C言語 学習教材

筑波大学 システム情報系 三谷純
最終更新日 2023/4/19

　　図表出典：三谷純著『C 言語 ゼロからはじめるプログラミング』（翔泳社 刊）

本資料の位置づけ

本資料は

『C言語 ゼロからはじめるプログラミング』

を専門学校・大学・企業などで教科書として採用された教員・指導員を対象に、教科書の内容を解説するための副教材として作られています。

上記に該当する場合は、自由にご使用ください。授業の進め方などに応じて、改変していただいて結構です。※ このページを削除して構いません

ただし、民間企業が商用、ビジネス目的で利用する際には別途許諾が必要ですので、著者までご連絡ください。

出版社　：翔泳社
発売日　：2022/9/20
ISBN　：9784798174655

はじめに

プログラミングを学ぶ意義（1/2）

- ソフトウェアを使う立場から、作る立場へ
  - パソコン、スマートフォンなどで動作するアプリ
  - TV、エアコン、洗濯機などの電子機器の制御
  - 電子決済、電子申請、各種のシステム

- 個人での簡単な開発
  - 電卓・Excelの一歩先
  - データ処理・データ解析
  - 個人用途のアプリ開発

- スキルアップ
  - 情報処理関係の資格取得
  - プログラミングコンテスト

プログラミングを学ぶ意義（2/2）

- 実際に開発する立場になる予定が無くても
  - アルゴリズム的思考（ものごとを処理する手順に関する合理的な考え方）が身につく
  - ソフトウェアの開発の様子、動作原理がわかることによる広い視野の獲得
  - 専門用語の理解、ITエンジニアとのコミュニケーション
  - 将来にプログラミングを独習したくなったときに役立つ（異なるプログラミング言語でも考え方の基本は同じ）

この講義で学ぶこと

- プログラミング言語に拠らない、プログラミング全般の基礎知識
- プログラミングに関する基本的な用語と考え方
- C言語というプログラミング言語活用の基礎

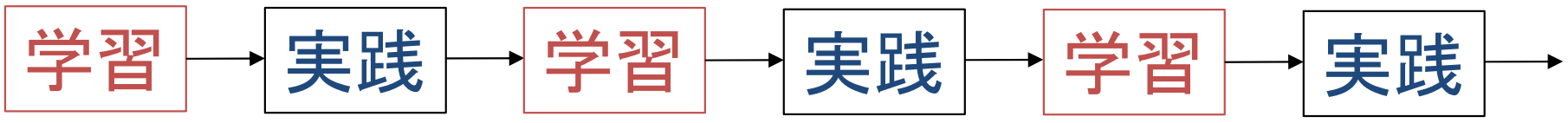
よりよく学ぶために

- 実際に手を動かしてプログラムコードを入力する、一部を変更して実験する

実際に試してみよう

- Webで公開されているプログラムコードを動かしてみる、一部を変更して実験する

- プログラミングコンテストに参加してみる
  （モチベーションアップ、実力向上、他者のコードからの学び）



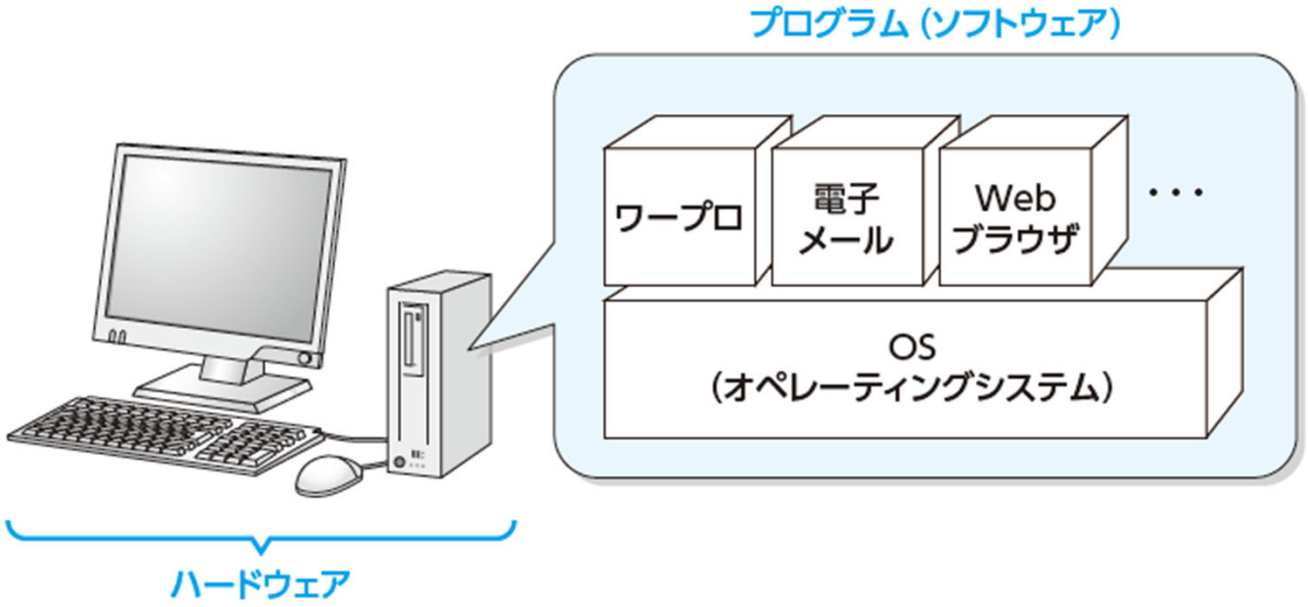
全体の流れ

1. C言語に触れる
2. C言語の基本
3. 条件分岐と繰り返し
4. 関数
5. アドレスとポインタ
6. 文字列の扱いと構造体
7. 一歩進んだC言語プログラミング
8. データ構造とアルゴリズム

第1章 C言語に触れる

プログラムとは

- コンピュータに命令を与えるものが「プログラム」
- プログラムを作成するための専用言語が「プログラミング言語」
- その中の1つに「C言語」がある

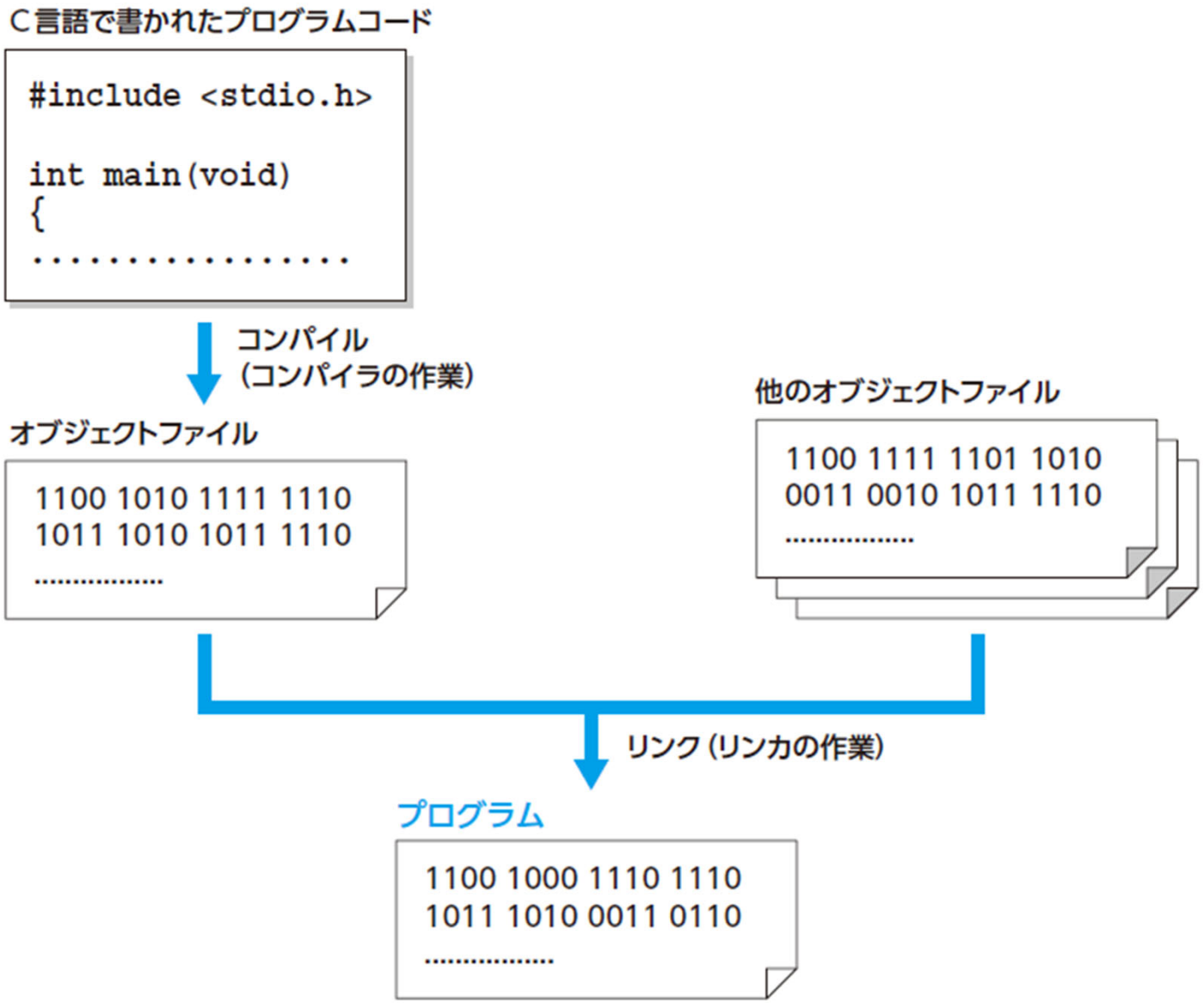


さまざまなプログラミング言語

- C　歴史のある言語、OS開発、組み込みプログラム
- C++　C言語の後継、オブジェクト指向
- C#　C++言語の後継、米マイクソロソフト
- Perl　スクリプト言語、手軽な開発
- PHP　サーバサイド、Webページ生成
- Java　オブジェクト指向、大規模システム
- JavaScript　ブラウザで動作、動的なWebページ
- Python　修得が容易、機械学習分野で普及

※ **C言語の特徴**
メモリーに直接アクセスできる。
うまく扱えば、効率的で高速の動作するプログラムを作成できる。
他の言語の基礎となる言語

プログラムコードが実行されるまで



C言語のプログラムコード

C言語の最も基本的なプログラムコードの形

```
#include <stdio.h>

int main(void)
{
    命令文
    return 0;
}
```

- 半角英数と記号で記述する
- 人が読んで理解できるテキスト形式

C言語のプログラムコード

「こんにちは」という文字列を出力するプログラムコード

```
#include <stdio.h>

int main(void)
{
    printf("こんにちは");
    return 0;
}
```

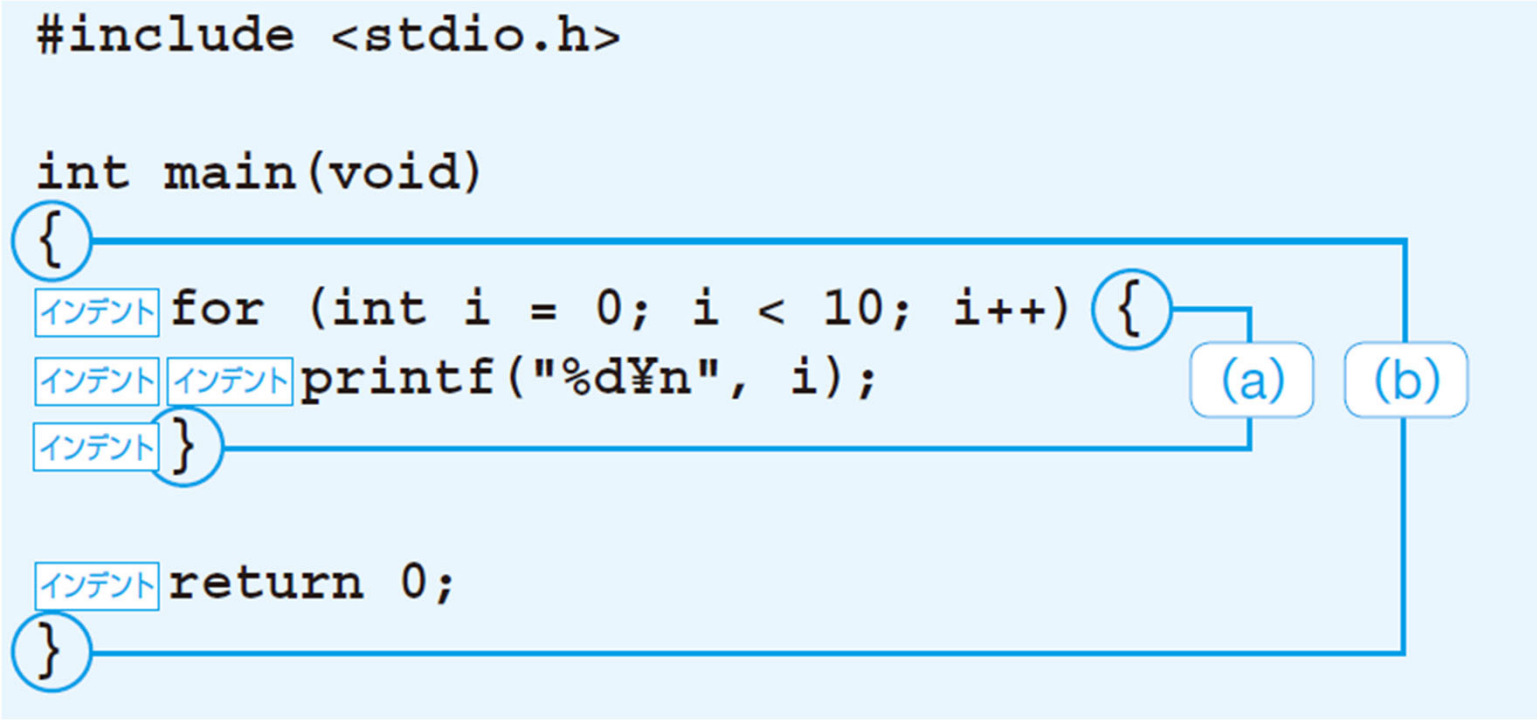
プログラムで使用する機能を読み込みます

プログラムの入り口です

「こんにちは」という文字列を表示させる命令文です

- 命令文の末尾にはセミコロン（;）をつける
- 空白や改行は好きな場所に入れてかまわない
- プログラムコードの中の大文字と小文字は区別される

ブロックとインデント



ブロック： { と } で囲まれた範囲
ブロックの中にブロックがあることも（**ブロックのネスト**）

インデント： プログラムコードを見やすくするために入れる行頭の空白
ブロックの階層の深さにあわせてインデントの数を変える
（最近のエディタは自動で調整してくれる）

※ インデントは無くてもプログラムに影響しない

コメント文

```
#include <stdio.h>
/*
     「こんにちは」という文字列を画面に表示するプログラム
     作成日:2022年6月1日
     作成者:三谷純
*/

int main(void)
{
     // 画面へメッセージを出力する
     printf("こんにちは");

     return 0;
}
```

複数行のコメント文

1行のコメント文

コメント文

- プログラムコードの中に記したメモ書き
- 1 行のコメント文には // を使用
- 複数行のコメント文は /* と */ で囲む。
- コンパイラに無視される（プログラムの動作には影響しない）

プログラムの作成と実行

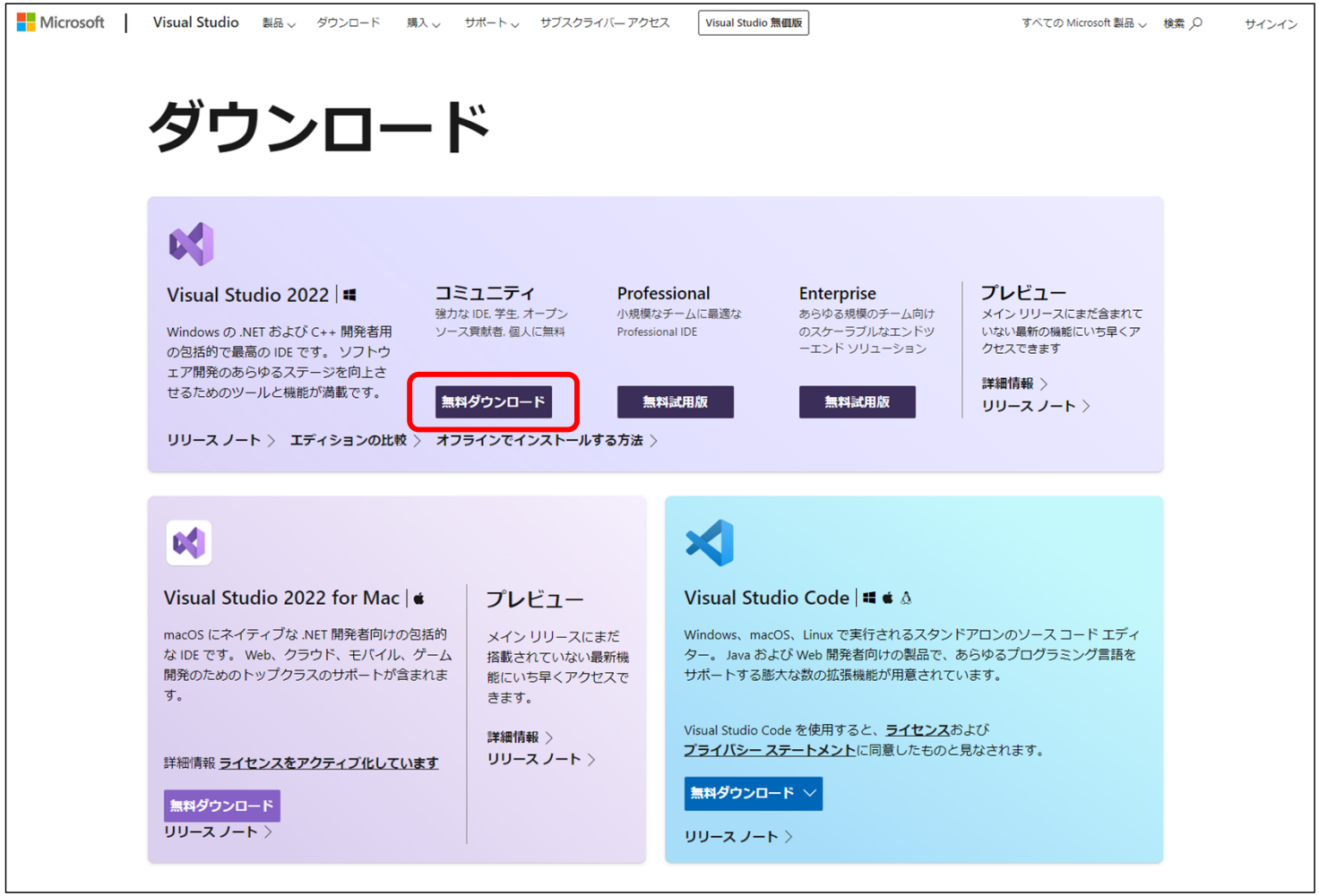
Visual Studio とは

- Visual Studio は以下のものを含む統合開発環境の1つ
  - **エディタ**：プログラムコードを記述する
  - **コンパイラ**：コンパイルを行う（オブジェクトファイルを生成する）
  - **リンカ**：リンクを行う（実行ファイルを生成する）

Visual Studio のインストール（Windows）

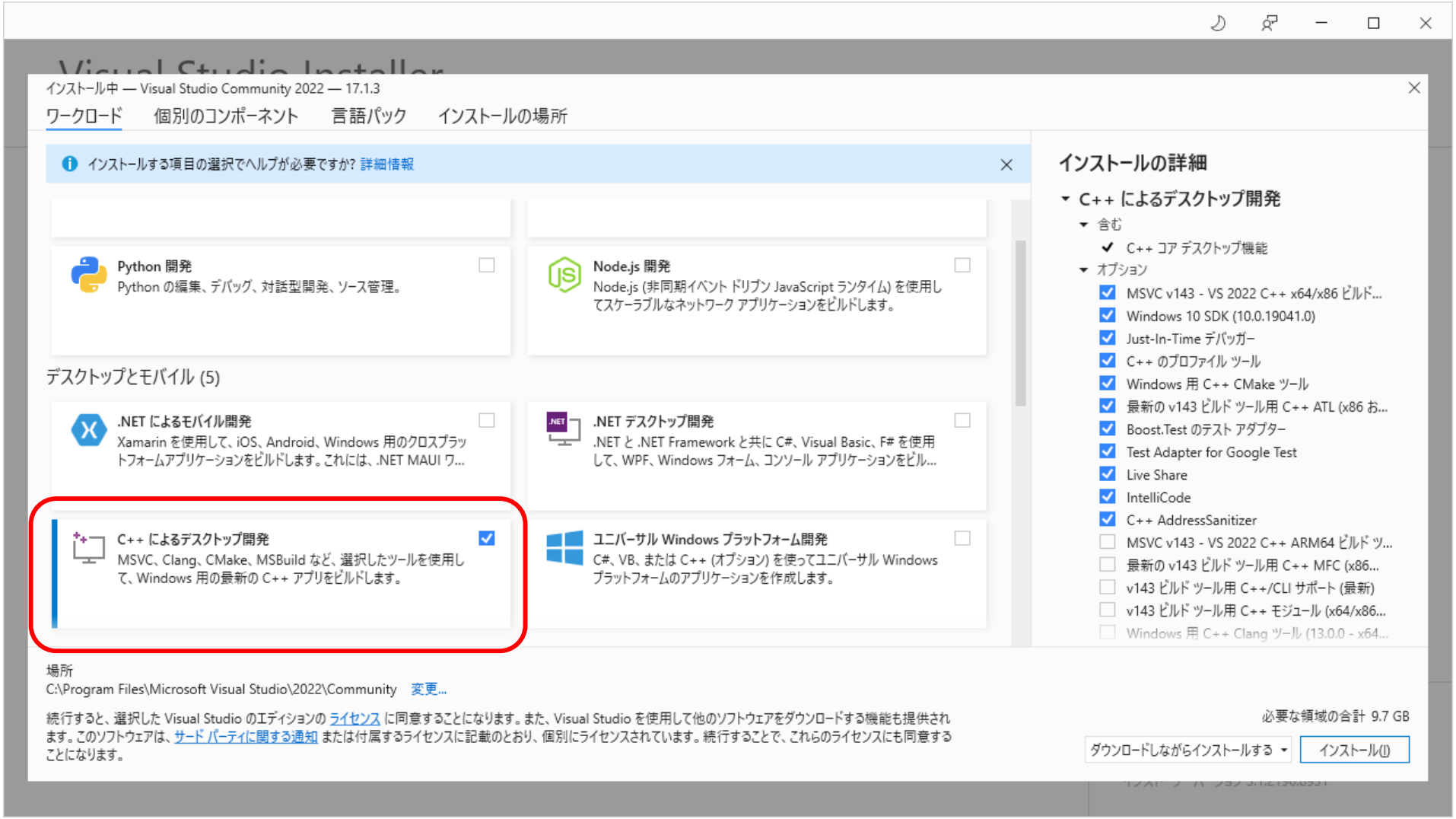
- Visual Studio の入手

  https://visualstudio.microsoft.com/ja/downloads/

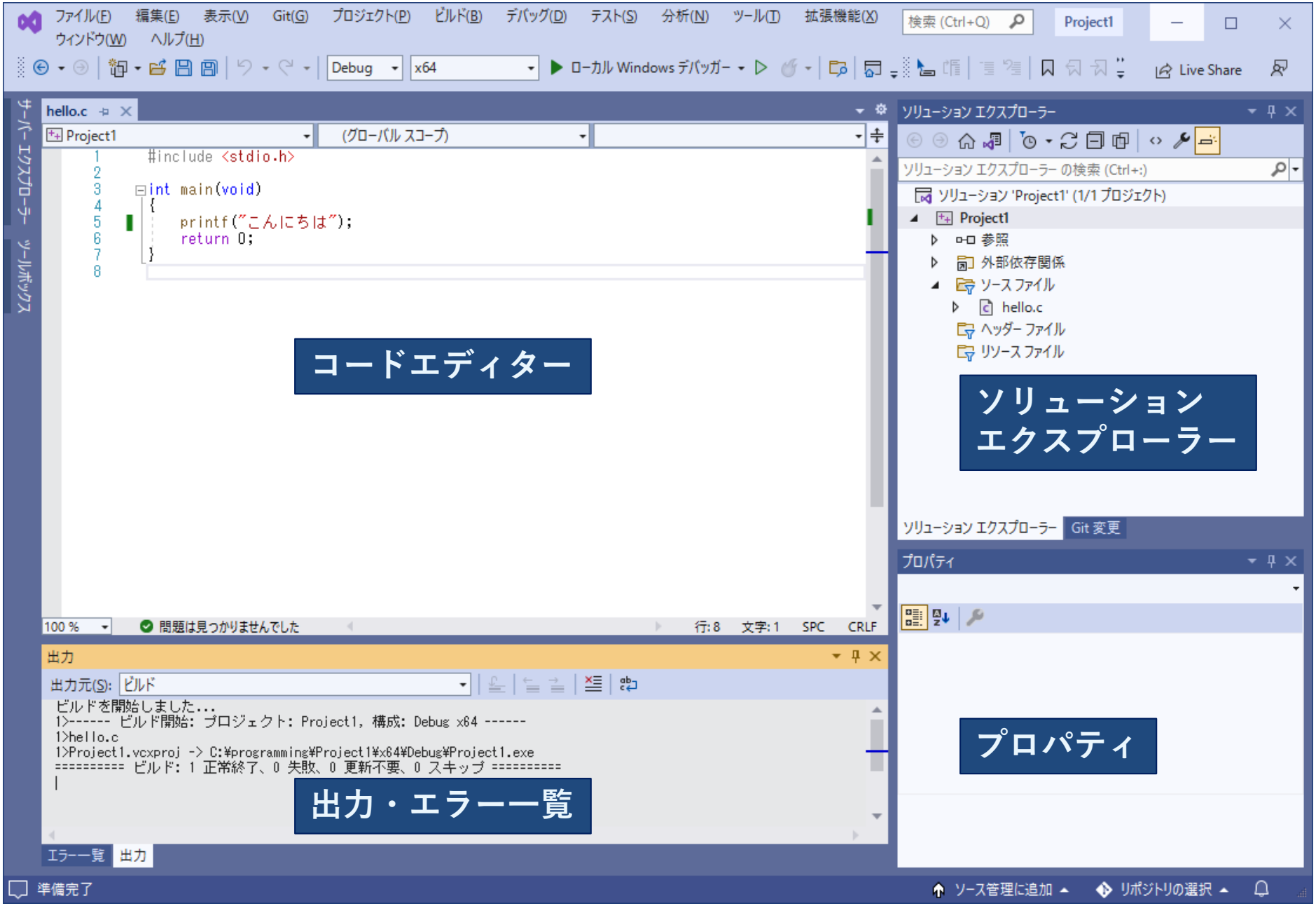


※ MacOSでの開発環境の構築は付録を参照

Visual Studio のインストール（Windows）



Visual Studioの画面構成



プログラムを作成して実行する

1. プロジェクトの作成
2. プログラムコードの作成
3. プログラムの実行

1. プロジェクトの作成

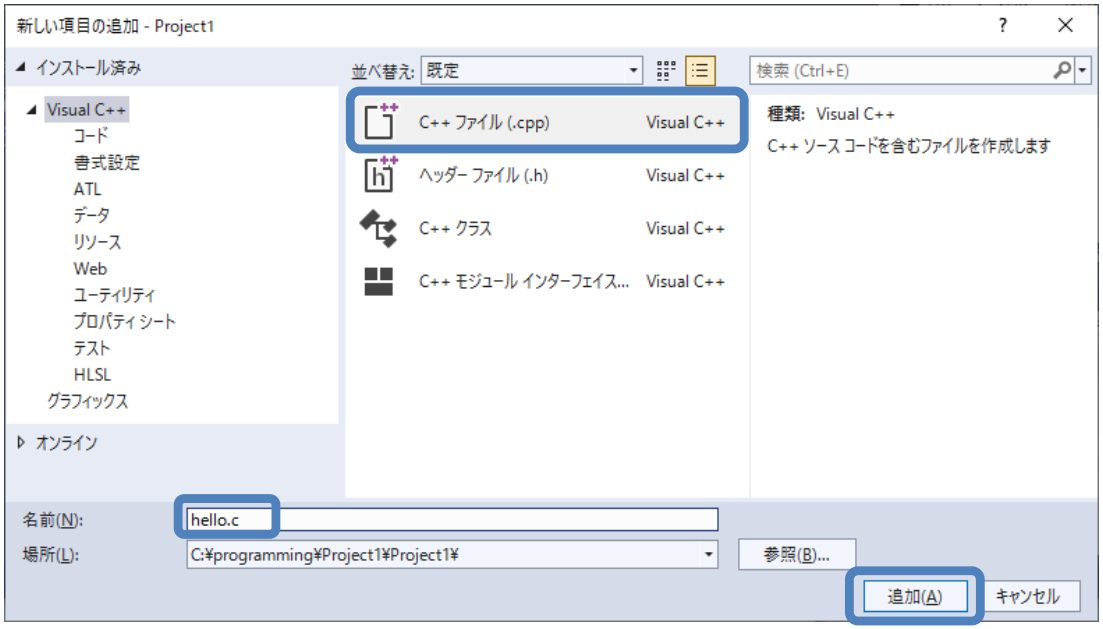

Visual Studio 2022
最近開いた項目(R)
Visual Studio を使用するとき、ユーザーが開くプロジェクト、フォルダー、ファイルはここに表示されるので、すばやくアクセスできます。
頻繁に開く項目は、ピン留めして常に一覧の先頭に表示することができます。
開始する
リポジトリのクローン(C)
GitHub や Azure DevOps などのオンライン リポジトリからコードを取得します
プロジェクトやソリューションを開く(P)
ローカルの Visual Studio プロジェクトまたは .sln ファイルを開きます
ローカル フォルダーを開く(F)
任意のフォルダー内のコードに移動して編集します
新しいプロジェクトの作成(N)
開始するには、コード スキャフォールディング付きのプロジェクト テンプレートを選択します
コードなしで続行(W) →


新しいプロジェクトの作成
テンプレートの検索(S) (Alt+S)
すべての言語(L)
すべてのプラットフォーム(P)
すべてのプロジェクトの種類...
最近使用したプロジェクト テンプレート(R)
空のプロジェクト C++
空のプロジェクト
Windows 用に C++ で最初から始めます。開始ファイルを提供しません。
C++ Windows コンソール
コンソール アプリ
Windows ターミナルでコードを実行します。既定では "Hello World" を出力します。
C++ Windows コンソール
CMake プロジェクト
.sln ファイルや .vcxproj ファイルに依存しない、最新のクロスプラットフォームの C++ アプリをビルドします。
C++ Windows Linux コンソール
Windows デスクトップ ウィザード
ウィザードを使用して、独自の Windows アプリを作成します。
C++ Windows デスクトップ コンソール ライブラリ
Windows デスクトップ アプリケーション
戻る(B)
次へ(N)


新しいプロジェクトを構成します
空のプロジェクト C++ Windows コンソール
プロジェクト名(J)
Project1
場所(L)
C:¥programming
...
ソリューション名(M)
Project1
ソリューションとプロジェクトを同じディレクトリに配置する(D)
戻る(B)
作成(C)

2. プログラムコードの作成

1. [プロジェクト]メニューの[新しい項目の追加]を選択
2. ファイルの拡張子を .c にする

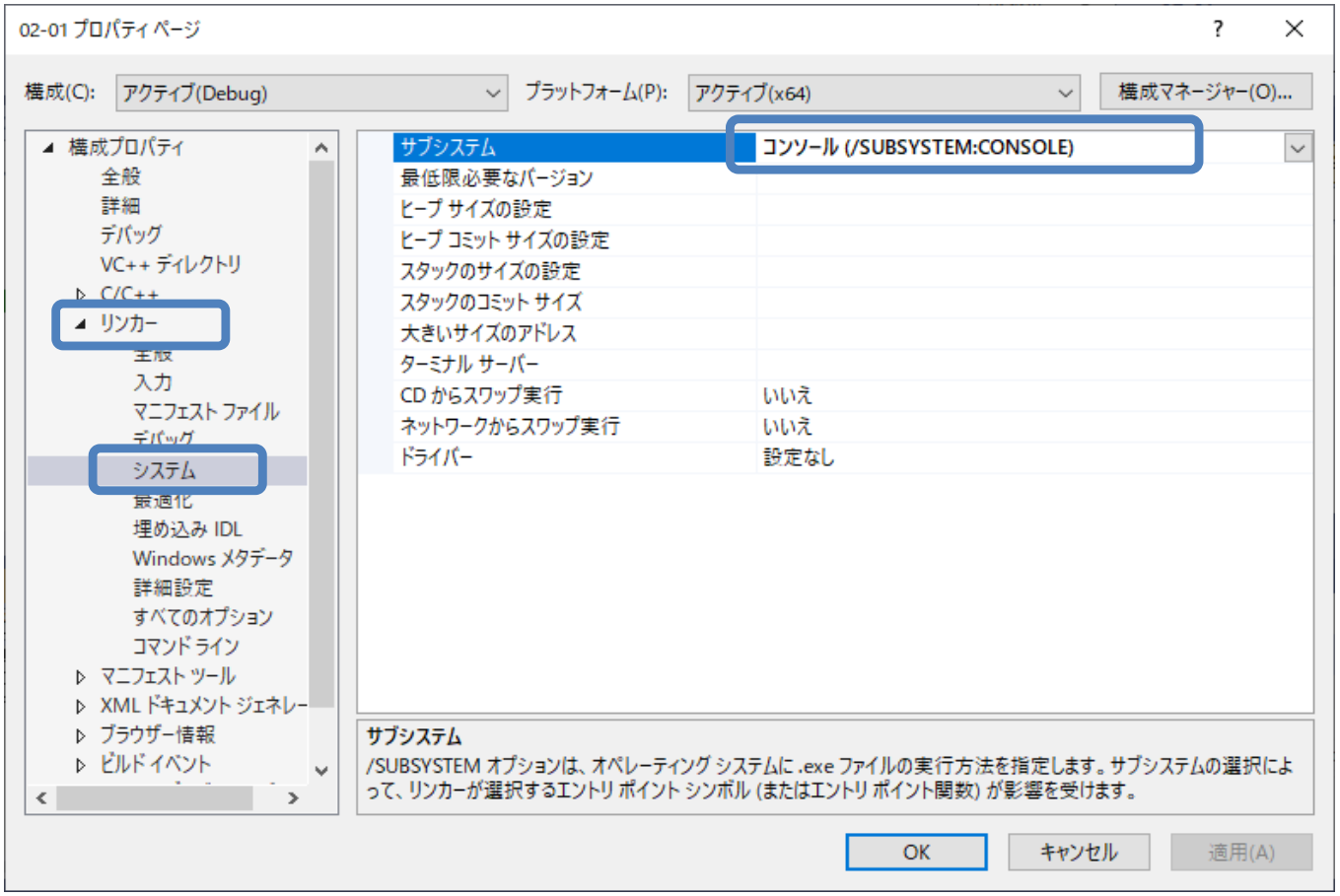



```
#include <stdio.h>

int main(void)
{
    printf("こんにちは");
    return 0;
}
```

3. コードエディターにプログラムコードを入力
4. ［ファイル］メニューの［すべて保存］を選択してプログラムコードを保存

3. プログラムの実行(1/2)

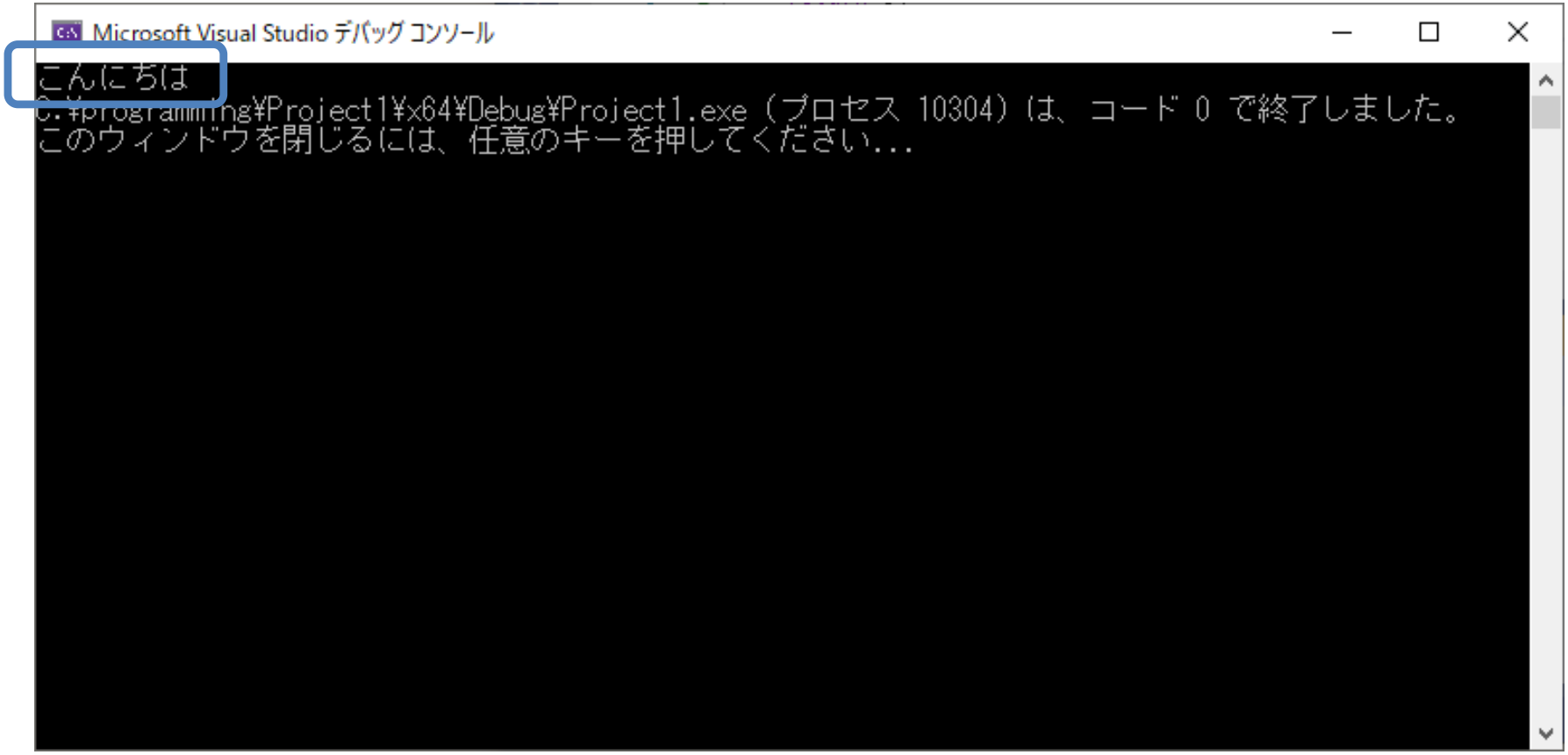
1. [プロジェクト]メニューの[（プロジェクト名）のプロパティ]を選択
2. [構成プロパティ]→[リンカー]→[システム]を選択
3. 右側の[サブシステム]に[コンソール（/SUBSYSTEM:CONSOLE）]を選択



※ プログラム実行直後にコンソールウィンドウが閉じてしまう場合には、この設定を確認

3. プログラムの実行(2/2)

[デバッグ]メニュー→[デバッグなしで開始]を選択



※ 練習用であれば、毎回新しいプロジェクトファイルを作る必要は無い。エディターの中でプログラムコードを更新して、その都度、実行結果を確認する

プログラムコードの間違い

Error（エラー）の種類

- **コンパイルエラー**（Compile Error）
  - キーワードのつづりミス
  - 文法上の間違い

- **ランタイムエラー** または **実行時エラー** （Runtime Error）
  - コンパイル時には発見されず、プログラムを実行している最中に見つかる問題

Visual Studioで
最初のプログラムを作ってみよう

第2章 C言語の基本

出力

画面へ文字列を出力する

文字列を出力する printf 関数

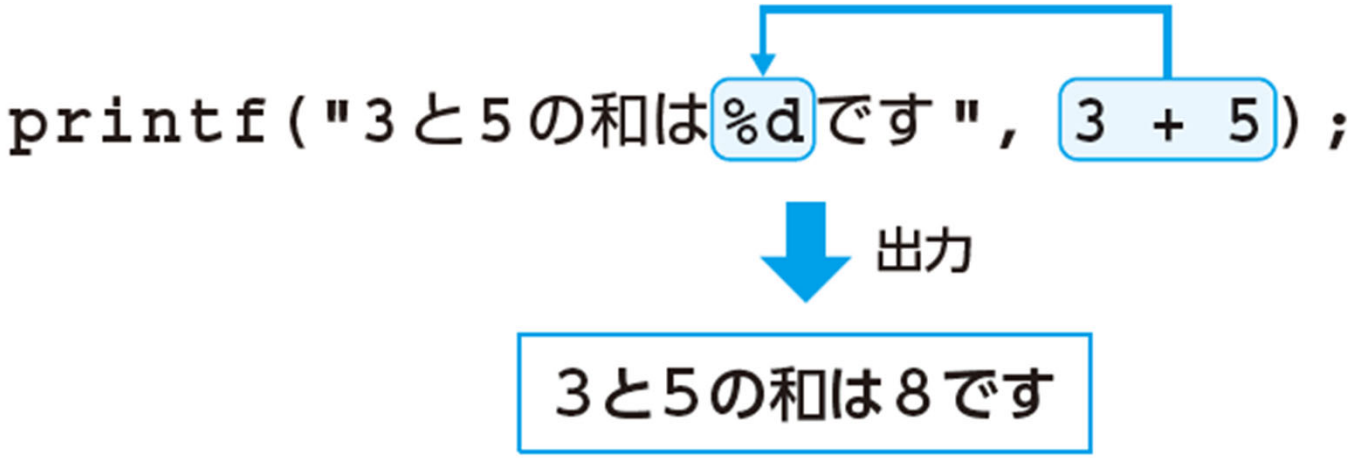
```
printf(出力する内容);
```

・文字列をダブルクォーテーション(")で囲む
・「¥n」で改行

```
#include <stdio.h>

int main(void)
{
    printf("こんにちは¥n");
    printf("今日もよい天気です");
}
```

数値の埋め込み

文字列の中に「%d」と記述した場所が数字に置き換わる



数値の埋め込み

```
#include <stdio.h>

int main(void)
{
    printf("今年は西暦 %d 年です¥n", 2022);
    printf("フルマラソンのコースは %f キロ¥n", 42.195);
    printf("%d歳の男の子の平均身長は %f cm¥n", 10, 138.9);
}
```

「**変換指定**」

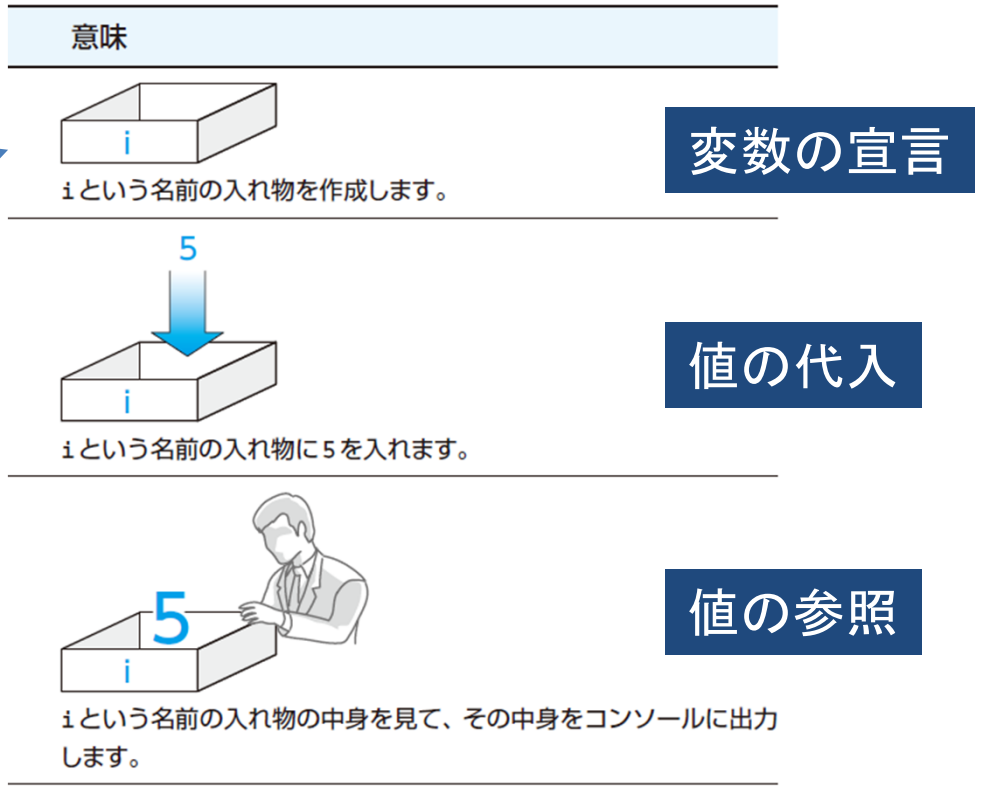
%d ：整数の埋め込み
%f ：小数点を含む数の埋め込み

表示する桁数の指定
%.2f ：小数点以下2桁を表示
%.5f ：小数点以下5桁を表示

変数

変数：値を入れておく入れ物のこと

```
#include <stdio.h>

int main(void)
{
    int i;
    i = 5;
    printf("%d", i);
}
```



変数の宣言

```
型名　変数名;
```

```
int i;
```

変数名
- 英字、数字、アンダースコア（ _ ）が使える
- 先頭の文字が数字であってはいけない
- 大文字と小文字が区別される
- C言語で用途が決まっている単語を変数名にはできない

値の代入

```
変数名 = 値;
```

```
i = 5;
```

整数名　　値　　左辺の変数に右辺の値を入れる操作

複数回実行した場合は、後から代入した値に上書きされる

```
i = 5;
i = 10;
printf("%d", i);
```

← 値の参照

↓ 実行結果

```
10
```

変数の初期化

初期化：変数に最初の値を入れること

```
int i;
i = 5;
```

1行にまとめられる

```
int i = 5;
```

変数の型

種類	型の名前	サイズ	格納できる値の範囲
整数型	short	2バイト	16ビット符号付き整数 -2^{15} (−32,768) ～$2^{15}-1$ (32,767)
	unsigned short	2バイト	16ビット符号なし整数 0～$2^{16}-1$ (65,535)
	int	4バイト	32ビット符号付き整数 -2^{31} (−2,147,483,648) ～$2^{31}-1$ (2,147,483,647)
	unsigned int	4バイト	32ビット符号なし整数 0～$2^{32}-1$ (4,294,967,295)
	long	4バイト	32ビット符号付き整数 -2^{31} (−2,147,483,648) ～$2^{31}-1$ (2,147,483,647)
	unsigned long	4バイト	32ビット符号なし整数 0～$2^{32}-1$ (4,294,967,295)
	long long	8バイト	64ビット符号付き整数 -2^{63} (−9,223,372,036,854,775,808) ～$2^{63}-1$ (9,223,372,036,854,775,807)
	unsigned long long	8バイト	64ビット符号なし整数 0～$2^{64}-1$ (18,446,744,073,709,551,615)
浮動小数点数型	float	4バイト	32ビット符号付き浮動小数点数 (注❷-6)
	double	8バイト	64ビット符号付き浮動小数点数
	long double	8バイト	64ビット符号付き浮動小数点数
文字型	char	1バイト	英数字1文字 (−128～127)
	unsigned char	1バイト	英数字1文字 (0～255)

※ 実行環境によって異なる場合がある

よく使用する型

・整数： int型
・小数点を含む数： double型
・文字：char型

※ char型は a～z,A～Z,0～9, #!?> といった半角英数字記号1文字

さまざまな型の変数

```
#include <stdio.h>

int main(void)
{
    int i = 1;
    double d = 0.2;
    char c = 'A';

    printf("iの値は %d¥n", i);
    printf("dの値は %f¥n", d);
    printf("cの値は %c¥n", c);
}
```

「**変換指定**」
%d ：整数の埋め込み
%f ：小数点を含む数の埋め込み
%c ：文字の埋め込み

※ char型の値は、文字をシングルクォーテーションで囲む

実行結果

```
iの値は 1
dの値は 0.200000
cの値は A
```

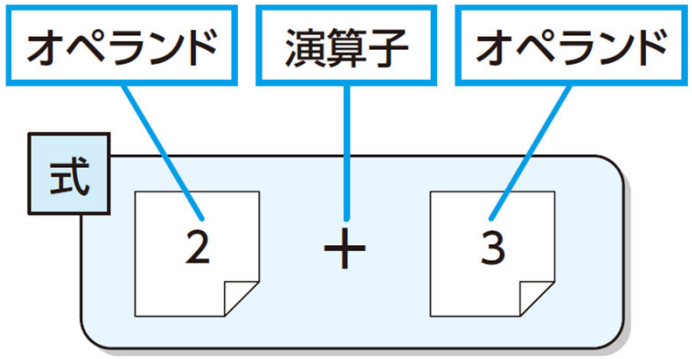
実際に試してみよう

算術演算子と式

計算を行う

```
#include <stdio.h>

int main(void)
{
    int i;
    i = 2 + 3;
    printf("%d", i);
}
```

用語



式の値は5

算術演算子

算術演算子

演算子	演算の内容
+	加算（足し算）
-	減算（引き算）
*	乗算（掛け算）
/	除算（割り算）
%	剰余

変数を含む算術演算

```
int i = 10;
int j = i * 2;
printf("jの値は %d", j);
```

↓ 実行結果

```
jの値は 20
```

いろいろな計算をしてみよう

変数の値を変更する

例：変数iの値を 3 増やす

```
i = i + 3;
```

※「i の値に3を加えた」値を、変数iに代入する
i ← i + 3 のイメージ

短縮表現

```
i += 3;
```

さまざまな短縮表現

演算子	演算の内容	使用例
+=	加算代入	a += 2;　（a = a + 2; と同じ）
-=	減算代入	a -= 2;　（a = a - 2; と同じ）
*=	乗算代入	a *= 2;　（a = a * 2; と同じ）
/=	除算代入	a /= 2;　（a = a / 2; と同じ）
%=	剰余代入	a %= 2;　（a = a % 2; と同じ）
++	インクリメント	a++;　（a = a + 1; と同じ）
--	デクリメント	a--;　（a = a - 1; と同じ）

いろいろな計算をしてみよう

キーボードからの入力を受け取る

キーボードからの入力を受け取ることで、
入力に応じた処理を行うプログラムを作成できるようになる

```
#include <stdio.h>

int main(void)
{
    int i;
    printf("整数を入力してください¥n");
    scanf_s("%d", &i);
    printf("%dが入力されました¥n", i);
}
```

キーボードからの入力を変数iで受け取る

実行結果

```
整数を入力してください
99
99が入力されました
```

キーボードからの入力

実際に試してみよう

演算と型

異なる型の値の代入

```
int i = 1.99;
printf("%d", i);
```

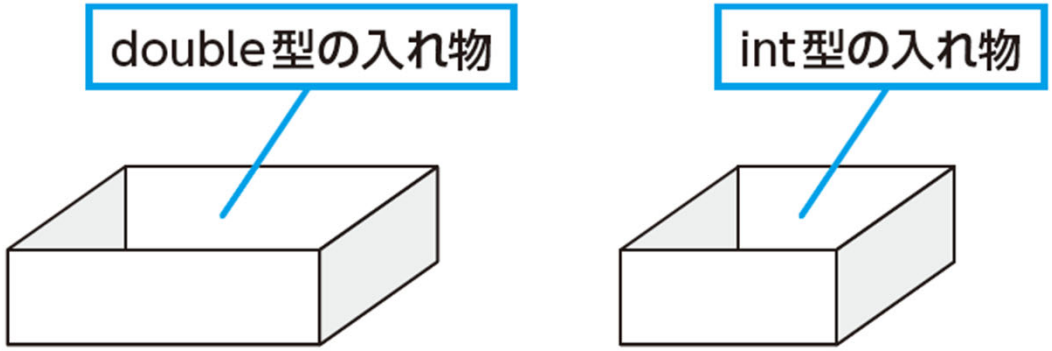
↓実行結果

```
1
```

int型の変数には整数しか代入できない。
小数点を含む数を代入すると、小数点以下が無視される

異なる型を含む演算

型によって変数（入れ物）の大きさが異なる



※ double 型の方が大きい

型の異なる変数が含まれる演算では、大きい型に統一されて演算が行われる

```
int i = 5;
double d = 0.5;
printf("%f¥n", i + d);
```

※ double 型の変数とint型の変数が含まれる演算では、double型に統一される

↓ 実行結果

```
5.500000
```

整数どうしの割り算

int 型どうしの割り算では、結果もint型になるので注意が必要

```
int a = 1;
int b = 2;
double c = a / b;
printf("cの値は %f¥n", c);
```

実行結果

```
cの値は 0.000000
```

double 型に**型変換（キャスト）**して計算する

```
int a = 1;
int b = 2;
double c = (double)a / (double)b;
printf("cの値は%f¥n", c);
```

実行結果

```
cの値は 0.500000
```

実際に試してみよう

第3章　条件分岐と繰り返し

条件分岐

条件分岐

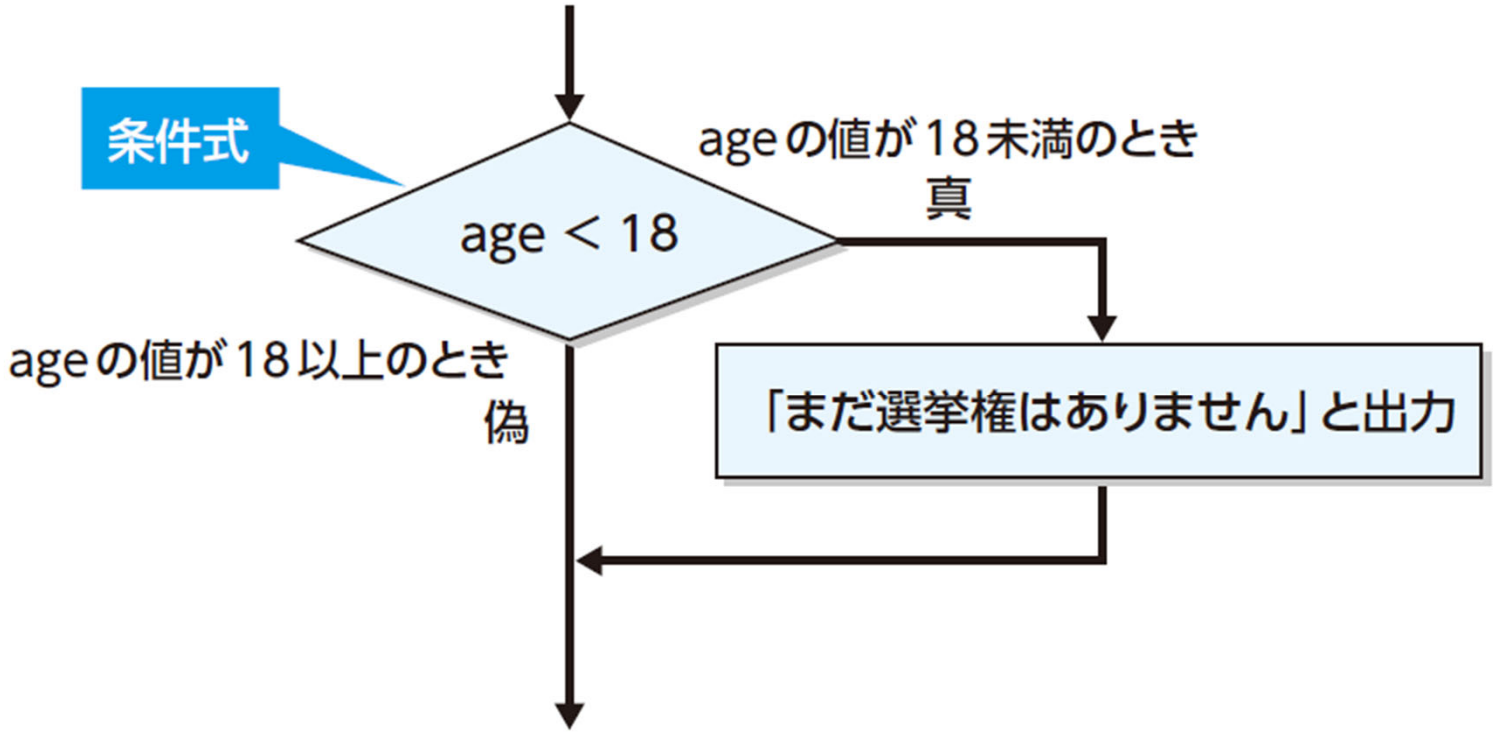
条件による処理の分岐
「もしも○○ならば××を実行する」

```
if (○○) {
    ××;
}
```

```
if (条件式) {
    命令文; //条件式が「真」の場合に実行される
}
```

条件分岐の例

```
if (age < 18) {
    printf("まだ選挙権はありません¥n");
}
```



関係演算子

- 関係演算子を使って、2つの値を比較できる
- 比較した結果は 「真」 または 「偽」になる

演算子	説明	例
==	左辺と右辺が等しい	a == 1（変数aが1のときに真）
!=	左辺と右辺が等しくない	a != 1（変数aが1でないときに真）
>	左辺が右辺より大きい	a > 1（変数aが1より大きいときに真）
<	左辺が右辺より小さい	a < 1（変数aが1より小さいときに真）
>=	左辺が右辺より大きいか等しい	a >= 1（変数aが1以上のときに真）
<=	左辺が右辺より小さいか等しい	a <= 1（変数aが1以下のときに真）

if ～ else 文

「もしも○○ならば××を実行し、そうでなければ△△を実行する」

```
if (○○) {
    ××;
} else {
    △△;
}
```

```
if (条件式) {
    //条件式が「真」の場合
    命令文1;
} else {
    //条件式が「偽」の場合
    命令文2;
}
```

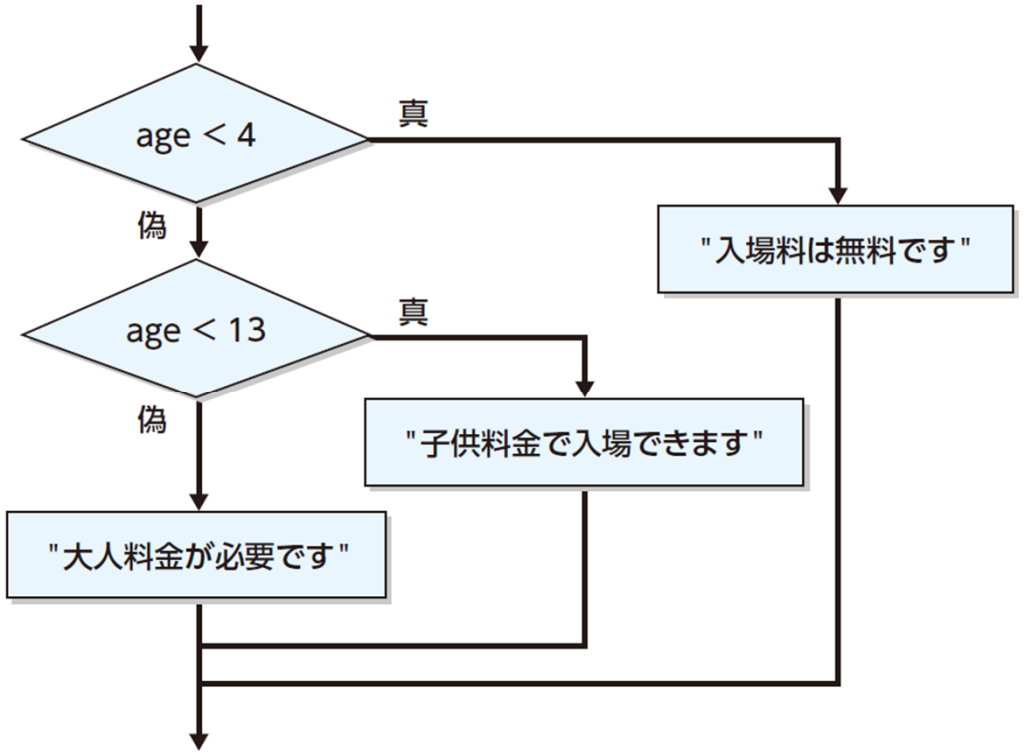
例

```
if (age < 18) {
    printf("まだ選挙権はありません¥n");
} else {
    printf("投票に行きましょう¥n");
}
```

複数の if ～ else 文

if～else文を連結して、条件に応じた複数の分岐を行える

```
if (age < 4) {
    printf("入場料は無料です¥n");
} else if (age < 13) {
    printf("子供料金で入場できます¥n");
} else {
    printf("大人料金が必要です¥n");
}
```



実際に試してみよう

ワン・モア・ステップ：条件式に数値を使う

- if 文の条件式に数値を使うことができる
- **0が偽、それ以外が真**であるとみなされる

```
if (0) {
   命令文
}
```

命令文は実行されない

```
if (1) {
   命令文
}
```

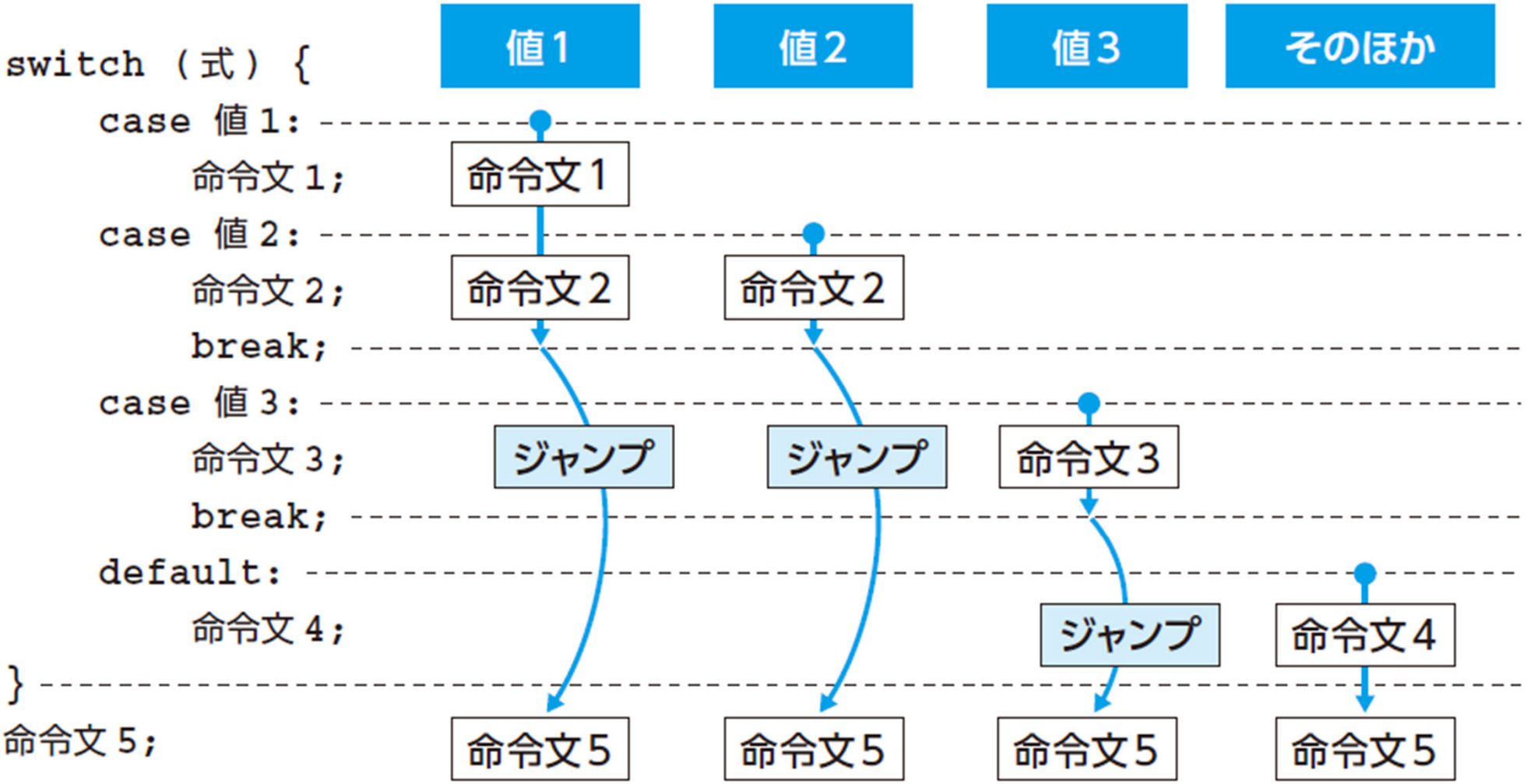
命令文は実行される

```
if (i) {
   命令文
}
```

変数iの値が0でない場合に命令文が実行される

実際に試してみよう

switch文



式の値によって処理を切り替える。**break;**でブロックを抜ける。

switch文の例(1)

```
switch (score) {
case 1:
    printf("もっと頑張りましょう¥n");
    break;
case 2:
    printf("もう少し頑張りましょう¥n");
    break;
case 3:
    printf("普通です¥n");
    break;
case 4:
    printf("よくできました¥n");
    break;
case 5:
    printf("大変よくできました¥n");
    break;
default:
    printf("想定されていない点数です¥n");
}
printf("switchブロックを抜けました¥n");
```

実際に試してみよう

switch文の例(2)

```
switch (score) {
case 1:
case 2:
    printf("もっと頑張りましょう¥n");
    break;
case 3:
case 4:
case 5:
    printf("合格です¥n");
    break;
default:
    printf("想定されていない点数です¥n");
}
```

実際に試してみよう

ワン・モア・ステップ ： 3項演算子

```
int c;
if (a > b) {
    c = a;
} else {
    c = b;
}
```

```
int c = (a > b) ? a : b;
```

（構文） 条件式？値1：値2

条件式が「真」の場合に、式の値が「値1」になる。「偽」の場合には「値2」になる

論理演算子

論理演算子を使って複数の条件式を組み合わせられる

演算子	演算の名前	式が真になる条件	使用例
&&	論理積	左辺と右辺の両方が真のとき	a > 0 && b < 0 (変数aが0より大きく、かつbが0より小さい場合に真)
||	論理和	少なくとも左辺と右辺のどちらかが真のとき	a > 0 || b < 0 (変数aが0より大きい、または変数bが0より小さい場合に真)
^	排他的論理和	左辺と右辺のどちらかが真で他方が偽のとき	a > 0 ^ b < 0 (変数a が0より大きく、かつbが0より小さくない場合に真。またはaが0より大きくなく、かつbが0より小さい場合に真)
!	否定	右辺が偽のとき(左辺はなし)	!(a > 0) (変数aが0より大きくない場合に真)

論理演算子の例

ageが13以上 かつ ageが65未満

```
age >= 13 && age < 65
```

ageが13未満 または age が65以上

```
age < 13 || age >= 65
```

ageが13以上 かつ age が65未満 かつ 20でない

```
age >= 13 && age < 65 && age !=20
```

演算子の優先度

算術演算子が関係演算子より優先される

```
a + 10 > b * 5
```

＝

```
(a + 10) > (b * 5)
```

関係演算子が論理演算子より優先される

```
a > 10 && b < 3
```

＝

```
(a > 10) && (b < 3)
```

カッコの付け方で論理演算の結果が異なる

```
x && ( y || z )
```

≠

```
(x && y ) || z
```

実際に試してみよう

処理の繰り返し

繰り返し処理

- ある処理を繰り返し実行したいことがよくある
- ループ構文を使用すると、繰り返し処理を簡単に記述できる
- C言語には3つのループ構文がある

  for文

  while文

  do ～ while文

for文

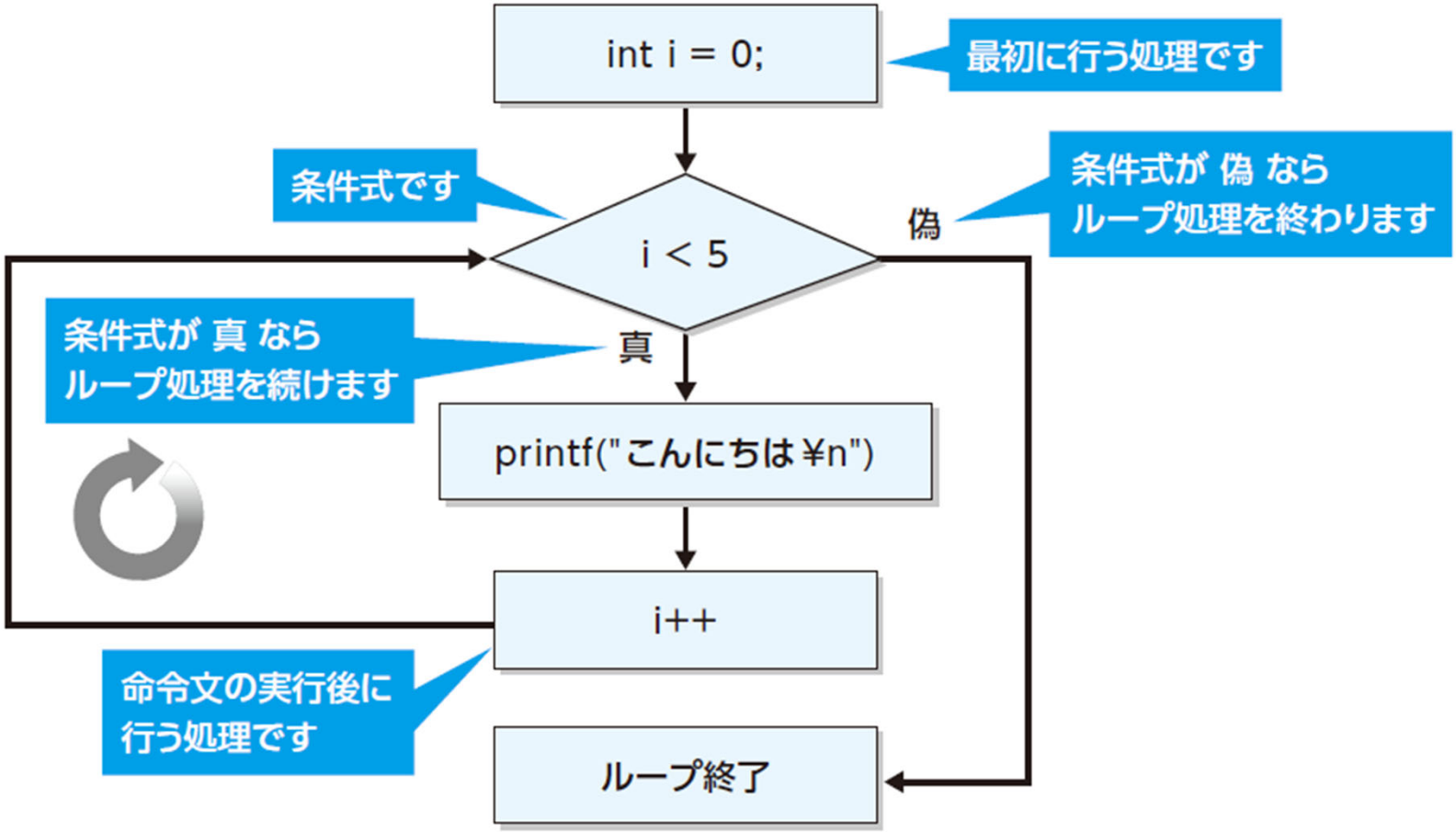
for文の構文

```
for (最初の処理; 条件式; 命令文の実行後に行う処理) {
    命令文
}
```

1. 「最初の処理」を行う
2. 「条件式」が真なら「命令文」を行う
   偽 ならfor文を終了する
3. 「命令文の後に行う処理」を行う
4. 2.に戻る

for文の例

```
for (int i = 0; i < 5; i++) {
    printf("こんにちは¥n");
}
```



forループ内で変数を使う

for ループ内で変数を使用することで、例えば1から100までを順番に足し合わせる計算ができる

```
int sum = 0;
for (int i = 1; i <= 100; i++) {
    sum += i;
    printf("%dを加えました¥n", i);
}
printf("合計は%dです¥n", sum);
```

↓実行結果

```
1を加えました
2を加えました
3を加えました
…（中略）…
99を加えました
100を加えました
合計は5050です
```

実際に試してみよう

変数のスコープ

- 変数には扱える範囲が決まっている。これを「変数のスコープ」と呼ぶ
- スコープは変数の宣言が行われた場所から、そのブロック{ } の終わりまで

```
int main(void)
{
    int sum = 0;
    for (int i = 1; i <= 100; i++) {
        sum += i;
        printf("%dを加えました¥n", i);
    }
    printf("合計は%dです¥n", sum);
}
```

while文

while文の構文

```
while (条件式) {
    命令文
}
```

1. 「条件式」が真なら「命令文」を行う
   偽 ならwhileループを終了する
2. 1.に戻る

※ for文と同じ繰り返し命令を書ける

while文の例

```
int i = 0;
while (i < 5) {
    printf("こんにちは¥n");
    i++; // この命令文が無いと「無限ループ」
}
```

```
int i = 5;
while (i > 0) {
    printf("こんにちは¥n");
    i--;  // この命令文が無いと「無限ループ」
}
```

実際に試してみよう

do ～ while文

do ～ while文の構文

```
do {
    命令文
} while (条件式);
```

必ず1回は実行される

1. 「命令文」を実行する
2. 「条件式」が真なら1.に戻る。
   偽ならdo～whileループを終了する

※ for文、while文と同じ繰り返し命令を書ける

do ～ while文の例

```
int i = 0;
do {
    printf("こんにちは¥n");
    i++;
} while (i < 5);
```

```
int i = 5;
do {
  printf("%d¥n", i);
  i--;
} while(i > 0);
```

実際に試してみよう

ループの処理を中断する「break」

`break;` でループ処理を強制終了できる

```
int sum = 0;
for (int i = 1; i <= 10; i++) {
    sum += i;
    printf("変数sumに%dを加えました。", i);
    printf("sumは%d¥n", sum);
    if (sum > 20) {
        printf("合計が20を超えました。¥n");
        break;
    }
}
```

ループ内の処理をスキップする「continue」

continue; でブロック内の残りの命令文をスキップできる

```
int sum = 0;
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue;
    }
    sum += i;
    printf("変数sumに%dを加えました。", i);
    printf("sumは%d¥n", sum);
}
```

実際に試してみよう

ループ処理のネスト

ループ処理の中にループ処理を入れられる

```
for (int a = 1; a <= 3; a++) {
    printf("a = %d¥n", a);//★
    for (int b = 1; b <= 3; b++) {
        printf("   b = %d¥n", b);//☆
    }
}
```

★の命令文は3回実行される

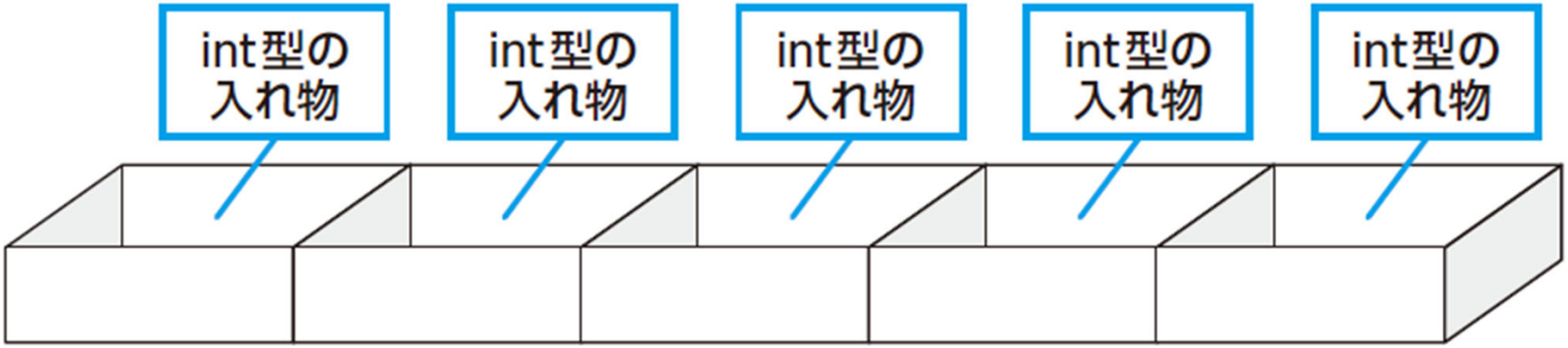
☆の命令文は9回実行される

実際に試してみよう

配列

1次元配列

- 複数の値の入れ物が並んだもの
  （1次元配列とも呼ぶ）
- 複数の値をまとめて扱うときに便利



配列の使い方

1. 配列を表す変数を宣言する

```
int scores[5];
```

← 要素の数を指定する

2. 配列に値を入れる

```
scores[0] = 50;
      ⋮
scores[4] = 80;
```

[]の中の数字はインデックス
0～(要素の数-1)を指定する

3. 配列に入っている値を参照する。

例：`printf("%d¥n", scores[i]);`

配列の使用

```
int scores[5];
scores[0] = 50;
scores[1] = 55;
scores[2] = 70;
scores[3] = 65;
scores[4] = 80;

for (int i = 0; i < 5; i++) {
    printf("%d¥n", scores[i]);
}
```

要素数が5のint型の配列を宣言しています

0から始まるインデックスを使って
要素を指定し、値を代入します

ループ処理で各要素
の値を出力します

配列の使用

配列は次のようにしても初期化できる

```
int scores[5] = {50, 55, 70, 65, 80};
```

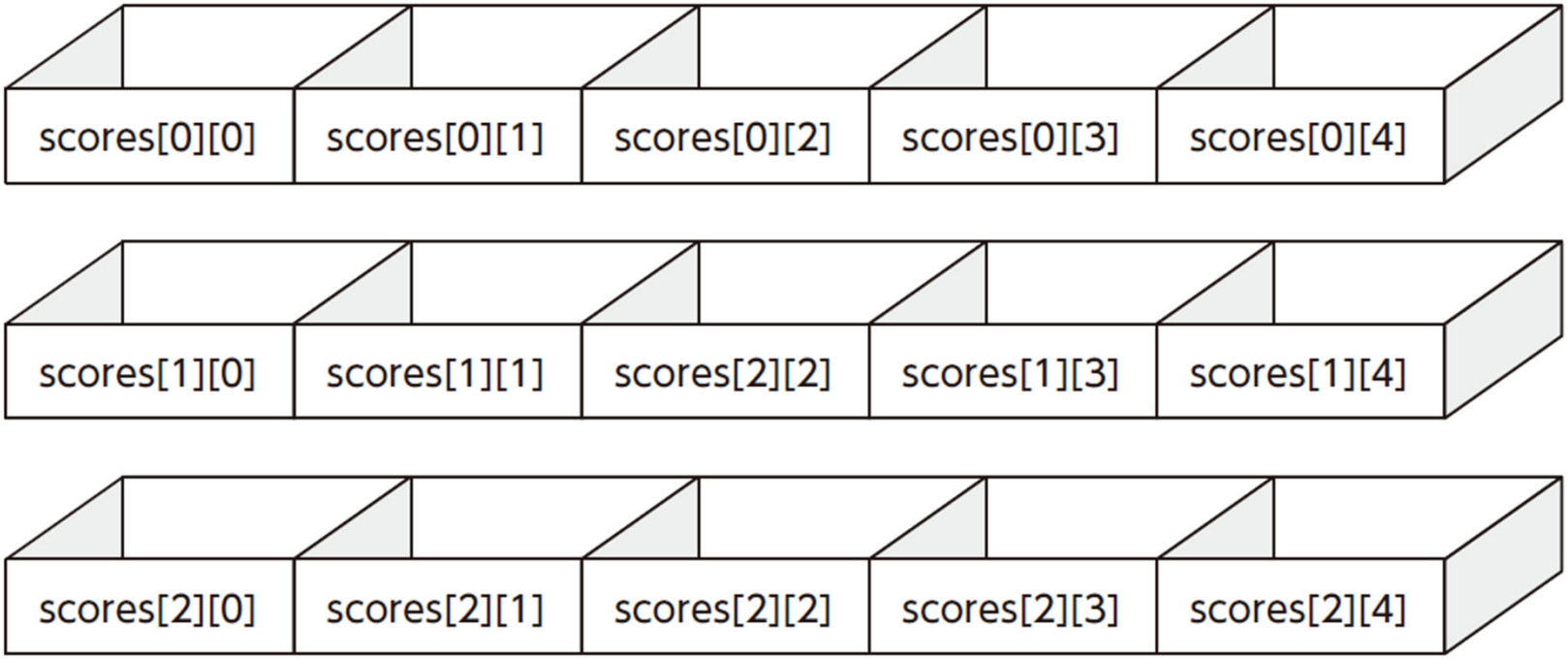
要素の数の記述を省略できる

```
int scores[] = {50, 55, 70, 65, 80};
```

実際に試してみよう

多次元配列（配列の配列）

```
int scores[3][5];
scores[0][0] = 50;
scores[2][3] = 65;
```



2次元配列の宣言と初期化

2次元配列は次のようにしても初期化できる

```
int scores[3][5] = {
{50, 55, 70, 65, 80},
{60, 77, 90, 73, 55},
{66, 85, 76, 95, 98}
};
```

最初の配列の要素数の記述は省略できる


```
int scores[][5] = {
{50, 55, 70, 65, 80},
{60, 77, 90, 73, 55},
{66, 85, 76, 95, 98}
};
```

第4章　関数

関数とは

関数とは

- 長いプログラムが必要になるときは、命令文を分けて管理した方が見通しがよくなる
- 関数は複数の命令文をまとめたもの

関数の定義

```
void 関数名(void) {
     命令文1
     命令文2
     ....
}
```

※ 関数をプログラムコードで記述することを「関数の定義」と呼ぶ

関数の定義の例

```
void countdown(void)
{
    printf("カウントダウンをします¥n");
    for (int i = 5; i >= 0; i--) {
        printf("%d¥n", i);
    }
}
```

countdownという名前をつけています

関数の定義がここから始まります

カウントダウンを実行するための命令文です

関数の定義はここで終わります

関数の呼び出し

```
#include <stdio.h>

void countdown(void)
{
    printf("カウントダウンをします¥n");
    for (int i = 5; i >= 0; i--) {
        printf("%d¥n", i);
    }
}

int main(void)
{
    countdown();
}
```

countdownという名前の関数を定義しています

countdownという名前の関数を呼び出します

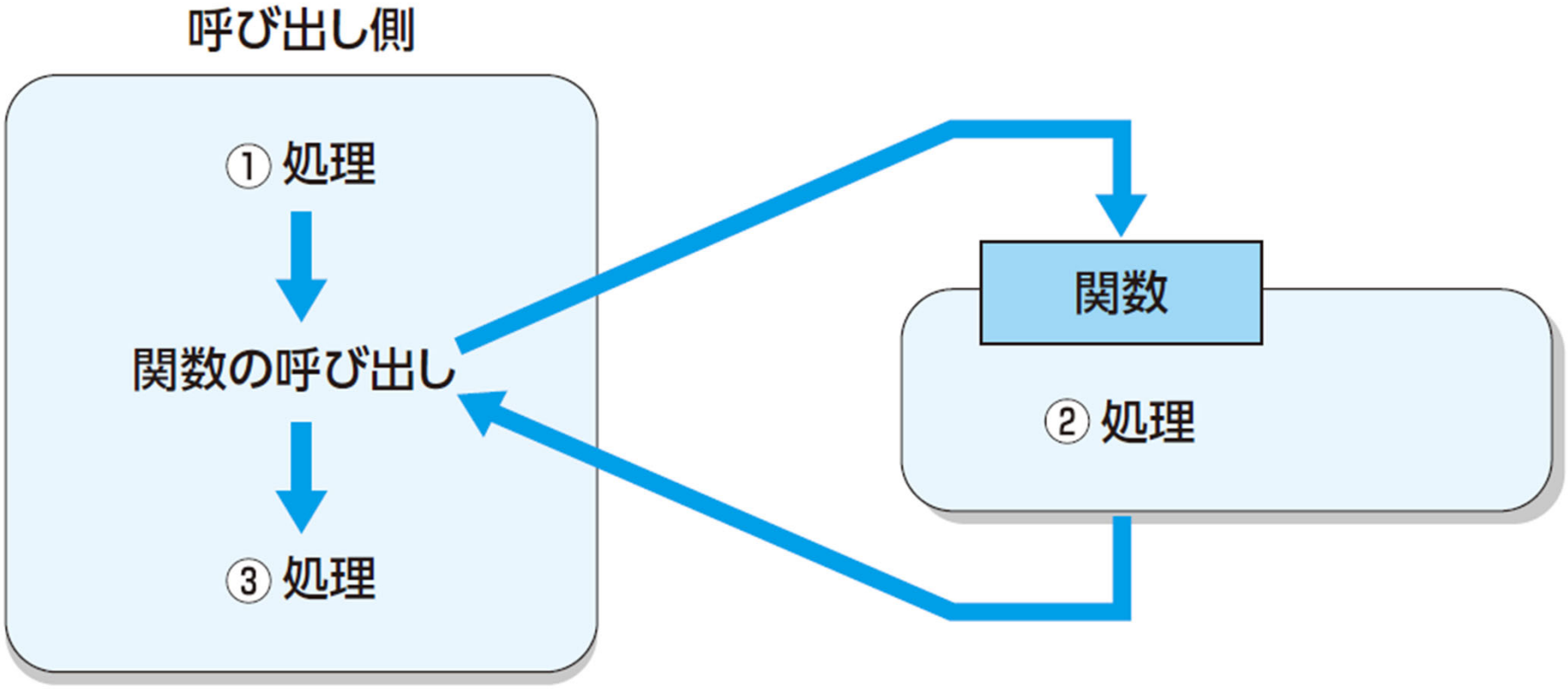
実際に試してみよう

main関数

```
int main(void)
```

- C言語では、プログラムが実行されるときに、このmain関数が呼び出される
- main関数は、プログラムの開始位置となる特別な関数

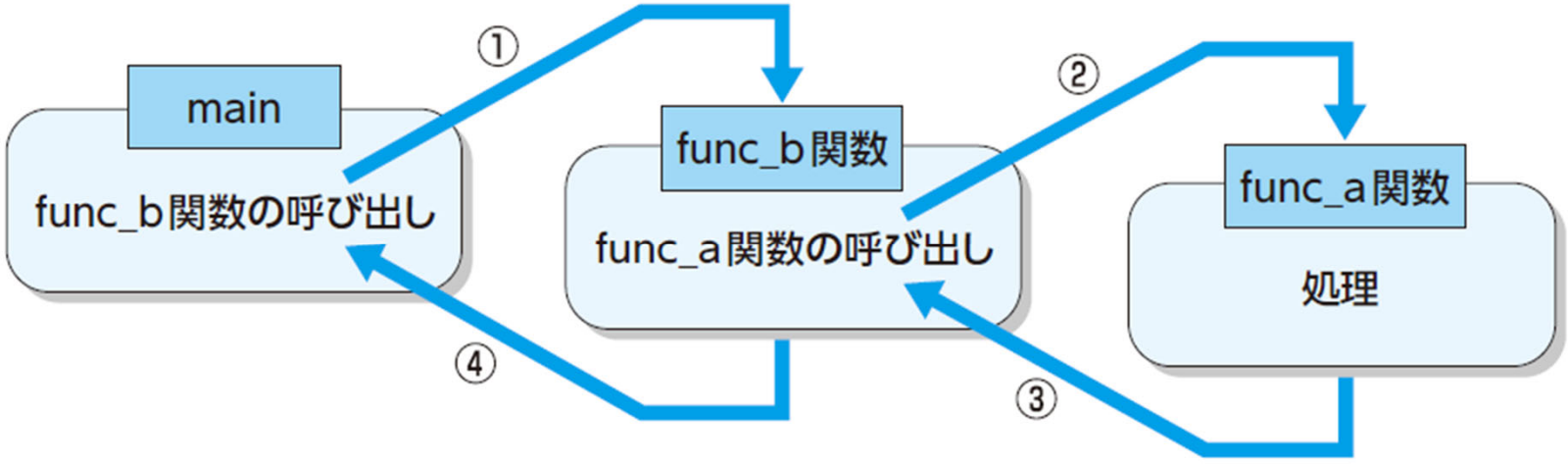
関数を呼び出すときの処理の流れ



関数の呼び出しの階層

```
#include <stdio.h>

void func_a(void)
{
}

void func_b(void)
{
    func_a(void);
}

int main(void)
{
    func_b();
}
```

実際に試してみよう



関数の引数

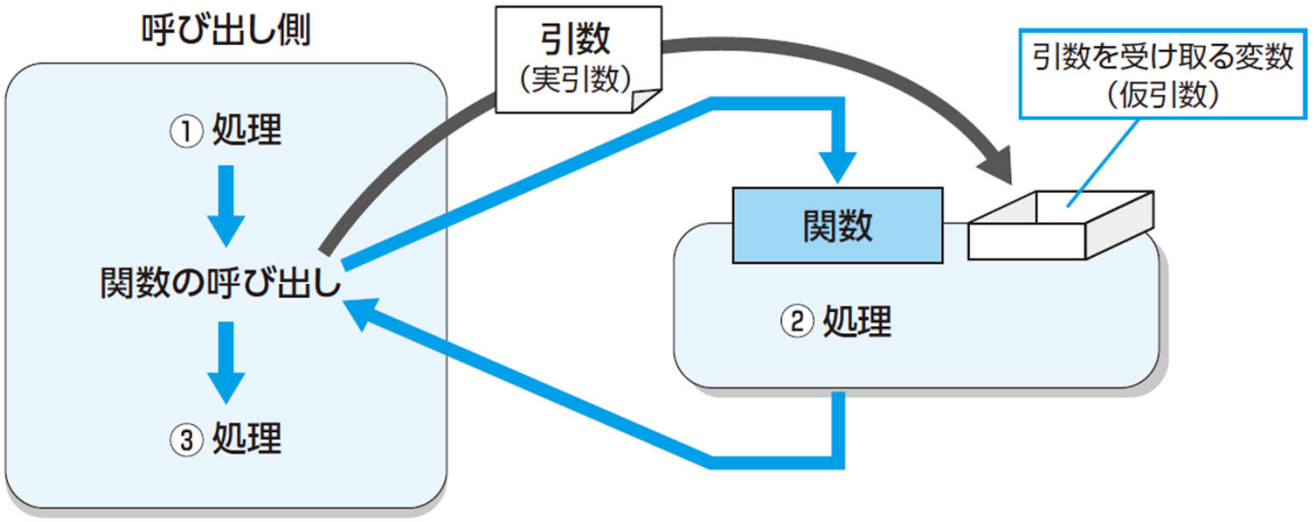
引数とは

引数（ひきすう）

関数には、呼び出すときに値を渡すことができる。この値を「引数」と呼ぶ。

引数のある関数の定義

```
void 関数名(型 変数名) {
        命令文
}
```



引数のある関数の例

引数の受け渡しには、関数名の後ろのカッコ()を使用する。

```
#include <stdio.h>

void countdown(int start)
{
    printf("関数が受け取った値：%d¥n", start);
    printf("カウントダウンをします¥n");
    for (int i = start; i >= 0; i--) {
        printf("%d¥n", i);
    }
}

int main(void)
{
    countdown(3);
    countdown(10);
}
```

startという名前のint型の変数で値を受け取る

実際に試してみよう

引数が複数ある関数の例

カンマで区切って複数の引数を指定できる

```
#include <stdio.h>

void countdown(int start, int end)
{
    printf("カウントダウンをします¥n");
    for (int i = start; i >= end; i--) {
        printf("%d¥n", i);
    }
}

int main(void)
{
    countdown(7, 3);
}
```

実際に試してみよう

実引数と仮引数

実引数：関数を呼び出すときに、関数に渡される値

実引数

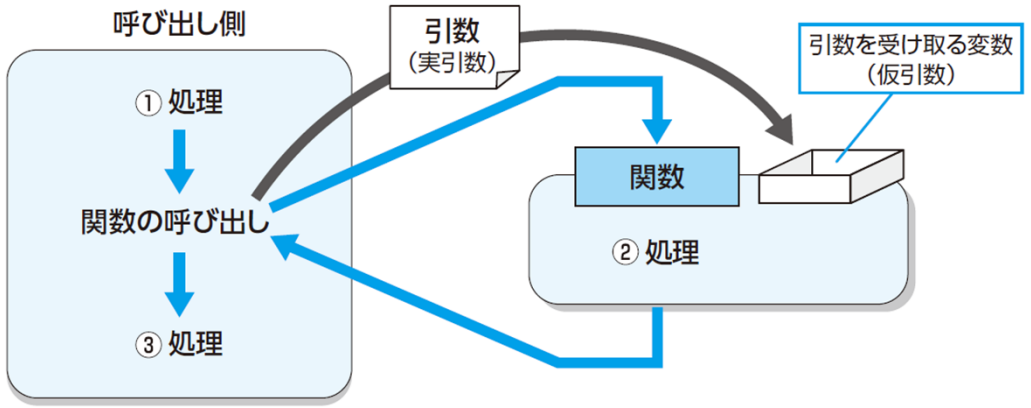
```
countdown(7, 3);
```

仮引数：関数側で値を受け取るために準備される変数

仮引数

```
void countdown(int start, int end)
```

実引数の値がコピーされて、それが仮引数に代入される



実引数と仮引数

```c
#include <stdio.h>

void func(int i) {
    i++;
}

int main(void)
{
    int i = 10;
    printf("(1) iの値は%d¥n", i);
    func(i);
    printf("(2) iの値は%d¥n", i);
}
```

実際に試してみよう

↓実行結果

```
(1) iの値は10
(2) iの値は10
```

iの値はfunc関数呼び出し前後で変化しない
func関数には、値の複製（コピー）が渡される

関数の戻り値

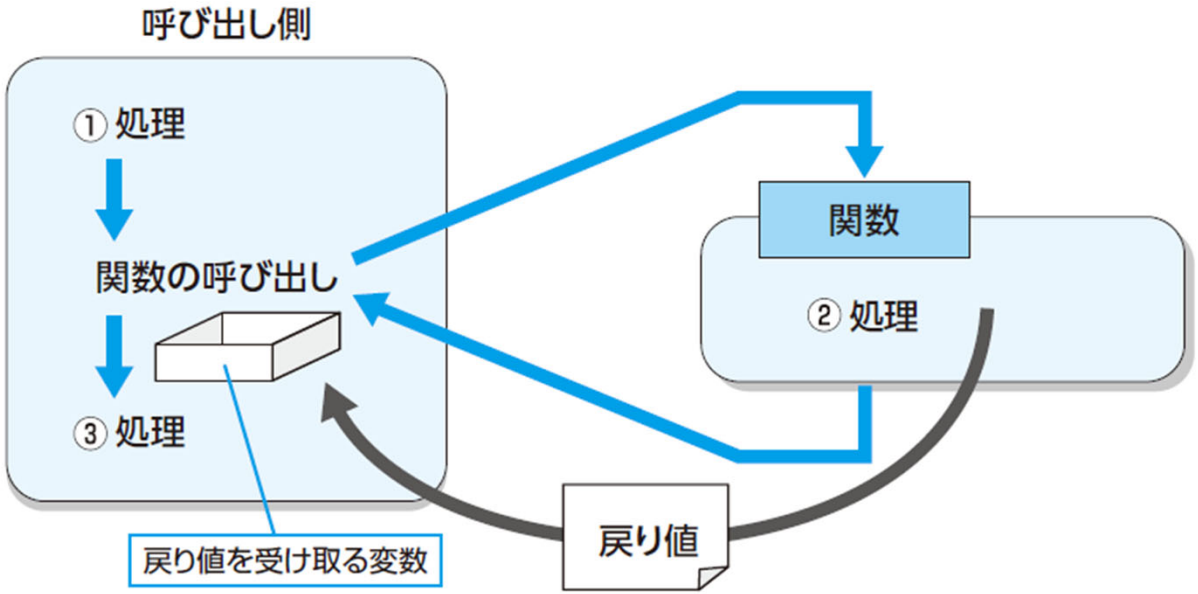
戻り値とは

戻り値

関数から返される値

戻り値のある関数の定義

```
戻り値の型 関数名(引数列) {
    命令文
    return 戻り値;
}
```



戻り値のある関数の例１

- return を使って値を戻すようにする
- 戻り値は1つだけ
- 戻り値の型を関数名の前に記す

```
#include <stdio.h>

double circle_area(double r)
{
    return r * r * 3.14159;
}

int main(void)
{
    double d = circle_area(2.5);
    printf("半径2.5の円の面積は %.2f¥n", d);
}
```

実際に試してみよう

関数のまとめ

引数なし、戻り値なし

```
void 関数名(void) {
    命令文
}
```

引数あり、戻り値なし

```
void 関数名(型 変数名) {
    命令文
}
```

引数あり、戻り値あり

```
戻り値の型 関数名(型 変数名) {
    命令文
    return 戻り値;
}
```

ワン・モア・ステップ：論理演算式の値

```
if (d > 0) {
  return 1;
} else {
  return 0;
}
```

```
return (d > 0);
```

論理演算式は値を持つ

真のときに1

偽のときに0

```
if (is_positive_number(d) == 1) { 命令文 }
```

```
if (is_positive_number(d)) { 命令文 }
```

関数のプロトタイプ宣言

main関数から、他の関数を呼び出すときには、その関数がどのようなものであるかをコンパイラが知っている必要がある

1. main関数よりも前に関数の定義を記述する

2. **プロトタイプ宣言**(どのような関数であるかを記したもの）を、関数の前に記述して、関数の定義はmain関数の後ろに記述する

関数のプロトタイプ宣言

```
戻り値の型  関数名(引数列);
```

関数のプロトタイプ宣言の例

```
#include<stdio.h>

void func1(int a);
void func2();

int main(void)
{
    func1(10);
    func2();
}

void func1(int a)
{
    printf("func1が呼び出されました¥n");
    printf("引数の値は%d¥n", a);
}

void func2()
{
    printf("func2が呼び出されました¥n");
}
```

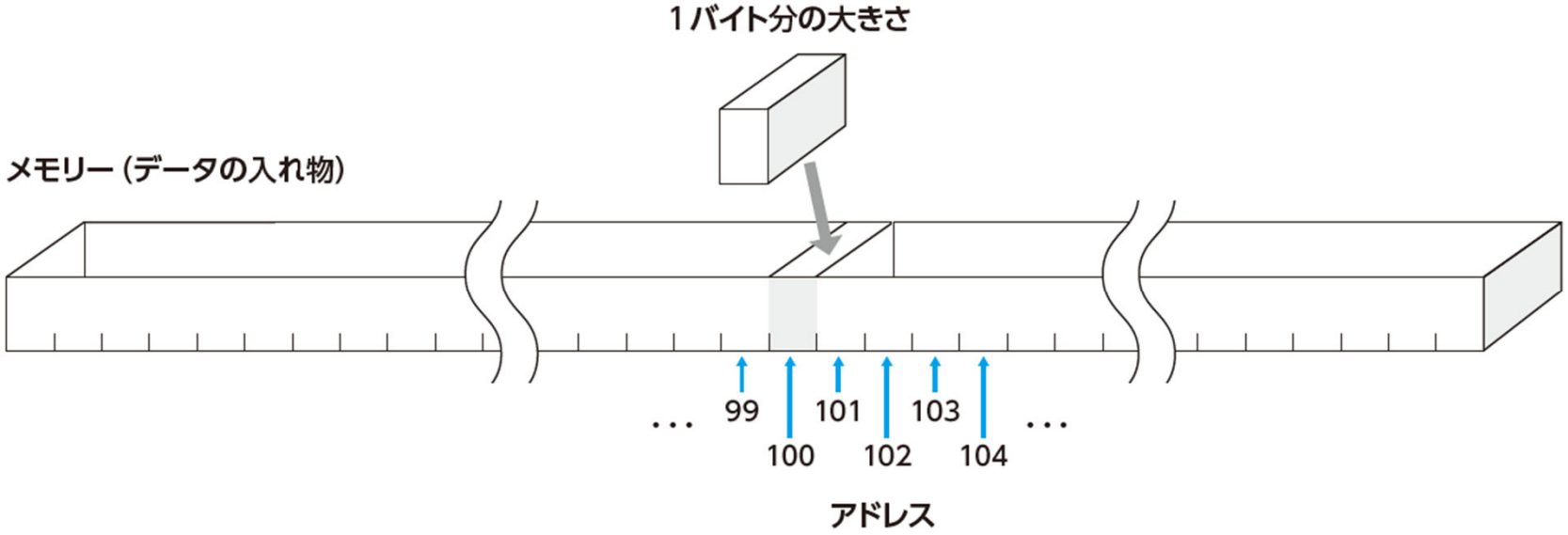
関数のプロトタイプ宣言

第5章　アドレスとポインタ

アドレスとポインタ

コンピューターのメモリー

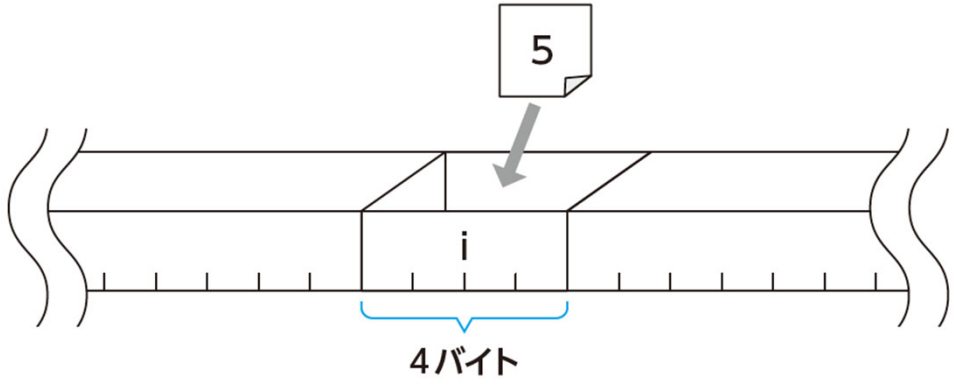
- コンピューターがプログラムを実行している途中に扱う、さまざまな情報がメモリーに格納される
- メモリーを、1バイト単位で区切れるように目盛りがついた、横に長い箱のようなものだとみなす
- 目盛りに割り当てられた数字（番地）を**アドレス**という



アドレス

```
int i = 5;
```

- int型の箱（サイズ 4バイト）がメモリー上に確保される
- その領域に、5という値のデータが格納される




```
printf("%p", &i);
```
← メモリー上の変数iの位置（アドレス）を知る

↓ 実行結果

```
0133FF00
```
← 16進数表現（実行のたびに変化する。具体的な値にあまり意味はない）

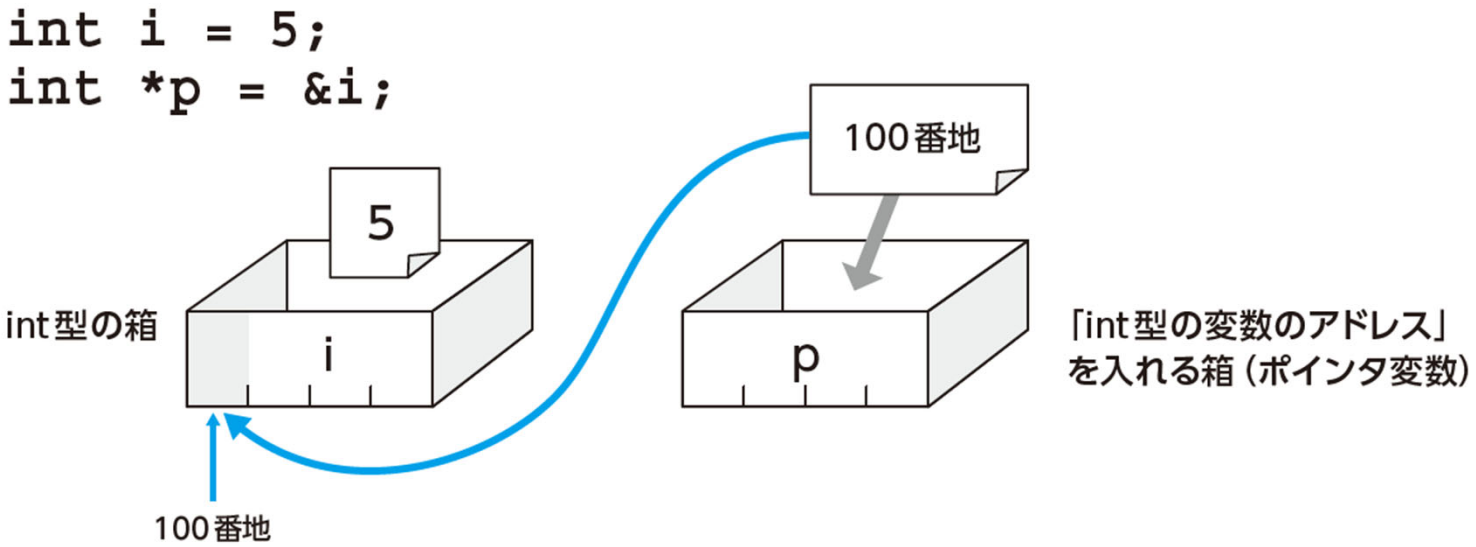
&変数名

変数のアドレスを参照するための記号「&」は**アドレス演算子**と呼ばれる。

ポインタ

- アドレスは**ポインタ変数**に代入できる

```
int i = 5;
int *p = &i;
```
← ポインタ変数（変数名の前に*記号をつける）



※ 図では、ポインタ変数の入れ物のサイズを4バイトで表していますが、このサイズは実行環境によって異なります。たとえば、64bit環境では8バイトになります。

ポインタを使った値の参照

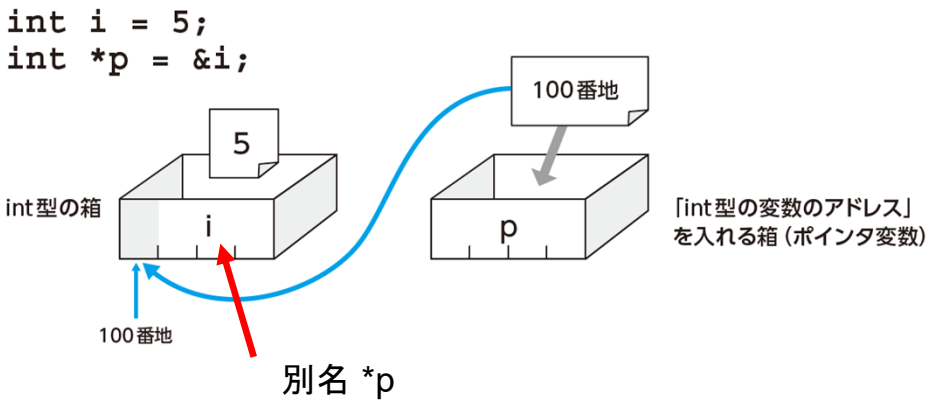
- pに入っているアドレスから、そこに存在する値を見にいくには、プログラムコードの中で「*p」と記述する
- 「i」と「*p」は、メモリー上の同じ値を参照する

```
int i = 5;
int *p = &i;
printf("iのアドレスは%p¥n", &i);
printf("pの値は%p¥n", p);
printf("iの値は%d¥n", i);
printf("*pの値は%d¥n", *p);
```

実際に試してみよう

↓実行結果

```
iのアドレスは00B3F77C
pの値は00B3F77C
iの値は5
*pの値は5
```



ポインタの活用

ポインタを使って値を変更する

```
int i = 5;
int *p = &i;
```

- *p は変数 i の別名としてふるまう
- これ以降のプログラムコードに*pが登場したときには、これを変数iに置き換えて読むと理解しやすくなる

- 「int j = *p;」は、「int j = i;」と同じように働き、変数jに変数iの値が代入される

- 「*p = 10;」は「i = 10;」と同じように働き、変数iの値が10になる。

- つまり、**ポインタを使って、変数の値を変更できる**

ポインタを使って値を変更する

```
int i = 5;
int *p = &i;

int j = *p;
printf("jの値は%d¥n", j);
*p = 10;
printf("iの値は%d¥n", i);
```

実際に試してみよう

実行結果

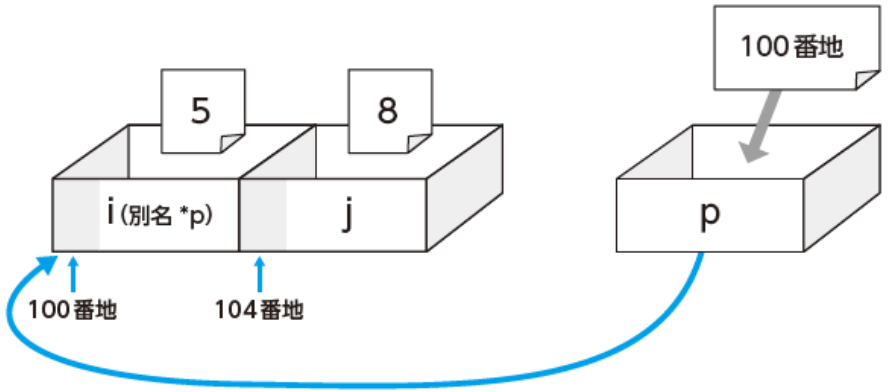
```
jの値は5
iの値は10
```

- 変数iと*pは、名前が違うだけで、まったく同じようにふるまう。
- *pはiの**エイリアス（別名）**であるという。

ポインタが指す先を変更する

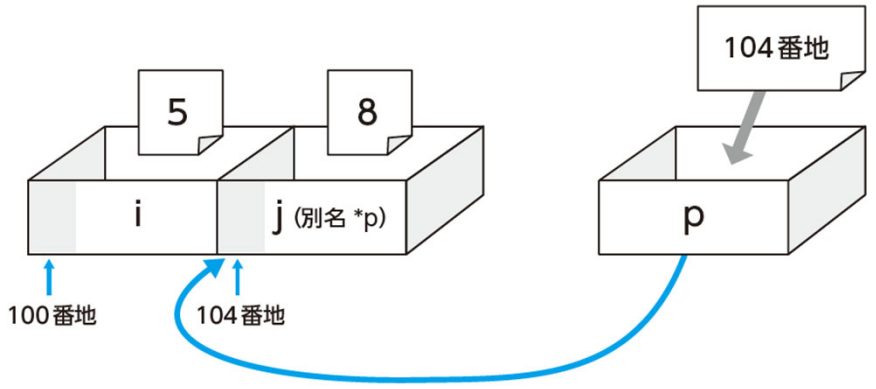
```
int i = 5;
int j = 8;

int *p = &i;
printf("*pの値は%d¥n", *p);

p = &j;
printf("*pの値は%d¥n", *p);
```

ポインタ変数に変数jのアドレスを代入



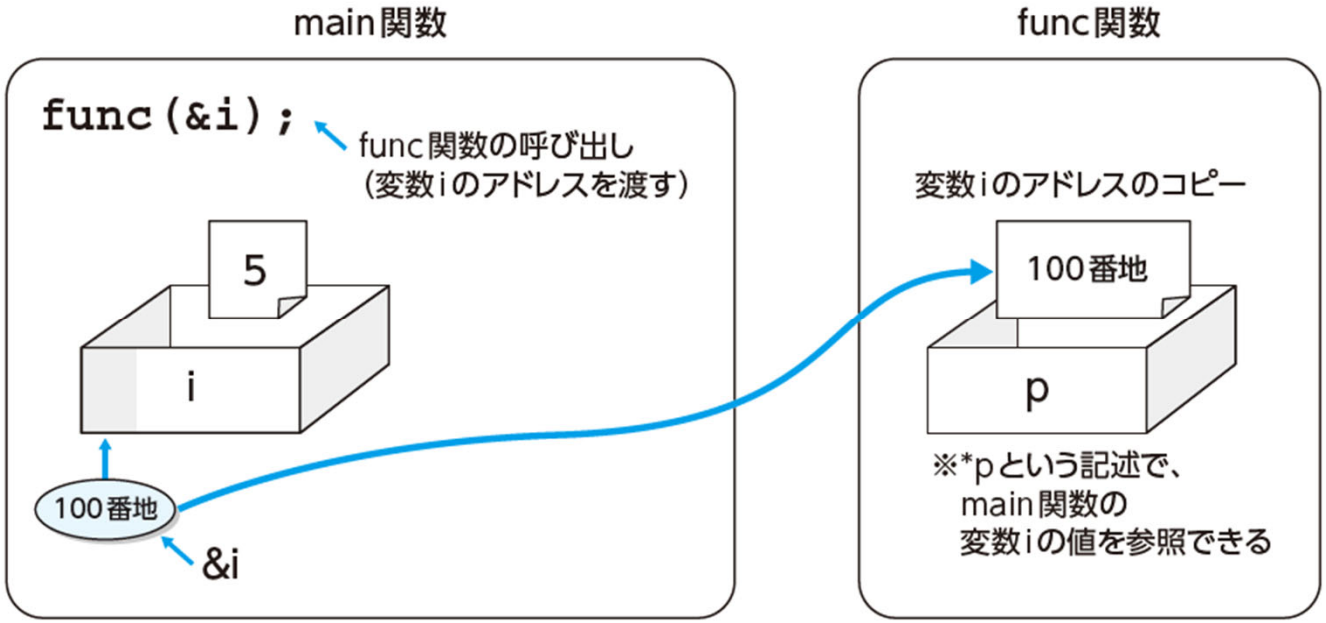
実行結果

```
*pの値は5
*pの値は8
```



ポインタを引数とする関数

```
#include <stdio.h>

void func(int *p)
{
    *p = 100;
}

int main(void)
{
    int i = 5;
    func(&i);
    printf("iの値は%d¥n", i);
}
```



実行結果

```
iの値は100
```

実際に試してみよう

関数にアドレスを渡すと、そのアドレスに格納されている値を、関数の中で変更できる

値の交換をする swap 関数

```c
#include <stdio.h>

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(void)
{
    int a = 2;
    int b = 3;
    printf("a=%d, b=%d¥n", a, b);
    printf("swap関数を呼び出します¥n");
    swap(&a, &b);
    printf("a=%d, b=%d¥n", a, b);
}
```

↓ 実行結果

```
a=2, b=3
swap関数を呼び出します
a=3, b=2
```

実際に試してみよう

配列とポインタ

配列の要素のアドレス

```
int scores[3] = {10, 20, 30};
printf("scores[0]のアドレス: %p¥n", &scores[0]);
printf("scores[1]のアドレス: %p¥n", &scores[1]);
printf("scores[2]のアドレス: %p¥n", &scores[2]);
```

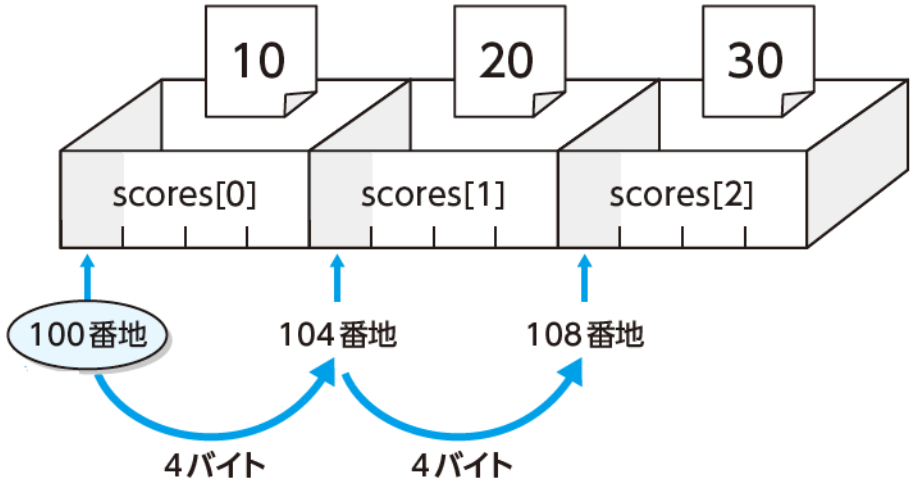
実行結果

```
scores[0]のアドレス: 012FFD90
scores[1]のアドレス: 012FFD94
scores[2]のアドレス: 012FFD98
```

4だけ増えています

4だけ増えています

- 配列の要素は、メモリ上に連続して格納される



配列とポインタ

```
int scores[3] = {10, 20, 30};
```

```
int *p = &scores[0];
```

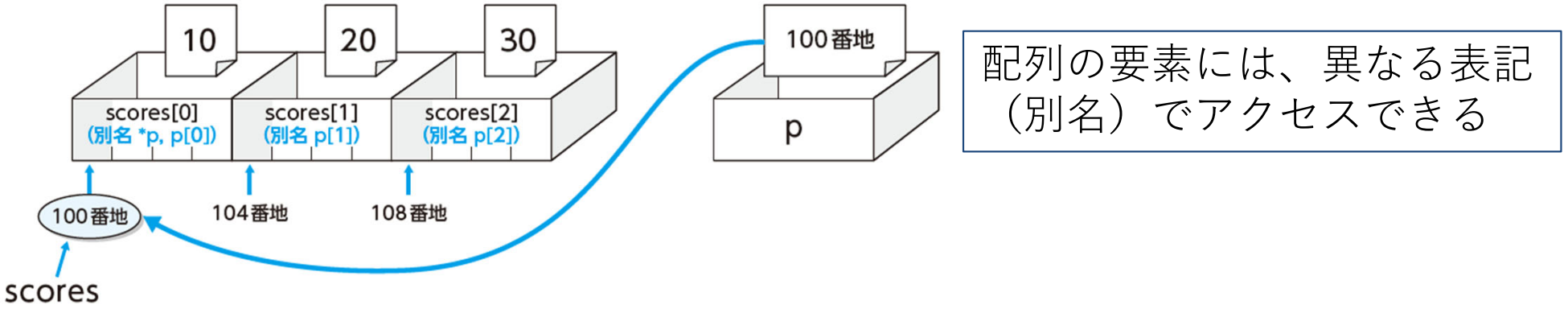
```
int *p = scores;
```

- どちらの表記も同じ意味
- **配列名だけの表記は、先頭要素のアドレスを表す**

```
scores[0] = 5;
```

```
*p = 5;
```

どちらの表記も同じ意味



配列の要素には、異なる表記（別名）でアクセスできる

ポインタに対する加算

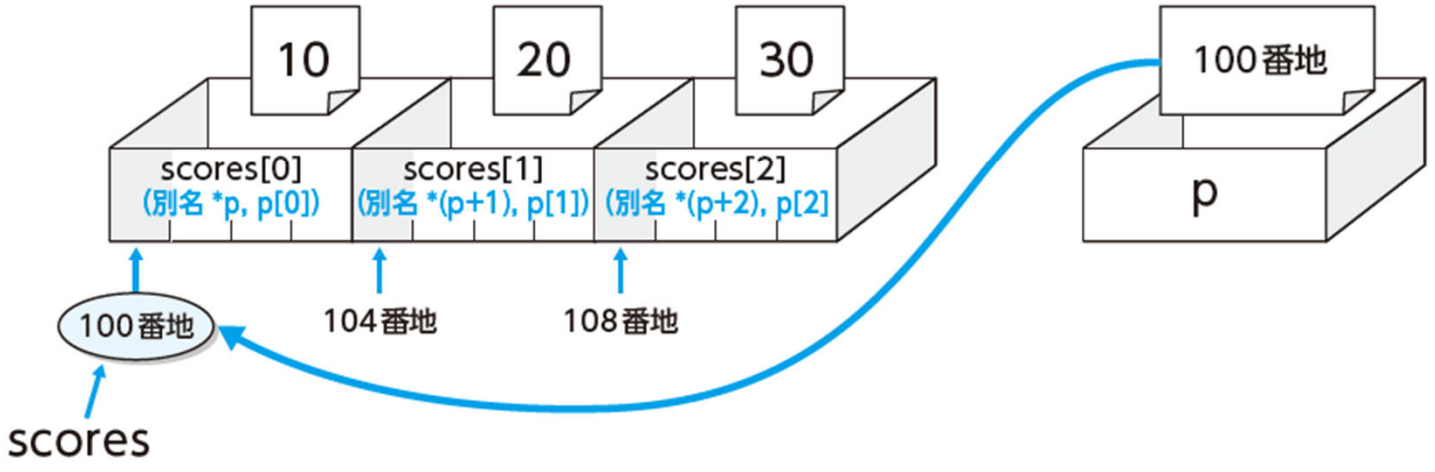
```
int scores[] = { 10, 20, 30 };
int *p = scores;

printf("*pの値は%d¥n", *p);
printf("*(p + 1)の値は%d¥n", *(p + 1));
printf("*(p + 2)の値は%d¥n", *(p + 2));
```

↓実行結果

```
*pの値は10
*(p + 1)の値は20
*(p + 2)の値は30
```

- ポインタ変数に対して1だけ加算すると、型のサイズ分だけ、アドレスの値が増える

「int *p = scores;」とした後



ポインタ変数のインクリメント

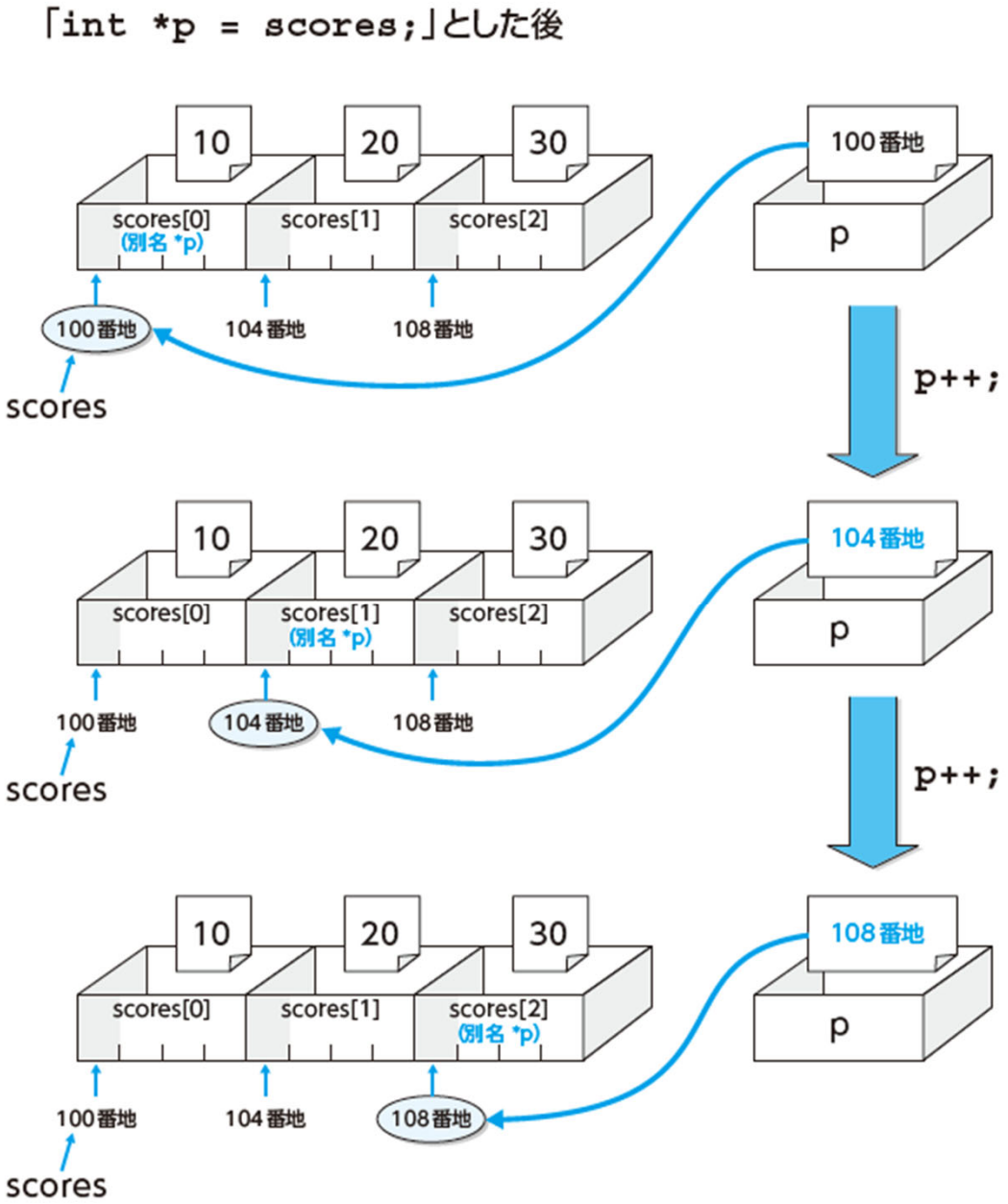
ポインタが配列の先頭の要素を指しているとき、ポインタ変数の値を1だけ増やすと、配列の次の要素を指すようになる。

```
int scores[] =
      { 10, 20, 30 };
int *p = scores;

printf("%d¥n", *p);
p++;
printf("%d¥n", *p);
p++;
printf("%d¥n", *p);
```

↓ 実行結果

```
10
20
30
```

実際に試してみよう



関数の引数に配列を使う

```
#include <stdio.h>

void func(int *p)
{
    p[0] = 98;
    p[1] = 99;
    p[2] = 100;
}

int main(void)
{
    int scores[] = { 10, 20, 30 };
    func(scores);

    for (int i = 0; i < 3; i++) {
        printf("scores[%d]の値は%d¥n", i, scores[i]);
    }
}
```

引数を int p[] と書くこともできる

あたかもp自体が配列かのように扱える

配列の先頭の要素のアドレスをfunc関数に渡している

実行結果

```
scores[0]の値は98
scores[1]の値は99
scores[2]の値は100
```

関数ポインタ

関数のアドレスと関数ポインタ

- 関数もメモリーに格納される
- **関数にもアドレスがある**
- 関数のアドレスは、関数名を書くだけで参照できる
  （アドレス演算子「&」をつける必要は無い）

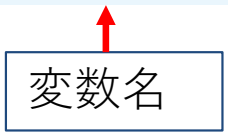
```
printf("関数funcのアドレスは%p¥n", func);
```

- 関数のアドレスを格納するための変数は、次のように宣言する。

```
関数の戻り値の型 (*変数名) (関数の引数の型);
```

例 「戻り値なし」「引数がint型の値」である関数のアドレスを格納するための変数

```
void (*pF) (int);
```



関数ポインタを使用する例

```
#include <stdio.h>

void func(int a)
{
    printf("func関数が呼び出されました。引数の値は%d¥n", a);
}

int main(void)
{
    void (*pF)(int) = func;
    (*pF)(10);
}
```

実行結果

```
func関数が呼び出されました。引数の値は10
```

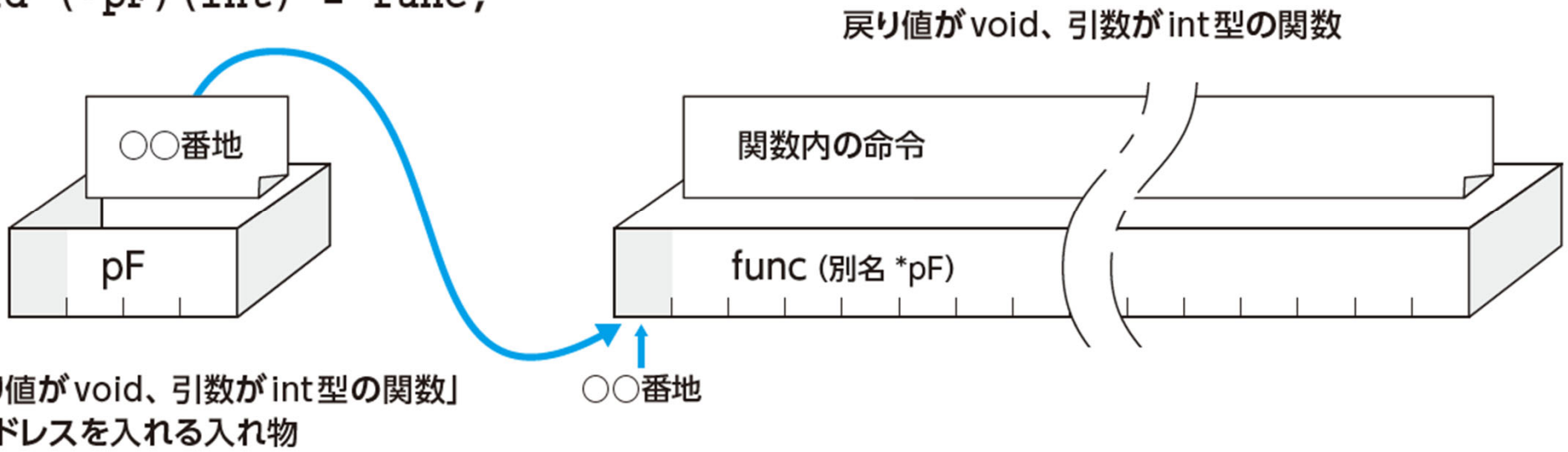
```
func(10);
```

```
(*pF)(10);
```

どちらの表記も同じ意味

関数のアドレスと関数ポインタのイメージ

```
void (*pF)(int) = func;
```



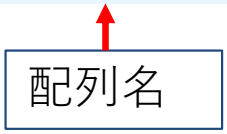
関数ポインタの配列

- 関数のアドレスを格納する配列の宣言

```
関数の戻り値の型 (*配列名[]) (関数の引数の型);
```

例 「戻り値なし」「引数なし」である関数のアドレスを格納する配列の宣言

```
void (*pFs[]) ();
```



関数ポインタの配列の例

```
#include <stdio.h>

void func_dog() { printf("わんわん¥n"); }

void func_cat() { printf("にゃー¥n"); }

void func_pig() { printf("ブーブー¥n"); }

int main(void)
{
    void (*pFs[])() = { func_dog, func_cat, func_pig };
    for (int i = 0; i < 3; i++) {
        (*pFs[i])();
    }
}
```

↓ 実行結果

```
わんわん
にゃー
ブーブー
```

高階関数

- 関数のアドレスを引数として受け取る関数を**高階関数**という

高階関数の例

```
void func(void (*pF)(int))
{
    (*pF)(10);
}
```

「戻り値無し」「引数がint型の値」の関数のアドレスを変数pFで受け取る

pFが指す『「戻り値無し」「引数がint型の値」の関数』に、値10を渡す

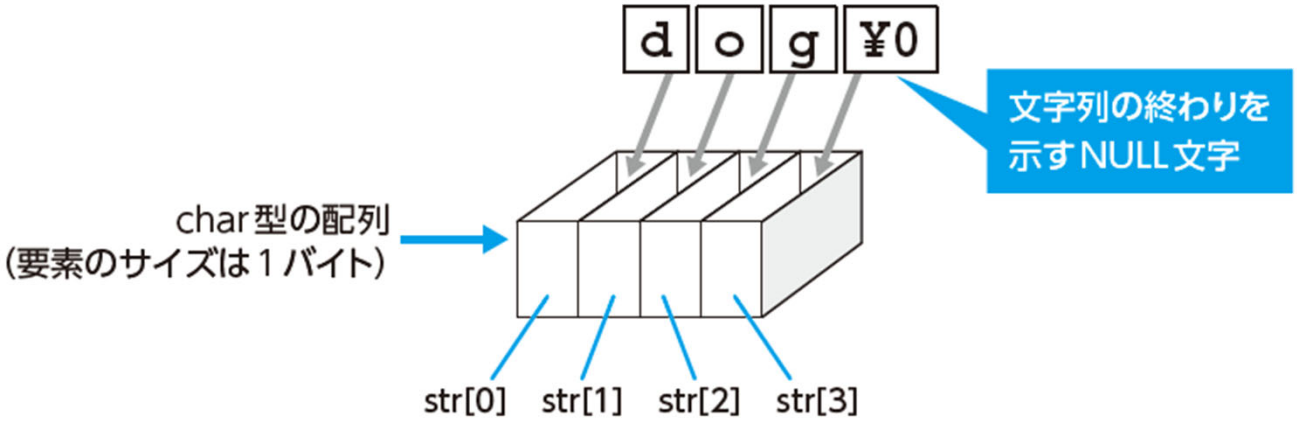
第6章　文字列の扱いと構造体

文字列と配列

配列への文字列の格納

- 文字列はchar型の配列に1文字ずつ格納する
- 文字列の末尾には、**NULL文字（'¥0'）**を格納する決まりがある

```
char str[4];
str[0] = 'd';
str[1] = 'o';
str[2] = 'g';
str[3] = '¥0';
```



- 3文字の"dog"を格納するには、末尾にNULL文字を入れることを考慮して、4以上の要素を持つchar型の配列を準備する

※ 扱う文字はchar型で表現可能な半角英数記号に限定する

char型の配列に格納された文字列の出力

- printf関数でchar型の配列に格納された文字列を出力するには、変換指定に%sを使用する

```
char str[4];
str[0] = 'd';
str[1] = 'o';
str[2] = 'g';
str[3] = '¥0';

printf("犬は英語で %s¥n", str);
```

↓実行結果

```
犬は英語で dog
```

char型の配列の初期化

char型の配列に文字列を格納するには、複数の方法がある

```
char str[4] = {'d', 'o', 'g', '¥0'};
```

```
char str[4] = "dog";
```

```
char str[] = "dog";
```

どれでも同じ

注意：このような書き方ができるのは、char型の配列の宣言と同時に代入を行う**初期化時だけ**

```
char str[4];
str = "dog";
```

← **コンパイルエラー**

実際に試してみよう

標準入力から文字列を受け取る

```
#include <stdio.h>

int main(void)
{
    char str[32];
    printf("名前を入力してください¥n");
    scanf_s("%s", str, 32);
    printf("%sさん、こんにちは¥n", str);
}
```

実行結果

```
名前を入力してください
jun
junさん、こんにちは
```

- 入力される文字列を格納するchar型の配列は、十分な要素数を持つようにする
- scanf_s関数を使って文字列の入力を受け取る

実際に試してみよう

※ Visual Studio 以外では scanf_s関数が使えないことがある。その場合はscanf関数を使用する

文字列の操作

文字列へのポインタ

- 文字列の扱いは配列の扱いと同じ
- 関数に文字列を渡すときには、先頭の要素のアドレスを渡す

（復習）配列名だけを書くと、その配列の先頭の要素のアドレスを表す

```
#include <stdio.h>

void func(char *s)
{
     printf("引数で渡された文字列: %s¥n", s);
}

int main(void)
{
    char str[] = "dog";
    func(str);
}
```

文字列の配列

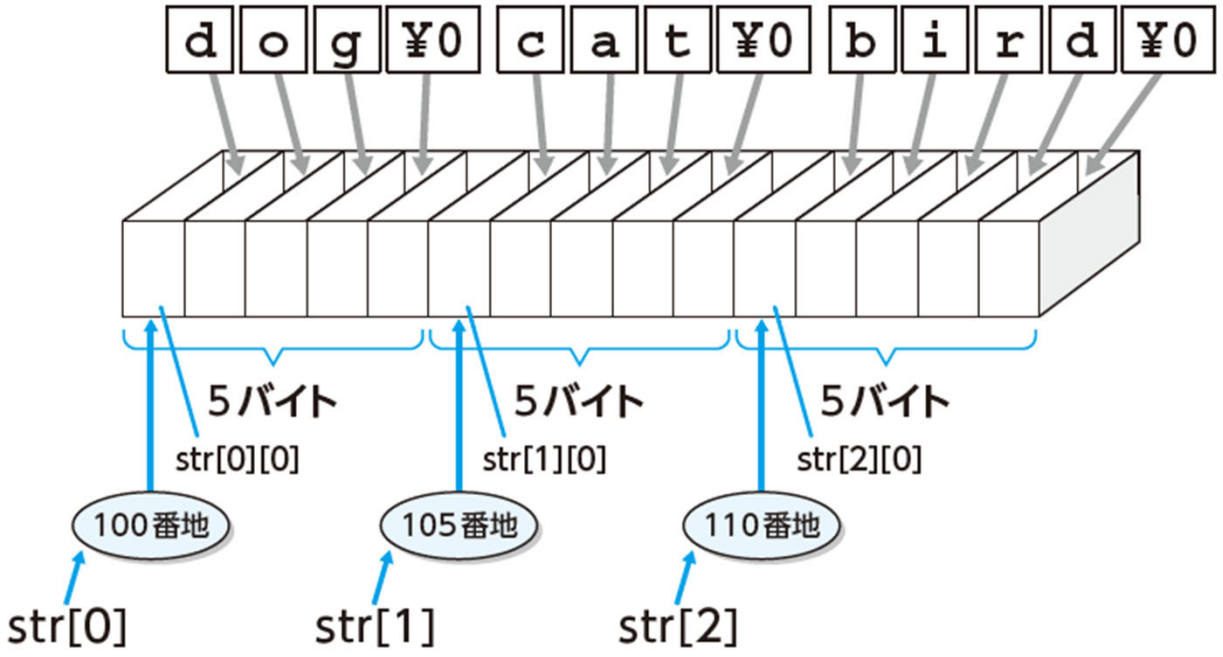
文字列の配列は、**配列の配列**

```
char strs[3][5] = { "dog", "cat", "bird" };
for (int i = 0; i < 3; i++) {
    printf("%s¥n", strs[i]);
}
```

実際に試してみよう

実行結果

```
dog
cat
bird
```



string.h のインクルード

string.h をインクルードすることで、文字列操作を行うための様々な関数を使用できるようになる

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

文字列操作を行う関数のプロトタイプ宣言が書かれている

例：

- 文字列のコピー strcpy_s （または strcpy）関数
- 文字列の比較 strcmp 関数
- 長さの取得 strlen関数
- 文字列の連結 strcat_s （または strcat）関数
- 文字列の書き出し sprintf_s （または sprintf）関数

ワン・モア・ステップ
Visual Studio 独自の「_s」関数

- strcpy_s関数のように末尾に「_s」がつく関数は、Visual Studioを開発したMicrosoft社が独自に作った関数
- 引数に「文字列を格納する配列の要素数」の指定が追加されることで、オーバーフロー（またはオーバーラン）を未然に防げる

- Visual Studio では、C言語標準の strcpy 関数を使うと、コンパイルエラーになるが

```
#define _CRT_SECURE_NO_WARNINGS
```

という1行を追加することで、コンパイルエラーにならないようにできる

文字列操作のための関数の使用例

・strcpy_s 関数による文字列の代入

```
char str[4];
strcpy_s(str, 4, "dog");
printf("%s", str);          // dog
```

・strcpy_s 関数による文字列のコピー

```
char str1[] = "Hello";
char str2[16];
strcpy_s(str2, 16, str1);
printf("%s", str2);         // Hello
```

・strcmp 関数による文字列の比較

```
char str1[] = "Hello";
char str2[] = "HELLO";
if (strcmp(str1, str2) == 0) {
    printf("同じ");
} else {
    printf("違う"); // こちらが出力される
}
```

文字列操作のための関数の使用例

・strlen 関数による文字数の取得

```
char str[32] = "Hello";
int len = strlen(str);
printf("%d", len); // 5
```

・strcat_s 関数による文字列の連結

```
char str[32] = "Hello";
strcat_s(str, 32, " world!");
printf("%s", str); // Hello world!
```

・sprintf_s関数による文字列への出力

```
char str[32];
char name[] = "Taro";
sprintf_s(str, 32, "Hello %s!", name);
printf("%s", str); // Hello Taro!
```

構造体

一緒に管理したい情報

- 自分で新しい型を定義できる ← **構造体**
- 構造体には「整数」と「文字列」といった複数の値をもたせることができる
- 構造体の名前には `struct` というキーワードをつける

名前（`char`型の配列）と年齢（`int`型）の情報を持つ、
`struct Person` という名前の構造体

```
struct Person
{
    char name[16];
    int age;
};
```

構造体の**メンバ**

構造体

```
struct Person
{
    char name[16];
    int age;
};
```

- 構造体 `struct Person` 型の変数の宣言と初期化

```
struct Person a = { "SUZUKI TARO", 17 };
```

- メンバ変数の参照

```
printf("%s¥n", a.name);
printf("%d¥n", a.age);
```

構造体の変数名.メンバ変数名　という表記でメンバ変数を参照できる

構造体の使用例

```
#include <stdio.h>

struct Person
{
    char name[16];
    int age;
};

int main(void)
{
    struct Person a = { "SUZUKI TARO", 17 };
    struct Person b = { "SATO HANAKO", 19 };

    printf("名前:%s, 年齢:%d¥n", a.name, a.age);
    printf("名前:%s, 年齢:%d¥n", b.name, b.age);
}
```

↓ 実行結果

```
名前:SUZUKI TARO, 年齢:17
名前:SATO HANAKO, 年齢:19
```

実際に試してみよう

typedef で型に別名をつける

・typedef 宣言によって、長い名前の型名を短くしたり、覚えやすい名前にできる

typedef 宣言

```
typedef 型名 別の呼び名;
```

例：int を seisuu と表記できるようにする

```
#include <stdio.h>

typedef int seisuu;

int main(void)
{
	seisuu i = 10;
	printf("iは%d¥n", i);
}
```

構造体の定義とtypedef宣言

構造体の定義とtypedef宣言を同時に行うことができる

```
typedef struct Person
{
    char name[16];
    int age;
} Person;
```

← `struct Person` の別名

`struct Person` 型の変数の宣言を次のように表記できる

```
Person a = { "SUZUKI TARO", 17 };
```

実際に試してみよう

構造体の応用

構造体と関数

これまでに学習した他の型と同じように構造体を関数の引数にできる

```
void print_info(Person p)
{
    printf("名前:%s, 年齢:%d¥n", p.name, p.age);
}
```

関数の中でメンバ変数の値を変更したい場合は、ポインタ渡し（アドレスを渡す）にする

```
void birthday(Person *p)
{
    (*p).age++;
}
```

アロー演算子（ ->）

```
void birthday(Person *p)
{
    (*p).age++;
}
```

どちらの表記でも動作は同じ

```
void birthday(Person *p)
{
    p->age++;
}
```

値渡しとアドレス渡し

```
void func(Person p)
{
    printf("%d", p.age);
}
```

Person型の変数の**複製**が
変数pに代入される

○ pのメンバ変数を参照する
× pのメンバ変数を変更する
（呼び出しもとには影響しない）

```
void func(Person *p)
{
    printf("%d", (*p).age);
    (*p).age++;
}
```

Person型の変数の**アドレス**が
変数pに代入される

○pのメンバ変数を参照する
○pのメンバ変数を変更する

※ 値を参照するだけであっても、
複製を行う処理が生じないので、
より効率的

const 修飾子

値が変化しない変数であることを明示するためにconst修飾子を使用する

```
const int a = 10;
a++;
```

変数aの値は変更できない

コンパイルエラー

メンバ変数の値を変更できない

```
void func(const Person *p)
{
    p->age++;
}
```

コンパイルエラー

実際に試してみよう

※ この関数にアドレスを渡しても、メンバ変数の値が変更されることが無いことを知ることができる

構造体と配列

構造体も他の型と同じように配列に格納できる

```
Person p[2];
```

次のようにして、宣言と初期化を同時にできる

```
Person p[] = { { "SUZUKI TARO", 17 },
               { "SATO HANAKO", 19 } };
```

配列の要素へのアクセス

```
for (int i = 0; i < 2; i++) {
    printf("名前:%s, 年齢:%d¥n", p[i].name, p[i].age);
}
```

ポインタを含む構造体

```
#include <stdio.h>

typedef struct Person
{
    char name[16];
    int age;
    struct Person *mother;
} Person;

int main(void)
{
    Person p0 = { "SUZUKI AKIKO", 44, NULL };
    Person p1 = { "SUZUKI TARO", 17, &p0 };

    printf("%s の母親は %s¥n", p1.name, p1.mother->name);
}
```

struct Person *型のポインタ変数

実行結果

```
SUZUKI TARO の母親は SUZUKI AKIKO
```

実際に試してみよう

第7章　一歩進んだC言語プログラミング

ファイル入出力

テキストファイルの読み込み

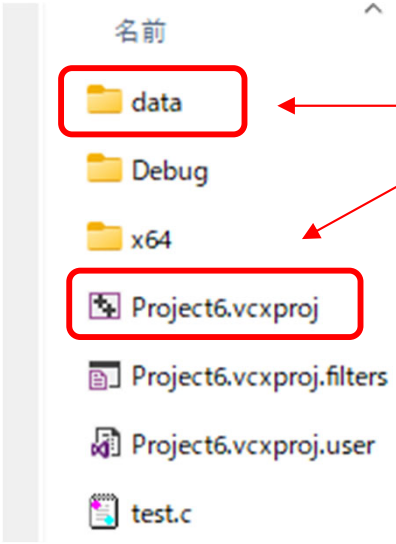
- **テキストファイル**：文字列が保存されたファイル
- **バイナリファイル**：テキストファイルではないファイル（画像データ、音声データなどのファイル）

- テキストファイルの読み込みの手順
  1. ファイルを開く（fopen_s関数〔またはfopen関数〕を使用）
  2. ファイルから文字列を読み込み、処理を行う（fgets関数などを使用）
  3. ファイルを閉じる（fclose関数を使用）

ファイルの読み込みとファイルの保存場所

```
FILE *fp = NULL;
fopen_s(&fp, "data/sample.txt", "r")
```

ファイルの場所（相対パス）の指定



プロジェクトファイル（.vcxproj）があるフォルダが基準

sample.txt の中身

```
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty together again.
```

テキストファイル読み込みの例

```c
#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    char line[128];

    if (fopen_s(&fp, "data/sample.txt", "r") != 0) {
        printf("ファイルを開けませんでした¥n");
        return 1;
    }

    while (fgets(line, 128, fp) != NULL) {
        printf(">>> %s", line);
    }

    fclose(fp);
    return 0;
}
```

ファイルポインタ（ストリームとも呼ぶ）

モード指定

実際に試してみよう

モード	説明
"r"	読み込み用に開く（ファイルが存在しないときはエラー）
"w"	書き込み用に開く（ファイルが存在しないときは新規作成）
"a"	追記用に開く（ファイルの末尾に追記。ファイルが存在しないときは新規作成）
"r+"	読み書き両用にする

テキストファイルに書かれたデータの読み込み

```
FILE *fp = NULL;
char line[128];

if (fopen_s(&fp, "data/scores.txt", "r") != 0) {
    printf("ファイルを開けませんでした¥n");
    return 1;
}

int total = 0;
int i;
while (fscanf_s(fp, "%d", &i) != EOF) {
    printf("%d を読み込みました¥n", i);
    total += i;
}

printf("合計は %d¥n", total);
fclose(fp);
return 0;
```

data/scores.txt

```
12
56
90
74
100
31
```

テキストファイルに書かれている数字を1つずつ読み込む

実際に試してみよう

テキストファイルの書き出し

- テキストファイルの書き出しの手順
  1. ファイルを開く（fopen_s関数〔またはfopen関数〕を使用）
  2. 文字列をファイルに書き出す（fprintf関数などを使用）
  3. ファイルを閉じる（fclose関数を使用）

書き出しのモード指定

```
FILE *fp;
if (fopen_s(&fp, "output.txt", "w") != 0) {
    printf("ファイルを開けませんでした¥n");
    return 1;
}

for (int i = 0; i < 10; i++) {
    fprintf(fp, "[%d]¥n", i);
}

fclose(fp);
return 0;
```

実際に試してみよう

グローバル変数と
複数ファイルへの分割

グローバル変数

- 変数が使用できる範囲は、宣言されたブロックの中に限られる（**変数のスコープ**）
- 関数の中で宣言した変数、または引数の受け取りのために使用する変数を**ローカル変数**とよぶ
- プログラム全体で使用できる**グローバル変数**というものがある
- グローバル変数は、関数の外側で宣言し、どの関数の中からも使用できる

```
#include <stdio.h>

int a = 10;

void func(void) {  a++;  }

int main(void)
{
    func();
    printf("%d¥n", a);
}
```

グローバル変数。プログラム全体のどこからでも参照できる

実際に試してみよう

複数のファイルに分ける

triangle.h (関数のプロトタイプ宣言)

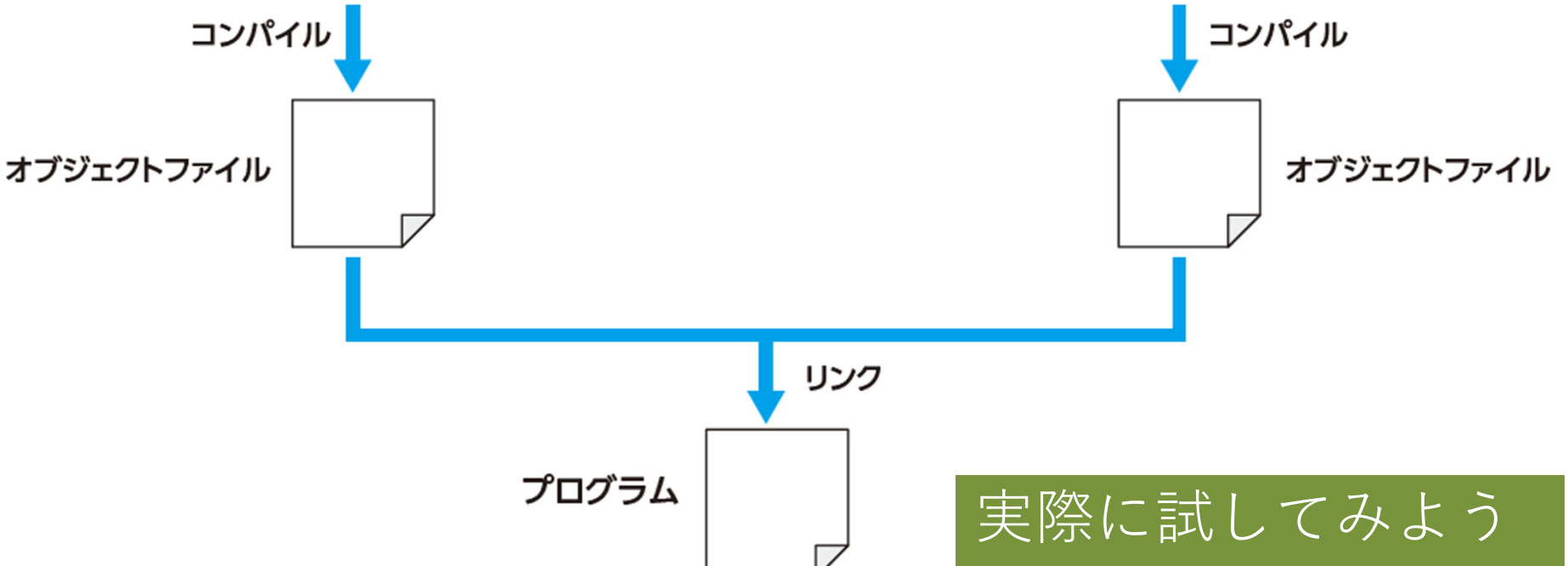
```
double triangle_area(double base,
                     double height);
```

triangle.c (関数の定義)

```
double triangle_area(double base, double height)
{
    return base * height / 2;
}
```

main.c

```
#include <stdio.h>
#include "triangle.h"

int main(void)
{
    double area = triangle_area(5, 8);
    printf("area=%f¥n", area);
}
```



実際に試してみよう

マクロと列挙

プリプロセッサの処理

- コンパイルが行われる前に、プログラムコードを自動編集する**プリプロセス**と呼ばれる操作が行われる。
- この操作を行うプログラムを**プリプロセッサ**と呼ぶ。

- 例： インクルード命令

```
#include <ファイル名>
```

インクルード文の場所に、指定されたファイルの中身を埋め込む

- 例：マクロ（置換処理）

```
#define 置き換えられる文字列 置き換える文字列
```

コンパイル前に文字列の置き換えが行われる

マクロの例

```
#define NUM 10

int x[NUM]
int y[NUM]
```

これ以降の「NUM」を「10」に置き換える

```
#define PI 3.14159265359
```

これ以降の「PI」を
「3.14159265359」に置き換える

関数形式マクロ

- マクロと関数の仕組みを組み合わせたもの

```
#define MAX(x, y) (x > y ? x : y)
```

```
MAX(x, y)
```



```
(x > y ? x : y)
```

置き換えられる

```
MAX(a, b)
```



```
(a > b ? a : b)
```

置き換えられる

```
MAX(p0.age, a1.age)
```



```
(p0.age > p1.age ? p0.age : p1.age)
```

置き換えられる

条件付きコンパイル

- 条件に応じてコンパイル対象となるプログラムコードを切り替えることができる

```
#ifdef マクロ名

プログラムコード

#endif
```

マクロ名が定義されている（「#define マクロ名」が存在する）場合のみ、コンパイル対象となる

```
#include <stdio.h>
#define ADDRESS_CHECK

int main(void)
{
    int a = 9;
#ifdef ADDRESS_CHECK
    printf("aのアドレスは%p¥n", &a);
#endif
    printf("aの値は%d¥n", a);
}
```

実際に試してみよう

列挙型

- **列挙型**という、限られた値のみ格納できる型がある
- 列挙型は、構造体と同じように、自分で新しく定義する型
- 列挙型は、自分で定義したキーワード（識別子）を格納するための型

- enum というキーワードを最初につける必要がある

例

```
enum Size { S, M, L, XL };
```

「enum Size」が型名

S, M, L, XL のいずれかの値をとることができる

```
enum Size a = XL;
```

実際に試してみよう

第8章　データ構造とアルゴリズム

アルゴリズムと計算量

アルゴリズムと計算量

- 目的を達成するための手順のことを**アルゴリズム**という
- アルゴリズムによる効率の違いを**計算量**によって示す
- データの大きさをnで表したときに、計算回数や計算時間がnに比例するとき、計算量を$O(n)$と表記する（**オーダー表記**）

線形探索と2分探索

目的：昇順に並んだデータ列から、目的の値の場所を見つける

- **線形探索**：前から順にデータを見ていく
  $O(n)$の計算量

- **2分探索**：対象を半分に絞りながら見ていく
  $O(\log n)$の計算量

ワン・モア・ステップ！　計算量と計算時間

計算量と計算時間の関係

n	$O(n)$	$O(\log n)$	$O(n \log n)$	$O(n^2)$
1	1ミリ秒	1ミリ秒	1ミリ秒	1ミリ秒
1,000	1秒	10ミリ秒	10秒	17分
1,000,000	17分	20ミリ秒	6時間	31年
1,000,000,000	12日	30ミリ秒	346日	3171万年

線形探索の例

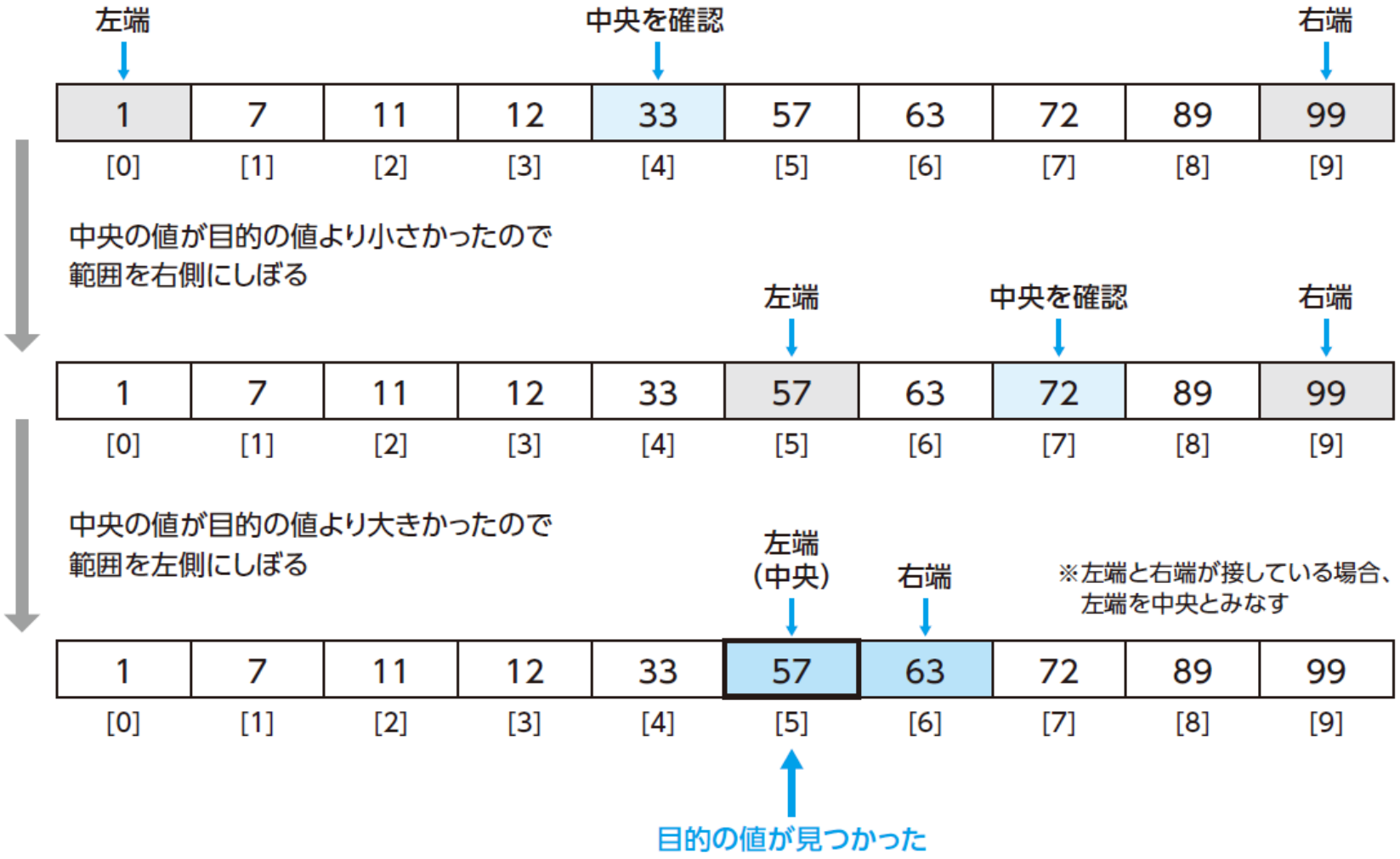
```
#include <stdio.h>

#define DATA_NUM 10
int data[] = { 1, 7, 11, 12, 33, 57, 63, 72, 89, 99 };

int main(void)
{
    int target = 57;
    for (int i = 0; i < DATA_NUM; i++) {
        if (data[i] == target) {
            printf("data[%d] に見つかりました¥n", i);
            return 0;
        }
    }

    printf("見つかりませんでした¥n");
}
```

for文で先頭から順に調べていく

実際に試してみよう

2分探索アルゴリズム

例： 57を見つけたい



```c
#include <stdio.h>

#define DATA_NUM 10
int data[] = { 1, 7, 11, 12, 33, 57, 63, 72, 89, 99 };

int main(void)
{
    int target = 57;
    int left_index = 0;
    int right_index = DATA_NUM - 1;
    int middle_index = 0;

    while (left_index <= right_index) {
        middle_index = (left_index + right_index) / 2;

        printf("left:%d, middle:%d, right:%d¥n",
            left_index, middle_index, right_index);

        if (data[middle_index] == target) {
            printf("値 %d は data[%d] に見つかりました¥n",
                target, middle_index);
            return 0;
        } else if (data[middle_index] < target) {
            left_index = middle_index + 1;
        } else {
            right_index = middle_index - 1;
        }
    }

    printf("値 %d は見つかりませんでした¥n", target);
}
```

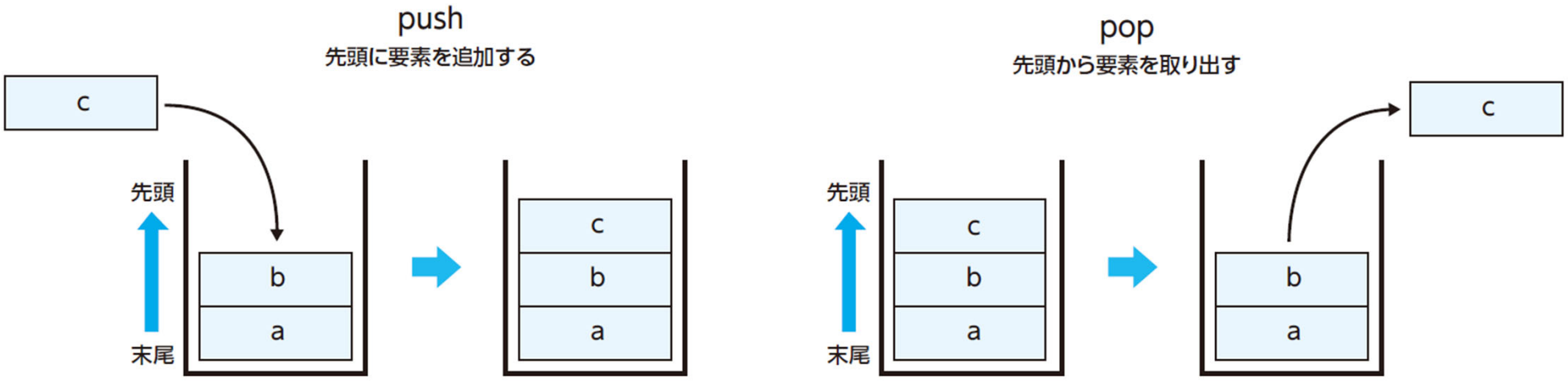
探索範囲を半分に絞っていく

データの格納

配列とスタック

- データの格納方法の1つに**スタック**がある
- 後に入れたものほど先に取り出される（**後入れ先出し**）

- push操作：先頭に要素を追加する
- pop操作：先頭から要素を取り出す

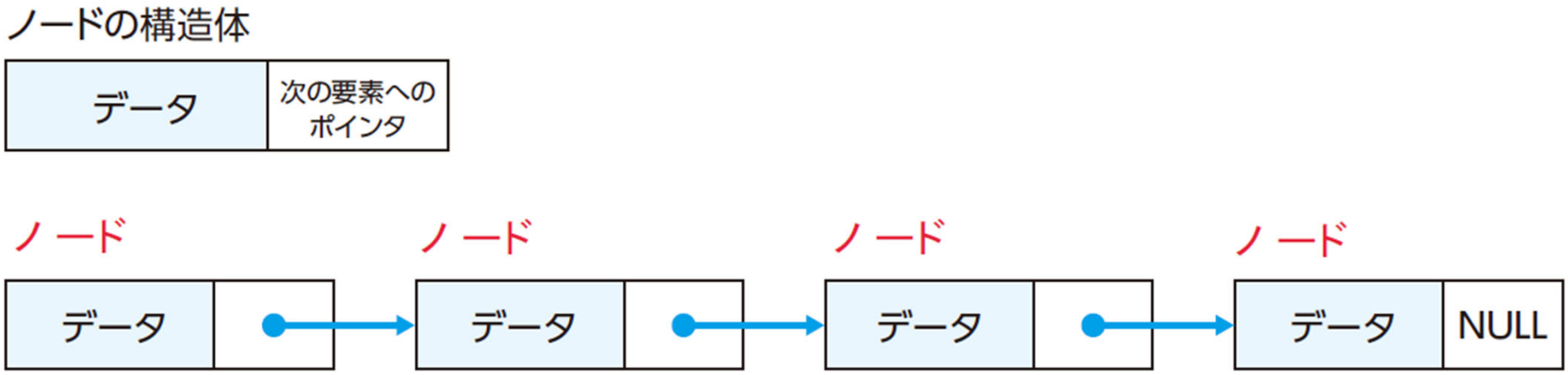


スタックの例

```
#include <stdio.h>
#define STACK_SIZE 5

int stack[STACK_SIZE];
int head_index = -1;

void push(int value) {
    head_index++;
    stack[head_index] = value;
}

int pop() {
    int value = stack[head_index];
    head_index--;
    return value;
}

void main() {
    push(5);    // [5][][][][]
    push(9);    // [5][9][][][]
    push(3);    // [5][9][3][][]
    printf("pop %d¥n", pop()); // 3,  [5][9][3][][]
    printf("pop %d¥n", pop()); // 9,  [5][9][3][][]
    printf("pop %d¥n", pop()); // 5,  [5][9][3][][]
}
```

リスト

- 複数のデータを管理する仕組み
- データを各**ノード**に格納する
- ノードは、次のノードへのポインタを併せ持つ

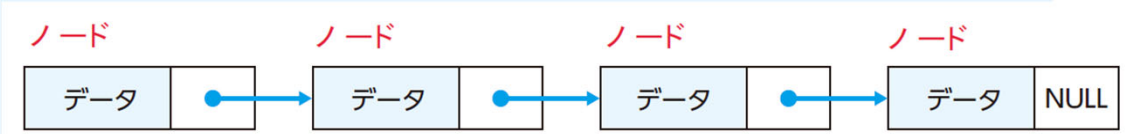


リストの例

```
#include <stdio.h>

typedef struct Node {
    char str[256];
    struct Node *next;
}Node;

Node *head = NULL;

void printout(void) {
    Node *node = head;
    while (node != NULL) {
        printf("[%s]", node->str);
        node = node->next;
    }
    printf("¥n");
}

int main(void) {
    Node n0 = { "This", NULL };
    Node n1 = { "is", NULL };
    Node n2 = { "a", NULL };
    Node n3 = { "cat", NULL };
    head = &n0;
    n0.next = &n1;
    n1.next = &n2;
    n2.next = &n3;
    printout();
    Node new_node = { "not", &n2 };
    n1.next = &new_node;
    printout();
}
```

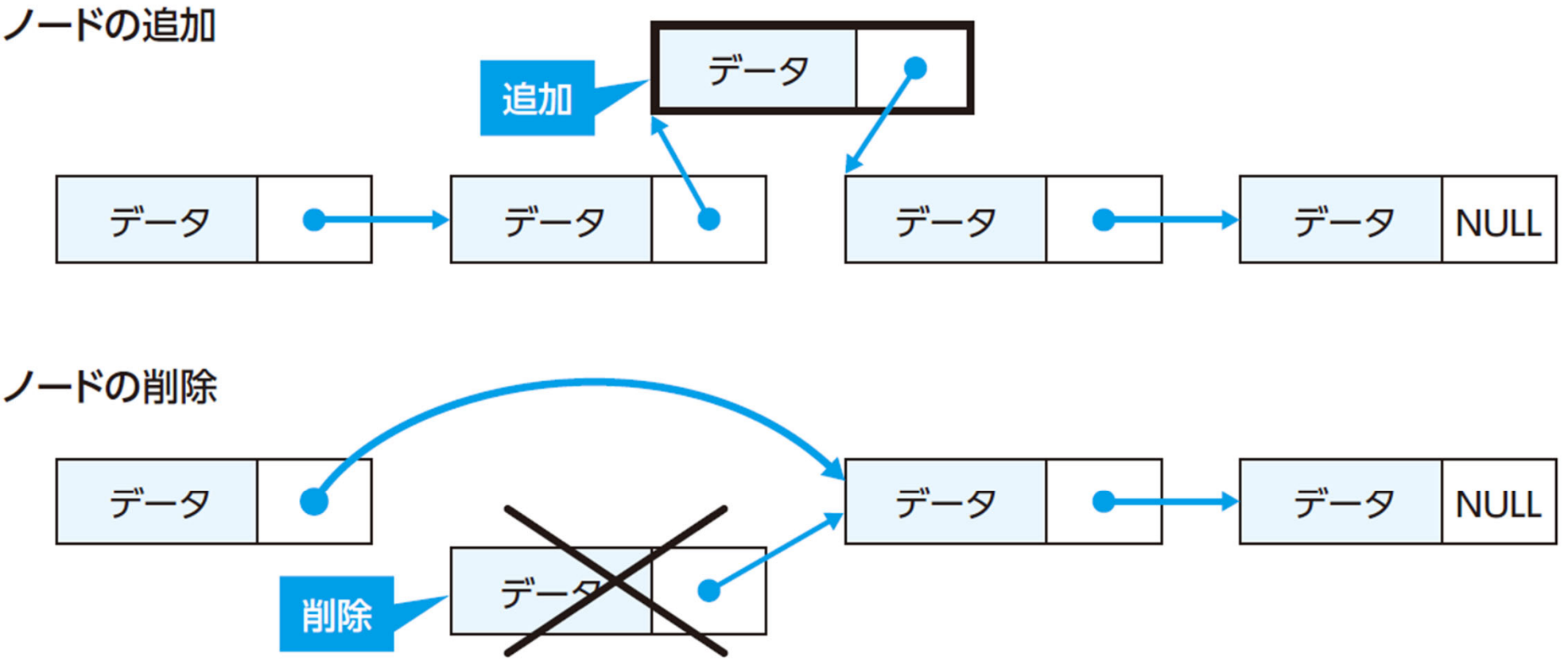
ノードの構造体

ノードを順にたどっていく



新しいノードの追加

リストでのノードの追加と削除



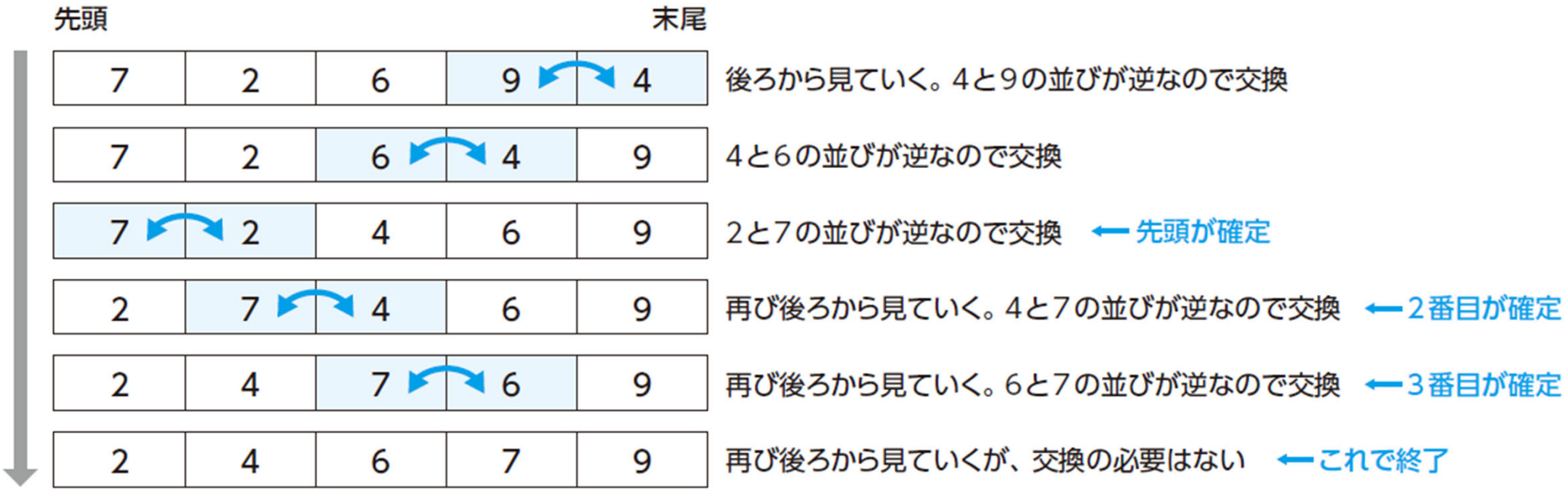
整列（ソート）

ソート

- データを順序に従って並べ替える処理を**ソート**という
- もっとも単純なソートアルゴリズムの1つに**バブルソート**がある
- 計算量は$O(n^2)$で、効率は悪い

バブルソートの仕組み

例：初期状態が 7, 2, 6, 9, 4 である数字の列を昇順に並び替える様子



バブルソートの例

```c
#include <stdio.h>

#define DATA_NUM 5
int data[] = { 7, 2, 6, 9, 4 };

void printout() {
    for (int i = 0; i < DATA_NUM; i++) {
        printf("%d ", data[i]);
    }
    printf("¥n");
}

int main(void) {
    printout();

    for (int i = 0; i < DATA_NUM - 1; i++) {
        for (int j = DATA_NUM - 1; j > i; j--) {
            if (data[j] < data[j - 1]) {
                int tmp = data[j];
                data[j] = data[j - 1];
                data[j - 1] = tmp;
                printout();
            }
        }
    }

    printf("---¥n");
    printout();
}
```

2つの値の交換

二重ループで並べ替えを行う

実行結果

```
7 2 6 9 4
7 2 6 4 9
7 2 4 6 9
2 7 4 6 9
2 4 7 6 9
2 4 6 7 9
---
2 4 6 7 9
```

終