 | C11のバグ修正版 |

#### 各規格の詳しい説明

**C90（ANSI C）**

* C言語の最初の国際標準規格です
* 最も基本的で、すべてのC言語コンパイラが対応しています
* シンプルで学習しやすく、初心者にお勧めです
* 古い組込みシステムでも確実に動作します

**C99**

* 多くの便利な機能が追加されました
* 例：forループの中で変数を宣言できるようになりました
* for (int i = 0; i < 10; i++) { /\* C99から可能 \*/ }
* 現在最も広く使われている規格です

**C11**

* マルチスレッド（並列処理）のサポートが追加されました
* より安全なプログラミングのための機能が強化されました
* 比較的新しいため、すべての環境で使えるとは限りません

**C17**

* C11の細かいバグを修正したマイナーアップデート
* 新機能の追加はほとんどありません

#### どの規格を選ぶべきか？

初学者の方への推奨。

* **学習目的なら**: **C90から始める**
  + 最も基本的な機能を確実に習得できます
  + どこでも動く汎用的なコードが書けるようになります
  + 段階的に新しい機能を学んでいけます
* **実用的なプログラムを書くなら**: **C99を使う**
  + 便利な機能が使えて、生産性が向上します
  + ほとんどの環境で問題なく動作します
  + 現在の業界標準と言えます
* **組込みシステムを作るなら**: **C90またはC99**
  + 使用するマイコンやコンパイラの対応状況を確認してください
  + 古い環境ではC90しか使えない場合があります

#### 規格による違いの例

/\* C90スタイル \*/  
int main(void) {  
 int i; /\* 変数宣言は関数の先頭で \*/  
   
 for (i = 0; i < 10; i++) {  
 printf("%d\n", i);  
 }  
 return 0;  
}  
  
/\* C99以降のスタイル \*/  
int main(void) {  
 for (int i = 0; i < 10; i++) { /\* forループ内で宣言可能 \*/  
 printf("%d\n", i);  
 }  
 return 0;  
}

この教材では、基本的にC90準拠のコードを示し、必要に応じてC99版も提供します。これにより、どんな環境でも動作するコードの書き方を学びながら、新しい機能も理解できるようになっています。

### 開発環境

C言語でプログラミングを始めるには、いくつかのツール（ソフトウェア）が必要です。最初は難しく感じるかもしれませんが、一度セットアップすればずっと使えます。

#### 必須ツール

プログラミングには以下の4つのツールが必要です。

1. **テキストエディター**: プログラムを書くためのソフト
   * メモ帳でも書けますが、専用エディターが便利です
   * 推奨：Visual Studio Code（無料）、Sublime Text、Atom
   * プログラミング用エディターは色分け表示などの便利機能があります
2. **コンパイラ**: 人間が書いたコードをコンピュータが理解できる形に変換
   * C言語で書いたテキストを実行可能なプログラムに変換します
   * 主なコンパイラ：GCC（無料）、Clang（無料）、Visual C++（Windows）
   * この教材ではGCCを使用します
3. **リンカー**: プログラムの部品を組み合わせる
   * 通常はコンパイラに含まれているので、別途インストールは不要です
   * 複数のファイルからなるプログラムを一つにまとめます
4. **デバッガー**: プログラムの誤り（バグ）を見つけるツール
   * プログラムを一行ずつ実行して動作を確認できます
   * GDB（GCCに付属）が一般的です

#### OS別の環境構築手順

お使いのOSに応じて、以下の手順で環境を構築してください。

**Windows:** 最も簡単な方法はMinGW-w64をインストールすることです。

1. MinGW-w64のインストーラーをダウンロード
2. インストール時の設定。
   * Architecture: x86\_64（64ビット版）
   * Threads: posix
   * Exception: seh
3. 環境変数PATHにMinGWのbinフォルダを追加
4. コマンドプロンプトで gcc --version と入力して確認 または、Microsoft Visual Studio Community（無料）も使えます。

* C++の開発環境をインストール
* 統合開発環境（IDE）なので初心者に優しい

**macOS:** macOSは開発者に優しいOSです。

# ターミナルを開いて以下のコマンドを実行  
xcode-select --install

これだけで、GCCとその他の開発ツールがインストールされます。

**Linux（Ubuntu/Debian）:** Linuxは最初から開発環境が整っていることが多いです。

# パッケージリストを更新  
sudo apt update  
# 開発ツール一式をインストール  
sudo apt install build-essential  
# 確認  
gcc --version

#### 環境構築の確認

インストールが完了したら、以下のコマンドで確認しましょう。

# コンパイラのバージョン確認  
gcc --version  
# 簡単なプログラムを作成してテスト  
echo 'int main(void) { return 0; }' > test.c  
gcc test.c -o test  
./test # Windowsの場合は test.exe  
echo $? # 0 が表示されれば成功

### コンパイル手順

C言語のプログラムが実行されるまでの流れを理解しましょう。料理に例えると。

1. **ソースコード作成**（レシピを書く）
   * 人間が理解できるC言語でプログラムを書きます
   * ファイルの拡張子は .c です（例：hello.c）
2. **プリプロセッサ処理**（材料の下準備）
   * #include で指定したファイルを取り込みます
   * #define で定義した値を置き換えます
   * コンパイラが処理する前の準備作業です
3. **コンパイル**（調理）
   * C言語のコードをアセンブリ言語に変換します
   * 文法エラーがあればここで検出されます
   * まだ実行はできません
4. **アセンブル**（盛り付け）
   * アセンブリ言語を機械語（0と1）に変換します
   * オブジェクトファイル（.o または .obj）が作成されます
5. **リンク**（完成）
   * 必要なライブラリと結合します
   * 実行可能ファイルが生成されます
   * Windowsでは .exe ファイル、Unix系では拡張子なし
6. **実行**（食べる）
   * 作成したプログラムを実行します

#### 基本的なコンパイルコマンド

実際のコンパイル方法を見てみましょう。

# シンプルな方法（すべての処理を一度に）  
gcc hello.c -o hello  
# hello.c をコンパイルして hello という実行ファイルを作成  
# 実行  
./hello # Unix/Linux/macOS  
hello # Windows  
# より詳細なオプション付き  
gcc -Wall -Wextra -o hello hello.c  
# -Wall: すべての警告を表示  
# -Wextra: 追加の警告も表示  
# -o hello: 出力ファイル名を指定  
# 段階的なコンパイル（大きなプログラムで使用）  
gcc -c hello.c # hello.o を作成  
gcc -c another.c # another.o を作成  
gcc hello.o another.o -o program # リンクして実行ファイル作成

#### よくあるエラーと対処法

初心者がよく遭遇するエラー。

1. **“gcc: command not found”**
   * コンパイラがインストールされていません
   * PATHが設定されていません
2. **“hello.c: No such file or directory”**
   * ファイル名が間違っています
   * 現在のディレクトリにファイルがありません
3. **“undefined reference to `main’”**
   * main関数が定義されていません
   * C言語のプログラムには必ずmain関数が必要です

## 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  
# C90準拠でコンパイル  
make STANDARD=c90  
# C99準拠でコンパイル   
make STANDARD=c99  
# C11準拠でコンパイル  
make STANDARD=c11  
# C17準拠でコンパイル  
make STANDARD=c17  
# クリーンアップ  
make clean

## 規格による違い

この章では基本的な環境確認のため、規格による大きな違いはありません。ただし、コンパイラの警告やエラーメッセージに違いが出る場合があります。

## 次の章へ

環境構築が完了したら、[基本文法・Hello World](../basics-syntax/README.md) に進んでください。

## 参考資料

* [GCC公式ドキュメント](https://gcc.gnu.org/documentation.html)
* [C言語規格書 (ISO/IEC 9899:2018)](https://www.iso.org/standard/74528.html)
* [GNU Make マニュアル](https://www.gnu.org/software/make/manual/)

# 基本構文

## 対応C規格

* **主要対象:** C90
* **学習内容:** プログラムの基本構造、main関数、printf関数

## 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言語プログラムの基本構造を理解する
* main関数の役割を理解する
* printf関数を使った文字列出力ができる
* コメントの書き方を覚える
* 基本的なエスケープシーケンスを使える

## 概要と詳細

### はじめてのC言語プログラム

プログラミングの学習は「Hello, World!」と画面に表示するプログラムから始まるのが伝統です。この章では、最初のC言語プログラムを作成し、その構造を詳しく学びます。

### プログラムの基本構造

まず、最もシンプルなC言語プログラムを見てみましょう。

/\*  
 \* ファイル名: hello\_world.c  
 \* 説明: C言語の最も基本的なプログラム  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 printf("Hello, World!\n");  
 return 0;  
}

このたった6行のプログラムにも、C言語の重要な要素がすべて含まれています。料理のレシピのように、各行には重要な役割があります。

#### 各部分の詳しい説明

1. **プリプロセッサ指令** (#include <stdio.h>)
   * プログラムの「準備」をする行です
   * stdio.h（Standard Input/Output Header）は「標準入出力ヘッダー」という意味
   * printfのような入出力関数を使うために必要です
   * 例えるなら、料理を始める前に必要な道具を準備するようなものです
2. **main関数** (int main(void))
   * C言語プログラムの「心臓部」です
   * プログラムは必ずここから実行が始まります
   * intは「整数を返す」という意味
   * voidは「引数を受け取らない」という意味
   * どんなに大きなプログラムでも、main関数は1つだけです
3. **関数本体**（{ と } で囲まれた部分）
   * 実際の処理を書く場所です
   * {（開き波括弧）で始まり、}（閉じ波括弧）で終わります
   * この中に書かれた命令が順番に実行されます
   * インデント（字下げ）をして読みやすくするのが慣習です
4. **処理文** (printf("Hello, World!\n");)
   * 実際に何かをする命令です
   * printfは画面に文字を表示する関数
   * 文の最後には必ず;（セミコロン）を付けます
   * これを忘れるとコンパイルエラーになります
5. **return文** (return 0;)
   * プログラムの終了を示します
   * 0は「正常終了」を意味する特別な数字
   * エラーが発生した場合は0以外の数字を返すことがあります

#### なぜこの構造なのか？

初心者の方は「なぜこんなに複雑なの？」と思うかもしれません。実は、この構造には理由があります。

* **#include**: 車輪の再発明を避けるため
  + すでに作られた便利な機能を使えるようにします
  + printf関数も誰かが作ってくれた機能です
* **main関数**: プログラムの開始点を明確にするため
  + コンピュータがどこから実行すればよいか迷わないようにします
* **波括弧{}**: プログラムの範囲を明確にするため
  + どこからどこまでがmain関数なのかを示します
* **セミコロン;**: 文の終わりを明確にするため
  + 日本語の「。」のような役割です

### main関数の詳細

main関数はプログラムの**エントリーポイント**（実行開始点）です。野球で言えばピッチャーマウンド、演劇で言えば主役が最初に登場する場所のようなものです。

#### main関数の2つの形式

C言語のmain関数には主に2つの書き方があります。 **1. 基本形式（引数なし）:**

int main(void)  
{  
 printf("Hello from main!\n");  
 return 0;  
}

この形式は最もシンプルで、初心者の学習に適しています。voidは「何も受け取らない」という意味です。 **2. コマンドライン引数を受け取る形式:**

int main(int argc, char \*argv[])  
{  
 printf("プログラム名: %s\n", argv[0]);  
 if (argc > 1) {  
 printf("引数1: %s\n", argv[1]);  
 }  
 return 0;  
}

この形式は、プログラム実行時に追加情報を受け取りたい場合に使います。例えば。

./myprogram file.txt -v

このように実行すると、file.txtや-vという情報をプログラム内で使えます。 （詳しくは後の章で学習します）

#### 戻り値の意味と重要性

main関数のreturnは、プログラムがどのように終了したかをOSに伝える重要な役割があります。

* **0**: 「すべて正常に終わりました」という合図
  + 成功を示す世界共通の値です
  + プログラムが期待通りに動作したことを意味します
* **0以外**: 「何か問題がありました」という合図
  + 1, 2, 3…など、異なる数字で異なるエラーを表現できます
  + 例：1 = ファイルが見つからない、2 = メモリ不足など

#### なぜ戻り値が重要なのか？

プログラムは単独で動くだけでなく、他のプログラムと連携することがあります。

# プログラムAが成功したらプログラムBを実行  
./programA && ./programB  
# プログラムCが失敗したらエラーメッセージを表示  
./programC || echo "エラーが発生しました"

このような連携を可能にするのが、main関数の戻り値です。

#### よくある疑問

* **Q: なぜintなの？**
* A: 戻り値として整数（integer）を返すためです。成功/失敗を数字で表現します。
* **Q: void main()と書いている例を見たけど？**
* A: 古い書き方で、現在は推奨されません。必ずint main()を使いましょう。
* **Q: return 0;を忘れたらどうなる？**
* A: C99以降では自動的に0が返されますが、明示的に書くことを推奨します。

### printf関数

printf関数は画面に文字を表示するための関数です。「print formatted（フォーマット付き出力）」の略で、単純な文字列だけでなく、数値や変数の値も表示できる強力な機能です。

#### 基本的な使い方

最もシンプルな使い方から始めましょう。

printf("Hello, World!\n");

この1行で「Hello, World!」と表示して改行します。\nは改行を意味する特殊な記号です。

#### なぜprintf関数を使うのか？

コンピュータとの対話には出力が欠かせません。

* 計算結果を確認したい
* プログラムの動作状況を知りたい
* エラーメッセージを表示したい これらすべてにprintf関数を使います。

#### フォーマット指定子（変数の値を表示する魔法）

printf関数の真の力は、変数の値を表示できることです。%記号を使った「フォーマット指定子」がその鍵です。

| 指定子 | データ型 | 説明 | 例 |
| --- | --- | --- | --- |
| %d | int | 整数（decimal） | printf("年齢は%d歳です", 20); → 年齢は20歳です |
| %c | char | 1文字（character） | printf("成績は%cです", 'A'); → 成績はAです |
| %s | char\* | 文字列（string） | printf("名前は%sです", "太郎"); → 名前は太郎です |
| %f | double | 小数（float） | printf("円周率は%fです", 3.14); → 円周率は3.140000です |

#### 複数の値を同時に表示

printf関数では、複数の値を一度に表示できます。

int age = 20;  
double height = 170.5;  
printf("年齢: %d歳、身長: %.1fcm\n", age, height);  
/\* 出力: 年齢: 20歳、身長: 170.5cm \*/

%.1fは「小数点以下1桁まで表示」という意味です。

#### エスケープシーケンス（特殊文字の表現）

画面に表示できない特殊な文字を表現するための記号です。\（バックスラッシュ）で始まります。

| シーケンス | 意味 | 使用例と効果 |
| --- | --- | --- |
| \n | 改行（newline） | printf("1行目\n2行目"); → 2行に分けて表示 |
| \t | タブ（tab） | printf("名前\t年齢"); → 名前と年齢の間に大きな空白 |
| \" | ダブルクォート | printf("彼は\"天才\"だ"); → 彼は"天才"だ |
| \\ | バックスラッシュ | printf("C:\\Program Files"); → C:\Program Files |

#### よくある使用例

/\* 変数は関数の先頭で宣言（C90準拠） \*/  
int a = 10, b = 20;  
  
/\* 基本的な挨拶 \*/  
printf("こんにちは！\n");  
  
/\* 計算結果の表示 \*/  
printf("%d + %d = %d\n", a, b, a + b); /\* 10 + 20 = 30 \*/  
  
/\* 表形式での表示 \*/  
printf("商品名\t\t価格\t個数\n");  
printf("りんご\t\t100円\t3個\n");  
printf("みかん\t\t80円\t5個\n");  
  
/\* エラーメッセージ \*/  
printf("エラー: ファイル \"%s\" が見つかりません。\n", "data.txt");

#### 初心者がよく間違えるポイント

1. **%の数と値の数が一致しない**

* /\* NG: %dが2つあるのに値が1つしかない \*/  
  printf("%d + %d = ", 10); /\* エラーまたは予期しない動作 \*/  
    
  /\* OK: 正しい書き方 \*/  
  printf("%d + %d = %d", 10, 20, 30);

1. **\nを忘れる**

* printf("Hello");  
  printf("World"); /\* HelloWorld と続けて表示される \*/  
    
  /\* 改行したい場合 \*/  
  printf("Hello\n");  
  printf("World\n");

1. **%と変数の型が一致しない**

* double pi = 3.14;  
  /\* NG: doubleに%dを使用 \*/  
  printf("円周率は%dです", pi); /\* おかしな値が表示される \*/  
    
  /\* OK: doubleには%f \*/  
  printf("円周率は%fです", pi);

### コメント

プログラムに説明を追加するためのコメント記法です。コメントはプログラムの動作に影響を与えません。

#### C言語のコメント（C90スタイル）

/\* これは複数行にわたる  
 コメントです \*/  
  
/\* 1行コメント \*/  
  
/\* ネストはできません  
 /\* これはエラー \*/   
\*/

**ポイント:**

* /\* で始まり \*/ で終わる
* 複数行にわたって記述可能
* コメントの中にコメントは書けない（ネスト不可）
* プログラムの説明や一時的なコード無効化に使用

**注意:** //スタイルの1行コメントはC99以降で使用可能ですが、この教材では移植性を重視してC90準拠の/\* \*/スタイルを使用します。

### 基本的なプログラム例

#### 例: 単純な文字列出力

/\*  
 \* ファイル名: hello\_world.c  
 \* 説明: C言語の最も基本的なプログラム  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 printf("Hello, World!\n");  
 return 0;  
}

#### 例: 複数実行の出力

/\*  
 \* ファイル名: formatted\_output.c  
 \* 説明: printf関数のさまざまなフォーマット指定子を学習  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 /\* 変数は関数の先頭で宣言（C90準拠） \*/  
 int positive\_num = 42;  
 int negative\_num = -123;  
 char letter = 'A';  
 double pi = 3.14159;  
 char name[] = "田中太郎";  
 int age = 25;  
 double height = 175.5;  
   
 /\* 基本的な文字列出力 \*/  
 printf("=== C言語フォーマット出力 ===\n");  
   
 /\* 整数データの出力 \*/  
 printf("整数: %d\n", positive\_num);  
 printf("負の整数: %d\n", negative\_num);  
   
 /\* 文字データの出力 \*/  
 printf("文字: %c\n", letter);  
 printf("文字コード: %c\n", 65); /\* Aのアスキーコード \*/  
   
 /\* 文字列の出力 \*/  
 printf("文字列: %s\n", "Hello, World!");  
   
 /\* 浮動小数点数の出力 \*/  
 printf("浮動小数点数: %f\n", pi);  
 printf("小数点以下2桁: %.2f\n", pi);  
   
 /\* エスケープシーケンスの使用 \*/  
 printf("\n--- エスケープシーケンス ---\n");  
 printf("改行\nとタブ\tの使用例：\n");  
 printf("項目1\t項目2\t項目3\n");  
 printf("値A\t値B\t値C\n");  
   
 /\* 特殊文字の表示 \*/  
 printf("\n--- 特殊文字の表示 ---\n");  
 printf("ダブルクォート: \"Hello\"\n");  
 printf("バックスラッシュ: C:\\Program Files\\\n");  
   
 /\* フォーマット指定子の組み合わせ \*/  
 printf("\n--- 複合表示 ---\n");  
 printf("名前: %s, 年齢: %d歳, 身長: %.1fcm\n", name, age, height);  
   
 return 0;  
}  
/\*  
学習ポイント:  
1. printf関数の基本的な使用方法  
2. 各種データ型の書式指定子（%d, %c, %s, %f）  
3. エスケープシーケンス（\n, \t, \", \\）  
4. フィールド幅や精度の指定（%.2f等）  
5. 複数の値を組み合わせた出力  
\*/

#### 例: エスケープシーケンスの使用

/\*  
 \* ファイル名: escape\_sequences.c  
 \* 説明: エスケープシーケンスの使用例  
 \* 規格: C90準拠  
 \*/  
#include <stdio.h>  
  
int main(void)  
{  
 /\* エスケープシーケンスの例 \*/  
 printf("=== エスケープシーケンスの使用例 ===\n\n");  
   
 /\* 改行とタブ \*/  
 printf("1. 改行とタブ:\n");  
 printf("1行目\n2行目\n3行目\n");  
 printf("名前\t\t年齢\t身長\n");  
 printf("田中太郎\t25\t170.5\n");  
 printf("山田花子\t23\t158.3\n\n");  
   
 /\* 特殊文字 \*/  
 printf("2. 特殊文字の表示:\n");  
 printf("ダブルクォート: \"Hello, World!\"\n");  
 printf("シングルクォート: \'A\'\n");  
 printf("バックスラッシュ: C:\\Program Files\\MyApp\\\n");  
 printf("パーセント記号: 100%%\n\n");  
   
 /\* バックスペースとキャリッジリターン \*/  
 printf("3. その他のエスケープシーケンス:\n");  
 printf("ABC\b\_"); /\* Cを消して\_を表示 → AB\_ \*/  
 printf("\n");  
 printf("Hello\rWorld"); /\* HelloをWorldで上書き → World \*/  
 printf("\n\n");  
   
 /\* ASCII文字の表示 \*/  
 printf("4. ASCII文字の直接指定:\n");  
 printf("ベル音: \\a → \a\n"); /\* ビープ音（環境依存） \*/  
 printf("文字コード65: \\x41 → \x41\n"); /\* A \*/  
 printf("8進数101: \\101 → \101\n"); /\* A \*/  
   
 return 0;  
}

## 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 hello\_world  
# 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

* コメントは /\* \*/ 形式のみ
* 変数宣言は関数やブロックの先頭で実行う
* forループ内での変数宣言は不可

### C99以降の拡張

* // 形式のコメント追加
* forループ内での変数宣言が可能
* 複合リテラル等の新機能

## 次の章へ

基本文法を理解したら、[データ型と変数](../data-types/README.md) に進んでください。

## 参考資料

* [printf関数リファレンス](https://ja.cppreference.com/w/c/io/fprintf)
* [エスケープシーケンス一覧](https://ja.cppreference.com/w/c/language/escape)
* [C90規格仕様](https://www.iso.org/standard/17782.html)

# 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

### データ型と変数の世界へようこそ

プログラミングでは、様々な種類のデータを扱います。数字、文字、文章など、それぞれのデータには適切な「入れ物」が必要です。この「入れ物」がデータ型で、その入れ物に名前を付けたものが変数です。

例えば、年齢を記録するなら整数の入れ物（int型）、身長を記録するなら小数の入れ物（double型）を使います。この章では、C言語で使えるデータ型と、変数の使い方を学びます。

### C言語の基本データ型

C言語には、扱うデータの種類に応じて様々なデータ型が用意されています。まずは最も基本的な型から見ていきましょう。

#### なぜデータ型が必要なのか？

コンピュータのメモリは「0」と「1」の集まりです。データ型は、この0と1の並びをどのように解釈するかを決める「ルール」です。同じ0と1の並びでも、整数として読むか、文字として読むかで意味が変わります。

また、データ型によってメモリの使用量も変わります。大きな数を扱うには大きな入れ物が必要ですが、小さな数なら小さな入れ物で十分です。適切なデータ型を選ぶことで、効率的なプログラムが書けます。

#### 整数型（整数を扱う型）

整数型は、小数点のない数（…、-2、-1、0、1、2、…）を扱うための型です。

**注意**: 以下の表のサイズと範囲は一般的な値です。実際のサイズは使用するコンピュータやコンパイラによって異なる場合があります。確実なサイズが必要な場合はsizeof演算子で確認してください。

| 型名 | サイズ | 範囲 | 使用例 |
| --- | --- | --- | --- |
| char | 1バイト | -128 〜 127 | 文字（‘A’、‘B’）、小さな数 |
| unsigned char | 1バイト | 0 〜 255 | 文字コード、画像のピクセル値 |
| short | 2バイト以上 | -32,768 〜 32,767以上 | 音声データ、座標値 |
| unsigned short | 2バイト以上 | 0 〜 65,535以上 | ポート番号、色情報 |
| int | 2バイト以上 | 約-32,768以上 | 一般的な整数（年齢、個数など） |
| unsigned int | 2バイト以上 | 0以上 | 大きな正の数、ビット演算 |
| long | 4バイト以上 | システム依存 | より大きな整数 |
| unsigned long | 4バイト以上 | 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（整数型）

* 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バイト以上 | 約6-7桁 | ゲームの座標、簡単な計算 |
| double | 8バイト以上 | 約15-16桁 | 科学計算、金額計算 |
| long double | 8バイト以上 | 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（浮動小数点数型）

* 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）

異なるデータ型の間で値を変換することを型変換（キャスト）といいます。C言語では自動的に行われる場合と、明示的に指定する場合があります。

#### 暗黙の型変換（自動型変換）

コンパイラが自動的に型を変換します。一般的に、小さい型から大きい型への変換は安全です。

int i = 42;  
double d;  
d = i; /\* int から double への自動変換 \*/  
  
char c = 'A';  
int num = c; /\* char から int への自動変換（65になる） \*/  
  
float f = 3.14f;  
double d2 = f; /\* float から double への自動変換 \*/

**変換の優先順位（低→高）**： char → short → int → long → float → double → long double

#### 明示的な型変換（キャスト）

プログラマが意図的に型を変換する場合は、キャスト演算子を使います。

double pi = 3.14159;  
int rounded = (int)pi; /\* 小数部分を切り捨て（3になる） \*/  
  
int total = 100;  
int count = 3;  
double average = (double)total / count; /\* 33.333... になる \*/  
  
char letter = (char)65; /\* 'A' になる \*/

#### 型変換の注意点

##### 1. 精度の損失

double big = 1234567890.123456;  
float small = (float)big; /\* 精度が失われる \*/  
printf("%.6f\n", small); /\* 1234567936.000000 など \*/

##### 2. 整数除算の罠

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) \*/  
result = a / (double)b; /\* 結果: 2.5 (これでもOK) \*/

##### 3. オーバーフローに注意

int big\_int = 100000;  
short small\_short = (short)big\_int; /\* オーバーフロー発生 \*/  
printf("%d\n", small\_short); /\* 予期しない値 \*/  
  
unsigned int positive = 4294967295; /\* unsignedの最大値 \*/  
int negative = (int)positive; /\* -1 になる \*/

##### 4. 符号の扱い

int neg = -100;  
unsigned int pos = (unsigned int)neg; /\* 大きな正の数になる \*/  
printf("%u\n", pos); /\* 4294967196 など \*/

### 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。変数を適切なスコープで管理することで、メモリの効率的な使用とバグの少ないプログラムが実現できます。

#### スコープを理解することの重要性

1. **名前の衝突を防ぐ** - 同じ名前の変数を異なるスコープで使える
2. **メモリの効率的利用** - 必要な期間だけメモリを使用
3. **プログラムの保守性向上** - 変数の影響範囲が限定される
4. **バグの削減** - 意図しない変数へのアクセスを防げる

#### 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。

int global\_var = 100; /\* ファイル全体で使える \*/  
  
int main(void)  
{  
 printf("グローバル変数: %d\n", global\_var);  
 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; /\* この関数内でのみ有効 \*/  
 printf("local\_a = %d\n", local\_a);  
}  
  
void function\_b(void)  
{  
 /\* local\_a はここでは使えない \*/  
 int local\_b = 20;  
 printf("local\_b = %d\n", local\_b);  
}

#### 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 x = 10;  
   
 if (x > 5) {  
 int y = 20; /\* ifブロック内でのみ有効 \*/  
 printf("x = %d, y = %d\n", x, y);  
 }  
 /\* y はここでは使えない \*/  
   
 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);  
   
 {  
 int x = 200; /\* ローカル変数（グローバルを隠蔽） \*/  
 printf("ローカルx = %d\n", x);  
 }  
   
 printf("グローバルx（再度）= %d\n", x);  
   
 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。スコープが「どこから見えるか」を決めるのに対し、生存期間は「いつまで存在するか」を決めます。

#### 生存期間とスコープの違い

* **スコープ**: 変数にアクセスできる範囲（空間的な概念）
* **生存期間**: 変数がメモリに存在する期間（時間的な概念）

同じスコープでも生存期間が異なる場合があり、この違いを理解することが重要です。

#### 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。

**特徴**:

* デフォルトの生存期間（何も指定しない場合）
* スタック領域に確保される
* 関数が呼ばれるたびに新しく作られる
* 初期化しないと不定値（ゴミ値）を持つ

void count\_calls(void)  
{  
 int count = 0; /\* 自動変数：毎回0に初期化 \*/  
 count++;  
 printf("呼び出し回数: %d\n", count); /\* 常に1が表示される \*/  
}  
  
/\* メイン関数での例 \*/  
int main(void)  
{  
 count\_calls(); /\* 1 \*/  
 count\_calls(); /\* 1 \*/  
 count\_calls(); /\* 1 \*/  
 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。

**特徴**:

* プログラム実行中ずっと存在
* データ領域（静的領域）に確保される
* 自動的に0で初期化される
* 関数を抜けても値を保持する

void count\_calls\_static(void)  
{  
 static int count = 0; /\* 静的変数：初回のみ初期化 \*/  
 count++;  
 printf("呼び出し回数: %d\n", count);  
}  
  
/\* メイン関数での例 \*/  
int main(void)  
{  
 count\_calls\_static(); /\* 1 \*/  
 count\_calls\_static(); /\* 2 \*/  
 count\_calls\_static(); /\* 3 \*/  
 return 0;  
}

**静的変数の用途**:

* 関数呼び出し間での状態保持
* カウンターやフラグの実装
* 初回実行時のみの処理制御

#### 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。

**特徴**:

* グローバル変数と同じ生存期間（プログラム全体）
* ファイルスコープに限定（他のファイルから見えない）
* 名前の衝突を防げる
* モジュール内部の状態管理に最適

/\* file1.c \*/  
static int file\_private = 100; /\* このファイル内でのみ有効 \*/  
  
int get\_private\_value(void)  
{  
 return file\_private;  
}  
  
void set\_private\_value(int value)  
{  
 file\_private = value;  
}  
  
/\* file2.c \*/  
/\* static int file\_private = 200; 他のファイルなら同名でもOK \*/

#### 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）。

**特徴**:

* CPUレジスタへの格納を要求（保証はされない）
* アドレス演算子（&）が使えない
* 自動変数と同じ生存期間
* 現代のコンパイラでは最適化で自動判断

int main(void)  
{  
 register int i; /\* 高速アクセス用（推奨） \*/  
   
 for (i = 0; i < 1000000; i++) {  
 /\* 繰り返し処理 \*/  
 }  
   
 /\* &i は使えない（エラーになる） \*/  
   
 return 0;  
}

**注意**: 現代のコンパイラは自動的に最適化するため、registerキーワードの効果は限定的です。

#### 生存期間の実践的な比較例

以下のプログラムで、各種変数の生存期間の違いを確認できます：

#include <stdio.h>  
  
int global = 100; /\* グローバル変数（プログラム全体） \*/  
static int static\_global = 200; /\* 静的グローバル（プログラム全体） \*/  
  
void demonstrate\_lifetime(void)  
{  
 int auto\_var = 0; /\* 自動変数（関数実行中のみ） \*/  
 static int static\_var = 0; /\* 静的変数（プログラム全体） \*/  
   
 auto\_var++;  
 static\_var++;  
 global++;  
 static\_global++;  
   
 printf("自動変数: %d, 静的変数: %d\n", auto\_var, static\_var);  
 printf("グローバル: %d, 静的グローバル: %d\n", global, static\_global);  
}  
  
int main(void)  
{  
 printf("=== 1回目の呼び出し ===\n");  
 demonstrate\_lifetime();  
   
 printf("\n=== 2回目の呼び出し ===\n");  
 demonstrate\_lifetime();  
   
 printf("\n=== 3回目の呼び出し ===\n");  
 demonstrate\_lifetime();  
   
 return 0;  
}  
/\* 出力結果:  
 自動変数は毎回1、静的変数は1,2,3と増加  
 グローバル変数も増加し続ける \*/

### 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 a; /\* 初期化なし：不定値 \*/  
 int b = 10; /\* 初期化あり：10 \*/  
   
 printf("a = %d (不定値)\n", a);  
 printf("b = %d\n", b);  
   
 return 0;  
}

#### 静的変数の初期化

static int global\_static; /\* 0に自動初期化 \*/  
static int global\_init = 100; /\* 明示的に初期化 \*/  
  
void show\_values(void)  
{  
 static int local\_static; /\* 0に自動初期化 \*/  
 printf("静的変数: %d, %d, %d\n",   
 global\_static, global\_init, local\_static);  
}

### 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 = 0; /\* 静的変数で状態を保持 \*/  
 return ++id;  
}  
  
int main(void)  
{  
 printf("ID: %d\n", get\_next\_id()); /\* 1 \*/  
 printf("ID: %d\n", get\_next\_id()); /\* 2 \*/  
 printf("ID: %d\n", get\_next\_id()); /\* 3 \*/  
 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

プログラムで変更されない値を定数として定義することで、コードの可読性と保守性が向上します。C言語では主に2つの方法で定数を定義できます。

#### なぜ定数を使うのか？

1. **マジックナンバーの排除** - 意味不明な数値をなくす
2. **一括変更が容易** - 定数の値を1箇所で管理
3. **意図の明確化** - 値の意味が明確になる
4. **誤った変更の防止** - 値が変更されないことを保証

#### constキーワード（型付き定数）

const修飾子を使って、変数を読み取り専用にします。

const int MAX\_SIZE = 100; /\* 整数定数 \*/  
const double PI = 3.14159265359; /\* 浮動小数点定数 \*/  
const char NEWLINE = '\n'; /\* 文字定数 \*/  
  
/\* 配列のサイズには使えない（C90） \*/  
/\* const int SIZE = 10; \*/  
/\* int array[SIZE]; エラー！ \*/  
  
/\* ポインタと const \*/  
const char \*str = "Hello"; /\* 文字列の内容を変更不可 \*/  
char \* const ptr = buffer; /\* ポインタ自体を変更不可 \*/  
const char \* const cptr = "World"; /\* 両方とも変更不可 \*/

**const の特徴**:

* 型安全性がある
* デバッガで値を確認できる
* スコープの制御が可能
* C90では配列サイズに使えない

#### #defineプリプロセッサ（マクロ定数）

プリプロセッサ指令で、コンパイル前にテキスト置換を行います。

#define MAX\_STUDENTS 50 /\* 数値定数 \*/  
#define PI 3.14159265359 /\* 浮動小数点定数 \*/  
#define TITLE "C Language Tutorial" /\* 文字列定数 \*/  
#define TRUE 1 /\* 論理定数 \*/  
#define FALSE 0  
  
/\* 配列のサイズに使える \*/  
int scores[MAX\_STUDENTS];  
  
/\* 式も定義可能（括弧推奨） \*/  
#define SQUARE(x) ((x) \* (x))  
#define MAX(a, b) ((a) > (b) ? (a) : (b))

**#define の特徴**:

* 型情報を持たない
* どこでも使える（配列サイズなど）
* プリプロセス時に単純置換
* デバッグが難しい場合がある

#### const vs #define の使い分け

| 特徴 | const | #define |
| --- | --- | --- |
| 型チェック | あり | なし |
| スコープ | あり | なし（ファイル全体） |
| デバッグ | 容易 | 困難 |
| 配列サイズ（C90） | 使用不可 | 使用可能 |
| メモリ使用 | 変数として確保 | 置換のみ |

**使い分けの指針**:

* 一般的な定数 → const を優先
* 配列サイズ → #define（C90）
* 条件コンパイル → #define
* 簡単なマクロ → #define

#### 定数定義のベストプラクティス

/\* 命名規則：全て大文字、単語はアンダースコアで区切る \*/  
#define MAX\_BUFFER\_SIZE 1024  
#define DEFAULT\_TIMEOUT 30  
#define ERROR\_FILE\_NOT\_FOUND -1  
  
/\* 関連する定数はグループ化 \*/  
/\* ファイルシステム関連 \*/  
#define MAX\_PATH\_LENGTH 260  
#define MAX\_FILENAME\_LENGTH 255  
  
/\* ネットワーク関連 \*/  
#define DEFAULT\_PORT 8080  
#define MAX\_CONNECTIONS 100  
  
/\* エラーコード \*/  
#define SUCCESS 0  
#define ERROR\_INVALID\_ARGUMENT -1  
#define ERROR\_OUT\_OF\_MEMORY -2

#### enum を使った定数定義

関連する定数のグループは列挙型を使うと便利です。

/\* 曜日の定数 \*/  
enum weekday {  
 SUNDAY, /\* 0 \*/  
 MONDAY, /\* 1 \*/  
 TUESDAY, /\* 2 \*/  
 WEDNESDAY, /\* 3 \*/  
 THURSDAY, /\* 4 \*/  
 FRIDAY, /\* 5 \*/  
 SATURDAY /\* 6 \*/  
};  
  
/\* 明示的な値の指定 \*/  
enum status {  
 STATUS\_OK = 0,  
 STATUS\_ERROR = -1,  
 STATUS\_BUSY = 100  
};

### 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。sizeof演算子には2つの形式があります。

#### sizeof演算子の2つの形式

1. **sizeof(型名)** - 型名には必ず括弧が必要
2. **sizeof 式** - 式には括弧は任意（つけてもよい）

/\* 形式1: sizeof(型名) - 括弧が必要 \*/  
size\_t size1 = sizeof(int); /\* 正しい \*/  
size\_t size2 = sizeof(double); /\* 正しい \*/  
/\* size\_t size3 = sizeof int; エラー！括弧が必要 \*/  
  
/\* 形式2: sizeof 式 - 括弧は任意 \*/  
int num = 42;  
size\_t size4 = sizeof num; /\* 正しい（括弧なし） \*/  
size\_t size5 = sizeof(num); /\* 正しい（括弧あり） \*/  
size\_t size6 = sizeof num + 1; /\* sizeof num の後に +1 \*/

#### sizeof演算子の特性

* **コンパイル時に評価**: 実行時ではなくコンパイル時にサイズが決定
* **型は size\_t**: 結果の型は size\_t（符号なし整数型）
* **副作用なし**: sizeof(i++) のような式でも i は増加しない

#### 実用的な使用例

/\* 配列の要素数を計算 \*/  
int array[10];  
size\_t count = sizeof array / sizeof array[0]; /\* 要素数 = 10 \*/  
  
/\* ポインタと配列の違い \*/  
char arr[] = "Hello"; /\* 配列: 6バイト（null終端含む） \*/  
char \*ptr = "Hello"; /\* ポインタ: 8バイト（64ビット環境） \*/  
  
/\* 動的メモリ確保での使用 \*/  
int \*p = malloc(sizeof(int) \* 100); /\* int型100個分 \*/

#### サンプルプログラム

完全な実装例は以下のファイルを参照してください：

* [sizeof\_demo.c](examples/sizeof_demo.c) - C90準拠版
* [sizeof\_demo\_c99.c](examples/sizeof_demo_c99.c) - C99準拠版（可変長配列対応）

## 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)

# 演算子

## 対応C規格

* **主要対象:** C90
* **学習内容:** 算術演算子、関係演算子、論理演算子、ビット演算子（概要）、演算子の優先順位

## 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。

* さまざまな演算子の種類と使い方を理解する
* 演算子の優先順位と結合規則を把握する
* 複雑な式を正しく記述できる
* インクリメント・デクリメント演算子を適切に使える
* ビット演算の基本概念を理解する
* 短絡評価の基本的な動作を理解する

## 概要と詳細

### 演算子の世界へようこそ

プログラミングでは、数学の計算だけでなく、データの比較、条件判断、ビット操作など、様々な「演算」を行います。演算子は、これらの操作を行うための「道具」です。 電卓を使ったことがある人なら、すでに演算子を使っています。+、-、×、÷などの記号が演算子です。C言語では、これらに加えて、プログラミング特有の便利な演算子がたくさん用意されています。 この章では、C言語の演算子を一つずつ学び、それらを組み合わせて複雑な計算や判断ができるようになることを目指します。

### 算術演算子（計算の基本）

算術演算子は、日常的な計算を行うための演算子です。小学校で習った四則演算がそのまま使えます。

| 演算子 | 意味 | 例 | 結果 | 日常での例 |
| --- | --- | --- | --- | --- |
| + | 加算（足し算） | 5 + 3 | 8 | 5個のリンゴに3個追加 |
| - | 減算（引き算） | 5 - 3 | 2 | 5個のリンゴから3個食べる |
| \* | 乗算（掛け算） | 5 \* 3 | 15 | 5個入りの箱を3箱 |
| / | 除算（割り算） | 7 / 3 | 2 | 7個のお菓子を3人で分ける |
| % | 剰余（余り） | 7 % 3 | 1 | 7個を3人で分けた余り |

#### なぜ剰余（%）演算子が重要なのか？

剰余演算子は日常生活でも意外と使われています。

* **曜日の計算**：今日から10日後は何曜日？ → (今日の曜日 + 10) % 7
* **偶数・奇数の判定**：number % 2 が 0 なら偶数、1 なら奇数
* **時計の計算**：15時から10時間後は？ → (15 + 10) % 24 = 1時

/\* 偶数・奇数の判定 \*/  
int number = 17;  
if (number % 2 == 0) {  
 printf("%dは偶数です\n", number);  
} else {  
 printf("%dは奇数です\n", number);  
}

#### 整数除算の罠（初心者が必ず遭遇する問題）

C言語では、整数同士の割り算は「整数除算」になり、小数点以下が切り捨てられます。

/\* よくある間違い \*/  
int a = 7, b = 3;  
double result = a / b; /\* 期待: 2.333... 実際: 2.0 \*/  
  
/\* なぜ？ \*/  
/\* a / b の計算時点で整数除算（7 ÷ 3 = 2）が行われ、  
 その後でdoubleに変換されるため \*/  
  
/\* 正しい方法1：どちらかをdoubleにキャスト \*/  
double result = (double)a / b; /\* 結果: 2.333... \*/  
/\* 正しい方法2：最初からdoubleで計算 \*/  
double x = 7.0, y = 3.0;  
double result = x / y; /\* 結果: 2.333... \*/

**覚え方のコツ**。

* 整数 ÷ 整数 = 整数（小数点以下は消える）
* 小数 ÷ 整数 = 小数（小数として計算される）
* 整数 ÷ 小数 = 小数（小数として計算される）

### 代入演算子（値を保存する）

代入演算子は、計算結果や値を変数に保存するための演算子です。=は数学の「等しい」ではなく、「右の値を左の変数に入れる」という意味です。

#### 基本の代入演算子

int score = 85; /\* 「scoreに85を入れる」と読む \*/  
score = 90; /\* scoreの値を90に更新 \*/

**重要な違い**。

* 数学：x = 5 は「xは5に等しい」
* C言語：x = 5 は「xに5を代入する（入れる）」

#### 複合代入演算子（ショートカット）

プログラミングでは、「現在の値に何かを足す/引く」という操作が頻繁に行われます。複合代入演算子はこれを簡潔に書くための便利な記法です。

| 演算子 | 意味 | 例 | 等価な記述 | 使用場面 |
| --- | --- | --- | --- | --- |
| = | 代入 | a = 5 | - | 初期値設定 |
| += | 加算して代入 | a += 3 | a = a + 3 | カウントアップ、合計計算 |
| -= | 減算して代入 | a -= 2 | a = a - 2 | カウントダウン、在庫減少 |
| \*= | 乗算して代入 | a \*= 2 | a = a \* 2 | 倍増処理、累乗計算 |
| /= | 除算して代入 | a /= 3 | a = a / 3 | 平均値計算、分割処理 |
| %= | 剰余を代入 | a %= 4 | a = a % 4 | 循環処理、範囲制限 |

#### 実用的な使用例

/\* ゲームのスコア計算 \*/  
int score = 0;  
score += 100; /\* 敵を倒した：100点追加 \*/  
score += 50; /\* アイテムゲット：50点追加 \*/  
score -= 30; /\* ダメージを受けた：30点減少 \*/  
printf("現在のスコア: %d\n", score); /\* 120 \*/  
  
/\* 在庫管理 \*/  
int stock = 100;  
stock -= 15; /\* 15個販売 \*/  
stock += 50; /\* 50個入荷 \*/  
printf("在庫数: %d\n", stock); /\* 135 \*/  
  
/\* レベルアップシステム \*/  
int experience = 80;  
experience += 40; /\* 経験値40獲得 \*/  
if (experience >= 100) {  
 printf("レベルアップ！\n");  
 experience %= 100; /\* 100で割った余りが次レベルへの経験値 \*/  
}

#### なぜ複合代入演算子を使うのか？

1. **コードが短くなる**

* /\* 長い書き方 \*/  
  total\_price = total\_price + item\_price;  
    
  /\* 短い書き方 \*/  
  total\_price += item\_price;

1. **意図が明確になる**

* health -= damage; /\* 「ヘルスからダメージを引く」と直感的に理解できる \*/

1. **タイプミスを防げる**

* /\* 変数名が長い場合、タイプミスしやすい \*/  
  very\_long\_variable\_name = very\_long\_variable\_name + 1;  
    
  /\* こちらの方が安全 \*/  
  very\_long\_variable\_name += 1;

### インクリメント・デクリメント演算子（1つずつ増減）

プログラミングでは「1増やす」「1減らす」という操作が非常に多く使われます。そのため、専用の演算子が用意されています。

| 演算子 | 意味 | 前置形 | 後置形 | 使用場面 |
| --- | --- | --- | --- | --- |
| ++ | 1増やす（インクリメント） | ++a | a++ | カウンター、ループ制御 |
| -- | 1減らす（デクリメント） | --a | a-- | カウントダウン、在庫減少 |

#### なぜこの演算子が必要なのか？

/\* カウンターの例（よくあるパターン） \*/  
int count = 0;  
count = count + 1; /\* 長い書き方 \*/  
count += 1; /\* 複合代入演算子を使った書き方 \*/  
count++; /\* 最も簡潔な書き方 \*/

ループ処理では特に頻繁に使われます。

/\* 10回繰り返す \*/  
int i;  
for (i = 0; i < 10; i++) { /\* i++ で毎回1増やす \*/  
 printf("繰り返し %d 回目\n", i + 1);  
}

#### 前置と後置の違い（重要！）

この演算子には「前置形」と「後置形」があり、微妙に動作が異なります。

int a = 5, b = 5;  
int x, y;  
  
/\* 前置形：先に増やしてから使う \*/  
x = ++a; /\* ①aを6に増やす → ②xに6を代入 \*/  
printf("x = %d, a = %d\n", x, a); /\* x = 6, a = 6 \*/  
  
/\* 後置形：使ってから増やす \*/  
y = b++; /\* ①yに5を代入 → ②bを6に増やす \*/  
printf("y = %d, b = %d\n", y, b); /\* y = 5, b = 6 \*/

#### 実用的な例で理解する

/\* 商品番号の発行（後置形の活用） \*/  
int product\_id = 1000;  
printf("商品A: ID=%d\n", product\_id++); /\* 1000を表示、その後1001に \*/  
printf("商品B: ID=%d\n", product\_id++); /\* 1001を表示、その後1002に \*/  
printf("商品C: ID=%d\n", product\_id++); /\* 1002を表示、その後1003に \*/  
  
/\* 在庫チェック（前置形の活用） \*/  
int stock = 1;  
if (--stock >= 0) { /\* 先に1減らして0に、その後で判定 \*/  
 printf("在庫あり、販売可能\n");  
} else {  
 printf("在庫なし\n");  
}

#### よくある間違いと対策

/\* 間違い例1：意図しない動作 \*/  
int n = 5;  
int result = n++ + n++; /\* 未定義動作！使ってはいけない \*/  
  
/\* 正しい書き方 \*/  
int n = 5;  
int result = n + (n + 1); /\* 明確に意図を表現 \*/  
n += 2;  
  
/\* 間違い例2：条件文での使用 \*/  
int count = 10;  
if (count-- > 0) { /\* 10 > 0 を判定してからcountを9に \*/  
 /\* この時点でcountは9 \*/  
}  
  
/\* より明確な書き方 \*/  
int count = 10;  
if (count > 0) {  
 count--;  
 /\* 処理 \*/  
}

#### 使い分けのガイドライン

* **単独で使う場合**：前置・後置どちらでも同じ
* i++; /\* これと \*/  
  ++i; /\* これは同じ結果 \*/
* **値を使いながら増減する場合**：意図に応じて選択
* array[index++] = value; /\* 現在の位置に代入してから次へ \*/  
  array[++index] = value; /\* 次の位置に移動してから代入 \*/

**C99版での詳細:** [increment\_decrement\_c99.c](solutions/increment_decrement_c99.c)

### 関係演算子（比較する）

関係演算子は、2つの値を比較して「正しい（真）」か「正しくない（偽）」かを判定します。日常生活での「大きい・小さい」「同じ・違う」の判断をプログラムで行うための演算子です。

#### C言語における真偽の表現

C言語では、真偽を表すために専用のブール型がありません（C99以降では\_Bool型が追加されましたが、この章ではC90準拠のため扱いません）。代わりに、整数値を使って真偽を表現します。

* **真（True）** = 0以外の値（通常は1）
* **偽（False）** = 0

この仕組みを理解することは、条件判断を正しく書くために非常に重要です。

| 演算子 | 意味 | 例 | 結果 | 日常での例 |
| --- | --- | --- | --- | --- |
| == | 等しい | 5 == 3 | 0 (偽) | パスワードが一致するか |
| != | 等しくない | 5 != 3 | 1 (真) | 在庫が0でないか |
| < | より小さい | 5 < 3 | 0 (偽) | 年齢が18歳未満か |
| <= | 以下 | 5 <= 3 | 0 (偽) | 体重が標準以下か |
| > | より大きい | 5 > 3 | 1 (真) | 得点が合格点より高いか |
| >= | 以上 | 5 >= 3 | 1 (真) | 残高が購入金額以上か |

#### 実用的な例

/\* パスワードチェック \*/  
int password = 1234;  
int input = 1234;  
if (password == input) {  
 printf("ログイン成功\n");  
}  
  
/\* 在庫管理 \*/  
int stock = 5;  
if (stock > 0) {  
 printf("在庫あり（%d個）\n", stock);  
} else {  
 printf("在庫切れ\n");  
}  
  
/\* 年齢制限チェック \*/  
int age = 16;  
if (age >= 18) {  
 printf("成人向けコンテンツを表示\n");  
} else {  
 printf("アクセス制限中\n");  
}

#### よくある間違い：= と == の混同

int x = 5;  
  
/\* 間違い：代入になってしまう \*/  
if (x = 10) { /\* xに10を代入し、10（真）として評価 \*/  
 printf("この部分は必ず実行される\n");  
}  
  
/\* 正しい：比較 \*/  
if (x == 10) { /\* xが10と等しいか比較 \*/  
 printf("xは10です\n");  
}

### 論理演算子（条件を組み合わせる）

論理演算子は、複数の条件を組み合わせて複雑な判断を行うための演算子です。

| 演算子 | 意味 | 例 | 説明 | 日常での例 |
| --- | --- | --- | --- | --- |
| && | かつ（AND） | a && b | 両方とも真なら真 | 「晴れ」かつ「暖かい」なら外出 |
| || | または（OR） | a || b | どちらか真なら真 | 「土曜」または「日曜」なら休み |
| ! | でない（NOT） | !a | 真偽を反転 | 「雨でない」なら洗濯 |

#### 真理値表で理解する

**AND（&&）の動作**

| A | B | A && B |
| --- | --- | --- |
| 真 | 真 | 真 |
| 真 | 偽 | 偽 |
| 偽 | 真 | 偽 |
| 偽 | 偽 | 偽 |

**OR（||）の動作**

| A | B | A || B |
| --- | --- | --- |
| 真 | 真 | 真 |
| 真 | 偽 | 真 |
| 偽 | 真 | 真 |
| 偽 | 偽 | 偽 |

#### 実用的な例

/\* 営業時間チェック（9時〜17時） \*/  
int hour = 14;  
if (hour >= 9 && hour <= 17) {  
 printf("営業中です\n");  
}  
  
/\* 休日判定 \*/  
int day = 6; /\* 1=月曜...7=日曜 \*/  
if (day == 6 || day == 7) {  
 printf("週末です\n");  
}  
  
/\* 割引条件（学生でない一般客） \*/  
int is\_student = 0;  
if (!is\_student) {  
 printf("一般料金です\n");  
}  
  
/\* 複雑な条件：優待条件 \*/  
int age = 65;  
int is\_member = 1;  
if ((age >= 65 || age < 18) && is\_member) {  
 printf("優待料金が適用されます\n");  
}

#### 短絡評価（ショートサーキット）

論理演算子には「短絡評価」という重要な特性があります。これは、式の評価中に結果が確定した時点で、残りの評価をスキップする仕組みです。

##### なぜ短絡評価が重要なのか？

1. **パフォーマンスの向上** - 不要な計算を省略できる
2. **エラーの回避** - 危険な操作を安全に書ける
3. **条件付き実行** - 関数呼び出しを制御できる

##### &&（AND）の短絡評価

左側が偽（0）の場合、右側を評価しなくても全体が偽になることが確定するため、右側は評価されません。

/\* 基本的な例 \*/  
int a = 0, b = 10;  
if (a && b) {  
 /\* a が 0（偽）なので、b は評価されない \*/  
 printf("この行は実行されない\n");  
}  
  
/\* エラー回避の例：ゼロ除算を防ぐ \*/  
int divisor = 0;  
int value = 100;  
if (divisor != 0 && value / divisor > 5) {  
 /\* divisor が 0 なので、value / divisor は計算されない \*/  
 /\* ゼロ除算エラーを回避できる！ \*/  
 printf("結果: %d\n", value / divisor);  
}  
  
/\* ポインタのNULLチェック \*/  
char \*ptr = NULL;  
if (ptr != NULL && \*ptr == 'A') {  
 /\* ptr が NULL なので、\*ptr へのアクセスは行われない \*/  
 /\* セグメンテーションフォルトを回避！ \*/  
}

##### ||（OR）の短絡評価

左側が真（0以外）の場合、右側を評価しなくても全体が真になることが確定するため、右側は評価されません。

/\* 基本的な例 \*/  
int is\_admin = 1;  
int has\_permission = 0;  
if (is\_admin || has\_permission) {  
 /\* is\_admin が 1（真）なので、has\_permission は評価されない \*/  
 printf("アクセス許可\n");  
}  
  
/\* 関数呼び出しの制御 \*/  
int cached\_result = 1;  
if (cached\_result || expensive\_calculation()) {  
 /\* cached\_result が真なので、高コストな関数は呼ばれない \*/  
 printf("処理を実行\n");  
}  
  
/\* デフォルト値の設定 \*/  
int user\_input = 0; /\* ユーザーが何も入力しなかった \*/  
int value = user\_input || get\_default\_value();  
/\* user\_input が 0 の場合のみ get\_default\_value() が呼ばれる \*/

**より高度なテクニック**：短絡評価を使った安全なプログラミングパターンについては、[高度な演算子テクニック](../advanced/README.md" \l "短絡評価による安全なプログラミング)の章を参照してください。

##### 注意点：副作用のある式

短絡評価により、期待した副作用（変数の更新など）が発生しない場合があります。

int x = 0, y = 0;  
  
/\* 注意が必要な例 \*/  
if (x++ > 0 && y++ > 0) {  
 /\* x は増加するが、y は増加しない！ \*/  
}  
printf("x = %d, y = %d\n", x, y); /\* x = 1, y = 0 \*/  
  
/\* より明確な書き方 \*/  
x++;  
y++;  
if (x > 0 && y > 0) {  
 /\* 意図が明確 \*/  
}

#### 論理演算子の優先順位

論理演算子を使用する際、最も重要なのは演算子の優先順位を理解することです。C言語では、AND演算子（&&）がOR演算子（||）よりも優先順位が高いという規則があります。この規則を理解していないと、意図しない動作を引き起こす可能性があります。

##### 基本的な優先順位規則

/\* 括弧なしの場合（&&が||より優先） \*/  
if (a || b && c) /\* a || (b && c) として評価される \*/  
  
/\* 上記は以下と同じ意味 \*/  
if (a || (b && c)) /\* 明示的に括弧で示した場合 \*/  
  
/\* もし OR を先に評価したい場合は括弧が必要 \*/  
if ((a || b) && c) /\* 意図を明確に表現 \*/

##### なぜ && が || より優先されるのか

この優先順位は、数学の掛け算が足し算より優先されるのと同じ考え方です： - AND（&&）は論理的な「掛け算」（両方が真である必要がある） - OR（||）は論理的な「足し算」（どちらかが真であればよい）

##### よくある間違いと正しい書き方

/\* 例1: ユーザーの権限チェック \*/  
int is\_admin = 0;  
int is\_moderator = 1;  
int can\_edit = 1;  
  
/\* 間違いやすい例 \*/  
if (is\_admin || is\_moderator && can\_edit) {  
 /\* 実際の評価: is\_admin || (is\_moderator && can\_edit) \*/  
 /\* 管理者、またはモデレーターかつ編集権限がある場合 \*/  
 printf("アクセス許可\n");  
}  
  
/\* 意図が異なる場合の正しい書き方 \*/  
if ((is\_admin || is\_moderator) && can\_edit) {  
 /\* 管理者またはモデレーターで、かつ編集権限がある場合 \*/  
 printf("アクセス許可\n");  
}

/\* 例2: 数値の範囲チェック \*/  
int x = 5, y = 15, z = 25;  
  
/\* 複雑な条件での優先順位 \*/  
if (x < 10 || y > 20 && z < 30) {  
 /\* 評価順序: (x < 10) || ((y > 20) && (z < 30)) \*/  
 /\* x < 10 が真なので、全体が真 \*/  
}  
  
/\* より明確な書き方（推奨） \*/  
if ((x < 10) || (y > 20 && z < 30)) {  
 /\* 意図が一目で分かる \*/  
}

##### 3つ以上の論理演算子を組み合わせる場合

/\* 複雑な条件式の例 \*/  
int a = 1, b = 0, c = 1, d = 0;  
  
/\* 括弧なしの場合 \*/  
if (a || b && c || d)  
 /\* 評価順序: a || (b && c) || d \*/  
 /\* 左から右へ: (a || (b && c)) || d \*/  
  
/\* 推奨される書き方1：意図を明確にする \*/  
if (a || (b && c) || d) {  
 /\* && の優先順位を明示 \*/  
}  
  
/\* 推奨される書き方2：複雑な条件を分割 \*/  
int condition1 = a || d; /\* 単純な OR 条件 \*/  
int condition2 = b && c; /\* AND 条件 \*/  
if (condition1 || condition2) { /\* 最終的な判定 \*/  
 /\* 各条件の意味が明確 \*/  
}

##### 否定演算子（!）との組み合わせ

否定演算子は論理演算子よりも優先順位が高いことに注意が必要です：

/\* 否定演算子の優先順位 \*/  
if (!a && b) /\* (!a) && b として評価 \*/  
if (!(a && b)) /\* 全体を否定したい場合は括弧が必要 \*/  
  
/\* 実用例：ログイン状態のチェック \*/  
int is\_logged\_in = 0;  
int is\_guest = 1;  
  
if (!is\_logged\_in && !is\_guest) {  
 /\* ログインしていない、かつゲストでもない \*/  
 printf("アクセス拒否\n");  
}  
  
if (!(is\_logged\_in || is\_guest)) {  
 /\* ログインしているか、ゲストである、のどちらでもない \*/  
 /\* 上記と同じ意味だが、ド・モルガンの法則を使用 \*/  
 printf("アクセス拒否\n");  
}

##### 可読性を高めるためのベストプラクティス

1. **常に括弧を使用する**

* /\* 良い例：意図が明確 \*/  
  if ((a > 0) && (b < 10)) { }  
  if ((x == 1) || (y == 2) || (z == 3)) { }

1. **複雑な条件は変数に分割する**

* /\* 複雑な条件 \*/  
  int has\_permission = (user->level >= 5) && (user->status == ACTIVE);  
  int is\_special\_user = (user->type == ADMIN) || (user->type == MODERATOR);  
    
  if (has\_permission || is\_special\_user) {  
   /\* 条件の意味が明確 \*/  
  }

1. **段階的な条件チェック**

* /\* ネストした if 文で意図を明確にする \*/  
  if (file != NULL) {  
   if (file->size > 0 && file->size < MAX\_SIZE) {  
   if (has\_read\_permission(file) || is\_owner(file)) {  
   /\* 各段階の条件が明確 \*/  
   }  
   }  
  }

##### 演算子優先順位の完全な順序

論理演算に関連する演算子の優先順位（高い順）：

1. ! ~ ++ -- （単項演算子）
2. \* / % （算術演算子）
3. + - （算術演算子）
4. < <= > >= （関係演算子）
5. == != （等価演算子）
6. && （論理AND）
7. || （論理OR）
8. ?: （条件演算子）
9. = += -= など（代入演算子）

この優先順位を覚えることも重要ですが、実際のコードでは括弧を使って意図を明確にすることがより重要です。 ### ビット演算子（概要）

ビット演算子は、数値をビット（0と1）のレベルで直接操作する演算子です。ここでは基本的な演算子の一覧のみを示します。

#### ビット演算子の種類

| 演算子 | 意味 | 例 | 説明 |
| --- | --- | --- | --- |
| & | ビットAND | a & b | 対応するビットが両方1の場合1 |
| | | ビットOR | a | b | 対応するビットのいずれかが1の場合1 |
| ^ | ビットXOR | a ^ b | 対応するビットが異なる場合1 |
| ~ | ビット反転 | ~a | 各ビットを反転（0→1、1→0） |
| << | 左シフト | a << 2 | ビットを左に2つシフト |
| >> | 右シフト | a >> 1 | ビットを右に1つシフト |

#### 基本的な例

unsigned char a = 5; /\* 00000101 (2進数) \*/  
unsigned char b = 3; /\* 00000011 (2進数) \*/  
  
printf("a & b = %d\n", a & b); /\* 1 (00000001) \*/  
printf("a | b = %d\n", a | b); /\* 7 (00000111) \*/  
printf("a ^ b = %d\n", a ^ b); /\* 6 (00000110) \*/  
printf("~a = %d\n", (unsigned char)~a); /\* 250 (11111010) \*/

#### 注意点

* ビット演算子は比較演算子より優先順位が低いので括弧に注意
* 符号付き整数の右シフトは実装依存

**詳細な学習**: ビット演算の詳細、ビットマスク、ビットフィールド、実践的な活用方法については、[第12章: ビット操作とビットフィールド](../bit-operations/README.md)で詳しく説明しています。

### 条件演算子（三項演算子）

条件に基づいて値を選択する演算子です。

条件 ? 真の場合の値 : 偽の場合の値

int a = 3, b = 7;  
int max = (a > b) ? a : b; /\* b が大きいので max = 7 \*/  
printf("大きい方: %d\n", max);

**C99版での詳細:** [conditional\_operator\_c99.c](solutions/conditional_operator_c99.c)

### sizeof演算子

データ型や変数のサイズを取得する演算子です。

printf("int のサイズ: %lu バイト\n", (unsigned long)sizeof(int));  
printf("double のサイズ: %lu バイト\n", (unsigned long)sizeof(double));  
int arr[10];  
printf("配列のサイズ: %lu バイト\n", (unsigned long)sizeof(arr));

### 演算子の優先順位

演算子には優先順位があり、計算の順序に影響します。

| 優先順位 | 演算子 | 結合規則 |
| --- | --- | --- |
| 1 | () [] -> . | 左から右 |
| 2 | ! ~ ++ -- + - \* & sizeof (型) | 右から左 |
| 3 | \* / % | 左から右 |
| 4 | + - | 左から右 |
| 5 | << >> | 左から右 |
| 6 | < <= > >= | 左から右 |
| 7 | == != | 左から右 |
| 8 | & | 左から右 |
| 9 | ^ | 左から右 |
| 10 | \| | 左から右 |
| 11 | && | 左から右 |
| 12 | \|\| | 左から右 |
| 13 | ?: | 右から左 |
| 14 | = += -= \*= /= %= &= ^= \|= <<= >>= | 右から左 |
| 15 | , | 左から右 |

#### 優先順位の例

int result = 2 + 3 \* 4; /\* 結果: 14 (乗算が先) \*/  
int result = (2 + 3) \* 4; /\* 結果: 20 (括弧が先) \*/

**C99版での詳細:** [precedence\_demo\_c99.c](examples/precedence_demo_c99.c)

## 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 operators\_demo  
# C99版をコンパイル  
make operators\_demo\_c99  
# 全て実行  
make run-all  
# クリーンアップ  
make clean

## 規格による違い

### C90での制限事項

* ビット演算子は整数型に対してのみ使用可能
* 論理演算子の結果は0または1
* ブロック先頭でのみ変数宣言可能

### C99以降の拡張

* \_Bool型の追加により、論理値の取り扱いが明確化
* 複素数型に対する演算子の追加
* forループ内での変数宣言が可能
* インライン関数の使用が可能

## よくある間違い

### 1. 代入と比較の混同

/\* NG: 代入を条件文で使用 \*/  
if (a = 5) /\* 常に真になる \*/  
 /\* ... \*/  
/\* OK: 比較演算子を使用 \*/  
if (a == 5) /\* a が 5 と等しいかチェック \*/  
 /\* ... \*/

### 2. 整数除算の結果

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

### 3. 演算子の優先順位

/\* NG: 意図しない計算順序 \*/  
int result = a + b \* c + d; /\* b \* c が先に計算される \*/  
/\* OK: 括弧で明示 \*/  
int result = (a + b) \* (c + d);

### 4. ビット演算の優先順位

/\* NG: 期待しない結果 \*/  
if (flags & MASK == 1) /\* flags & (MASK == 1) として評価される \*/  
/\* OK: 括弧で明示 \*/  
if ((flags & MASK) == 1) /\* 正しい評価順序 \*/

## 次の章へ

演算子と式を理解したら、[制御構造（条件分岐）](../control-if/README.md) に進んでください。

## 参考資料

* [C言語演算子リファレンス](https://ja.cppreference.com/w/c/language/operator_precedence)
* [ビット演算詳細](https://ja.cppreference.com/w/c/language/operator_arithmetic)
* [演算子優先順位表](https://ja.cppreference.com/w/c/language/operator_precedence)

# 条件分岐

## 対応C規格

* **主要対象:** C90
* **学習内容:** if文、else文、switch文、条件演算子による分岐処理

## 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。

* if文を使った条件分岐ができる
* else if文で複数条件を処理できる
* switch文による多分岐処理ができる
* 条件演算子を適切に使える
* ネストした条件分岐を理解する

## 概要と詳細

### 条件分岐の世界へようこそ

私たちは日常生活で常に「もし〜なら」という判断をしています。「もし雨が降ったら傘を持つ」「もし電車が遅れたらバスで行く」など、状況に応じて行動を変えています。

プログラミングでも同じように、条件によって処理を変える必要があります。この章では、C言語で条件分岐を実現する方法を学びます。

### なぜ条件分岐が必要なのか？

条件分岐がないプログラムは、いつも同じ処理しかできません。それでは。

* ユーザーの入力に応じた処理ができない
* エラーが発生しても対処できない
* 状況に応じた最適な処理を選べない 条件分岐を使うことで、プログラムに「知能」を与えることができます。

### if文の基本（もし〜なら）

if文は最も基本的な条件分岐です。日本語の「もし〜なら〜する」をそのままプログラムで表現できます。

#### 基本構文

if (条件式) {  
 /\* 条件が真（正しい）の場合に実行される処理 \*/  
}

**重要なポイント**。

* 条件式は括弧 () で囲む
* 処理部分は波括弧 {} で囲む（1行でも推奨）
* 条件が偽（正しくない）の場合は、if文の中は実行されない

#### 実生活に例えた単純なif文

int temperature = 28;  
if (temperature > 25) {  
 printf("暑いのでエアコンをつけます\n");  
}

#### 波括弧の重要性

/\* 危険な書き方（波括弧なし） \*/  
if (score >= 80)  
 printf("合格です\n");  
 printf("おめでとう！\n"); /\* これは常に実行される！ \*/  
  
/\* 安全な書き方（波括弧あり） \*/  
if (score >= 80) {  
 printf("合格です\n");  
 printf("おめでとう！\n"); /\* 両方とも条件が真の時だけ実行 \*/  
}

### if-else文（そうでなければ）

現実の判断では「もし〜なら〜する、そうでなければ〜する」という二者択一の場面がよくあります。if-else文はまさにこの状況を表現します。

#### 基本構文

if (条件式) {  
 /\* 条件が真（正しい）の場合の処理 \*/  
} else {  
 /\* 条件が偽（正しくない）の場合の処理 \*/  
}

**ポイント**。

* どちらか一方の処理が必ず実行される
* elseは「それ以外すべて」を意味する

#### 実用的なif-else文の例

/\* 成人判定 \*/  
int age = 19;  
if (age >= 20) {  
 printf("成人です\n");  
printf("選挙権があります\n");  
} else {  
 printf("未成年です\n");  
 printf("あと%d年で成人です\n", 20 - age);  
}  
/\* パスワード認証 \*/  
int password = 1234;  
int input = 1235;  
if (password == input) {  
 printf("ログイン成功！\n");  
printf("ようこそ！\n");  
} else {  
 printf("パスワードが違います\n");  
printf("もう一度お試しください\n");  
}  
/\* 営業判定 \*/  
int hour = 18;  
if (hour >= 9 && hour < 17) {  
 printf("営業中です\n");  
} else {  
 printf("営業時間外です\n");  
}

#### よくある使い方：エラー処理

int balance = 1000; /\* 残高 \*/  
int withdraw = 1500; /\* 引き出し額 \*/  
if (balance >= withdraw) {  
 balance -= withdraw;  
 printf("引き出し成功\n");  
 printf("残高: %d円\n", balance);  
} else {  
 printf("残高不足です\n");  
 printf("現在の残高: %d円\n", balance);  
}

### if-else if文（複数の選択肢）

3つ以上の選択肢から選ぶ場面では、if-else if文を使います。上から順番に条件をチェックし、最初に真になった処理を実行します。

#### 基本構文

if (条件1) {  
 /\* 条件1が真の場合 \*/  
} else if (条件2) {  
 /\* 条件1は偽で、条件2が真の場合 \*/  
} else if (条件3) {  
 /\* 条件1,2は偽で、条件3が真の場合 \*/  
} else {  
 /\* すべての条件が偽の場合 \*/  
}

**重要な特徴**。

* 上から順番にチェックされる
* 最初に真になった部分だけ実行される
* それ以降の条件はチェックされない

#### 成績判定システムの例

int score = 85;  
printf("点数: %d点\n", score);  
if (score >= 90) {  
 printf("成績: A（優秀）\n");  
 printf("素晴らしい成績です！\n");  
} else if (score >= 80) {  
 printf("成績: B（良好）\n");  
 printf("よく頑張りました！\n");  
} else if (score >= 70) {  
 printf("成績: C（普通）\n");  
 printf("合格です\n");  
} else if (score >= 60) {  
 printf("成績: D（可）\n");  
 printf("ギリギリ合格です\n");  
} else {  
 printf("成績: F（不可）\n");  
 printf("もう少し頑張りましょう\n");  
}

#### 時間帯による挨拶の例

int hour = 14; /\* 14時（午後2時） \*/  
if (hour >= 5 && hour < 12) {  
 printf("おはようございます\n");  
} else if (hour >= 12 && hour < 17) {  
 printf("こんにちは\n");  
} else if (hour >= 17 && hour < 21) {  
 printf("こんばんは\n");  
} else {  
 printf("夜遅いですね、お疲れさまです\n");  
}

#### 順序が重要な理由

/\* 間違った順序 \*/  
int age = 15;  
if (age >= 0) {  
 printf("生まれています\n"); /\* 15歳でもここで止まる！ \*/  
} else if (age >= 13) {  
 printf("中学生以上\n"); /\* 実行されない \*/  
} else if (age >= 18) {  
 printf("成人\n"); /\* 実行されない \*/  
}

/\* 正しい順序（厳しい条件から） \*/  
if (age >= 18) {  
 printf("成人\n");  
} else if (age >= 13) {  
 printf("中学生以上\n");  
} else if (age >= 0) {  
 printf("生まれています\n");  
}

### ネストしたif文（入れ子構造）

if文の中にさらにif文を書くことで、より複雑な条件判断ができます。

/\* 天気と気温による服装アドバイス \*/  
int temperature = 25;  
int is\_raining = 0; /\* 0:晴れ、1:雨 \*/  
if (temperature >= 25) {  
 printf("暖かい日です\n");  
 if (is\_raining) {  
 printf("半袖で傘を持っていきましょう\n");  
 } else {  
 printf("半袖で大丈夫です\n");  
 }  
} else if (temperature >= 15) {  
 printf("過ごしやすい気温です\n");  
 if (is\_raining) {  
 printf("長袖で傘を持っていきましょう\n");  
 } else {  
 printf("長袖がおすすめです\n");  
 }  
} else {  
 printf("寒い日です\n");  
 if (is\_raining) {  
 printf("コートと傘が必要です\n");  
 } else {  
 printf("コートを着ていきましょう\n");  
 }  
}

#### ネストを避ける工夫

ネストが深くなると読みにくくなるので、論理演算子を使って簡潔に書くこともできます。

/\* ネストを使った場合 \*/  
if (age >= 18) {  
 if (has\_license) {  
 printf("運転できます\n");  
 }  
}

/\* 論理演算子を使った場合 \*/  
if (age >= 18 && has\_license) {  
 printf("運転できます\n");  
}

### 条件式の詳細

#### 比較演算子

int a = 5, b = 3;  
if (a == b) printf("等しい\n");  
if (a != b) printf("等しくない\n");  
if (a < b) printf("a は b より小さい\n");  
if (a <= b) printf("a は b 以下\n");  
if (a > b) printf("a は b より大きい\n");  
if (a >= b) printf("a は b 以上\n");

#### 論理演算子の組み合わせ

int age = 25;  
int income = 300000;

/\* AND演算子 \*/  
if (age >= 20 && income >= 200000) {  
 printf("ローン審査に通りました\n");  
}

/\* OR演算子 \*/  
if (age < 6 || age > 65) {  
 printf("特別料金が適用されます\n");  
}

/\* NOT演算子 \*/  
if (!(age >= 18)) {  
 printf("18歳未満です\n");  
}

### switch文

複数の値に対する分岐処理を効率的に記述できます。

#### switch文って何？

プログラムを書いていると、「この変数の値が1なら処理A、2なら処理B、3なら処理C…」というように、一つの変数の値によって多くの分岐をしたい場面があります。 これをif-else ifで書くと。

if (choice == 1) {  
 printf("処理A\n");  
} else if (choice == 2) {  
 printf("処理B\n");  
} else if (choice == 3) {  
 printf("処理C\n");  
} else if (choice == 4) {  
 printf("処理D\n");  
} else {  
 printf("その他の処理\n");  
}

このように長くなってしまいます。switch文を使うと、もっとスッキリ書けます！

#### 日常生活での例え

switch文は「自動販売機」のようなものです。

1. ボタンを押す（変数の値）
2. 押したボタンに応じて違う商品が出る（caseごとの処理）
3. 存在しないボタンを押したら何も出ない（default処理）

int button = 2; /\* 2番のボタンを押した \*/  
switch (button) {  
 case 1:  
 printf("コーラが出ました\n");  
 break;  
 case 2:  
 printf("オレンジジュースが出ました\n");  
 break;  
 case 3:  
 printf("お茶が出ました\n");  
 break;  
 default:  
 printf("そのボタンには商品がありません\n");  
 break;  
}

#### 基本構文

switch (変数または式) {  
 case 値1:  
 /\* 値1の場合の処理 \*/  
 break;  
 case 値2:  
 /\* 値2の場合の処理 \*/  
 break;  
 case 値3:  
 /\* 値3の場合の処理 \*/  
 break;  
 default:  
 /\* どの値にも一致しない場合の処理 \*/  
 break;  
}

#### 曜日判定の例

int day = 3;  
switch (day) {  
 case 1:  
 printf("月曜日\n");  
 break;  
 case 2:  
 printf("火曜日\n");  
 break;  
 case 3:  
 printf("水曜日\n");  
 break;  
 case 4:  
 printf("木曜日\n");  
 break;  
 case 5:  
 printf("金曜日\n");  
 break;  
 case 6:  
 printf("土曜日\n");  
 break;  
 case 7:  
 printf("日曜日\n");  
 break;  
 default:  
 printf("無効な曜日です\n");  
 break;  
}

#### break文の重要性

break文を忘れると、次のcaseも実行されます（フォールスルー）。 **初心者がよくやるミス：breakを忘れる** switch文で最も多いミスが「break文の書き忘れ」です。これは本当によくあるミスなので、必ず確認しましょう！

int grade = 'B';  
switch (grade) {  
 case 'A':  
 printf("優秀\n");  
 /\* break がないので次のcaseも実行される \*/  
 case 'B':  
 printf("良好\n");  
 /\* break がないので次のcaseも実行される \*/  
 case 'C':  
 printf("普通\n");  
 break;  
 default:  
 printf("要努力\n");  
 break;  
}  
/\* grade が 'B' の場合、"良好" と "普通" の両方が出力される \*/

#### 意図的なフォールスルー

時には意図的にbreakを省略することもあります。 **フォールスルーが便利な場面** 「複数の値で同じ処理をしたい」ときは、あえてbreakを書かないことで、コードを簡潔に書けます。

char ch = 'a';  
switch (ch) {  
 case 'a':  
 case 'e':  
 case 'i':  
 case 'o':  
 case 'u':  
 printf("母音です\n");  
 break;  
 default:  
 printf("子音です\n");  
 break;  
}

### 条件演算子（三項演算子）の復習

条件分岐の簡潔な書き方として条件演算子があります。

int a = 5, b = 3;  
int max;  
/\* if-else文での記述 \*/  
if (a > b) {  
 max = a;  
 } else {  
 max = b;  
}

/\* 条件演算子での記述 \*/  
max = (a > b) ? a : b;  
printf("最大値: %d\n", max);

### 条件式での注意点

初心者が陥りやすい間違いをいくつか紹介します。これらを知っておけば、デバッグ時間を大幅に短縮できます！

#### 代入と比較の混同

int x = 5;

/\* NG: 代入になってしまう \*/  
if (x = 10) { /\* ← これは間違い！ \*/  
 printf("常に実行される\n"); /\* x に 10 が代入され、常に真 \*/  
}

/\* OK: 比較演算子を使用 \*/  
if (x == 10) { /\* ← 正しい書き方 \*/  
 printf("x が 10 の場合のみ実行\n");  
}

/\* 防御的プログラミング：定数を左側に書く \*/  
if (10 == x) { /\* もし = を1つしか書かなかったらコンパイルエラーになる \*/  
 printf("x が 10 の場合のみ実行\n");  
}

#### 浮動小数点数の比較

コンピュータでは小数を完全に正確に表現できないことがあります。これは初心者にとって意外な落とし穴です。

double d = 0.1 + 0.2;

/\* NG: 浮動小数点の誤差で期待通りにならない可能性 \*/  
if (d == 0.3) {  
 printf("等しい\n"); /\* 実行されないかも！ \*/  
}

/\* OK: 誤差を考慮した比較 \*/  
if (d >= 0.299999 && d <= 0.300001) {  
 printf("ほぼ等しい\n");  
}

/\* より実践的な方法：許容誤差を定義 \*/  
#define EPSILON 0.00001  
if (fabs(d - 0.3) < EPSILON) {  
 printf("実質的に等しい\n");  
}

**なぜこんなことが起きるの？** コンピュータは2進数で計算するため、10進数の0.1や0.2を正確に表現できません。人間にとっての「0.1」は、コンピュータにとっては「0.1に限りなく近い値」なのです。

## 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 if\_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

* switch文の case ラベルは整数定数のみ
* 複合文内での変数宣言は先頭のみ

### C99以降の拡張

* switch文でlong long型が使用可能
* ブロック内での変数宣言位置の制限緩和

## よくある間違い

### 1. セミコロンの位置

/\* NG: if文の後にセミコロン \*/  
if (condition);  
{  
 printf("常に実行される\n"); /\* 条件に関係なく実行 \*/  
}

/\* OK: 正しい記述 \*/  
if (condition) {  
 printf("条件が真の場合のみ実行\n");

### 1. switch文でのbreak忘れ

/\* NG: break忘れでフォールスルー \*/  
switch (value) {  
 case 1:  
 printf("1です\n");  
 /\* break; を忘れると次のcaseも実行される \*/  
 case 2:  
 printf("2です\n");  
 break;  
}

/\* OK: 適切なbreak文 \*/  
switch (value) {  
 case 1:  
 printf("1です\n");  
 break;  
 case 2:  
 printf("2です\n");  
 break;  
}

### 1. 複雑な条件式での括弧不足

/\* NG: 意図しない優先順位 \*/  
if (a == 1 || b == 2 && c == 3)   
 /\* && が || より優先される \*/

/\* OK: 括弧で明示 \*/  
if ((a == 1) || (b == 2 && c == 3))   
 /\* 意図が明確 \*/

## 次の章へ

条件分岐を理解したら、[制御構造（ループ）](../control-loop/README.md) に進んでください。

## 参考資料

* [if文詳細](https://ja.cppreference.com/w/c/language/if)
* [switch文詳細](https://ja.cppreference.com/w/c/language/switch)
* [条件演算子](https://ja.cppreference.com/w/c/language/operator_other)

# 繰り返し処理

## 対応C規格

* **主要対象:** C90
* **学習内容:** for文、while文、do-while文、break文、continue文、ネストしたループ

## 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。

* for文を使った繰り返し処理ができる
* while文とdo-while文の違いを理解する
* break文とcontinue文を適切に使える
* ネストしたループを理解して活用できる
* ループを使った実践的なプログラムを作成できる

## 概要と詳細

### 繰り返し処理とは？

プログラミングでは、同じ処理を何度も繰り返したい場面がたくさんあります。例えば。

* 1から100までの数を全て表示する
* 成績データを100人分処理する
* ゲームのメインループ（ユーザーが終了するまで続ける）
* ファイルのデータを最後まで読み込む

これらを一つ一つ手作業で書くのは大変です。そこで「ループ（繰り返し）」を使います！

#### なぜループが重要なの？

コンピュータの強みは「高速で正確な繰り返し処理」です。人間が苦手とする単調な繰り返し作業を、コンピュータは疲れることなく実行できます。この能力を活用するのがループです。

### for文

最も一般的なループ文で、初期化・条件・更新を一箇所で記述できます。

#### for文の仕組みを理解しよう

for文は「カウンター付きの繰り返し」と考えると分かりやすいです。

例えば、ラジオ体操を10回行う場合を考えてみましょう。

1. 最初は1回目から始める（初期化）
2. 10回以下なら続ける（条件）
3. 1回終わったら回数を増やす（更新）

この流れをコードで表現すると、次のようになります。

for (int i = 1; i <= 10; i++) {  
 printf("%d回目のラジオ体操\n", i);  
}

#### 基本構文

for (初期化; 条件式; 更新式) {  
 /\* 繰り返し実行される文 \*/  
}

#### 基本的な使用例

#include <stdio.h>  
int main(void)  
{  
 int i;  
   
 /\* 基本的なfor文 \*/  
 printf("1から5まで: ");  
 for (i = 1; i <= 5; i++) {  
 printf("%d ", i);  
 }  
 printf("\n");  
   
 return 0;  
}

#### for文の詳細動作

for文がどのように動作するか、ステップごとに見てみましょう。

for (i = 0; i < 5; i++) {  
 printf("i = %d\n", i);  
}  
  
/\* 上記は以下と同等 \*/  
i = 0; /\* ① 初期化（1回のみ実行） \*/  
while (i < 5) { /\* ② 条件チェック \*/  
 printf("i = %d\n", i); /\* ③ ループ本体 \*/  
 i++; /\* ④ 更新式 \*/  
}

**実行の流れ：**

1. 初期化 → i = 0
2. 条件チェック → 0 < 5は真、続行
3. ループ本体実行 → “i = 0”を表示
4. 更新 → i = 1
5. 条件チェック → 1 < 5は真、続行
6. …これを繰り返し…
7. 更新 → i = 5
8. 条件チェック → 5 < 5は偽、ループ終了

#### さまざまなfor文のパターン

for文は非常に柔軟で、色々な書き方ができます。

/\* 逆順ループ（カウントダウン） \*/  
for (i = 10; i >= 1; i--) {  
 printf("%d ", i);  
}  
printf("発射！\n");  
  
/\* 2つずつ増加（偶数だけ表示） \*/  
for (i = 0; i <= 20; i += 2) {  
 printf("%d ", i);  
}  
  
/\* 複数変数の制御（両端から中央へ） \*/  
for (i = 0, j = 10; i < j; i++, j--) {  
 printf("i=%d, j=%d\n", i, j);  
}  
  
/\* 無限ループ（条件式を省略） \*/  
for (;;) {  
 /\* Ctrl+Cで止めるまで続く \*/  
 printf("無限ループ中...\n");  
}

**初心者のためのヒント：**

* 最初は基本形（for (i = 0; i < n; i++)）をマスターしましょう
* 複雑なfor文は読みにくくなりがちなので、シンプルに保つことが大切です

### while文

条件が真である間、繰り返し処理を実行します。

#### while文って何？

while文は「〜である間、繰り返す」という意味です。条件が満たされている限り、処理を続けます。 **日常生活での例：**

* お風呂にお湯を入れる → 「満タンになるまで」続ける
* 宿題をする → 「全部終わるまで」続ける
* 電車を待つ → 「電車が来るまで」待つ これらは全て「ある条件が満たされるまで続ける」パターンです。

#### 基本構文

while (条件式) {  
 /\* 繰り返し実行される文 \*/  
}

#### while文の使用例

#include <stdio.h>  
int main(void)  
{  
 int count = 1;  
   
 printf("while文の例: ");  
 while (count <= 5) {  
 printf("%d ", count);  
 count++;  
 }  
 printf("\n");  
   
 return 0;  
}

#### while文の実践例

while文は「回数が分からない繰り返し」に最適です。

/\* ユーザー入力の処理 \*/  
int number;  
printf("正の数を入力してください（0で終了）: ");  
while (scanf("%d", &number) == 1 && number > 0) {  
 printf("入力された数: %d\n", number);  
 printf("次の数を入力してください（0で終了）: ");  
}  
printf("プログラムを終了します。\n");

**for文とwhile文の使い分け：**

* **for文**：繰り返し回数が分かっている場合（例：10回繰り返す）
* **while文**：条件で終了を判断する場合（例：ユーザーが0を入力するまで）

### do-while文

最低1回は実行され、その後条件をチェックするループです。

#### do-while文の特徴

while文との大きな違いは「条件チェックのタイミング」です。

* **while文**：最初に条件をチェック（条件が偽なら一度も実行されない）
* **do-while文**：最後に条件をチェック（必ず一度は実行される） **日常生活での例：** レストランでの注文を考えてみましょう。

1. メニューを見せる（必ず一度は見せる）
2. 注文を聞く
3. 「他に注文はありますか？」と聞く
4. 「はい」なら1に戻る、「いいえ」なら終了 これがdo-while文の考え方です！

#### 基本構文

do {  
 /\* 最低1回は実行される文 \*/  
} while (条件式);

#### do-while文の使用例

#include <stdio.h>  
int main(void)  
{  
 int num;  
   
 do {  
 printf("正の数を入力（0で終了）: ");  
 scanf("%d", &num);  
 if (num > 0) {  
 printf("入力値: %d\n", num);  
 }  
 } while (num != 0);  
   
 printf("終了します\n");  
 return 0;  
}

### break文とcontinue文

ループの流れを制御する特別な文です。

#### 途中でループを抜けたいとき

プログラムを書いていると、「ある条件を満たしたらループを抜けたい」という場面があります。そんなときに使うのがbreak文とcontinue文です。 **日常生活での例：**

* **break**：探し物が見つかったら探すのをやめる
* **continue**：嫌いな野菜は飛ばして次の料理を食べる

#### break文

ループを強制的に終了します。

#include <stdio.h>  
int main(void)  
{  
 int i;  
   
 for (i = 1; i <= 10; i++) {  
 if (i == 5) {  
 printf("\n5で終了  
");  
 break;  
 }  
 printf("%d ", i);  
 }  
   
 return 0;  
}

#### continue文

現在の繰り返しをスキップして、次の繰り返しに進みます。

#include <stdio.h>  
int main(void)  
{  
 int i;  
   
 printf("奇数のみ: ");  
 for (i = 1; i <= 10; i++) {  
 if (i % 2 == 0) {  
 continue; /\* 偶数をスキップ \*/  
 }  
 printf("%d ", i);  
 }  
 printf("\n");  
   
 return 0;  
}

### ネストしたループ

ループの中にさらにループを含む構造です。

#### 二重ループを理解しよう

ネストしたループは「ループの中にループ」がある構造です。 **日常生活での例：** 学校の時間割を考えてみましょう。

* 外側のループ：月曜日から金曜日まで（5回）
* 内側のループ：1時間目から6時間目まで（6回）
* 合計：5日 × 6時間 = 30コマ これがネストしたループの考え方です！

#### 二重ループの例

#include <stdio.h>  
int main(void)  
{  
 int i, j;  
   
 /\* 九九の表（3x3のみ） \*/  
 for (i = 1; i <= 3; i++) {  
 for (j = 1; j <= 3; j++) {  
 printf("%2d ", i \* j);  
 }  
 printf("\n");  
 }  
   
 return 0;  
}

#### 三角形パターンの出力

#include <stdio.h>  
int main(void)  
{  
 int i, j;  
 int n = 5;  
   
 /\* 上向き三角形 \*/  
 for (i = 1; i <= n; i++) {  
 for (j = 1; j <= i; j++) {  
 printf("\* ");  
 }  
 printf("\n");  
 }  
   
 return 0;  
}

### ループでのbreak・continueの応用

#### ネストしたループでのbreak

/\*  
 \* ネストしたループでのbreak文の動作  
 \*   
 \* breakは最も内側のループのみを抜ける  
 \*/  
#include <stdio.h>  
int main(void)  
{  
 int i, j;  
   
 printf("===== ネストしたループでのbreak =====\n");  
   
 /\* 例1: 内側のループでbreak \*/  
 printf("\n例1: 内側のループでbreak\n");  
 for (i = 1; i <= 3; i++) {  
 printf("外側ループ: i = %d\n", i);  
 for (j = 1; j <= 5; j++) {  
 if (j == 3) {  
 printf(" 内側でbreak (j = %d)\n", j);  
 break; /\* 内側のループのみ終了 \*/  
 }  
 printf(" 内側ループ: j = %d\n", j);  
 }  
 printf("外側ループは継続\n");  
 }  
   
 /\* 例2: 特定の条件で二重ループを抜ける \*/  
 printf("\n例2: 九九の表で答えが50を超えたら終了\n");  
 int found = 0;  
 for (i = 1; i <= 9 && !found; i++) {  
 for (j = 1; j <= 9; j++) {  
 int result = i \* j;  
 if (result > 50) {  
 printf("%d × %d = %d で終了\n", i, j, result);  
 found = 1; /\* フラグを立てる \*/  
 break; /\* 内側のループを抜ける \*/  
 }  
 printf("%d×%d=%2d ", i, j, result);  
 }  
 if (!found) printf("\n");  
 }  
   
 return 0;  
}

#### continueを使った複雑な条件制御

/\*  
 \* continueを使った複雑な条件制御  
 \*   
 \* 特定の条件をスキップしながら処理を継続  
 \*/  
#include <stdio.h>  
int main(void)  
{  
 int i, j;  
   
 printf("===== continueを使った条件制御 =====\n");  
   
 /\* 例1: 3の倍数と5の倍数をスキップ \*/  
 printf("\n1から20で3の倍数と5の倍数以外:\n");  
 for (i = 1; i <= 20; i++) {  
 if (i % 3 == 0 || i % 5 == 0) {  
 continue; /\* 次の繰り返しへ \*/  
 }  
 printf("%d ", i);  
 }  
 printf("\n");  
   
 /\* 例2: ネストしたループでのcontinue \*/  
 printf("\n九九の表（偶数の段は計算をスキップ）:\n");  
 for (i = 1; i <= 9; i++) {  
 if (i % 2 == 0) {  
 printf("%d の段: スキップ\n", i);  
 continue; /\* 偶数の段をスキップ \*/  
 }  
 printf("%d の段: ", i);  
 for (j = 1; j <= 9; j++) {  
 printf("%2d ", i \* j);  
 }  
 printf("\n");  
 }  
   
 return 0;  
}

### 無限ループ ♾

意図的に終了しないループを作成することもあります。

#### 無限ループが必要な場面

「無限ループ」と聞くと怖く感じるかもしれませんが、実は多くのプログラムで使われています。

* ゲームのメインループ（ユーザーが終了するまで続く）
* サーバープログラム（常に要求を待ち受ける）
* 組み込みシステム（電源が入っている間ずっと動作） **重要：** 無限ループを作るときは、必ず脱出方法（breakやプログラム終了条件）を用意しましょう！

#### 無限ループの作成方法

/\* 方法1: for文 \*/  
for (;;) {  
 /\* 無限ループ \*/  
 if (条件) break;  
}  
  
/\* 方法2: while文 \*/  
while (1) {  
 /\* 無限ループ \*/  
 if (条件) break;  
}  
  
/\* 方法3: do-while文 \*/  
do {  
 /\* 無限ループ \*/  
 if (条件) break;  
} while (1);

#### 無限ループの実用例

#include <stdio.h>  
int main(void)  
{  
 int choice;  
   
 while (1) {  
 printf("\n1. 続行\n");  
 printf("0. 終了\n");  
 printf("選択: ");  
 scanf("%d", &choice);  
   
 if (choice == 0) {  
 break;  
 }  
   
 printf("処理を続行します\n");  
 }  
   
 printf("終了しました\n");  
 return 0;  
}

### ループの最適化とベストプラクティス

初心者がループを書く際に気をつけるべきポイントを紹介します。

#### 効率的なループ

/\* NG: 毎回strlen()を呼び出し \*/  
for (i = 0; i < strlen(str); i++) {  
 /\* 処理 \*/  
}  
  
/\* OK: 長さを事前に計算 \*/  
len = strlen(str);  
for (i = 0; i < len; i++) {  
 /\* 処理 \*/  
}

#### ループ変数の適切な使用

#include <stdio.h>  
int main(void)  
{  
 int i; /\* C90では先頭で宣言 \*/  
   
 for (i = 0; i < 5; i++) {  
 printf("i = %d\n", i);  
 }  
   
 /\* ループ後もiは有効 \*/  
 printf("最終的なi = %d\n", i);  
   
 return 0;  
}

## 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 loop\_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

* ループ変数は関数の先頭で宣言する必要がある
* for文内での変数宣言は不可

### C99以降の拡張

* for文内での変数宣言が可能
* for (int i = 0; i < 10; i++) /\* C99以降 \*/  
   /\* 処理 \*/

## よくある間違い

初心者がループで陥りやすいミスと、その対策を紹介します。

### 1. 無限ループの作成

/\* NG: 終了条件の記述ミス \*/  
for (i = 10; i > 0; i++) /\* 最初から条件が偽 \*/  
 /\* 実行されない \*/

/\* NG: 更新式の記述ミス \*/  
for (i = 0; i < 10; i--) /\* 負の方向に進む \*/  
 /\* 無限ループ \*/

### 2. セミコロンの位置

/\* NG: for文の後にセミコロン \*/  
for (i = 0; i < count; i++); /\* 空文が実行される \*/  
 printf("%d\n", i); /\* ループ外で実行 \*/

/\* OK: 正しい記述 \*/  
for (i = 0; i < count; i++)   
 printf("%d\n", i);

### 3. ループ変数の範囲外アクセス

配列を使うときに特に注意が必要です。

/\* NG: 配列の範囲を超える \*/  
int arr[10];  
for (i = 0; i <= 10; i++) { /\* i=10で範囲外！ \*/  
 arr[i] = i; /\* 配列は0～9までしかない \*/  
}  
/\* OK: 正しい範囲 \*/  
for (i = 0; i < 10; i++) {  
 arr[i] = i;  
}

**覚え方：** 配列のサイズがNなら、インデックスは0からN-1まで！

## 次の章へ

ループ処理を理解したら、[配列（基本編）](../arrays-basics/README.md) に進んでください。

## 参考資料

* [for文詳細](https://ja.cppreference.com/w/c/language/for)
* [while文詳細](https://ja.cppreference.com/w/c/language/while)
* [break・continue文](https://ja.cppreference.com/w/c/language/break)

# 終わりに

基礎編の学習お疲れさまでした。ここまでで、C言語の基本的な文法と制御構造を学びました。

## 次のステップ

基礎編を修了したら、応用編に進んでください。応用編では以下の内容を学習します：

* 配列とポインタ
* 文字列処理
* 関数
* ビット操作

プログラミングの学習は継続が大切です。焦らず、着実に理解を深めていきましょう。