C++ 学習教材

筑波大学 システム情報系 三谷純
最終更新日 2025/8/29

(C) 2025 Jun Mitani　　図表出典：三谷純著『C ++ 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

本資料は専門学校・大学・企業などで

『C++ ゼロからはじめるプログラミング』

を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。

（個人で使用される分には許諾を得る必要はありません）

出版社　：翔泳社
発売日　：2025/9/8
ISBN　：9784798191218

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関数の一歩先
  - 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を獲得できる
  - 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

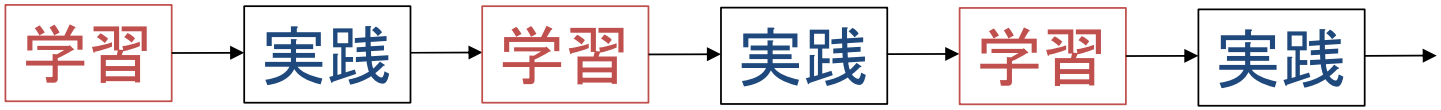
よりよく学ぶために

- 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）



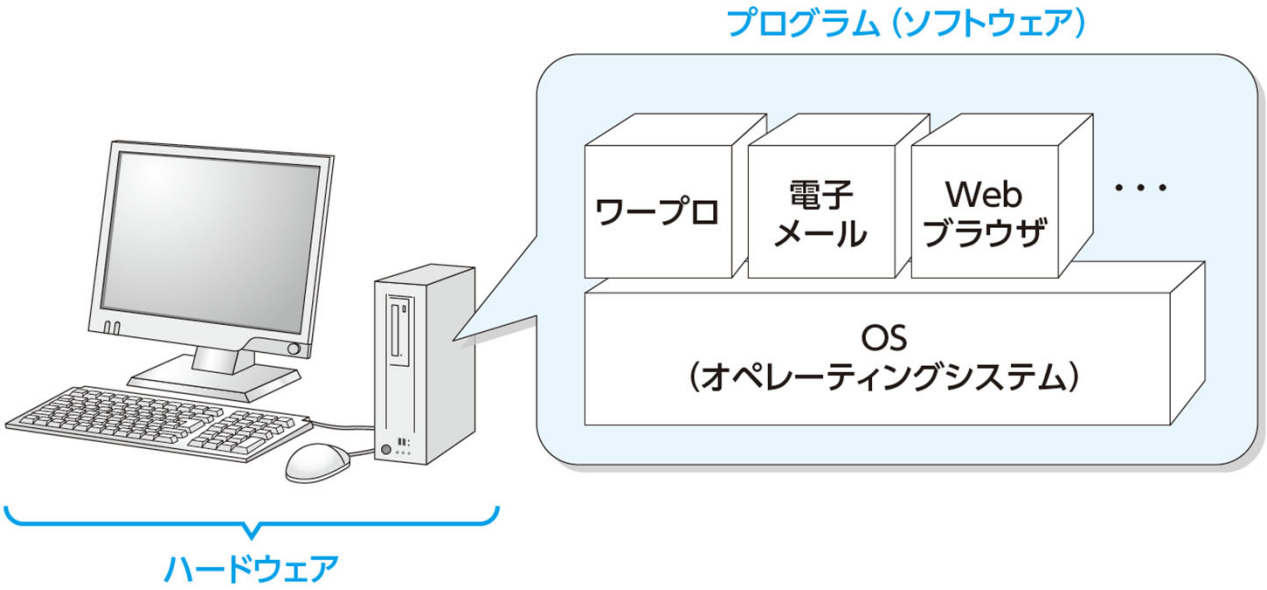
全体の流れ

1. C++言語に触れる
2. 条件分岐と繰り返し
3. 関数
4. クラスの基本
5. クラスの一歩進んだ使い方
6. 標準ライブラリ
7. アドレスとポインタ
8. 一歩進んだC＋＋プログラミング

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

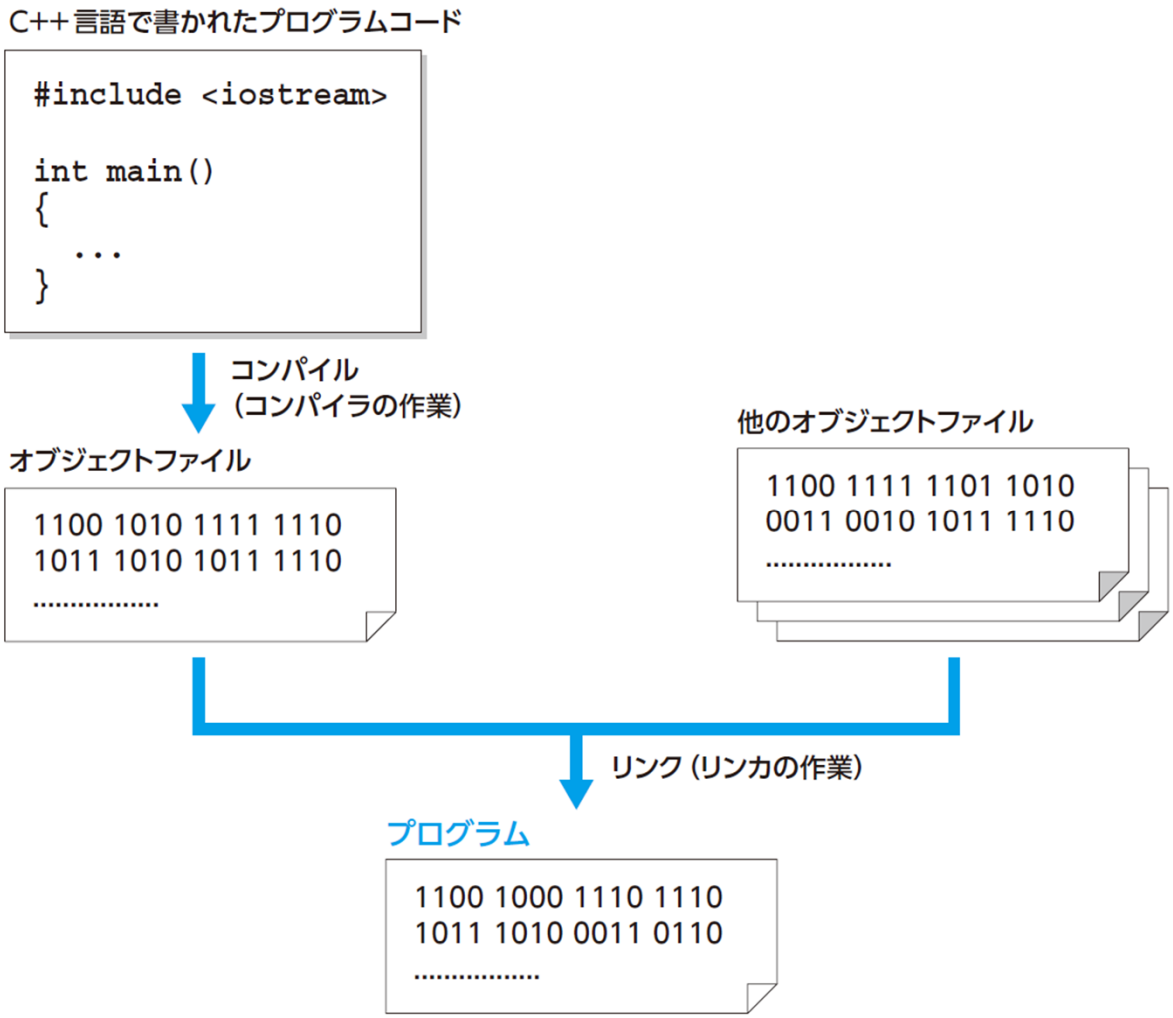


さまざまなプログラミング言語

- 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**
最も歴史のあるC言語の後継として、現在も幅広い分野で使用されている。
効率的で高速に動作するプログラムを作成できる。
オブジェクト指向の仕組みに基づいた、規模の大きなシステム開発にも適している。

プログラムが作成されるまでの流れ



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、最も基本的なプログラムコードの例

```
#include <iostream>

int main()
{
    std::cout << "こんにちは" << std::endl;
    return 0;
}
```


- 半角英数と記号で記述する
- 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
- 大文字と小文字は区別される

ブロックとインデント

```
#include  <iostream>

int main()
{
    for (int i = 0; i < 10; i++) {
        std::cout << i << std::endl;
    }

    return 0;
}
```

(b) (a)

ブロック： { と } で囲まれた範囲
ブロックの中にブロックが含まれることもある（**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 <iostream>
/*
    「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:2025年8月1日
    作成者:三谷純
*/

int main()
{
    // 画面にメッセージを出力する
    std::cout << "こんにちは" << std::endl;
    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

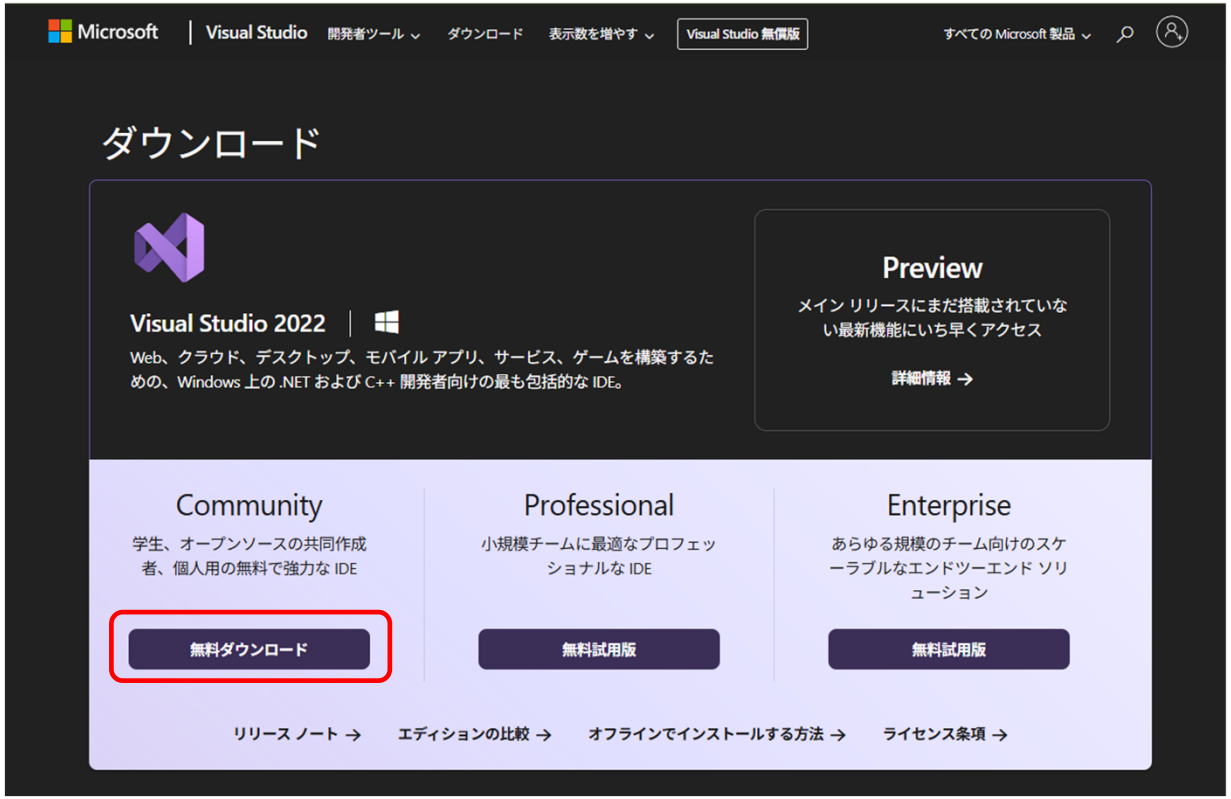
コンパイラ：コンパイルを行う（オブジェクトファイルを生成する）

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 (Community) の入手

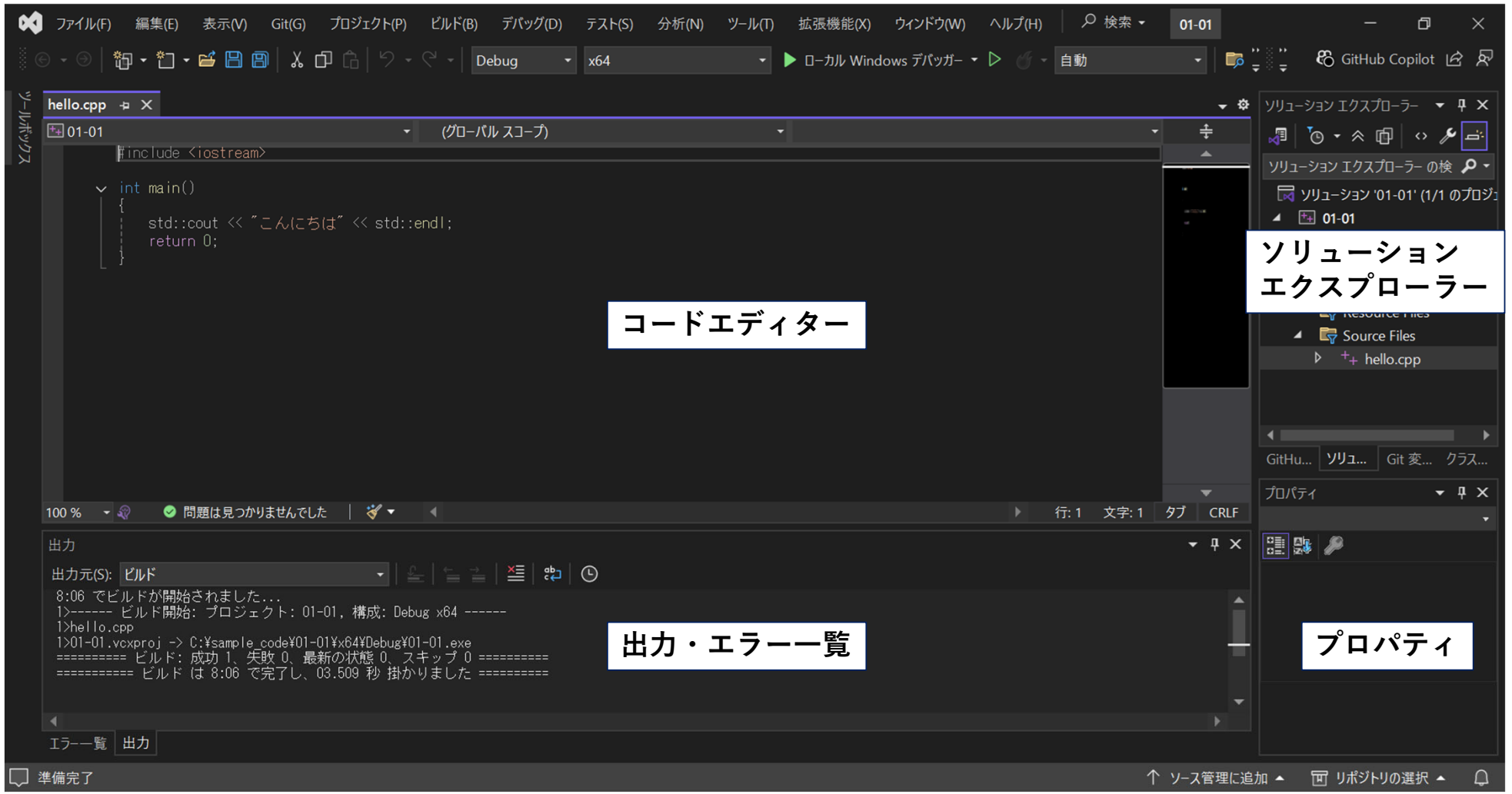
https://visualstudio.microsoft.com/ja/downloads/



※ 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

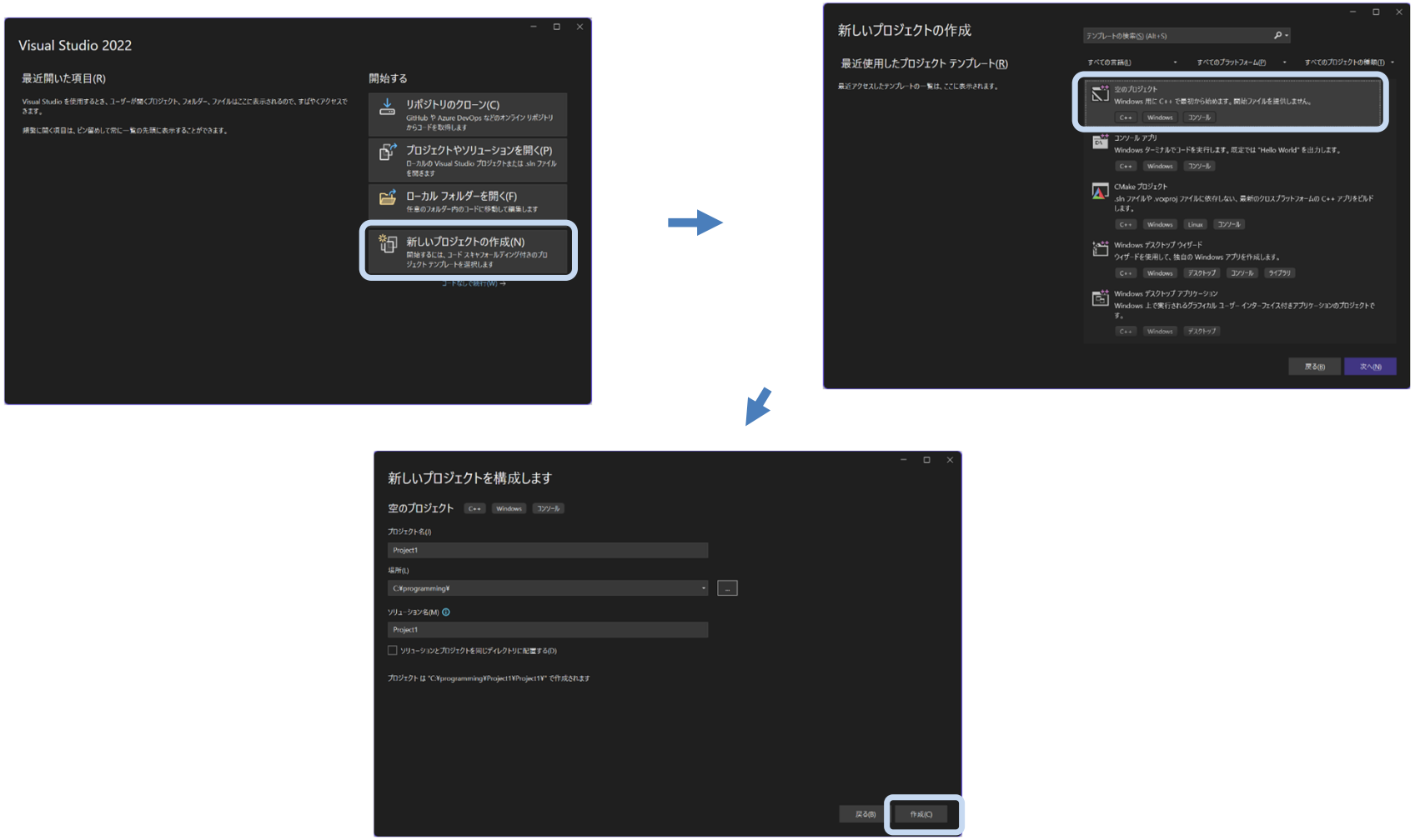


プログラムを作成して実行する

実行までの3つのステップ

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)
開始する
リポジトリのクローン(C)
プロジェクトやソリューションを開く(P)
ローカル フォルダーを開く(F)
新しいプロジェクトの作成(N)
新しいプロジェクトの作成
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
コンソール アプリ
CMake プロジェクト
Windows デスクトップ ウィザード
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
プロジェクト名(J)
Project1
場所(L)
C:¥programming¥
ソリューション名(M)
Project1
戻る(B)
作成(C)

2. プログラムコードの作成

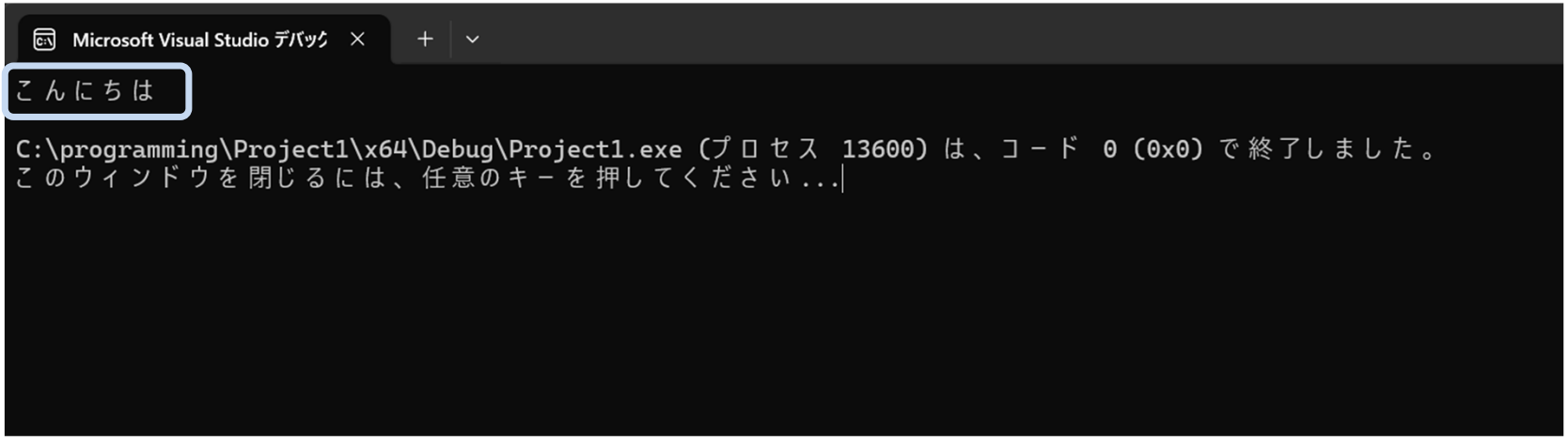
1. [プロジェクト]メニューの[新しい項目の追加]を選択
2. ファイル名を入力（拡張子は .cpp）


```
#include <iostream>

int main()
{
    std::cout << "こんにちは" << std::endl;
    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

[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

出力と変数

画面へ文字列を出力する

・画面へ文字列を出力する例

```
std::cout << "こんにちは。";
```

・文字列をダブルクォーテーション(")で囲む
・<< を続けることで、複数の文字列を連続して出力できる。

```
std::cout << "こんにちは。" << "よい天気ですね。";
```

・文字列の代わりに std::endl と書くことで改行を出力できる

```
std::cout << "こんにちは。" << std::endl;
```

画面へ文字列を出力するプログラムコード

```
#include <iostream>

int main()
{
    std::cout << "こんにちは" << std::endl;
    std::cout << "C++言語の学習を" << "がんばりましょう";
    return 0;
}
```

「こんにちは」という文字列と、それに続く改行を出力します

この行は省略してもかまいません

2つの文字列を続けて出力します

いろいろな文字列を出力してみよう

名前空間stdを使用する

プログラムコードの中に

```
using namespace std;
```

と書くと、それ以降では std:: の表記を省略できる

```
#include <iostream>
using namespace std;

int main()
{
    cout << "こんにちは" << endl;
    cout << "C++言語の学習を" << "がんばりましょう";
}
```

これ以降はstdという名前空間を使用することを意味します

std::の表記を省略できます

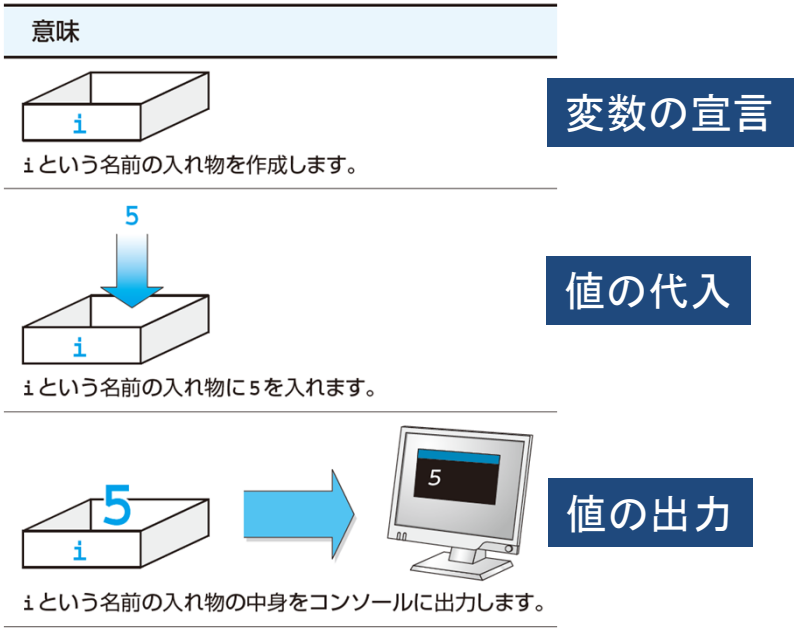
変数

変数とは値を入れておく入れ物のこと

変数の宣言：変数を作成すること

値の代入：変数に値を入れること

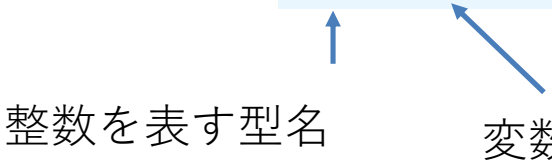
```
#include <iostream>
using namespace std;

int main()
{
    int i;
    i = 5;
    cout << i;
}
```



変数の宣言

```
型名　変数名;
```

```
int i;
```



整数を表す型名

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;
cout << i;
```

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;
```

種類	型の名前	サイズ	格納できる値の範囲
論理型	bool	1バイト	trueまたはfalse (注❶-24)
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❶-25)
	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)

変数の型

よく使用する型

- 整数　　：int型
- 小数点を含む数：double型
- 文字　　：char型
- 論理値　　：bool型

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

※ bool型は true または false

さまざまな型の変数

```
#include <iostream>
using namespace std;

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

    cout << "I の値は" << i << endl;
    cout << "d の値は" << d << endl;
    cout << "c の値は" << c << endl;
    cout << "b の値は" << b;
}
```

※ 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0.2
cの値はA
bの値は1
```

※ bool型のtrueは整数の1として出力される

実際に試してみよう

算術演算と型

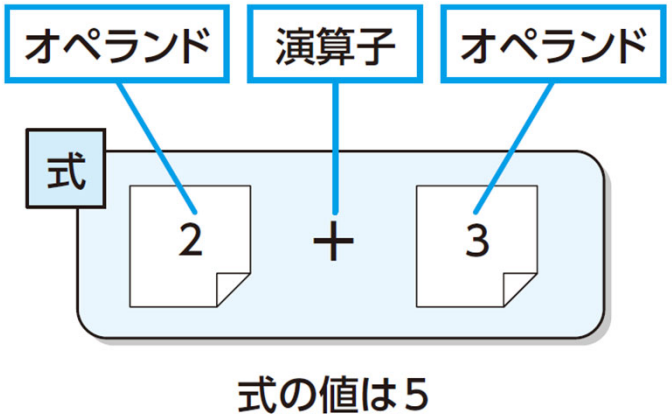
計算を行う

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    i = 2 + 3;
    cout << i;
}
```

実行結果

```
5
```

用語



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;
cout << "jの値は" << 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 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

異なる型の値の代入

```
int i = 1.99;
cout << i;
```

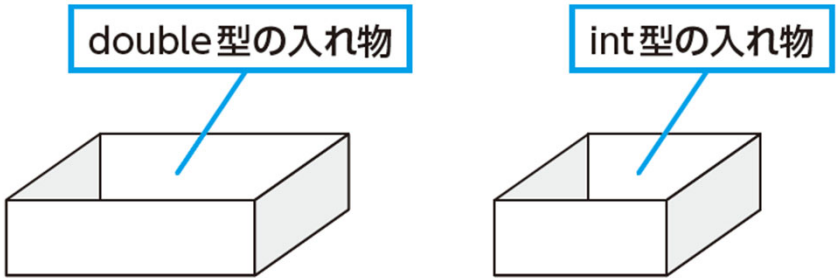
実行結果

```
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;
cout << i + d << endl;
```

↓ 実行結果

```
5.5
```

※ 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

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;
cout << "cの値は" << c;
```

実行結果

cの値は 0

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;
cout << "cの値は" << c;
```

実行結果

cの値は 0.5

実際に試してみよう

string型による文字列の扱い

文字列はstring型の変数に代入できる

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string message = "こんにちは";
    cout << message;
}
```

string型の変数を使用できるようにします

string型の変数messageを初期化しています

変数messageの値を出力します

+演算子を使って文字列を連結できる

```
string message1 = "こんにちは。";
string message2 = "今日はよい天気ですね。";
string message3 = message1 + message2;
cout << message3;
```

実行結果

こんにちは。今日はよい天気ですね。

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 <iostream>
using namespace std;

int main()
{
    int i;
    cout << "整数を入力してください" << endl;
    cin >> i;
    cout << i << "が入力されました";
}
```

← キーボードから入力された整数を変数iで受け取ります

↓ 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です

※ 小数を含む値はdouble型、
文字列はstring型の変数で受け取れる

実際に試してみよう

第2章　条件分岐と繰り返し

条件分岐

条件分岐

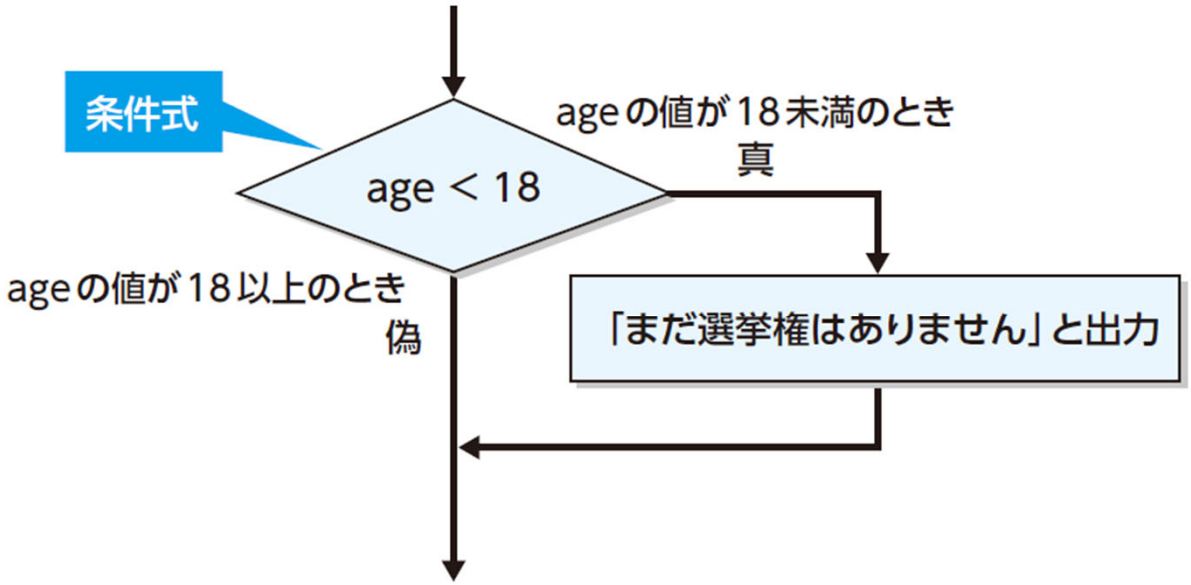
条件による処理の分岐
「もしも○○ならば××を実行する」

```
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) {
    cout << "まだ選挙権はありません";
}
```



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;
}
```

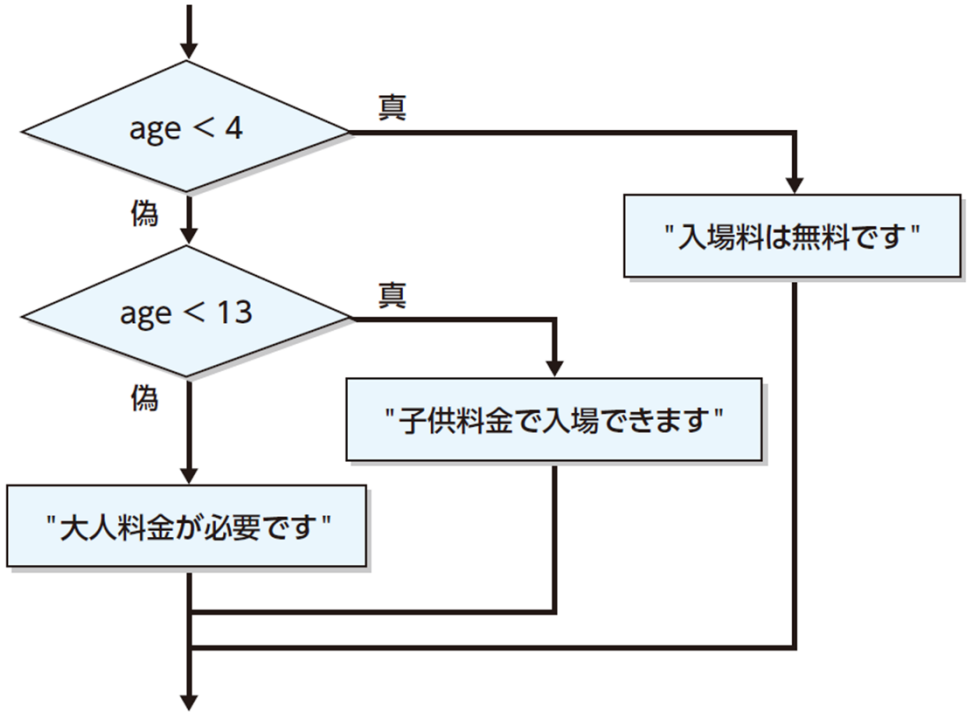
例

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

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



実際に試してみよう

ワン・モア・ステップ：条件式に変数を使用する

- if 文の条件式に変数を使うことができる

```
bool b = ture;
if (b) {
  命令文
}
```

変数 bの値が true のときに命令文が実行される
bの値を false にすると実行されない

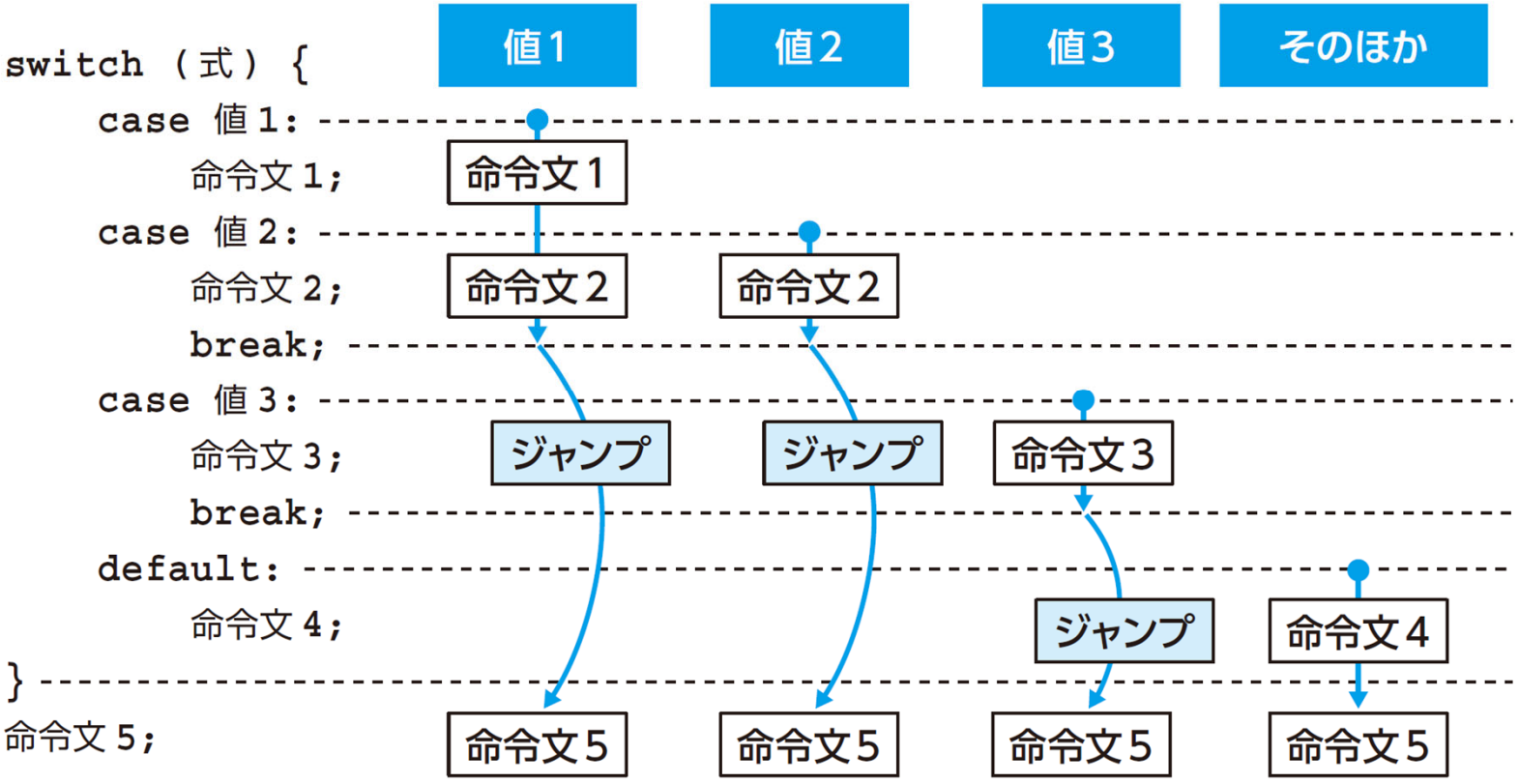
- 変数の型が整数型の場合、値が0でないときに命令文が実行される

```
int i = 1;
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
iの値を0にすると実行されない

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

実際に試してみよう

switch文の例(2)

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

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** age が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

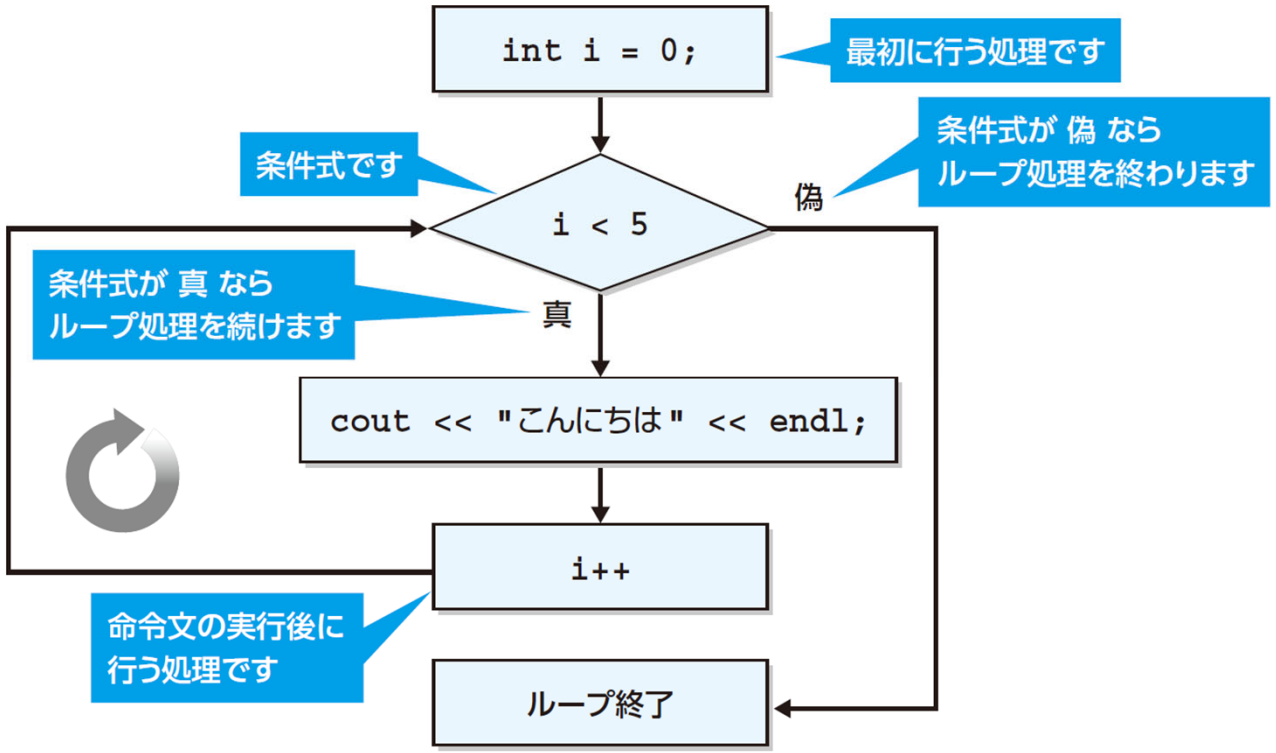
for文の構文

```
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++) {
    cout "こんにちは" << endl;
}
```



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;
    cout << i << "を加えました" << endl;
}
cout << "合計は" << sum;
```

↓実行結果

```
1を加えました
2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()
{
    int sum = 0;
    for (int i = 1; i <= 100; i++) {
        sum += i;
        cout << i << "を加えました" << endl;
    }
    cout << "合計は" << 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) {
    cout << "こんにちは" << endl;
    i++; // この命令文が無いと「無限ループ」
}
```

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

※ 無限ループにならないように注意が必要
（ [Ctrl]+c キーで強制終了できる）

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 {
    cout << "こんにちは" << endl;
    i++; // この命令文が無いと「無限ループ」
} while (i < 5);
```

```
int i = 5;
do {
    cout << i << endl;
    i--; // この命令文が無いと「無限ループ」
} while (i > 0);
```

※ 無限ループにならないように注意が必要

実際に試してみよう

ループの処理を中断する break

`break;` でループ処理を中断できる

```
int sum = 0;
for (int i = 1; i <= 10; i++) {
    sum += i;
    cout << "変数sumに" << i << "を加えました。";
    cout "sumは" << sum << endl;
    if (sum > 20) {
        cout << "合計が20を超えました。";
        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;
    cout << "変数sumに" << i << "を加えました。";
    cout << "sumは" << sum << endl;
}
```

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++) {
    cout << "a = " << a << endl;//★
    for (int b = 1; b <= 3; b++) {
        cout << "   b = " << b << endl;//☆
    }
}
```

★の命令文は3回実行される

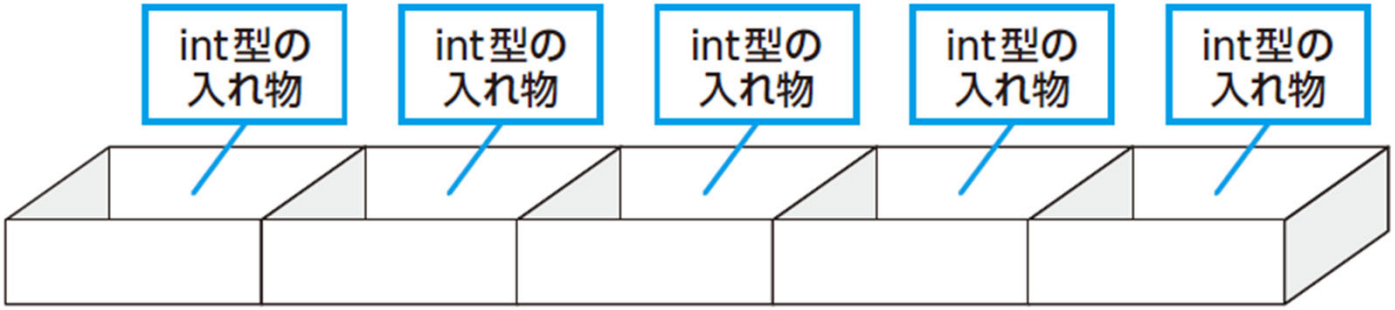
☆の命令文は9回実行される

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. インデックスを指定して要素の値にアクセスする
   例：`cout << 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++) {
    cout << 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};
```

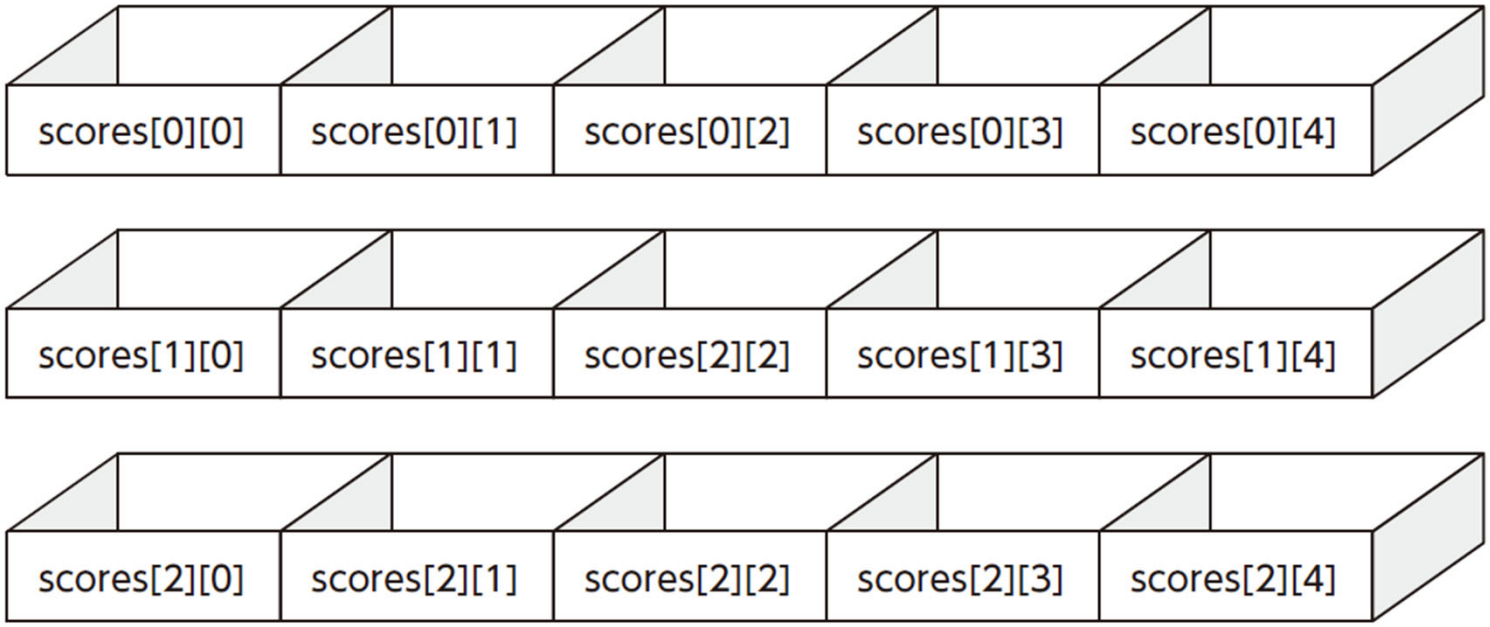
要素の数の記述を省略できる

```
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;
```



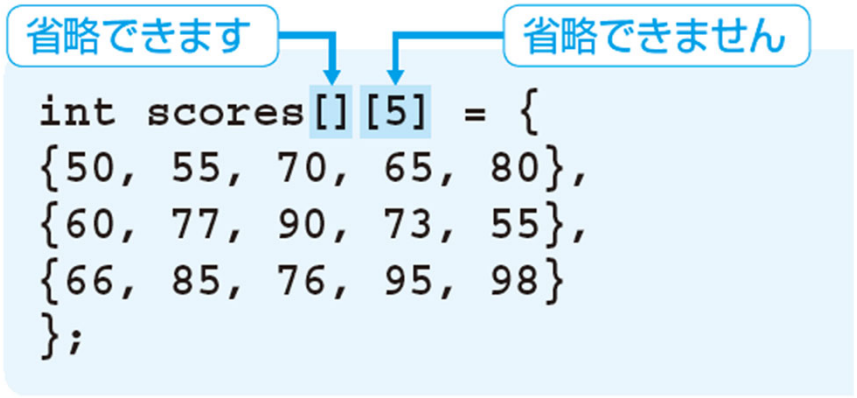
※ 横に並んだ入れ物の列が、さらに縦にも並んでいるイメージ

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



第3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()
{
    命令文
}
```

※ 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()
{
    cout << "カウントダウンをします" << endl;
    for (int i = 5; i >= 0; i--) {
        cout << 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 <iostream>
using namespace std;

void countdown()
{
    cout << "カウントダウンをします" << endl;
    for (int i = 5; i >= 0; i--) {
        cout << i << " ";
    }
   cout << endl;
}

int main()
{
    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

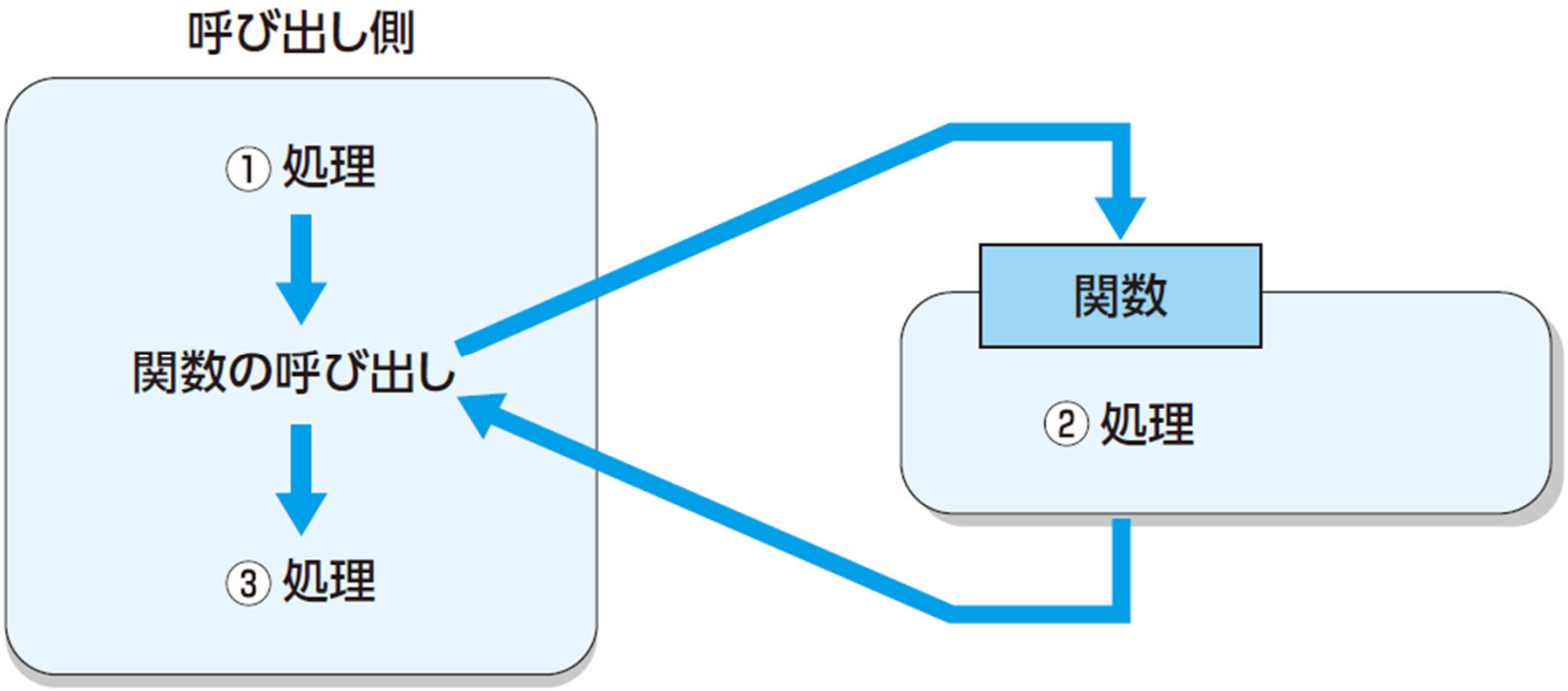
実際に試してみよう

main関数

```
int main()
```

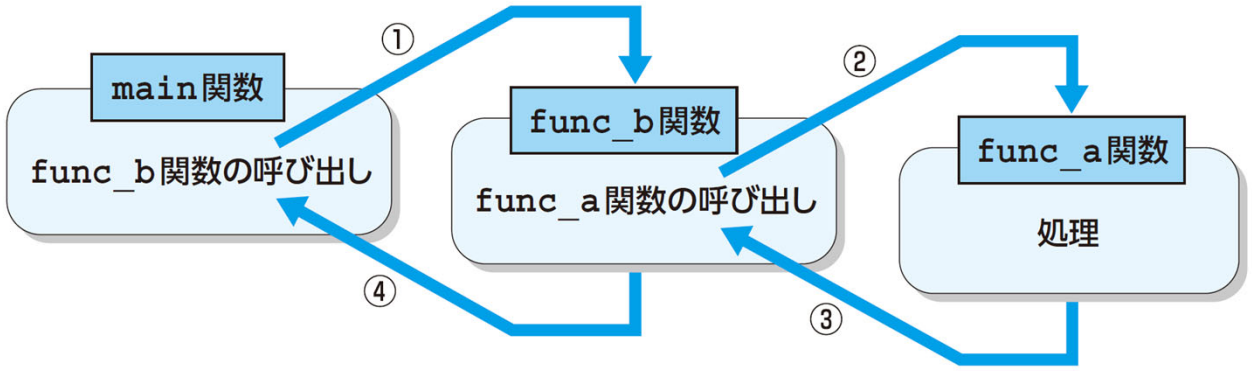
- C++言語では、プログラムが実行されるときに、この main関数が呼び出される
- main関数は、プログラムの開始位置となる特別な関数

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 <iostream>
using namespace std;

void func_a()
{
    cout << "func_a" << endl;
}

void func_b()
{
    cout << "func_b開始" << endl;
    func_a();
    cout << "func_b終了" << endl;
}

int main()
{
    cout << "main開始" << endl;
    func_b();
    cout << "main終了" << endl;
}
```



実際に試してみよう

関数の引数

引数とは

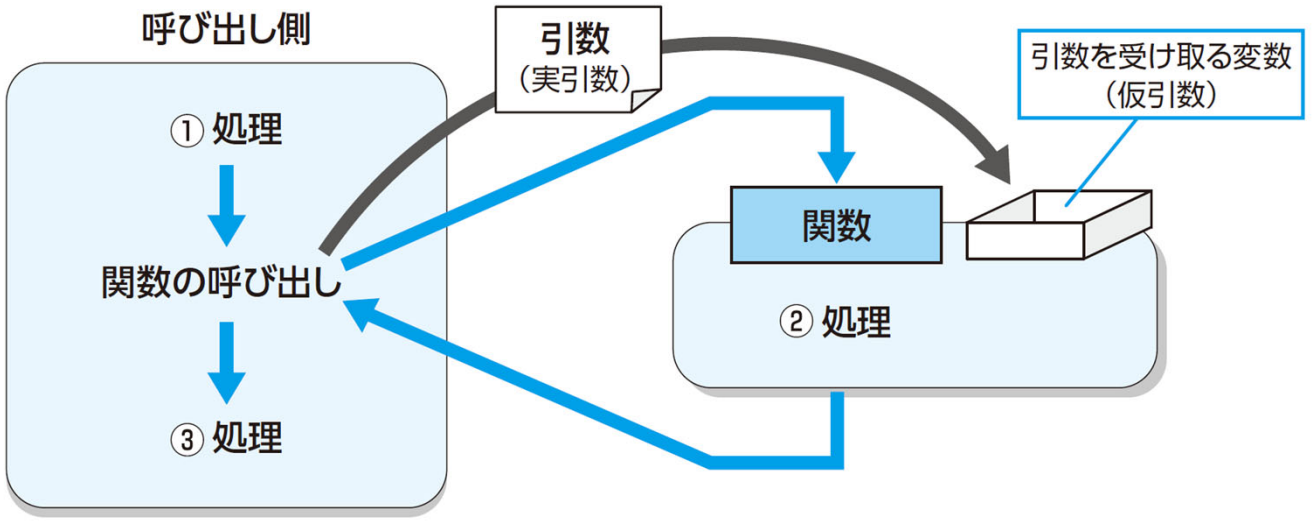
引数（ひきすう）

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
}
```



引数の受け渡しには、関数名の後ろのカッコ() を使用する。

引数のある関数の例

```
#include <iostream>
using namespace std;

void countdown(int start)
{
    cout << "関数が受け取った値：" << start << endl;
    cout << "カウントダウンをします" << endl;
    for (int i = start; i >= 0; i--) {
        cout << i << " ";
    }
    cout << endl;
}

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

startという名前のint型の変数で値を受け取る

異なる値を引数として
countdown関数を呼び出す

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 <iostream>
using namespace std;

void countdown(int start, int end)
{
    cout << "カウントダウンをします" << endl;
    for (int i = start; i >= end; i--) {
        cout << i << " ";
    }
}

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

2つの値をcountdown関数に渡す

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

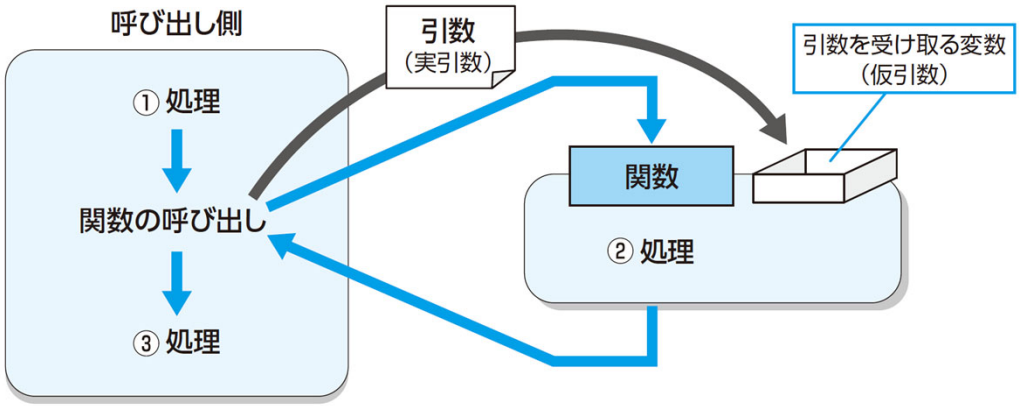
```
countdown(7, 3);
```

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

```
#include <iostream>
using namespace std;

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

int main()
{
    int i = 10;
    cout << "(1) iの値は" << i << endl;
    func(i);
    cout << "(2) iの値は" << i;
}
```

↓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

実際に試してみよう

関数の戻り値

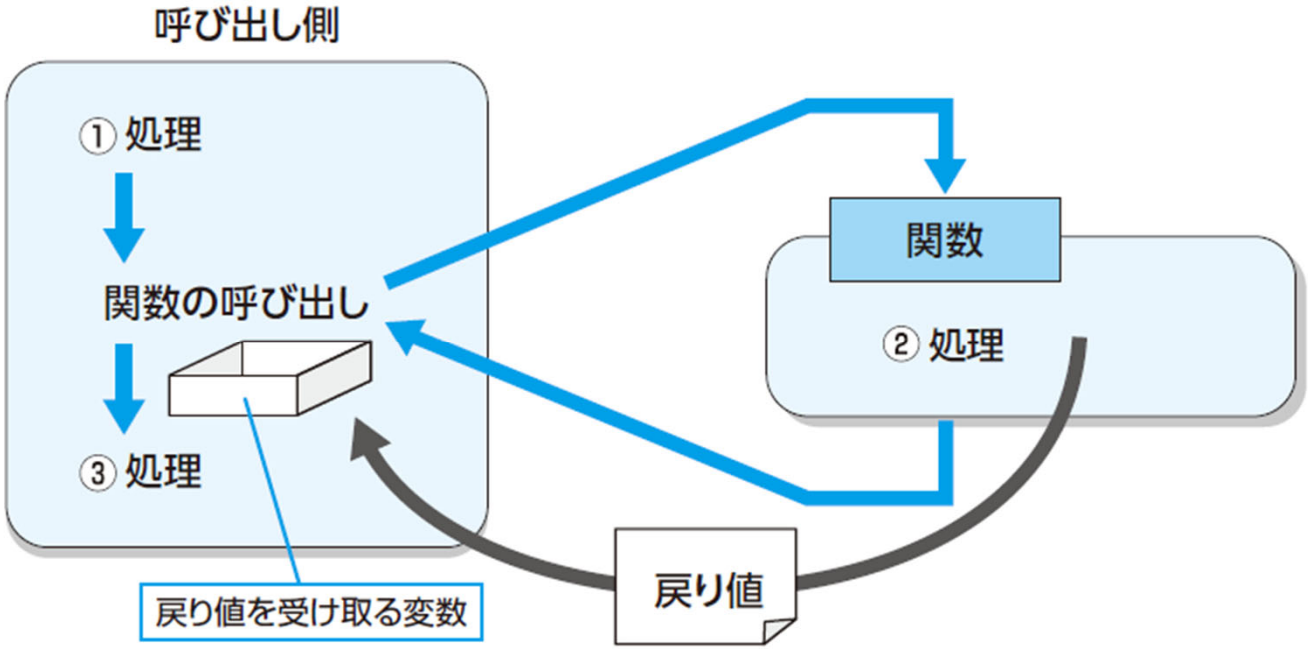
戻り値とは

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(引数リスト) {
    命令文
    return 戻り値;
}
```



戻り値のある関数

- `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 <iostream>
using namespace std;

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

int main()
{
    double d = circle_area(2.5);
    cout << "半径2.5の円の面積は " << d;
}
```

実際に試してみよう

真偽値を戻り値とする関数

```
#include <iostream>
using namespace std;

bool is_positive_number(double d)
{
    if (d > 0) {
        return true;
    }
    else {
        return false;
    }
}

int main()
{
    double d = -1.5;
    if (is_positive_number(d)) {
        cout << "dの値は正です";
    }
    else {
        cout << "dの値は正ではありません";
    }
}
```

引数の値が「正」ならばtrueを返します

引数の値が「正」でないならばfalseを返します

戻り値の値によって条件分岐を行います

- 戻り値の型がbool型の関数は、true または false の値を戻す
- if文の条件式の代わりに関数の呼び出しを書くことができる

[ワン・モア・ステップ]

is_positive_number 関数の中身を、次のように1行で書くこともできる

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

関数のまとめ

引数なし、戻り値なし

```
void 関数名() {
    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;
}
```

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（引数と戻り値）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書く

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、main関数の前に書いて、関数の定義はmain関数の後ろに書く

関数のプロトタイプ宣言

```
戻り値の型　関数名(引数リスト);
```

関数のプロトタイプ宣言の例

```
#include <iostream>
using namespace std;

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

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

void func1(int a)
{
    cout << "func1が呼び出されました" << endl;
}

void func2()
{
    cout << "func2が呼び出されました" << endl;
}
```

関数のプロトタイプ宣言

関数のオーバーロード

同じ名前を持つ関数

- 名前が同じで、引数が異なる関数を複数定義できる
- 同じ名前の関数を定義することを「関数のオーバーロード」とよぶ
- 右のプログラムコードでは、funcという名前の関数が4つ定義されている

※ 関数名と引数列の組み合わせをシグネチャと呼ぶ。シグネチャが同じ関数を複数定義することはできない

```
#include <iostream>
#include <string>
using namespace std;

void func()
{
    cout << "引数はありません" << endl;
}

void func(int i)
{
     cout << "int型の値" << i << "を受け取りました" << endl;
}

void func(double d)
{
     cout << "double型の値" << d << "を受け取りました" << endl;
}

void func(string s)
{
    cout << "文字列" << s << "を受け取りました" << endl;
}

int main() {
    func();
    func(1);
    func(0.1);
    func("Hello");
}
```

第4章　クラスの基本

クラスとオブジェクト

クラスとオブジェクト

クラス

オブジェクトがどのような情報と機能を持つかを定義したもの

オブジェクト

クラスによって定義された情報と機能を持つ1つ1つの実体

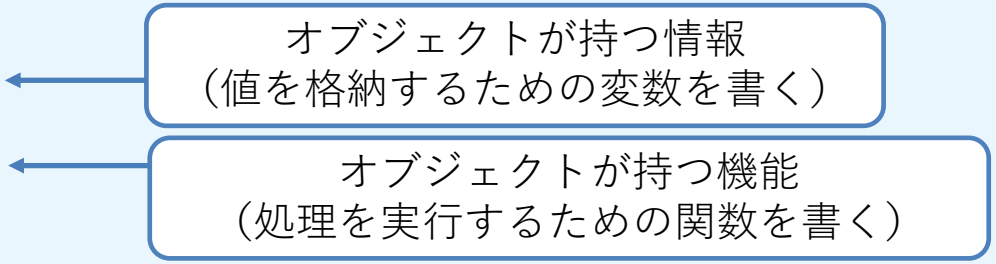
例

ソフトウェアで管理するもの	クラス	オブジェクト
学生情報	学籍番号、氏名など、どのような項目を管理するか定めたもの	学生Aの情報、学生Bの情報、学生Cの情報、・・・
サッカーゲームのサッカー選手情報	体力、走力、ポジションといった、どのようなステータスを管理するか定めたもの	選手A、選手B、選手C、・・・

こういった、クラスとオブジェクトという概念に基づいてプログラムを作ることを「オブジェクト指向」とよぶ

クラスの定義

```
class クラス名 {
アクセス指定子:
    メンバ変数の定義
    メンバ関数の定義
};
```



- 末尾にセミコロン(;)をつける
- メンバ変数とメンバ関数をあわせて「メンバ」と呼ぶ
- アクセス指定子はメンバへのアクセスを制御するもの（詳しくは後で学ぶ。ここでは、 public と書くものとする）。

メンバ変数を持つクラスの定義

学籍番号(id)と氏名(name)を持つ学生証を扱うための
StudentCardクラスの定義

```
class StudentCard {
public:
  int id = 0;
  string name = "未定";
};
```

メンバ変数

- idとnameという2つの値をセットにして扱える

オブジェクトの生成とメンバ変数の変更

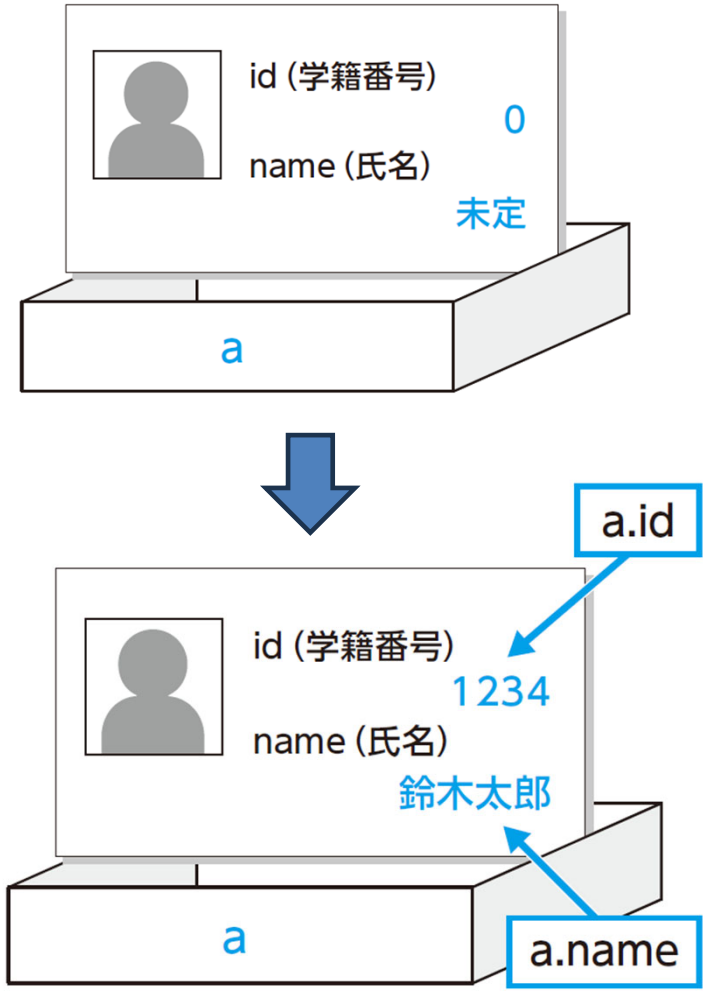
オブジェクトの生成

```
StudentCard a;
```

オブジェクトaのメンバ変数の変更

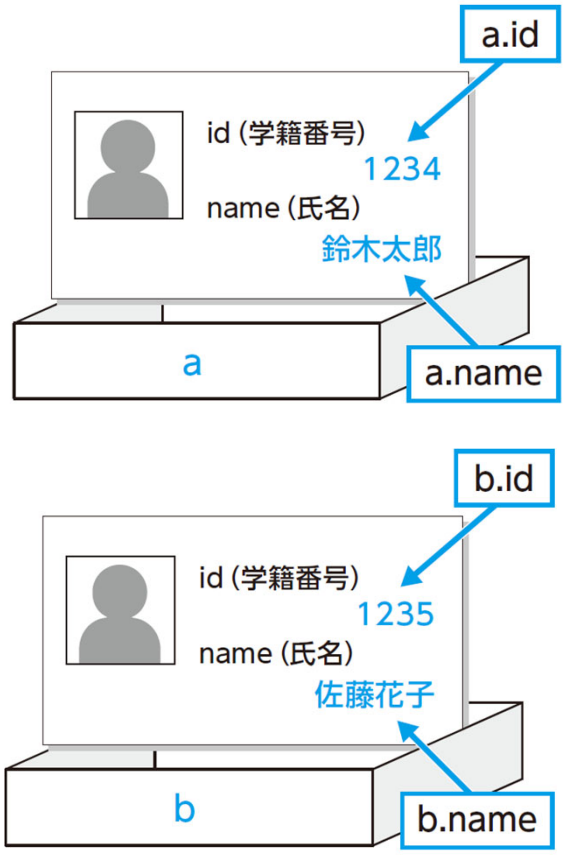
```
a.id = 1234;
a.name = "鈴木太郎";
```

※（オブジェクトを代入した変数の名前）.（メンバ変数名）でオブジェクトのメンバ変数にアクセスできる



複数のオブジェクトを生成する例

```
class StudentCard {
public:
    int id; // 学籍番号
    string name; // 氏名
}

int main()
{
    StudentCard a;
    a.id = 1234;
    a.name = "鈴木太郎";

    StudentCard b;
    b.id = 1235;
    b.name = "佐藤花子";

    cout << "aのidは" << a.id << endl;
    cout << "aのnameは" << a.name << endl;
    cout << "bのidは" << b.id << endl;
    cout << "bのnameは" << a.name;
}
```



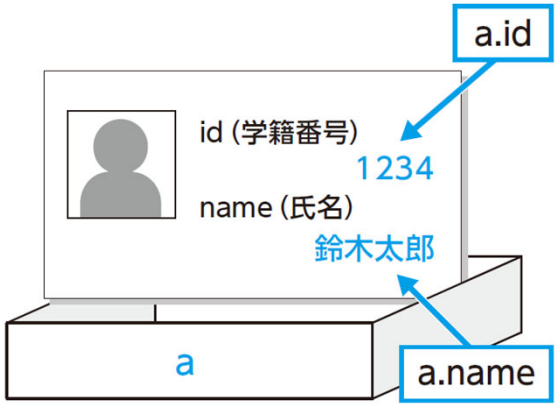
実際に試してみよう

参照

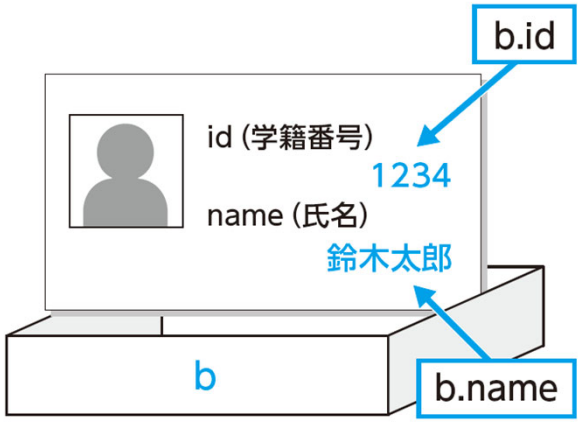
オブジェクトの複製

```
StudentCard a;
a.id = 1234;
a.name = "鈴木太郎";

StudentCard b = a;
```

オブジェクトが生成される



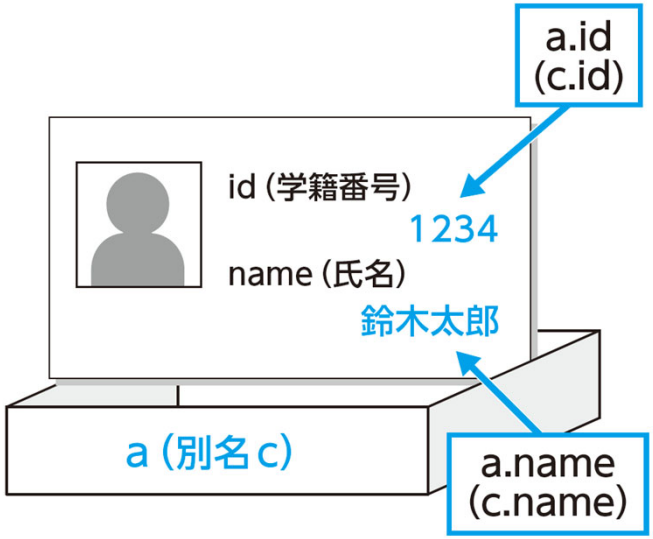
オブジェクトの複製が生成される



※ 全部で2つのオブジェクトが生成される

参照型

```
StudentCard a;
a.id = 1234;
a.name = "鈴木太郎";

StudentCard& c = a;
```



- オブジェクトに別名cを与えている
- a.id = 1001; と書くことと c.id = 1001; と書くことは同じ意味を持つ
- 「cはaを参照する」「cはaの参照である」という
- cの型を**参照型**という
- 全体でオブジェクトは1つしか存在しない

関数への参照渡し

参照を引数とする関数

```
Void print_info(StudentCard& card)
{
    cout << "学籍番号：" << card.id << endl;
    cout << "氏名：" << card.name << endl;
}
```

仮引数cardは参照型

※ 関数の中では、cardという別名を使用する

関数の呼び出し

```
StudentCard a;
a.id = 1234;
a.name = "鈴木太郎";

print_info(a);
```

関数にStudentCardオブジェクトを渡す

実際に試してみよう

メンバ関数

メンバ関数とは

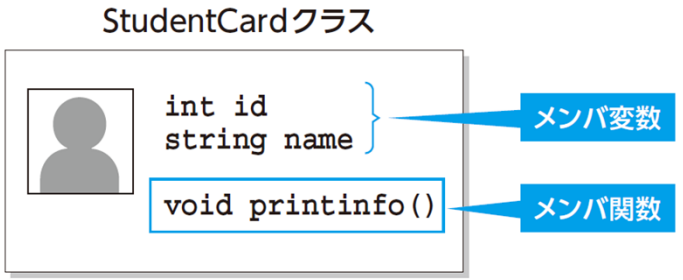
- オブジェクトに機能を持たせることができる
- これを、クラスに関数（メンバ関数）を持たせることで実現する

```
class クラス名 {
アクセス指定子:
    戻り値の型 関数名(引数リスト) {
        命令文
        return 戻り値;
    }
};
```

メンバ関数の定義です

メンバ関数の具体例

```
class StudentCard {
public:
    int id = 0; // 学籍番号
    string name = "未定"; // 氏名

    void printInfo() {
        cout << "学籍番号：" << this->id << endl;
        cout << "氏名：" << this->name << endl;
    }
};
```



※ メンバ関数の中でメンバ変数にアクセスするには、this->の後に変数名を続ける

```
StudentCard a;
a.id = 1234;
a.name = "鈴木太郎";

a.printInfo();
```

※（オブジェクトの変数名）.（メンバ関数名）で、メンバ関数を呼び出す

実際に試してみよう

メンバ関数の宣言と定義を分けて書く

メンバ関数の宣言だけをクラスの定義の中に書いて、メンバ関数の定義をクラスの外に書くことができる

```
class StudentCard {
public:
    int id = 0;              // 学籍番号
    string name = "未定";    // 氏名

    void printInfo();
};

void StudentCard::printInfo()
{
    cout << "学籍番号：" << this->id << endl;
    cout << "氏名：" << this->name << endl;
}
```

メンバ関数の宣言です

StudentCardクラスのメンバ関数printInfoの定義です

※ クラスの定義をシンプルにして、全体の見通しを良くできる

thisキーワードの省略

「this->変数名」の表記で、キーワードがなくてもメンバ変数であることが明らかな場合は、this->の部分を省略できる

```
void printInfo() {
    cout << "学籍番号：" << this->id << endl;
    cout << "氏名：" << this->name << endl;
}
```

```
void printInfo() {
    cout << "学籍番号：" << id << endl;
    cout << "氏名：" << name << endl;
}
```

this-> を省略しました

this-> を省略しました

実際に試してみよう

コンストラクタとデストラクタ

コンストラクタとは

コンストラクタ：オブジェクトが生成されるときに自動的に実行される特別なメンバ関数

```
StudentCard a(1234, "鈴木太郎");
```

※オブジェクトを生成するときに引数を渡せる

コンストラクタの定義

```
クラス名（引数リスト） {
    命令文
}
```

例

```
StudentCard（int i, string s）
{
    id = i;
    name = s;
}
```

引数を使ってメンバ変数を初期化する

コンストラクタの定義（メンバ初期化リストがある場合）

```
クラス名（引数リスト）: メンバ変数1(初期値1), メンバ変数2(初期値2) {
    命令文
}
```

例

```
StudentCard（int i, string s）: id(i), name(s)
{
}
```

メンバ初期化リストでメンバ変数を初期化する場合は、コンストラクタの中身は空でよい

コンストラクタの使用例

```
#include <iostream>
#include <string>
using namespace std;

class StudentCard {
public:
    int id; // 学籍番号
    string name; // 氏名

    StudentCard(int i, string s) : id(i), name(s) {
        cout << "StudentCardクラスのコンストラクタでの処理" << endl;
    }

    void printInfo() {
        cout << "学籍番号：" << id << endl;
        cout << "氏名：" << name << endl;
    }
};

int main()
{
    StudentCard card(1234, "鈴木太郎");
    card.printInfo();
}
```

メンバ初期化リストを持つ
コンストラクタ

実行結果

```
StudentCardクラスのコンストラクタでの処理
学籍番号:1234
氏名:鈴木太郎
```

コンストラクタ内の処理が実行されています

コンストラクタに渡した引数でメンバ変数が初期化されています

オブジェクトの生成
※ コンストラクタが自動で呼び出される

実際に試してみよう

コンストラクタのオーバーロード

- 引数が異なる複数のコンストラクタを定義できる
  （これをコンストラクタのオーバーロードとよぶ）
- オブジェクトを生成するときに、引数に応じて対応するコンストラクタが自動で呼び出される
- 引数がないコンストラクタを**デフォルトコンストラクタ**とよぶ

```
StudentCard() : id(0), name("未定") {
    cout << "引数のないコンストラクタ（デフォルトコンストラクタ）が実行されました" << endl;
}

StudentCard(int i) : id(i), name("未定") {
    cout << "引数が1つのコンストラクタが実行されました" << endl;
}

StudentCard(int i, string s) : id(i), name(s) {
    cout << "引数が2つのコンストラクタが実行されました" << endl;
}
```

デストラクタとは

- オブジェクトには「寿命」という概念がある
- オブジェクトは、スコープから外れるときに、自動的に破棄される
- オブジェクトが破棄されるときに自動的に実行される特別なメンバ関数を**デストラクタ**と呼ぶ

デストラクタの定義

```
~クラス名 () {
        デストラクタでの処理
}
```

```
class MyClass {
public:
    ~MyClass() {
        cout << "デストラクタが呼ばれました" << endl;
    }
};

int main()
{
    MyClass a;

    for (int i = 2; i < 5; i++) {
        MyClass b;
    }
}
```

オブジェクトaはmain関数が終わるときに破棄される

オブジェクトbはforループで次の処理にうつるときに破棄される

実際に試してみよう

アクセス制御とconst修飾子

アクセス指定子

- オブジェクト指向の考え方でプログラムコードを作成するときには、外部からのクラス内部へのアクセスを制御することが大切。
- アクセス指定子を使って、メンバへのアクセスを制御できる。

アクセス指定子	説明
public	クラスの外部からもアクセスできる
protected	自身のクラス内部および派生クラス(*)からアクセスできる。クラスの外部からはアクセスできない
private	自身のクラス内部からのみアクセスできる

*派生クラス：第5章で説明

アクセス指定子の影響が及ぶ範囲

```
class クラス名 {
        メンバ変数a
        メンバ変数b
public:
        メンバ変数c
        メンバ変数d
protected:
        メンバ変数e
        メンバ変数f
private:
        メンバ変数g
};
```

これより前にアクセス指定子がないのでprivateとして扱われます

publicとして扱われます

protectedとして扱われます

privateとして扱われます

アクセス指定子を使用する例

```
#include <iostream>
using namespace std;

class Car {
private:
    int speed; // 車の速度 (km/h)
public:
    Car() : speed(0) {}

    void speedUp() {// 速度を1増やす（最大80まで）
        if (speed < 80) { speed++; }
    }

    void speedDown() { // 速度を1減らす（0未満にはならない）
        if (speed > 0) { speed--; }
    }

    int getSpeed() const { return speed; }
};

int main() {
    Car c;
    cout << c.getSpeed() << endl;

    for (int i = 0; i < 100; i++) { c.speedUp(); }
    cout << c.getSpeed() << endl;

    for (int i = 0; i < 20; i++) { c.speedDown(); }
    cout << c.getSpeed();
}
```

- メンバ変数 speed は、外部からアクセスできない（privateアクセス指定子の範囲）。
- main関数で c.speed = 10; のように書くことは許されない。
- speedの値を変更するには、 CarオブジェクトのspeedUpまたはspeedDownメンバ関数を呼び出すことで実現する。

実行結果

```
0
80
60
```

実際に試してみよう

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 x = 10;
x++;
```

変数xを変更できなくなる

コンパイルエラー

※ 定数を定義するために使用する

```
int memberFunc() const {
    this->speed++;
}
```

自身のメンバ変数を変更できなくなる

コンパイルエラー

```
void func(const StudentCard& card)
{
    card->name = "山本俊介";
}
```

引数で受け取ったオブジェクトのメンバ変数を変更できなくなる

コンパイルエラー

※ 「メンバ変数の値が変更されない」ということが保証される

第5章クラスの一歩進んだ使い方

静的メンバ変数と静的メンバ関数

静的メンバ変数

- 静的メンバ変数とは、宣言のときに`static`キーワードをつけたもので、オブジェクトを生成しなくても使用できる（オブジェクト間で共有する値を持たせる目的で使用できる）
- 静的メンバ変数の定義はクラスの定義の外に書く。
- 静的メンバ変数にアクセスするには 「クラス名`::`変数名」 のように書く。

静的メンバ変数の宣言

```
static 変数の型 変数名;
```

静的メンバ変数を使用する例

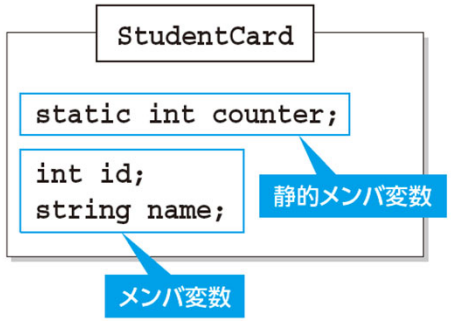
```
#include <iostream>
#include <string>
using namespace std;

class StudentCard {
public:
    static int counter;  // 学生証発行枚数
    int id; // 学籍番号
    string name; // 氏名

    StudentCard(int i, string s) : id(i), name(s) {
        cout << "コンストラクタが呼び出されました" << endl;
        StudentCard::counter++;
    }
};

int StudentCard::counter = 0;

int main()
{
    cout << "counterの値:" << StudentCard::counter << endl;
    StudentCard a(1234, "鈴木太郎");
    cout << "counterの値:" << StudentCard::counter << endl;
    StudentCard b(1235, "佐藤花子");
    cout << "counterの値:" << StudentCard::counter;
}
```

静的メンバ変数の宣言

静的メンバ変数の定義と初期化

静的メンバ変数へのアクセス

StudentCardクラスの構造

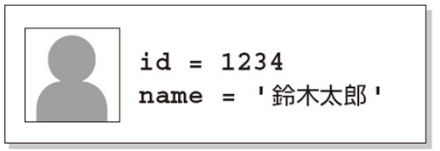


プログラム実行時の様子

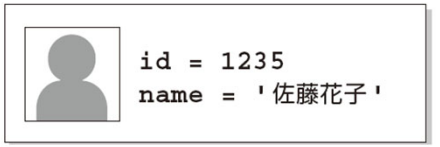
静的メンバ変数

```
StudentCard::counter = 2
```

オブジェクト



オブジェクト



実行結果

```
counterの値:0
コンストラクタが呼び出されました
counterの値:1
コンストラクタが呼び出されました
counterの値:2
```

静的メンバ関数

- 静的メンバ関数とは、宣言のときに`static`キーワードをつけたもので、オブジェクトを生成しなくても使用できる
- 静的メンバ関数の定義はクラスの定義の中でも外でもよい。
- 静的メンバ関数を呼び出すには 「クラス名`::`関数名()」 のように書く。

静的メンバ関数の宣言

```
static 戻り値の型 関数名(引数リスト);
```

静的メンバ関数を使用する例

```
#include <iostream>
using namespace std;

class AreaCalculator {
public:
    static double getTriangleArea(double base, double height) {
        return base * height / 2.0;
    }

    static double getCircleArea(double radius);
};

double AreaCalculator::getCircleArea(double radius)
{
    return radius * radius * 3.14159;
}

int main()
{
    cout << "底辺が10、高さが5の三角形の面積は"
        << AreaCalculator::getTriangleArea(10, 5) << endl;
    cout << "半径5の円の面積は"
        << AreaCalculator::getCircleArea(5) << endl;
}
```

静的メンバ関数の宣言と定義

静的メンバ関数の宣言

静的メンバ関数の定義

静的メンバ関数の呼び出し

- 静的メンバ関数は、オブジェクトを生成しなくても使用できる
- 静的メンバ関数の中からメンバ変数にアクセスできない
- 静的メンバ関数の中からメンバ関数を呼び出せない

実行結果

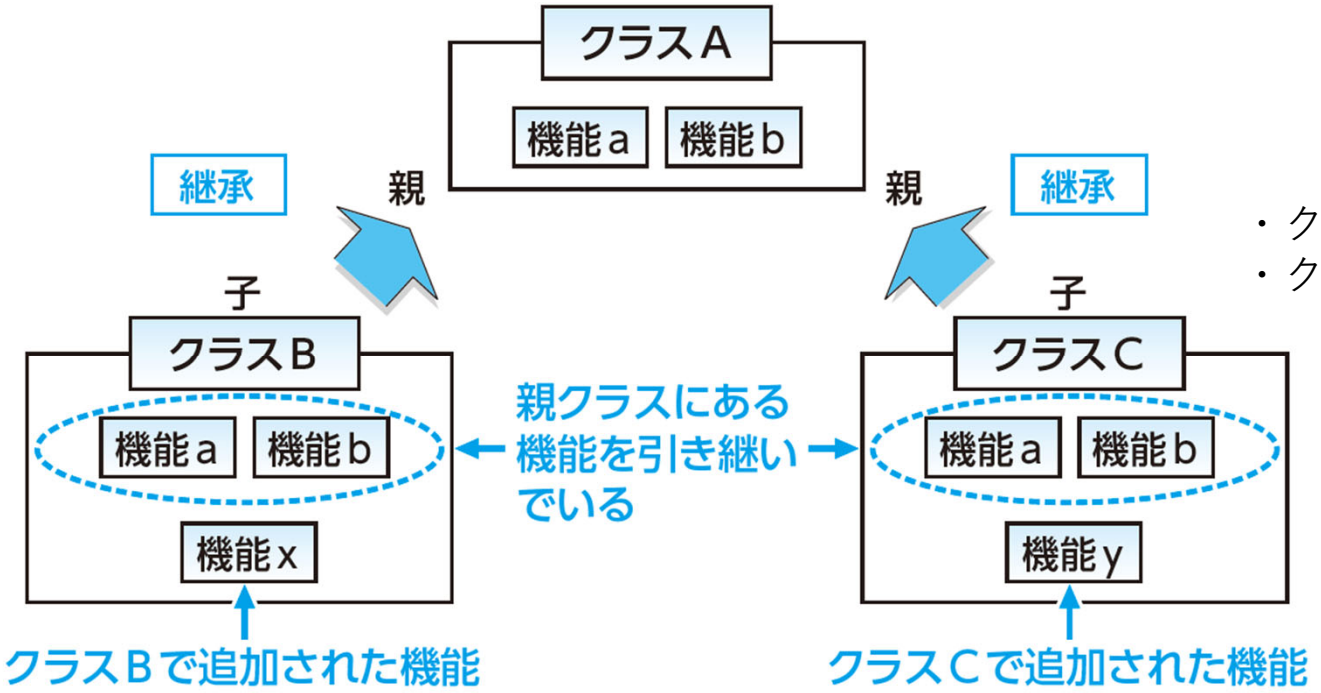
```
底辺が10、高さが5の三角形の面積は25
半径5の円の面積は78.5397
```

実際に試してみよう

継承

継承とは

- すでにあるクラスの機能を新しいクラスが引き継ぐこと。
- 機能の拡張が容易にできる



・クラスAはクラスBの**基底クラス**（親クラス）
・クラスBはクラスAの**派生クラス**（子クラス）

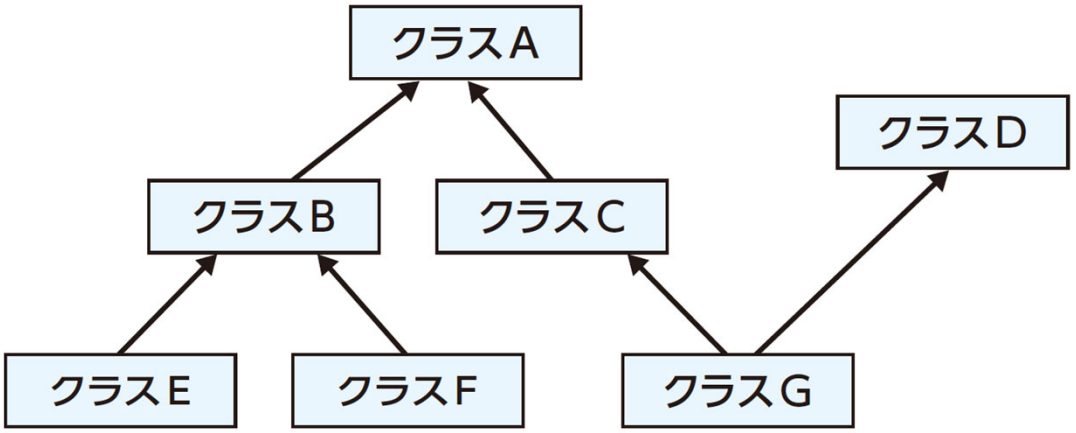
C++言語での継承

- あるクラスから複数の派生クラスを作れる

  （例： クラスBとクラスCはクラスAの派生クラス）

- 複数の基底クラスを持つクラスを作れる（**多重継承**）

  （例： クラスDとクラスCはクラスGの基底クラス）



※ 矢印は派生（子）クラスから基底（親）クラスに向かう
※ 本講義では多重継承を扱わない

継承を行う

他のクラスを継承するクラスの定義

```
class クラス名 : アクセス指定子 基底クラス名 {
    派生クラスのメンバ変数やメンバ関数
};
```

例：クラスAを継承するクラスBの定義

```
class B : public A {
};
```

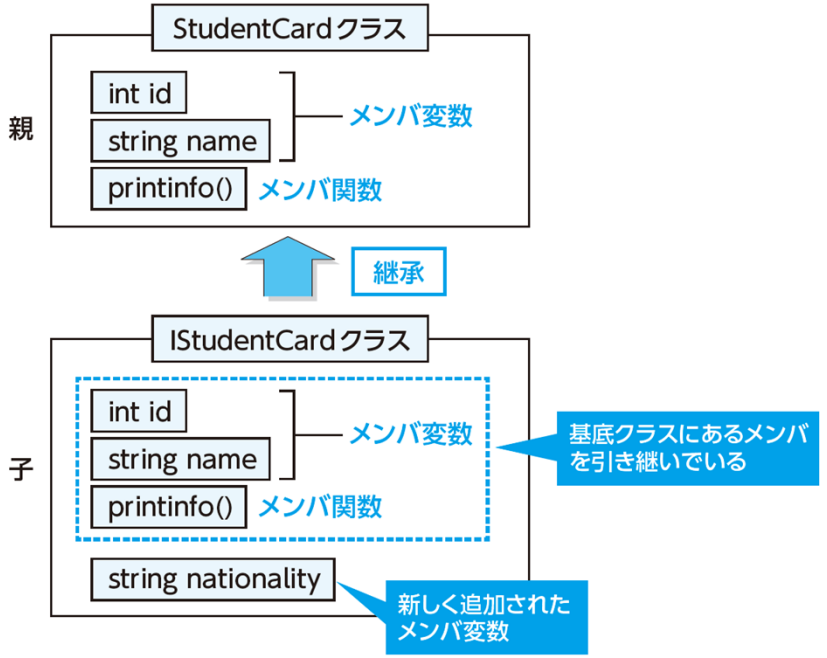
※ クラスBはクラスAのメンバ変数・メンバ関数を引き継ぐ

アクセス指定子の働き

アクセス指定子	説明
public	基底クラスのpublicメンバは派生クラスのpublicメンバになる
protected	基底クラスのpublicメンバもprotectedメンバとして継承される
private	基底クラスのpublicメンバもprivateメンバとして継承される

継承の例

```
#include <iostream>
#include <string>
using namespace std;

class StudentCard {
public:
    int id = 0; // 学籍番号
    string name = "未定"; // 氏名

    void printInfo() {
        cout << "学籍番号：" << id << endl;
        cout << "氏名：" << name << endl;
    }
};

class IStudentCard : public StudentCard {
public:
    string nationality = "未定"; // 国籍
};

int main()
{
    IStudentCard a;
    a.id = 2345;
    a.name = "John Smith";
    a.nationality = "イギリス";
    a.printInfo();
}
```



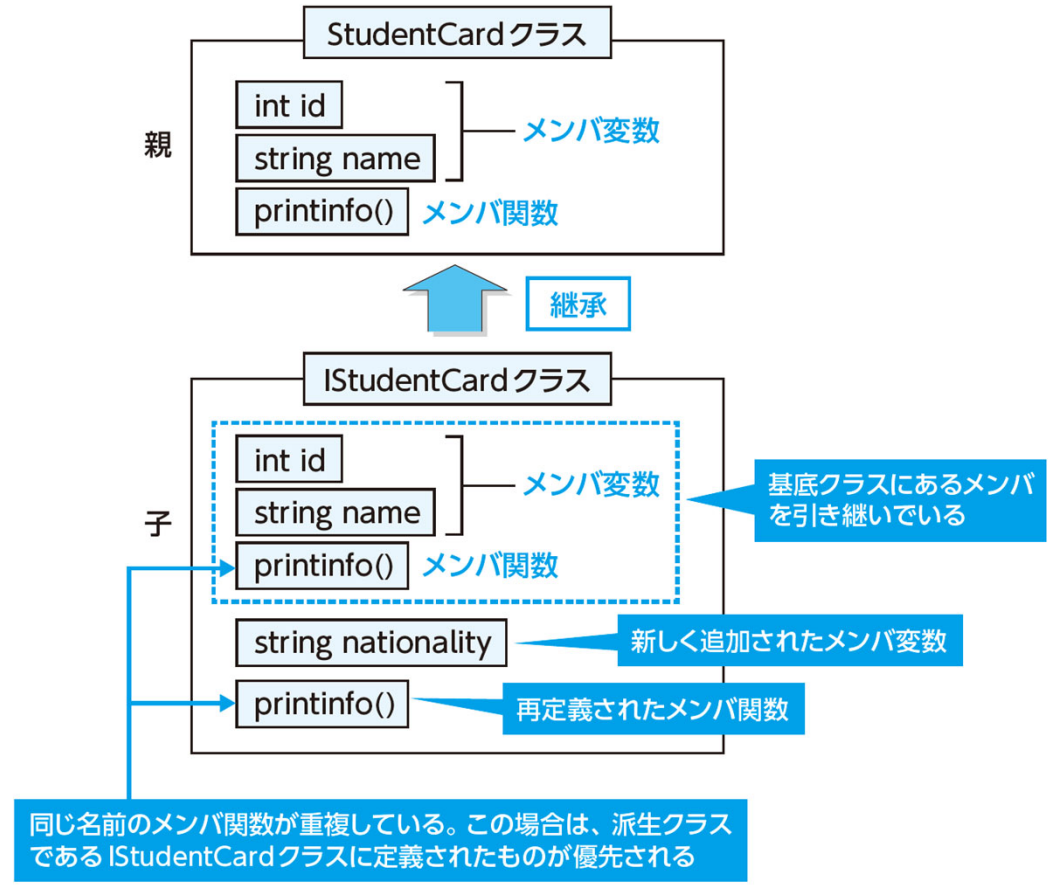
- IStudentCardクラスは StudentCardクラスを継承する
- メンバ変数 nationality が IStudentCardクラスに追加されている
- StudentCardオブジェクトが持つメンバを、IStudentCardオブジェクトも持つ

オーバーライドとポリモーフィズム

派生クラスによるメンバ関数の再定義

```
class StudentCard {
public:
    int id = 0;
    string name = "未定";

    void printInfo() {
        cout << "学籍番号：" << id << endl;
        cout << "氏名：" << name << endl;
    }
};

class IStudentCard : public StudentCard {
public:
    string nationality = "未定";

    void printInfo() {
        cout << "国籍：" << nationality << endl;
    }
};
```

基底クラスが持つメンバ関数のオーバーライド



- 基底クラスのメンバ関数と同じ名前のメンバ関数を、派生クラスで再定義できる。
- 派生クラスのメンバ関数が優先され、基底クラスのメンバ関数は実行されなくなる。これを基底クラスのメンバ関数が**隠蔽**される、という。

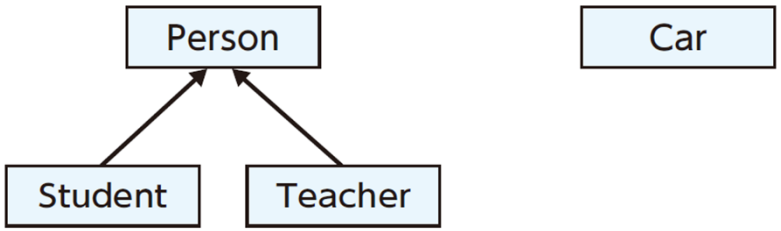
基底クラスのメンバ関数を呼び出す

```
class StudentCard {
public:
    int id = 0;
    string name = "未定";

    void printInfo() {
        cout << "学籍番号：" << id << endl;
        cout << "氏名：" << name << endl;
    }
};

class IStudentCard : public StudentCard {
public:
    string nationality = "未定";

    void printInfo() {
        cout << "国籍：" << nationality << endl;
        StudentCard::printInfo();
    }
};
```

基底クラスが持つメンバ関数を呼び出す

基底クラス名の後ろにコロン(:)を2つ並べてから関数名を書くことで派生クラスから基底クラスのメンバ関数を呼び出せる

クラスの継承と参照



上図の継承関係を持つ4つのクラス

```
class Person { };
class Student : public Person { };
class Teacher : public Person { };
class Car { };
```

Person型オブジェクトの参照を受け取る関数

```
void receive_a_person(Person& p) {
}
```

関数に渡せるオブジェクト

```
Person p;
receive_a_person(p);    ← 当然、何も問題ありません

Student s;
receive_a_person(s);    ← Student型のオブジェクトを渡せます

Teacher t;
receive_a_person(t);    ← Teacher型のオブジェクトを渡せます

Car c;
receive_a_person(c);    ← エラー。Car型のオブジェクトは渡せません
```

✕

※ 派生クラスのオブジェクトも渡すことができる

オーバーライドとポリモーフィズム

```
class Person {
public:
  virtual void work() { cout << "人：仕事する" << endl; }
};

class Student : public Person {
public:
  void work() override { cout << "学生：勉強する" << endl;}
};

class Teacher : public Person {
public:
  void work() override { cout << "教員：授業する" << endl;}
};

void execute_work(Person& person) { person.work(); }

int main() {
  Person person;
  Student student;
  Teacher teacher;
  execute_work(person);
  execute_work(student);
  execute_work(teacher);
}
```

実行結果

```
人：仕事する
学生：勉強する
教員：授業する
```

- virtual キーワードがついたメンバ関数を**仮想関数**と呼ぶ
- 仮想関数を派生クラスが再定義することを**オーバーライド**と呼ぶ
- 基底クラスの型の参照を通じてメンバ関数を呼び出したとき、派生クラスのメンバ関数が実行される。この仕組みを**ポリモーフィズム**と呼ぶ

実際に試してみよう

第6章　標準ライブラリ

テンプレートの仕組み

テンプレートとは

テンプレートを使うと、1つのプログラムコードで複数の型に対応する関数やクラスを定義できる。

例：2つの引数のうち大きい方の値を返す get_max関数を作りたい

型の数だけ関数を定義する必要があるのは不便　→　テンプレートを使用する

int型に対応した関数

```
int get_max(int a, int b) {
    return a > b ? a : b;
}
```

double型に対応した関数

```
double get_max(double a, double b) {
    return a > b ? a : b;
}
```

```
template <typename T>
T get_max(T a, T b) {
    return a > b ? a : b;
}
```

※ 型名を T （プレースホルダ）で表現する
※ 関数を呼び出すときは
get_max<int>(3, 10)
のようにして型を指定する

関数テンプレートの使用例

```
#include <iostream>
using namespace std;

template <typename T>
T get_max(T a, T b) {
    return a > b ? a : b;
}

int main() {
    int i = get_max<int>(10, 4);
    double d = get_max<double>(1.2, 5.2);
    cout << i << ", " << d;
}
```

↓実行結果

```
10, 5.2
```

- 関数テンプレートを使用すると、異なる型に対応する関数を一度書くだけで済む
- 関数を呼び出すときに、型を指定する <int> や <double> の箇所を省略できる（これは、コンパイラが引数を見て推測できるため）

実際に試してみよう

クラステンプレート

- 異なる型のメンバ変数に対応した汎用的なクラスを定義するための仕組み
- `template <typename T>` に続けてクラスの定義を書く

```
template <typename T>
class Box {
private:
    T value;

public:
    Box(T v) : value(v) {}

    // Boxの中身を出力する
    void show() {
        cout << "Boxの中身: " << value << endl;
    }
};
```

メンバ変数の型がTになっています

コンストラクタの引数の型がTになっています

クラステンプレートを使って定義された
Boxクラスの使用例

```
Box<int> boxA(42);
Box<double> boxB(3.14);
Box<string> boxC("hello");
```

コンテナ

標準ライブラリとコンテナ

- **標準ライブラリ**：あらかじめ開発環境に標準で用意されている関数やクラスの集まり
- **コンテナ**：値やオブジェクトを格納し、管理するためのクラス
- **範囲ベースのforループ**： forループの新しい書き方

構文

```
for (型 変数名 : 範囲) {
    各要素に対する処理
}
```

例

```
int array[] = {1, 5, 7, 2, 3};
for (int x : array) {
    cout << x << " ";
}
```

新しい書き方です。配列の要素が順番に変数xに代入されます

※ インデックスを使用しないで各要素にアクセスできる

コンテナの基本：vector（動的配列）

- vectorは配列と同じ用途で使用できる
- 格納できる要素の数が自動的に調整される
- 最初にサイズを指定する必要がなく、後から要素を好きなだけ追加できる

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v = { 1, 2, 3 };
    v.push_back(4);
    v.push_back(5);
    cout << v.size() << endl;
    cout << v.front() << endl;
    cout << v.back() << endl;
    v.pop_back();
    v[1] = 10;

    for (int x : v) {
        cout << x << " ";
    }
}
```

実行結果

```
5
1
5
1 10 3 4
```

vectorクラスの主なメンバ関数

関数	説明
size()	現在の要素数を返す
empty()	空である場合にtrueを返す
clear()	全要素を削除して、サイズを0にする
front()	最初の要素を返す
back()	最後の要素を返す
push_back(value)	要素を末尾に追加する
pop_back()	末尾の要素を削除する

実際に試してみよう

コンテナの基本：list（双方向連結リスト）

- listはvectorとほぼ同じ用途で使用できる
- インデックスで要素にアクセスできない
- 要素の追加と削除をvectorよりも高速に実行できる

```
#include <iostream>
#include <list>
using namespace std;

int main() {
    list<int> l = { 10, 20, 30 };
    l.push_front(5);
    l.push_back(40);
    l.remove(20);

    for (int x : l) {
        cout << x << " ";
    }
}
```

実行結果

```
5 10 30 40
```

listクラスの主なメンバ関数

関数	説明
size()	現在の要素数を返す
empty()	リストが空である場合にtrueを返す
clear()	全要素を削除してサイズを0にする
front()	最初の要素を返す
back()	最後の要素を返す
push_back(value)	要素を末尾に追加する
push_front(value)	要素を先頭に追加する
pop_back()	末尾の要素を削除する
erase(it)	指定した要素を削除する

実際に試してみよう

コンテナの基本： map （連想配列）

- mapは「キー」と「値」のペアで要素を管理する
- キーを使って要素にアクセスする
- 重複するキーを持つことはできない

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {
    map<string, int> scores;
    scores["Suzuki"] = 86;
    scores["Yamada"] = 92;

    cout << scores["Suzuki"] << endl;
    cout << scores["Yamada"] << endl;
}
```

実行結果

```
86
92
```

mapクラスの主なメンバ関数

関数	説明
`size()`	現在の要素数を返す
`empty()`	空である場合に`true`を返す
`clear()`	全要素を削除して、サイズを0にする
`erase(key)`	指定したキーの要素を削除する
`find(key)`	指定したキーを検索する

実際に試してみよう

コンテナの基本： set （集合）

- setは格納されるオブジェクトに重複がないことを保証する
- insertメンバ関数で要素を追加する
- インデックスで要素にアクセスできない
- 要素は昇順（値が小さい順）に格納される

```
#include <iostream>
#include <set>
using namespace std;

int main() {
    set<int> s = { 10, 2, 8 };

    s.insert(2);
    s.insert(5);

    for (int x : s) {
        cout << x << " ";
    }
}
```

実行結果

```
2 5 8 10
```

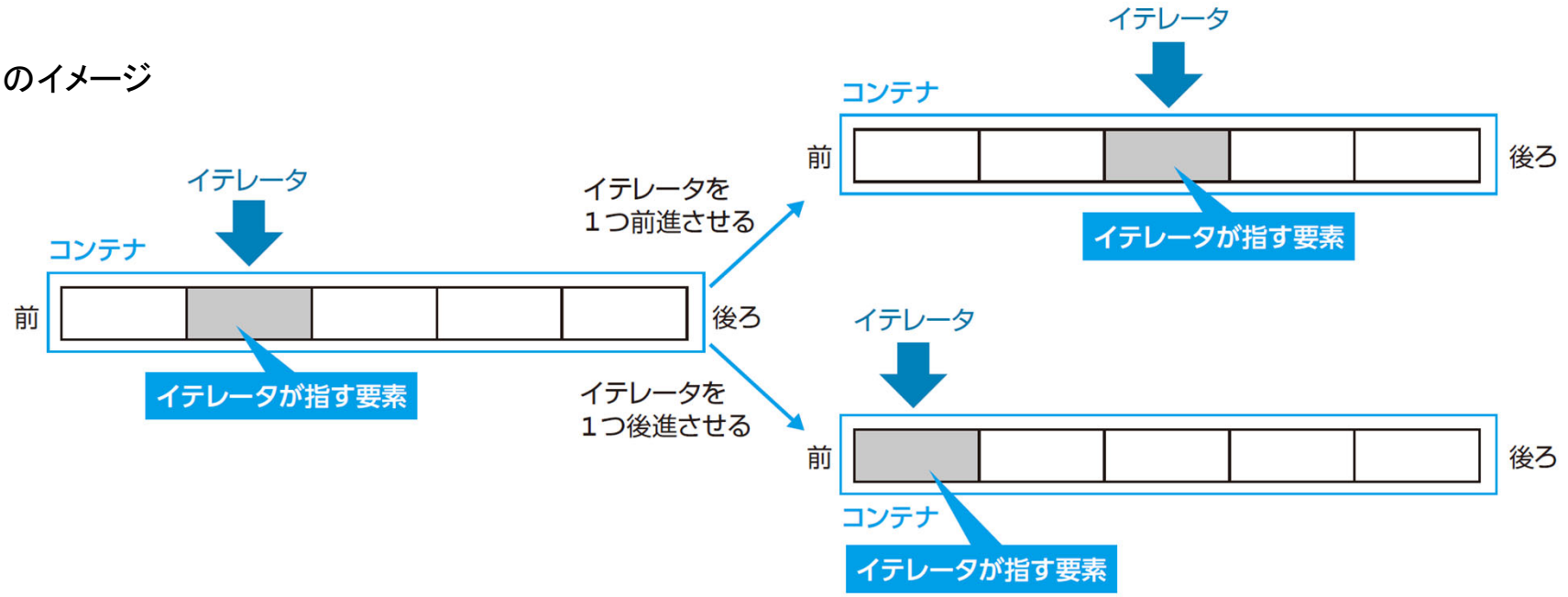
実際に試してみよう

アルゴリズムとイテレータ

イテレータ

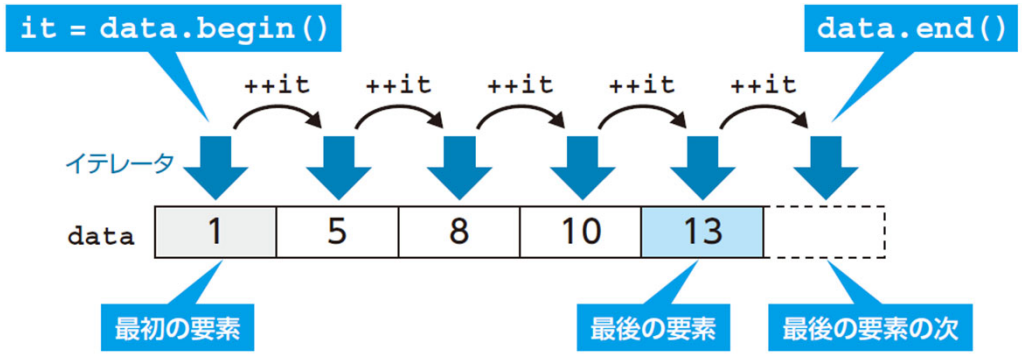
- イテレータは、コンテナに格納されている要素を指し示すことができるオブジェクト
- イテレータを使うと、どのようなコンテナに対しても同じような書き方で、コンテナに格納されているオブジェクト1つ1つにアクセスできる

イテレータのイメージ



イテレータの使用

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<int> data = { 10, 20, 30 };
    list<int>::iterator it = data.begin();

    while (it != data.end()) {
        cout << *it << " ";
        ++it;
    }
}
```



- int型の値を格納するlistの要素にアクセスするためのイテレータ it の型は list<int>::iterator
- 先頭の要素を指すイテレータを取得するには、listクラスのbegin()関数を使用する
- end()関数の戻り値は「最後の要素の次」を指すイテレータ
- *it で要素の値にアクセスできる
- ++it でイテレータを1つ先に進める

アルゴリズム

- 標準ライブラリには、コンテナに格納されている要素の並べ替え（sort）、検索（find）、削除（remove）などを行う関数が数多く用意されている
- これらをアルゴリズム（algorithm）とよぶ
- 使用するには、次のようにして algorithm ヘッダーをインクルードする

```
#include <algorithm>
```

アルゴリズムの活用（sort）

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> vec = {5, 3, 8, 1, 4};
    sort(vec.begin(), vec.end());

    for (int x : vec) {
        cout << x << " ";
    }
}
```

実行結果

```
1 3 4 5 8
```

- sort関数を用いて、コンテナ内の要素を値が小さい順、または大きい順に並べ替えることができる
- sort関数には、2 つのイテレータを引数に渡すことで、並べ替えを行う範囲の始まりと終わりを指定する
- 第1引数のイテレータが指す要素と、第2引数のイテレータの1つ手前の要素までが並べ替えの対象となる

実際に試してみよう

アルゴリズムの活用（find）

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main() {
    vector<char> data = {'A', 'D', 'Z', 'P'};
    vector<char>::iterator it =
       find(data.begin(), data.end(), 'D');

    if (it != data.end()) {
        cout << "見つかりました";
    }
    else {
        cout << "見つかりませんでした";
    }
}
```

実行結果

```
見つかりました
```

- find関数を使用して、コンテナの中に特定の要素が含まれるかどうか知ることができる
- 検索を行う範囲の始まりと終わりを指定する2つのイテレータと、検索したい値を引数にする
- 要素が見つかった場合は、その要素を指すイテレータが戻り値になる。要素が見つからなかった場合は、末尾の要素の次を指すイテレータ（コンテナのend関数の戻り値）が戻り値になる

実際に試してみよう

代表的なアルゴリズム

- std::min_element　範囲内の最小値を見つける

  例
```
vector<int>::iterator min_it = std::min_element(v.begin(), v.end());
```

- std::max_element　範囲内の最大値を見つける

  例
```
vector<int>::iterator max_it = std::max_element(v.begin(), v.end());
```

- std::count 指定した値が範囲内にいくつあるか数える

  例
```
int num = std::count(v.begin(), v.end(), 5);
```

- std::reverse 範囲内の要素の並び順を逆にする

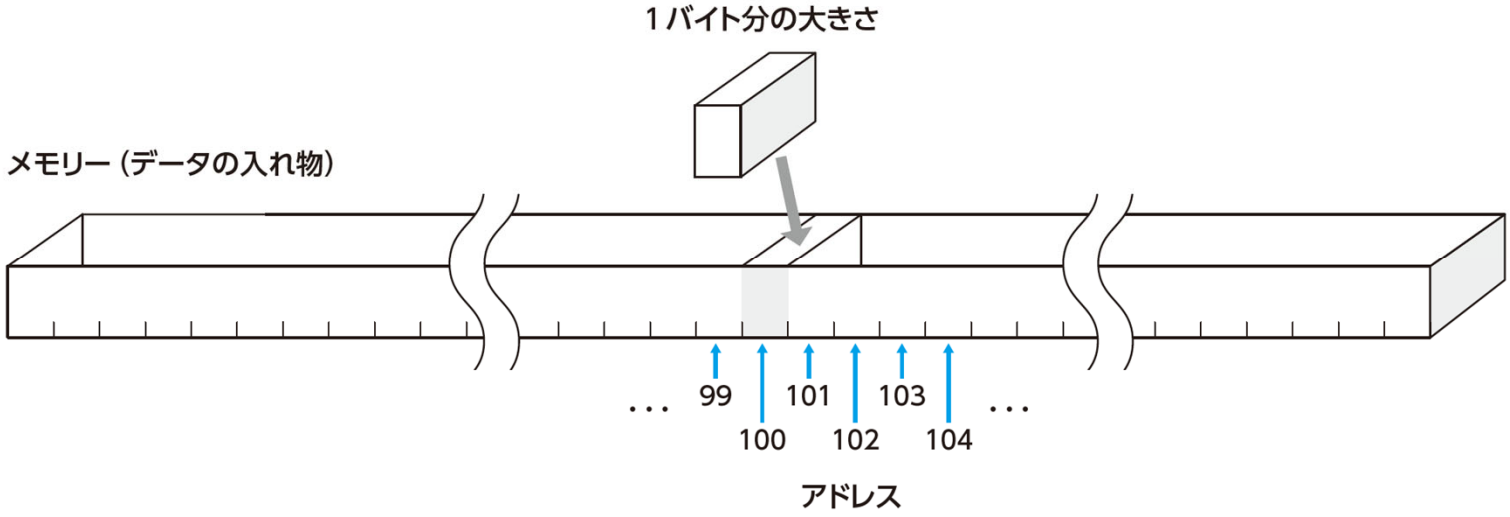
  例
```
std::reverse(v.begin(), v.end());
```

第7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

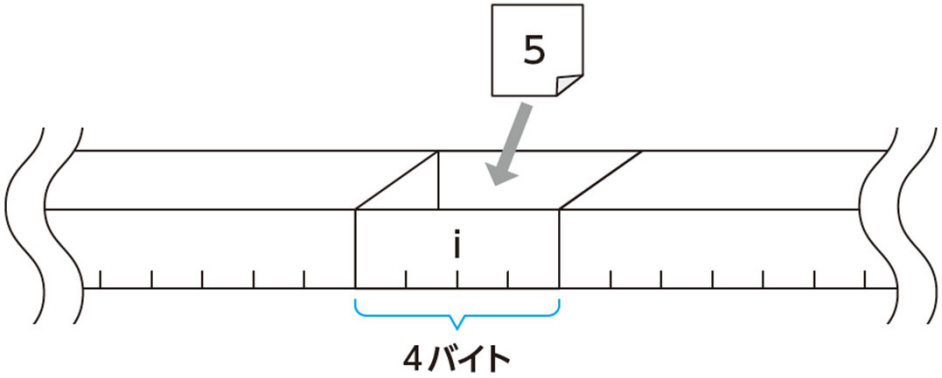
- コンピューターがプログラムを実行している途中に扱う、さまざまな情報がメモリーに格納される
- メモリーを、1バイト単位で区切れるように目盛りがついた、横に長い箱のようなものだとみなす
- 目盛りに割り当てられた数字（番地）を**アドレス**という



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




```
cout << &i;
```

← メモリー上の変数iの位置（アドレス）を知る

↓ **実行結果**

```
000000A7046FF6E4
```

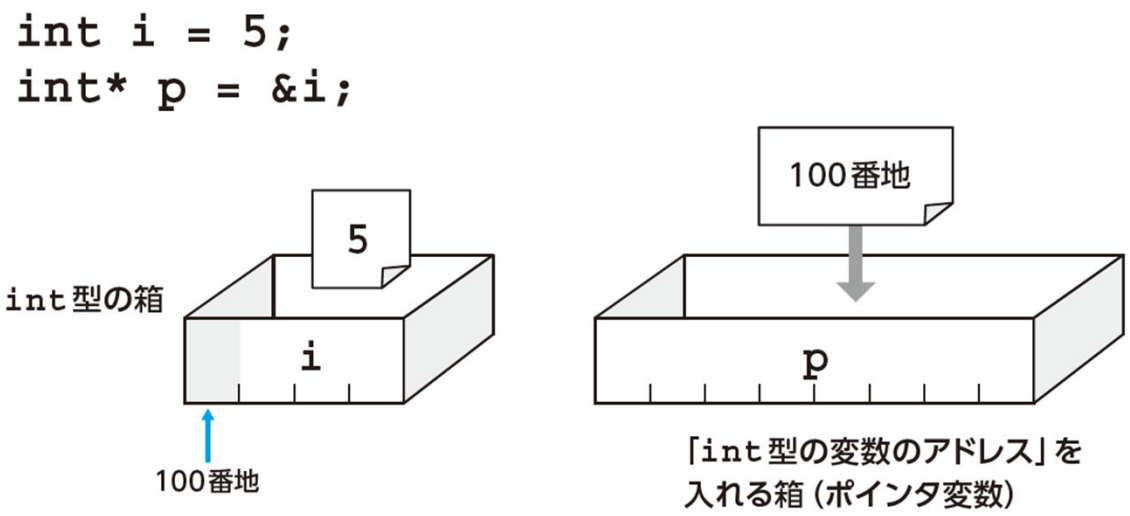
← 16進数表現（実行のたびに変化する。具体的な値にあまり意味はない）

&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8バイトで表している

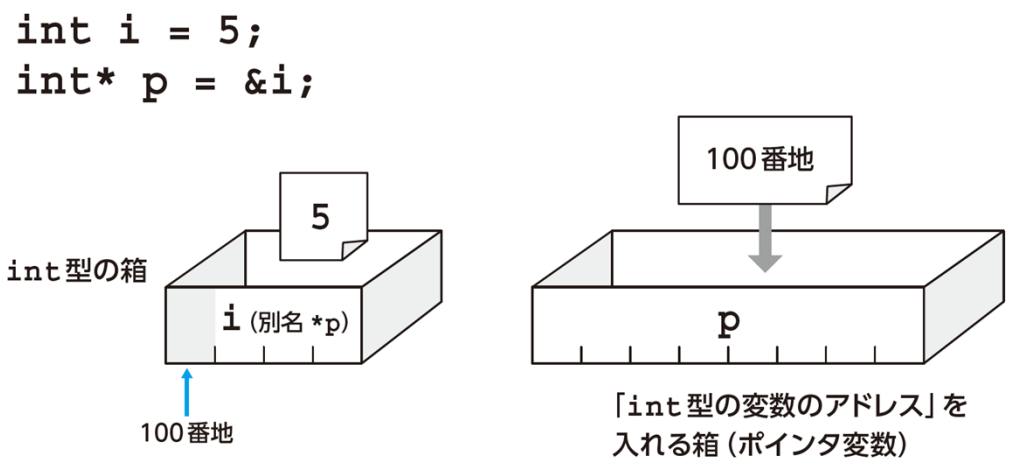
ポインタを使った値の参照

- ポインタ変数pに入っているアドレスの場所に存在する値にアクセスするには、プログラムコードの中で「*p」と書く
- 「i」と「*p」は、メモリー上の同じ値を参照する

```
int i = 5;
int* p = &i;
cout << "iのアドレスは" &i << endl;
cout << "pの値は" p << endl;
cout << "iの値は" i << endl;
cout << "*pの値は" *p << endl;
```

実行結果

```
iのアドレスは0000002376EFFD24
pの値は0000002376EFFD24
iの値は5
*pの値は5
```



実際に試してみよう

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;
cout << "jの値は" << j << endl;
*p = 10;
cout << "iの値は" << 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

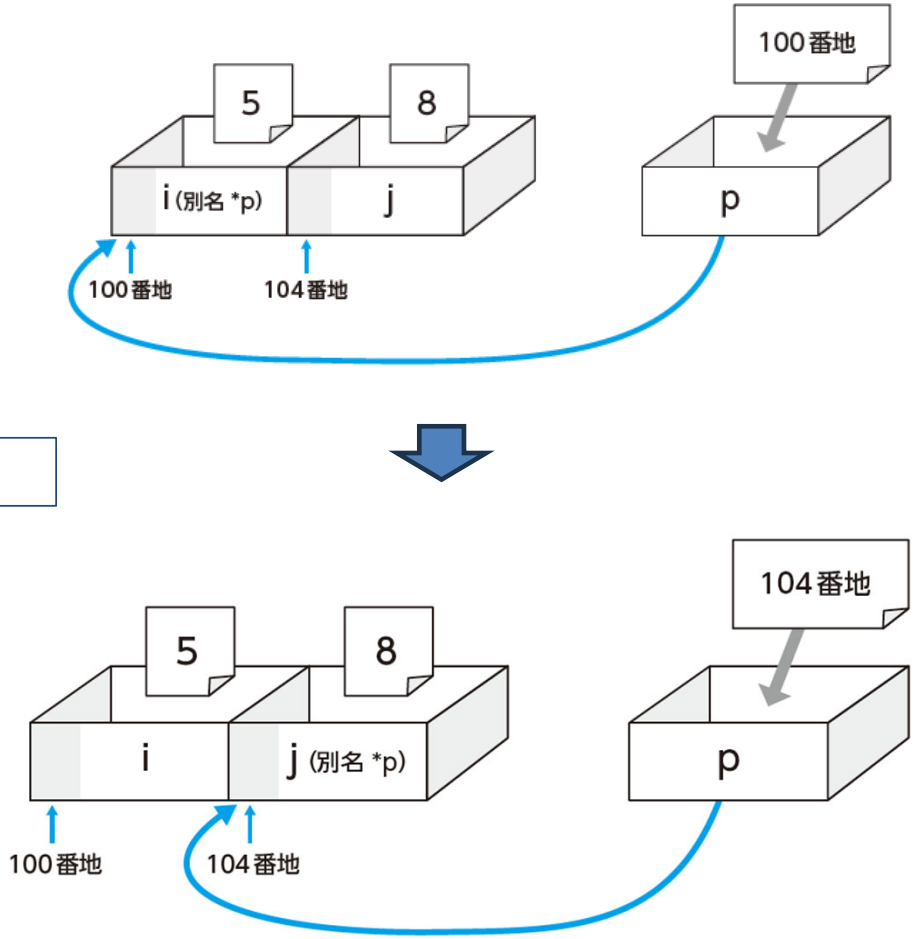
ポインタが指す先を変更する

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

int* p = &i;
cout << "*pの値は" << *p << endl;

p = &j;
cout << "*pの値は" << *p;
```

ポインタ変数p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
```

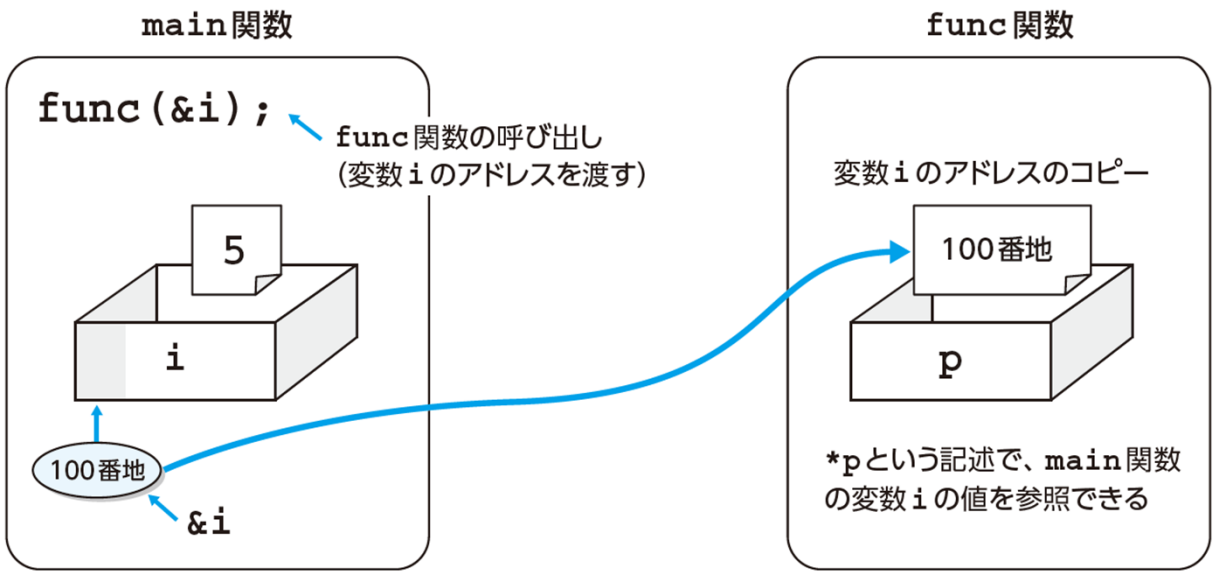


ポインタを引数とする関数

```
#include <iostream>
using namespace std;

void func(int* p)
{
    *p = 10;
}

int main()
{
    int i = 5;
    func(&i);
    cout << "iの値は" << 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

```
#include <iostream>
using namespace std;

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

int main() {
    int a = 2;
    int b = 3;
    cout << "a=" << a << ", b=" << b << endl;
    cout << "swap関数を呼び出します" << endl;
    swap(&a, &b);
    cout << "a=" << a << ", b=" << 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

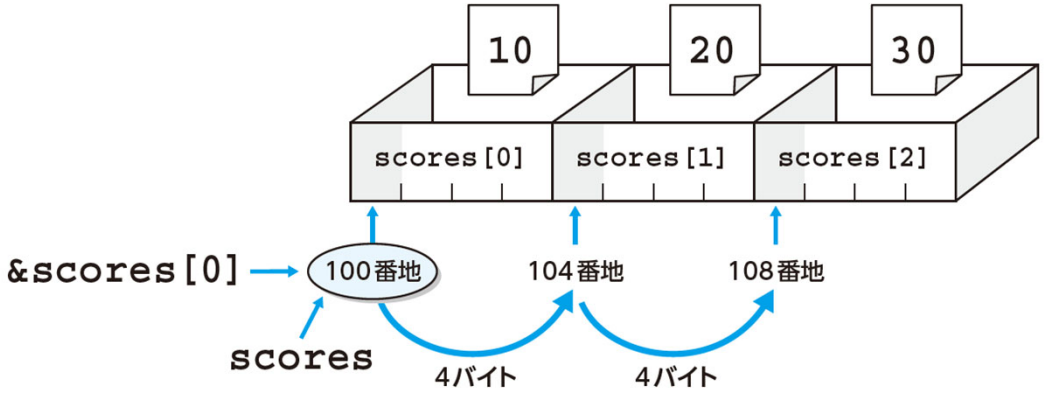
```
int scores[3] = {10, 20, 30};
cout << "scores[0]のアドレス:" << &scores[0] << endl;
cout << "scores[1]のアドレス:" << &scores[1] << endl;
cout << "scores[2]のアドレス:" << &scores[2];
```

↓ 実行結果

```
scores[0]のアドレス:000000A6698FF6A8
scores[1]のアドレス:000000A6698FF6AC
scores[2]のアドレス:000000A6698FF6B0
```

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];
```

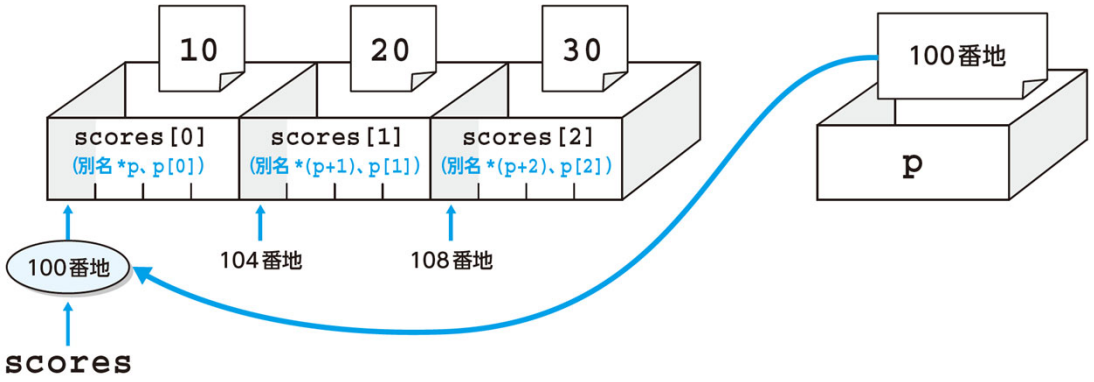
```
int* p = scores;
```

- 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

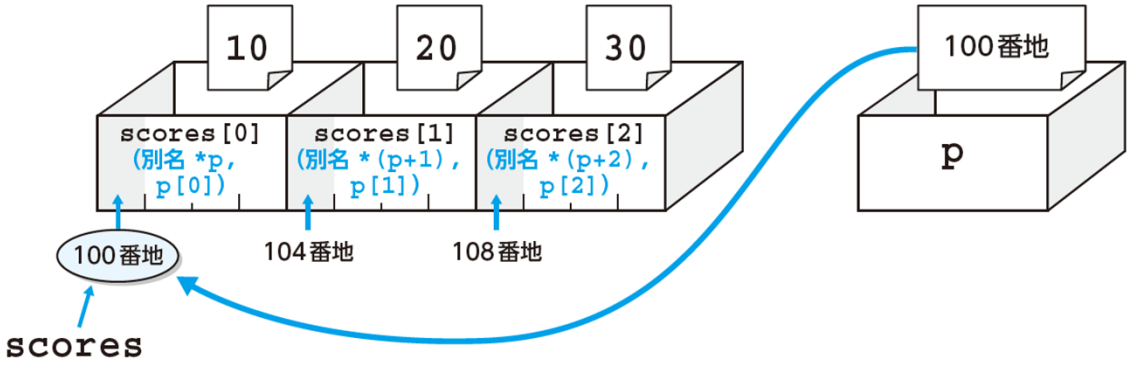
```
int scores[] = {10, 20, 30};
int* p = scores;

cout << "*pの値は << *p << endl;
cout << "*(p + 1)の値は" << *(p + 1) << endl;
cout << "*(p + 2)の値は" << *(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

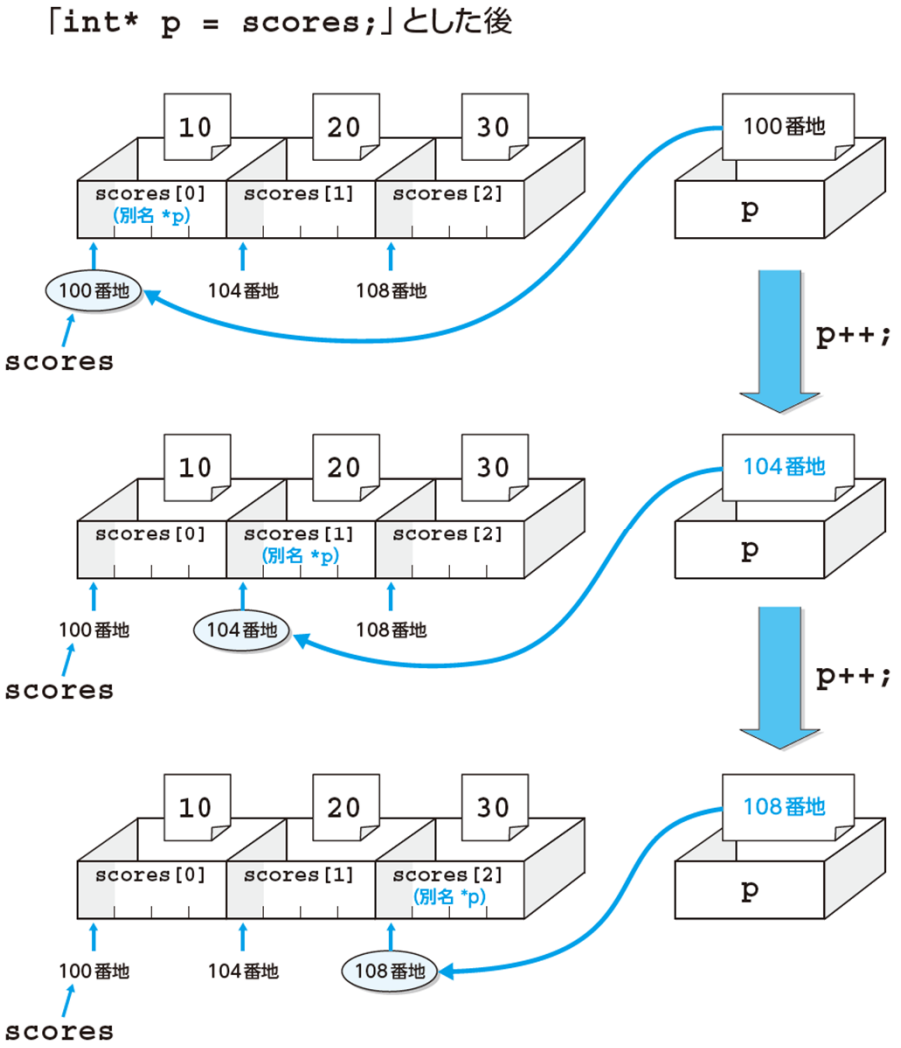
ポインタが配列の先頭の要素を指しているとき、ポインタ変数の値を1だけ増やすと、配列の次の要素を指すようになる。

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

cout << *p << endl;
p++;
cout << *p << endl;
p++;
cout << *p ;
```

↓実行結果

```
10
20
30
```

実際に試してみよう



newを用いた動的なメモリー確保

```
#include <iostream>
using namespace std;

int main() {
    int size;
    cout << "配列のサイズを入力してください: ";
    cin >> size;

    int* a = new int[size];

    for (int i = 0; i < size; i++) {
        a[i] = i + 1;
    }

    for (int i = 0; i < size; i++) {
        cout << a[i] << " ";
    }

    delete[] a;
}
```

実行結果

```
配列のサイズを入力してください:
5
1 2 3 4 5
```

- 配列ははじめに要素の数が決まっている必要がある
- new キーワードを使うとプログラム実行時に必要な大きさのメモリーを確保できる（**動的なメモリー確保**という）
- int* a = new int[size]; と書くとプログラム実行時のsizeの値で配列を生成できる
- a[0] = 5; のように、通常の配列と同じように使用できる
- 使用する必要がなくなったときには delete[] a; で、メモリーを解放する

実際に試してみよう

第8章　一歩進んだC++プログラミング

複数ファイルへの分割

関数の定義を別のファイルで行う

triangle.h (関数のプロトタイプ宣言)

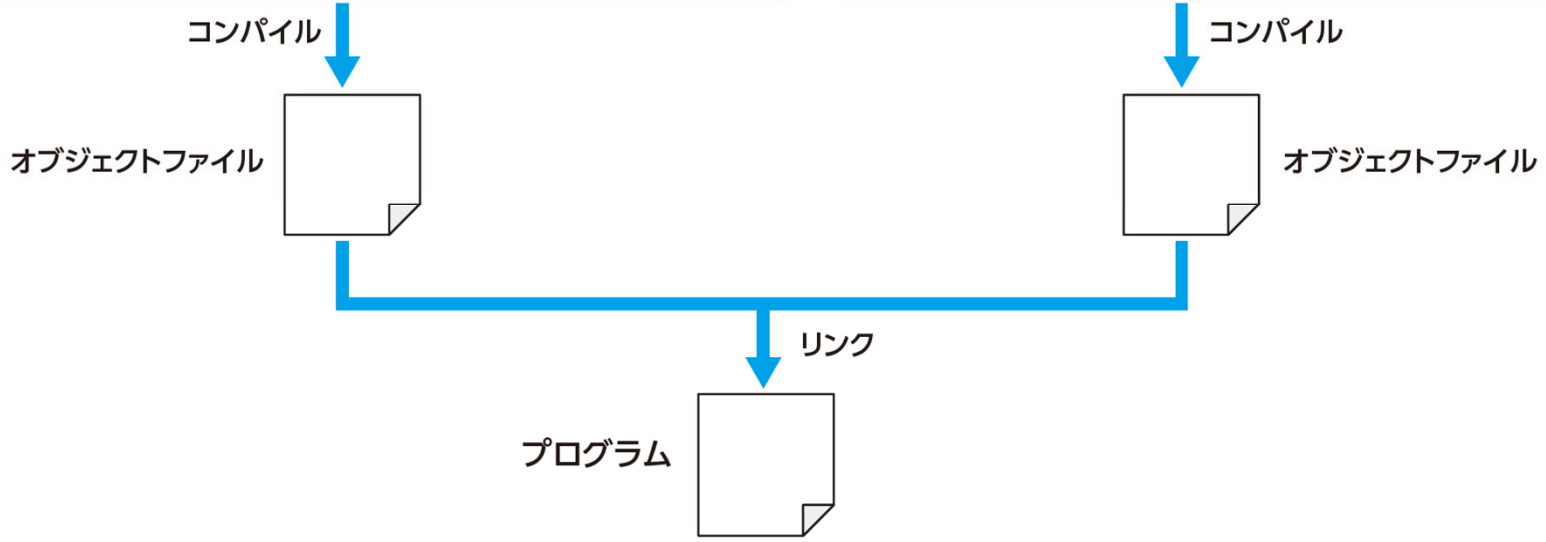
```
double triangle_area(double base,
                     double height);
```

triangle.c (関数の定義)

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

main.cpp

```
#include <iostream>
#include "triangle.h"
using namespace std;

int main(void)
{
    double area = triangle_area(3.0, 5.0);
    cout << "面積は" << area;
}
```



クラスの定義を別のファイルに分ける

StudentCard.h

```
#pragma once
#include <string>
using namespace std;

class StudentCard {
public:
    int id = 0;
    string name = "未定";

    void printInfo();
};
```

StudentCard.cpp

```
#include <iostream>
#include "StudentCard.h"
using namespace std;

void StudentCard::printInfo() {
    cout << "学籍番号:" << id << endl;
    cout << "氏名:" << name << endl;
}
```

main.cpp

```
#include "StudentCard.h"

int main()
{
    StudentCard a;
    a.id = 1234;
    a.name = "鈴木太郎";

    a.printInfo();
}
```

- ファイルを分けることでプログラムコードの管理がしやすくなる
- クラス単位での再利用がしやすくなる

- ヘッダーファイルの冒頭にと `#pragma once` 書く。（複数回インクルードされるのを防ぐ）

ローカル変数とグローバル変数

```
#include <iostream>
using namespace std;

int a = 10;

void func() {
    a++;
}

int main()
{
    func();
    cout << a;
}
```

グローバル変数です

グローバル変数の値を変更しています

グローバル変数の値を出力します

- 変数は、宣言されたブロックの内部だけで使用できる（**ローカル変数**）
- すべてのブロックの外側で宣言された変数は、どの関数でも使用できる（**グローバル変数**）
- 同じ名前が使用された場合はローカル変数が優先される

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();
    cout << "%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

グローバル変数を複数のファイルで共有する

StudentCard.h

```
#pragma once
extern int g_count;
double triangle_area(double base, double height);
```

StudentCard.cpp

```
#include "triangle.h"

int g_count = 0;
double triangle_area(double base, double height)
{
    g_count++;
    return base * height / 2.0;
}
```

main.cpp

```
#include <iostream>
#include "triangle.h"
using namespace std;

int main()
{
    double area = triangle_area(3.0, 5.0);
    cout << "面積は" << area << endl;
    cout << "g_countの値は" << g_count;
}
```

ファイル入出力と例外処理

テキストファイルの読み込み

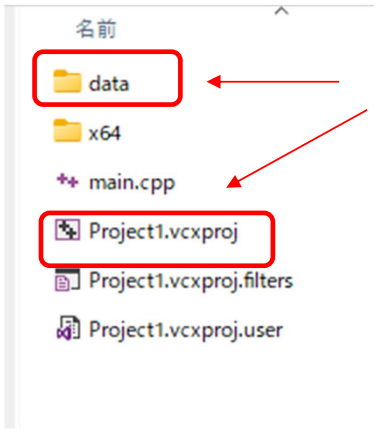
- **テキストファイル**：文字列が保存されたファイル
- **バイナリファイル**：テキストファイルではないファイル（画像データ、音声データなどのファイル）

- テキストファイルの読み込みの手順
  1. ファイルを開く（`std::ifstream`を使用する）
  2. ファイルから文字列を読み込み、読み取った内容に対して処理を行う（`std::getline`関数や`>>`演算子を使用する）
  3. ファイルを閉じる（`std::ifstream`の`close`関数を使用する）ファイルを開く

ファイルの読み込みとファイルの保存場所

```
ifstream ifs("data/sample.txt");
```

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

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    // 1. ファイルを開く
    ifstream ifs("data/sample.txt");
    if (!ifs) {
        cerr << "ファイルを開けませんでした。";
        return 1;
    }

    // 2. ファイルから文字列を読み込んで処理する
    string line;
    int line_count = 1;
    while (getline(ifs, line)) {
        cout << line_count << ": " << line << endl;
        line_count++;
    }

    // 3. ファイルを閉じる
    ifs.close();
}
```

実際に試してみよう

実行結果

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

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（std::ofstreamを使用する）
2. 文字列をファイルに書き出す（<< 演算子を使用する）
3. ファイルを閉じる（std::ofstreamのclose関数を使用する）

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ofstream ofs("output.txt");
    if (!ofs) {
        cerr << "ファイルを開けませんでした";
        return 1;
    }

    for (int i = 0; i < 10; i++) {
        ofs << "[" << i << "]" << endl;
    }

    ofs.close();
}
```

実際に試してみよう

例外処理

- プログラムを実行させたときに発生するエラーを「ランタイムエラー」という
- プログラムを実行している途中に発生する「通常の処理では対処できない問題」のことを**例外**と呼ぶ
- こうした問題を検知して、呼び出し元に通知し、適切な処理を行う仕組みが「例外処理」

```
void divide(double a, double b)
{
    double c = a / b;
    cout << c;
}
```

↑ このようなプログラムコードでは、
引数の**b**に値0が渡されたときに例外が発生する

※ 例外処理が必要となる

例外処理の構文

例外処理に使用するキーワード

- try　　例外が起きそうな処理を{ }で囲む
- throw　　例外オブジェクトを投げる
- catch　　投げられた例外を受け取って処理する

例外処理の構文

```
try {
    例外が発生するかもしれない処理
    if （例外が発生する条件） {
        throw 例外オブジェクト;　← 例外オブジェクトを投げます
    }
    例外が発生しなかった場合の続きの処理
}
catch （例外型 変数名） {　← 例外オブジェクトを受け取ります
    例外を受け取ったときの処理
}
```

※ 標準ライブラリで提供される
例外オブジェクトのためのクラス

std::exception
すべての例外クラスの基底クラス

std::runtime_error
実行時の一般的なエラー

std::logic_error
プログラムロジック上のエラー

std::out_of_range
コンテナの範囲外アクセスなどのエラー

std::bad_alloc
new 演算子によるメモリ確保失敗

例外処理の例

```
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

int main() {
    double a, b;
    cout << "2つの数値を半角スペース区切りで入力してください" << endl;
    cin >> a >> b;

    try {
        if (!cin) { throw runtime_error("数として正しく読み取れません"); }
        if (b == 0) { throw runtime_error("0で除算できません"); }

        double result = a / b;
        cout << a << " / " << b << " = " << result << endl;
    }
    catch (const runtime_error& e) {
        cerr << "エラー: " << e.what() << endl;
    }
    cout << "プログラム終了";
}
```

実際に試してみよう

演算子のオーバーロードと型推論

演算子のオーバーロード

算術演算子 +、-、*、/ に新しい役割を与えることができる

例：2つのPoint オブジェクトのx,y座標値をそれぞれ足し合わせた新しいPointオブジェクトを + 演算子で作りたい

```
class Point {
private:
    int x;
    int y;
public:
    Point() : x(0), y(0) {}
    Point(int x, int y) : x(x), y(y) {}
};
```

```
Point p1(1, 3);
Point p2(2, 5);
```

これまでに学習した方法

```
Point p3;
p3.x = p1.x + p2.x;
p3.y = p1.y + p2.y;
```

+演算子のオーバーロードした場合

```
Point p3 = p1 + p2;
```

※ 通常の足し算のように書くことができる

演算子のオーバーロードの実現

```
class Point {
private:
    int x;
    int y;

public:
    Point() : x(0), y(0) {}
    Point(int x, int y) : x(x), y(y) {}

    // 2項演算子+のオーバーロード
    Point operator+(const Point& other) const {
        return Point(x + other.x, y + other.y);
    }

    // 2項演算子-のオーバーロード
    Point operator-(const Point& other) const {
        return Point(x - other.x, y - other.y);
    }

    // 2項演算子==のオーバーロード
    bool operator==(const Point& other) const {
        return (x == other.x && y == other.y);
    }
};
```

演算子のオーバーロードの構文

```
演算結果を表す型 operator 演算子(引数リスト) {
    処理内容
    return 演算結果;
}
```

左のようにして演算子をオーバーロードすると、
次のようなプログラムコードを書くことができる

```
Point p1(1, 3);
Point p2(2, 5);

Point p3 = p1 + p2; // Pointオブジェクトどうしの加算
Point p4 = p2 - p1; // Pointオブジェクトどうしの減算

if (p3 == p4) { // Pointオブジェクトどうしの比較
    cout << "同じ座標です";
}
```

実際に試してみよう

自動型推論

autoキーワードを使用すると、コンパイラが型を自動的に推論するため、プログラムコードに型を書かなくても済む

```
int i = 5;
double d = 2.3;
```

↓ autoキーワードを使った書き方

```
auto i = 5;
auto d = 2.3;
```

```
vector<int> v = {1, 3, 5};
vector<int>::iterator it = v.begin();
```

↓ autoキーワードを使った書き方

```
vector<int> v = {1, 3, 5};
auto it = v.begin();
```

実際に試してみよう

終