C++プログラミングI
テキスト

２０２０年９月

成蹊大学理工学部
情報科学科

目 次

プログラミング言語 C++の位置づけ

コンパイラとインタプリタ

プログラミング言語を学ぶにはコンパイラとインタプリタというソフトウエアについて知っておくことが重要です。コンパイラはある言語で書かれたプログラムを結果が同一となる別言語のプログラムに変換します（図 1a)。この図はハードウエア上でコンパイラというソフトウエアが実行され、ソースプログラムがコンパイラの入力、ターゲットプログラムが出力を表しています。多くのコンパイラは特定のコンピュータ用のアセンブリ言語や機械語を変換対象とします。機械語プログラムはハードウエア上でそのまま実行できます (図 1b)。一方でインタプリタはある言語で書かれたプログラムを解釈すると同時に実行も行います (図 2)。この図はハードウエア上でインタプリタが実行されて、その上でソースプログラムが実行されているイメージです。インタプリタはソフトウエアを指す用語ですが、コンピュータの CPU は機械語を解釈し同時に実行するものなので、CPU をハードウエアのインタプリタと考えることもできます。結局、プログラムは解釈され、必要に応じて変換され、そして実行されるのです。この三つのステップが、どのタイミングで、何によって行われるかを意識することが大切です。

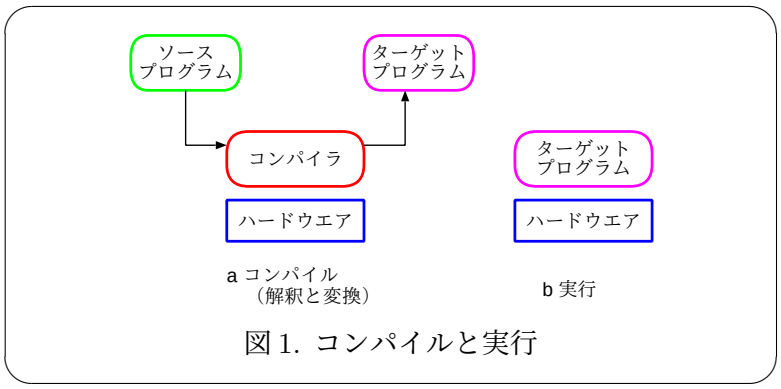


図 1. コンパイルと実行

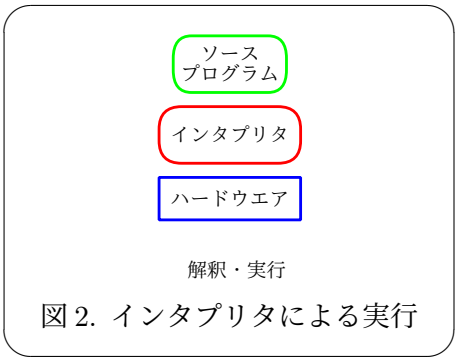


図 2. インタプリタによる実行

コンパイラを使ってある言語のプログラムを機械語に変換するのは、実行の準備段階で行われます。変換作業をコンパイルと呼びます。コンパイルはある程度の時間がかかる作業ですが、結果となる機械語は CPU で直接実行できるため、何度も利用する場合には事前準備に価値があります。つまり、時間のかかる解釈と変換の作業を事前に行っておき、何度も行われる実行を短時間で終わらせるようにするのがコンパイラを使う一つの目的です。一方でインタプリタを利用すると、開発途中のプログラムの一部を試しに実行してみたり、事前にコンパイルする方法よりも自由度の高いプログラムを書くことができます。そのような利点はあるのですが、ソースコードの解釈、必要に応じた変換、そして実行が全体の時間に含まれており、それが長く感じられることがあります。

もう一点知っておくべきことは、コンパイラとインタプリタはまったく別のソフトウエアではないということです。ある言語のプログラムを入力として受け付けて解釈する部分はほぼ共通です。インタプリタはその解釈と実行の過程において、解釈したプログラムを内部の中間表現プログラムに変換することがあります。逆に、コンパイラがコンパイルの過程で、変換対象のプログラムを部分的に実行してしまう場合もあります。

コンパイラとインタプリタのイメージをつかんだところで、具体的なプログラミング言語を例を見てみましょう。

様々なプログラミング言語

プログラミング言語には様々なものがありますが、その中でも C, C++, Python, Java の四つの言語はいつも人気ランキングの上位に入っています[1]。C と C++はコンパイラの使用を想定して設計され、Python

[1] https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020

はインタプリタの使用を想定して設計されました。Java はコンパイラの使用を想定していますが、変換対象は Java バイトコードと呼ばれる Java 仮想マシンの機械語で、これはインタプリタによって実行されます。どのプログラミング言語もデスクトップアプリ、機械学習、科学技術計算に使用されます。その他の用途としては、Python と Java が Web サイトとそのアプリ開発に多く使われ、C, C++, Java が携帯機器のアプリ開発に多く使われます。Java が携帯機器アプリ開発で多く使われるのは、Android 携帯電話の開発言語となっているためです。また、C, C++, Python は組み込み機器のプログラム開発にも使われます。Python が組み込み機器のプログラムに多く使われるのは、Raspberry Pi というシングルボードコンピュータの開発言語に選ばれているためです。

C++の特徴と規格

C++に絞って特徴を説明しましょう。この言語は、手続き型プログラミング・データ抽象・オブジェクト指向プログラミング・ジェネリックプログラミング、そして関数型プログラミングといった、複数のプログラミングパラダイムを利用できるようになっています。また、C 言語を拡張して設計されているために、ハードウエアを直接操作する低水準言語としても利用できます。そのため、オペレーティングシステム、Web ブラウザ、オフィススイート、電子メールクライアント、マルチメディアソフトウエア、データベースなどの著名なソフトウエアの開発言語となっています。C++は 1983 年に初めて公開され、上記の四つの言語の中では C 言語に次いで二番目に長い歴史を持っています[2]。良く使われるコンパイラには、GNU g++, Clang, Microsoft Visual C++, Intel C++ Compiler など複数あります。年月とともに言語機能が拡張されていますが、国際規格として標準化されていったために、これら開発者の異なるコンパイラを使用してもソースコードを変更することなくコンパイルできます。このように長く様々なところで使用され、最新の技術を取り込んで拡張されて、そして、様々な安定したコンパイラが利用可能なプログラミング言語が C++なのです。

ただしこの言語を学ぶ上で一つ注意があります。複数回の言語拡張で規格も複数あります。そのためにプログラマは自分のコンパイラがどの規格をサポートしているかを確認する必要があります。重要な C++の規格は、C++98, C++11, C++14, C++17, C++20 です。これらは非公式な名称ですが、それぞれの数字が規格の発表された西暦年の下二桁を示していて、良く使われる名称です。この教科書では、C++17 を対象とします。2020 年時点で多くのコンパイラが標準で C++14 を有効にしていますが、それらはどれも C++17 をサポートしています。そして、数年後にはそのどれもが C++17 に移行するからです。著名な C++コンパイラのサポート状況は、https://cpprefjp.github.io/implementation.html にまとめられているので、確認してみてください。

[2]公開年は C:1972 年, Python:1991 年, Java:1995 年

1　基本データ型と入出力

この章では、C++で扱う基本的なデータ型と入出力の方法についてまとめます。C++の機能をすべて説明しようとすると、辞書の厚さの本が必要となるため、比較的良く使われるものに絞って説明します。

1.1　文字列を出力する例

プログラムを作る大概の目的は、コンピュータからの出力を得ようというものです。出力は文字・音声・映像そしてなんらかの機器の制御信号など様々に考えられますが、文字の出力が最も基本的です。次のプログラムは、Hello! という文字列を出力します。

ソースコード 1: hello プログラム

```
1 // Hello! を出力するプログラム
2 #include <iostream>
3 int main()
4 {
5    std::cout << "Hello!\n";
6    return 0;
7 }
```

プログラム中で // から行末までは**コメント**と呼ばれるプログラムの説明書き (または注釈) です。コメントはコンパイル結果に無関係です。一行におさまらないコメントは、以下のように/*で始まり*/で終わるように書きます。

```
/*
 * 複数行にわたるコメントは
 * このように書きます。
 */
```

2 行目と 3 行目の*は見た目を揃えるためで必要はありませんが、これはよくあるスタイルです。
プログラム例に戻って、2 行目の#include <iostream>は、iostream という名前のファイルを読み込んで前準備を指定しています。「<iostream>**ヘッダファイル**を**インクルード**する」という言い方します。その後の int main() { ... } の部分が、main() 関数と呼ばれる、プログラムの主（メイン）となる処理の指定です。<<は演算子で、"Hello!\n"という文字列を std::cout に書き込むことで出力を指定しています。std::cout は iostream で準備されたもので、文字の出力に使います。通常の出力先はコンピュータ画面ですが、プログラムを実行する際に他に変えることもできます。std は標準 (standard)、cout は character output の省略と考えると良いでしょう。\n は改行文字と呼ばれています。これは他の文字のように出力時に目に見える文字ではありません。画面出力では Hello! という 6 個の文字の後に改行がなされます。return 0; は main() の処理が正常に終了したことを意味します。なお、使用するテキストエディタによっては、\n は￥n と表示されるかもしれません。これは日本のコンピュータの特徴で、歴史的な経緯により、\記号と￥記号が同じものとして扱われていることに起因します。

少し古い教科書やインターネット上で見つかるソースコードでは、using namespace std; という行を 3 行目の include の次の行に書いてあるかもしれません。この方法は std に関して namespace という枠組みを無効にするもので、プログラムの規模が大きくなった際に使用する名前にトラブルを持ち込みます。理由の詳細については”avoid using namespace std”とインターネット検索してみてください。トラブルを避けるためにこの教科書では using ... の 1 行を書かないことにします。

コンパイルと実行の方法は以下のように行います。

```
$ g++ -std=c++17 hello.cpp
$ ./a.out
Hello!
$ g++ -std=c++17 hello.cpp -o hello
$ ./hello
Hello!
```

g++はC++コンパイラの名前で、システムがこの名前を持つプログラムを探してコマンドとして実行します。-std=c++17 は、g++のコマンドオプションで、コンパイラに C++17 をサポートするように指定しています。hello.cpp はソースプログラム（ソースコード）のファイル名です。この指定でコンパイルを行うと a.out という実行ファイルが作られます。生成される実行ファイルの名前を hello としたい場合には、コマンドオプションとして-o hello を指定します。プログラムを実行するには、./a.out のように実行ファイルの前に./をつけるのが間違いが起こらずに良いのですが、環境設定によっては a.out だけでも実行できます。

1.2 変数を用いた例

次の例は整数値を変数に保存する方法の基本を示しています。

ソースコード 2: 変数の利用

```
1  // 変数を使ったプログラム
2  #include <iostream>
3  int main()
4  {
5     // int 型変数の宣言と初期値の指定
6     int x {1};
7     std::cout << x << std::endl;
8
9     // 初期値に他の変数の値を用いた例
10    int y {x + 3};
11    std::cout <<"y = "<< y << std::endl;
12
13    // 代入によって変数の値を変える
14    x = 2;
15    y = x;
16    std::cout <<"x = "<< x
17              <<", y = "<< y << std::endl;
18
19    // 入力値を読み込む例
20    std::cin >> x;
21    std::cout <<"x = " << x << "\n";
22    std::cin >> x >> y;
23    std::cout <<"("<<x<<","<<y<<")\n"; // (x,y)の形式で出力
24    return 0;
25 }
```

main() 関数の最初の int x {1}; という指定によって、整数値を保存するための変数 x の使用を宣言し、その初期値を 1 とします。これらを**変数宣言**とその**初期化**と呼びます。x と{1}の間のスペースはなくても構いません。変数 x の値は std::cout << x << std::endl; によって、改行付きで出力されます。std::endl(end of line) は前の例の"\n"指定と同じく改行を意味します。std::endl と"\n"の指定に違いはあるのですが、はじめのうちは気にする必要はありません (詳細は付録のマニピュレータを見てください)。

変数 x と同様に、変数宣言 int y {x + 3}; によって、変数 y の使用を宣言し、その初期値を x+3, すなわち 4 とします。変数は必要になった時点でいつでも変数宣言で使えるようにできます。変数には必ず値が必要なので、変数宣言する際にはできる限り初期値を与えましょう。

次の std::cout の指定は文字列と変数値を続けて書き込む例です。<<で区切って並べてください。これも間のスペースは任意です。見やすさを考慮してスペースを入れると良いです。

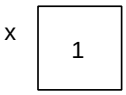
続く、x=2; や y=x; は、**代入**と呼びます。これは既に利用している変数の値を変えるための指定です。std::cin >> x; という指定も変数の値を変えますが、これは外部からの入力を変数 x に読み込むことで値を変更します。どこから入力するかはプログラムを実行する際に決まります。何も指定しなければキーボードからの入力です。

このプログラムの実行例を以下に示します。

```
1
y = 4
x = 2, y = 2
5                          <<--- これをキーボードから入力
x = 5
3 2                        <<--- これをキーボードから入力
(3,2)
```

1.3 変数と変数宣言

変数は整数や実数の値を保存するための「**オブジェクト** (もの)」です。C++では変数とオブジェクトという言葉を、同義語として用いるのでやや混乱するのですが、値を格納するための入れ物と考えると良いでしょう。例えば、整数を保存するための変数 x の値が 1 であるとき、次の図ように、x という名前タグのついた入れ物に値 1 が格納されていると考えることができます。



入れ物には 1 個の値のみを格納できるものや、決められた複数個の値を格納できるものがあります。何が格納できるかは変数の**型**によって定まるので、その型を明確にするためにも変数の使用にはまず変数宣言が必要です。変数宣言は次の形式で指定します。

```
型名 変数名;
または
型名 変数名 1, 変数名 2, 変数名 3;
```

まず変数の種類を表す型名、次に宣言したい変数の名前を続け、そしてセミコロン (;) と続けます。複数個の同じ型の変数を一度に宣言する場合には、カンマ文字 (,) で区切って名前を並べます。

```
int x;        // 一つの変数を宣言
int a, b, c;  // 複数まとめて宣言
```

int は整数型を表します。宣言と同時に初期値が指定できる場合には次のような形式で指定します。

```
int x {1};              // C++11形式
int a{0}, b{1}, c{2};   // C++11形式
int y = 1;              // C++98とCの形式
```

さらに、ゼロで初期化する場合には、数字の 0 を省略しても良いことになっています。

```
 int a{}, b{}, c{};
```

なお、これらの指定で中括弧 ({、})、イコール (=)、カンマ (,)、セミコロン (;) といった記号文字の前後のスペースは任意です。このテキストでは単一の変数宣言の場合にはスペースをいれて、複数まとめて宣言する場合にはまとまりを示すためにスペースを除くことにします。

変数宣言を行う時点で初期値が決められるときには、可能な限りこれらの方法で初期化を行いましょう。初期値の指定はとても重要です。C++の元となった C 言語では宣言時に初期値を指定しないある種の変数（局所変数）は、でたらめな数値のままで良いという方針で設計されました。これは余計なことをしてプログラムを遅くしないためです。C++でも互換性のために組み込み型の変数には同様の措置がとられます。しかし、変数に初期値を指定しないことでプログラムが問題を抱える例はこれまでにたくさんありました。後から考えれば、初期化なしで良いという指定にすれば良かったのかもしれませんが、そうはなっていません。結果として、変数は宣言時に何らかの初期値を指定して、それが無駄と判断される場合にはその指定を外すことを習慣とするのが良いようです。

変数宣言の書き方が 2 種類ある点はやや注意が必要です。C++11 形式と C++98 形式のどちらを使っても良いのですが、C++11 形式は型チェックが厳しいために、次のような違いがでます。

```
int x {1.5};  // エラーになる
int x = 1.5;  // エラーにならない
```

これは int 型の変数にそのままでは保存できない実数を初期値として指定している例です。C++98 形式では、これをエラーとはせずに、単に小数点以下を切り捨てて変数を初期化します。これは C 言語との互換性のためです。後で例を見ますが、C++11 形式でエラーや警告がでる場合には、それなりに理由があります。単にメッセージを消す目的で C++98 形式を使うのはよくありません。

1.4 基本データ型と組み込み型

すべての変数は格納する値の種類を示す型を事前に定めなければなりません。使用できる型には言語の中核として元々用意されている**基本データ型** と後から定義して加える**ユーザ定義型**があります。主に使われる基本データ型は以下の 4 種です。なお、bool・char・int をまとめて**整数系データ型**と呼びます。

型名	扱う内容
bool	論理値である true と false の 2 種類の値
char	英数記号の 1 文字 (0 から 127 の整数)
int	整数
double	実数 (浮動小数点数)

bool 型は 2 種類のものを区別する論理演算 (またはブール演算) 用のデータ型です。C 言語では長らく 0 を真理値の偽 (false)、それ以外の値を真 (true) として処理をしていました。しかし、プログラムの中で論理演算と整数の処理を混在させないために、この bool 型が C++で追加され、C 言語でも後からサポートされました。

char 型は基本的に文字処理のためのデータ型です。コンピュータの中で英数文字は、比較的値の小さい整数で表されているために、この型は文字を表す名前が付いていますが、127 以下の整数を表すデータ型としても利用できます。負数や 128 以上の整数も状況に応じて扱えますが、それらが必要な場合には int を使います。

int 型は正数・0・負数の扱いを想定しています。プログラムの内容によってマイナスを扱わない、つまり、非負の整数を扱いたい場合には、unsigned(符号なしの意味) を指定します。これには char 型も含まれていて、unsigned char や unsigned int と指定すると、0 以上の数値のみを扱うようになります。この unsigned の指定により非負数の最大値がほぼ 2 倍になることも指定する理由の一つです。int 型の場合には、unsigned とは別に、short, long, long long の指定をさらに加えられます。これは標準とは異なるビット数で整数を表すためのものです。

実数を扱うための通常の型は double です。これは倍精度浮動小数点数の”倍”という言葉に由来します。この型も int などと同様に変種があり、異なる有効桁数や指数の範囲を持つ実数型として、float 型と long double 型が用意されています

int や double がどの範囲の数まで扱えるかは、C++の規格で決まっていません。実は使用するコンパイラが決めることになっています。現代の PC の C++コンパイラでは、int が 32 ビット ($-2,147,483,648 \cdots 2,147,483,647$)、double が 64 ビット (10 進数の有効桁数が約 15 桁, およその範囲が $\pm 2.22 \times 10^{-308} \cdots \pm 1.8 \times 10^{308}$) を想定できます。唯一問題となるのはそれらのサイズの表現です。コンピュータやコンパイラの種類によってデータ型のサイズが変わるので、異なる環境でも最大となるサイズ情報を保存できないと困る場合があります。それを解決するために size_t という型の別名 (alias) が用意されています。これは基本データ型ではないのですが、コンパイラが扱うどんなサイズ情報でも格納できる非負の整数型の名前として使えます。この型名の使い方については後で説明していきます。

基本データ型とそれらの配列やポインタなどを総称して**組み込み型**と呼びます。組み込み型はC言語から引き継いだもので、コンピュータの基本機能をそのまま利用するものです。組み込み型の使用は、実行性能の面やメモリ使用率の点で良いのですが、使用方法を誤るとプログラムに脆弱性を持ち込む場合が出てきます。例えば、前述の初期値のなしの変数宣言です。初期値を指定しない変数は状況によってランダムな値が初期値と与えられてしまい、さらに使い方を誤ればプログラムは暴走します。その他にも単純な注意では回避できない場合や使い勝手の悪い部分もあります。そのため、安全面や便利さを考慮して、組み込み型といくつかのプログラムコードを組み合わせた型であるstringやvectorがユーザ定義型として事前に用意されています。これらの型の変数は明示的に指定しなくても妥当な初期値が与えられるようになっています。

1.5 リテラル

リテラル (literal) とは「文字どおりの」という意味で、何らかの値をプログラム中でそのまま表すものです。特に、変数に値を設定したり、変数に格納された値と比べるために使用します。基本データ型のリテラルの例を次に示します。

名前	例
bool リテラル	true false
文字リテラル	'a' 'Z' ' ' '$' '%'
整数リテラル	123 1 0 +456 -567
浮動小数点数リテラル	3.1415 1.0 0.0 1.0e-5

bool型はtrueかfalseの2種類のみの値を扱うデータ型なので、これ以外のリテラル表現はありません。文字リテラルは単一引用符で指定したい英数文字や記号文字を囲みます。ただし指定は1バイトで表せる文字だけで、複数バイト文字のひらがなや漢字はそのままでは指定できません。

エスケープシーケンス

制御文字などの表示できない特殊な文字と一部の記号はエスケープシーケンスを使って表します。これは、バックスラッシュ(\) または環境によっては円マーク (¥) を前置きすることで、その後ろにくる文字リテラルの意味を別のものに変えるものです。文字リテラルでよく使われるエスケープシーケンスには次のものがあります。

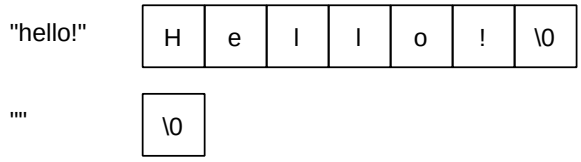
```
'\n'    // 改行を表す文字 (new line)
'\t'    // 水平タブを表す文字 (tab)
'\0'    // char 型の値 0, ヌル文字と呼ばれる
'\''    // 「'」という文字
'\"'    // 「"」という文字
'\\'    // 「\」という文字
```

改行文字・タブ文字・ヌル文字用のエスケープシーケンスは見えない文字を表すためのものです。引用符 (' や") とバックスラッシュのエスケープシーケンスは、それらの文字がリテラルやエスケープシーケンスの区切り文字であるために必要となる表現です。

文字列リテラル

文字列リテラルは、2つの二重引用符 (") で囲まれた文字の並びです。この指定によって、二重引用符で囲まれた内側の文字の並びと、その後ろに**ヌル文字** ('\0') が付加されたデータが作られます。ヌル文字は文字列の終わりを示すために使われます。したがって、"hello!" は下図のように7文字を意味します。また特殊な例として、""はヌル文字だけからなる文字列リテラルです。文字列リテラルが他の整数

系データ型のリテラルと大きく異なるのは、使用によりそれを格納するための場所がメモリ中に確保される点です。これは変数 (またはオブジェクト) と似ていますが、個々の文字は変更できず確保された場所に名前も与えられません。文字列リテラルは char 型の配列で string 型やポインタと関係します。なお、ひらがなや漢字の文字は複数バイトで表現されるので文字列リテラルで表しても問題ありません。



n 進数の整数リテラル

プログラムが実行される際には、整数データは内部で 2 進数を基本としたデータ表現で処理されます。一方、テキスト形式の C++のプログラム中では、数字は文字表現なので、整数は 10 進数以外の 16 進数・8 進数・2 進数でも指定できます。

```
1234              // 10 進数
0x4d2             // 16 進数
02322             // 8 進数
0b010011010010    // 2 進数, C++14
```

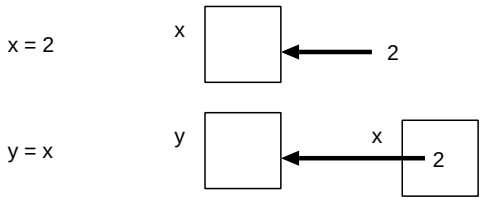
プログラム中で 10 進数の次に良く使われるのが 16 進数の記法です。これは 0x または 0X を値の前に置き、値の部分に 0〜9 と a〜f の文字を並べた書き方です。a〜f の英字は大文字の A〜F でも構いません。0x の x は hexadecimal という 16 進数を表す英単語の発音から生じた省略文字です。8 進数の場合には、0 が値の前に置かれ、値の部分が、0〜7 の数字の並びになります。0（ゼロ）で始まるのは octal という 8 進数を表す英単語の先頭文字の O（オー）を想像させるものと思われますが、定かではありません。2 進数は、C++14 で導入された記法で、0b または 0B が値の前に置かれます。値の部分は、0 1 の並びです。0b の b は binary の頭文字です。

桁数の多い整数は間違いがないかを確認するのが大変です。日本では 3 桁ごとにカンマ (,) を数字に入れて、読むときには万億兆京と 4 桁ごとにまとめた表現を使います。何桁ごとにまとめるか、記号に何を使うは国と文化によって異なります。C++14 以降では整数リテラルに、単一引用符 (') を桁の区切りに使用して、どの桁でまとめても良いことになっています。以下に例を示します。

```
1'234'567                          // 10 進数
0x0012'd687                        // 16 進数
0455'3207                          // 8 進数
0b0001'0010'1101'0110'1000'0111 // 2 進数
```

1.6 代入

代入は、変数の値を変更する式です。例えば、int 型の変数 x に対して、x = 2 と書くと、x の値が 2 となります。その後に、y = x と書くと、y の値が x と同じ 2 となります。これを図で示すと次のようになります。



この図が示すように、=演算子の左側は値を入れる場所が重要です。その場所に元々あった値は上書きされてしまうので何であっても構いません。反対に、右側は読み出す値が重要であって、変数に保存されているかどうかはあまり重要ではありません。このような特徴から、値をいれる場所が重要となるオペランド (演算数) を `lvalue` (location value, 左辺値) と呼び、読み出す値が重要となるオペランドを `rvalue` (read value, 右辺値) と呼びます。例えば、リテラルは `rvalue` です。右辺値、左辺値という言葉が示すように、この用語には、=演算子の左側 (left hand) と右側 (right hand) という意味もあります。

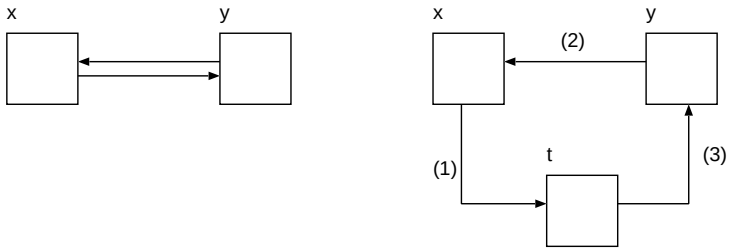
直感的ではないのですが、代入を表す=演算子は、+-などの計算をするための演算子と同様に結果を持ちます。その結果とは代入した値そのものです。これを利用すると、複数の変数に同一の値を代入する操作を一つの式で書けます。例えば、変数 `a` と `b` が `double` 型の変数であったときに、以下の書き方で、一度に二つの変数に値を代入できます。

```
b = a = 1.5;
```

=演算子は**右結合** という特徴を持つので、二つの=が一つの式にある場合には右側が優先です。この例は `b = (a=1.5);` を意味します。まず、`a=1.5` の代入を行い、その演算結果である `1.5` を使って `b` に代入を行います。

値の入れ替え

データ処理において二つの変数の値の入れ替えは頻繁に行われます。次の図を見てください。例えば図の左側のように `x`, `y` のお互いの変数の値の入れ換えを考えます。値の設定は代入ですが、代入は一つずつ行わねばならず、例えば、`x = y;` を先に行えばその代入によって元の `x` の値が失われます。そこで、図の右側のように第 3 の一時的な変数を用意して、(1), (2), (3) の順番に代入を行う必要があります。この方法だと `x` の値は `t` に退避してあるので失われることはありません。



次のプログラムはこれを考慮して二つの変数の値を入れ替える基本的な方法を示しています。

ソースコード 3: 値の入れ替え

```
1  // 二つの変数の値を入れ替える
2  #include <iostream>
3  int main()
4  {
5      // 最初の状態
6      int x{3}, y{5};
7      std::cout << x <<", "<< y <<"\n";
8
9      // 値の入れ替えを行う
10     int t {x};
11     x = y;
12     y = t;
13
14     // 結果を出力
15     std::cout << x <<", "<< y <<"\n";
16     return 0;
17 }
```

変更できない変数

プログラムを修正する際に、変数の値を適当に変更していると、あとで何が起こっているか分からなくなる場合があります。変数の中には値を変えない方が良いものもあるので、変更不可という指定ができると便利です。次の例は整数変数を初期値とともに宣言していますが、内容を変更できないように指定しています。

```
const int total_count {349};
```

const の指定は constant の省略をした予約語で、これを宣言時に指定したことにより、total_count 変数は変更できなくなります。変更できない変数なので宣言時の初期値は必須です。そして、この変数に代入しようとするとコンパイルエラーとなります。const 指定はプログラムの結果に影響を与えるものではありませんが、実行を速くする場合があります。しかし、その理由を知るには他にいろいろな知識が必要となるので、まずはプログラマの考えを明確にするものと思って使うようにしましょう。

1.7 基本入出力

コンピュータの内部では整数や実数は 2 進数を基本としたデータ表現で保存されていますが、プログラムからの出力は、人間が見たり、違う種類のコンピュータが入力に使います。そのため、内部のデータ表現を出力には使わずに、文字コードとして出力するのが基本です。文字コードは、例えば 160 ページにある ASCII コードの番号です。文字コードが画面に出力されれば対応する図形としての文字が表示されるので、人間は文字コードが表す文字を読むことができます。つまり、プログラムの外は文字 (character) で、内部では 2 進数を基本としたデータ表現であるために、変換が必要となるのです。std::cin と std::cout の c は文字 (character) を意味しており、その変換の役割を果たします。

std::cout による出力

std::cout << x; の形式で指定されると変数の値が文字コードとして出力されます。例えば、変数 x が int 型で値が 12 の場合には、'1' という文字と'2' という文字のコードが並んで出力されます。そのため、std::cout <<"x = "<< x; という指定によって、文字列リテラルで指定された文字の並びと、変数 x の値から変換された文字の並びのコードが出力されます (出力には文字列リテラルの最後のヌル文字は含まれません)。整数値の出力は何も指定しなければ 10 進数です。16 進数や 8 進数で出力したい場合には次のようにします。

```
int x{1234567}, y{0x0012d687}, z{004553207};
std::cout << std::showbase  // 16進数の0xや8進数の0を表示
          << std::hex << x <<"\t"<< y <<"\t"<< z <<"\n"
          << std::oct << x <<"\t"<< y <<"\t"<< z <<"\n"
          << std::dec << x <<"\t"<< y <<"\t"<< z <<"\n";
```

std::showbase はコメントにあるように、それ以降の出力で 16 進数の 0x や 8 進数の 0 が現れます。std::hex, std::oct, std::dec はそれぞれ指定以降の出力を 16 進数, 8 進数, 10 進数に変更します。一度指定すると、次に変更されるまで指定が有効です。

2 進数の出力は使用される頻度が少ないために、std::hex のような簡易な指定がサポートされていません。bitset と呼ばれる別のプログラムの力を借りることになります。<bitset>というヘッダファイルインクルードして、以下のように指定します。ここで std::bitset<32>(x) の 32 はビット数を表します。

ソースコード 4: 2 進数での出力

```
1 // 2進数での出力
2 #include <iostream>
```

```cpp
#include <bitset>
int main()
{
   int x {1234567};
   std::cout << std::bitset<32>( x ) <<"\n";
   return 0;
}
```

実数の出力には様々な指定方法がありますが、デフォルトの指定以外で時おり必要となるのは小数点以下の桁数の指定です。それには次のように指定します。

ソースコード 5: 小数点以下の桁数を指定した出力

```cpp
// 小数点以下第 3位までの出力
#include <iostream>
#include <iomanip>
int main()
{
   double x {3.1415};
   std::cout << x <<"\n"
             << std::fixed << std::setprecision(3) << x <<"\n";
   return 0;
}
```

出力の 1 行目は 3.1415、2 行目は小数点以下 3 桁を指定しているので 3.142 です。std::setprecision() を指定するには<iomanip>ヘッダファイルのインクルードが必要です。

bool 型の値を出力する機会はそれほどないのですが、プログラムのデバッグ（間違いを探して直す作業）では必要かもしれません。bool 型の値は true と false という 2 種類の単語で表されます。そころがその出力の方はデフォルトで 1 と 0 です。これは C 言語が論理式に整数を使っていたためで、C++の出力もその影響を受けたと考えられます。そのため true と false を出力するには std::boolalpha の指定が必要です。この指定は一度行えば以降も有効となります。

```cpp
bool x{true}, y{false};
std::cout << x <<" "<< y <<"\n"    // 1 0
          << std::boolalpha        // アルファベットでbool値を出力
          << x <<" "<< y <<"\n";   // true false
```

std::cin による入力

入力では、指定された変数型の内部表現に文字の並びを変換しなければなりません。この変換は常に成功するとは限らず失敗もあります。例えば std::cin >> x の x が int 型の場合には、"12"のような数字の並びは数値の 12 に変換されます。しかし、abc という文字列の入力は int 型の値に変換できません。つまり変換失敗です。変換に失敗した場合には変数 x は変更されないので、その先の処理をどうするか考えなければなりません。また、入力された文字列中に現れるスペース文字 (' ') は、前後の文字を分ける意味はありますが、変換の対象ではなく、読み飛ばされることになっています。これらについての詳細は後の章で調べることにします。

10 進数と実数の値は std::cin >> x; のように対応する型の変数への入力を指定すればよいことになっています。bool 型は整数を表す文字列を読み込んで、0 に変換できれば false、それ以外は true です。英単語による bool 型の true と false, 16 進数, 8 進数の値の入力は、std::cout への指定と同じく追加の指定が必要です。次のプログラムを実行して、いろいろな入力で試してみましょう。

```cpp
bool b {false};
std::cin  >> std::boolalpha >> b;
std::cout << std::boolalpha << b <<"\n";
int h {0};
std::cin  >> std::hex >> h;
std::cout << std::hex << h <<"\n";
int o {0};
```

```
std::cin  >> std::oct >> o;
std::cout << std::oct << o <<"\n";
```

このプログラムは、true/false, 16 進数の数値, 8 進数の数値を表す 3 種の文字の並びを入力として期待しています。16 進数と 8 進数は、0x などの接頭辞をつけてもよいです。ただし、途中の入力で不正な文字列を読み込めば、残りの入力はすべて失敗します。実用的なプログラムでは入力の失敗への対処も指定しておく必要があります。これについても後の章で学んでいきます。

1.8 算術演算のための演算子

算術演算は+-*/%といった 2 つの項を指定する演算子を指定して行います。これらはどれもコンピュータ上では通常一つの機械語の実行で計算されるものです。そのため累乗の演算子はありません。累乗の計算が必要な場合には後に挙げるライブラリ関数を利用します。

演算子	操作
+	加算
-	減算
*	乗算
/	除算
%	剰余算（あまり）

これらの演算子は二つのオペランドをとるので、**二項演算子**と呼ばれます。一つのオペランドをとる演算子は、**単項演算子**です。このことからプラスマイナスは、+x, -y のように使われれば単項演算子で、2+3 のように使われれば二項演算子です。

複数の演算をまとめて指定する場合には、基本的な算術計算と同じく、*/による演算が+-による演算よりも優先されます。つまり、1+2*3 は 2*3 が先に行われます。また、これらの演算子は、**左結合**なので、*/の優先を除けば左側から計算を行います。例えば、5-4-1 は 5-4 を先に行います。単純な式の場合には問題ありませんが、3*-2+6/-2 のように単項演算が混在したり、8.5/4.3/2.1 のように除算が続くなど混乱をまねきそうな場合には、3*(-2) + 6/(-2) や (8.5/4.3)/2.1 のように括弧をつけた式にするのが良いです。

割り算に関係する演算は注意が必要です。10/3 のような除算を行った場合に、int 型では結果の小数点以下の値は切り捨てられます。double では小数点以下の値を保持しますが、精度に限界があるので、数学で定義されている計算結果とは異なる場合があります。また剰余は int 型どうし値の演算に使われます。そして、何よりも重要なのは 0 で割るような演算が実行されると、そのプログラムは異常終了の形で停止する点です。プログラムの中でチェックを行い、ゼロ除算とかゼロ割と呼ばれるこの状況は避けなければなりません。

int と double の組み合わせ

プログラムで整数と実数を組み合わせて使う場合には、さらに注意が必要です。コンピュータのハードウエアは整数の演算と実数の演算はまったく別のものです。計算はどちらかで行わなければなりません。そのため C++では、int 型の値と double 型の値を組み合わせた計算を指定すると、int 型の値を double 型に変換してから計算を行います。例を見てみましょう。次のプログラムは整数と実数を組み合わせた割り算の結果を出力するものです。

ソースコード 6: int と double の混在

```
1 // int と double が混在した場合の演算
2 #include <iostream>
3 int main()
```

```
4  {
5     // 値の組み合わせの違いによる割り算結果の確認
6     std::cout << 10/3 <<" "<< 10.0/3 <<" "<< 10/3.0 <<" "<< 10.0/3.0 <<"\n";
7
8     int x{10}, y{3};
9     double dx {x}; // 値としては問題はないが、C++11形式では警告がでる
10    double dy = y; // C++98形式では警告がでない
11    dy = y; // 代入でも警告はでない
12
13    // 変数の組み合わせの違いによる割り算結果の確認
14    std::cout << x/y <<" "<< dx/y <<" "<< x/dy <<" "<< dx/dy <<"\n";
15
16    // double の初期値に int の割り算を使う
17    // 小数点以下を切り捨てた値が初期値となる
18    double dz {x/y}; // C++11形式では警告がでる
19    std::cout << dz <<"\n";
20    return 0;
21 }
```

このプログラムの実行結果は次のようになります。

```
3 3.33333 3.33333 3.33333
3 3.33333 3.33333 3.33333
3
```

まず、整数どうしの割り算の結果は割り切れない場合でも整数となることに注意してください。結果の小数点以下の値は切り捨てです。整数と実数が混ざった割り算では、除数と被除数のどちらが整数の場合でも、結果は実数となります。したがって、平均値を計算するような計算において、合計値を int 型で計算した場合には、この例の dx のように、一度、double 型の変数に値を代入してから割り算を行うのが良いでしょう。

実数型の変数宣言の初期値に整数を指定したり、その型の変数に整数を代入しても、多くの場合は問題ありません。しかしまったく問題がないわけでもありません。問題が起こるのは、整数の値が非常に大きくかつ何桁も 0 以外の値が下位側の桁に続く場合です。実数型の変数は有効桁数に制限があるため、そのような大きな整数を初期値とすると値が変わってしまいます。32 ビットの int 型と 64 ビットの double 型ならば大丈夫ですが、これ以外の組み合わせでは問題となる可能性があります。逆に、整数型の初期値や代入に実数を用いると、小数点以下が切り捨てられてしまいます。さらに実数値が対象となる整数型の表せる範囲を越えていれば誤った変換が起こります。

これらはコンパイラによる暗黙の型変換と呼ばれる処理です。型変換は仕方がないとしても暗黙のうちに変換されるのはトラブルの元です。何か問題が起こった場合でもその問題の場所の特定が難しいからです。例えば、上記 19 行目の結果が値の切り捨てを期待していなかったら、どこが問題かがすぐに分からないかもしれません。そのため C++11 の初期化指定ではこれに警告を出すようになっています。コンパイラに警告を出させないためには、プログラム中で明示的に型変換を指定します。

```
double dx{static_cast<double>(x)}; // 初期値の場合
double dy {};
dy = static_cast<double>(y); // 代入の場合
double dz {static_cast<double>(x)/y}; // 切り捨てを避ける
```

static_cast<変換後の型>(変数) は型変換が目立つようにわざわざ長ったらしい表現になっています。これよりテキストエディタで容易に型変換を行っている場所を探せます。C++は元となった C 言語が暗黙の型変換を採用していたために、それと同じ動作となっています。しかし、暗黙の型変換は問題が多いので、このように後から警告を出すようになりました。最近の新しいプログラミング言語では暗黙の型変換を行わないものも現れています。

代入演算子

プログラム中の変数は更新されますが、`point = point * 2;` のように現在の変数の値に基づいて、新しい値を計算することがよくあります。この場合に、同じ変数名を二度入力するのは面倒ですし、同じ変数名が冗長に並ぶので見間違えることがあります。その場合には**代入演算子**が有用です。これは次のように省略した書き方ができます。

```
point *= 2;   // ポイント2倍
x *= y + 1;   // y+1倍
```

二つ目の例は、`x = x*(y+1);` と同じ結果です。良く使用される代入演算子には以下があります。

演算子	操作
+=	加算代入
-=	減算代入
*=	乗算代入
/=	除算代入
%=	剰余算代入

増分と減分の演算子

代入演算子には任意の値を使って変数を更新していましたが、プログラムを書いていると、+1 や-1 で更新する場合が頻繁に出てきます。その場合に以下のように書けます。

```
x++;   // x += 1
y--;   // y -= 1
```

++演算子は**増分演算子**または**インクリメント演算子**と呼びます。--は**減分演算子**または**デクリメント演算子**です。どちらの演算子もオペランドに左辺値が必要です。この演算子は、変数の前側に指定する前置きと、後ろ側に指定する後ろ置きの二種類があります。演算子を単独で使う分には、前置きと後ろ置きに違いはないのですが、=演算子同様に演算結果を持つので、その結果を使う場合には注意が必要です。次の例を見てみましょう。

```
int x{}, y{}, z{};
z = x++;                  // z=x; x+=1;
std::cout << z <<"\n";    // 0
z = ++y;                  // y+=1; z=y;
std::cout << z <<"\n";    // 1
```

x と y はどちらも 0 から 1 に増えますが、出力は一つ目が 0 で二つ目が 1 です。これは後ろ置きの++演算子が更新前の変数の値を式の値として、後から更新を行うのに対して、前置きの場合には先に更新を行って、その結果を式の値とするためです。++と--の演算子は簡略のための表記ですが、このように前置きと後ろ置きで微妙に異なるので、複雑な式では使わないようにしましょう。

1.9 数学用の定数と関数

<cmath>ヘッダファイルをインクルードすると、数学に関係する重要な定数とよく使用する関数を利用できるようになります。よく使われる数値演算定数には以下のものがあります。

シンボル	意味	値
M_E	ネイピア数 e	2.7182818284590452354
M_PI	円周率 π	3.14159265358979323846
M_SQRT2	$\sqrt{2}$	1.41421356237309504880

これらの定数はC言語では昔からよく使われるのですが、C++では標準として定義されていないのでstd::が付きません。g++では<cmath>をインクルードするだけで使用できますが、その他のコンパイラではソースコードの先頭に以下を書く必要があるかもしれません[3]。

```
#define _USE_MATH_DEFINES
```

次によく使われる数学関数を示します。これらは標準化されているのでstd::が付きます。

関数	意味
std::abs(arg)	argの絶対値を返す
std::ceil(arg)	天井関数：argよりも小さくなくもっとも近い整数を返す
std::floor(arg)	床関数：argよりも大きくなくもっとも近い整数を返す
std::round(arg)	argにもっとも近い整数を返す
std::exp(arg)	自然対数の底eのarg乗(e^{arg})を返す
std::log(arg)	argの自然対数を返す
std::log10(arg)	argの常用対数（底が10の対数）を返す
std::pow(arg1, arg2)	arg1のarg2乗($arg1^{arg2}$)を返す
std::sqrt(arg)	argの平方根のうち負でない方を返す
std::sin(arg)	arg（ラジアン）の正弦（サイン）の値を返す
std::cos(arg)	arg（ラジアン）の余弦（コサイン）の値を返す
std::tan(arg)	arg（ラジアン）の正接（タンジェント）の値を返す
std::asin(arg)	argの逆正弦を$[-\pi/2, \pi]$の範囲のラジアン単位の値として返す
std::acos(arg)	argの逆余弦を$[-\pi/2, \pi]$の範囲のラジアン単位の値として返す
std::atan(arg)	argの逆正接を$[-\pi/2, \pi]$の範囲のラジアン単位の値として返す

これらの関数の引数の型は、abs関数を除き、どれも実数型です。実数のならばdoubleまたはfloatのどちらでも構いません。abs関数はargが整数の場合には整数が結果となり、実数の場合には実数が結果となります。それ以外の関数の引数としてint型の値を指定すると暗黙の型変換により、double型になります。そのため引数として整数式を指定する際には、以前に説明した「intとdoubleの組み合わせ」で生じる問題を意識する必要があります。

round関数は、基本的にはargにもっとも近い整数を返すのですが、特別な場合として、argが二つの整数の真ん中の値の場合には、0から離れる方向の整数となります。そのため正数の場合には四捨五入の用途として使用できます。同様にして、切り捨てにはceil関数、切り上げにはfloor関数を使います。また、三角関数はすべて角度の単位にラジアンを指定します。そのため、例えば、度数法で60度を指定したい場合には、60*M_PI/180.0などとラジアンに直してから引数に指定する必要があります。

1.10 名前空間とusing宣言とusingディレクティブ

これまでにstd::で始まる名前がたくさんでてきました。stdはstandardの意味で、事前に用意された標準の名前であることを表しています。std::などの::で区切った名前の集まりは、**名前空間**と呼ばれるものを構成しています。これは電話の内線番号のような使い方を意図したもので、指定された名前空間内でプログラムを書く際には::指定なしに名前を使用し、他が関係する場合には::をつけてフルネームを使うというものです。変数や関数などを指定の名前空間に所属させることで、名前の衝突を気にせずに単純な名前を使用できます。

プログラムを書くには自分で変数や関数に名前をつけることになります。事前に用意されている名前と衝突することは避けなければならないので、std名前空間の名前をフルネームで指定するのはある程

[3] C++20になるとstd::numbers:piのように標準化された定数が使えるようです。

度しかたがありません。しかし、std::cout や後に学ぶ std::string などは何度も使用するので、省略して cout や string と書いた方が便利な場合があります。using **宣言**は、::による長い名前を個別に指定して、省略した名前の使用を宣言するものです。以下に例をします。

ソースコード 7: using 宣言の例

```
1  // 簡易乗算プログラム
2  #include <iostream>
3  using std::cin, std::cout; // using 宣言
4  int main()
5  {
6     cout << "input>> ";
7     int x{0}, y{0};
8     cin >> x >> y;
9     cout << x <<"x"<< y <<" = "<< x*y <<"\n";
10    return 0;
11 }
```

using の後に省略して使いたい名前を書きます。C++17 から複数ある場合にはカンマ (,) で区切って並べられるようになりました。例のようにプログラムの最初の方でこの宣言をすればプログラムファイル内のすべてで省略名が使えるようになります。main などの関数の中で宣言すれば、宣言の有効範囲は関数の終わりまでとなります。

using **ディレクティブ**（指示の意味）は、名前空間を指定することでその中に所属する名前をすべて取り込みます。以下に例をします。

ソースコード 8: using 宣言の例

```
1  // 簡易乗算プログラム
2  #include <iostream>
3  using namespace std; // using ディレクティブ
4  int main()
5  {
6     cout << "input>> ";
7     int x{0}, y{0};
8     cin >> x >> y;
9     cout << x <<"x"<< y <<" = "<< x*y <<"\n";
10    return 0;
11 }
```

これは std の名前空間をすべて取り込んだものです。ソースコード 7 とあまり変わりませんが、std のすべての名前が取り込まれるので、このプログラムを今後修正して、min, max, data などの名前を使用する際に名前の衝突について注意しなければなりません。C++17, C++20 と規格が変更されると std に名前が次々と追加されていき、以前にコンパイルできていたプログラムがトラブルに巻き込まれることになるかもしれません。そのためなるべく使用しないようにしましょう。

2　論理式と条件文

プログラムの中では、ある時点で得られている値に応じて、その次にすべきことを判断をします。判断は、大きいかそうでないか、良いか悪いかなどの二者択一を基本として行います。プログラムでこれらを表すには、bool 型の値と論理演算を使います。この章では、まず論理式を用いた演算の基本を説明し、その後でその結果を用いた条件文について説明していきます。

2.1　論理式に関係する演算子

論理演算に用いる bool 型は true と false の値のみを用いるデータ型です。変数宣言の初期化では、これら二つの値を指定するのが良いのですが、std::cout と std::cin の説明で見たように、1 と 0 でも初期化が行えます。なお、デフォルトの値は 0 に対応する false です。

```
bool x{true}, y{};  // {}は0を意味して結果としてfalseとなる
std::cout << std::boolalpha << x <<" "<< y <<"\n";
bool a{1}, b{0};  // 1->true, 0->false
std::cout << a <<" "<< b <<"\n";
// bool c{2}; // error
```

まずは bool 型の値を計算するための演算子を見ていきます。

比較演算子

計算結果から何かしらの判断を行う場合、値の大小や等しさを比べることになります。以下の表は**比較演算子**と呼ばれる二つの値を比べるための演算子です。

演算子	意味
<	小なり
<=	小なりイコール
>	大なり
>=	大なりイコール
==	等しい
!=	等しくない

演算結果は bool 値となります。そのため、結果を bool 型の変数に代入でき、更なる判断の計算に使用できます。

```
int i{3}, j{4};
bool x{3 < 4}, y{};   // xの初期値は比較結果
y = i > j;            // yの値は変数の比較結果
std::cout << std::boolalpha << x <<" "<< y <<"\n";
```

整数と同様に実数値も比較ができますが、==と!=の等値比較には注意が必要です。

```
double z = 0.1 + 0.2;
std::cout << std::boolalpha << (z==0.3) <<"\n";
```

このプログラム片の出力はなんと false です。これはコンピュータの実数の扱いが概数であり、正確ではない場合があるためです。したがって、double 型の値に対しては、< <= > >=の四つの演算子を使います。==と!=を dobule 型で使うことは禁止されていませんが、避けるべきです。その代わりとして、等しいことを判断する方法は後で紹介します。

次の例は文字変数の比較です。

```
char a{'m'}, b{'n'};
bool x{}, y{};
x = a == b;
y = a <= b;
std::cout << x <<" "<< y <<"\n";
```

char 型の文字は、コンピュータ内部では ASCII コードに基づいた 127 以下の整数値として表現されています。このコードにおける比較は文字コードの大小さを見ていますが、文字の順序関係（前か後ろか）を判定しているとも言えます。この順序のことを ASCII コード順とか辞書順と呼びます。

論理演算子

ある値が指定した範囲にあるかどうかを判定するには、範囲の上限値と下限値の二つを比べる必要があります。さらに、ある範囲に入っていないことを示すには、論理否定を使うことがあります。それらを行うのが論理演算子である && || ! の三つです。&&演算子は論理積（AND）を表し、A && B と書くと、この式は A と B の式がともに true ときに結果が true であり、それ以外は false です。一方で、|| 演算子は論理和（OR）を表し、A || B と書くと、この式は A または B のどちらかが true ならば結果が true であり、双方とも false ならば式の結果も false です。言い換えると、双方が false のときのみ結果が false です。!は論理否定を表し、true と false が逆転します。次の表を頭にいれておくと良いでしょう。

A	B	A && B	A || B	!A
true	true	**true**	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	**false**	true

次の例は複数条件を指定して値の範囲を判定するプログラムです。

ソースコード 9: 複数条件の指定

```
1  // 複数条件の指定
2  #include <iostream>
3  int main()
4  {
5     int x {3};
6     std::cout << std::boolalpha << x <<":"
7               << (1 <= x && x <= 3) <<" " // 1以上かつ 3以下（1から 3の範囲）
8               << (x <= -50 || x >= 100) <<"\n"; // -50以下または 100以上
9     x = 9;
10    std::cout << x <<":" <<
11         ((4 <= x && x <= 5)||(8 <= x && x <= 10)) <<"\n"; // 2個の範囲のどちらか
12    x = 4;
13    std::cout << x <<":"
14              << !(1 <= x && x <= 3) <<" " // 1から 3の範囲ではない
15              << (x < 1 || 3 < x) <<"\n"; // 1未満または 3を越える
16    return 0;
17 }
```

&&や||の演算子は、<=などの比較演算子よりも結びつきが弱い（優先度が低い）ので、7 行目の 1<=x && x<=3 は、(1<=x) && (x<=3) という意味です。したがってこの例では、二つの条件が同時に成り立ったときに true を出力します。次の x<=-50 || x>=100 は、(x<=-50) || (x>=100) という意味になり、どちらかの条件が成り立ったときに true になります。その次は、&&と||を組み合わせたもので、指定した二つの範囲のどちらかに x の値がある時に true です。&&と||の組み合わせでは、&&の方が||よりも結びつきが強いため、この指定は、4<=x && x<= 5 || 8<=x && x<=10 と書くことができますが、コンパイラによってはお節介にも括弧をつけよと警告します。最後の例は、論理否定を使う

場合と使わない場合の例で同じ結果となります。どちらの書き方を使うかはプログラムで行いたい内容しだいとなります。

大小比較の話をする際は、「以上」・「以下」・「未満」・「越える」を正しく使い分けましょう。普段の生活で厳密なことを扱っていない場合にはそれほど意識する必要はありませんが、プログラムで処理をする話の場合には、これらの言葉はそれぞれ>=・<=・<・>の演算子に対応し、異なる判断結果となるからです。

短絡評価

&&と||による論理演算は、それぞれ二つのオペランドから結果を計算するように指定します。ところが、演算子の左側にある第一オペランドを評価した段階で式全体の結果が定まる場合には、右側の第二オペランドの評価を省略することになっています。これを **短絡評価** と呼びます。例えば、`a < 0 && x <= 3` という式において、もし `a < 0` が成り立たない場合には、x の値に関わらずこの式の結果は `false` と分かります。その場合には `x <= 3` を調べません。論理演算の二つのオペランドが単純な式の場合には、短絡評価によって演算を省略してもしなくてもそれほど違いはありません。しかし、右側の第二オペランドが `std::cin` や `std::cout`、代入の式、増分演算子などのように、それを行うことで他に影響のある（副作用がある）操作の場合には、重要な意味を持ってきます。次の例を見てみましょう。

ソースコード 10: 短絡評価の例 (short-circuit evaluation)

```
1  // 剰余の計算
2  #include <iostream>
3  int main()
4  {
5     int x{}, y{};
6     std::cin >> x >> y;
7     x >= 0 || std::cout <<"x の入力エラー\n";
8     y > 0 || std::cout <<"y の入力エラー\n";
9     y != 0 && std::cout <<"x%y = "<< x%y <<"\n";
10    return 0;
11 }
```

このプログラムは二つの入力値から剰余を計算して結果を出力します。6 行目で x と y の値を入力した後に、7 行目で x の値が 0 以上であるかを確認します。||演算子の短絡評価により x>=0 以上ならばその右側の出力操作は行いません。8 行目も条件は異なりますが同様です。これらはユーザに入力不正を伝えるだけですが 9 行目はもっと大事です。0 による除算や剰余算はプログラムをクラッシュさせるので、避けなければなりません。そこで&&の短絡評価を利用して、y が 0 の場合には x%y の計算を行わないようにしています。&&の短絡評価は||とは条件が逆で、第一オペランドが `true` のときのみ右側の第二オペランドを行います。

入力に関してはもう一つテクニックがあります。`std::cin >> x` による入力は、これ自体が他の演算と同じく結果を持ちます。そして `bool` 型が必要となるところに、この入力指定が使われると、入力自体が成功したか失敗したかを示す `bool` 値が結果となります。したがって、この例の x と y の入力で、数字以外の入力がなされた場合にエラーメッセージを表示させようと思った場合には、次のように書けます。

```
std::cin >> x >> y || std::cout<<"value error\n";
```

この書き方はインタプリタを使うプログラミング言語でよく使われる記法です。短絡評価を使うとちょっとしたチェックでプログラムを暴走させないための策を講じることができます。そのためいくつかのよく使われる書き方があります。後の章でそれらを見ていくことにしましょう。

2.2 条件文

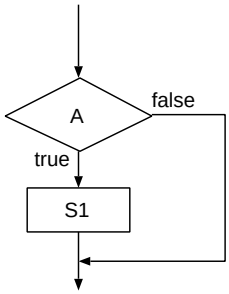
論理演算子の&&や||を使うことで条件に応じた処理を行う例を見てきました。しかし、この方法は万能ではなく限られ場合にしか対応できません。ある条件が満たされたときに行う処理の指定は、**条件文**で行います。この節では条件文の基本について見てみましょう。

if 文と if-else 文

変数の値に応じて処理を変える指定には if **文**や if-else **文** を使います。

if 文は次のような形式です。条件 A のところには bool 型の結果となる式、文 S1 のところに実行文を指定します。条件 A が true の時に文 S1 が実行されます。文 S1 のところには、代入・入力・出力などの他に、if 文や後に説明する for 文・while 文が一つ指定できます。複数の文を指定するには、それらを中括弧 ({と}) でまとめます。中括弧でまとめた部分を**複文**と呼び、全体を 1 個の文として扱います。

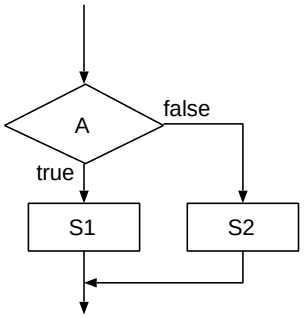
```
if（条件 A）
  文 S1
```



右側のフローチャートで流れを確認してみましょう。矢印に沿って、まず A の bool 式を計算します。結果が true の場合には菱形の下側の矢印に沿って進み、S1 を実行します。false の場合には菱形の右側に進み、S1 を迂回します。その後に合流して次の実行文に進んでいきます。

二つ目の書き方は if-else 文と呼びます。これは if 文に加えて、else 部が指定されており、条件 A が true の時に文 S1 が実行され、条件 A が false の時に文 S2 が実行されます。この場合のフローチャートは菱形の条件 A の右側 false 方向に進んだところに、S2 があるところだけが異なります。なお、文 S1 と文 S2 の指定も複数の処理を指定したい場合には複文とします。

```
if（条件 A）
  文 S1
else
  文 S2
```



実際のプログラムを見てみましょう。次のプログラムは入力で得た実数の値に応じてその値を変化させて、結果として必ず 10 を越える数値を出力します。処理内容にあまり意味はありませんが 0 が入力場合にはエラーを出力します。最初の if 文で変数 x が負の値を持つ場合にそれを正の値に変えます。次の if-else 文では、変数 x が 1 未満の場合には 10.0 を加算し、それ以外の場合には 10 倍します。3 個目の if-else 文では文 S1 や文 S2 の部分を{と}で複文にまとめています。return 1; の実行で main 関数が戻りプログラムが終了します。

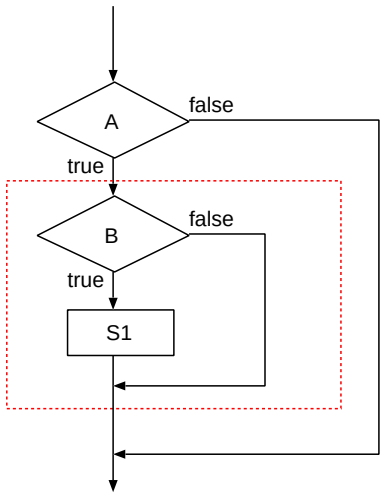
ソースコード 11: if 文/if-else 文の例

```
1  // 実数の入力値を 10以上の値に変換して出力する
2  #include <iostream>
3  int main()
4  {
5     double x {};
6     std::cin >> x;
7
8     if (x < 0.0)
9        x = -x;
10
11    if (x < 1.0)
12       x += 10.0;
13    else
14       x *= 10.0;
15
16    if (x <= 10.0) { // 複数の文をまとめる方法
17       std::cout << "input error";
18       return 1;
19    } else { // 1個の文を複文としても良い
20       std::cout << "Result is "<< x <<"\n";
21    }
22    return 0;
23 }
```

入れ子の if 文

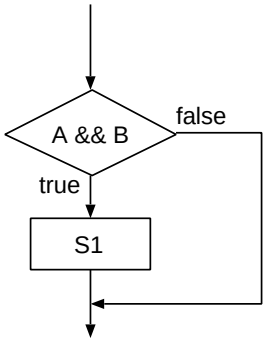
if 文の文 S1 の部分に別の if 文を指定する場合を考えてみましょう。if 文は一つの文なので、中括弧は不要ですが範囲を明確にするためにつけています。条件 B の if 文はこのように字下げ（インデント）つけて内側に配置するのがよく使われる方法です。これは右側のフローチャートと同じような形になります。なお、赤の点線枠は注目してほしいという意味で書いています。フローチャートとしては書かない点線です。

```
if (条件 A) {
  if (条件 B) {
     文 S1
  }
}
```



このプログラムが表す状況をよく考えてみましょう。条件 A が true で、かつ、条件 B も true の際に文 S1 を行います。条件 A が false の場合には、条件 B を調べることもありません。これは以下のように&&演算子の短絡評価を使うと、一つの if 文で表せます。この書き方ならば、混乱もないので複文を表す中括弧も不要かもしれません。

```
if (条件 A && 条件 B) {
    文 S1
}
```



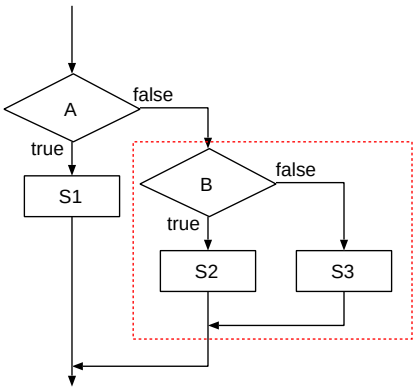
確認

||演算子を if 文の条件とした場合のフローチャートを書いてみましょう。そして、その図から||を使った式を if 文と if-else 文を使って書き直してみましょう。この確認をやっておくと、こんがらがった if 文を整理する力が付きます。

else 部の if-else 文

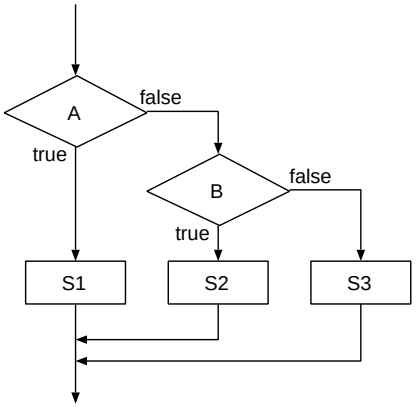
if-else 文が入れ子になる場合には、if 側と else 側およびその両方という三つのパターンが考えられます。その中でも else 側に他の if 文または if-else 文が入れ子になる場合は書き方に注意が必要です。例を見てみましょう。

```
if (条件 A) {
    文 S1
} else {
    if (条件 B) {
        文 S2
    } else {
        文 S3
    }
}
```



この書き方は else 部の中に if-else 文を書いています。フローチャートをみると、文 S1 と条件 B の if-else 文が並んでいるので、プログラムでもインデントをつけて S1 と if-else 文が並ぶようにしています。この処理は別の見方をすると、条件 A と条件 B によって文 S1、S2、S3 を選んでいるとも見ることができます。それが次の書き方です。

```
if (条件 A) {
    文 S1
} else if (条件 B) {
    文 S2
} else {
    文 S3
}
```



今回のフローチャートは S1 の位置が変わり、S3 からでる矢印のつながり先も少し変わりましたが、前のものと同じ内容の処理です。プログラムの方は、条件 A の if-else 文の else 部の中括弧を省略しただけですが、else の直後に二つ目の if を書いているので、大きく変わったように見えます。

どちらの書き方を採用するかは、プログラムの内容しだいです。どちらが良いとか悪いとかということはありません。ただし、前者の方法は、文 S2 や S3 のインデントが深くなっているので、これを続けるとプログラムが右側に膨らんでいきます。Python のような他の言語では elif 文があり、後者の例の形を表現します。C++でも非公式に else-if 文と呼ぶことがあります。

bool 型の定数比較

bool 型変数の取りうる値は true または false の 2 種類です。そこで、if (x == true) や if (x == false) などと書きたくなります。しかし、式 x == true の結果はやはり true または false なので、この書き方は冗長です。if (x) や if (!x) という形で書きましょう。

```
bool x{true};
if (x == true) std::cout << "redundant";  // 冗長な書き方
if (x) std::cout << "simple";             // 簡潔な書き方
```

ぶらさがり else 問題に対する注意

if 文は else 部を指定してもしなくても良いために、複数の if 文や if-else 文を組み合わせた場合に else 部がどの if に対応するかが分かりづらくなります。文法としては else は一番近い if に対応することになっていますが、プログラム中のインデントを間違えると、思った内容とは異なる指定になっていることがあるので注意が必要です。

次の例はこのぶらさがり else 問題を確認するプログラムです。コードの中の a) は誤ったインデントをしています。b) のように複文を使うと if と else の対応関係をはっきりさせられます。さらに、b) の二つ目の if 文や else 部にも中括弧をつけて完全に曖昧さをなくす方法もあるでしょう。中括弧による複文の使用はうっかり間違えをなくす効果があるので良いです。ただし、中括弧だらけになると改行が増えたり、コードが雑然となることがあり、読みやすさを低下させることになります。そのために、文章を書くのと同様に、読みやすさと正確さの双方に気をつけてプログラムを書くようにしましょう。Python が if 文などで中括弧を使わない理由を考えてみるのも、読みやすいコードを書く際の判断につながるかもしれません。

ソースコード 12: ぶら下がり else 問題の例

```
1  // ぶら下がりelse
2  #include <iostream>
3  int main()
4  {
5      int x {};
6      std::cin >> x;
7
8      // a) 字下げを間違えて記述してもコンパイルはできる
9      if (x <= 10)
10        if (x == 10) std::cout << "Equal to 10";
11     else // このelse は if(x==10)に対応する
12        std::cout << "Larger than 10?";
13
14     std::cout << "\n--------------------\n";
15
16     // b) これとa)は同じではない。
17     if (x <= 10) {
18        if (x == 10) std::cout << "Equal to 10\n";
19     } else
20        std::cout << "Larger than 10\n";
21     return 0;
22 }
```

いくつかの入力に対する実行結果を示します。9 や 11 を入力した際に、a) の出力がおかしくなっていることが確認できます。

```
$ ./a.out
9                   <<-- これを入力
Larger than 10?
--------------------
$ ./a.out
10                   <<-- これを入力
Equal to 10
--------------------
Equal to 10
$ ./a.out
11                   <<-- これを入力

--------------------
Larger than 10
```

三項演算子

プログラムを書いていると、ある条件に応じて、二つのどちらかの値を変数に代入したいことがよく出てきます。次の例を見てみましょう。

```
int x{}, a{};
std::cin >> a;
if (a < 0) {
  x = -a * 10;
} else {
  x = a * 20;
}
```

この例では入力の内容に応じて変数 x の値を設定しています。ただそのためだけに 5 行もプログラムを書くことになっています。この長ったらしさ解消するのが**三項演算子**です。これは条件となる一つの bool 型の式と結果となる二つの式といった三つのオペランドをとります。

```
条件 A ? 式 1 : 式 2
```

式 1 と式 2 の結果は同じ型でなければなりません。そして、どちらかがこの演算子の結果となります。先ほどの例は、以下のように書き換えることができます。

```
int x{}, a{};
std::cin >> a;
x = (a < 0) ? (-a * 10) : (a * 20);

または

int a {};
std::cin >> a;
int x {(a < 0) ? (-a * 10) : (a * 20)};
```

丸括弧をつけていますが、プログラムが曖昧にならない場合には無理に括弧をつける必要はありません。さらに、この演算子は if 文と同様に入れ子で使うと、3 個以上の選択肢から選ぶ処理を簡潔に表現できます。

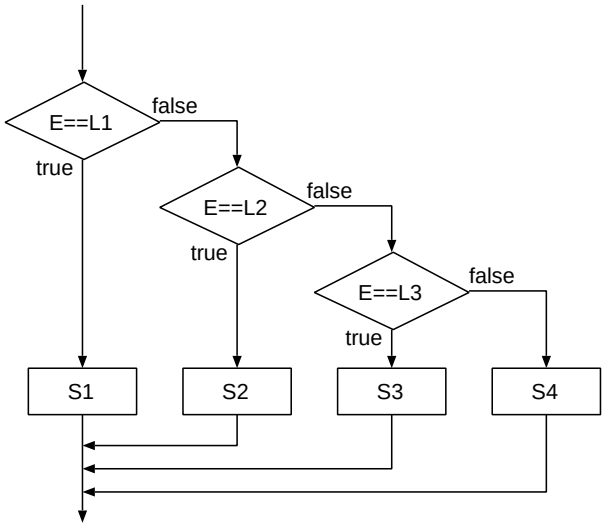
```
std::cout << (
   (a == 1) ? "one":
   (a == 2) ? "two":
   (a == 3) ? "three":
   (a == 4) ? "four": "other"
) << "\n";
```

これは前述の非公式用語の else-if の処理に相当する書き方です。

switch 文

非公式用語の else-if の処理やそれと類似の三項演算子の連結処理は、様々な場面で使われます。そのため、int や char の整数系データ型では、変数の値に応じていくつかの処理を別々に行うための switch 文が用意されています。それは次のような形をしています。

```
switch（条件となる式 E）{
case ラベル L1:
    文 S1;
    break;
case ラベル L2:
    文 S2;
    break;
case ラベル L3:
    文 S3;
    break;
default:
    文 S4;
}
```



break 文は switch 文の処理を終えること指定するための実行文です。フローチャートは break 文がこの通り指定された場合を表しています。上の方から見ていきましょう。まず**条件となる変数** E は整数系データ型の式や変数です。整数系データ型なので bool 型も使用できますが、使うべきではありません。**ラベル** L1 などには条件式 E に対応する値を指定します。これを case **ラベル**と呼びます。ただし、これにはリテラルなどのコンパイル時に定まっている値が必要で、const 指定のない変数は指定できません。最後の default:はどのラベルの値にもマッチしなかった場合を表すラベルです。これは省略しても構いませんが、指定以外の処理はエラー処理となることが多く、また、条件の列挙し忘れもよく起こるので、省略しない内容を考えるようにしましょう。このように switch 文を使うことで指定した条件に応じて、いわゆる else-if と同じように、複数の処理から一つを選んで実行させるプログラムが書けます。注意点は break 文の使用の有無を確認することです。case ラベルは選択された処理の始めを表すだけです。break 文がなければ次の case ラベルに進みます。

switch 文を使った具体的な例を以下に示します。

```
char ch {};
int apple{}, banana{}, candy{};
std::cin >> ch;
switch (ch) {
case 'a':
   std::cout << "apple\n";
   ++ apple;
   break;
case 'b':
   std::cout << "banana\n";
   ++ banana;
   break;
case 'c':
   std::cout << "candy\n";
   ++ candy;
   break;
default:
   std::cout << "unknown\n";
}
```

この例は入力された文字に応じて、apple, banana, candy そして unkown を表示するものです。単に表示するだけならば先ほどの三項演算子の例と同じですが、これは式ではないので、必要に応じて他の処理を行う文を指定できます。次の例は case ラベルごとに break 文を書かない例です。

```
int longnamevariable {};
std::cin >> longnamevariable;
switch (longnamevariable) {
case 1: case 2: case 3:
case 5: case 7: case 11:
    std::cout << " あたりです\n";
    break;
default:
    std::cout << " はずれです\n";
}
```

条件が複数の場合には、単に case ラベルを並べた形となり、break 文を省略した形でもあります。また、この例はプログラムが簡潔に書ける例にもなっています。if 文や三項演算子を使う場合には、複数の条件で比較する値のみが異なる場合でも、個々の条件式にはすべてを指定しなければなりません。そのため、longnamevariable のような長い変数名を何度も書く羽目になります。一方で、switch 文のこの書き方は case と対象となる値を並べるだけで済みます。

default:の部分の処理に break 文を書くならば最後ではなく最初や途中に指定しても良いことになっています。以下の例では、1 と-1 以外を不正な値とします。

```
int n {};
std::cin >> n;
switch (n) {
case  1:
   std::cout << "1を検出\n";
   break;
default:
   std::cout << " 不正な値\n";
   break;
case -1:
   std::cout << "-1を検出\n";
   break;
}
```

switch 文はコンパイル時の値を指定するという制限に加えてもう一つ注意点があります。case ラベルの後には複数の文を書けるのですが、変数を新たに宣言する場合には**複文**が必要という点です。

```
switch (a+b) {
case 3:
     {  // 複文にする
        int x {};  // 新たな変数宣言
        ....       // xを使った処理
        break;
     }
case 4:
...
```

この一貫性のなさは継承元の C 言語において、そもそもこの場所に変数宣言を書くことが想定されていなかったことが起因しているようなので、仕方がありません。case ラベルはアセンブリ言語という別の言語のやり方に由来していて、その面でも変数についての制限が異なっているのは仕方がありません。Go などの最近のプログラミング言語では switch 文に改良を加えた構文を持っています。どこが違うかを調べてみるのも良いでしょう。

3 string と vector

これまでのプログラム例では単一の値を扱う変数を見てきました。様々な処理を行うにはこれに加えて複数の値をまとめて扱うことも重要です。複数の値をまとめて格納するデータ構造には配列と呼ばれるものがあります。C++は C 言語を拡張して作られているために、C 言語の配列がそのまま使えます。しかし、C 言語の配列は最低限の管理機能しか提供されておらず、ポインタと呼ばれる概念と組み合わせて使うことが必須であったり、データの個数が変わる場合にメモリ管理が必要など、使用するには多くの知識が必要です。さらに、使い方に問題があるとコンピュータウィルスの標的になり危険という問題も抱えています。C++ではそれらの問題を緩和するために string と vector が提供されています。string は文字専用の配列として使用し、vector は汎用の配列として使用します。

3.1 string 型

電子メール・ワープロ・Web アプリケーションなど、現代のコンピュータアプリケーションにおいて、文字・文・文章の処理はとても重要です。文を構成する文字列は C 言語では char 型の配列を使用していました。これは C-string と呼ばれます。C++では C-string も使えますが、string 型と呼ばれる標準ライブラリのデータ型が導入され、文字処理プログラムが簡潔に書けるようになっています。

string 型の基本的な使い方

string 型（正式には std::string) は int や double などの基本データ型とは異なり、複数の型を組み合わせて作られたクラスと呼ばれるデータ型です。そのため単独で使用するには、<string>ヘッダファイル のインクルードが必要ですが、<iostream>ヘッダファイルをインクルードしているならば、<string>ヘッダファイルは内部で取り込まれるために明示的な指定は不要です。

ソースコード 13: string の基本

```
1  // string 型を使った例
2  #include <iostream>
3  #include <string> // iostream を指定しているので省略できる
4  int main()
5  {
6     std::string s {"Hello!"}; // 初期化には文字列リテラルを指定
7     std::cout << s <<"\n"; // 基本データ型と同様にcout が利用できる
8
9     s = "How are you?"; // 後から文字列リテラルを代入しても良い
10    std::string s2 {s}; // 他の変数の初期値としても良い
11    std::string s3; // 初期化を指定しないと ""での初期化と同じ
12    s3 = s2; // 他の変数に代入しても良い
13    std::cout << s3 <<"\n";
14
15    std::cin >> s2; // cin も利用できる
16    s3 = s2 + "! "; // 文字列リテラルとの連結
17    s3 = s3 + s3; // 変数どうしの連結
18    std::cout << s3 <<"\n";
19
20    s2 += ", thank you,"; // 文字列リテラルを後ろにつなぐ
21    s3 = " and You";
22    s2 += s3; // 他の変数を後ろにつなぐ
23    s2 += '?'; // 一文字を後ろに加える
24    std::cout << s2 <<"\n";
25    return 0;
26 }
```

string 型の扱う値は文字列なので、初期値には文字列リテラルを指定します。基本データ型と同様に、cin と cout で入出力が行えます。文字列を処理するための主要な演算は連結です。string 型の変

数に対する+演算子の指定は、文字列の連結を行うという意味です。s2 += s3; は、s2 = s2 + s3; を意味しており、s2 の後ろに s3 の文字列を連結します。なお、連結は文字列だけでなく char 型の一文字をつなぐこともできます。

このプログラムの実行例は以下のようになります。

```
Hello!
How are you?
Fine                          <<----- これを入力
Fine! Fine!
Fine, thank you, and You?
```

連結については注意点があります。連結できるのは、string 型の変数どうし、string 型変数と文字列リテラル、string 型変数と文字リテラル（char 型変数の値）です。従って、"abc" + "xyz"のように文字列リテラルどうしの連結はできません。文字列リテラルは C-string であり、+演算子での連結はできないからです。ただし、+演算子は左結合であるために、string 型変数の s を含んだ s + "abc" + "xyz" は連結できます。この連結は（s + "abc"）+ "xyz"と解釈され、先に s + "abc"がなされ、その結果が string 型となるためです。

string 型変数の比較

string 型の変数は、同じ string 型の変数および文字列リテラルと比較ができます。次のプログラムはユーザからの入力文字列をあらかじめ用意した文字列と比較するプログラムです。

ソースコード 14: string 型変数の比較

```
1  // string 型変数の比較
2  #include <iostream>
3  int main()
4  {
5      std::cout << "Let's play a guessing game!\n"; // 当てっこゲームをしよう!
6      std::cout << "A string with 3 letters begins with \"C\"\n"; // C で始まる 3 文字
7      std::string x, ans{"C++"};
8      std::cin >> x;
9      if (x == ans)
10        std::cout << " correct!\n";
11     else if (x == "CPU")
12        std::cout << " no...\n";
13     else
14        std::cout << " incorrect...\n";
15     return 0;
16 }
```

これを実行すると例えば以下のようになります。

```
$ ./a.out
Let's play a guessing game!
A string with 3 letters begins with "C"
CPU                           <<----- これを入力
 no...
$ ./a.out
Let's play a guessing game!
A string with 3 letters begins with "C"
CAD                           <<----- これを入力
 incorrect...
$ ./a.out
Let's play a guessing game!
A string with 3 letters begins with "C"
C++                           <<----- これを入力
 correct!
```

string 型は==と != による同じ文字列であるかの比較だけでなく、<や<=などの順序比較（または大小さの比較）もできます。比較は先頭の文字から ASCII コード表 (P.160) に基づき行われます。例えば、abcd < abxa は true です。これは文字列の先頭から比較して 1 文字目の a と 2 文字目の b は同じですが、3 文字目は c < x のためです。make < maker の比較のように、途中まで同じで文字列の長さが違う場合には、短い方が小とされます。従ってこの例も true です。

次の例は文字列の順序比較（どちらが小さいかの比較）を行っています。

ソースコード 15: string 型変数の大小比較

```
// string の大小比較
#include <iostream>
using std::cout;
int main()
{
   std::string a {"abc"};
   if (a < "xyz")
      cout << a <<" < xyz is true\n";

   std::string b;
   cout <<">> " && std::cin >> b;

   cout << a <<" <= "<< b <<" is ";
   if (a <= b)
      cout <<"true\n";
   else
      cout <<"not true\n";
   return 0;
}
```

実行してみましょう。

```
$ ./a.out
abc < xyz is true
>> abc                          <<----- これを入力
abc <= abc is true
$ ./a.out
abc < xyz is true
>> nnn                          <<----- これを入力
abc <= nnn is true
$ ./a.out
abc < xyz is true
>> ab                           <<----- これを入力
abc <= ab is not true
```

区間の表し方

ある範囲に含まれる実数を表すために**区間**という言葉が数学で使われます。下限と上限の値でその間にある実数の集合を指定しますが、区間に値を含めるか否かで四種類の区間ができます。それらは下限の値を a、上限の値を b、集合の要素を x とした場合に以下のように表されます。

名前	表記法	要素の条件
閉区間	[a, b]	$a \leq x \leq b$
開区間	(a, b)	$a < x < b$
半開区間（左開右閉）	(a, b]	$a < x \leq b$
半開区間（左閉右開）	[a, b)	$a \leq x < b$

数学では実数が対象ですが、C++では整数などの離散値が対象で半開区間 [a, b) を使います。

string 型変数の文字へのアクセス

string 型変数が持つ内部のそれぞれの文字にアクセスするには、何番目の文字かを示す 0 から始まる序数を指定します。例えば、"abc"を値に持つ string 変数 s の先頭文字にアクセスするには、s[0] と指定します。この序数は一般に**配列の添字**と呼ばれます。注意点は 0 から始まることです。文字列が n 文字であったならば、半開区間 [0,n) に入る i を使い、s[i] などと表記して文字列中の文字を指定します。i の部分は変数以外に整数の値や式を指定できます。例を見てみましょう。

ソースコード 16: string 型変数の文字へのアクセス

```
1  // string の文字へのアクセス
2  #include <iostream>
3  int main()
4  {
5     std::string s {"abcXefg"};
6     std::cout << s <<"\n"; // abcXefg
7     char ch = s[3]; // 'X'が取り出される
8     std::cout << ch <<" "<< s[3] <<"\n"; // X X
9     s[3] = 'd'; // 'X'を'd'に変更
10    std::cout << s <<"\n"; // abcdefg
11
12    std::string t;
13    std::cin >> t;
14    if (!t.empty()) {
15       std::cout << t[0] <<" "<< t[t.size()-1] <<"\n";
16       std::cout << t.front() <<" "<< t.back() <<"\n";
17       t.front() = 'X';
18       t.back() = 'Z';
19       std::cout << t <<"\n";
20    }
21    return 0;
22 }
```

この例はまず、string 型変数 s の添字 3 の文字 ('X') を取り出し、そして変更しています。次に、t に対して文字列を cin で入力し、それが空文字列ではない場合に、先頭と末尾の文字を処理しています。t.empty() や t.size() は string 型の**メンバ関数**と呼ばれるもので、ドット (.) で区切って一緒に指定された string 変数（この場合は t) に関する処理を行います。t.empty() は変数 t が空文字列であるかどうかを判定し、結果を bool 型の値で返します。t.size() は変数 t が持つ文字列の文字数を整数で返します。そのため、t.size() == 0 と t.empty() は同じ結果です。そして t.size()-1 は、空文字列でない場合の末尾文字の添字の値となります。また、先頭と末尾の要素は良く使われるので t.front() と t.back() というメンバ関数でアクセスができます。これらはメンバ関数の呼び出しですが、t.front() = 'X'; というように値の変更にも使用できます。これは t.front() が t[0] と同じく左辺値として使えるためです。

このプログラムの実行結果は以下のようになります。

```
abcXefg
X X
abcdefg
ijklmn      <<----- これを入力
i n
i n
XjklmZ
```

string 型のメンバ関数

string 型のメンバ関数は size() や empty() 以外にも多数ありますが、その中でも比較的良く使われるものを紹介します。

ソースコード 17: string 型のメンバ関数

```
// string 型のメンバ関数
#include <iostream>
int main()
{
   std::string s {"abcdefghijabc"};
   int i, j, k, l;
   i = s.find("cde");
   j = s.find("d");
   k = s.rfind("abc");
   l = s.rfind("b");
   std::cout << i <<" "<< j <<" "<< k <<" "<< l <<"\n";

   if (s.find("X") == std::string::npos)
      std::cout << "X is not found\n";

   std::cout << s.substr(i, 3) <<"\n";
   std::cout << s.substr(k) <<"\n";
   s.replace(k, 3, "klmn");
   std::cout << s << "\n";
   return 0;
}
```

s.find(x) は引数で指定した部分文字列 x を s 内で探索して、見つけた場合に x[0] が対応する s[i] の添字の値 i を返します。引数 x は、string 型変数、文字列リテラル、char 型の値を指定できます。char 型が指定された場合には、探索対象が部分文字列ではなく、単に一文字を探すことになります。s.rfind() は s の末尾から探索を行います。rfind() の r は reverse の r と考えれば良いでしょう。これら二つのメンバ関数は、指定された文字列を見つけられない場合があります。その場合にはサイズを表す符号なし整数の最大値を返すことになっていますが、その値は実行するコンピュータによって変わります。そのためその大きな値を std::string::npos で表すことになっています。npos は、not a position（位置を示す値ではない）と覚えれば良いでしょう。

s.substr() は s から部分文字列を取り出すメンバ関数で、整数 a と b を用いて s.substr(a,b) と指定したら、s の添字 a の文字から b 文字分が取り出す対象となります。もし、s.substr(a) と引数を一つだけ指定すると、添字 a の文字から末尾までが対象となります。

s.replace(a,b,c) はその名のとおり置き換えるためのメンバ関数で、a の添字の文字から b 文字分を c の文字列で置き換えます。上記のコード例が示すように、b の文字数と c の文字数は同じである必要はありません。また、このメンバ関数も s.substr() と同様に、いくつかの引数の指定方法があるのですが、この基本的な使い方だけ覚えておけば良いでしょう。

このプログラムの実行結果は以下となります。

```
2 3 10 11
X is not found
cde
abc
abcdefghijklmn
```

数値と文字列の相互変換

文字列を操作していると int 型や double 型の値を文字列にしたり、数値を表す文字列を int 型や double 型の変数に格納したりと、相互に変換したい場合が出てきます。次の例はそれを行う関数の紹介です。

ソースコード 18: to_string()stoi()stod()

```
1  // 数値と文字列の相互変換
2  #include <iostream>
3  int main()
4  {
5     std::string s {"i = "};
6     int i {10};
7     s += std::to_string(i);
8     double r {2.5};
9     s += ", r = " + std::to_string(r);
10    std::cout << s <<"\n";
11
12    std::string ten{"10s"}, pi{"3.1415 rad"};
13    i = std::stoi(ten);
14    r = std::stod(pi);
15    std::cout << i <<", "<< r <<"\n";
16    return 0;
17 }
```

std::to_string() は引数に int や double の数値を指定して呼び出すと、それを表す 10 進数の文字列を返します。引数の型で整数と実数を区別できるので、関数の名前は一つです。std::stoi() は引数としてされた文字列を 10 進数の数と解釈して、対応する int 型の値を返します。例のように文字列の後ろ側には数字以外の文字が含まれていても構いません。std::stod() は 10 進数の実数と解釈して double 型の値を返します。

このプログラムの実行結果は以下となります。

```
i = 10, r = 2.500000
10, 3.1415
```

std::stoi() や std::stod() は単に変換をするだけでなく、変換対象とならなかった先頭文字の添字の値を取得する手段も提供しています。インターネットなどで使い方を調べてみましょう。

3.2 vector 型

vector は同一の型の値を複数同時に扱うためのものです。これは string 型と同様に C++に元々備わっている組み込み型ではなく、後から加えたものです。そのため、vector を用いるには、<vector>ヘッダファイルをインクルードする必要があります。また、string 型とは異なり配列の要素となる型が固定ではないので、変数宣言の際には型を指定する必要があります。一般に、T 型配列として vector を使用するためには、要素の型 T を用いて std::vector<T> という形式で指定します。例えば、

```
std::vector<int>    x;
std::vector<double> y;
```

と宣言すると、x は int 型データを要素とする配列、y は double 型データを要素とする配列として使用できます。ただし、初期値によって要素となるデータ型がはっきりしている場合には、<T>の部分を省略できます。この省略は C++17 の拡張機能です。

vector 型の基本的な使い方

次の例は vector の変数の基本的な使い方を示したものです。

ソースコード 19: vector の基本

```
// vector の基本操作
#include <iostream>
#include <vector>
int main()
{
   std::vector<int> a; // 要素 0個
   std::cout << a.size() <<"\n"; // 0

   std::vector<int> b(3); // 要素 3個，初期値 0
   b[0] = 1;
   b[2] = 3;
   std::cout << b[0] <<" "<< b[1] <<" "<< b[2] <<"\n"; // 1 0 3
   std::cout << b.size() <<"\n"; // 3

   std::vector c {2,3,4,5}; // std::vector<int>
   std::cout << c[0] <<" "<< c[1] <<" "<< c[2] <<" "<< c[3] <<"\n"; // 2 3 4 5
   std::cout << c.size() <<"\n"; // 4

   std::vector d {b}; // 要素のコピーによる初期化
   d[1] = 2; // 要素の書き換え
   std::cout << d[0] <<" "<< d[1] <<" "<< d[2] <<"\n"; // 1 2 3
   std::cout << d.size() <<"\n"; // 3

   a = b; // 要素のコピー
   std::cout << a[0] <<" "<< a[1] <<" "<< a[2] <<"\n"; // 1 0 3
   std::cout << a.size() <<"\n"; // 3
   return 0;
}
```

vector は要素の数を増やしたり減らしたりすることができ、std::vector<int> a; のように宣言をすると要素数は 0 個です。string 型と同様に a.size() メンバ関数によって要素数を知ることができます。次の std::vector b(3); は要素数を指定して変数宣言をしています。ここに指定する数は入力や計算で得た値でも構いません。そして、string 型と同様に、要素数を n とした時に半開区間 [0,n) の値の添字を使って各要素にアクセスできます。つまり、i を添字とした時に、$0 \leq i < b.size()$ の範囲の b[i] にアクセスできます。

変数 b のように要素数だけを指定して宣言した場合は、すべての要素が 0 の配列になります。それ以外の初期値をすべての要素に指定したい場合には、次のように要素数に加えてその初期値を指定します。

```
std::vector<int>    x(5, 10);   // すべての値が10の5個の配列
std::vector<double> y(6, 1.5);  // すべての値が1.5の6個の配列
std::vector<bool>   z(4, true); // すべての値がtrueの4個の配列
```

コード例に戻って、変数 c の宣言は初期値を個別に指定した例です。std::vector c {2,3,4,5}; の宣言では、初期値によって要素が int 型と分かるので、<int>の指定を省略できます。全体の要素数は初期値で指定された要素の数と同じです。

4 番目の変数 d の宣言も std::vector d {b}; と書いた初期値の b によって要素の型と要素数がはっきりしているので、<int>の指定を省略しています。

最後は代入文の例です。a = b; によっては a の要素数は 0 個から b の要素数に変更となり、値がすべてコピーされます。ただし、この要素のコピーは、たとえ要素数が 1000 や 10000 であっても行われます。要素をコピーする操作は 10 や 20 ならばたいしたことはありませんが、数が増えれば無視できない時間になります。そのため、不用意にコピーをしないように気をつけなければなりません。

このプログラムの実行結果は以下となります。

```
0
```

```
1 0 3
3
2 3 4 5
4
1 2 3
3
1 0 3
3
```

vector 型変数の宣言の仕方をまとめると次のようになります。丸括弧を使うと要素数や共通の初期値の指定となり、中括弧を使うと個別の初期値により個数も同時に指定することになります。そして、初期値により要素の型が明確な場合には<T>の部分が省略できます。丸括弧と中括弧はまったく違う意味となるので間違わないようにしましょう。

```
std::vector<double> a;                  // 要素数0の変数
std::vector<double> b(5);               // 要素数5個，初期値はすべて0.0
std::vector<double> c(5, 1.4);          // 要素数5個，初期値はすべて1.4
std::vector<double> d {1.2, 2.5, 3.5}; // 初期値で個数を指定
std::vector c2(5, 1.4);                 // cの省略形
std::vector d2 {1.2, 2.5, 3.5};         // dの省略形
```

要素の追加と削除

vector は要素を後から追加したり削除したりできます。次の例はそのやり方を示しています。

ソースコード 20: vector 要素の追加と削除

```
1  // vector 要素の追加と削除
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     std::vector<double> x;
7     x.push_back(1.5);
8     x.push_back(2.8);
9     x.push_back(3.3);
10    std::cout << x.size() <<": "<< x[0] <<" "<< x[1] <<" "<< x[2] <<"\n\t"
11              << x.front() <<" "<< x.back() <<"\n";
12
13    x.pop_back();
14    std::cout << x.size() <<": "<< x[0] <<" "<< x[1] <<"\n\t"
15              << x.front() <<" "<< x.back() <<"\n";
16
17    x.clear();
18    std::cout << x.size() <<"\n";
19    return 0;
20 }
```

メンバ関数の x.push_back() は末尾に要素を加える操作です。この例では要素数 0 個で宣言した vector 変数に追加していますが、宣言時に要素数を指定した場合にもさらに追加する形で機能します。これによって、x.size() の値も変化します。x.front() と x.back() は string 型と同様に先頭と末尾の要素にアクセスするための方法です。つまり、x.front() が x[0] に、x.back() が x[x.size()-1] に対応し、これらはどれも左辺値となります。

x.pop_back() は末尾の要素を削除して、x.size() の結果に影響を与えます。x.clear() は一度にすべての要素を削除します。

このプログラムの実行結果は以下となります。

```
3: 1.5 2.8 3.3
1.5 3.3
```

```
2: 1.5 2.8
1.5 2.8
0
```

vector 変数の比較

vector は string と同様にすべての要素をまとめて比較ができます。比較方法も string 型の変数と同様で、先頭の要素からそれぞれ比較していき、すべて同じならば true、一つでも異なれば false です。これによって、複数の要素を持つ変数を int や double のような単一の値のデータ型のように扱えます。ただし、ユーザ定義型を要素とする場合には、その型の要素が比較可能である必要があります。

以下の例は int 型の配列を比較するプログラムです。

ソースコード 21: vector 変数の比較

```
1  // vector 変数の比較
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6
7     std::vector v1 { 1, 2, 3, 4, 5 };
8     std::vector v2(v1);
9
10    if (v1 == v2)
11       std::cout << "equal\n";
12
13    v2.front() = -10; // 先頭要素を書き換える
14    v2.back() = 10; // 末尾要素を書き換える
15
16    if (v1 == v2)
17       std::cout << "equal\n";
18    if (v1 != v2)
19       std::cout << "not equal\n";
20    if (v1 < v2)
21       std::cout << "less than\n";
22    if (v1 <= v2)
23       std::cout << "less than equal\n";
24    if (v1 > v2)
25       std::cout << "greater than\n";
26    if (v1 >= v2)
27       std::cout << "greater than equal\n";
28    return 0;
29 }
```

このプログラムの実行結果は以下となります。

```
equal
not equal
greater than
greater than equal
```

string を要素とする vector 変数

vector は指定により任意の型の配列を作れます。そのため、std::vector<char>とすれば std::string のようなものもできます。ただし、std::string は十分に高機能なのでこれを使うことはほとんどありません。しかし、std::vector<std::string>は多くの場面で使われます。以下はその単純な例ですが、いくつかの注意点を含んでいます。

ソースコード 22: string を要素とする vector

```
1  // string を要素とする vector
```

```
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     std::vector<std::string> vs {"abc", "def", "ghi"};
7     vs.push_back("jkl");
8     std::string s {"mnop"};
9     vs.push_back(s);
10    std::cout << vs.size() <<" "<< vs.front() <<" "<< vs.back() <<"\n";
11
12    s = vs[0];
13    std::cout << s[1] <<" "<< vs[0][1] <<"\n"; // b b
14    return 0;
15 }
```

まず、初期値を指定した変数の宣言ですが、<std::string>は面倒でも省略はできません。省略して宣言すると 6 行目は問題ありませんが、9 行目でエラーとなります。次のように省略した宣言をするとどうなるのでしょうか?

```
std::vector y {"abc","efg"};  // 型に注意! これだけならばエラーではない。
```

この変数宣言では vector の要素に文字列リテラルを指定しています。これはもともと C-string であるために、vector の型を省略すると std::vector<char const*>と同等になります。char const*は文字列リテラルに対応する型で、C 言語の char 型配列の指定に使われます。9 行目で std::string 型の値を push_back() しようとすると型が違うのでエラーとなります。一度、std::vector<std::string> の形で変数宣言をしてしまえば、vs.push_back("jkl") のように文字列リテラルを指定しても構いません。これは文字列リテラルが std::string に変換されるためです。理由は難しいのですが、基本として、std::string が関係する時には省略しないと覚えておくのがよいでしょう。

次に注意すべき点は文字列の要素へのアクセスの指定です。例にあるように、s = vs[0]; のような方法で要素にアクセスすると、std::string 型の値が取り出されます。変数 s は std::string 型であるために、個々の文字は s[1] という形で取り出せます。この 2 段階の取り出しは、vs[0][1] と省略できます。これは、(vs[0])[1] と括弧をつけて考えれば、2 段階で取り出していることが理解できると思います。

このプログラムの実行結果は以下となります。

```
5 abc mnop
b b
```

vector の操作

vector の様々な操作の中でよく使われるものを以下に示します。

```
v.push_back(t)   // vの末尾に値tを追加する
v.pop_back()     // vの末尾要素を削除する
v.clear()        // vの全要素を削除する
v[n]             // vのn番目の要素へのアクセス(読み書き可能)
v.size()         // vが持つ要素数を返す
v.front()        // vの先頭要素へのアクセス(読み書き可能)
v.back()         // vの末尾要素へのアクセス(読み書き可能)
v.empty()        // vが空ならばtrue それ以外は falseを返す
v1 = v2          // v2の要素のコピーでv1を上書きする
v1 == v2         // v1とv2が同一要素を持つならばtrue,それ以外はfalseを返す
v1 != v2         // v1とv2が同一要素を持たないならばtrueそれ以外は falseを返す
```

要素へのアクセスや要素の削除の操作は、vector 変数に要素が 1 個以上あることが前提です。空の vector 変数にこれらの操作を行ってはなりません。そのため、v.empty() メンバ関数を使って要素の有無を確認しながら行う方法がよく使われます。

4　繰り返し処理

一人が多数の仕事を抱えている場合には、一つ一つの仕事を順番にこなしていくことになります。それぞれがまったく異なる種類の仕事ならば、種類にあった別々の作業を行うのですが、種類が似ているならばどこかで繰り返しの作業が発生します。繰り返しの作業は一ヶ所の効率化によって全体の作業時間を劇的に短くできる可能性があるので、そのやり方を良く考える必要があります。一方で、もし複数人で仕事をこなす場合には、仕事の分担と最終的なまとめをうまく進めると多くの仕事が早期に終わります。繰り返し作業と同じく、分担した作業内容の種類が似ているならば、分けた仕事を同じ方法で効率化を行うと良いことが分かっています。これは知識やノウハウの共有と呼べるでしょう。結局、多数の仕事は、一人の繰り返し作業か多人数の分担作業となるわけですが、繰り返される部分の効率化はもっとも重要です。

現代のコンピュータは一つずつ処理をこなす形で開発されてきたために、複数の作業は繰り返しで行うことになります。結果としてプログラムでは繰り返し作業の指定が基本です。昨今では、マルチコア、複数 CPU、コンピュータ・クラスタといった仕事を分担するための仕組みも揃ってきたので、プログラムの中で仕事の分担と最終的なまとめといった並列処理を指定できる場合もでてきています。しかし、並列処理を基本としてプログラムを書くスタイルは主流ではありません。ただ様々な研究や開発で分かってきたことは、繰り返し作業をうまく構成して、ある種のパターンとして指定しておくと、それを利用して並列の作業に移行できるということです。そのため、現代のプログラミング学習では、処理のパターンを意識してプログラムを書くことが、今まで以上に大切になってきています。

プログラムで多数の仕事をこなすとは、多数のデータを計算処理していくことです。多数のデータを処理していく状況は様々に考えられますが、以下の 3 種類は良く起こる状況なので意識しましょう。

- 対象データは配列などのデータ構造に収められており、全体を把握できている。
- 個々の対象データは数式や数学関数で取り出すことができるが、全体は数学で扱う空間から範囲を考えて切り出さねばならない。
- 対象データがプログラムの外部にあって、センサーデータのように次々入ってくる。

これら多数のデータをどのようにしていくかも様々に考えられますが、以下の 3 種類を意識すると良いでしょう。

- 多数のデータから何かを探す処理
- 多数のデータをまとめる方向で処理していく
- 多数のデータのそれぞれの値を変更していく

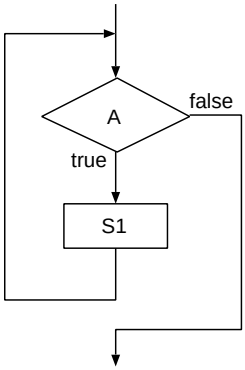
例えば、現代のインターネット社会ではデータの暗号化が欠かせませんが、その暗号には素数が深く関わっています。ある数が素数であるか否かを判定するには、整数全体から素数を探すことに対応します。また、成績処理を考えるならば科目ごとの点数がどこかにあり、それをまとめていく方向で計算していきます。

それぞれの処理パターンにプログラムのパターンを対応させると良いです。この章ではいくつかの典型的なプログラムのパターンを示しますので、やるべき作業とプログラムのパターンの関係を考えながら学習してください。

4.1　while 文による繰り返し処理

`while` 文は、条件を満たしている間だけ指定した文を繰り返し実行します。これは汎用的でかつプログラムの中で一番多く使われる繰り返しのパターンです。次のような形式で指定します。

```
while (条件 A)
    文 S1
```



条件 A には、if 文と同様に、bool 型の式を指定します。まずは右側のフローチャートで流れを確認しましょう。矢印に沿って、まず条件 A の bool 式を計算します。そして if 文と同じく結果が true のときに下側に沿って進んで S1 を行います。S1 は単一の実行文なので、複数の文にする場合には複文を指定します。false の場合には S1 を迂回するように次の処理に進みます。if 文と異なるのは S1 を行った場合で、次の処理に進まずに A の計算に戻ります。つまり、条件 A が満たされている限り、S1 を行うことになります。気をつける点は、S1 の実行または外部からの影響により A の計算に変化が起こるかどうかです。A に変化がなければ際限のない繰り返しとなります。これは無限ループと呼ばれます。

次のプログラムは、入力を変数 x に読み込み、その x の値が 0.01 より大きい間だけ、繰り返し x の値を半分に減らしていきます。出力は減らしていく過程と最終結果です。

ソースコード 23: 数値計算の繰り返し

```
1  // 指定の数値以下になるまで値を 1/2にする
2  #include <iostream>
3  int main()
4  {
5     double x {0.0};
6     std::cin >> x || std::cout<<"error\n"; // 入力失敗の場合に備える
7     while (x > 0.01) { // x が 0.01より大きいという条件で
8         std::cout << x <<"\n"; // {}で囲まれた２行を実行する
9         x /= 2;
10    }
11    std::cout << x <<"\n";
12    return 0;
13 }
```

x > 0.01 という条件が true の間だけ、8-9 行目の 2 個の文を繰り返し実行します。細かなことですが、x の初期値に 0.0 を指定したことにも注意しましょう。入力が失敗して x が変更されないならば、while 文に入らなくて済みます。このプログラムを実行した結果は例えば以下のようになります。

```
7                <<-- これを入力
7
3.5
1.75
0.875
0.4375
0.21875
0.109375
0.0546875
0.0273438
0.0136719
0.00683594
```

while 文と cin の組み合わせ

あらかじめ入力される値の個数が分かっていない場合には、while 文と入力の組み合わせがよく使われます。

ソースコード 24: 繰り返しの入力

```
// while 文による未知の個数の値入力
#include <iostream>
int main()
{
   int sum {0};
   int x;
   while (std::cin >> x) // 入力が成功する限り繰り返す
       sum += x;

   std::cout << sum << "\n";
   return 0;
}
```

std::cin >> x は入力を x に読み込む指定ですが、これは a+b という式のように、2 個のオペランドを>>という演算子に指定した式でもあります。式 cin >> x の結果は cin 自身です。一方、while 文の条件では bool 型となる式が必要です。cin は bool 型ではないためにおかしいように感じられますが、問題はありません。cin が bool 型が必要な所で使われると bool 型に変換されるからです。変換結果の値は入力が成功している場合には true、失敗した場合には false です。これらの理由により、while(std::cin >> x) と書くことで、入力が成功する限り繰り返すという処理が書けるのです。

入力が失敗するのは、すべての入力を読み尽くしてそれ以上読めなくなった場合と、不正な入力を読みとった場合です。一般に、すべての入力を読み尽くした状態を、「EOF(ファイルの終端) に到達した」と言います。この while 文の条件の書き方で x に処理の必要な値が読み込まれたかどうかを判断でき、入力の処理が不要になった時点で while 文のループから抜けることができます。

なお、>>演算子は左結合です。そのため cin >> x >> y と指定すると、まず cin >> x という式が先に評価され、結果が cin なので、次に cin >> y が評価されます。つまり、(cin >> x) >> y と同じです。

実行例は以下のようになります。

```
1 2 3 4 5            <<-- これを入力
a                    <<-- これを入力，数字以外なので入力は失敗する
15
```

キーボードからの入力で EOF に到達した状態を作り出す方法は、オペレーティングシステム (OS) によって異なります。Unix OS ならば Ctrl-D を、Windows OS ならば Ctrl-Z を押します。

while 文による vector への読み込み

入力個数が事前に分からない場合の配列への読み込みは、vector を使うと簡単にプログラムが書けます。次の例はその基本的な例です。

ソースコード 25: vector への入力

```
// while 文による vector への未知の個数の値入力と出力
#include <iostream>
#include <vector>
int main()
{
   std::vector<int> v;
   int x;
   while (std::cin >> x) // 入力が成功する限り繰り返す
       v.push_back(x);

   std::cout << v.size() << "\n";
   return 0;
}
```

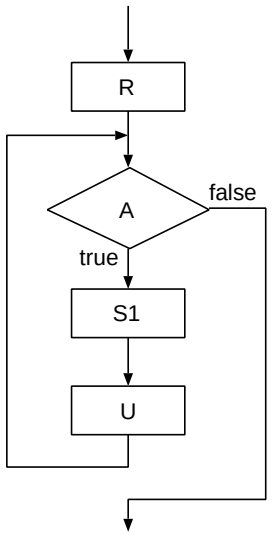
std::cin >> x の部分は前のプログラムと同じです。vector 型の push_back() メンバ関数を使ってデータを格納していき、後で配列内のデータ数を出力しています。注意点は、前のプログラムと違い、扱えるデータ数に上限があることです。入力されたデータは v に蓄えられていきますが、その数には上限があるからです。数万個ならば問題ないはずですが、それを越えるならば別の処理方法を考えた方が良いかもしれません。以下に実行例を示します。

```
1 3 5 7 9 11 13      <<-- これを入力，その後に Ctrl-D を入力
7
```

4.2 for 文による繰り返し処理

for 文は、繰り返しを始める前の処理 R と繰り返しごとの更新処理 U を、while 文に付加した形の文です。形式は次のようです[4]

```
for (初期化つき変数宣言 R; 条件 A; 式 U)
    文 S1

または

for (代入 R; 条件 A; 式 U)
    文 S1
```



while 文で良く使われるパターンを別の書き方で表せるようにしたと考えても良いでしょう。良く使われるパターンとは、ある変数を初期値から指定の値まで変化させ、その間だけ同じ文を繰り返すような処理です。フローチャートがやや複雑に見えますが、while 文のそれと比較すれば、R と U が増えただけです。for 文では、準備となる R、判定となる A、更新となる U を関係づけて指定することで、処理の内容を分かりやすく書けるようになります。関係づけに使われる変数を**制御変数**や**ループ制御変数**と呼ぶことがあります。多くの処理では、繰り返しにより指定された値の範囲を変化させます。ループの本体となる S1 では、この制御変数の値を使って計算したり、その値に応じて処理内容を変化させたりします。

次の例は、変数 i を 1 から 30 まで変化させることで、それらの合計と奇数だけの合計を求めています。

ソースコード 26: for 文による加算

```
1  // 1から30までの合計と奇数のみの合計を加算する
2  #include <iostream>
3  int main()
4  {
5     // 1から30までの合計
6     int sum {0};
7     for (int i = 1; i <= 30; i++) // i が 30 以下という条件で
8         sum += i;
9     std::cout << sum <<"\n"; // 465
10
11    // 3乗の値が1357までの奇数だけの合計
12    sum = 0;
13    int i;
```

[4]文法的にはこの形式の説明は正確でありません。詳しくは付録 (P.165) を見てください。

```
14     for (i = 1; i*i*i <= 1357; i++)
15         if (i % 2) // 奇数は2で割って余りがある
16             sum += i;
17     std::cout << i-1 <<":"<< sum <<"\n"; // 11:36
18     return 0;
19 }
```

for 文の int i = 1; という指定によって、i という制御変数を宣言します。ここで宣言された変数は、この for 文の中だけで使用できます。つまり有効範囲がこの for 文の中だけです。変数の宣言と初期化なので、int i{1}; と書いても良いですが、C++の設計者であるストラウストラップ氏の本でも int i = 1; の書き方をしているのであわせました。int 型の変数を for 文で使う場合には、この書き方で良いと思われます。次の i <= 30 が繰り返しを続ける条件です。そして、次の i++は i=i+1 と同じ意味の式で、ここに指定された式は、繰り返された文の後に必ず処理されます。その繰り返される sum += i; という文は、繰り返しのたびに変数 i の値だけ sum を増やすので、結果として、1 から 30 までの合計を計算します。

下側の for 文は i の変数宣言ではなく代入文になっています。これは繰り返しの後に制御変数の最終値が必要なためです。繰り返しの文が if 文になっていて、変数 i が奇数のときのみ加算をするようになっています。上側の for 文で宣言した制御変数 i の有効範囲は、その for 文の中だけなので、変数 i を下側の for 文で使用するには、下側の for 文より前に別途変数宣言が必要な点に注意してください。

for 文と vector

for 文の制御変数は、よく vector の添字の指定に使われます。その場合、for 文の準備 R・条件 A・更新 U は、vector の添字の範囲を明示する目的で指定されます。以下の例を見てみましょう。

ソースコード 27: for 文と vector の添字

```
1  // 実数の合計と平均の計算
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     std::vector v {1.5, 8.4, 2.3, 4.6, 3.5};
7     double sum {0.0};
8     for (int i = 0; i < v.size(); i++)
9         sum += v[i];
10    std::cout <<"sum: "<< sum <<"\n";
11    std::cout <<"avg: "<< sum/v.size() <<"\n";
12    return 0;
13 }
```

vector v の添字の範囲は、半開区間 [0, 要素数)、つまり $0 \leq i < v.size()$ です。そのためプログラム 8 行目は、この書き方ではっきりと範囲を示しています。条件の部分は i <= v.size()-1 と書いても間違いではありませんが、i < v.size() と書く方が良いです。

for 文による繰り返しの入力

ソースコード 24 で見た while 文と cin の組み合わせの例では、値を変数 x に入力していました。変数 x は繰り返しの中でしか使っていませんが、変数宣言は while 文の前で行っていたので、この変数の有効範囲は main 関数の最後までとなっています。短いプログラムの場合には問題ないのですが、長いプログラムにおいて変数の有効範囲を不必要に長くすると間違いの原因となります。for 文では、準備にあたる R の部分で変数宣言ができ、宣言された変数の有効範囲は for 文の中だけとなります。次のプログラムでは変数 x の有効範囲を限定しています。

ソースコード 28: for による繰り返しの入力

```
// for 文による未知の個数の値入力
#include <iostream>
int main()
{
   int sum {0};
   for (int x=0; std::cin >> x; ) // 更新処理を書かない
       sum += x;

   std::cout << sum <<"\n";
   return 0;
}
```

for 文の更新部分がないので、指定しないままとなっていますが、cin >> x; のセミコロン (;) は必要です。この書き方の良い点は、変数 x の有効範囲が狭いだけでなく、プログラムの行数が短くなり、中心となる変数 sum を目立たせる形となっているところです。

範囲 for 文による全要素の処理

string や vector のすべての要素に何かを行う場合には、**範囲** for **文** (range for statement) が適しています。

for (型名 要素用の変数名 e : 複数データの変数 v)
または
for (型名& 要素用の変数名 e : 複数データの変数 v)

この for 文で、string や vector に対応する変数 v の要素を一つずつ e に代入することで繰り返しを行います。二つ目の書き方は & が違うだけですが、e を更新することで、変数 v を更新できるようにする書き方です。範囲 for 文は C++11 で追加された書き方です。以下の例では、初期化した vector 変数の要素を個別に出力したり、各要素を 2 倍したりする際に、範囲 for 文を利用しています。

ソースコード 29: 範囲 for 文の例

```
// 範囲for 文の例
#include <iostream>
#include <vector>
using std::cout;
int main()
{
   // vector の場合
   std::vector v {1.2, 3.4, 5.6, 7.8, 9.0 };
   for (double d : v) // v から d に値を取り出す
      cout << d <<" ";
   cout <<"\n";

   for (double& d : v) // d を通して v を更新する
      d *= 2.0;

   for (double d : v)
      cout << d <<" ";
   cout <<"\n";

   // string の場合
   std::string s {"abcdefg"};
   for (char ch : s)
      cout << ch <<" ";
   cout <<"\n";
   return 0;
}
```

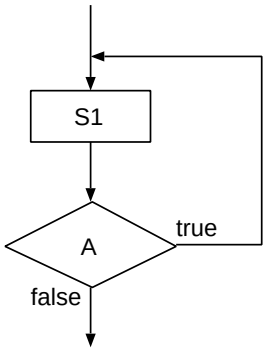
範囲 for 文は変数 v の各要素を d に取り出して処理をする方法です。v[0], v[1] と順番に要素が d にコピーされて、それぞれのコピーに対して、出力の処理を行います。要素を変更する場合には、& をつけて 12 行目の形で指定します。これはあとの章で扱うリファレンスという機能です。リファレンスにすると要素のコピーではなく、要素そのものを変数 d と結びつけて繰り返すことになります。そのため範囲 for 文が終わった後に元の v が変更されることになります。string の例では、要素の型は char 型なので、char 型変数の ch に文字コードである値を一つずつ取り出して出力しています。

範囲 for 文は string や vector だけのものではなく、コンテナと呼ばれる複数の値を保持する入れ物の役割をする型に使用できます。

4.3 do-while 文による繰り返し処理

do-while 文は繰り返しに対する処理を必ず 1 回は行う場合に指定するループです。これは次のような形式です。

```
do {
  文 S1
} while ( 条件 A );
```



他の繰り返し同様に中括弧 ({}) は省略しても良いのですが、do-while 文で省略されることはまずありません。また、フローチャートの形が while 文や for 文に比べて単純で、これはコンピュータの実行で無駄がないことを意味します。ところが do-while 文はあまり使われません。それは、必ず 1 回は処理を行う場面がそれほど多くない点と、繰り返しの条件がとても重要にも関わらず、それを後方に書かねばならず分かりづらいからです。do という単語が直感的に繰り返しと結びつかないという点もあるかもしれません。以下に例を示します。

ソースコード 30: do-while 文の例

```
1  // 対話的な温度の入力と平均値の計算
2  #include <iostream>
3  int main()
4  {
5     double sum {0.0};
6     int cnt {0};
7     char ch;
8     do {
9        double x {0.0};
10       std::cout << "Enter a temperature value: ";
11       std::cin >> x;
12       sum += x;
13       ++ cnt;
14       std::cout << "Do you have another one?(y/n) ";
15       std::cin >> ch || (ch = 'n');
16    } while (ch == 'y');
17    std::cout <<"avg: "<< sum/cnt <<"\n";
18    return 0;
19 }
```

このプログラムは対話的に温度を入力していき、最後に平均温度を出力します。必ず 1 回は入力を行うこととしたので、do-while 文を使っています。条件の書き方には注意点があります。この例の変数 ch のように、条件 A に相当する部分で使用する変数は、do-while 文の前に宣言しておく必要があります。条件部分はループ本体の外側であるために、中括弧の内側で宣言された変数は使用できないのです。

結果として、変数宣言と条件判定が離れた場所に配置されます。このような制限も do-while 文があまり使われない要因かもしれません。実行例を以下に示します。

```
Enter a temperature value: 35.2
Do you have another one?(y/n) y
Enter a temperature value: 36.3
Do you have another one?(y/n) y
Enter a temperature value: 38.2
Do you have another one?(y/n) n
avg: 36.5667
```

4.4 無限ループ

繰り返しを終わらせないプログラムも世の中には存在します。例えば、インターネットのサーバプログラムは、コンピュータを停止するまで終了することなく同じ処理をし続けます。また、ロボットや機械を制御するプログラムも電源を切るまで動き続けるかもしれません。そのような繰り返しを**無限ループ**と呼びます。無限ループの指定は、while 文でも for 文でも do-while 文でも条件を true に設定しておくだけです。for 文は特殊で条件を書かないという方法も許されています。良く見かける無限ループの書き方は次の 2 種類です。

```
    while (true) {
        // 無限に行う処理
    }

    for (;;) {
        // 無限に行う処理
    }
```

for 文による無限ループは、FreeBSD と呼ばれる OS のプログラムを書く際の C 言語の基本スタイルとして指定されています。C++では while を用いた書き方が良く使われるかもしれません。do-while 文で無限ループを指定することはほとんどありません。

4.5 入れ子の繰り返し指定

これまでに見てきた繰り返しはループ本体となる文として、代入文や if 文そして繰り返し文を指定できます。特に、繰り返しの中に入れ子の形で繰り返しを指定する書き方を**二重ループ**と呼びます。以下は、二重ループの例です。

ソースコード 31: 二重ループの例

```
1  // 120未満の素数の一覧
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     std::vector<bool> a(120, true); // 初期値はすべて true
7
8     // 条件に合わない数を探す
9     a[0] = a[1] = false;
10    for (int i = 2; i < a.size(); i++) {
11       for (int j = 2; i*j < a.size(); j++) {
12          a[i*j] = false;
13       }
14    }
15
16    // 結果の出力
17    for (int i = 2; i < a.size(); i++)
18       if (a[i]) std::cout << i <<" ";
19    std::cout <<"\n";
```

```
20      return 0;
21  }
```

これは 120 未満の自然数を調べて素数の一覧を出力します。素数とは 1 より大きい自然数のうちで、約数が 1 と自分自身の二つのみである数です。プログラムは vector を使って、要素数が 120 個で初期値がすべて true の bool 型の配列を作り、条件に合わない要素を false に変更していきます。最後に true を持つ配列の添字が素数となります。まず、定義により 0 と 1 は素数ではありません (9 行目)。次に二重ループにより、false の部分を決めていきます。$2 \leq i < 120$ の範囲で i に対して、その i の 2 倍, 3 倍, $\cdots$ の値は i で割りきれる値なので、その添字が対応する配列の要素を false にしていきます。これは「エラトステネスのふるい法」と呼ばれる計算方法です。

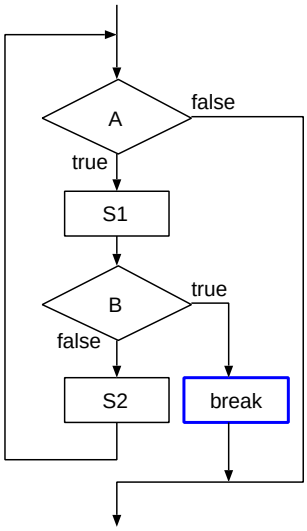
演習

このプログラムは i++や j++に関係する部分を修正すると格段に速くなります。考えてみましょう。

4.6 break 文と continue 文による繰り返し制御

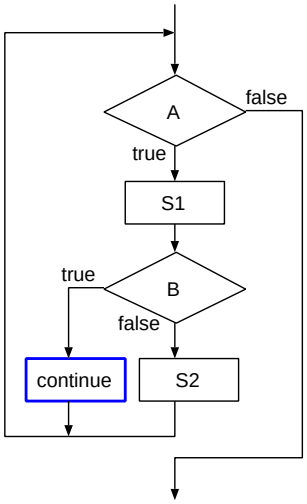
while 文と for 文はどちらも繰り返しの条件判定を行ってから、ループ本体を実行します。反対に do-while 文はループ本体を実行してから最後に次の繰り返しを行うかどうかを判定します。処理によっては、ループ本体を構成する複数の文のどれか途中で繰り返しを止めてしまったり、残りの処理をスキップして次の繰り返しを始めたい場合があります。その用途に使うのが break 文と continue 文です。これら二つはループの中で使う文ですが、break 文は switch 文でも使用します。通常はどちらも if 文と組み合わせて指定します。以下は while 文・if 文・break を組み合わせた場合のフローチャートです。

```
while ( 条件 A ) {
  文 S1
  if ( 条件 B )
    break;
  文 S2
}
```



if 文で繰り返しを途中でやめる条件 B を確認してから break 文を実行します。それにより文 S2 を行わずに繰り返しを終了します。次のフローチャートは break 文を continue 文に置き換えた例です。

```
while ( 条件 A ) {
  文 S1
  if ( 条件 B )
    continue;
  文 S2
}
```



図は条件 B を確認して true の際に左に進みます。そして continue 文によりループの先頭に進みます。もし二重ループや三重ループで break 文や continue 文が指定された場合には、その文を囲む一番内側のループが対象となります。

break や continue はフローチャートの形を不規則にします。これはプログラムの流れを追うことを難しくすることを意味し、あまり良いことではありません。そのため、本当に必要かどうかを良く判断して使用するように心がけましょう。

逐次探索

次のプログラムは、vector<int>型の配列の中から特定の値を持った要素を探し、それが先頭から何番目であるかを表示しています。要素を見つけた際には残りの繰り返しを続ける必要がないので、break 文でループを抜けています。ループを抜けた後、見つけたかどうかは i の値で分かります。見つからない場合には、ループは i が a.size() になるまで繰り返されるためです。i がこの値未満の場合には見つかったことになります。この例のような配列の先頭から一つずつ探す方法を**逐次探索 (または線形探索)** と言います。逐次探索は汎用的な探し方のなので、配列以外のデータ構造でも適用できます。探索の詳細は後の章で見ることにしましょう。

ソースコード 32: 逐次探索

```
1  // 整数のリストの中から特定の要素を探す
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     std::vector a {3, 6, 2, 8, 1, 5, 2, 9, 3, 7}; // このデータ中から探す
7
8     // 逐次探索でデータを探す
9     const int x {5}; // この値の要素を探す
10    int i; // 変数i に結果をいれる
11    for (i = 0; i < a.size(); i++)
12       if (a[i] == x)
13          break;
14
15    // 結果の表示
16    std::cout << x <<" is "<< (i < a.size()?"":"not ")
17                  <<"found\n";
18    return 0;
19 }
```

4.7 size_type 型と size_t 型

これまでの例では、vector<T> v 内の要素数 v.size() の値を int 型の変数と比べて来ました。しかし、厳密には両者の型は異なるために、コンパイラに細かなチェックをさせると、以下のような警告が発せられます。

```
% g++ -Wall -std=c++17 v7.cpp
v7.cpp: In function 'int main()':
v7.cpp:21: 警告: 符合付きと符合無しの整数式同士の比較です
v7.cpp:22: 警告: 符合付きと符合無しの整数式同士の比較です
```

int は ± の符号付きの整数型で、v.size() の値は負の値を持たない符合無しの整数型です。v.size() の戻り値の型を正確に書けば、vector<T>::size_type です。符号無しの整数型は負の数を扱わない分、符号付きの整数よりも大きな整数値を扱うことができます。しかし、その異なる部分は非常に大きな値であるため、要素の添字として使われる可能性の低いものです。そこで、多くの場合に指定の簡単な int が使われます。この型の不一致は string クラスのメンバ関数 size() に対しても同様に起こります。string の size() の戻り値の正確な型は、string::size_type であるためです。

以下は、コンパイラに警告を出させないように正確に型を書いたプログラムです。

ソースコード 33: size_type の使用

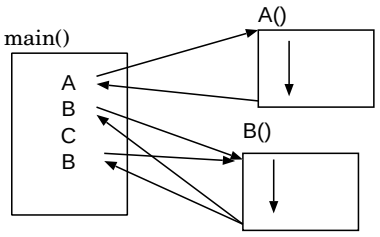
```
1  // string や vector の添字に使う型とsize()の戻り値の型は符号なし整数
2  #include <iostream>
3  #include <string>
4  #include <vector>
5  using std::cout, std::string, std::vector;
6  int main()
7  {
8     vector<double> v {1.2, 2.4};
9
10    // 厳密には添字に int でなく以下を使うのが正しい
11    // vector<double>::size_type は v.size()の戻りの型でもある
12    for (vector<double>::size_type i = 0; i < v.size(); i++)
13       cout << v[i] <<"\n";
14
15    // string も同様に size()の戻りの型は string::size_type が正しい
16    string s {"Hello World!"};
17    for (string::size_type i = 0; i < s.size(); i++)
18       cout << s[i] <<"\n";
19
20    // size_t で代用しても問題はない
21    for (size_t i = 0; i < v.size(); i++)
22       cout << v[i] <<"\n";
23
24    return 0;
25 }
```

他のプログラマに使ってもらうようなプログラムでは、このように型を正確に指定することが望まれます。しかし、省略をしなければ、std::vector<double>::size_type ととても長くなり、これをループのたびに入力するのは面倒です。そこでよく使われるのが size_t です。これは型名の別名として用意されており、vector や string の size_type の代用として使えます。

5　関数定義と変数のスコープ

長い文章を節や章などで区切って読みやすくするように、行数の多い長いプログラムも関数と呼ばれる単位に区切って読みやすくできます。関数とは一連の処理のまとまりに名前をつけたものです。関数の名前をうまく使うと処理の詳細を考えずに概要だけを追えるようなプログラムが書けます。また、似たような処理をまとめて一つの関数として表すことで、プログラムを簡潔にしたり、再利用可能な一般的な関数を作ることもできます。

例えば A, B, C という三つのまとまった処理があり、これを A, B, C, B という順に実行する場合を考えてみます。A と B に対する処理を関数 A と関数 B と分けてプログラムを記述し、C に対する処理はライブラリ関数 C を利用することにすれば、右の図が示すようなプログラムのイメージとなります。`main()` 関数には、関数 A(), 関数 B(), 関数 C() の呼び出しを指定するのみです。プログラムの概要はこれだけではっきりします。関数 A と関数 B の詳細はそれぞれを見れば良く、更新管理も関数ごとに行えます。関数 C の中身は見ることができませんが、信用のおけるライブラリ関数を使用しているならば、登録されている関数の詳細を知らなくても良い場合がほとんどです。



5.1　関数定義

`main()` 関数などから呼び出すことのできる関数を定義するには、以下の形式で記述します。

戻り値の型　関数名 (仮引数の宣言のリスト) 複文

ここで、**戻り値の型**とは関数の中で何らかの処理をして最後に戻す値に対応する型のことで、組み込み型やユーザ定義型などを指定します。**仮引数**とは、定義する関数が呼ばれたときに値が設定される変数のことで、関数の中でだけ有効な変数です。仮引数に対して、関数呼び出し側で指定する引数を**実引数**と呼びます。

次のプログラムは、関数 `sumup` を定義しています。これは、引数に指定された `from` から `to` までの整数値の合計を関数の結果として返します。

ソースコード 34: 関数定義の例

```
1  // 合計を求める関数を使用した例
2  #include <iostream>
3  // from から to までの整数値の合計を求める関数の定義
4  int sumup(int from, int to)
5  {
6     int sum {0};
7     for (int i = from; i <= to; i++)
8         sum += i;
9     return sum;
10 }
11
12 int main()
13 {
14    int s1;
15    s1 = sumup(1, 10); // 関数呼び出しの結果を代入
16    std::cout << s1 << "\n";
17    int s2 {sumup(s1, s1+10)}; // 関数呼び出しの結果で初期化
18    std::cout << s2 << "\n";
```

```
19     std::cout << sumup(s2, s2+10) << "\n"; // 関数呼び出しの結果を直接出力
20     return 0;
21 }
```

sumup 関数の定義では、仮引数として int 型の from と to を宣言しています。これらの初期値は sumup 関数が呼び出されたときに決まります。実引数の値が仮引数に渡されるので、この方法を**値渡し**と呼びます。これは=演算子による代入と同じです。そして、この関数内で宣言した変数 sum に合計値を計算します。最後に、return sum; とすることで変数 sum の値を戻り値とします。

main 関数では、この関数を 3 箇所で呼び出しています。最初は、変数 s1 に代入する値を得るためで、sumup(1, 10) として呼び出しています。関数の戻り値が結果の値として代入されます。次は、変数宣言の初期値を指定するためで、sumup(s1, s1+10) として呼び出しています。その次は、関数呼び出しの結果をそのまま cout で出力しています。この例が示すように、結果を返す関数呼び出しは式の中の変数と同じ場所で指定できます。

5.2 void 関数

通常の関数は引数を指定して結果を返すという形で定義しますが、結果を返さない関数も定義できます。例えば出力のみを行う関数です。そのような関数では戻り値の型として void を指定します。これを void 関数と呼ばれます。void は無効という意味で契約書などで使われる英単語です。プログラム例を見てみましょう。

ソースコード 35: void 関数の例

```
1  // 整数ペアの出力
2  #include <iostream>
3
4  // 引数をプリントする関数の定義，結果を返さないのでvoid
5  void print(int x, int y)
6  {
7     std::cout <<"("<< x <<", "<< y <<")"<<" ";
8  }
9
10 // さまざま引数でprint()を呼び出してテストする
11 int main()
12 {
13    for (int a = 1; a <= 5; a++) {
14       for (int b = 6; b <= 8; b++)
15          print(a, b);
16       std::cout <<"\n";
17    }
18    return 0;
19 }
```

このプログラムの print 関数は、引数として与えられた x と y の座標の値に括弧をつけて出力します。特に何かを計算するのではないために戻り値がありません。したがって、戻り値の型に void を指定しています。このプログラムを実行すると、以下のような出力を得ます。

```
(1, 6) (1, 7) (1, 8)
(2, 6) (2, 7) (2, 8)
(3, 6) (3, 7) (3, 8)
(4, 6) (4, 7) (4, 8)
(5, 6) (5, 7) (5, 8)
```

void 関数では呼び出し元に戻るという意味で値を伴わない return 文が指定できます。これの用途は関数を途中で終了させることです。例えば、以下の関数は cout で値を出力するものですが、引数の値が 0 の場合には何もせず途中で呼び出し元へ戻ります。

```
void print2(int x)
{
  if (x == 0) return;
  std::cout << x <<"\n";
}
```

値を返す関数でも呼び出し側で結果を無視することができます。関数を呼び出すだけで結果を代入や演算に使わなければ良いのです。実際に、C 言語ができた当時には void 指定はなく、戻り値の型指定を省略することもできました。省略した場合の戻り値の型は int 型であるという暗黙のルールもあり、複数人でプログラムを作る場合に誤解を生むケースもあったようです。関数は呼び出す側と呼び出される側にある種の約束が必要です。void の指定は、その約束をはっきりとさせる要素の一つなのです。

5.3 関数の引数

関数の呼び出しにおける約束を決めるもう一つの要素が関数の引数と呼ばれる変数です。引数は 0 個でも複数個の指定でも良いことになっています。次のプログラムはさまざまな個数の引数の例を示しています。

ソースコード 36: 引数の例

```
1  // 引数の指定方法の例
2  #include <iostream>
3  #include <vector>
4  #include <cmath>
5  int zero() { return 0; } // 引数をとらない関数
6  int one(int x) { return x+1; } // 1個のint 型引数の関数
7  int two(int x, int y) { return x+y; } // 2個のint 型引数の関数
8  double dist(double x, double y) { return sqrt(x*x + y*y); } // 2個のdouble 型引数の関数
9
10 // 型の異なる引数をとる関数
11 int mix(int x, double d)
12 {
13    if (d < 0) return 0;
14    return x+1;
15 }
16
17 int main()
18 {
19    int x{3}, a{}, b{}, c{};
20    a = zero();
21    b = one(10);
22    c = two(x+3, 4);
23    std::cout << a <<" "<< b <<" "<< c <<" "
24              << dist(5.3, 6.3) << '\n';
25
26    std::vector da {3.4, 4.3, -1.0, -3.0};
27    std::vector ia {3, 4, -1, -3, 2};
28    size_t n { std::min(da.size(), ia.size()) };
29    for (size_t i = 0; i < n; i++)
30        std::cout << mix(ia[i], da[i]) <<"\n";
31    return 0;
32 }
```

関数 zero() は引数を持ちません。このような場合でも空の引数を示す丸括弧の対 () を指定しなければなりません。また、関数 two() や dist() のように同一の型の仮引数が並ぶ場合でも、それぞれの引数に型を指定しなければなりません。これらの呼び出し側の実引数には、リテラル定数・単純な変数・配列の要素・それらの式などが指定できます。ただし、仮引数 y と実引数 x とでは、双方の型が y = x; という代入が可能な型である必要があります。

このプログラムでは、30 行目に変数 n の初期値として、std::min() というライブラリ関数を使っています。これは二つの実引数のうち小さい方を返します。vector 変数 da と ia を for 文で指定してい

るのですが、二つの配列は要素数が違います。繰り返しの回数を小さい方にあわせる処理として、この関数を使っています。

5.4 リファレンス引数

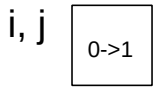
これまで見てきた引数の例は、関数の呼び出し時に実引数の**値**を仮引数の初期値として設定していました。実引数は仮引数の初期値として使われるだけなので、仮引数の値を変更しても、呼び出し側の実引数の変数には何ら影響はありません。呼び出された関数側で実引数の変数の値を変更するには、**リファレンス**という機能を用います。リファレンスとはある変数の別名となるものです。実体が 1 個で複数の名前がついた変数が作られます。関数の仮引数をリファレンスにすると、実引数として渡された変数を直接に操作できます。

lvalue リファレンス

まず、リファレンスによる変数宣言の例について見てみます。以下のプログラムの変数 j の宣言では、変数名の前に & （アンパサンドと読みます）記号が付いています。これがリファレンス指定です。& の両側のスペースはあってもなくても構いません。これで変数 j は初期値として指定された変数 i の別名となります。別名をつけることが目的なので対象は lvalue（左辺値）でなければなりません。lvalue とは変数のようにメモリの場所が関連するものです。リテラル値のような rvalue（右辺値）は指定できません。イメージとしては以下の図のように変数に 2 個の名前がついたと考えてください。i と j の実体は同一ですから、++j; によって変数 i の値も変更されます。この例のようにリファレンス指定の変数は、必ず初期値として他の変数を指定しなければなりません。また後で他の変数の別名となるようなつけ変えもできません。

ソースコード 37: lvalue リファレンスの例

```
1  // lvalue リファレンスの例
2  #include <iostream>
3  int main()
4  {
5     int i {0};
6     int& j {i}; // j はi の別名になる(つまり実体は同じ)
7     ++ j; // i も更新される
8     std::cout << i <<" "<< j <<"\n"; // 1 1 と出力
9     return 0;
10 }
```

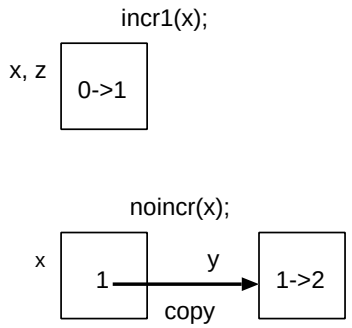


リファレンス引数

リファレンス指定の仮引数について見てみます。次のプログラムにおいて、4 行目の int& z がリファレンスによる仮引数です。仮引数の場合には、関数呼び出し時の実引数の指定が初期値となります。この例では、関数 main() で incr1() を呼び出すときに、実引数として x や n を指定しています。そのため関数 incr1() では main() で宣言した x や n を直接に更新できます。リファレンスではない仮引数をとる noincr() と比べてみましょう。noincr() の仮引数 y は、この関数に局所的な変数です。したがって、この変数 y の値を更新しても実引数には影響がありません。

ソースコード 38: リファレンス引数の例

```
1  // リファレンス引数の例
2  #include <iostream>
3  // 引数の値を+1する
4  void incr1(int& z) // リファレンス引数は実引数の別名
5  {
6     ++ z; // 実引数として指定された変数を直接更新
7  }
8
9  // 値渡しの引数を持った関数（比較用）
10 void noincr(int y) // 値渡しの y はnoincr()に局所的
11 {
12    ++ y; // 更新は仮引数のみ、実引数には無関係
13 }
14
15 int main()
16 {
17    int x{0}, n{10};
18    incr1(x); // x は関数呼び出しによって変化する
19    incr1(n); // n は関数呼び出しによって変化する
20    std::cout << x <<" "<< n <<"\n";
21    // incr1(2); // エラー:整数リテラルは指定できない
22    noincr(x); // x 関数呼び出しによって変化しない
23    std::cout << x <<"\n";
24    return 0;
25 }
```



図にすると右側のようになります。`incr1(x);` という呼び出しでは、関数 `incr1()` が `main()` の変数 `x` を `z` という名前で、そのまま使います。`incr1()` では`++ z` としているので、対応する `x` や `n` も変化します。`noincr(x);` という呼び出しでは、変数 `x` の値が関数 `noincr()` の仮引数 `y` に代入されます。変数 `x` と `y` は別物なので、`y` を更新しても `x` の値は変化しません。

`&` 記号は複数の用途で使われるので注意が必要です。この記号を変数宣言のところで使うとファレンスを意味しますが、式中で使うと演算子を意味するようになります。また、リファレンスは変数の別名を作る機能なので、コメントにあるように、関数 `incr1()` の実引数に整数リテラルを指定できないことにも注意しましょう。

関数の呼び出し時に値を使った引数の渡し方を**値渡し** (Pass by value) と呼びます。一方で、リファレンスを使った引数の渡し方を、**参照渡し** (Pass by reference) と呼びます。値渡しと参照渡しは引数の渡し方に関する重要な用語です。

変数の入れ替え関数

以下のプログラムはリファレンス引数を効果的に用いる関数を示しています。値を入れ替える関数の目的は、二つの実引数の値を変更することです。そのため関数側でリファレンス引数を用いています。入れ替えの操作はよく行われるのでライブラリ関数が用意されています。`std::swap()` 関数は整数に限らずさまざまな型の変数の入れ替えに指定できます。

ソースコード 39: 入れ替え

```
1  // リファレンスを使ったswap 関数
2  #include <iostream>
3  void swap_int(int& a, int& b)
4  {
5     int t {a};
6     a = b;
7     b = t;
8  }
9
10 int main()
11 {
```

```
12      int x{3}, y{5}; // 最初の状態を出力
13      std::cout << x <<", "<< y <<"\n";
14      swap_int(x, y); // 値の入れ替え
15      std::cout << x <<", "<< y <<"\n"; // 結果を出力
16      std::swap(x, y); // ライブラリ関数での値の入れ替え
17      std::cout << x <<", "<< y <<"\n"; // 結果を出力
18      return 0;
19  }
```

vector と引数

vector の変数を関数の引数する場合も int 型や double 型の変数と同様に指定できますが、引数の型の指定方法に注意が必要です。次のプログラムでは、関数を使って、vector<int>の要素の合計を求めています。sum_value() と sum_ref() は引数の型指定が異なります。vector は代入によってすべての要素のコピーが作られます。これと同様に sum_value() のような値渡しの引数指定では、関数の処理が始まる前に実引数 v のすべての要素が a にコピーされます。この不要なコピーは要素数が数千個や数万個となったときにプログラムの実行時間を長くします。一方で、sum_ref() ではリファレンス引数を使用しているので、実引数 v と仮引数 a が同一となり、要素はコピーされません。

ソースコード 40: vector 引数の例

```
1   // vector と引数
2   #include <iostream>
3   #include <vector>
4
5   // 値渡しの場合には全要素がコピーされる：効率が悪い
6   int sum_value(std::vector<int> a)
7   {
8      int s {0};
9      for (int i : a)
10        s += i;
11     return s;
12  }
13
14  // リファレンス引数では要素はコピーされない：効率が良い
15  int sum_ref(const std::vector<int>& a)
16  {
17     int s {0};
18     for (int i : a)
19        s += i;
20     return s;
21  }
22
23  int main()
24  {
25     std::vector v { 1, 2, 4, 7, 14 };
26     std::cout << sum_value(v) << "\n";
27     std::cout << sum_ref(v) << "\n";
28     return 0;
29  }
```

vector を引数にする際のリファレンス指定は不要なコピーを避けられるので良いのですが、リファレンスでは仮引数の内容を変更することで実引数も変更されてしまいます。そしてこのうっかりミスは起こりがちです。そこで実引数の変更がなされないように、仮引数に const を指定しています。一般的に、vector<T>の変数を関数 f() で処理する場合には、次のどちらかの形式で引数を指定すると良いでしょう。

```
void f(const vector<T>& a);  // 実引数に変更を加えない場合
void f(vector<T>& a);        // 実引数に変更を加える場合
```

5.5 main 関数の仮引数

これまで main 関数は int main(){ return 0; }という形で、引数なしで int の値を返す関数としてきました。戻り値はプログラムの呼び出した側で使われる値で、通常は、return 0; がプログラムの正常終了を表し、それ以外の数値が失敗したことを表すのに使われます。プログラムを実行する OS によっては、main 関数に引数を指定できます。その場合の main 関数の引数は、プログラムのファイル名とコマンド引数です。コマンド引数とはコマンドとなるプログラムを実行をする際に一緒に指定される文字列です。

コマンド引数を受けとるには main 関数を以下のように宣言します。

```
int main(int argc, char *argv[]) { ...省略... }
```

OS が呼び出し側の別のプログラムとの仲立ちとなって、これらの引数の設定と戻り値の回収を行います。変数となる argc と argv の名前は他に変えても良いのですが、argc が argument count を、argv が argument vector を意味しており、ほとんどのプログラムでこの変数名が使われています。この引数を使うには、C 言語固有の配列やポインタの知識が必要です。しかし、6 行目の方法で std::vector<std::string>として使えるようになります。

ソースコード 41: コマンド引数の例

```
1  // argv から vector への変換
2  #include <iostream>
3  #include <vector>
4  using std::vector, std::string, std::cout;
5  int main(int argc, char *argv[])
6  {
7     // argv を初期値としてを a に取り込む
8     vector<string> a(argv, argv+argc);
9
10    for (const string& s : a)
11       cout << s <<"\n";
12    return 0;
13 }
```

細かい説明は省略しますが、この vector 変数の初期化の仕方を覚えればコマンド引数がプログラム中で使えるようになります。実行例を以下に示します。この例で a.out がプログラムのファイル名です。ab や cd などの文字列がコマンド引数です。

```
$ ./a.out
./a.out
$ ./a.out ab cd
./a.out  ab  cd
$ ./a.out a b+c d:ef
./a.out  a  b+c  d:ef
```

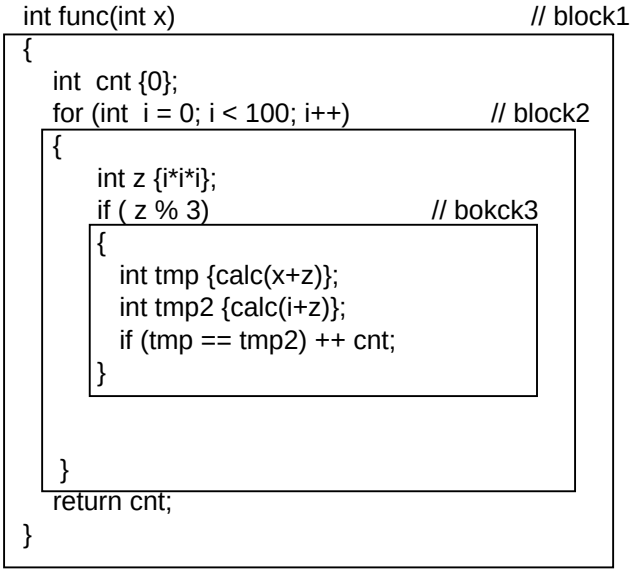
5.6 局所変数のスコープ

変数名や関数名などのプログラム中で使う名前は、それが何を示すかをコンパイラに明示するために宣言が必要です。宣言で使われた名前には、その名前を使うことのできる範囲が定められます。この範囲のことを**スコープ**と呼びます。関数の中で宣言された変数を**局所変数**と呼びます。局所変数のスコープは次のルールで決められています。

- 二つの中括弧 ({}) で囲まれた範囲を**ブロック**と呼びます。これ文法的には複文です。
- 関数の本体部分はブロックです。
- ブロックの中には局所変数を宣言できます。
- 局所変数のスコープは宣言された位置からその宣言を含むブロックの末尾までです。

- ブロックの内側には別のブロックを**入れ子**の形で指定でき、その内側のブロックにも変数を宣言できます。
- ブロック内で宣言された変数の名前が、その外側のブロックで宣言された変数と同じ名前の場合には、外側で宣言された変数を一時的に隠します。外側のブロックで宣言された変数にはアクセスできません。
- 以下はブロックからはずれた例外的なルールです。
  - 関数の仮引数は関数本体のブロックがそのスコープです。
  - `for` 文の初期設定で宣言された変数は、その `for` 文の末尾までがスコープです。
  - `if`・`while`・`switch` などの文の条件部で宣言された変数は、その文の末尾までがスコープです。

例:以下のプログラムで変数 `x` と `cnt` のスコープは block1、変数 `i` と `z` のスコープは block2、変数 `tmp` と `tmp2` のスコープは block3 です。



外側のブロックの変数を隠す例

次のプログラムは、ブロックを入れ子に指定して、その中にそれぞれ `x` という同じ名前の変数を宣言した例です。内側のブロックで宣言された変数が、外側で宣言された変数を一時的に隠して使えなくしますが、内側のブロックの末尾をすぎると、隠された変数が再び使えます。この例はブロックの特徴を示すために意図的に作ったプログラムです。名前を隠すルールがあるということを確認したら、本当に必要な時のみ利用し、それ以外ではそのようなプログラムを作らないように心がけましょう。

ソースコード 42: 入れ子のブロック

```
1  // 入れ子のブロック内の変数が外側のブロックの変数を隠す例
2  #include <iostream>
3  using std::cout;
4  void func()
5  {
6     int x {1}; // 宣言 a)
7     cout << x <<"\n";
8     if (x > 0) {
9         int x {2}; // 宣言b): 外側の宣言a) のx を一時的に隠す
10        cout << x <<"\n";
11    }
12    cout << x <<"\n"; // 一時的に隠されていた宣言a)のx が再び見える
13    if (x > 0) {
14        int x {3}; // 宣言 c):外側の宣言a)のx を一時的に隠す
```

```
        cout << x <<"\n";
        if (x >= 3) {
            int x {4}; // 宣言 d):外側の宣言c)のx を一時的に隠す
            cout << x <<"\n";
        }
        cout << x <<"\n"; // 一時的に隠されていた宣言c)のx が再び見える
    }
    cout << x <<"\n"; // 一時的に隠されていた宣言a)のx が再び見える
}

int main()
{
    func();
    return 0;
}
```

for 文の初期設定で宣言された変数

次のプログラムは、for 文の初期設定で変数を宣言した場合の特徴を調べるプログラムです。各 for 文にそれぞれ i，j という変数を宣言しています。それぞれの変数のスコープは、ループ本体が単純な文であっても、複文となるブロックであっても同じで、for 文の末尾までです。

ソースコード 43: for 文の変数宣言

```
// for 文の初期指定部で宣言された変数の例
#include <iostream>
int main()
{
    for (int i=0; i < 10; i++) // 変数i はfor 文の末尾までのスコープ
        std::cout << i << " ";
    // std::cout << i <<"\n"; // エラー: for 文の後では変数 i は使えない
    std::cout <<"\n";

    for (int j=0; j < 10; j++) { // ループ本体がブロックでも末尾までのスコープ
        std::cout << j+2 << " ";
    }
    std::cout <<"\n";
    return 0;
}
```

if 文や while 文の条件部で宣言された変数

次のプログラムは、if 文や while 文の条件部で変数を宣言した場合の例です。abs() は整数の絶対値を計算します。abs() の結果の値を除算に利用しているのですが、関数の結果が 0 の時には if 文の条件を満たさないので計算を行いません。この例のように、条件の中で行った計算結果に基づいて、変数の宣言と初期化を同時に行うことで、変数を if 文に局所的なものとできます。

string 変数 s は初期値に文字列を指定していますが、文字列の最後に'\0' を指定しています。これはヌル文字と呼ばれるもので、整数の値としては 0 です。while 文の例も同様に条件部で宣言した変数 ch はその while 文の末尾までが有効な範囲です。このループは文字列の最後にヌル文字があり、これが値の 0 であるという特徴を利用しています。

ソースコード 44: if 文・while 文の変数宣言

```
// if 文，while 文の条件で宣言された変数の例
#include <iostream>
#include <cstdlib> // 整数用のabs()を使うために必要
int main()
{
    int x{};
```

```
7     std::cin >> x;
8     if (int n = std::abs(x)) { // abs()は絶対値を計算
9        std::cout << 10/n <<"\n"; // n のスコープはif 文の末尾まで
10    }
11
12    std::string s {"abcdefg\0"};
13    int i {0};
14    while (char ch = s[i++]) { // ch = s[i]; i++
15       std::cout << ch <<" "; // ch のスコープはwhile 文の末尾まで
16    }
17    std::cout <<"\n";
18    return 0;
19 }
```

これらの指定は変数宣言の初期化と同時に条件を確認しているのですが、一つの変数しか指定できない上に、その初期値が 0 かどうかという条件しか指定できません。つまり、複雑な条件には対応できないのです。while 文は for 文に書き換えれば、初期化と条件指定を分離できるので良いのですが、if 文は対応できませんでした。C++17 からは if 文と swich 文で初期化と条件を分離して指定できるようになりました。書式は次のとおりです。

```
if（初期化つき変数宣言; 条件）

swtich（初期化つき変数宣言; 条件）
```

for 文と同じようにセミコロン (;) で区切って、初期化つきの変数宣言を指定します。宣言された変数のスコープは関係する if 文や switch 文の中だけとなります。これに加えて for 文と同じように変数宣言ではなく、代入文などの式文と呼ばれるものが指定できますが、あまり用途はないでしょう。次のプログラムはこの初期化つきの if 文や switch 文の例を示しています。

ソースコード 45: C++17 の if 文 swtich 文

```
1  // 初期化つきのif 文と switch 文
2  #include <iostream>
3  int main()
4  {
5     int x{}, y{};
6     std::cin >> x >> y;
7     if (int z {std::min(x, y)}; z > 10) {
8        std::cout << z << " is too large\n";
9     }
10
11    switch (int z {std::max(x, y)}; z*3) {
12    case 3:
13       std::cout << "three\n";
14       break;
15    case 6:
16       std::cout << "six\n";
17       break;
18    case 9:
19       std::cout << "nine\n";
20       break;
21    }
22    return 0;
23 }
```

ライブラリ関数の std::min() と std::max() を使って一時的な変数を初期化して、条件や実行文で使っています。

局所変数の用途とスコープの関係

局所変数の用途に応じてスコープを設定することはプログラムを読みやすくします。次の例は、ある特性を持った数を数える関数を二つ示しています。

ソースコード 46: 局所変数の活用

```
// 1から 100まででその 2乗の値が 3で割り切れる数の個数
#include <iostream>
// スコープをうまく利用した例: cnt が重要であることが明確
int count1()
{
   int cnt{0};
   for (int i = 1; i <= 100; i++) {
      if (int x {(i*i)%3}; x == 0)
         ++ cnt;
   }
   return cnt;
}

// 局所変数の用途が明確でない例
int count2()
{
   int i, x, cnt{0}; // 3個の変数が同等に宣言されている
   for (i = 1; i <= 100; i++) {
      x = (i*i) % 3;
      if (x == 0) ++ cnt;
   }
   return cnt;
}

int main()
{
   std::cout << count1() <<"\n";
   std::cout << count2() <<"\n";
}
```

どちらの関数も cnt という変数で数えて、その値を関数の結果として返しています。そして、i はループを制御するための変数で、x は一時的な計算結果を保持する変数です。

count1() と count2() の違いは、これらの変数を宣言している場所です。count1() では、cnt だけが最初に 0 の初期値で宣言され、return によってその変数の値が返されているので、全体を通して使われる変数であるということが明確です。また、i を for 文の初期設定で宣言することで、これをループ制御に利用することを明示しています。そして、x を if 文で宣言することで, 一時的な計算でのみ使うということがはっきりします。一方で、count2() では、3 個の変数が同等に宣言されており、そのうち i と x は初期値も分かりません。

たった 6 行のプログラムで変数の重要度の理解に差が付くのですから、これが 100 行を越えるような関数となった場合には、プログラムの読みやすさの差が大きく開きます。一般に読み難いプログラムは間違いを抱える可能性が高くなります。

同じ名前の局所変数

次のプログラムでは、同じ名前を持つ局所変数が複数の関数で使われています。

ソースコード 47: 同名の局所変数

```
// 局所変数の特徴を調べる
#include <iostream>
void func1()
{
    int x{}; // func1 に局所的な変数 x
    x = 1; // func の局所変数を更新
    std::cout <<"func1:"<< x <<"\n";
}

void func2(int x) // 関数の仮引数 x, func2 に局所的
{
    std::cout <<"func2:"<< x <<"\n";
    x = 2; // func2 に局所的な仮引数 x を更新
```

```
14 }
15
16 int func3(int x) // 関数の仮引数 x, func3 に局所的
17 {
18     std::cout <<"func3:"<< x <<"\n";
19     return x+1; // 戻り値
20 }
21
22 int main()
23 {
24     int x {0}; // main に局所的な変数 x
25     std::cout << x <<"\n";
26     func1();
27     std::cout << x <<"\n";
28     func2(x+1); // x+1は実引数
29     std::cout << x <<"\n";
30     x = func3(2); // 2 は実引数
31     std::cout << x <<"\n";
32     return 0;
33 }
```

関数 main() と func1() とでは、それぞれ局所変数 x を宣言しています。それぞれの局所変数は名前が同じだけでまったく関係がありません。関数 func2() は仮引数 x を宣言しています。main() では実引数として x+1 を使って func2() を呼び出しています。仮引数はブロックの外側に宣言されますが、関数の局所変数です。関数の中で宣言された他の局所変数と異なるところは、仮引数の初期値が関数呼び出しの実引数によって決まるという部分だけです。そのほかは他の局所変数と同じです。実引数の値を仮引数となる変数にコピーすることから関数 func3() では return を使って関数から値を戻しています。関数 main() も同じ形をしていることに注意して下さい。関数 main() の戻す値は、このプログラムを実行するシステムに戻されます。

実引数の値が仮引数に設定され、return で指定した値が関数の戻り値として呼び出し側に戻されるのは、関数を使って処理を記述する場合の基本パターンです。

演習

1. 仮引数と同じ名前の局所変数を作ってコンパイルしてみましょう。

5.7 大域変数

関数間の値の受け渡しは、実引数で渡して、戻り値で戻してもらうという方法の他に、関数間で共通に使える変数を利用する方法があります。このような変数を**大域変数**と呼びます。大域変数は関数の外側で宣言されます。一つのプログラムファイルの中では、大域変数として宣言された変数は、その宣言より下側で定義されたどの関数でも使用できます。

次のプログラムは sum・from・to という三つの大域変数を宣言して、関数 sumup() と main() の双方で利用しています。二つの関数での値の受け渡しに大域変数を利用しているので、関数 sumup() は引数なしの void 関数であることに注意してください。

ソースコード 48: 大域変数の例

```
1 // 合計を求める関数の例
2 #include <iostream>
3 int sum {0}; // 大域変数の宣言
4 int from, to;
5
6 // from から to までの整数値の合計を求める関数の定義
7 void sumup()
8 {
9    sum = 0; // 大域変数sum の利用
```

```
10      for (int i = from; i <= to; i++) // from と to も大域変数
11          sum += i;
12  }
13
14  int main()
15  {
16      from = 1; to = 10;
17      sumup(); // 呼び出しによりsum が変化する
18      std::cout << sum <<"\n"; // main()でもsum を利用できる
19      return 0;
20  }
```

このプログラムでは、関数 sumup() の i が局所変数です。大域変数はどの関数でも共通に参照や変更ができて手軽ですが、この例から分かるように、関数の役割が分かりづらくなります。さらに、大規模なプログラムではどの関数が大域変数を変更しているのかが把握しづらくなり誤りの元となります。そのためできる限り大域変数は使用しない方が良いと言われています。

大域変数の名前

多くのプログラムにおいて、局所変数には短めの名前がつけられるのに対して、大域変数には少し長めで用途のはっきり分かる名前がつけられます。これは局所変数の使用される範囲が限定的であるのに対して、大域変数が複数の関数の様々な場所で使用されるためです。x や i などの 1 文字の名前の変数を大域変数につけると局所変数と混同して誤ったプログラムを作ってしまうことになります。変数の名前のつけ方には工夫しましょう。

変数の初期値

変数の宣言において初期値を指定しておけばその値が設定されます。また、仮引数は関数が呼ばれる際に実引数によって初期値が決められます。しかし、それ以外の変数には次のルールが適用されます。

- 組み込み型の大域変数は値の指定がなければ実行開始までに 0 に初期化される
- 組み込み型の仮引数以外の局所変数の初期値はどのような値になるか分からない
- ユーザ定義型の変数 (例えば string 型の変数や cin, cout など) の初期値は型を定義する際に決められる。

次のプログラムは上記のルールを確認するものです。大域変数、仮引数、そして string 型の変数は決まった値を持ちますが、局所変数の l1, l2 はどのような値になるか分かりません。

ソースコード 49: 変数の初期値

```
1   // 変数の初期値を調べるプログラム
2   #include <iostream>
3   #include <string>
4   int globalint;
5   double globalreal;
6   std::string globalstr;
7
8   void func(int p1, double p2)
9   {
10      std::cout <<"parameters: "<< p1 <<", "<< p2 <<"\n";
11      int l1; // 値は不定
12      double l2; // 値は不定
13      std::string l3; // 空文字列で初期化
14      std::cout <<"local vars: "<< l1 <<", "<< l2 <<", "<< l3 <<"\n";
15  }
16
17  int main()
18  {
19      std::cout <<"global vars: "<< globalint <<", "<< globalreal <<
```

```
20                                          ", "<< globalstr <<"\n";
21      int l1; // 値は不定
22      double l2; // 値は不定
23      std::string l3; // 空文字列で初期化
24      std::cout <<"local vars: "<< l1 <<", "<< l2 <<", "<< l3 <<"\n";
25      func(12, 34.5);
26      return 0;
27  }
```

以下が実行例です。string 型の変数は空文字列で初期化されるようにユーザ定義型として作られているので、出力結果は表示されません。double 型の局所変数は非常に小さな値となっています。このように変数の初期化を行わないとプログラムの結果が不定となってしまうので、初期値についてはいつも気にして、できる限り指定しましょう。

```
global vars: 0, 0,
local vars: 0, 2.07404e-317,
parameters: 12, 34.5
local vars: 53, 1.12533e-312,
```

大域変数と局所変数との関係

次のプログラムは大域変数 x 宣言し、関数 func1()・func2()・func3() でそれぞれ x という名の変数を更新する例です。func1() は大域変数 x を更新しています。func2() は局所変数 x を宣言しているので、大域変数 x は一時的に隠されて、局所変数 x を更新します。func3() では同じく局所変数 x を宣言していますが、**スコープ解決演算子** (::) を用いることで隠されている大域変数を指定できます。入れ子のブロックの内側で宣言された変数と同名の外側ブロックの変数にアクセスする方法はありませんが、このように大域変数にはアクセスできます。

ソースコード 50: 大域変数へのアクセス

```
1   // 大域変数の特徴を調べる
2   #include <iostream>
3   int x {0}; // 大域変数 x 初期値 0
4
5   void func1()
6   {
7       x = 1; // 大域変数 x に代入
8   }
9
10  void func2()
11  {
12      int x{}; // func2 に局所的な変数 x
13      x = 2; // 局所変数 x に代入
14  }
15
16  void func3()
17  {
18      int x{}; // func3 に局所的な変数 x
19      ::x = 3; // 大域変数 x に代入
20  }
21
22  int main()
23  {
24      std::cout << x <<"\n";
25      func1();
26      std::cout << x <<"\n";
27      func2();
28      std::cout << x <<"\n";
29      func3();
30      std::cout << x <<"\n";
31      return 0;
32  }
```

5.8 ※標準ライブラリと名前空間 std

これまでのプログラムでは、標準入出力ストリーム用の std::cin や std::cout を大域変数のように使っていました。cin や cout は、クラスのオブジェクトなので変数ですが正確には大域変数ではありません。

C++では**名前空間**という名前にスコープを与えるメカニズムがあります。局所変数が関数内のブロックによってスコープが与えられるのと同じように、名前空間によって関数の外側で宣言された変数などの名前にスコープを与えられるのです。cin や cout は、標準ライブラリと呼ばれる事前に用意された C++ プログラムの中で宣言されています。標準ライブラリで宣言される名前は名前空間 std に入れられています。using の指定をすると、その指定したスコープにおいて、名前空間 std に属する名前を省略名で使えます。

次のプログラムは、std 名前空間を関数の中にすべて取り込む場合と、一部の名前だけを取り込んだ場合の違いを確認しています。プログラムの先頭で using を指定しないで、cin という大域変数を宣言して、関数 f1() の中で使っています。f1() ではこの大域変数の cin と局所変数の cout を使っています。関数 f2() では、using ディレクティブを使って、std 全体を取り込んでいますが、cin が int 用なのか入力用なのか区別が付かないため、単に cin と書くとエラーになります。関数 main() のように、using 宣言を使って個別指定すると、指定した cin と cout が優先されます。int の cin にはやはりスコープ解決演算子が必要です。

ソースコード 51: using の特徴

```
1  // using の特徴を調べる
2  #include <iostream>
3  int cin {10}; // int 型の大域変数 cin を宣言
4  int f1()
5  {
6     int cout {2}; // 局所変数のcout
7     return cin + cout; // 大域変数のcin を加える
8  }
9
10 int f2(int n)
11 {
12    using namespace std; // std 空間をこの関数に取り込む
13    std::cin >> ::cin; // std::と::はどうちらも必要
14    return ::cin+n+f1();
15 }
16
17 int main()
18 {
19    using std::cin, std::cout; // このスコープでこれらの名前を使う
20    cout << "input 2 values:\n"; // cin と cout はそのまま使えるが、
21    cin >> ::cin; // int の cin には::演算子が必要
22    cout << f2(::cin);
23    return 0;
24 }
```

6　多次元配列

絵のような 2 次元の情報は、一つの添字でデータの場所を特定するよりも、縦横の二つの値で点の情報を指定した方が扱い易くなります。さらに動画のような情報は、縦横に加えて時刻やフレーム番号といった三つ目の情報を使う場合もあるでしょう。このような 2 次元や 3 次元といった多次元の情報は、一般に多次元の配列で表します。そして C++では、「配列を要素とする配列」という形でこの多次元配列を表現します。特に、**配列の配列**を 2 **次元配列**と呼び、$n \times m$ の表の情報の処理に利用します。これに対して、これまで扱ってきた配列を 1 **次元配列**と呼びます。

6.1　2 次元配列の宣言と要素へのアクセス

vector 型は要素となる型を指定することで、int 型や double 型の配列を作れました。2 次元の配列を作るには、vector<int>や vector<double>を配列の型として指定します。一般には、以下のように vector<T>を型として指定することで、T 型の 2 次元の配列の変数を宣言します。

```
std::vector<std::vector<T>> x;
```

これは std::vector<T>という配列を要素とする配列という形で指定されています。注意すべき点は、この宣言による配列の要素数です。この宣言では変数 x は 0 個の要素を持つ配列であり、その要素となる T 型の配列も要素数は 0 個です。したがって、T 型のデータを保存するには、まず、2 次元配列の下位の次元となる 1 次元の配列が必要です。次のプログラムで具体的に例を見てみましょう。

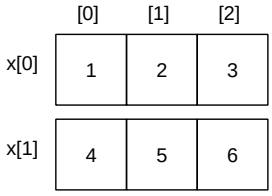
ソースコード 52: 2 次元配列の例

```
1  // 2x3 の 2 次元配列
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6      using std::cout, std::vector;
7      vector<vector<int>> x;
8      vector a{1,2,3}, b{4,5,6};
9      x.push_back(a);
10     x.push_back(b);
11     cout << x[0][0] <<" "<< x[0][1] <<" "<< x[0][2] <<"\n"; // 1 2 3
12     cout << x[1][0] <<" "<< x[1][1] <<" "<< x[1][2] <<"\n"; // 4 5 6
13     return 0;
14 }
```

プログラムは std::の繰り返しを避けて見やすくするために、using 宣言を使っています。これ以降の例も省略するので、この using 宣言が使用されていると仮定してください。

7 行目の x の宣言では要素数は 0 個です。8 行目ではこれとは別に a，b の 3 要素の 1 次元配列を用意しています。9, 10 行目でこの 1 次元配列 a，b を x に 2 個追加しています。push_back() は内容の複製なので、この後 a，b と x は無関係となりますが、これで x が 2×3 の 2 次元配列となります。

変数 x は以下の図のように 2 個の配列を持っています。その要素は vector<int>型でそれぞれ x[0] と x[1] でアクセスできます。さらに、これらそれぞれは int 型の vector 配列なので添字を加えて x[0][0] の形で整数にアクセスできます。この指定の仕方はこの図のように 2 次元の表での指定と同じです。つまり、x[0][1] は 2 で、x[1][2] は 6 です。



そして x[i][j] は左辺値として指定できるので、値の代入や入力に使用できます。

```
x[0][0] = 10;  // 先頭要素を変更
std::cin >> x[0][1]; // 2番目の要素に入力
```

6.2 2 次元配列の初期化

データが初めから決まっている場合には、前の例のように 1 次元配列を用意する必要はありません。初期値を指定することで指定個数の 2 次元配列を作れます。2 次元配列の初期化は次のように書きます。

```
vector<vector<int>> y {{3,4,5,6,7},{5,6,7,8,9},{7,8,9,10,11}};
```

この記述によって 3×5 の 2 次元配列の変数宣言とともに以下の初期値が設定されます。

	[0]	[1]	[2]	[3]	[4]
y[0]	3	4	5	6	7
y[1]	5	6	7	8	9
y[2]	7	8	9	10	11

この初期化の記述方法は、{3,4,5,6,7}のような 1 次元の配列の初期値を複数個並べています。変数ではないので名前はありませんが、この記述方法からも 2 次元配列は配列の配列であることが分かります。

次に、各要素の初期値がすべて同じ値の場合を考えます。要素数が 50×100 などの場合には、たとえ初期値がすべて 1 あっても上記の書き方は大変です。その場合の変数宣言はやや厄介ですが以下のように指定できます。

```
vector<vector<int>> z(50, vector<int>(100, 1));
または
vector z(50, vector(100, 1));  // 省略形
```

vector の初期化には丸括弧と中括弧の 2 種類があることを思い出しましょう。丸括弧を使って vector<int> a(50, 2) とすると、50 要素で初期値がすべて 2 の整数配列の変数 a が宣言できました。上記はこの初期値の 2 の部分に配列を指定した形になっています。vector<int>(100, 1) というような変数名を省略した形は、名前なしの一時的な変数と考えることができ、100 要素で値 1 の配列を指定したことになります。下側の書き方は、コンパイラに型を推測させた方法で、int 型のリテラル 1 によって、vector(100,1) は vector<int>型であり、z はそれを要素とする vector<vector<int>>であると判断されます。なお、初期値が 0 の場合には、vector<int> a(50) のような指定ができるので、これを使う場合には省略方法がやや変わります。

```
vector z(50, vector<int>(100));  // 別の省略形
```

6.3 3 次元配列

3 次元以上の配列の宣言も、1 次元配列を 2 次元配列に拡張したのと同様に考えられます。例えば、

```
vector xx(4, vector(5, vector(2, 0)));
```

と指定した場合には、変数の型は vector<vector<vector<int>>>で、3 **次元配列**の宣言です。この変数 xx は、この段階で、$4 \times 5 \times 2$ 要素の 3 次元配列、xx[i] は 5×2 要素の 2 次元配列, xx[i][j] は 2 要素の

1 次元配列です。そして、xx[i][j][k] が 1 個の int 変数です。それぞれの次元の配列は push_back() などで要素数を変更できる点に注意してください。

初期値を指定して宣言する場合には、以下のように三重の入れ子の形に書きます。

```
vector<vector<vector<int>>> xx {
       {{1,2}, {2,3}, {3,4}, {5,6}, {6,7}},
       {{2,7}, {3,8}, {4,2}, {6,9}, {7,3}},
       {{3,4}, {4,2}, {5,1}, {7,3}, {8,8}},
       {{7,3}, {0,8}, {9,6}, {1,4}, {7,2}}
};
```

先頭の要素に値を代入するには xx[0][0][0] = 10; とし、一番後ろの要素を出力したい場合には cout << xx[3][4][1]; と書きます。

確認

- 上記の 3 次元配列で xx[1][3][1] の値は何でしょうか？

6.4 多次元配列とループの関係

多次元配列の処理では配列の中のある要素の取り出しが基本です。1 次元配列とは異なり複数の添字によって要素を特定するために、すべての要素を処理するにはたいがい二重や三重のループが必要となります。この節ではループとの関係をいくつの例で確認しましょう。

2 次元配列の例

次のプログラムは 2 次元配列の初期化と出力の基本的な例を示しています。

ソースコード 53: 2 次元配列とループ

```
1  // 2次元配列の初期化と出力
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     using std::cout, std::vector;
7     vector<vector<double>> a {
8           {2.3, 8.4, 4.3, 1.2},
9           {8.2, 1.3, 8.1, 7.5},
10          {3.3, 3.1, 9.8, 5.3} };
11
12    for (size_t i = 0; i < a.size(); i++) {
13       for (size_t j = 0; j < a[i].size(); j++)
14          cout << a[i][j] <<" ";
15       cout <<"\n";
16    }
17    return 0;
18 }
```

変数 a は 2 次元配列ですが、1 次元配列を要素に持つ vector なので、その要素数は a.size() で分かります。そして、a[i] も vector なので、a[i].size() として要素数を調べることができます。

実行例は次のようになります。

```
2.3 8.4 4.3 1.2
8.2 1.3 8.1 7.5
3.3 3.1 9.8 5.3
```

次のプログラムは 2 次元配列の入力と出力の基本的な例を示しています。前の例とは異なり、配列の次元数が数値として指定できるので、それを const 変数の n と m を使って表しています。この変数を用意しておくと、ループで現れる size() を省略したり、ループ制御変数の型を int としてもまったく問題がなくなります。変数 n や m を使わずに 3 や 4 を変数宣言や for 文に書いてもプログラムの意味は変わりません。しかし、この数字が変わった場合のプログラム修正が大変になり、誤りの原因となります。条件が変わってプログラム修正が必要となるのはよくあることなので、それを事前に考えておくことは良いことです。この他に後半の出力の方法にも注意してください。各数値の間にカンマを出力していますが、最後の数値の後ろにはカンマがありません。最後の要素を特別扱いする方法もありますが、このプログラムでは行の最初の要素を特別扱いしています。

ソースコード 54: 2 次元配列に対する入出力の基本

```
// 2次元配列の入力と出力の例
#include <iostream>
#include <vector>
int main()
{
   using std::cout, std::vector;
   const int n{3}, m{4};
   vector b(n, vector<int>(m));
   // 入力
   for (int i = 0; i < n; i++)
      for (int j = 0; j < m; j++)
         std::cin >> b[i][j];

   // 出力
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         if (j != 0) cout <<", ";
         cout << b[i][j];
      }
      cout <<"\n";
   }
   return 0;
}
```

実行例は次のようになります。

```
3 4 5 6  7 8 9 10 1 2 3 4          <<--- これを入力
3, 4, 5, 6
7, 8, 9, 10
1, 2, 3, 4
```

範囲 for 文と auto

vector のような複数の値を持つデータ型には範囲 for 文が有用なことがあります。

ソースコード 55: 2 次元配列と範囲 for 文

```
1  // 範囲for 文の二重ループ
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     using std::cout, std::vector;
7     vector<vector<double>> a {
8           {2.3, 8.4, 4.3, 1.2},
9           {8.2, 1.3, 8.1, 7.5},
10          {3.3, 3.1, 9.8, 5.3} };
11
12    // 範囲for 文の利用
13    for (const vector<double>& x : a) {
14       for (double y : x)
15          cout << y <<" ";
16       cout <<"\n";
17    }
18    cout <<"\n";
19    return 0;
20 }
```

2 次元配列に対して、範囲 for 文を使う場合は、要素を取り出しながら繰り返しを続けるイメージになります。2 次元配列 a の中から 1 次元配列を x に一つずつ取り出しながら繰り返し、内側のループでは x の中から double 型の要素を y に取り出して出力しています。しかし、型指定の記述が面倒です。

このような型指定の面倒さは auto を使って解決できることがあります。auto によるコンパイラの型推定は C++11 から使用できるようになりました。注意点はあるものの、いろいろなことが簡潔に記述できるようになっています。以下が auto の基本的な使い方を示しています。

ソースコード 56: auto の例

```
1  // auto の基本的な使い方
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     using std::cout, std::vector;
7     auto i {1}; // int
8     auto d {2.0}; // double
9     cout << i <<" "<< d <<"\n";
10
11    auto v {vector(3,1)}; // vector<int>
12    for (auto x : v) // int
13       cout << x <<"\n";
14
15    auto t {vector(3, vector(4,2.0))}; // vector<vector<double>>
16    for (const auto& x : t) { // const vector<double>&
17       for (auto y : x) // double
18          cout << y <<" ";
19       cout <<"\n";
20    }
21 }
```

変数 i や d は int や double とは書かずに変数宣言ができています。変数宣言の初期値によって、型が明確になるので auto の指定でコンパイラに型の決定を任せているのです。これは vector<T>型の変数宣言において、初期値から T 型を推定できることを利用して T の指定を省略するのと同じです。int や double ではあまり変化がありませんが、vector の場合に違いが現れています。11 行目や 15 行目の変数宣言では v や t が変数名として目立っています。これまでの例で見てきた vector 変数の宣言では、どうしても変数名が他の文字列に埋もれがちですが、auto を使うことで、変数宣言を auto+変数名+初期値 と統一できるのではっきりするのです。

そして、auto が本領を発揮するのは複雑な型を省略できる場合です。12 行目のような範囲 for 文の場合には取り出す要素の型が明らかです。そのため auto を使ってプログラムを簡潔に記述できます。ただし、範囲 for 文はデフォルトで要素を取り出した際に値をコピーするために注意が必要です。この例では、12 行目の範囲 for 文では int 型の値をコピーしながら繰り返すので問題はありませんが、16 行目の場合は、もし for (auto x: t) と書いた場合には、毎回 vector<double>をコピーするために、わずかですが実行時に無駄が生じます。内側の範囲 for 文では x の要素は double です。一つの基本データ型の値はコピーをしてもそれが時間の無駄にはつながりません。そのため、for (auto ...) と外側とは別の書き方をしています。プログラムを読む人に対して、効率について注意していることを目立たせることにもなっているのです。

この例では基本機能を示すためにあえて使っていますが、基本データ型の変数宣言などでやみくもに auto 指定することはお薦めしません。使わない方が良いという意見もあります。それはあまりに単純なところに auto を使うとプログラムが分かりづらくなるからです。効果的な場所での auto の使用を心がけましょう。

3 次元配列の例

次のプログラムは int 型の 3 次元配列のすべての要素を調べて、負の値を持つ要素の数を出力します。三重のループで変数の i・j・k が、どのように変化するかを確認してください。

ソースコード 57: 3 次元配列の処理

```
1  // ３次元配列中の負の要素の数を数える
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     using std::vector;
7     vector<vector<vector<int>>> a {
8           {{0, -1, 2, 3}, {-4, 5, 6, 7}, {8, -9, 10, 11}},
9           {{0, 1, -2, 3}, { 4, 5, -6, 7}, {8, 9, -10, 11}}
10       };
11    // 負の要素を数える
12    int n {0};
13    for (size_t i = 0; i < a.size(); i++)
14       for (size_t j = 0; j < a[i].size(); j++)
15          for (size_t k = 0; k < a[i][j].size(); k++)
16             if (a[i][j][k] < 0) ++ n;
17    std::cout << n <<"\n"; // 6
18
19    // 同じことを範囲for 文で行う
20    n = 0;
21    for (const auto& x : a)
22       for (const auto& y : x)
23          for (auto z : y)
24             if (z < 0) ++ n;
25    std::cout << n <<"\n"; // 6
26    return 0;
27 }
```

6.5 行列を扱うプログラム

行列の入力と出力

$n \times m$ の行列を読み込み、読み込んだ行列とその転置行列の双方を出力するプログラムを示します。auto を使って行列に対応する変数名を目立たせて宣言してみました。以前の例で見たように、ここでも const 指定の変数 n と m を使って行と列の個数を指定しています。std::cin は入力であることを目立たせるために、あえて using 宣言を使わずに指定しました。行列の処理としては、転置を作るために二重になった for 文の内側と外側を入れ換えています。i と j の添字の扱いを確認してください。

ソースコード 58: 行列の処理

```
// 3x4 の行列を読み込み転置行列を出力
#include <iostream>
#include <vector>
int main()
{
   using std::cout, std::vector;
   const int n{3}, m{4};
   auto data { vector(n, vector<int>(m)) };

   // n x m の 2 次元配列に入力
   for (int i = 0; i < n; i++) // 行方向
      for (int j = 0; j < m; j++) // 列方向
         std::cin >> data[i][j];
   cout <<"\n";

   // 入力した行列をそのまま出力
   for (int i = 0; i < n; i++) { // 行方向
      for (int j = 0; j < m; j++) // 列方向
         cout << data[i][j] <<" ";
      cout <<"\n";
   }
   cout <<"\n";

   // m x n の転置行列を出力
   for (int j = 0; j < m; j++) { // 転置行列の行方向
      for (int i = 0; i < n; i++) // 転置行列の列方向
         cout << data[i][j] <<" ";
      cout <<"\n";
   }
   cout <<"\n" ;
   return 0;
}
```

実行例は次のようになります。

```
1 2 3 4                 <<--- 1 行目の入力 i=0, j=0,1,2,3
5 6 7 8                 <<--- 2 行目の入力 i=1, j=0,1,2,3
9 10 11 12              <<--- 3 行目の入力 i=2, j=0,1,2,3

1 2 3 4
5 6 7 8
9 10 11 12

1 5 9
2 6 10
3 7 11
4 8 12
```

逆行列

2×2 の行列の逆行列を求めて、正しい逆行列が求まったかどうかを確認しているプログラムを以下に示します。次の逆行列を求める式をそのままプログラムで表しています。

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad A^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

行列式 det(determinant of matrix) の値が 0 かどうかを確認するのに、<cmath>ヘッダファイルをインクルードして利用可能な abs() 関数を用いています。

ソースコード 59: 逆行列の処理

```
// 2x2 の行列の逆行列を求める
#include <iostream>
#include <vector>
#include <cmath> // 実数用のabs()のために必要
int main()
{
   using std::cout, std::vector;
   vector<vector<double>> a {{1.0, 3.0}, {5.0, 7.0}};
   // 行列a を出力
   cout << a[0][0] <<" "<< a[0][1] <<"\n"
        << a[1][0] <<" "<< a[1][1] <<"\n\n";

   // a の逆行列が存在するかを判定
   const double epsilon {1.0e-8};
   const double det {a[0][0]*a[1][1] - a[0][1]*a[1][0]};
   if (abs(det) < epsilon) {
      cout <<"no inverse\n";
      return 1;
   }

   // a の逆行列を計算
   vector<vector<double>> ia(2);
   ia[0] = { a[1][1]/det, -a[0][1]/det };
   ia[1] = {-a[1][0]/det, a[0][0]/det };

   // a の逆行列を出力
   cout << ia[0][0] <<" "<< ia[0][1] <<"\n"
        << ia[1][0] <<" "<< ia[1][1] <<"\n\n";

   // 確認，単位行列が出力されればOK
   cout << a[0][0]*ia[0][0] + a[0][1]*ia[1][0] <<" "
        << a[0][0]*ia[0][1] + a[0][1]*ia[1][1] <<"\n"
        << a[1][0]*ia[0][0] + a[1][1]*ia[1][0] <<" "
        << a[1][0]*ia[0][1] + a[1][1]*ia[1][1] <<"\n";
   return 0;
}
```

逆行列のための変数 ia(inverse of a) の計算にも注意しましょう。ここでは変数 a と同じ方法ですべての要素の初期値を指定しても良いのですが、vector<T>要素の代入の特徴を示すために代入文としました。変数 ia は宣言によって vector<double>型の 2 要素の配列です。ia[0] と ia[1] は、初期化と同じく、中括弧を使った double 値のリストを代入に指定できます。これは push_back() の場合も同様です。さらには以下のような代入もできます。

```
ia = {{0.0, 1.1}, {2.1, 3.2}, {4.3, 5.4}};
```

この代入の場合には、ia の要素は完全に入れ替わるので要素数は指定した値の数によって任意に決められます。これらの代入の機能は C++11 で導入されたものです。

このプログラム例では行列の出力が 3 回ほど登場しました。すべて cout を main 関数の中に書いてあります。2×2 の行列なのでこれでも問題ありませんが、一般の $n \times m$ の行列をループで書く場合には関数を使うことが望まれます。次節ではその方法を見ていきましょう。

6.6　2 次元配列の引数

vector は何次元の配列であろうとも、int 型などの組み込み型と同様に関数の引数として指定できます。ただし、関数のところで説明したように、引数に値渡しを選択すると、vector のような複数の値を持つデータ型は要素のコピー起こり、実行を遅くする要因となります。そのため、const std::vecrtor<T>& または std::vector<T>&のどちらかの指定が望まれます。

ソースコード 60: 2 次元配列の引数

```
1  // ２次元配列を引数として渡す例
2  #include <iostream>
3  #include <vector>
4  using std::cout, std::vector;
5  // 配列の各要素を+1する
6  void incr(vector<vector<int>>& x)
7  {
8     for (auto& a : x)
9        for (auto& b : a)
10          ++ b;
11 }
12
13 // 配列のすべての要素を出力する
14 void print(const vector<vector<int>>& x)
15 {
16    for (const auto& a : x) {
17       for (auto b : a)
18          cout << b <<" ";
19       cout <<"\n";
20    }
21 }
22
23 int main()
24 {
25    vector<vector<int>> a {{1,2,3,4,5}, {2,3,4,5,6}, {3,4,5,6,7}};
26    print(a);
27    cout <<"\n";
28    incr(a);
29    print(a);
30    return 0;
31 }
```

このプログラムでは、引数の値渡しと同様の理由で、範囲 for 文による vector 要素のコピーを避けるために、const auto&と auto&を使っています。ところが、16 行目の指定は auto&でも問題ありません。これは、この関数の仮引数 x に const 指定が付いているために、auto&の指定でも自動的に const となるためです。複雑になるので const auto&を指定しておくのが良いです。

6.7　2 次元配列の一部を実引数とする例

2 次元配列は配列の配列なので、vector 配列の 2 次元のうち 1 次元分のみを関数呼び出しの実引数に指定できます。次のプログラムでは 1 次元配列を仮引数とする関数に対して、2 次元配列の一部を実引数として指定しています。

ソースコード 61: 2 次元配列の一部を引数とする例

```
// ２次元配列の一部を引数として指定する
#include <iostream>
#include <vector>
using std::cout, std::vector, std::string;

// int の 1 次元配列を出力する
void print_int_array(const vector<int>& x)
{
   for (size_t i = 0; i < x.size(); i++) {
      if (i != 0) cout << " ";
      cout << x[i];
   }
   cout <<"\n";
}

// 文字列を 1文字ずつ間隔をあけて出力する
void print_capital(string s)
{
   for (size_t i = 0; i < s.size(); i++) {
      if (i != 0) cout << " ";
      cout << s[i];
   }
   cout <<"\n";
}

int main()
{
   vector<vector<int>> a {{1,2,3,4,5}, {2,3,4,5,6}, {3,4,5,6,7}};
   print_int_array( a[1] );

   vector<string> b {"a1cd", "e2g", "hi"};
   print_capital( b[1] );

   return 0;
}
```

main 関数で vector の 2 次元配列 a を実引数として a[1] と指定すると、この配列の{2,3,4,5,6}の部分の 1 次元配列が引数として渡されます。関数 print_int_array() の仮引数 x は 1 次元配列扱うのみでそれが 2 次元配列の一部であるかどうかは問いません。

変数 b は string を要素とする vector 配列です。これは 2 次元配列ではありませんが、string は char 型の配列として扱えるので、print_capical() は print_int_array() と同じ構造です。

6.8 配列を返す関数

vector 配列を引数ではなく戻り値として扱うこともできます。次のプログラムは vector の 1 次元配列に入力を行う関数を使って、2 次元配列の入力を行っています。

ソースコード 62: 1 次元配列を返す関数の利用

```
1  // 表の各行の和を出力する
2  // 1次元配列の入力関数の利用
3  #include <iostream>
4  #include <vector>
5  using std::vector;
6
7  vector<int> input(int n)
8  {
9     vector<int> v(n);
10    for (auto& e: v)
11       std::cin >> e;
12    return v;
13 }
14
15 int main()
16 {
17    int row{}, col{};
18    std::cin >> row >> col;
19    vector<vector<int>> t(row);
20
21    // 2次元配列の入力
22    for (auto& e: t)
23       e = input(col);
24
25    // 2次元配列と各行の合計の出力
26    for (const auto& v: t) {
27       int s {0};
28       for (auto e: v) {
29          s += e;
30          std::cout << e <<" ";
31       }
32       std::cout << s <<"\n";
33    }
34 }
```

input 関数は引数で指定された個数のデータを局所変数である v に入力し、その vector 変数を返します。main 関数では、2 次元データの列 (col) と行 (row) の数を入力した後に、col×row 個の 2 次元データを入力します。そのため、input の引数には列の個数に相当する col を実引数として指定し、それを行の数分だけ繰り返します。入力の後は、範囲 for 文の二重ループにより、各行の出力と合計を行数分だけ繰り返しています。

input 関数は配列を戻り値として指定していますが、vector 変数の値渡しの引数とは違って、局所変数の vector を返す場合には要素ごとの複製が起こりません。関数呼び出しが終わればその中の局所変数の役割も終わりなので、vector 配列の各要素は複製されることなく再利用されるためです。

実行例を見てみましょう。

```
$ ./a.out
4 3
11 12 13
21 22 23
31 32 33
41 42 43        <<---- ここまでを入力
11 12 13 36
21 22 23 66
31 32 33 96
41 42 43 126
```

1 行目の入力で 4 行 3 列の 2 次元データであることを示し、次の行から 12 個のデータを入力しています。各行の入力および出力をプログラムの場所の対応を付けてください。さて、12 個程度のデータならばキーボードから入力してもそれほど無理はありませんが、それ以上となるとファイルを使用した方が良いでしょう。次の章ではファイルを使った入力と出力の方法について見ていきます。

7　ファイル操作とファイルストリーム

この章ではファイルからデータ入力をしたり、出力結果をファイルに残したりする処理について説明します。

7.1　標準入出力とファイル

まず、C++のプログラムと OS とのインタフェースについて説明します。C++側で用意されている標準入出力ストリームの機能と OS の機能とを組み合わせると、OS によって管理されているファイル・画面・キーボードを、C++のプログラム側から統一したプログラミングスタイルで扱うことができます。C++のほとんどのプログラムは、データを入力し、そして出力します。入力や出力を指定する方法はいくつかあるのですが、基本的な方法は、標準入出力ストリーム cin・cout・cerr の使用です。cin や cout は <iostream>ヘッダファイルを インクルードすることで使えるようになるオブジェクトです。正確には、

- std::cin は std::istream 型の変数（クラスのオブジェクト）
- std::cout は std::ostream 型の変数 (クラスのオブジェクト)
- std::cerr は std::ostream 型の変数 (クラスのオブジェクト)

として宣言されています。

プログラムを実行する際に特別な指定をしなければ、cin はキーボードからの入力を扱い、cout と cerr は画面への出力を扱います。これらの関係はプログラムを実行する際のユーザの指定によって変えられます。そのため cout にユーザへのプロンプトメッセージを表示し、cin でキーボードから入力を受け取るように、と思ってプログラムを書いたとしても、そのプログラムを実行する際に、cin や cout をファイルや他のプログラムの入出力に関連づけることができます。cerr が無駄に思えるかもしれませんが、cout をファイルに関連付けた際に、エラーなどを表示するための画面出力に cerr が使えます。出力や入力の対象を変える方法には、リダイレクションとパイプの二種類があるので、それらを見ることにしましょう。

リダイレクション

リダイレクションは Unix OS が提供する機能で、プログラムの標準入出力を指定したファイルにつなげるものです。Windows で MobaXterm や Cygwin を使っている場合にも同じことができます。この機能は、通常、ユーザがプログラムを実行する際に入力するコマンドといっしょに指定することで利用できます。これによって cin・cout・cerr がキーボードや画面以外を扱うように変更できます。以下のプログラムでは、ユーザに入力を促すプロンプトを cerr で表示して、cin から整数値を読み込み、改行をつけて cout に出力します。

ソースコード 63: ユーザからの入力とその出力

```
1 // 入力ストリームから整数値を読み込み、改行をつけて出力する
2 #include <iostream>
3 int main()
4 {
5    for (int x{}; std::cerr<<"input number --> " && std::cin >> x; )
6       std::cout << x << std::endl;
7    return 0;
8 }
```

std::cin >> x の処理では、ファイルの終り (EOF) を検出するか、x の型から不正と判断されるデータを読み込むと、この式の結果が false になります。そして for 文が終了します。for (int x{}; ...) にも着目してください。変数 x は入力ループだけで使う一時的な変数です。for 文で宣言することで変

数のスコープをループ内に制限しています。さらに x はすぐに入力に使うので初期値に意味がありません。そのため、他の for 文とは異なり int x=0; とはしていません。このようにちょっとした書き方でプログラムの書く側がどのような注意を払っているかを示すことも、良いプログラムを書くための大事な点です。

コンパイルと実行の例は以下のようになります。

```
$ g++ -std=c++17 file01.cpp -o f1
$ ./f1
input number --> 1
1
input number --> 2
2
input number --> 3
3
input number --> $                <<- Ctrl-D を入力
```

リダイレクションを使って、cout の出力先をファイルにすると以下のようになります。

```
$ g++ -std=c++17 file01.cpp -o f1
$ ./f1 > result
input number --> 1
input number --> 2
input number --> 3
input number --> $                 <<- Ctrl-D と enter を入力
$ cat result
1
2
3
```

実行時の > result の指定により、cerr の出力が画面に現れて、cout の結果だけがファイル result に入ります。本来 cerr の出力は、何らかのエラーを画面に出力することを目的としていますが、プログラムが作り出す数値などの出力とユーザへのメッセージを分けるという考えに立てば、このような使い方も有効です。

リダイレクションは、出力だけでなく入力にも使えます。次の実行例を見てみましょう。これは前の例の結果をプログラムの入力として使っている例です。cout と cerr の出力が混ざって画面に現れています。

```
$ ./f1 < result
input number --> 1
input number --> 2
input number --> 3
input number --> $
```

まとめると、リダイレクションはコマンドを実行する際のプログラムの標準入力と標準出力を変更する機能で、> **ファイル名**によって標準出力が指定したファイルとを結びつかせ、< **ファイル名**によって標準入力が指定ファイルと結びつかせます。双方を同時に指定することもできます。C++の<<と>>とは向きが逆に思えますが、どちらもデータの向かう対象を指していると思えば一貫しています。

パイプ

パイプも Unix OS が提供する機能で、あるプログラムの標準出力を他のプログラムの標準入力につなげるものです。この機能も Windows 上の MobaXterm や Cygwin で使えます。あるプログラムの出力を他のプログラムの入力につなげることで、複数のプログラムによる協調作業が可能となります。Unix

という OS のコマンド群は、単純で着実なプログラムをそれぞれつなぐことで、高機能を実現するという設計思想で作られました。

実行例には先程と同じ、整数値を読み込み、改行をつけて出力するプログラムを使います。また、入出力をつなげる他のプログラムとして、Unix OS で良く使われる wc コマンド (word count の略) を使ってみましょう。このコマンドは、指定されたファイル中の文字の行数・語数・文字数 (バイト数) を出力するコマンドです。ファイルを指定しない場合には標準入力から入力を読み込みます。

```
$ ./f1 | wc
input number --> 3
input number --> 4
input number --> 5
input number --> 6
input number --> 19
input number -->        5       5      11            <<- Ctrl-D を入力
$
```

出力結果の最終行の 5　5　11 が wc コマンドの出力です。これは左から行数・語数・バイト数を意味します。f1 のプログラムの中で cout に出力した文字のみが wc の入力となっています。11 バイトとなるのは改行文字を 1 文字と数えているためです。

演習

1. ./f1 < result | wc | ./f1 を試してみましょう。

7.2　指定ファイルからのデータ入力

C++のファイル操作はデータの流れを扱うストリームという概念に基づいています。ファイルストリームはこれまで扱ってきた cin や cout のような入出力ストリームと同じ方法でファイルへの入出力を操作するものです。つまり、cin・cout・cerr の他に特定のファイルに関連したストリームを操作するためのオブジェクトをプログラム中で作れます。それには <fstream>ヘッダファイルをインクルードして以下の型の変数を使います。

- 入力ファイル用に std::ifstream 型の変数
- 出力ファイル用に std::ofstream 型の変数

型名を注意深く見ると数ページ前に示した cin や cout の型とはわずかに違います。しかし、これらの型の変数は、cin・cout と使い方が同じで併用もできます。次に ifstream を使ったプログラムの例を示します。

ソースコード 64: ファイル入力の基本

```
1  // 指定したファイルから整数データを読み込み、1行ごとに出力する
2  #include <iostream>
3  #include <fstream> // ifstream を使う場合に必要
4  int main()
5  {
6     // 入力用ファイルオブジェクトの準備
7     std::ifstream fin {"file02.dat"}; // 入力用ファイルオブジェクトの宣言
8     if (!fin) { // ファイル入力の準備ができたか?
9        std::cerr << "cannot open\n";
10       return 1;
11    }
12
13    // ファイルから整数を入力し1行ごとに出力
14    for (int x{}; fin >> x; ) // cin と同じように使える
```

```
15      std::cout << x <<" ";
16    std::cout <<"\n";
17    return 0;
18 }
```

7 行目が fin オブジェクト（変数）の宣言です。このオブジェクトを通してファイルからの入力を扱います。文字列リテラル"file02.dat" はオブジェクトの初期化に使われます。初期化の処理では、この文字列を名前とするファイルと fin オブジェクトとが関連付けられて、入力のための準備が行われます。

次の if 文は、宣言時の準備処理が正しく行われたかをチェックしています。指定したファイルがなかったり、読み込みの許可が与えられていない場合には、準備処理が失敗します。ifstream と ofstream クラスのオブジェクトは、'!' という演算子によって準備が整っているかどうかを調べることができます。!fin が true の場合には、準備失敗なので、return 1; とすることで、main 関数を終わり、同時にプログラム実行も終了します。準備が正しく行われていることが確認できたならば、あとは、cin と同じように入力ストリームとして使います。実行例を見てみます。

```
$ ./f2
cannot open
$ echo 1 2 3 4 5 6 7 8 > file02.dat
$ cat file02.dat
1 2 3 4 5 6 7 8
$ ./f2
1 2 3 4 5 6 7 8
```

1 回目の実行では、file02.dat というファイルが存在しなかったためにプログラムがすぐに終了しています。次に、 echo コマンドの結果をリダイレクションによってファイルに入れて、データファイルを作成します。2 回目の実行では、作成したファイルからデータを読み込んで表示することに成功します。

7.3 指定ファイルへのデータ出力

次のプログラムは 1 から 10 の整数とその二乗の値を、1 行ずつ output.dat というファイルに出力します。ファイル入力とほぼ同じように、7 行目の指定によって fout という出力用ファイルオブジェクトの宣言を行い、if (!fout) でエラーチェックをしています。出力ファイルのエラーチェックは奇異に感じるかも知れませんが、コンピュータの設定でファイルを作成できない場合やすでに書き込み不可のファイルが存在する場合はエラーになるので、必要な指定です。そして、出力を行うところでは cout と同じように fout を使って指定できます。

ソースコード 65: 指定ファイルへの出力

```
1  // 指定したファイルに整数データを出力する(成功時には画面に何も表示しない)
2  #include <iostream>
3  #include <fstream> // ofstream を使う場合に必要
4  int main()
5  {
6     // 出力用ファイルオブジェクトの準備
7     std::ofstream fout {"output.dat"}; // 出力用ファイルオブジェクトの宣言
8     if (!fout) { // ファイル出力の準備ができたか?
9        std::cerr <<"cannot open\n";
10       return 1;
11    }
12
13    // 1から 10までの整数とその 2 乗の値をファイルに出力
14    for (int i = 1; i <= 10; i++)
15       fout << i <<" "<< i*i <<"\n"; // cout と同じように使える
16    return 0;
17 }
```

7.4 ファイル入力と標準入出力

次のプログラムは入力データをファイルから読み込み、キーボードから検索したいデータを入力するプログラムです。つまりファイル入力と標準入力の 2 種類の入力を操作するプログラムです。ちょっと長い例ですが、これまでのプログラムの組み合わせでできています。ファイルからの入力を fin、標準入力からの入力を cin から読み込んでいますが、プログラムの書き方はどちらも同じです。

ソースコード 66: 標準入力とファイル入力

```
// ファイルからデータを読み込んだ後に、
// 標準入力ストリームから検索したいデータを入力する
#include <iostream>
#include <fstream>
#include <vector>
int main()
{
   // 入力用ファイルオブジェクトの準備
   std::ifstream fin {"file04.dat"}; // 入力用ファイルオブジェクトの宣言
   if (!fin) { // ファイル入力の準備ができたか?
      std::cerr << "cannot open\n";
      return 1;
   }

   // 変数宣言とファイルからのデータを入力
   std::vector<int> data(100);
   int num {0};
   while (num < data.size() && fin >> data[num])
      ++ num;

   // 標準入力ストリームを使った検索
   for (int x{}; std::cout <<"search for(EOF to quit) " && std::cin >> x; ) {
      // 逐次探索
      int i;
      for (i = 0; i < num; i++)
         if (data[i] == x)
            break;

      // 結果の出力
      std::cout << x <<" is "<< ((i < num)?"":"not ") <<"found\n";
   }
   return 0;
}
```

このプログラムは入力データを 100 個に制限しています。data の宣言時に配列のサイズを 100 個としているので、push_back() で増やさない限り data[100] は使用できません。一つ目の while 文の条件に注目してください。

```
num < size && fin >> data[num]
```

まず num の大きさを確認し、短絡評価を使って、入力をしています。これにより配列のサイズより多いデータを読み込まないようにしています。配列のサイズを限定した場合の典型的な書き方なので覚えましょう。また、出力の際に 3 項演算子を使って"not "の文字列の出力を制御しているので確認しておいてください。実行例を示します。

```
$ cat file04.dat
1 3 4 6 8 10 12
$ ./a.out
search for(EOF to quit) 3
3 is found
search for(EOF to quit) 5
5 is not found
search for(EOF to quit) 2
2 is not found
search for(EOF to quit) 4
4 is found
search for(EOF to quit)
```

データ数に特に制限を持たせなくても良い場合には、vector のサイズを増やす機能を使うことができます。その場合の例も見ておきましょう。

```
// 変数宣言とファイルからのデータを入力
std::vector<int> data;
for (int x{}; fin >> x; )
  data.push_back(x);
```

data.push_back(x); によって x に入力した値を次々と data に入れることができます。

7.5 入力用ファイル名の読み込み

次のプログラムは、入力用ファイルの名前をプログラム中に書かないものです。プログラムの実行を開始した後に、ファイル名を cin から読み込み、前の例のように fin を宣言するとともにファイル操作の準備をします。C++の変数は使いたいところで宣言できるという特徴をうまく利用しています。

ソースコード 67: 入力したファイル名の利用

```
1  // 入力データのファイル名を標準入力ストリームから読み込む
2  #include <iostream>
3  #include <fstream>
4  int main()
5  {
6     // ファイル名の入力
7     std::string nm;
8     std::cout << "input file name ---> ";
9     std::cin >> nm;
10
11    // 入力用ファイルオブジェクトの準備
12    std::ifstream fin {nm}; // 入力用ファイルオブジェクトの宣言
13    if (!fin) { // ファイル入力の準備ができたか?
14       std::cerr <<"cannot open "<< nm <<"\n";
15       return 1;
16    }
17
18    // データの読み込みと改行をつけた出力
19    for (int data{}; fin >> data; )
20       std::cout << data <<"\n";
21    return 0;
22 }
```

7.6 複数ファイルからのデータ入力

次は複数の入力ファイルからデータを読み込んで、全体の合計を計算します。ファイル名は vector による文字列の配列です。範囲 for 文によってこの配列から一つずつ文字列を file 変数に取り出し、ループの中で処理しています。ループ内の変数 in の宣言に注目してください。ファイルを扱うこの in 変数のスコープはループの終わりまでです。つまり、繰り返しごとに in 変数は作成されて、ループ終わりで後処理が行われます。これによって in という一つの変数名で複数のファイルを処理できます。

ソースコード 68: 複数ファイルの処理

```
1  // 複数ファイルの読み込み
2  #include <iostream>
3  #include <fstream>
4  #include <vector>
5  int main()
6  {
7     std::vector<std::string> file_list
8           {"data1.txt", "data2.txt", "data3.txt", "data4.txt"};
9     int total {0};
10    for (auto file : file_list) {
11        // 入力用ファイルオブジェクトの準備
12        std::ifstream in {file}; // 入力用ファイルオブジェクトの宣言
13        if (!in) { // ファイル入力の準備ができたか?
14            std::cerr <<"cannot open "<< file <<"\n";
15            continue; // 次のファイルへ
16        }
17
18        // ファイルごとのデータ読み込みと積算
19        for (int data{}; in >> data; )
20            total += data;
21    }
22
23    std::cout <<"total = "<< total <<"\n";
24    return 0;
25 }
```

実行例は次のようになります。data1.txt から data4.txt のファイルを用意しておくと、それぞれのファイルを処理して合計を出力するプログラムとなりました。

```
$ cat data1.txt
1 2 3 4 5
$ cat data2.txt
6 7 8 9 10
$ cat data3.txt
11 12 13 14 15
$ cat data4.txt
16 17 18 19 20
$ ./a.out
total = 210
```

この例ではプログラム中にファイル名が書き込まれているので、他の名前のデータファイルに対応できません。ファイル名は cin から入力するか、コマンドの引数として指定する方が良いです。次のプログラムは、以前にみた main() 関数の引数を操作する方法を応用して、コマンドの引数からファイル名を取り込むものです。argv+1, argv+argc の部分は半開区間です。以前に見た例は argv, argv+argc でした。argv の情報の先頭には実行するプログラムファイルの名前が保存されていて、コマンド引数の文字列は 2 番目以降にあります。そのため開始位置を argv+1 としています。

ソースコード 69: コマンド引数で複数ファイルを指定

```
// 複数ファイルの読み込み(コマンド引数版)
#include <iostream>
#include <fstream>
#include <vector>
int main(int argc, char* argv[])
{
   std::vector<std::string> file_list(argv+1, argv+argc);
   int total {0};
   for (auto file : file_list) {
       // 入力用ファイルオブジェクトの準備
       std::ifstream in {file}; // 入力用ファイルオブジェクトの宣言
       if (!in) { // ファイル入力の準備ができたか?
           std::cerr <<"cannot open "<< file <<"\n";
           continue; // 次のファイルへ
       }

       // ファイルごとのデータ読み込みと積算
       for (int data{}; in >> data; )
           total += data;
   }

   std::cout <<"total = "<< total <<"\n";
   return 0;
}
```

実行例を以下に示します。最初の実行ではデータとなるファイル名を実行ファイルのコマンド引数として並べています。これらのデータファイルは数字の部分が異なるだけです。Unix のシェルでは、コマンド引数の中で ? という文字を指定すると、その部分の 1 文字だけが異なるファイルを探し、コマンド引数として展開します。cat と./a.out の実行では、これ利用して実行時のコマンド入力文字数を減らしています。

```
$ ./a.out data1.txt data2.txt data3.txt data4.txt
total = 210
$ cat data?.txt
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
$ ./a.out data?.txt
total = 210
```

7.7 ファイルオブジェクトの引数

ifstream や ofstream クラスのオブジェクトを関数の引数にする場合には、リファレンス引数とします。このようにするのは、ファイルオブジェクトの複製が禁止されているためです。つまり、このオブジェクトは代入や関数の値渡しの引数指定ができません。以下のプログラムは複数ファイルへの出力を関数を使って行っています。

ソースコード 70: 複数ファイルへの出力

```
// 複数ファイルへの書き込み
#include <iostream>
#include <fstream>
#include <vector>

void print(std::ofstream &o, int i) // o はリファレンス引数
{
   for (int j = 0; j < 20; j++)
       o << (i+1)*j << "\n";
}

int main()
{
   std::vector<std::string> file_list {"out1.txt", "out2.txt", "out3.txt"};
   for (size_t i = 0; i < file_list.size(); i++) {
      std::ofstream out {file_list[i]}; // それぞれのファイルに対して
      if (!out) {
         std::cerr << "cannot open "<< file_list[i] <<std::endl;
         continue; // 次のファイルへ
      }
      std::cout << file_list[i] <<"\n";
      print(out, i); // ファイル別の出力
   }
   return 0;
}
```

ファイルストリームはこれまで扱ってきた cin や cout のような入出力ストリームと同じ方法でファイルへの入出力を操作するものであると、説明してきました。実際に cin のデータ型である istream および cout のデータ型である ostream は、ファイルストリームのデータ型である ifstream および ofstream と密接な関係にあります。そのため、この例のプログラムの print() 関数の仮引数を以下のように指定すると、実引数にファイルストリームのオブジェクトと cout の双方が指定できるようになります。

```
void print(std::ostream &o, int i) // 型名がわずかに異なる
{
  // 上記と同じ
}

print(std::cout, i); // coutはostream型
print(out, i);       // outはofstream型であるがostream型に指定できる
```

7.8 読み込み位置の変更

cin による処理とファイルを直接扱う ifstream の入力処理は似ている部分が多いのですが、既に存在しているファイルにはデータがすべて揃っている点や入力を終えたデータも残っている点が大きく異なります。これまで見てきた例では、入力がファイルの先頭データからであったため、cin と ifstream オブジェクトの処理はほとんど同じでした。しかし、isftream オブジェクトを操作するとファイルの途中にあるデータを直接読み込んだり、以前に読み出したデータを再度読み込むことができます。それには、seekg() と tellg() というメンバ関数を使います（それぞれの関数の g は get を表すようです）。

- seekg(オフセット, 起点) : 次の入力を起点からバイト単位のオフセット分だけずれた場所から読み込むように設定する。起点は以下のどれかを指定する

  - beg: ストリームの先頭
  - cur: 現在位置
  - end: ストリームの末尾
- seekg(先頭からの位置)：tellg() の結果を使い読み込む場所を直接指定
- tellg()：現在の読み込み位置の取得（整数型）

seekg() 関数には 2 種類の指定方法がありますが、単独で使う場合と tellg() と対で使う場合とで使い分けます。単一引数の seekg() と tellg() の戻り値は std::treampos 型です。これは整数系の型なので、int 型の変数にも保存できます。しかし、厳密には std::treampos と int は型が異なり、ファイルサイズが大きい場合には問題となります。その対処も含めて以下に使い方の例を示します。

ソースコード 71: seekg と tellg の例

```
1  // seekg and tellg (get position)
2  // test.txt: 1000 2000 3000 4000 5000 6000 7000
3  #include <iostream>
4  #include <fstream>
5  int main()
6  {
7     std::ifstream in {"test.txt"};
8     if (!in) {
9        std::cout << "cannot open\n";
10       return 1;
11    }
12
13    int x, y;
14    in.seekg(10, in.beg); // 先頭から 10バイト
15    in >> x >> y;
16    std::cout << x <<" "<< y <<"\n";
17
18    auto pos { in.tellg() }; // 読み込み位置を保存
19    in.seekg(0, in.beg) >> x >> y; // 先頭
20    std::cout << x <<" "<< y <<"\n";
21
22    in.seekg(-5, in.cur) >> x >> y; // 5バイト戻る
23    std::cout << x <<" "<< y <<"\n";
24
25    in.seekg(pos) >> x >> y; // pos の位置
26    std::cout << x <<" "<< y <<"\n";
27    return 0;
28 }
```

実行例とともにプログラムの動作を説明します。このプログラムは、in 変数を宣言してファイル処理の準備をした後に、14 行目でファイルの先頭から 10 バイト目を次に読み込むように指定しています。入力ファイルの text.txt がコメントにあるようなデータの場合、数字はスペース文字を含めて 5 バイトごとになっています。そのため、先頭から 10 バイト目は 3000 の 3 の部分です。18 行目で現在の読み込み位置を pos 変数に保存しています。変数の型は整数系ですが、厳密には int ではないので auto を指定することでコンパイラに任せます。その後、読み込み位置を先頭に変更してから二つの整数を読み込みます。in.seekg() 関数の呼び出しの戻り値は設定変更した in なので、この例のように関数の戻り値を>>演算子に使用できます。そして、その時点での読み込み位置から 5 バイト戻って再度二つの整数を読み込みます。最後に、pos を使って以前に保存しておいた読み込み位置に戻ります。

```
$ cat test.txt
1000 2000 3000 4000 5000 6000 7000
$ ./a.out
3000 4000
1000 2000
2000 3000
5000 6000
```

小さいなサイズのファイル読み込みでは、ファイル全体を読み込んでしまった方が簡単かもしれませんが、ギガバイトのサイズの場合にはこの方法が有効になってきます。

異なる目的の使い方ですが、読み込み位置をストリームの末尾に移動して tellg() を行うことで、操作しているファイルのサイズを知ることができます。知っておくと便利です。

```
in.seekg(0, in.end);       // ファイルの末尾へ移動
auto size { in.tellg() }; // 読み込み位置はファイルサイズと同じ
std::cout << file <<" "<< size <<"\n";
```

7.9 書き込み位置の変更

読み込み位置の変更と同じく、ofstream の書き込み位置も変更できます。使用する関数は seekp() と tellp()（p は put を表すようです）の二つで、使い方は読み込み位置の変更と同じです。使用する際の注意点は、書き込み位置を戻して書き込むと以前に書き込んだデータを上書きする点です。意味のある書き込みを残すには、上書きする分のバイト数を確認しながら行う必要があります。次の例でそれを見ましょう。このプログラムは、標準入力から得た整数をファイルに書き込んでいきますが、そのデータ数を出力するデータの前側に保存します。

ソースコード 72: seekp と tellp の例

```
1  // seekp and tellp (put position)
2  #include <iostream>
3  #include <iomanip>
4  #include <fstream>
5  int main()
6  {
7     std::ofstream out {"out.txt"};
8     if (!out) {
9        std::cout << "cannot open\n";
10       return 1;
11    }
12
13    out << "data file\n";
14    auto pos { out.tellp() };
15    for (int count{1}, x; std::cin >> x; count++) {
16       out.seekp(pos) << std::setw(5) << count <<"\n";
17       out.seekp(0, out.end) << x <<" ";
18    }
19    out <<"\n";
20    return 0;
21 }
```

14 行目で読み込んだデータ数を保存する場所を変数 pos に記録します。for 文の変数 count が読み込んだデータ数で、これを pos の場所に書き込み、その後、書き込み位置をストリームの末尾に移動して読み込んだデータを出力します。データ数の書き込みを毎回同じバイト数となるようにするために、count の値を書き込む際に setw() を使って書き込み幅を 5 桁に設定しています。この関数は cout にも利用できるもので、<iomanip>ヘッダファイルをインクルードして使用します。count の書き込みは右寄せで 5 桁分の場所を使って行われます。数値が 5 桁より少ない場合には余った場所の左側にスペースが出力されます。実行例を見てみましょう。

```
$ ./a.out
1 3 5 7 9 11 13 15 17 19 21 <<---- これを入力して最後に Ctrl-D を入力
$ cat out.txt
data file
   11
1 3 5 7 9 11 13 15 17 19 21
```

out.txt の 2 行目の値が読み込んだデータ数です。このプログラムではデータ数の書き込みに 5 桁幅の数字と改行文字を指定しているので、データ数が 6 桁になると最初のデータが上書きされてしまうことに注意しましょう。

7.10 追加書き込み

ofstream によるデフォルトの書き込み動作では、指定ファイルが存在しなければ新規に作成し、存在する場合には内容を切りつめて（消去して）、新規に作成したのと同じ状態にします。サーバプログラムがログファイルを作成するような場合には、既に存在するファイルの末尾に追加で書き込む必要があります。これを行うには、ファイルストリームオブジェクトの宣言時に二つ目の引数を指定します。

ソースコード 73: 追加書き込みの例

```
1  // openmode
2  #include <iostream>
3  #include <fstream>
4  int main()
5  {
6     std::ofstream out {"outlog.txt", std::ios::app};
7     if (!out) {
8        std::cout << "cannot open\n";
9        return 1;
10    }
11    out << "append data\n";
12    return 0;
13 }
```

std::ios::app を指定してオブジェクトを宣言すると、指定ファイルが存在しなければ新規に作成する点は変わりありませんが、すでにファイルが存在するとファイルの末尾に書き込むようになります。なお、追加書き込みでは seekg() を使っても既存ファイル部分には移動できません。

型構成の関係から app の指定は、次のどれでも同じ意味となります。

- std::ios_base::app
- std::ios::app
- std::ostream::app
- std::ofstream::app

書籍やインターネット上のサンプルをみると統一した使われ方はないようなので、このテキストでは一番短い文字列を採用しました。

同じような追加の機能は最初の節でみたリダイレクションにもあります。プログラムを実行する際に、>> result とすると result というファイルに出力内容を追加することになります。

```
$ rm result
$ echo 1 2 3 >> result
$ echo 4 5 6 >> result
$ echo 7 8 9 >> result
$ cat result
1 2 3
4 5 6
7 8 9
```

8　テキスト入力処理

コンピュータは数値を素早く計算する目的で作られましたが、現代では電子メールをはじめとしてテキストを処理することがとても重要になっています。ここでは、テキストファイルを構成する文字の処理に焦点を絞って見ていきます。

8.1　１文字ずつの処理

std::cin >> x やファイル操作の例の fin >> x という入力方法では、スペース・タブ・改行が読み飛ばされます。これらの文字はまとめてホワイトスペースまたは空白と呼ばれます。x が数値のためのデータ型ならばホワイトスペースの読み飛ばしは何の問題もありませんが、文字を処理する場合にはホワイトスペースを処理対象に含めなければならないこともあります。まずは読み飛ばしの動作を確認してみましょう。次のプログラムは char 型の変数に１文字ずつ入力を読み込み、そのまま出力するものです。

ソースコード 74: 文字の入出力

```
1 // 空白を除いて入力ストリームから１文字ずつ入力し、１文字ずつ出力する
2 #include <iostream>
3 int main()
4 {
5    char ch {};
6    while (std::cin >> ch)
7       std::cout << ch;
8    return 0;
9 }
```

このプログラムを実行してみると、以下のようになり、ホワイトスペースが取り去られていることが分かります。

```
This is a test.     <<--- 入力
Thisisatest.        <<--- プログラムの出力
```

ホワイトスペースを含めて処理を行うには、cin に対してホワイトスペースを取り去らないように指示する必要があります。以下のプログラムは１文字ずつ入力を行いホワイトスペースを含めて１文字ずつ出力します。

ソースコード 75: 文字の入出力によるコピー

```
1 // 空白を含めて入力ストリームから１文字ずつ入力し、１文字ずつ出力する
2 #include <iostream>
3 int main()
4 {
5    char ch {};
6    while (std::cin >> std::noskipws >> ch)
7       std::cout << ch;
8    return 0;
9 }
```

cin に対して、noskipws マニピュレータを指定しています。この名前は”no skip white space”の省略形、つまり ”ホワイトスペースのスキップはなし”だと考えれば良いでしょう。これによって、ホワイトスペースも含めた文字が入力の繰り返しごとに変数 ch に入ります。これはあるファイルを別のファイルに正確にコピーするプログラムとなっていることに注意してください。実行例は以下のようになります。

```
This is a test.     <<--- 入力
This is a test.     <<--- プログラムの出力
```

8.2 カウント

テキスト処理でよく行われるのは文字数や行数を数えることです。ここではその基本について見てきます。

入力文字のカウント

まずは入力文字数を数えます。次のプログラムは、前節のプログラムを少し変更して、文字数を数えています。変数が n と ch の二つになったので、ch のスコープを for 文でループ内に限定しています。

ソースコード 76: 文字のカウント

```
// 入力された文字数をカウントする
#include <iostream>
int main()
{
   int n {0}; // 入力文字数の合計
   for (char ch{}; std::cin >> std::noskipws >> ch; )
      ++ n;
   std::cout << n <<"\n";
   return 0;
}
```

入力行のカウント

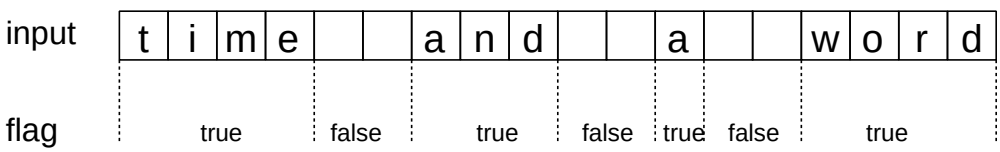
テキストエディタでは改行により文字が 1 行下の位置から表示されます。一方、ファイルの中では上や下という概念はなく、改行文字 ('\n') という特殊文字が区切りとして置かれるだけです。テキストエディタは、改行文字を見つけると、その次の文字の表示場所を隣ではなく 1 行下の先頭に変えます。したがって、入力テキストの行数を知るには、この改行文字を数えれば良いことになります。それには上記のプログラムの数える部分に条件を付けるだけで行えます。改行文字はホワイトスペースの一つなので、入力において noskipws を指定しなければ決して ch には代入されないことに注意してください。

ソースコード 77: 行のカウント

```
// 入力された行数をカウントする
#include <iostream>
int main()
{
   int n {0}; // 入力行数の合計
   for (char ch{}; std::cin >> std::noskipws >> ch; )
      if (ch == '\n')
         ++ n;
   std::cout << n <<"\n";
   return 0;
}
```

単語のカウント

入力中の単語の数を数えることを考えます。これを行うには、まず入力中の文字がホワイトスペースであるか、それ以外の文字であるかを確認しておく必要があります。確認方法はいくつかありますが、flag 変数を次の図のように入力に合わせて変化させることにします。



これまでの延長でプログラムを作ると次のようになります。

ソースコード 78: 単語のカウント

```
// 入力された単語をカウントする
#include <iostream>
int main()
{
   int n {0}; // 単語の数
   bool flag {false}; // 入力文字が単語中の文字かどうかを示す
   for (char ch{}; std::cin >> std::noskipws >> ch; ) {
      if (ch == ' ' || ch == '\n' || ch == '\t')
         flag = false;
      else if (!flag) {
         flag = true; // 単語の先頭が見つかった
         ++ n;
      }
   }
   std::cout << n <<"\n";
   return 0;
}
```

入力文字がホワイトスペースであるか、または、単語の文字であるかに応じて、bool 型の変数 flag を変化させます。変数 flag が、false から true に変化するときが、単語の先頭文字を読み込んだときなので、そのタイミングで変数 n を 1 だけ増やしています。

8.3 文字の変換

プログラムへの入力文字が ASCII コード (American Standard Code for Information Interchange) で表されていると仮定できる場合には、文字変換の処理を計算で行えます。ASCII コードではアルファベットの A〜Z の文字と a〜z の文字に対して、それぞれ順番に連続するコード値が割り当てられており、対応する大文字小文字の値は、その差が一定になっていることが利用できるためです。次のプログラムは、入力された単語の先頭文字を大文字に変換して出力します。前の例の単語数を数えるプログラムが、各単語の先頭文字を見つけていたので、このプログラムではそれを利用して各単語の先頭文字を変換しています。

ソースコード 79: 大文字への変換

```
// 入力された単語の先頭文字を大文字にする
// ASCII コードを仮定する
#include <iostream>
int main()
{
   bool flag {false}; // 入力文字が単語中の文字かどうかを示す
   for (char ch{}; std::cin >> std::noskipws >> ch; ) {
      if (ch == ' ' || ch == '\n' || ch == '\t')
         flag = false;
      else if (!flag) {
         flag = true;
         if (ch >= 'a' && ch <= 'z') // ch が英小文字ならば
            ch = ch - 'a' + 'A'; // それを大文字に変換する
      }
      std::cout << ch;
   }
   return 0;
}
```

大文字と小文字のコード値の差が一定であることから、変数 ch が持つ英小文字を大文字に変換するには、プログラムにあるように、ch-'a'+'A' と計算できます。プログラムの実行例は以下のようになります。

```
can i help you?    <<--- 入力
Can I Help You?    <<--- 出力
```

文字に関するライブラリ関数の利用

<cctype>をインクルードすると、文字処理に関するライブラリ関数が使用できます。例えば、ch == ' ' || ch == '\n' || ch == '\t' という条件は、std::isspace(ch) というライブラリ関数の呼び出しで置き換えることができます。また、英小文字かどうかの判定は std::islower(ch)、大文字かどうかの判定は std::isupper(ch) が使用できます。そして、大文字や小文字への変換は、std::toupper(ch) と std::tolower(ch) が使用できます。これを使って先ほどの例を書き換えてみましょう。

ソースコード 80: 大文字への変換 2

```
1  // 入力された単語の先頭文字を大文字にする
2  #include <iostream>
3  #include <cctype>
4  int main()
5  {
6     bool flag {false}; // 入力文字が単語中の文字かどうかを示す
7     for (char ch{}; std::cin >> std::noskipws >> ch; ) {
8        if (std::isspace(ch))
9           flag = false;
10       else if (!flag) {
11          flag = true;
12          if (std::islower(ch))
13             ch = std::toupper(ch);
14       }
15       std::cout << ch;
16    }
17    return 0;
18 }
```

関数名が処理の内容を表しているので、コメントが不要になりました。プログラムが簡潔になったのは良いのですが、これらの関数はもともと C 言語用に作られているので少し注意が必要です。C 言語の文字処理のライブラリ関数では文字に int 型を使います。文字を変換する toupper(ch) や tolower(ch) の戻り値は int 型なので、std::cout でそのまま出力すると文字ではなく整数値となってしまいます。そのためこの関数の結果を出力する場合には、例の様に結果を一度 char 型の変数に代入してから出力します。

8.4 ホワイトスペースの明示的な読み飛ばし

次のプログラムは入力中の行頭のホワイトスペースと改行のみの行を削除します。このような処理は計測データなどを納めたファイルを処理する場合の前処理として行うことがあります。

ソースコード 81: WS 読み飛ばし

```
1  // 行頭のホワイトスペースと改行のみの行を削除する
2  #include <iostream>
3  int main()
4  {
5     using std::cin, std::ws;
6
7     cin >> ws; // ホワイトスペースを読み飛ばす
8
9     for (char ch{}; cin >> std::noskipws >> ch; ) {
10       std::cout << ch; // ホワイトスペースを含めてそのまま出力
```

```
11        if (ch == '\n')
12           cin >> ws; // 行末だったら次の行頭の空白を読み飛ばす
13     }
14     return 0;
15  }
```

cin >> ws; でホワイトスペースを読み飛ばします。この後の入力は、EOF に到達しない限り、必ずホワイトスペース以外の文字です。ホワイトスペースには改行文字 ('\n') が含まれているので、複数行が一度に読み飛ばされる場合もあります。ws は noskipws と同じくマニピュレータの一つです。実行結果は以下のとおりです。

```
$ cat linehead.dat
   abc
 bcd
       cde

   defg
 efghi
$ ./a.out < linehead.dat
abc
bcd
cde
defg
efghi
```

演習

1. ファイル中のスペース文字 (' ') の数を数えるプログラムを作ってみましょう。
2. 文字列リテラルのように、"Hello C++ World!" という形の文字列のまとまりが、多数含まれているファイルがあるとします。二つの「"」で囲まれた部分を取り出して、1 行ずつ出力するプログラムを作ってみましょう。

8.5 string による単語のカウント

テキスト入力処理と string 型は密接に関係しています。string 型の変数には保持する文字列を 1 文字ずつ取り出す手段が用意されているので、前節の処理を string 型で行うこともできます。例えば、ソースコード 78 の単語数を数えるプログラムは、string を使うと次のように単純に書くことができます。

ソースコード 82: string 版単語カウント

```
1  // 入力された単語をカウントする
2  #include <iostream>
3  int main()
4  {
5     int n {0}; // 入力単語数の合計
6     for (std::string s; std::cin >> s; )
7        ++ n;
8     std::cout << n <<"\n";
9     return 0;
10 }
```

cin >> s は単語単位の入力となるため、入力が成功した回数がそのまま単語の数です。これは、cin からの string 変数への入力でも、ホワイトスペースが読み飛ばされるためです。

8.6 string による英字の大文字変換

次のプログラムは、string 型の変数を用いて、入力中の単語の英字をすべて大文字に変換しています。89 ページの例とは異なり、ホワイトスペースの個数を無視して単語中の文字のみを処理したい場合には、string が有効です。

ソースコード 83: string 版大文字変換

```
// 入力された単語のアルファベットをすべて大文字に変換する
// 1行に 1単語を出力する
#include <iostream>
#include <cctype>
int main()
{
   for (std::string s; std::cin >> s; ) {
      for (auto& ch : s) {
         if (std::islower(ch))
            ch = std::toupper(ch);
      }
      std::cout << s <<"\n";
   }
   return 0;
}
```

8.7 string ストリーム

32 ページで見たように、string から int や double の値を取り出したり、逆に、int や double の値を string にするには、std::to_string()・std::stoi()・std::stod() を使います。これだけでも多くの場合では十分ですが、cin・cout のように文字列を扱えると便利なことがあります。これを可能にするのが<sstream> ヘッダファイルをインクルードして使用する string ストリームです。

以下のプログラムの前半は string 変数から int や double の値を取り出す例です。まず、string 変数の内容を cin のようなストリームとして操作するために、istringstream 型の変数を宣言します。iss が宣言した変数です。取り出し元となる string 変数の str がその初期値となっています。後は、この iss を cin のように用いるだけです。後半は、ostringstream 型の変数 oss を宣言します。この変数は cout のように出力を指定すると、結果を文字列として蓄えていきます。そしてメンバ関数 str() により、その時点の文字列を取りだせます。

ソースコード 84: string ストリーム

```
// string ストリームの基本
#include <iostream>
#include <sstream>
int main()
{
   using std::cout, std::string;
   // cin のような入力変換を行う文字列ストリーム
   string str = "1 2 3.4 2 4 6.8";
   std::istringstream iss {str};
   int x, y; double z;
   while (iss >> x >> y >> z)
      cout << 2*x <<", "<< 2*y <<", "<< 2*z <<"\n";

   // cout のような出力変換を行う文字列ストリーム
   std::ostringstream oss;
   oss <<"abc: "<< 6 <<" "<< 1.5 <<" ";
   string s1 = oss.str(); // この時点の文字列の取り出し
   oss <<"xyz: "<< 7 <<" "<< 2.5; // さらに追加
   string s2 = oss.str(); // この時点の文字列の取り出し
   cout << s1 << " ::: " << s2 <<"\n";
   return 0;
}
```

実行結果は以下のようになります。

```
2, 4, 6.8
4, 8, 13.6
abc: 6 1.5  ::: abc: 6 1.5 xyz: 7 2.5
```

出力の最初の 2 行では文字列から取り出した整数や実数の値が 2 倍になっています。次の出力は abc: 6 1.5 の部分が 2 回出力されていることから、str() で一度文字列を取り出した後でも、oss に << の指定でさらに追加できていることが確認できます。

8.8 1 行ずつの処理: getline()

これまでに見てきた三種のストリームを確認しましょう。

- 入出力ストリーム：istream・ostream 型 (cin・cout)
- ファイルストリーム：ifstream・ofstream 型
- string ストリーム：istringstream・ostringstream 型

これらのストリームは変数の準備の仕方が異なるだけで、<< や >> の演算子を使った指定方法が同じです。さらに getline() という行単位に処理するための関数も共通に使えます。

次のプログラムは、1 行単位で入力を行い、行の数を数え、さらに最も長い行を覚えておきます 1 行単位の入力には getline() 関数を使っています。この関数は、第 1 引数に上記ストリームのいずれかの変数を、第 2 引数には string 変数を、それぞれリファレンス引数としてとります。そして、第 1 引数で指定したストリームから 1 行分の文字を読みだし、第 2 引数の string 型の変数に設定します。ただし、区切り文字の改行文字 ('\n') は読み込むだけで、第 2 引数の line には保存しません。

getline() の戻り値は第 1 引数そのもので、この場合 cin です。ここでは戻された結果を for() 文の条件として使用しているため、入力に失敗した場合には cin が false と評価されてループを終わります。考えられる入力の失敗には、1 文字も入力しないで EOF に到達するか、または、滅多に起きませんが 1 行が長すぎて変数に代入しきれない場合です。

ソースコード 85: 行単位の入力

```
1  // 入力ファイル中の行数を数えて最も長い行を探す。
2  #include <iostream>
3  int main()
4  {
5     int num {0}; // 読み込んだ行の数
6     int maxline_num {0}; // もっとも長い行の行番号
7     std::string maxline; // もっとも長い行の内容
8
9     // 1行ずつ読み込んで処理する
10    for (std::string line; std::getline(std::cin, line); ) {
11       ++ num;
12       if (maxline.size() < line.size()) {
13          maxline = line;
14          maxline_num = num;
15       }
16    }
17
18    // 結果の表示
19    std::cout <<"number of lines: "<< num <<"\n"
20              <<"longest line:" << maxline_num <<"\n"
21              <<"-----> " << maxline <<"\n";
22    return 0;
23 }
```

12 行目の処理に着目してください。これは既に読み込んだ行の最大行の文字数と、読み込んだばかりの行の文字数との比較を行っています。maxline 変数の初期値はどうなっているでしょうか？maxline

変数はループの前で宣言されていますが、その際に初期値が指定されていません。組み込み型の変数とは異なり、string の変数は、宣言のみで初期値が指定されていないと空文字列で初期化されます。したがって、size() メンバ関数は 0 を返し、何かを代入する前でも文字列の長さを比較できます。

行の途中から行末までの入力

getline() という関数は行ごとの処理ですが、正確な内容は、指定されたストリームに対して関数が呼ばれた際の読み取り位置の文字から改行文字 ('\n') またはファイルの末尾までを読み込み、改行文字があればそれを取り除いて string 変数に設定することです。したがって、入力の行頭でこの関数を呼び出せばその 1 行を読み込みますが、何らかの入力を行った後の行の途中で getline() を呼び出せば、そこから行末までが取り出されます。次のプログラムは、ファイルストリームを使って getline() を試しています。入力ファイルの各行にスイーツの売上個数と単価の情報が現れるような場合に、その 2 つの整数の積とスイーツの名前を出力します。

ソースコード 86: 行の途中からの入力

```
1  // 数値と複数単語の文字列が混在した入力データの処理
2  #include <iostream>
3  #include <fstream>
4  int main()
5  {
6     std::ifstream in("input.txt");
7     if (!in) { return 1; }
8     int x, y;
9     std::string s;
10    while (in >> x >> y && std::getline(in, s))
11        std::cout << x * y << s << "\n";
12    return 0;
13 }
```

実行例は以下のようになります。

```
$ cat input.txt
32 300 White Chocolate
42 430 Orange Cookie
53 380 Lemon Macaroons
$ ./a.out
9600 White Chocolate
18060 Orange Cookie
20140 Lemon Macaroons
```

区切り文字ごとの処理

1 行ごとの処理とは、区切り文字を改行文字 ('\n') として処理することです。つまり、改行文字が現れるまで 1 文字ずつ string 変数に入力していきます。getline() は、この区切りとなっている改行文字を別の文字に変更できます。以下のプログラムは、string ストリームを使っており、カンマ (',') を区切り文字として、区切られた文字データを出力します。getline() の第 3 引数として、カンマを指定することで、区切り文字を変更しています。

ソースコード 87: カンマ区切りの入力

```
1  // カンマ区切りデータの読み込み
2  #include <iostream>
3  #include <sstream>
4  int main()
5  {
```

```
    std::string line {"a,b,c,d"};
    std::istringstream iss(line);
    for (std::string s; std::getline(iss, s, ','); )
        std::cout << s <<'\n';
    return 0;
}
```

実行例は以下のようになります。

```
a
b
c
d
```

CSV ファイルの処理

表計算ソフトやデータベースのデータをテキスト形式のファイルに保存する際に、CSV (Commna-Separated Values) と呼ばれる形式がよく使われます。これは古くからデファクトスタンダード（どこかで承認されてたものではないが事実上の標準）として使われているテキスト形式で、2005 年にインターネット技術の標準化を推進する団体が、RFC4180 として、処理するソフトウエアの実装の仕方をまとめています。この規格でもいくつかの変種が認められているのですが、基本的には次のものです。

- レコードと呼ぶ関係するデータを 1 行ごとにまとめる
- レコードが複数の値を持つならばカンマ文字で区切って値を並べる

次のプログラムは CSV ファイルの処理を行うプログラムです。入力ファイル (input.csv) には、前の例と同じ、スイーツの売上個数と単価の情報が CSV 形式で入っており、行ごとに売上金を計算しています。CSV 形式にしたことにより、スイーツの名前を行の後ろに配置する必要がなくなりました。そのためこの例では、スイーツの名前、個数、単価の順としています。

ソースコード 88: CSV ファイルの処理

```
// スイーツの売上金の計算
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
int main()
{
   using std::cout, std::string, std::getline;
   std::ifstream ifs("input.csv");
   if (!ifs) { cout <<"cannot open\n"; return 1; }
   for (string line; getline(ifs, line); ) { // 行ごとに
      std::vector<string> v;
      std::istringstream iss(line);
      for (string s; getline(iss, s, ','); ) // カンマ区切りごとに
         v.push_back(s);
      if (v.size() < 3) {
         cout << "line error\n";
         continue;
      }
      int num {std::stoi(v[1])};
      int price {std::stoi(v[2])};
      cout << v[0] <<": "<< num*price << "\n";
   }
   return 0;
}
```

このプログラムはファイルストリームと string ストリームを使ってデータを読み込んでいくので、それぞれを区別するために、ifs と iss という変数名にしました。12 行目の getline() で 1 行をまとめ

て読み込みます。次に、その 1 行を string ストリームに入れて、16 行目の getline() でカンマ区切りごとに値を取り出し、そして vector に挿入していきます。3 個の値が取り出せたならば、v[0] が名前、v[1] が個数、v[2] が単価であると仮定して、計算結果を出力しています。std::stoi() は 文字列を解釈して int 型の値に変換します。プログラムの 17 行目までは、CSV ファイルを読み込む際の基本として使用できるので、参考にすると良いでしょう。実行例は以下のようになります。

```
$ cat input.csv
White Chocolate, 32, 300
Orange Cookie, 42, 430
Lemon Macaroons, 53, 380
$ ./a.out
White Chocolate: 9600
Orange Cookie: 18060
Lemon Macaroons: 20140
```

9 構造体

組み込み型に加えて、ユーザが後から追加するデータ型を**ユーザ定義型**と呼びます。構造体はそのうちの一つで、任意の型を要素とし、それらをまとめてデータ型とするものです。これまでに学んだ vector の配列は同じ型の要素を並べたもので、各要素の指定に添字を使いました。構造体の場合には要素の型がそれぞれ異なるので、要素ごとに名前をつけて、その名前を使って各要素を指定します。それぞれの要素を構造体の**データメンバ**と呼びます。

9.1 構造体の定義

整数と実数と論理値をまとめたデータ型 MyType を考えてみましょう。ユーザ定義型の名前は変数と区別するために単語の 1 文字目を大文字にする慣習があるので、本書でもそれに従います。構造体 MyType は以下のように定義します。

```
struct MyType {
    int    x;
    double y;
    bool   z;
};
```

ここで struct は if や for などと同じ予約語で、struct という名前を変数や関数の名前に使うことはできません。この定義により MyType という名前で、x, y, z という 3 個のデータメンバ持った新しいデータ型が使えるようになります。

```
MyType a;
```

このように変数 a を宣言すると、メモリ上にこれら 3 種のデータ型を並べたような大きな変数が作られます。つまり、変数 a は整数・実数・bool 値をまとめて保存できる入れ物となります。

注意

関数とは異なり閉じ中括弧 ('}') の後ろにセミコロン (';') が必要です。これは新しいデータ型の定義と同時にその型の変数も宣言できるためです。例えば、T という構造体を定義して、同時にその型の変数 x を宣言するには、次のように書くことができます。

```
struct T {
    int a;
    int b;
    double c;
    std::string s;
} x;
```

この書き方は、struct T { ... }の部分を型名と考えれば、int x; という変数宣言と同じ形です。

9.2 構造体変数の基本

次のプログラムでは、会社スタッフの名前と年齢の情報をまとめて、Staff 型という構造体を定義しています。構造体の各データメンバにアクセスするには、ドット ('.') 演算子を用います。x.name や x.age と指定することで、Staff 型の変数 x 内のメンバにアクセスできます。また、変数 x は int や double といった基本データ型よりも大きなサイズの変数ですが、20 行目のように、y = x; と指定することで代入もでき、さらには、23 行目のように、中括弧で各メンバの値を並べた値リストを代入に指定できます。また、構造体の vector 配列も作ることができます。ただし、構造体を定義しただけでは、cin や cout で直接入出力はできません。また、構造体変数の比較もできません。入力出力や構造体変数の比較はメンバごとに指定するのが基本で、組み込み型と同じように指定するには後の節で見る設定が必要です。

ソースコード 89: 構造体の基本

```
// 構造体の基本
#include <iostream>
#include <vector>

// Staff 構造体の定義 --> Staff 型として使用できる
struct Staff {
   std::string name; // 構造体メンバのname
   int age; // 構造体メンバのage
};

int main()
{
   using std::cout;
   Staff x; // Staff 型の変数宣言
   x.name = "John"; // メンバ変数のアクセスにはドットを使う
   x.age = 38;
   cout << x.name <<", "<< x.age <<"\n";

   Staff y;
   y = x; // 他の変数への代入は全体をまとめて行える
   cout << y.name <<", "<< y.age <<"\n";

   y = {"Jane", 35}; // 値を直接指定しても良い
   cout << y.name <<", "<< y.age <<"\n";

   Staff z;
   std::cin >> z.name >> z.age; // 入力はメンバ変数ごとに行う
   cout << z.name <<", "<< z.age <<"\n";

   std::vector<Staff> list(3); // 構造体のvector 配列(Staff 型の配列)
   list[0] = x; // 配列の各要素は構造体変数そのもの
   list[1] = y;
   list[2] = z;
   for (int i = 0; i < 3; i++)
      cout <<"("<< list[i].name <<", "<< list[i].age <<")\n";
   return 0;
}
```

演習

1. 名前 (name)、学籍番号 (id)、学年 (grade) をメンバに持つ構造体 Student を定義してみましょう。
2. Student 変数 a に対して入力と出力を行うプログラムを書いてみましょう。
3. Student 型の vector 配列を作成して、3 人分の学生データを代入文で設定し、for 文で出力してみましょう。

9.3 構造体変数の初期化

変数はその宣言を行った際に初期値を確定しておくことが大切です。構造体変数も他の種類の変数と同様に宣言と同時に初期値を指定できます。初期値の指定の方法は他の変数と同様です。複数の値が初期値に必要なので、vector 変数と同じく、中括弧の内側に各データメンバの値をカンマで区切って並べます。

次のプログラムは構造体の初期化の例を示したものです。構造体 Employee は string と int をデータメンバに持つので、これを中括弧の中に並べています。変数 y の初期化のように、すでに x のような構造体変数が宣言済みならばその変数を初期値に指定しても構いません。構造体の vector 配列を初期化する場合にも、vector の初期化と構造体の初期化の方法を組み合わせるだけです。最後の範囲 for 文でリファレンスを指定している点に注意しましょう。リファレンスではなく、for(auto e:s) の指定で

もコンパイルはできますが、繰り返しのたびに e には構造体の値がコピーされるのことになります。値のコピーは組み込み型の値よりも負荷の高い処理であることを意識しましょう。

ソースコード 90: 構造体変数の初期化

```
// 被雇用者の年齢と給与
#include <iostream>
#include <vector>

struct Employee {
    std::string name;
    int age, salary;
};

int main()
{
    Employee x {"John", 38, 300000}; // 値による初期化
    Employee y {x}; // 変数による初期化
    y.name = "Tim";

    std::vector<Employee> d {{"Robert", 23, 220000}, {"David", 17, 180000}};
    for (size_t i = 0; i < d.size(); i++)
       std::cout << d[i].name <<", "<< d[i].age <<", "<< d[i].salary <<"\n";

    std::vector<Employee> s { x, y };
    for (const auto& e : s)
       std::cout << e.name <<", "<< e.age <<", "<< e.salary <<"\n";
    return 0;
}
```

データメンバのデフォルト値

60 ページで説明した変数の初期値の話を思い出しましょう。組み込み型の局所変数は変数宣言の際に初期値を指定しないと値が不定となります。構造体の変数もこれまでのような定義の場合には、組み込み型の変数と同じ扱いとなります。しかし、構造体は定義時に初期値を設定しておくことができます。これをデフォルト値と呼びます。構造体のデータメンバのデフォルト値を指定するには、変数宣言と同じく、構造体の定義のところで初期値を指定するだけです。次の例を見てみましょう。

ソースコード 91: データメンバのデフォルト値

```
// データメンバのデフォルト値
#include <iostream>

struct Type1 {
   int x{10}, y{20};
   std::string name {"none"};
};

struct Type2 {
   int x{}, y{};
   std::string name; // デフォルトは空文字列
};

int main()
{
   Type1 a;
   std::cout << a.name <<": "<< a.x <<" "<< a.y <<"\n"; // none: 10 20
   Type2 b;
   std::cout << b.name <<": "<< b.x <<" "<< b.y <<"\n"; // : 0 0
   return 0;
}
```

構造体 Type1 はデータメンバ x, y, name にそれぞれのデフォルト値を指定しています。Type2 は x と y に {} を指定しているので、デフォルト値は 0 です。name は string 型でこれはユーザ定義型であるた

め、デフォルト値がもともと空文字列と指定されています。変数にデフォルト値を持たせることはプログラムの質を高めることにつながります。具体的なデフォルト値がない場合でも {} を指定しておくと良いでしょう。

9.4 構造体や配列を要素に持つ構造体

構造体は任意の型の要素をメンバに持つことができます。例えば、他の構造体や vector 配列などです。次のプログラムはこれを利用して図形のためのデータ型を定義しています。三角形や六角形は点の集まりと考えられます。2 次元の座標系ならば点は名前と x，y の座標値で表せるでしょう。点を表す構造体を作り、その構造体を使って三角形や多角形の構造体を定義できます。このような入れ子の関係を Triangle や Polygon で利用しています。Triangle は Point をそのままメンバとして扱い、Polygon では Point 型の vector 配列をメンバとしています。

ソースコード 92: さまざまな構造体メンバ

```
// 構造体をメンバ要素に持つ構造体
#include <iostream>
#include <vector>
using std::cout, std::string;

struct Point {
    string name;
    int x{}, y{};
};
struct Triangle {
    string name;
    Point A, B, C;
};
struct Polygon {
    string name;
    std::vector<Point> pt;
};

int main()
{
    // 三角形t
    Triangle t {"T1", {"a", 0, 0}, {"b", 3, 0}, {"c", 2, 2}};
    cout << t.name <<": "
         <<"("<< t.A.name <<", "<< t.A.x << ", "<< t.A.y <<"), "
         <<"("<< t.B.name <<", "<< t.B.x << ", "<< t.B.y <<"), "
         <<"("<< t.C.name <<", "<< t.C.x << ", "<< t.C.y <<")\n";
    // 多角形(六角形)h
    Polygon h = {"H1", {{"h1", 0, 0}, {"h2", 3, 0}, {"h3", 4, 2},
                        {"h4", 4, 6}, {"h5", 3, 6}, {"h6", 0, 4}} };
    cout << h.name <<":\n";
    for (size_t i = 0; i < h.pt.size(); i++)
        cout <<" ("<< h.pt[i].name <<", "<< h.pt[i].x << ", "<< h.pt[i].y <<")\n";
    return 0;
}
```

それぞれの変数の宣言とその初期化では、この入れ子の関係を正しく示すように値が並べられています。構造体データメンバも配列要素も同じ中括弧でまとめていることに注意が必要です。慣れてしまえば一貫した記述に迷うことはありませんが、最初のうちはどの部分がデータメンバで、どの部分が配列要素であるかの見分けに苦労するかもしれません。また、t.A.name や h.pt[i].name といったドット ('.') を並べる形で内側のメンバを指定していることにも注意してください。t は Triangle 型、t.A は Point 型、t.A.name は string 型の変数をそれぞれ意味しています。同様に h は Polygon 型、h.pt は Point 型の配列、h.pt[i] は Point 型、そして h.pt[i].name は string 型です。

実行結果は次のようになります。プログラムと出力を対応させてみてください。

```
T1: (a, 0, 0), (b, 3, 0), (c, 2, 2)
H1:
 (h1, 0, 0)
 (h2, 3, 0)
 (h3, 4, 2)
 (h4, 4, 6)
 (h5, 3, 6)
 (h6, 0, 4)
```

9.5 構造体の値渡し引数と戻り値

ユーザ定義型の一つである構造体は、他の基本データ型と同様に値渡しの引数にできます。値渡しとは実引数の値を仮引数側の変数にコピーする方法です。また、構造体の値を戻り値としても使用でき、呼び出された関数の局所変数の値を戻り値とし、呼び出し側で指定した変数に代入できます。

次のプログラムはこれを確認するものです。関数 `input()` では局所変数 `p` に `cin` からの入力値を設定し、`return` 文で戻り値として指定しています。呼び出し側の `main()` 関数では、変数の初期化や代入において、この `input()` を利用しています。また、関数 `output()` では仮引数として構造体 `Point` を指定しています。この仮引数 `g` は変更する必要がないので、`const` とリファレンスの指定が良いのですが、確認のために値渡しを指定しています。この指定では代入文と同じく `output({"abc", 10, 20});` のような呼び出しの指定もできます。

ソースコード 93: 構造体の引数

```
1  // 構造体の値渡し引数と戻り値
2  #include <iostream>
3  #include <vector>
4
5  struct Point {
6     std::string name;
7     int x{}, y{};
8  };
9
10 Point input() // 構造体を返す関数
11 {
12    Point p {"error", 0, 0};
13    std::cin >> p.name >> p.x >> p.y;
14    return p;
15 }
16
17 void output(Point g) // 構造体を値渡しの引数で受け取る
18 {
19    std::cout << g.name <<","<< g.x <<","<< g.y <<"\n";
20 }
21
22 int main()
23 {
24    Point a { input() };
25    output(a);
26    output( {"abc", 10, 20} );
27
28    std::vector<Point> x;
29    for (int i = 0; i < 10; i++) {
30       x.push_back(input());
31       output(x.back());
32    }
33    return 0;
34 }
```

構造体を引数や戻り値とすることは、まとめて複数の値のやりとりできるので便利です。特に関数の戻り値は一つと決められているので、後の節で見るように関数の成功の有無を表す `bool` 値とその結果

の 2 つをまとめた構造体を戻り値とするような使い方が考えられます。しかし、配列をメンバに持つようなサイズの大きな構造体をこの方法で受け渡すのは実行時間の増加を招きます。その場合にはリファレンス引数の使用を考えるべきです。つまり、値のコピーが本当に必要かどうかを常に考えながらプログラムを作る必要があります。

9.6 構造体のリファレンス引数

サイズの大きな構造体の受け渡しにはリファレンス引数が使われます。リファレンス引数は仮引数を実引数の別名に設定するもので、引数の受け渡し時のオーバヘッドが小さくなります。次のプログラムはリファレンス引数を使った関数の例です。リファレンス引数の場合でも const を指定すると、{"Y",3,4} のような値リストが呼び出しに指定できます。逆に、const がなければ値リストの指定はエラーです。そのため、値を更新する update() に値リストを指定をしても、意味が無いだけでなく、コンパイルもできません。

ソースコード 94: 構造体のリファレンス引数

```
// 関数の引数に構造体のリファレンスを使用する例
#include <iostream>
#include <vector>

struct Point {
    std::string name;
    int x{}, y{};
};

void update(Point& p) // リファレンス引数で受け取る
{
    p.x += 2;
    p.y += 3;
}

void print(const Point& p) // 更新なしのリファレンス引数で受け取る
{
    std::cout << p.name <<","<< p.x <<","<< p.y <<"\n";
}

int main()
{
    Point a {"X", 1, 2};
    update(a);
    print(a);
    print( {"Y", 3, 4} );

    std::vector<Point> b {{"A", 3, 4}, {"B", 5, 6}};
    for (auto& x : b) {
       update(x);
       print(x);
    }
    return 0;
}
```

この例にあるように、構造体引数は基本的にリファレンスとして、関数内で引数の値の更新の有無に応じて const 指定を決めるのが良いでしょう。つまり、構造体 T があったときに、その引数の指定として次の 2 つのパターンのどちらかを選ぶということです。

```
void func1(T& x){ ....  }          // 関数内でxを更新する場合
void func2(const T& x){ ....  }    // 関数内でxを更新しない場合
```

9.7 構造体メンバの取り出し

構造体のメンバへのアクセスは、これまでに例を見てきたように、ドット演算子の使用が基本です。しかし、構造体の変数名やメンバ名がそれぞれ長い場合には、メンバにアクセスするための指定が長くなり、改行が必要となるなど、プログラムが見づらくなることがあります。そのような場合には、C++17の構造化バインディング (structured binding, 構造化束縛) という機能が役に立ちます。これは auto を使って宣言した変数にメンバを取り出すものです。

ソースコード 95: 構造体メンバの取り出し

```
1  // auto による構造体メンバの取り出し
2  #include <iostream>
3  #include <vector>
4
5  struct Staff {
6     std::string first_name;
7     int total_year;
8  };
9
10 Staff get()
11 {
12    std::string n;
13    int y;
14    std::cin >> n >> y;
15    return {n, y};
16 }
17
18 int main()
19 {
20    Staff john {"John", 10};
21    std::cout << john.first_name <<" "<< john.total_year <<"\n";
22
23    auto [n,y] {john}; // メンバ値のコピー
24    std::cout << n <<" "<< y <<"\n";
25
26    auto& [nm,ty] {john}; // メンバのリファレンス
27    nm += "son";
28    ++ ty;
29    std::cout << john.first_name <<" "<< john.total_year <<"\n";
30
31    auto [a, b] {get()}; // 関数の結果
32    std::cout << a <<" "<< b <<"\n";
33
34    std::vector<Staff> list {{"John", 5},{"Jane", 8},{"James",3}};
35    for (auto [n, y] : list) // 範囲for 文との組み合わせ
36       std::cout <<"("<< n <<", "<< y <<")\n";
37    return 0;
38 }
```

23 行目は、変数 n と y の宣言です。これらの変数の型と初期値は、{} で指定した構造体変数のメンバと同じになります。これが構造化バインディングと呼ぶ機能です。かぎ括弧の中の変数の数と構造体のメンバの数が一致しなければなりませんが、変数名は自由に決められます。26 行目はリファレンス変数なので、構造体のメンバを変更できることになります。31 行目のように構造体を返す関数の結果を分けるのにも有効です。良く使われるのが、35 行目の書き方で、範囲 for 文をシンプルに表現できるようになります。この部分は前節のリファレンス引数と同じ考え方で、メンバのサイズに応じて const auto& [n, y] としても良いでしょう。

15 行目の書き方にも注意しておきましょう。これは 関数 get() の戻り値の型が Staff と指定されているので、{n, y}の指定で Staff 構造体が作られて関数の戻り値となります。構造体は複数の型の値をまとめるものなので、このように別々の値をまとめたり、逆に取り出したりといった機能になれることが大切になります。

9.8 std::pair

構造体を定義すれば複数の型をまとめた名前付きのデータ型が作れます。ところが、まとめるだけで型に名前を付けるほどでもないデータもあります。特に二つの値をまとめると便利なことがあるため、std::pair と呼ばれるデータ型が事前に用意されています。これを使用するには<utility>ヘッダファイルをインクルードします。まずは使い方の基本を見てみましょう。

```
std::pair<int,double> p1 {10, 1.5};
std::cout << p1.first <<" "<< p1.second <<"\n";

std::pair p2 {10, 1.5};
auto [i, d] {p2};
std::cout << i <<" "<< d <<"\n";
```

変数 p1 と p2 のデータ型はともに std::pair<int,double> です。これは vector のように要素の型によって変わるデータ型で、first と second というメンバを持った構造体です。vector の宣言と同じように、初期値によってメンバの型を特定できるため、変数 p2 のような省略した宣言も可能です。データメンバへのアクセスも first と second を指定すれば良いのですが、これらの単語は文字数が多く指定が面倒です。しかし、前節で紹介した構造化バインディングと組み合わせれば std::pair を覚えておくだけで使えます。

これを踏まえて std::pair が有用とされるプログラムのパターンを見てみましょう。以下のプログラムは最大 10 個までの点データを入力して、vector 配列に格納するものです。入力部分は後でユーザインターフェイスを作り込むことを想定して input() という関数に分けてあります。

ソースコード 96: std::pair と関数

```
1  // std::pair を返す関数の利用
2  #include <iostream>
3  #include <utility>
4  #include <vector>
5
6  struct Point {
7     std::string name;
8     int x{}, y{};
9  };
10
11 auto input() // std::pair を返す関数
12 {
13    Point p;
14    bool f {std::cin >> p.name >> p.x >> p.y};
15    return std::pair{f,p};
16 }
17
18 int main()
19 {
20    const int max = 10;
21    std::vector<Point> v;
22    for (int i = 0; i < max; i++) {
23       if (auto [f, p] {input()}; !f)
24          break;
25       else
26          v.push_back(p);
27    }
28
29    for (auto [n, x, y] : v)
30       std::cout << n <<" "<< x <<" "<< y <<"\n";
31
32    return 0;
33 }
```

まず 11 行目の input() 関数に着目してみましょう。これは戻り値の型が auto になっています。これまで見た auto の例は変数宣言の際の変数の型を初期値から決めるものでした。この例では、関数の戻り値の型を return 文で指定した式の型で決めています。std::pair{f,p}の指定で構造体の値が作られ、

結果は std::pair<bool,Point>型です。これを関数の戻り値の型に指定しても良いのですが、この型名の文字数は多く、しかも型の名前からたいした情報も得られないので auto が有効です。14 行目の変数 f の初期値は、3 個の入力がすべて成功すれば true、どれかが失敗していれば false です。3 個の入力が成功せずに EOF に到達した場合も false です。23 行目ではこの関数を使って入力を行い、結果の構造体から取り出した bool 値が false ならばループを抜けています。この if 文の書き方は 57 ページで説明した、初期化付き変数宣言と条件を分離した書き方の応用です。ここでは auto による変数宣言と組み合わせています。

9.9 構造体の cin/cout 対応

構造体によって作られたユーザ定義型の変数は、基本データ型の変数と同じように宣言・初期化・代入・引数の指定ができました。しかし、cin や cout 使った入出力ではそれぞれのデータメンバを指定する必要があり、このままでは基本データ型と同じようには記述できません。その記述を行うには、<< と >> 演算子に対する設定を事前に行っておきます。具体的には operator>> という名前の関数と operator<< という名前の関数を定義します。前者が cin 用で、後者が cout 用です。これらの関数を用意すると、例えば、構造体変数 p に対して、cin >> p; や cout << p <<"\n"; といった記述ができます。また、これは getline() のところで見たように、cin, cout の入出力ストリームだけでなく、ファイルストリームと string ストリームの三種のストリームで使えます。次のプログラムでその設定方法を見てみましょう。

ソースコード 97: 構造体と cin/cout

```
1  // cin/cout から構造体への入力/出力を可能にする方法
2  #include <iostream>
3
4  struct Staff {
5      std::string name;
6      int age {};
7  };
8
9  std::istream& operator>>(std::istream& in, Staff& s)
10 {
11    return in >> s.name >> s.age;
12 }
13
14 std::ostream& operator<<(std::ostream& out, const Staff& s)
15 {
16    return out <<"("<< s.name <<", "<< s.age <<")";
17 }
18
19 int main()
20 {
21    Staff x;
22    std::cin >> x;
23    std::cout << x <<"\n";
24    return 0;
25 }
```

cin 用の operator>> 関数は、cin の型である istream 型のリファレンス引数と入力に対応させたい構造体のリファレンス引数を持ち、istream のリファレンスを戻り値としています。出力用の operator<< 関数もほぼ同様ですが、cout の型である ostream 型を指定しています。そして、どちらも共通して、第 1 引数を cin や cout に見立てて処理を記述して、その処理の式をそのまま return に指定しています。これだけの指定で main() 関数にあるような基本データ型の変数と同じ入出力の指定ができます。リファレンスを戻り値とする関数は慣れないと何を表しているか分からないと思いますが、とりあえずこのパターンを覚えると良いでしょう。

9.10 構造体変数の比較演算子

構造体は C 言語から引き継いだ考え方なので、その変数の比較はできません。しかし、前節と同じような関数を作ることで a == b などの比較を指定できるようになります。比較に使う演算子は、==, !=, <, <=, >, >= の 6 種類です。それらの演算子に対して operator== のような名前の関数を作ります。引数と戻り値はすべての関数で同じです。比較したい二つの変数を const リファレンスの引数とし、bool 型の値を返すようにします。次の例を見てみましょう。

ソースコード 98: 構造体比較の演算子

```
// 構造体比較のサポート
#include <iostream>

struct Point {
    std::string name;
    int x{}, y{};
};

bool operator==(const Point& a, const Point& b)
{
   return a.x == b.x && a.y == b.y;
}

bool operator<(const Point& a, const Point& b)
{
   return a.x < b.x || (a.x == b.x && a.y < b.y);
}

int main()
{
   Point a{"A", 1, 2}, b{"B", 1, 2}, c{"C", 2, 1};
   if (a == b) std::cout << "a == b\n"; // 出力される
   if (a < b) std::cout << "a < b\n";
   if (a == c) std::cout << "a == c\n";
   if (a < c) std::cout << "a < c\n"; // 出力される
   return 0;
}
```

この例ではデータメンバ name を無視して座標値となるメンバだけを比べています。最初の operator== 関数はメンバの x と y の双方が同じであることを確認しています。operator< 関数では、まず x だけを比較して判定し、x が同じ値の場合には y も比較してから大小を決めています。

比較演算子の設定では、論理演算と一貫性を持たせるために以下の表の関係を満たすべきです。そして、この表を注意深く見ると、< 演算子を定義すれば後は機械的に定まることが分かります。

演算子	代替の計算
a==b	!(a<b) && !(b<a)
a!=b	(a<b) || (b<a)
a<=b	!(b<a)
a>b	b<a
a>=b	!(a<b)

== と != は演算が複雑なので < 演算子とは別に作成した方が良いかもしれませんが、!=は!(a==b) の代替演算が利用できます。それにしても 6 種類の演算子をすべて定義するのは何とも面倒な話です。C++20 ではこれらを 1 行の指定で自動作成する機能が追加されました。対応したコンパイラが広く普及すればその機能を使えるようになりますが、それまでもうしばらく待つことになります。

10　関数の名前

関数とは処理手続きに名前を付けたものです。ひとかたまりの処理に名前を付けて手順を整理することで、全体の見通しを良くすることができます。これは大きなプログラムを作る場合に欠かせない言語機能です。特に関数の名前は重要です。この章では関数につける名前に関する言語の機能を見ていきます。

10.1　関数宣言

これまでの例では、main() 関数をプログラムの最後に定義して、main() から呼び出される incr() や print() などの関数を、main() より前に定義していました。このようなプログラムの書き方をしていれば、main() における他の関数の呼び出しで、その関数の引数と戻り値の型がはっきりします。そして、コンパイラが呼び出し方に間違いがないかをチェックできます。しかし、プログラムを書いていると、main() をファイルの最初の方に書きたい場合もあります。その希望にそって、単に incr() や print() の定義を main() の後に移動させると、main() における関数呼び出しのところがエラーとなります。関数は呼び出しを行う前に宣言が必要というルールがあるためです。この移動を行うには**関数宣言**[5]が必要となります。関数宣言は関数の本体を書かずに呼び出しのチェックに必要な情報のみを記述するものです。これによりコンパイラがその関数の呼び出し方の正しさを確認できるのです。例を見てみましょう。次のプログラムにおいて main() 関数の直前にある 2 行が関数宣言です。

ソースコード 99: 関数宣言の例

```
1  // 関数宣言を使って main 関数をプログラムの前方に書いた例
2  #include <iostream>
3  #include <vector>
4  using std::vector;
5
6  // 関数宣言
7  void incr(vector<int>&, int);
8  void print(const vector<int>&);
9
10 int main()
11 {
12    vector a { 1,2,3,4,5 };
13    print(a);
14    incr(a, 3);
15    print(a);
16    return 0;
17 }
18
19 // 配列の各要素の値を増やす
20 void incr(vector<int>& v, int x)
21 {
22    for (auto& e : v)
23       e += x;
24 }
25
26 // 配列のすべての要素を出力する
27 void print(const vector<int>& v)
28 {
29    for (const auto& e : v)
30       std::cout << e << " ";
31    std::cout << "\n";
32 }
```

関数宣言は、関数定義における関数本体部分 ({と}で囲まれた部分) を、セミコロン (;) に置き換えた形をしています。この宣言によって、関数の名前・引数の数・引数の型・戻り値の型が明確になります。そして、incr() や print() の呼び出しを指定したときに、コンパイラが実引数の型の指定に矛盾がな

[5]関数宣言は関数プロトタイプと呼ばれることもあります。

いかをチェックできます。あとは、incr() や print() の関数定義で、同じ引数の型と戻り値を指定すれば良いのです。逆に、同じにしないとエラーになります。

この例では引数に型だけが指定されていることに注意してください。関数宣言では引数の名前を書いても構わないので、以下のように書くこともできます。

```
void incr(vector<int>& v, int x);
void print(const vector<int>& v);
```

さらに宣言と定義で仮引数の名前が異なっていても構いません。関数宣言中の引数の名前はコンパイラには無視されるからです。引数に関して重要なのは、引数の数とそれぞれの型のみです。また、関数宣言は内容に矛盾がなければ何度書いても構いません。これはヘッダファイルに関数宣言を書いて#includeで取り込む場合に、複数回取り込まれることを許すためです。関数宣言を持ったヘッダファイルを作ると、プログラムファイルの分割をスムーズにするので、大きなプログラムで良く使用されます。このようなヘッダファイルは、関数定義を書いておいて、後から関数の頭書きにあたる部分 (関数の本体以外の部分) をコピーして、関数の一覧にする方法で作られることがあります。そのため、引数に名前を持つ関数宣言が許されているのも面倒を減らします。

これまでの話を正確な言葉でまとめてみましょう。

- 関数は予め宣言されていない限り呼び出しを指定できません。
- 関数宣言は関数の名前・戻り値の型・引数の数とそれぞれの型を規定したものです。
- 関数定義は関数宣言に関数の本体の情報を加えたものです。
- 対応する関数の定義と宣言では、同じ型が指定されていなければなりません。
- 同じ情報の関数宣言は複数回書けます。
- 関数宣言に引数名を指定しても無視されます。

10.2 関数名の多重定義 (オーバーロード)

意味的にまとまりのある処理に名前をつけて、プログラムを独立した複数の関数に分割すると、プログラム全体が整理されて読みやすくなります。特定の細かい処理を独立した関数の中に記述し、その関数の呼び出しを記述すると処理の概要が明確になるからです。これは階層的なプログラムの構成につながります。さらに、関数は重複した記述を減らす目的でも利用します。異なるデータに対して同じ処理を施さなければならないときには、引数付きの関数を作ってしまえば何度も同じ処理を書かずに済みます。そして、そのように書かれたプログラムは、簡潔になり、意味的な間違いの入り込む可能性が低くなります。時には実行時の性能も良くなります。さらにうまく書かれた関数は、自分の他のプログラムや他人のプログラムで利用できるので、再利用という面でも役に立ちます。

さて、関数にどのような名前を付けるべきでしょうか? プログラムの読みやすさを決める重要な考慮事項です。例えば、

```
foobar()
```

のような意味の不明な関数名が並べば、他の人が見て理解できないプログラムになり、時が経過すれば書いた本人すら分からなくなります。また、

```
getUserInputValueFromKeyboard()
```

といった、すべてを説明するような長い名前の関数名が並べば、やはり、読みにくいプログラムになってしまいます。適切な長さで分かりやすい関数名を考えなければなりません。

C++では、関数宣言のところで説明したように、コンパイラが関数呼び出しの間違いをチェックするときに、関数の名前、引数の数とそれぞれの型、そして戻り値の型を調べます。関数呼び出しの指定に対

応する関数定義または宣言が見つからない場合には、どこかに間違いがあるとみなします[6]。さらにこの厳密なチェックを利用して、関数の名前づけに自由度を与えています。具体的には、引数が異なれば同じ名前の関数を複数定義できます。これを関数名の**多重定義**または**オーバーロード (overloading)** と呼びます。

次のプログラムは、引数の異なる二つの `find()` 関数を定義しています。関数定義と呼び出しの対応関係は、引数の型の違いで区別できます。

ソースコード 100: 関数名の多重定義の例

```
// 要素を探す２つの関数（関数名の多重定義の例）
#include <iostream>
#include <vector>
#include <cmath>
using std::vector;

size_t find(const vector<int>&, int);
size_t find(const vector<double>&, double);

int main()
{
   vector ia {3, 6, 2, 8, 5, 1, 2, 9, 3, 7};
   size_t n { find(ia, 5) };
   if (n < ia.size())
      std::cout <<"ia["<< n <<"] = "<< ia[n] <<"\n";

   vector da = {3.10, 6.21, 2.33, 3.66, 1.51,
                5.54, 2.73, 9.88, 3.23, 7.28};
   n = find(da, 11.0/3.0);
   if (n < da.size())
      std::cout <<"da["<< n <<"] = "<< da[n] <<"\n";
   return 0;
}

// 配列の中から指定要素を探し、配列の添字を返す
size_t find(const vector<int>& a, int x)
{
   // 逐次探索でデータを探す
   for (size_t i = 0; i < a.size(); i++)
      if (a[i] == x)
         return i;
   return a.size(); // 見つからない場合
}

// 配列の中から指定要素に近い値を探し、配列の添字を返す
size_t find(const vector<double>& a, double x)
{
   // 逐次探索でデータを探す
   const double epsilon { 1.0e-2 };
   for (size_t i = 0; i < a.size(); i++)
      if (std::abs(a[i] - x) < epsilon) // 処理の仕方がint の場合と異なる
         return i;
   return a.size(); // 見つからない場合
}
```

異なる関数にそれぞれの名前をつけることは、通常では良いことです。しかし、処理の仕方が引数の種類によって異なるだけで概念的に同じような処理の関数には、同じ名前をつけた方が良い場合があります。例えば、`print()`・`find()`・`get()`・`count()` といった名前の関数です。これらの関数名は単純な動詞で、引数を目的語と考えれば、処理の内容はある程度想像がつきます。戻り値の型のみが異なる多重定義は許されていないのですが、引数の数やそれぞれの型を変えれば複数の同じ名前の関数が作成できます。この機能は大規模なプログラムやライブラリの作成で役に立ちます。

次のプログラムは、二つの入力関数 `get()` を定義しています。この例では引数の型だけでなく引数の数も異なっています。(`cin >>a>>b`) という入力を表す式は `bool` 型を必要としているところで使われる

[6] 厳密には後で説明するように引数の型変換も試みて対応する関数宣言を探します。

と、この入力が成功していれば true で、失敗していれば false です。通常は if 文や while 文の条件式で直接使います。C++98 までは関数の戻り値としても使用できたのですが、C++11 以降その使い方はできなくなりました。そこで、bool 型の変数 success を用意して結果をその変数に一度入れてから返すようにしています。また、cin.eof() は入力が EOF に到達していれば true、まだ残りの入力があれば false を返す関数で、すべての入力を読み込んだかどうかを確認しています。

ソースコード 101: 関数名の多重定義の例 2

```
// 2つの入力関数（関数名の多重定義の例）
#include <iostream>
#include <vector>
using std::cin, std::cout, std::vector;

bool get(int&, int&);
bool get(vector<int>&);

int main()
{
   int x, y;
   if (get(x, y))
      cout << "get " << x <<" "<< y <<"\n";
   else
      cout << "input error\n";

   vector<int> data;
   if (get(data)) {
      for (auto x : data)
         cout << x <<"\n";
   } else
      cout << "input error\n";
   return 0;
}

// １個の整数を読み込む
bool get(int& a, int& b)
{
   cout << "Input an integer value :";
   bool success {cin >> a >> b};
   return success;
}

// 配列に整数を読み込む
bool get(vector<int>& a)
{
   cout << "Input integer values :";
   for (int x{}; std::cin >> x; )
      a.push_back(x);
   return cin.eof(); // EOF に達していれば入力はすべて成功
}
```

10.3 デフォルト引数

関数の宣言または定義を行う際に、仮引数に対してデフォルトの値を指定しておくと、その関数の呼び出し時に実引数を省略できます。これは**デフォルト引数**と呼ぶ機能です。例えば、特殊な状況だけ実引数を指定し、その他の通常の呼び出しでは省略して、デフォルトの値で処理を行うような関数を用意できます。

次のプログラムは、デフォルト引数の例を示しています。仮引数側の指定であるデフォルト引数は末尾の引数から指定するのがルールです。このルールにより呼び出し時の実引数の数が関数宣言と異なっていても、どの引数が省略されているのかが分かります。なお、省略する実引数は１個でもすべてでも構いません。

ソースコード 102: デフォルト引数の例

```
// 呼び出し時の引数の省略
#include <iostream>
#include <vector>
using std::cout, std::vector;

// 関数宣言に指定したデフォルト引数
int sum(int, int, int = 0, int = 0); // 2個のデフォルト引数
void print(const vector<int>& a, int start = 0); // 1個のデフォルト引数（引数名あり）

// 関数定義に指定したデフォルト引数
int multiply(int x, int y, int z = 1) { return x*y*z; }

int main()
{
   cout << sum(1, 2, 3, 4) <<' '<< sum(1, 2, 3) <<' '<< sum(1, 2) <<'\n';
   cout << multiply(200, 300) <<' '<< multiply(200, 300, 500) <<'\n';
   vector data {3, 2, 6, 8, 4, 9, 1, 5, 7, 0};
   print(data); // 添字 0からすべてを出力
   print(data, 3); // 添字 3から最後までを出力
   return 0;
}

// ４個の整数の合計
int sum(int a, int b, int c, int d) { return a + b + c + d; }

// 指定位置から最後までの配列の要素を出力
void print(const vector<int>& a, int start)
{
   for (size_t i = start; i < a.size(); i++)
      cout << a[i] <<' ';
   cout <<'\n';
}
```

sum() の関数宣言のように引数名を書かないデフォルト引数と print() のように引数名を書くデフォルト引数のどちらの形式でも構いません。また、multiply() の例のように、関数呼び出しの前側に関数定義を記述するならば、関数定義でデフォルト引数を指定しても構いません。関数定義は関数宣言を含んでいるからです。ただし指定はどこかで一回のみです。このプログラムの出力結果は以下のようになります。

```
10 6 3
60000 30000000
3 2 6 8 4 9 1 5 7 0
8 4 9 1 5 7 0
```

10.4 関数テンプレート

関数名の多重定義を使用すると複数の同じ名前の関数を定義できますが、それぞれを詳しくみると、引数や局所変数の型が異なるだけで、後は同一という場合があります。その場合には、**関数テンプレート**を一つ作成するだけで多重定義と同じ効果が得られます。これができると関数を変更する際に、一つのコードを修正するだけで複数の型に対応した関数の修正が可能となるので、プログラムの管理がしやすくなります。関数テンプレートでは、それぞれの関数の違いを型の引数で指定します。これを**テンプレート引数**と呼びます。

関数名の多重定義の例として、int 型の配列から値を探す find() という関数がありました。この関数は引数の型を変更するだけで、string クラスの vector 配列から文字列を探す関数を作れます。次のプログラムは、int 型と string クラスの双方用の find() 関数を、関数テンプレートによって一つにまとめて記述しています。

ソースコード 103: 関数テンプレートの例

```
// 要素を探す関数テンプレート
#include <iostream>
#include <vector>
#include <cmath>
using std::cout, std::vector, std::string;

// 配列の中から指定要素を探し､配列の添字を返す関数テンプレート
template <typename T>
size_t find(const vector<T>& a, T x)
{
   // 逐次探索でデータを探す
   for (size_t i = 0; i < a.size(); i++)
      if (a[i] == x) return i;
   return a.size(); // 見つからない場合
}

int main()
{
   vector ia {3, 6, 2, 8, 1};
   size_t n0 { find(ia, 8) };
   if (n0 < ia.size()) cout <<"ia["<< n0 <<"] = "<< ia[n0] <<"\n";

   vector<string> sa {"Tokyo", "Osaka", "Fukuoka", "Nagoya", "Sapporo"};
   string city {"Fukuoka"};
   size_t n1 { find(sa, city) };
   size_t n2 { find(sa, string("Nagoya")) }; // find(sa, "Nagoya")はエラー
   size_t n3 { find<string>(sa, "Osaka") };
   if (n1 < sa.size()) cout <<"sa["<< n1 <<"] = "<< sa[n1] <<"\n";
   if (n2 < sa.size()) cout <<"sa["<< n2 <<"] = "<< sa[n2] <<"\n";
   if (n3 < sa.size()) cout <<"sa["<< n3 <<"] = "<< sa[n3] <<"\n";
   return 0;
}
```

関数定義の `template <typename T>` が、関数テンプレートであることを意味しています。`template` と `typename` は予約語です。歴史的な経緯により `typename` は `class` と書くことがあり、文字数が少ないからか `class` の方を好むプログラマが多くいます。関数テンプレートも関数宣言として本体部分を省略できるのですが、通常はこの例のように関数定義の形で記述します。`T` がテンプレート引数であり、変数名と同様に任意の名前を付けられます。残りは通常の関数と同じですが、型名の部分に引数の `T` が指定されています。そして、関数呼び出しのところで指定された実引数の型から、`T` に対応する型がコンパイラによって推測されます。推測がうまく行かない場合に明示的に `T` に対応する型を与えることもできます。

最初の例では `n = find(ia, 8);` という関数呼び出しから `T` を `int` として関数が生成されます。次の呼び出しの `n = find(sa, city);` では、`T` が `string` になります。その次の 3 番目の `find()` の呼び出しでは、`"Nagoya"`が `char` 型の配列であるため第 1 引数の `sa` と `T` を合わせられません。そこで `string()` 指定により、この文字列から `string` 型の値を作っています。これによって、2 番目の `find()` の呼び出しと同じになります。4 番目の `find()` の呼び出しでは、`<string>`と書くことで、`T` を `string` に対応させるように明示的に指定しています。このプログラムの出力結果は以下のようになります。

```
ia[3] = 8
sa[2] = Fukuoka
sa[3] = Nagoya
sa[1] = Osaka
```

関数 `find` は関数の引数の型をテンプレート引数として指定していました。テンプレート引数は型名の指定できるところならばどこに使っても構いません。次の例では、局所変数と戻り値の型にもテンプレート引数を指定しています。`return T();` の部分は `T` 型のデフォルト値つまりゼロに相当する値を返すことになります。

ソースコード 104: 関数テンプレートの例 2

```
// 最小値を見つける関数テンプレート
#include <iostream>
#include <vector>
using std::cout, std::vector;

template<class T>
T find_min(const vector<T>& a)
{
   if (a.empty()) return T(); // T 型のデフォルト値を返す
   T min { a[0] };
   for (size_t i = 1; i < a.size(); i++)
      if (a[i] < min) min = a[i];
   return min;
}

int main()
{
   vector x {3,5,2,4,7};
   cout << find_min(x) <<"\n" ;

   vector y {8.2, 3.4, 1.5, 6.3, 7.5};
   cout << find_min(y) <<"\n";

   vector<std::string> z {"FYI", "EOM", "NRR", "PFA", "BTW"};
   cout << find_min(z) <<"\n";
   return 0;
}
```

この他に関数テンプレートの多重定義や、テンプレート引数のデフォルト引数 (デフォルトの型に相当) という機能もあるのですが、話が複雑になるのでここでは説明をしないことにします。

便利な関数テンプレートライブラリ

<algorithm>ヘッダファイル には様々な関数テンプレートが用意されています。その中でも以下の 3 つは基本的かつ有用な関数なので知っておくと良いでしょう。

```
template<typename T> const T& min(const T& a , const T& b); // aとbの小さい方を返す
template<typename T> const T& max(const T& a , const T& b); // aとbの大きい方を返す
template<typename T> void swap(T& a , T& b); // aとbを入れ換える
```

プログラム例を以下に示します。

ソースコード 105: 便利なテンプレート関数

```
// 予め用意されている関数テンプレート
#include <iostream>
#include <algorithm>
int main()
{
    int x{3}, y{4};
    std::cout <<"min:"<< std::min(x, y) <<", max:"<< std::max(x, y) <<"\n";
    std::swap(x, y);
    std::cout <<"x = "<< x <<", y = "<< y <<"\n";
    return 0;
}
```

実行結果は以下のとおりです。

```
min:3, max:4
x = 4, y = 3
```

10.5 s リテラル

vector は関数ではなく構造体とほぼ同じですが、これもテンプレートで作られています。これまでに、vector やテンプレート関数において、型をコンパイラに推測させる場面がいくつかありました。型名を省略してコンパイラに推測させた方がプログラムが簡潔になります。しかし、vector のところでも前節のテンプレート関数のところでも文字列リテラルと string の関係に注意が必要でした。文字列リテラルは char 型の C 配列 (c-string) です。一方で、string は構造体の一種で文字列リテラルとは異なるデータ型です。ただ、文字列リテラルを使って string 変数の初期値に使用できるので同じデータ型と思ってしまいます。これは string をユーザ定義型として後から加えたところの限界を示しています。

C++がテンプレートの機能で拡張されるようになると、初期値を指定するリテラルの重要性が認識されるようになりました。そして、C++11 からユーザ定義リテラルの機能が追加されて、C++14 から string 用のリテラルが標準で使えるようになりました。string リテラルまたは s リテラルと呼ばれるこの機能により、今までの文字列リテラルと string の混乱をスマートな形で回避できるようになりました。次のプログラムで例を示します。

ソースコード 106: string_literals

```
1  // string_literal の例
2  #include <iostream>
3  #include <vector>
4  using std::cout, std::vector, std::string;
5  using namespace std::string_literals;
6
7  template <typename T>
8  size_t find(const vector<T>& a, T x)
9  {
10    for (size_t i = 0; i < a.size(); i++)
11       if (a[i] == x)
12          return i;
13    return a.size();
14 }
15
16 int main()
17 {
18    vector sa {"Tokyo"s, "Osaka"s, "Fukuoka"s, "Nagoya"s, "Sapporo"s}; // <string>を省略
19    size_t n2 = find(sa, "Nagoya"s); // エラーなくstring でテンプレートを展開
20    if (n2 < sa.size()) cout <<"sa["<< n2 <<"] = "<< sa[n2] <<"\n";
21    return 0;
22 }
```

後から追加された機能なので、5 行目の using 指定がまず必要です。正確にはこれを using ディレクティブと呼びます。これは他の using の指定と同じく、有効範囲を考えて、関数の中でも外でも指定できます。そして、これにより、18 行目や 19 行目のように、"文字列"s の形で後ろに s のついた string 用のリテラルが使えます。この s のことをサフィックス (suffix, 接尾辞) と呼びます。

10.6 関数名に関する注意事項

関数名の多重定義やデフォルト引数では同じ名前の関数を作れますが、戻り値の型のみが異なる同名の別関数は作れません。そのような関数宣言や関数定義が現れるとコンパイラはエラーを出力します。これは、関数の戻り値が必ずしも使われないという慣習があったり、暗黙の型変換によって戻り値が型の異なる変数に代入できたりするためです。

さまざまな方法で定義された同名の関数の中から、コンパイラが関数呼び出しに対応する関数を見つけるには、引数の数と型、および戻り値の型を調べます。これらが正確に一致する関数が見つかるならば問題はありません。しかし、見つからなかった場合には、コンパイラはすぐにエラーとはせず、以下のような型の変換などを試みて一致する関数がないかを調べます。

1. 格上げ：bool や char などを int に、float を double に、 といった基本データ型に対する格上げを試みます。
2. 標準変換：int と double の相互変換や、int から unsigned int (符号無し整数) への変換、そしてポインタに関係する変換を試みます。
3. ユーザが定義した型変換を試みます。
4. 引数の数が不定と宣言された関数に一致するかを調べます（ややこしいのでこれの方法は学びません)。

関数の引数に 0 や 0.0 などのリテラル定数をそのまま指定した場合に、変換によって思わぬエラーになることがあります。例えば、0 は int 型ですが標準変換で double に変換されてしまい、それがエラーの要因になることがあります。

デフォルト引数で可能となる複数の形の関数呼び出しは、関数名の多重定義でも実現できます。関数名の多重定義、デフォルト引数、関数テンプレートのいずれを使うかは、次の指針を参考にすると良いでしょう。

1. 基本的に関数にはそれぞれ別の名前をつける。
2. 関数の処理の仕方が異なり、引数の数や型が異なるが、概念的に同じ処理内容の関数ならば (特に単純な単語で表現できる場合) 関数名の多重定義とする。
3. 引数の数が異なるだけならばデフォルト引数とする。
4. 引数の型のみが異なり処理内容が同じならば関数テンプレートとする。

演習

1. 以下の二つの関数を多重定義で作ってみましょう。

   `int count(const vector<int>& a, int x)` : int 型の vector 配列 a の中にある x の値を数えて返す関数

   `int count(const vector<string>& s, char ch)` : string 型の vector 配列 s の各要素が持つ 文字 ch の数をそれぞれ数え、配列全体に含まれる ch の数の総計を返す関数 (例えば、文字列が 10 個でそれぞれが ch を 3 個ずつ持つならば 30 を返す)

2. 110 ページのデフォルト引数の例で示したプログラムを関数名の多重定義を使って、書き直してみましょう。
3. 関数名の多重定義の例でみた double 型の配列用の find() を、関数テンプレートの例のプログラムに定義して、関数テンプレートと同名の関数が共存できるかどうか確認してみましょう。もし、共存できた場合には、関数テンプレートと通常の関数のどちらが呼び出されるでしょうか？
4. 平面座標系における指定された 2 点間の距離を計算する関数と、ある点の原点からの距離を計算する関数の二つを作るとすると、関数名の多重定義とデフォルト引数とでどちらが良いでしょうか？

11　再帰呼び出し

関数の中では計算・代入・if 文・for 文などの処理の他に、関数呼び出しで制御を別の場所に移すような指定ができます。これまで学んだ関数呼び出しは、別に定義にした関数を呼び出すものでしたが、自分自身を直接または間接的に呼び出すこともできます。これを**再帰呼び出し**といい、再帰呼び出しを含む関数を**再帰的関数**といいます。自分自身を呼び出して何が変わるだろうか？最初に聞いたときにはそう思うかもしれません。ところが再帰呼び出しは数学的な考え方と相性がよく、プログラムを簡潔に表すのに欠かせない機能となっています。

11.1　階乗の計算

階乗の計算について考えてみましょう[7]。階乗とは、非負の整数 n がある時に、1 から n までのすべての整数の積のことで、n! と表します。式で表すと、整数 n の階乗は次のようになります。

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

ただし、$0! = 1$ です。この式を C++の関数で表してみましょう。

```
int fact1(int n)
{
   int x {1};
   for (int i=1; i<=n; i++)
      x *= i;
   return x;
}
```

階乗は英語で factorial なので、fact1 という関数名にしています。for 文によって、1 から n までの整数を初期値 1 の変数 x に掛けています。n が 0 の場合を確認しましょう。この場合には、i<=n が最初から false なので for 文の本体の計算が一度も行われません。そのため x の初期値である 1 が関数の結果となり、これは数学の定義と一致します。考えていた式に 0! の情報がないので別途考える必要がありました。

掛け算を行う順番を変えても結果は同じなので、

$$n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$$

と表しても良いでしょう。順列や組み合わせを考えているときには、このような逆順で式を思い浮かべると思います。その場合には次のような関数となるかもしれません。

```
int fact2(int n)
{
   if (n == 0) return 1;
   int x {n};
   while (-- n > 0)
      x *= n;
   return x;
}
```

逆順にしても 0! の情報が式にないので考えなければなりません。そしてこの場合には、最初の if 文で対応しないとおかしなことになります。その後は、変数 x の初期値を n として、1 まで積を計算するだけです。-- n > 0 に注意しましょう。-- n は、先に n を 1 だけ減らし、減らした結果を次の比較に使います。もし、n--と書くと、次の比較にはそのままの値を使い、その後で n を 1 だけ減らすことになります。++の演算子も同様で、++と--には前置きと後置きの二種類があります。数学的には逆順で考えることはそれほど不自然ではないにも関わらず、このプログラムでは注意しなければならない点が多く、しかも式とプログラムの対応が分かりづらくなりました。

[7] この節は『プログラマの数学』結城浩 著, ソフトバンクパブリッシング刊を参考にしています。

整数 n の階乗は以下のように漸化式を使っても定義できます。

$$n! = \begin{cases} 1 & (n = 0 \text{ の場合}) \\ n \times (n-1)! & (n \geq 1 \text{ の場合}) \end{cases}$$

これも逆順で考えた形ですが 0! も考慮されています。0 以外の n の値に対しては、(n-1)! の値に n を掛けた値を n! としています。この漸化式の定義は再帰呼び出しによりそのままプログラムにできます。

```
int factorial(int n)
{
   if (n == 0) return 1;          // 再帰呼出しの終了条件
   return n * factorial( n-1 );   // 再帰呼出し
}
```

自分自身呼び出していることを明確にするために、関数名を factorial と省略なしにしました。最初の if 文が 0! を決めるものです。n * factorial(n-1) のところで再帰呼び出しが指定されています。これは、factorial(n-1) で計算した (n-1)! の値に n を掛けて、 n! の値としています。つまり、漸化式の定義そのものです。

プログラムの実行により、どのように計算が進むかを確認してみましょう。例えば、2! を計算するために factorial(2) と呼び出すと、

```
return 2 * factorial( 1 );
```

という式で結果を返すことになります。そのため、factorial(1) の値が必要となり、その呼び出しは、

```
return 1 * factorial( 0 );
```

という式を計算することになります。factorial(0) という呼び出しは

```
retun 1;
```

となるので、結果として、

$$2! = 2 \times 1 \quad \times 1$$

という計算結果となります。1 を二回掛けたことになりますが、これも漸化式の定義のとおりです。

if (n == 0) return 1; が自分自身を呼び出す繰り返しの終了条件となっていることに注意してください。この条件がうまく設定されていないと、無限ループと同様に無限に自分自身を呼び出すことになってしまいます。

main() 関数を作って実行してみましょう。

```
int main()
{
   for (int i = 0; i < 14; i++)
      std::cout <<i<<":"<<fact1(i)<<" "<<fact2(i)<<" "<<factorial(i)<<"\n";
   return 0;
}
```

プログラムの実行例は次のようになります。

```
0:1 1 1
1:1 1 1
2:2 2 2
3:6 6 6
4:24 24 24
5:120 120 120
6:720 720 720
7:5040 5040 5040
8:40320 40320 40320
```

```
9:362880 362880 362880
10:3628800 3628800 3628800
11:39916800 39916800 39916800
12:479001600 479001600 479001600
13:1932053504 1932053504 1932053504
```

再帰呼び出しとは関係ありませんが、13!は 6227020800 なので、このプログラムの最後の出力は誤りです。32 ビットの int 型で表せる整数の最大値は $2^{31}-1$ つまり 2147483647 です。そのため 13!は計算できませんでした。このように階乗の値はすぐに大きな値となることを覚えておきましょう。

再帰呼び出しとループ

再帰呼び出しを含むプログラムはループを使って書くことができます。実行性能の点で考えると、再帰呼び出しのプログラムはループのプログラムよりも悪くなります。しかし、階乗の計算プログラムのように、再帰的関数で記述すると簡潔になる場合があります。簡潔なプログラムは誤りを含む可能性が低くなるため、プログラムの開発期間や維持コストを優先して、再帰呼び出しが使われます。

11.2 再帰呼び出しと局所変数

再帰呼び出しのプログラムを学ぶときに、誰もが最初に思う疑問は、自分自身を呼び出した後に仮引数や局所変数が上書きされないだろうかという点です。それを確かめるためのプログラムをみてみましょう。次のプログラムで再帰的関数 recur() は、仮引数 n と局所変数 x を持っています。関数 main() から呼び出されたときには n=3, x=9 です。再帰呼び出しが起こったときに、n や x の値が上書きされそうです。しかし実際には、再帰呼び出しでこれらの変数が上書きされることはありません。それは、

- 関数呼び出しのたびに、その呼び出し用の仮引数や局所変数の場所が新たに割り当てられる

ためです。

ソースコード 107: 再帰呼び出しの特徴

```
1  // 再帰呼び出しの前と後の変数を表示
2  #include <iostream>
3
4  void recur(int n) // recursive call
5  {
6     int x { n*n };
7     std::cout << n <<":"<< x <<"(before)\n";
8
9     if (n < 0) return; // 再帰呼び出しを終了させる条件
10    recur(n-1); // 再帰呼び出し
11
12    std::cout << n <<":"<< x <<"(after)\n";
13 }
14
15 int main()
16 {
17    recur(3);
18    return 0;
19 }
```

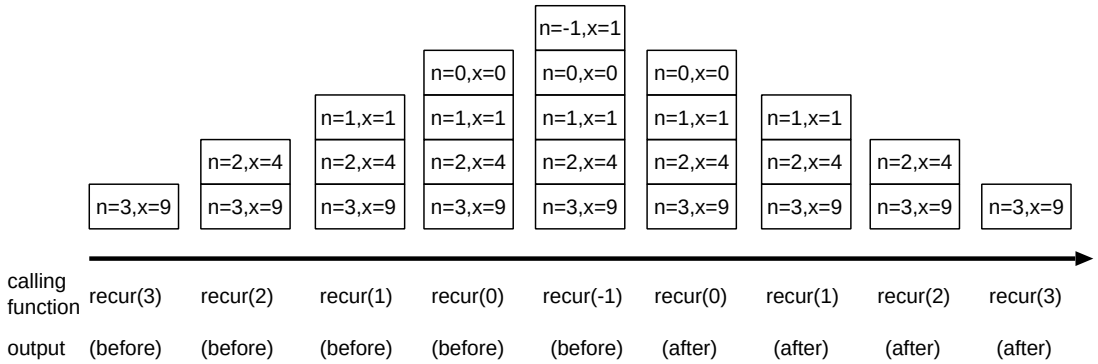
このプログラムの実行結果は次のようになります。

```
3:9(before)
2:4(before)
1:1(before)
0:0(before)
-1:1(before)
```

```
0:0(after)
1:1(after)
2:4(after)
3:9(after)
```

再帰呼び出しの前と後で n と x の値を出力しています。同じ数値の (before) と (after) が対応していて、同一の変数の値を出力していることが分かります。ただし、n=-1, x=1 の場合には、(after) を出力する前に return してしまうので、(before) に対応する (after) の出力がありません。

次の図は関数の呼び出しごとに変数用の場所がどのように割り当てられ、そして関数が return した後にどうなっているかを示しています。最初に関数 main() から recur(3) と呼び出された後には、図の左端のように n=3, x=9 の 2 個の変数の場所だけが割り当てられます。ここで最初の出力がなされ、3:9(before) となります。次に recur(2) として再帰呼び出しが起こったときには、最初の 2 個の変数はそのままで、新たに n=2, n=4 の 2 個の変数の場所が割り当てられます。関数の実行は、図中の積み上げられた変数の一番上の物を利用するため、出力は 2:4(before) となります。以下同様に、-1:1(before) の出力がなされるまで繰り返し関数が呼び出され、そのたびに新たな変数の場所が割り当てられて、図中では四角の枠が積み上げられていきます。



n=-1, x=1 の時に if 文で return すると、一番上の n=-1, x=1 の変数がなくなり、recur(0) の呼び出しの続きが再開されます。このとき、積み上げられた変数の一番上には n=0, x=0 があるため、0:0(after) と出力されます。そして、recur(0) の呼び出しが関数の最後に到達して終了します。そうすると、n=0, x=0 の変数がなくなり、今度は、recur(1) の呼び出しの続きが再開されます。以下同様に呼び出しで使用した変数がなくなっていきます。そして、最後には関数 recur() 用のすべての変数がなくなり、関数 main() に戻ります。

11.3 入力と逆順の出力

前節では、関数の呼び出しごとに局所変数の場所が割り当てられることを学びました。以下のプログラムは、この割り当てを利用して、入力した複数の文字列を入力とは逆の順序で出力しています。関数 reverse_print() は、string 型の変数に文字列を入力します。cin>>s により、入力が成功したら再帰呼び出しを行います。局所変数の s は呼び出しごとに場所が割り当てられるので、入力された文字列はそれぞれの呼び出しの変数 s が保持します。

入力が EOF に達するなどで cin>>s の結果が false になると、if 文が行われず、その呼び出しの関数は終了します。制御が戻ってきた呼び出し元の関数では、自らが保持する s の文字列を出力して終了します。その後は main() に制御が戻るまで入力文字列が逆順に出力されます。

ソースコード 108: 入力と逆順

```
// 入力で得た複数の文字列を逆順に出力
#include <iostream>
void reverse_print()
{
   if (std::string s; std::cin >> s) {
      reverse_print();
      std::cout << s <<"\n";
   }
}

int main()
{
   reverse_print();
   return 0;
}
```

プログラムの実行例は次のようになります。

```
abcd nnnn 12345 xyz        << --- これを入力 (Enter の後に Ctrl-D)
xyz
12345
nnnn
abcd
```

11.4 回文の判定

「たけやぶやけた」のように前から読んでも後ろから読んでも同じになる文を回文といいます。英語は palindrome です。通常は、言葉遊びとしてある程度意味の通る文字の並びを回文といいますが、ここでは文字の並び順だけに着目して、回文の形かどうかを判定するプログラムを考えます。

n 文字の回文は、n-2 文字の回文の両端に同一の文字を置いた形をしています。これは前にみた階乗の定義に似ています。無理矢理に階乗の定義に似せて定義すると以下を満たしたものが回文と言えます。

$$n \text{ 文字の回文} : \begin{cases} \text{空文字列} & (n = 0 \text{ の場合}) \\ \text{任意の } 1 \text{ 文字} & (n = 1 \text{ の場合}) \\ X(n-2 \text{ 文字の回文})X & (n \geq 2 \text{ の場合}, 2 \text{ 個の } X \text{ は同一の任意の } 1 \text{ 文字}) \end{cases}$$

階乗の定義と異なり、n=0 と n=1 の二つの場合があるのは、漸化式に対応する部分が 2 文字ずつ長さを減らして確認しているためです。判定したい元々の文字列の長さが偶数の場合には最終的に n は 0 になり、奇数の場合には 1 となります。ここで、空文字列を回文と言って良いかには議論があるかもしれませんが、空文字列は簡単に排除できるので処理は別に考えます。

次のプログラムの関数 is_palindrome() では、この定義を利用して引数 s が持つ英文字列が回文かどうかを判定しています。最初に空文字列と 1 文字はそれぞれ回文であると判定します。つづく、2 文字以上の場合には、まず両端の文字が同じかどうかを確認します。両端の文字が異なれば回文ではありません。同じならば、両端の 2 文字を除いた内側の文字列を再帰呼び出しで判定し、その結果をそのまま返します。

string 型の substr() メンバ関数は変数から部分文字列を取り出します。s.substr(1, s.size()-2) は、変数 s の s[1] の文字から s.size()-2 個の文字を取り出して部分文字列を作ります。ここではすでに s が 2 文字以上であることを確認した後なので、先頭と末尾を除いた部分文字列が問題なく作れて、再帰呼び出しの実引数となります。

check_palindrome() 関数は先ほど後回しにした問題を解決します。つまり元々の文字列が空文字列の場合を除外する処理です。これを調べてから is_palindrome() の再帰的関数を呼び出します。

```
is_palindrome(s) ? "" : "not "
```

これは空文字列または"not "を関数呼び出しの結果に応じて選択する式です。この結果に応じて変数 x が初期化されます。必ずではありませんが、再帰的関数を作成する場合には、このような前処理を行うもう一つの関数を作ることがあります。再帰呼び出しは繰り返しと似た性質があるために、同じ処理の先頭部分をあわせるような処理が必要になることがあるためです。

ソースコード 109: 回文の判定

```
1  // 回文 (palindrome)を判定する再帰呼び出し関数
2  #include <iostream>
3  using std::string;
4
5  bool is_palindrome(string s) // 再帰呼び出し関数
6  {
7     if (s.size() == 0) return true; // 終了条件１(偶数)
8     if (s.size() == 1) return true; // 終了条件２(奇数)
9     return (s.front() == s.back()) && is_palindrome(s.substr(1, s.size()-2));
10 }
11
12 void check_palindrome(string s) // 空文字を除いて回文を判定
13 {
14    if (s.empty()) {
15       std::cout << "null string\n";
16       return;
17    }
18    string x { is_palindrome(s)?"":"not " };
19    std::cout << s <<" is "<< x <<"a palindrome.\n";
20 }
21
22 int main()
23 {
24    check_palindrome("abccba");
25    check_palindrome("abcdcba");
26    check_palindrome("abcdecba");
27    for (string s; std::cin >> s; )
28       check_palindrome(s);
29    return 0;
30 }
```

11.5 加減算の式の評価

括弧付きの加減算の式を表す文字列を解析して、その式の計算結果を出力するプログラムについて考えます。これはインタプリタが行っている処理で、式の評価と呼ばれるものです。話を簡単にするために、最初は対象を 1 桁の整数の加減算とし、式を表す文字列中にスペースなどの関係のない文字も現れないこととします。

括弧なし加減算の式の評価

準備段階として、括弧なしの加減算の式の評価を行うプログラムを示します。関数 expr() は大域変数の data が持つ式を評価して、その文字列が表す加減算の式の計算結果を返します。変数 data には式となる文字列を保存するのですが、文字列の終わりをはっきりさせるために、ヌル文字 ('\0') を末尾に加えておきます。そのため、例えば、data が"3+4-2+1\0"の場合には expr() が値の 6 を返すようにします。data 中の文字列を調べるために、idx という大域変数をもう一つ用意しています。これは変数 data 中の処理対象の文字の場所を示します。

ソースコード 110: 括弧なし加減算式の評価

```
1  // 一桁数の加減算のみの式の評価
2  // 構文は以下の通り
3  // expr は fact に +fact(または -fact) が 0回以上後に続いたもの
```

```
4  // fact は num
5  #include <iostream>
6  #include <cctype>
7
8  std::string data {"3+4-2+1\0"};
9  int idx {0};
10 char lookahead() { return data[idx]; }
11 void match(char x) { if (data[idx]==x) idx++; else std::cerr <<"error\n"; }
12 int getnum() { return data[idx++] - '0'; } // '0'..'9'の文字を値に変換
13
14 int fact() // 因数は数字
15 {
16    char x { lookahead() };
17    if (std::isdigit(x)) {
18       return getnum();
19    }
20    std::cerr << "error\n";
21    return 0;
22 }
23
24 int expr() // 式は項or 加減の 2 項演算
25 {
26    int left { fact() };
27    while (char op = lookahead()) {
28       if (op == '+') { match('+'); left += fact(); }
29       else if (op == '-') { match('-'); left -= fact(); }
30       else break;
31    }
32    return left;
33 }
34
35 int main()
36 {
37    std::cout << expr() <<"\n";
38    return 0;
39 }
```

最初の 3 つの関数は data と idx を扱う専用関数として働きます。あとで修正してしっかりとした関数に書き換えるので、現時点ではそれぞれ一行の簡易処理のみとなっています。lookahead() は先読みということで、次に現れる文字を返します。match() も次に現れるべき文字を確認する点で似ていますが、仮引数に指定された文字と異なればエラーを出力します。getnum() は idx の場所の数字を数値に変換します。match() と getnum() は、どちらも対象を次の文字に変更するために idx を更新します。これらは次に説明する関数の下請けとして働きます。

再帰呼び出しに関係するのは、関数 expr() とその補助となる関数 fact() です。このプログラムではまだ再帰呼び出しとはなっていません。fact() は因数 (factor) を処理する関数で、式中の個別の数字を扱います。この関数を後で改良していくのですが、最初は単一の処理です。まず lookahead() で対象文字を取り出します。そして、isdigit() でその文字が数字であるかどうかを判定し、数字である場合には getnum() を使って対応する数値を関数の結果として返します。それ以外の場合にはエラーを出力します。

次の expr() は式 (expression) を処理します。式となる文字列では、数字と+-の演算子が交互に現れます。別の見方をすれば、最初に「数字」が現れ、次に「＋数字」または「－数字」が 0 回以上繰り返す形です。そこで expr() ではまず関数 fact() を呼び、最初の数字が表す値を変数 left の初期値とします。次に while 文の条件式で演算子と想定される文字を取り出します。もし、式となる文字列の最後の文字であるヌル文字が現れたならば、ループを終了します。そうでなければ、変数 op は+または-のはずなので、確認したのちに match() で評価対象の文字を更新します。この部分の match() の使い方はやや冗長ですが我慢してください。その次の数字を関数 fact() で処理して、結果を演算子の種類に応じて加算または減算していきます。op が+でも-でもなければループを終了して、それまでに計算した left の値を返します。これで加算や減算が交互に現れる式を評価できるようになりました。

括弧付き加減算の式の評価

次のプログラムは前の加減算の式に加えて、括弧を指定できるように拡張しています。この処理に再帰呼び出しが使われます。括弧付きの式の場合には、括弧の中の式を先に計算しなければなりません。例えば、"4+3+(2-1)"といった式の場合には、4+3 の計算の後に (2-1) を計算し、最後に 7+1 を行うようにします。ここで (2-1) を a と置き換えると全体は 4+3+a という形で、括弧の式 (2-1) は因数です。また、左括弧'(' は文字列の先頭かまたは+-の演算子の次に現れるのみなので、この括弧の式を因数として関数 fact() で扱えます。さらに'(' と')' でくくられた括弧の中の部分式は、関数 expr() で処理できる形になっています。そこで関数 fact() では、左括弧を見付けたならば、以下の処理をするだけで括弧の式を処理できます。

```
    match('(');
    int n { expr() };          // 括弧内の部分式の処理
    match(')');
    return n;
```

これは、左括弧'(' を確認し、括弧の内側の式を expr() で処理し、右括弧')' を確認した後に、expr() の結果をそのまま返す処理です。前のプログラムから変更のあった部分だけを以下に示します。

```
int expr();       // 関数宣言, fact()から再帰的に呼ばれる

int fact()                        // 因数は数字or括弧付きの式
{
   char x { lookahead() };
   if (std::isdigit(x)) {
      return getnum();
   } else if (x == '(') {
      match('(');
      int n { expr() };           // 括弧内の部分式の処理
      match(')');
      return n;
   }
   std::cerr << "error\n";
   return 0;
}
```

関数 fact() の中で expr() を呼び出しますが、もともと expr() は fact() の下側で定義していました。関数呼び出しを指定するには、関数が宣言されていなければなりません。そこで、int expr(); という関数宣言を関数 fact() の前で行っています。

結果として、expr() が fact() を呼び出し、fact() が expr() を再帰的に呼び出す形になりました。関数 fact() から再帰的に expr() が呼ばれていた場合には、expr() では括弧の内側の式だけを対象として計算する必要がありますが、この関数では、+-以外の文字によってループを抜けて return するので、再帰呼び出しでの処理においては、右括弧')' が対象となったところで expr() が戻ります。

11.6 四則演算の式の評価

前節のプログラムをさらに拡張して、掛け算と割り算を加えた、括弧付きの四則演算の式を表す文字列を評価して、その式の計算結果を返す処理を考えます。例えば、"5+4*(5-2)-9/3"といった文字列の場合には、14 という計算結果を返すプログラムです。これができるとインタプリタにかなり近づきます。

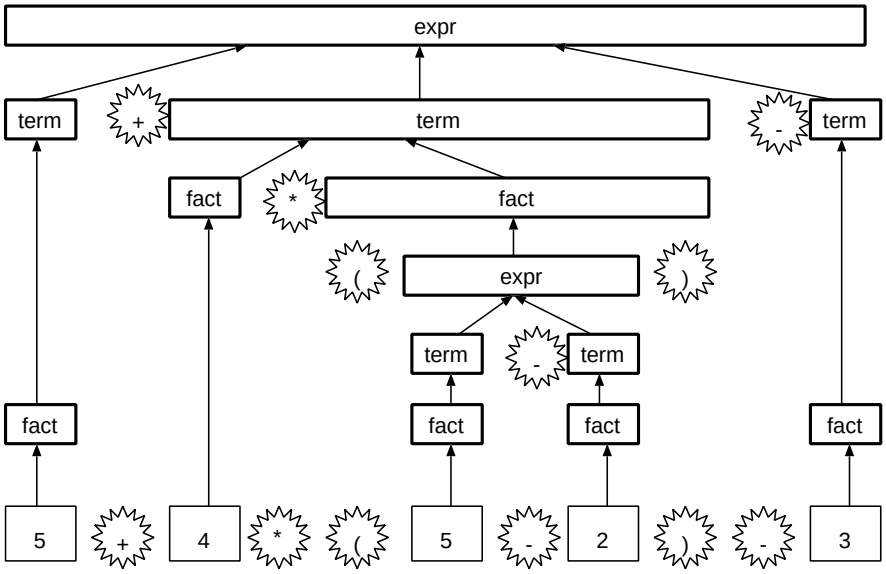
乗除算の演算子 (*/) は加減算の演算子 (+-) よりも優先して計算しなければなりません。しかし、括弧でまとめられた式は、その外側のどの演算子よりも優先して計算しなければなりません。したがって、式"5+4*(5-2)-9/3"では、(5-2) を最初に計算し、次に 4*3 を、そして 5+12 を、さらに 9/3 をそれぞれ計算し、最後に 17-3 をいった順序での計算が必要です。

演算子や括弧の優先順位の関係を理解するために 3 個の用語を定義してみましょう。**因数** (factor, fact と省略) とは、数字そのものか、または括弧でまとめられた式です。**項** (term) とは、乗除算の演算子で

因数をまとめたもので、特別な場合として 1 個の因数も項とします。**式** (expression, expr と省略) とは、加減算の演算子で項をまとめたもので、やはり、特別な場合として 1 個の項を式とします。これら 3 個は、因数, 項, 式の優先順位で値を決定すれば正しい式の結果の値を得ることができます。これらの構成がどうなっているかも含めて定義すると次のようになります。

- fact は 数字　または　'(' expr ')'
- term は fact に　*fact(または　/fact) が 0 回以上後に続いたもの
- expr は term に　+term(または　-term) が 0 回以上後に続いたもの

例えば、"5+4*(5-2)-3"という式をこの構成の定義沿って考えると、式なので 5 + 4*(5-2) - 3 と分けられて、term + term - term の形になります。さらにそれぞれの term を展開すると図ようになります。



図を上側から見ると全体の式は、「term +term -term」の形となっていて、第 2 項は「fact *fact」の形です。さらにこの第 2 項の後側の fact は「(expr)」という形になっていて、その式は、「term -term」の形です。図中の同じ高さの部分が同一の関数で処理する部分で、上下の関係を関数呼び出しで表現すると優先順位の処理ができます。

次のプログラムではこの図の「expr」, 「term」, 「term」の部分を、関数 expr(), term(), fact() という名前の関数でそれぞれ処理するように作られています。関数 expr() は、前節の同名の関数から fact() の呼び出しを term() に置き変えただけです。また関数 term() は、expr() と同じ構造で、扱う演算子と下請けの関数が異なるだけです。前のプログラムとの違いはそれだけです。これで括弧付きの四則演算の式を表す文字列を評価して計算するプログラムができました。

前のプログラムから変更したのは、term() と expr() だけですが、一応、プログラム全体を示しておきます。

ソースコード 111: 四則演算の評価

```
// 式の評価（一桁数の四則演算，0除算のチェックなし）
// 構文は以下の通り
// expr は term に +term(または -term) が0回以上後に続いたもの
// term は fact に *fact(または /fact) が0回以上後に続いたもの
// fact は num または (expr)
#include <iostream>
#include <cctype>

std::string data {"3+4*5-(6*7+8/2)\0"};
int idx {0};
char lookahead() { return data[idx]; }
void match(char x) { if (data[idx]==x) idx++; else std::cerr <<"error\n"; }
int getnum() { return data[idx++] - '0'; } // '0'..'9'の文字を値に変換

int expr(); // 関数宣言，fact()から再帰的に呼ばれる

int fact() // 因数は数字or 括弧付きの式 or
{
   char x { lookahead() };
   if (std::isdigit(x)) {
      return getnum();
   } else if (x == '(') {
      match('(');
      int n { expr() }; // 括弧内の部分式の処理
      match(')');
      return n;
   }
   std::cerr << "error\n";
   return 0;
}

int term() // 項は因数or 乗除の 2 項演算
{
   int left { fact() };
   while (char op = lookahead()) {
      if (op == '*') { match('*'); left *= fact(); }
      else if (op == '/') { match('/'); left /= fact(); } // 0除算のチェックが必要
      else break;
   }
   return left;
}

int expr() // 式は項or 加減の 2 項演算
{
   int left { term() };
   while (char op = lookahead()) {
      if (op == '+') { match('+'); left += term(); }
      else if (op == '-') { match('-'); left -= term(); }
      else break;
   }
   return left;
}

int main()
{
   std::cout << expr() <<"\n";
   return 0;
}
```

ホワイトスペースと複数桁数字への対応

四則演算の式を評価するプログラムができあがりました。しかし、最初に設定した制限によって、数字と演算子の間にスペースを含めることができず、数字も一桁しか指定できません。それらの制限は lookahead()・match()・getnum() の修正で取り去ることができます。

ソースコード 112: 複数桁への対応

```
1  char lookahead()
2  {
3     while (std::isspace(data[idx])) // ホワイトスペースを除外する
4        idx++;
5     return data[idx];
6  }
7
8  void match(char x)
9  {
10    if (lookahead() == x) // ホワイトスペース以外の次の文字を確認
11       idx++;
12    else
13       std::cerr <<"error\n";
14 }
15
16 int getnum()
17 {
18    int n {0};
19    while (std::isdigit(data[idx])) { // 数字が続く限り 10進数の値を計算する
20        n *= 10;
21        n += data[idx++] - '0';
22    }
23    return n;
24 }
```

修正点はコメントで十分と思いますが、以下の点は確認しましょう。idx は data 変数の添字として使われていますが、idx の値が増えるのは、data[idx] が、ホワイトスペース、match() で指定された文字、数字の三つの場合のみです。さらに文字列の最後には必ずヌル文字があります。そのため idx が data.size() の値に達することはない点を確認してください。

インタプリタに向けて

この節のプログラムはこのテキストの最初に説明したインタプリタの中核をなすものです。入力した文字列を式として解釈して、計算した結果を出力することがインタプリタの基本的な役割です。インタプリタに近づくように関数を作成してみましょう。data への入力を行う input() 関数と入力した文字をすべて解析したかどうかを確認する関数 eos() を作ります。eos() が必要なのは、expr() の処理が+-以外の文字を見つけた時点で終わってしまうからです。このままでは、例えば、"1+3&"という誤った文字列に対して 4 を出力して、&に対してエラーを出しません。

```
bool input()
{
   idx = 0;
   data.clear();
   std::cout << ">> ";
   if (!getline(std::cin, data))
      return false;
   data += '\0';
   return true;
}

bool eos() // End of String
{
   return data[idx] == '\0';
}
```

input() により data と idx を設定し、expr() により読み込んだ式の文字列を評価します。expr() から戻った際に文字列中の文字をすべて調べていれば、入力した文字列が正しかったことになります。それを eos() で確認します。main() 関数を次のように修正すると、式を繰り返し入力して評価できるようになります。

```
int main()
{
   while (input()) {
      std::cout << expr();
      std::cout << (eos()?"\n":" error\n");
   }
   return 0;
}
```

なお、完璧を目指すにはもう少しプログラムに追加する必要があります。'/' の除算の処理で 0 除算してしまうとプログラムが異常終了してしまうので、term() 数において fact() の結果が 0 でないことを確認しなければなりません。また、+-演算子は二項演算子としての使い方しか実装していません。-1 や+3 のような単項演算子として機能を有効にするには、fact() を改良しなければならないでしょう。それらの修正は演習として残しておきますので、チャレンジしてみてください。

プログラムを注意深くみると expr()・term()・fact() といった式を処理する関数は、data と idx を直接操作していないことに気がつくと思います。最後に修正した lookahead() などの三つの関数と追加した input()・eos() が、変数 data と idx の操作に責任を持っているのです。

データと対応する操作の結びつき

大規模なプログラムは特定の変数と関数に結びつきを持たせるように設計されます。そうすることで、プログラム全体をグループ化し、それぞれのグループの機能を明確にし、独立した開発や修正を可能にします。C++は、**クラス**と呼ばれる言語機能により、このデータと操作をまとめた形のプログラム設計ができるようになっています。クラスは、string 型の s.size() や s.empty() のようなメンバ関数を持ったユーザ定義型です。クラスについては C++プログラミング II で学びます。それまでは関数や構造体の使い方をしっかり学修してください。

12　アルゴリズム

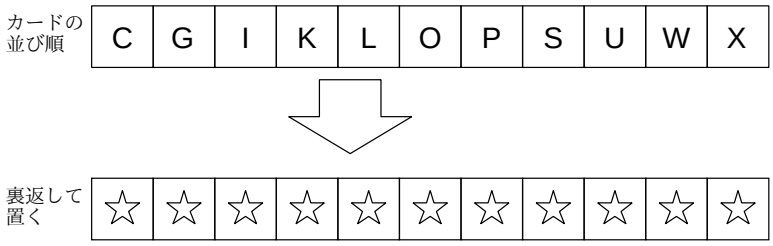
プログラムを作るにはデータ処理の手順を決めなければなりません。この章ではアルゴリズムと呼ばれるいくつかの一般的な処理手順と、アルゴリズムに対応するプログラムを見ていきます。

12.1　2 分探索法

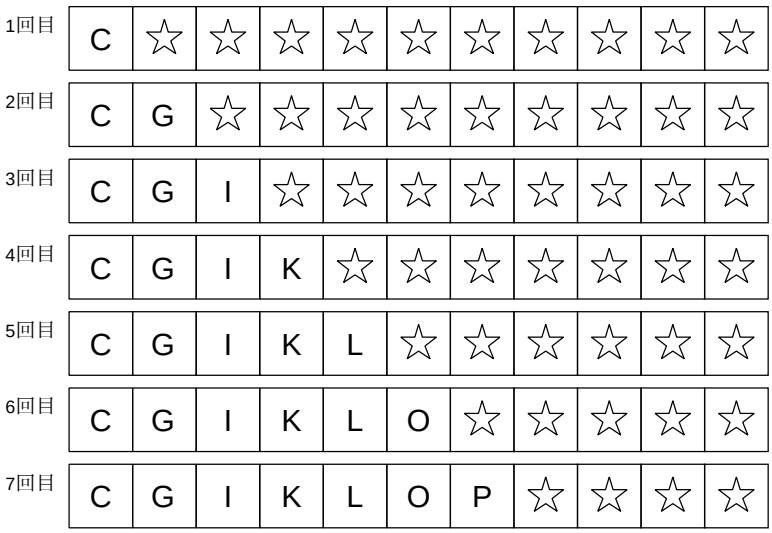
46 ページで逐次探索というデータの探し方をみました。これは並んだデータの先頭から末尾に向かって、1 つ 1 つ順番に、すべてのデータをチェックしていくという方法でした。逐次探索は、いわゆる片端からしらみつぶしに探す方法です。運がよければ、探索を開始してすぐに見つかる場合もありますが、最悪の場合には探索範囲の全体を調べる必要があります。

探す対象となるデータが何らかの順番で並んでいる場合には全データを見ずに判定できます。2 分探索法はそのうちの一つで、探索範囲を狭めながら調べていきます。まず探索範囲を対象データ全体とします。1 回に 1 個のデータを調べて、もし目的のデータが見つからなければ探索範囲を半分に狭めます。この方法では、うまくいけば 1 回で目的のものが見つかる点で逐次探索と同じです。しかし、最悪ケースは大きく違います。1 回調べるごとに探索範囲が半分になるので、探索対象のデータが n 個ある場合に、最悪でも $\log_2 n + 1$ 回だけ調べれば、目的のものが見つかるか、または存在しないことが分かります。

具体的に見ていきましょう。次のような 11 枚のカードが ABC 順に並んでいるとします。ただし裏返してあって、それぞれのカードの内容は見えないことにします。この中から P という文字の書かれたカードを 1 枚ずつ開いて探すことにします。



逐次探索を使ってこの P の文字を探すならば、左から 1 枚ずつカードを覗いて 7 枚目にあることが分かります。

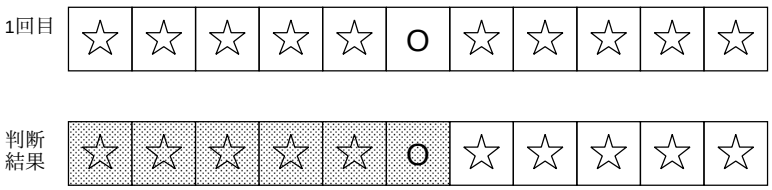


カードがどのように並べられているかの情報がまったくなければ、逐次探索で探すのは妥当です。しかし、裏返してあっても、11 枚のカードが ABC 順に並んでいることが分かっている場合には、次のように効率的に探すことができます。各ステップごとに見ていきましょう。

1. 最初は 1 番目から 11 番目のすべてのカードが探索対象です。まず、11 枚のカードのまん中の 6 番目のカードを調べてみます。6 番目のカードを調べてみると、O の文字が現れます。この事実によって次のように考えを進めることができます。

   (a) 6 番目のカードは P ではなく、O であった。
   (b) カードは ABC 順に並んでいるので、6 番目のカードより左側には A から O のカードしかないはずである。
   (c) したがって、7 番目以降のカードに P があるはずである。まだ見ていない 1 から 5 番目のカードを調べる必要はない。

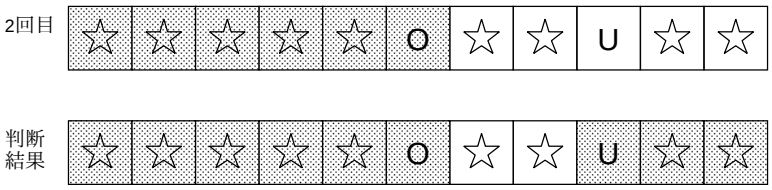
   図の下側の網掛けカードは、間違いなく P ではないと判断できるカードを示しています。これは別の見方をすると、調べる範囲、つまり探索範囲を全体のほぼ半分にしたということになります。「ほぼ」というのは探索対象が 11 から 5 になったからです。



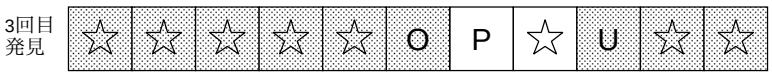
2. 次に 7 番目から 11 番目のカードを探索対象として前のステップと同じことをします。7 番目から 11 番目のカードのまん中は 9 番目です。そこで、9 番目のカードを調べてみると、U が現れます。この事実によって、前のステップと同じように考えを進めることができます。

   (a) 9 番目のカードは P ではなく、U であった。
   (b) カードは ABC 順に並んでいるので, 9 番目のカードより**右**側には U から Z のカードしかないはずである。
   (c) したがって、7 番目から 8 番目のカードに P があるはずである。1 から 5 番目のカードに加えて、まだ見ていない 10 から 11 番目のカードを調べる必要はない。

   下側の図では同じく P ではないと判断できるカードを網掛けにしています。このステップでも探索範囲がほぼ半分になりました。



3. 残り 2 枚になりましたが、これまでと同じ作業をすることにします。7 番目と 8 番目のカードのまん中はありませんが、$(7+8)/2 = 7.5$ なので、小数点以下を切り捨てて 7 ということにしましょう。7 番目のカードを調べてみると、P が現れます。ここで目的のカードが見つかったので探索は終了です。



結果として 3 枚のカードを調べただけで、目的のカードが見つかりました。もし、7 番目のカードが目的のカードでなかったとしても、あと 1 枚調べれば探しているカードがあるか、またはカードがないかがはっきりとします。逐次探索が最悪 11 枚調べなければならないのに対して、2 分探索では 4 枚が最悪の場合の枚数です。

これまで見てきた各ステップはどれも同じ操作だったので、これを一般的な手順としてまとめてみましょう。次のような問題を解決するための手順のまとめを「**アルゴリズム**」と呼びます。

名前 ABC の文字が 1 つずつ書かれたカードの並びを対象とする 2 分探索法
入力 ABC 順に並べられた複数のカード, 探したい文字
出力 並べられたカード中に、探したい文字の書かれたカードがあるかないかの判定
手順
1. 探索範囲の先頭と末尾のカードを覚えておくために、カード全体におけるカードの順序数 (何番目かを示す数) を、それぞれ、$First$ と $Last$ とする。最初は、探索範囲にすべてのカードが含まれるので、$First$ を 1、$Last$ をカードの枚数の数とする。
2. $First \leq Last$ である限り以下の手順を繰り返す。$First > Last$ となった場合は、探したい文字の書かれたカードはないと判定して探索を終了する。
   (a) $\lfloor \frac{First+Last}{2} \rfloor$ を計算し $Middle$ とする。
   (b) 順序数が $Middle$ のカードを調べ、それが探したい文字ならば、カードがあると判定して探索を終了する。
   (c) 探したい文字でなければ、$First$ または $Last$ を次のように更新する。
   - 調べた文字が探したい文字よりも ABC 順で前ならば、$First$ を $Middle + 1$ とする
   - そうでなければ、$Last$ を $Middle - 1$ とする

上記で、$\lfloor x \rfloor$ は床関数と呼ばれるもので、実数 x に対して x 以下の最大の整数を表します。簡単に言えば、小数点以下を切り捨てた値です。ちなみに、$\lceil x \rceil$ は天井関数で、実数 x に対して x 以上の最小の整数を表します。合わせて覚えましょう。

このアルゴリズムは ABC の書かれたカード専用ですが、何らかの順序が決められるものであれば、カードに限らず同じ手順で探索できます。

演習

- 例に挙げた 11 枚のカードを対象に、G を探した場合、K を探した場合、W を探した場合をそれぞれ考え、例に習って各探索ステップを図示してみましょう。
- 例に挙げた 11 枚のカードを対象に、C を探した場合、X を探した場合をそれぞれ考え、例に習って各探索ステップを図示してみましょう。
- 例に挙げた 11 枚のカードを対象に、A を探した場合、J を探した場合、Q を探した場合、Z を探した場合をそれぞれ考え、例に習って各探索ステップを図示してみましょう。
- カードを 12 枚にして探索の様子を調べてみましょう。

C++による 2 分探索法

以下のプログラムは、前節のアルゴリズムを C++のプログラムとして書き直したものです。演習問題を想定して、カードの文字は `string` の文字列で表し、カードの並びを `vector` の配列としています。アルゴリズムの文章で表したものが、そのままプログラムとして書くことができればよいのですが、実際にはいくつか考えなければならないことがあります。2 分探索法のプログラムもほぼアルゴリズムの文章のとおりですが、注意点があるので見ていきましょう。

アルゴリズムでは探索範囲が 1 からカードの枚数でしたが、配列の添字を用いるために範囲を `0` から `a.size()-1` と変更しています。ただ、この変更をしたことで変数の型にも注意が必要になります。

ソースコード 113: 2 分探索法

```
1 // ２分探索を用いて配列から指定の文字列を探す。
2 #include <iostream>
3 #include <vector>
4 using std::string, std::vector;
```

```
5  // ２分探索 (binary search)で文字列データを探す
6  bool binarySearch(const vector<string>& a, string x)
7  {
8     int first{0}, last {int(a.size()-1)};
9
10    while (first <= last) {
11       int middle { (first + last)/2 }; // 探索範囲の中央の添字を計算
12
13       if (a[middle] == x) // 目的の値ならば探索終了
14          return true;
15
16       if (a[middle] < x) // 探索範囲を狭める
17          first = middle + 1;
18       else
19          last = middle - 1;
20    }
21    return false;
22 }
23
24 int main()
25 {
26    vector<string> a {"C", "G", "I", "K", "L", "O", "P", "S", "U", "W", "X"};
27    for (string x; std::cin >> x; )
28       std::cout << x <<" is "<< (binarySearch(a,x)?"":"not ") <<"found.\n";
29    return 0;
30 }
```

配列の添字にはこれまで size_t 型を使う例が多くありましたが、今回は int 型としています。計算が進むにつれて上限と下限を表す first と last が更新されます。first は増加のみ、last は減少のみです。繰り返しを続ける条件は first<=last なので、first が 0 の場合には last が負数にならないと while 文は終了しません。つまり無限ループです。そのため今回は負数を表現できる int 型を用います。

範囲の変数を int 型にした影響で last の初期値が int(a.size()-1) となりました。int(x) という書き方は、x を使って int 型の値を指定する方法です。x は実数値でもよく、強制的な型変換ともいえます。1 章で見たように、C++98 形式の宣言や代入文で型変換してしまえば、表面上は問題がありません。

```
int last = a.size() - 1;        // 警告はない(C++98)
int last; last = a.size() - 1; // 代入文でも警告は出ない
int last {a.size()-1};          // コンパイラが警告を発する(C++11)
```

しかし、C++98 の書き方は型変換が明らかではなく、後で問題が起きた場合に確認の対象とはないづらくなります。後で目立つように、強制的な型変換には、static_cast<int>(x) という書き方もあります。なお、middle の初期値は (first+last)/2 ですが、これはが割り切れない場合には小数点以下が切り捨てられるため、アルゴリズムの文章と整合し、問題もありません。

実は int 型にしたことでもう一つ問題を抱え込んでいます。あるコンパイラでは a.size() の最大値は $2^{64}-1$(つまり 18,446,744,073,709,551,615) で、int 型の正の最大値は $2^{31}-1$(つまり 2,147,483,647) です。データ数が大きい場合には、int 型の last の値は不正になる可能性があるということです。これは長い間動作していたプログラムがある日突然動かなくなる原因の一つです。インターネットなどで人気が殺到して、データ数が急に増える場面で使われれば、そのような事態が起こります。問題が生じたとしても、static_cast<int>() の書き方をしていれば割合簡単に原因となる場所は見つかります。for 文で vector の添字に int 型変数を指定した場合も同様のことが起こりますが、これも前に示したように、コンパイラに警告をださせることができます。

大量データの問題はメモリ不足が先に発生して分かるかもしれませんが、プログラムとして対応できるようにするには、例えば、以下のように修正します。

```
size_t first{0}, last{a.size()-1};

while (first <= last && last < a.size()) {
  auto middle {(first + last)/2};
```

size_t 型を指定することで、a.size()-1 がどんな値であっても last に格納できます。そして、無限ループを避けるために条件を一つ追加しています。符号なし整数の size_t 型では、0 から 1 を引くと負数ではなく、表現できる正の最大値になります。例え a.size() が最大値だとしても、添字は 0 から最大値-1 の範囲なので、この条件で配列の添字の範囲内であることを保証できます。

演習

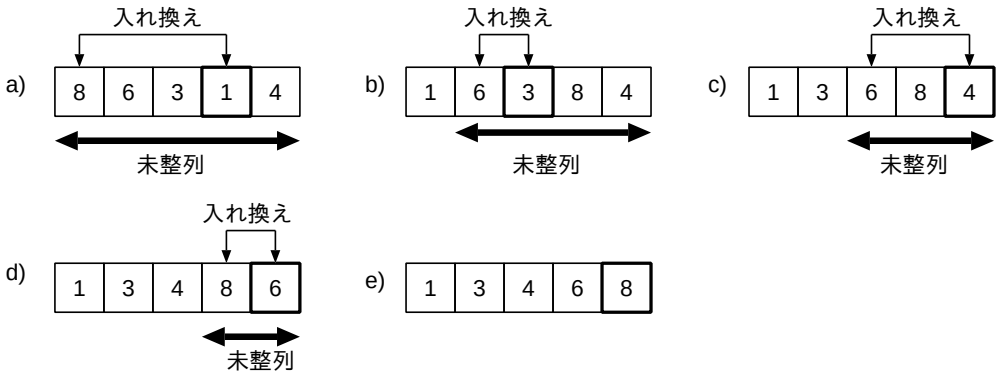
- while 文の最後に std::cout <<first<<" "<<last<<"\n"; を追加して、last が負数になる場合を観察してください。
- 配列の初期値の一部が ABC 順でないデータを作り実行した結果を考察してください。
- template を使って int 型や string 型などさまざまな配列を 2 分探索で探せるようにプログラムを拡張してみましょう。
- 探索対象の配列データをファイルから入力できるようにし、多数のデータで試してみましょう。
- bool binarySearch(const vector<string>& a,int first,int last,string x); を考えて、再帰呼び出しのプログラムを書いてみましょう。

12.2 選択ソート

2 分探索アルゴリズムでは探索範囲のデータが ABC 順に並べられている必要がありました。一般に、与えられた複数個のデータをある基準に従って並べ替えることを**ソート**または**整列** といいます。ソートを行うアルゴリズムは多数ありますが、ここでは比較的単純な**選択ソート**をとりあげます。

選択ソートでは、対象データ全体から 1 番目となるデータを選択し、次に選択されていないデータから 2 番目となるデータを選択し、という手順で、前側にくるべきデータを順々に選択していきます。図を使って具体的な手順を説明します。

始めに{8,6,3,1,4}の順で並んだ数字のカードがあります。この中で最小値を持つカードは、図 a) の太枠が示すように 1 です。そこでこの 1 を選択し、1 番目のカードと入れ換えます。すると図 b) のようになります。ここでは最小値が 1 番目にありますが、それ以外の並びはランダムです。つまり整列していません。この未整列の中から最小値を持つカードを探すと 3 となるので、これを 2 番目に小さいカードとして、2 番目のカードと入れ換えます。以降、同様に未整列のカードから最小値を持つカードを探して、i 番目のカードと入れ換えます。最後まで入れ換えを進めると整列が終了します。それでは、アルゴリズムとして手順をまとめてみましょう。



名前 数字の書かれたカードを昇順に並べ替える選択ソート
入力 ランダムに並べられた数字の書かれたカード
出力 昇順に並べられた数字のカード
手順
1. カードの置かれた場所を示す順序数を i とする。
2. i を 1 からカードの合計枚数-1 まで順に変化させて以下を繰り返す。
   (a) i 番目から最後のカードまで調べて最小値を持つカードを選択する。
   (b) 選択したカードと現在 i 番目に置かれているカードを入れ換える。

C++による選択ソート

次のプログラムは文字を辞書順 (ASCII コード表の順) に並べ替えて出力しています。ライブラリ関数の swap() を使用するために<algorithm>ヘッダファイルをインクルードしています。swap() 関数は sort() の下請けの役割を果たして、vector 配列の二つの要素を入れ替えます。関数 sort() では、添字変数 i を 0 から a.size()-2 まで変化させて、文字の入れ換えを進めます。この他にもアルゴリズムの記述と異なる点があります。まず配列の要素数が 0 の場合にうまくいかないので、最初に除外します。配列の添字が 0 から始まるためにアルゴリズムで使った序数とは異なります。そして、i 番目とそれ以降の文字を比較していくので、内側の for 文のループ変数 j は i+1(i の右隣の文字) から a.size()-1(最後の文字) まで変化させます。

ソースコード 114: 選択ソート

```
// 選択ソート (selection sort)の例
#include <algorithm> // swap()用
#include <iostream>
#include <vector>
void sort(std::vector<int>& a) // 選択ソート (昇順)
{
   if (a.empty())
      return;

   for (size_t i = 0; i < a.size()-1; i++) {
      size_t min { i };
      for (size_t j = i+1; j < a.size(); j++) // i 番目以降で最小値を探す
         if (a[j] < a[min]) min = j;
      std::swap(a[i], a[min]); // i 番目の最小値が決まる
   }
}

int main()
{
    std::vector data {8, 6, 3, 1, 4};
    sort(data);
    for (auto x : data)
       std::cout << x <<" ";
    std::cout << "\n";
    return 0;
}
```

プログラムの初心者の多くはソートのプログラムを単なる二重ループと思ってしまう場合が多いようです。選択ソートも確かに二重ループのプログラムですが、内側のループは最小値を探す役割を持っています。これを強調してプログラムを書き直すと以下のようになります。この書き方にすると、文章で示したアルゴリズムの手順とプログラムがはっきりと対応します。理解を深めるために確認しておきましょう。

```
// i番目以降の最小値を持つ配列要素の添字を返す
size_t find_min(const std::vector<int>& a, size_t i)
```

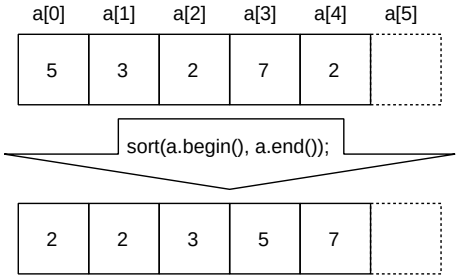
```
{
   size_t min { i };
   for (size_t j = i+1; j < a.size(); j++)
      if (a[j] < a[min]) min = j;
   return min;
}

// 選択ソート(昇順)
void sort(std::vector<int>& a)
{
   if (a.empty()) return;
   for (size_t i = 0; i < a.size()-1; i++) {
      size_t min { find_min(a, i) };  // i番目以降の最小値を探す
      std::swap(a[i], a[min]);        // i番目の最小値が決まる
   }
}
```

12.3 標準アルゴリズムの利用

多くのプログラムでよく使われるアルゴリズムは、標準アルゴリズムと呼ばれる C++のライブラリ関数として利用できるようになっています。それらの関数は<algorithm>ヘッダファイルをインクルードして使います。標準アルゴリズムは大半が関数テンプレートの形で提供されるので、さまざまな型のデータに適用できます。この節では、その中からデータの整列を行う sort() 関数、2 分探索を行う binary_search() 関数、そして逐次探索を行う find() 関数の使い方を見ていきます。

vector 配列の範囲指定

標準アルゴリズムの関数で配列型のデータを処理させるには、配列の範囲を指定する必要があります。vector 配列の範囲は、begin() と end() というメンバ関数を使って指定します。これらは**イテレータ**という種類の値を返す関数です。イテレータは vector などの複数のデータを保持するデータ構造で、繰り返しの処理をするためのものですが、最初のうちは詳しいことを知る必要はありません。begin() が先頭要素を表すもので、end() が**末尾要素の次**を表すものであることだけ覚えておきましょう。次の図は、5 個の要素の int 型の配列を sort() で整列させるときのイメージです。



配列の添字 5 の要素 (a[5]) は存在しないので点線で書いてあります。end() によって a[5] の要素が範囲外であると指定されます。そのため sort() 関数を含めて標準アルゴリズムの関数は、末尾要素が分かり、添字 5 の存在しない要素を操作することはありません。

標準アルゴリズムの sort() を使った例

標準アルゴリズムの void sort() を使った例を次に示します。出力には範囲 for 文を用いていますが、これも実は begin() と end() が関係しています。

ソースコード 115: 標準アルゴリズムのソート sort()

```
// 標準アルゴリズムのsort()を使った整数データの整列
#include <algorithm> // sort()にはこのヘッダファイルが必要
#include <iostream>
#include <vector>
int main()
{
   std::vector a {5, 3, 2, 7, 2};
   std::sort(a.begin(), a.end()); // 標準アルゴリズム
   for (auto x : a)
      std::cout << x <<' ';
   std::cout <<'\n';
   return 0;
}
```

次のプログラムは、sort() 関数でデータを整列させた後に、前節で作った関数 binarySearch() (130 ページ) でデータを探すものです。前節の binarySearch() は、探索対象を size() から決めていて、sort() の範囲指定とは異なっています。また、このプログラムの sort() は 関数テンプレートであるために、一つ前の sort() 関数では vector<int>を、今回は vector<string>型の配列を扱っています。これらの違いをプログラム中で確認しましょう。

ソースコード 116: 標準ソート sort() と 2 分探索

```
// ２分探索を用いて配列内から指定の文字を探す。標準アルゴリズムのsort()を使用。
#include <algorithm>
#include <iostream>
#include <vector>
using std::string, std::vector;
// ２分探索 (binary search)でデータを探す
bool binarySearch(const vector<string>& a, string x)
{
   int first{0}, last{int(a.size()-1)};

   while (first <= last) {
      int middle { (first + last)/2 };
      if (a[middle] == x)
         return true;
      if (a[middle] < x)
         first = middle + 1;
      else
         last = middle - 1;
   }
   return false;
}

int main()
{
   vector<string> a {"Z", "G", "W", "K", "Y", "T", "P", "S", "C", "I", "X"};
   std::sort(a.begin(), a.end());
   for (string x; std::cin >> x; )
      std::cout << x <<" is "<< (binarySearch(a, x)?"":"not ") <<"found.\n";
   return 0;
}
```

構造体配列のソート

標準アルゴリズムとして用意されている関数のほとんどはテンプレートです。そのためさまざまな要素の型を持つ vector の配列を並べ替えることができます。並べ替えをするための条件は一つで、要素となるデータ型が < 演算子を持っていることです。9.10 節では、構造体の変数を < 演算子で比較する方法を示しました。この方法に従い必要な関数定義を行うと、ユーザ定義型である構造体の配列も std::sort() で並べ替えができます。次の例を見てみましょう。

ソースコード 117: 標準ソート sort() と構造体

```
// 並べ替えて出力
#include <algorithm>
#include <iostream>
#include <vector>
using std::vector, std::ostream;

struct Point {
    std::string name;
    int x, y;
};

// 座標値で順序づけを行う
bool operator<(const Point& a, const Point& b)
{
   return a.x < b.x || (a.x == b.x && a.y < b.y);
}

// 出力に対応させる
ostream& operator<<(ostream& out, const Point& a)
{
   return out <<"("<< a.name <<","<< a.x <<","<< a.y <<")";
}

int main()
{
   vector<Point> vp {{"Ari",0,0}, {"Tau",3,1}, {"Gem",2,2},
                     {"Cnc",2,1}, {"Leo",1,0}, {"Vir",2,4},
                     {"Lib",5,8}, {"Sco",4,2}, {"Sgr",3,0},
                     {"Cap",5,2}, {"Aqr",3,5}, {"Psc",4,3} };
   std::sort(vp.begin(), vp.end());
   for (const auto& p : vp)
      std::cout << p <<"\n";
   return 0;
}
```

構造体の変数を比較するために、operator< という名前の関数を定義しています。また、cout の出力用に operator<< という関数もついで定義しています。これによって、int や double の vector 配列と同様に、Point 構造体の配列も std::sort() での並べ替えと出力の指定が可能となっています。出力結果は以下となります。

```
(Ari,0,0)
(Leo,1,0)
(Cnc,2,1)
(Gem,2,2)
(Vir,2,4)
(Sgr,3,0)
(Tau,3,1)
(Aqr,3,5)
(Sco,4,2)
(Psc,4,3)
(Cap,5,2)
(Lib,5,8)
```

標準アルゴリズムの binary_search() を使った例

次のプログラムは、標準アルゴリズムの bool binary_search() を使った例です。この関数は 2 分探索を行う関数テンプレートです。引数の指定を変えるだけで、自作の 2 分探索の関数が不要になり、前のプログラムの半分の長さのプログラムになります。binary_search() の戻り値の型は bool であり、第 3 引数で指定した値が見つかれば true を返し、見つからなければ false を返します。そのためこの関数の呼び出しを、そのまま if 文の条件として指定できます。また sort() と同じように、この関数は <

演算子を持つすべてのデータ型の配列用に使えます。データの等値比較には、!(a<b) && !(b<a) の計算を行います。これの効率が悪い場合には、別の方法ですが等値比較を行うための関数を設定できます。

ソースコード 118: 標準ソート sort() と標準 2 分探索 binary_search()

```
1  // 2分探索を用いて配列内から指定の文字を探す。
2  #include <algorithm>
3  #include <iostream>
4  #include <vector>
5  using std::string, std::vector;
6  int main()
7  {
8     vector<string> a {"Z", "G", "W", "K", "Y", "T", "P", "S", "C", "I", "X"};
9     std::sort(a.begin(), a.end()); // 標準アルゴリズム
10    for (string x; std::cin >> x; ) {
11       bool b { std::binary_search(a.begin(), a.end(), x) }; // 標準アルゴリズム
12       std::cout << x <<" is " << (b ?"":"not ") <<"found.\n";
13    }
14    return 0;
15 }
```

このプログラムを見ていると、配列の初期化の指定以外は、他のデータ型でも同じ内容でプログラムが書けることに気がつきます。例えば、検索対象となるデータがファイルの中にあり、ファイルから読み込む関数を書くならば、string であるとか、Point のような自作の構造体であるとかは、その関数の中だけの話になり、さらに cin や cout に対応する構造体用の関数を書けば、この main 関数の内容からは、データ型を指定するだけで、その他の対象データ固有の処理を取り除くことができます。つまり、データを探して表示するという一連の流れは、データの設定や入出力部分を別にすれば、関数テンプレートで記述できるということになります。

標準アルゴリズムの find() を使った例

次のプログラムは、以前に紹介した、標準アルゴリズムの中で逐次探索を行う find() を使った例です。このアルゴリズムは対象となるデータ型が ==演算子を持つ必要があります。探索が一回で終わるような場合には sort() を使って準備をするよりも、探してしまった方が短時間で処理できます。

find() の戻り値の型は、binary_search() のそれとは異なり、イテレータです。この関数は第 3 引数で指定した値が見つかると、その要素のイテレータの値（場所に関係する情報）を返し、見つからない場合には第 2 引数の値をそのまま返します。第 2 引数は探索範囲の末尾要素の次の情報なので、見つからなかったことを示すのに用います。型名が分からなくても第 2 引数と比較できます。そのため変数が必要ならば auto を用いて変数を宣言します。

ソースコード 119: 標準逐次探索 find() の例

```
1  // 逐次探索を用いて配列内から指定の文字を探す。
2  #include <algorithm>
3  #include <iostream>
4  #include <vector>
5  using std::cout, std::string, std::vector;
6  int main()
7  {
8     vector<string> a {"Z", "G", "W", "K", "Y", "T", "P", "S", "C", "I", "X"};
9     for (string x; std::cin >> x; ) {
10       auto it { std::find(a.begin(), a.end(), x) }; // 標準アルゴリズム
11       cout << x <<" is "<< (it != a.end()?"":"not ") <<"found.\n";
12    }
13    return 0;
14 }
```

find() は要素を見つけたときにイテレータと呼ばれる場所情報を示す値を返すので、見つけた要素に対して何らかの処理を行うこともできます。しかし、その要素に対する処理を記述するには、イテレー

タについてもう少し学ぶ必要があります。そのため現段階では、binary_search() と同じように、対象要素の有無を調べる関数としての使い方を覚えましょう。つまり、以下のように考えて使用します。

- find() が第 2 引数と同じ値を返したら探索失敗
- それ以外の値を返したら探索成功

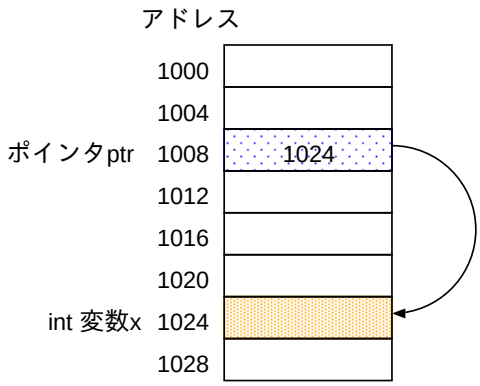
binary_search() と find() の選択

binary_search() と find() の探索時間を比べれば、binary_search() の方が平均的に短いといえます。しかし、binary_search() はデータが整列していなければいけないという前提条件があります。そして、整列には sort() などが必要になり、これには find() を何度か呼び出すのと同じくらいの時間がかかります。したがって、プログラムの中で、binary_search() と find() のどちらを使うかを考えるときには、この整列の時間も考慮に入れなければなりません。いろいろな状況が考えられますが、まずは、次の点を考えてみるのが良いでしょう。

- データを入力した時点でそれらが整列しているのか？
- 探索を何度も行う必要があるか？
- データを並びかえてしまって問題がないか？

13　ポインタの基本

これまで変数は値のための入れ物であると説明してきました。この入れ物はタグに相当する変数名によって特定でき、その名前を使って内容物を操作できます。一方で、変数は多くの場合においてコンピュータのメモリ上に作られるので、メモリ上の場所を表す「**アドレス**」によって特定でき、かつ、その内容物を操作できます。「**ポインタ**」はそのアドレスを覚えておくための変数です。アドレスを使って間接的に他の変数を指定できるのでポインタ (指し示すもの) と呼ばれています。ポインタは C 言語からある組み込み型で、int 型と同様に単一の値として扱われます。



右の図はポインタと int 型変数 x の関係を表しています。この図では変数 x がメモリ上に作られていて、そのアドレスは 1024 番地です。ptr という変数は、1024 という数値を保存しており、これはまさに変数 x が置かれた場所を指しています。この関係から変数 ptr を使うことで、変数 x の値を取り出したり、変更したりといった操作ができるのです。

13.1　ポインタの宣言

メモリアドレスは非負の整数で、ポインタはそのメモリアドレスを保存する変数です。どのような変数もアドレスで場所の特定ができるので、ポインタは単一の種類に思えるかもしれません。しかし、これまで見てきた通常の変数と同様に、ポインタにも型があります。それは指し示す変数のメモリ中の場所情報の他に、指し示す変数の型情報が必要なためです。例えば、

- int 型変数のアドレス
- string 型変数のアドレス

という二種の変数のアドレスの値は、どちらもメモリのアドレスということでは同じです。しかし、指し示す先の変数の種類が異なるので、異なる型のポインタ変数に格納しなければなりません。ポインタの宣言の例を以下に示します。

```
// 型    *変数名;
int      *p1;     // int型変数を指すためのポインタ
char     *p2;     // char型変数を指すためのポインタ
double   *p3;     // double型変数を指すためのポインタ
string   *p4;     // string型変数を指すためのポインタ
```

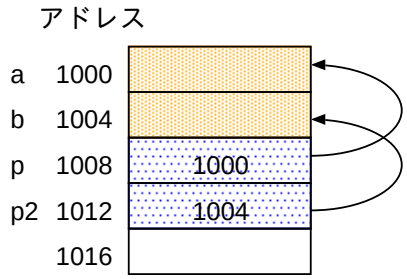
これらの宣言で、変数 p1 は int*型、 p2 は char*型、p3 は double*型、p4 は string*型のポインタとなります。なお、*の両側のスペース文字はあってもなくても良いため、以下の宣言はどれも同じです。

```
int       *ptr;    // *の前にスペース文字がある
int*      ptr;     // *の後ろにスペース文字がある
int*ptr;           // *の両側にスペース文字がない
```

13.2　アドレス演算子 &

ポインタ変数の値となる対象変数のアドレスを得るには、アドレス演算子と呼ばれる & 記号を使用します。この演算子のオペランドには左辺値が必要です。次の例では、ポインタ変数 p の宣言と同時に、その初期値として変数 a のアドレスを指定しています。また、宣言済みのポインタ変数 p2 に対しては、変数 b のアドレスを代入しています。一番下の行ではアドレスを関数の実引数として指定しています。この関数の仮引数の型は int*型でなければなりません。

```
int a, b;
int *p { &a }; //ポインタpの初期化
int *p2;
p2 = &b;    // 代入
func(&a, &b); // 引数
```



上の右側の図はこれらの結果のイメージです。変数 a，b，p，p2 にはそれぞれのメモリアドレスが割り当てられています（この図のアドレスは適当な値です)。そして、変数 p が保持する値は変数 a のアドレスであり、変数 p2 の値は変数 b のアドレスであるため、それぞれ矢印が示す関係になります。

& 記号の使われ方

C++において & という記号は 3 種類の意味で使われます。

- リファレンス指定: 変数の宣言で & を指定すると、宣言した変数は初期値変数のリファレンスとなります。これは仮引数の宣言も同様で実引数のリファレンスを指定したことになります。

```
int& x { a };                // x はaの別名となる
void func(int& n) { ... }    // 仮引数n は実引数の別名となる
```

- アドレス取得: 式において変数の直前に & を指定すると、その変数のアドレスを取得する演算子を意味します。演算結果はポインタ変数に代入できる型の値です。

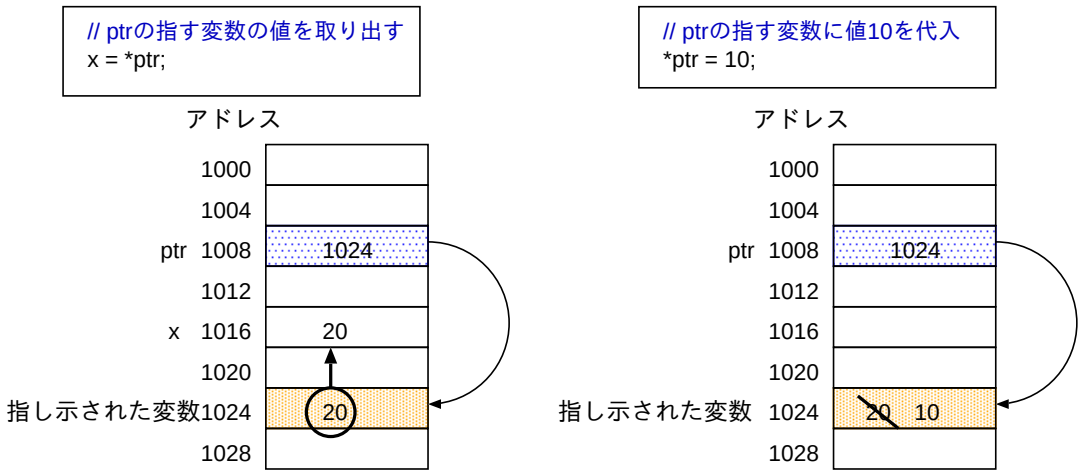
```
ptr = &n;      // 変数nのアドレスをポインタ変数ptrに代入
func(&x);      // 実引数は変数xのアドレス
func(&a[0]);   // 実引数は配列aの添字０の要素(a[0])のアドレス
```

- ビットごとの論理積:式において二つの整数系データ型 (bool, char, int など) の値の間に & を指定すると、二項演算子として両者のビットごとの論理積を意味します。演算結果はオペランドと同じ整数データ系の型の値です。

```
x = y & z;    // yとzのビットごとの論理積をxに代入
```

13.3 間接演算子＊

ポインタ変数の値であるアドレスは同じ型の他の変数に代入できます。保持する値を他の変数に代入できるという点は、int や double などの型の変数と同じです。ポインタ変数には、これに加えて、指し示した先の変数を間接的に操作する機能があります。間接演算子（または逆参照演算子, 参照剥がし演算子) と呼ばれる * 記号の指定がその間接操作を可能にします。間接演算子の結果は左辺値として指定できるので、指し示した変数に値を代入できます。次の図は間接演算子の例を示しています。



左の図はポインタ変数の指す変数の値を取り出す場合で、右の図はこれとは反対にポインタ変数の指す変数に値を代入している例です。

＊記号の使われ方

C++において * という記号は 3 種類の意味で使われます。

- ポインタ変数の宣言: 引数を含めて変数の宣言で * を指定すると、宣言した変数はポインタ変数であること意味します。

```
int *x;                          // x はポインタ変数
void func(int *n) { ... }        // 仮引数n はポインタ変数
```

- 間接参照: 式においてポインタ型の変数の直前に * を指定すると、そのアドレスの示す先の変数を操作する演算子と解釈されます。これは変数だけでなくポインタ型の結果となる式の直前にも指定できます。演算結果の型はポインタ型から決まります。

```
n = *ptr;       // nにポインタ変数ptrの指す変数の値を代入
*ptr = 10;      // ポインタ変数ptrの指す変数の値に10を代入
```

- 乗算: 式において二つの算術型データ (char, int, double など) の間に * を指定すると、二項演算子として両者の乗算を意味します。

```
x = y * z;    // yとzの乗算結果をxに代入
```

x = y**p; は、乗算と間接参照の組み合わせを意味します[8]。

[8] Python とは異なり累乗を意味しないので注意しましょう

13.4 ポインタ変数が取り得る値

ポインタ変数は他の変数を指し示すためにあります。この種類の変数が保持できる値は変数のアドレスです。無効なアドレスを保持したポインタ変数に、間接演算子を適用すると、プログラムが暴走したり異常終了する結果につながります。これは初期化せずに宣言したポインタで特に問題となります。ポインタに限らず局所変数は初期化せずに宣言すると、その局所変数の値は不定です。つまり、宣言時に初期値を与えず、さらに代入も行わなかったポインタ変数に対して間接演算子を適用すると、プログラムの動作はどうなるか分かりません。

```
int *p;      // 宣言しただけではpの値は不定
....         // なんらかの操作
*p = 10;     // 危険!  pは何かの変数を指しているか?
```

ヌルポインタ

プログラムによってはポインタ変数がどの変数も指していないという状況を表現したい場合があります。これには**ヌルポインタ** と呼ばれるアドレスのゼロを値に持つポインタ変数を使います。値はゼロですが、その変数の初期化・代入・比較には nullptr という予約語を用います。この予約語は特殊で int* や double*などのポインタ型の種類とは関係なく、ただアドレスとしての値ゼロを表します。

```
int *ptr {nullptr};              // ポインタはどの変数も指していない
....                             // なんらかの操作
if (ptr != nullptr) *ptr = 10;   // チェックしてから利用する
```

C++98 まではこの予約語がなく、リテラルの 0 を代わりに使っていました。ポインタ変数に対する 0 の代入と 0 との比較が可能なのです。しかし、0 というリテラルは int 型を表すので、トラブルを抱える可能性が指摘されて、この予約語が作られました。この予約語の使用はテキストエディタで探しやすい点でも有功です。互換性を考慮してポインタへの 0 の代入は未だに使用できるので、目にすることはあると思いますが、使用しないようにしましょう。

推奨されるポインタの使い方

- ポインタ宣言は初期化とともに。
  - 指すべき変数が決まるまで可能な限りポインタ宣言をしない。
  - 指すべき変数が決まる前にポインタ宣言が必要ならば nullptr で初期化する。
- ポインタ変数の値が nullptr でないことを確認してから使う
  - nullptr で初期化したポインタ変数の場合
  - 仮引数がポインタ変数の場合

13.5 ポインタの使用例

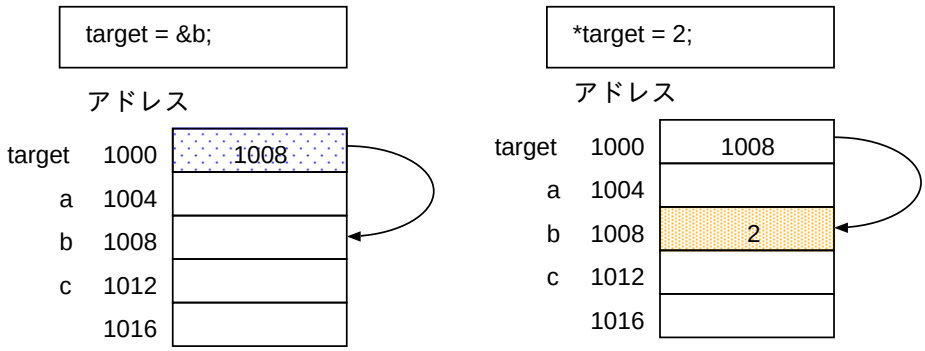
基本的な使用

次のプログラムはポインタの基本的な使用方法を示しています。ポインタ変数 target が指し示す先の変数は、初期化や代入によるプログラムの進行状況によって、a, b, c と変わっていきます。そのため *target に対する代入は、状況に応じて異なる変数を更新します。

ソースコード 120: ポインタによる変数の更新

```
1  // 一つのポインタ変数を使って複数の変数を更新する
2  #include <iostream>
3  int main()
4  {
5     int a{}, b{}, c{};
6
7     int *target { &a }; // 初期値はa のアドレス
8     *target = 1; // a を更新
9
10    target = &b; // b のアドレスに切り替え
11    *target = 2; // b を更新
12
13    target = &c; // c のアドレスに切り替え
14    *target = 3; // c を更新
15
16    std::cout <<"a = "<< a <<", " <<"b = "<< b <<", " <<"c = "<< c <<"\n";
17    return 0;
18 }
```

target 変数そのものへの代入 (初期化を含む) と*target への代入は明確に異なるということに注意しましょう。下の左の図が示すように、target=&b; によって変化するのは、変数 target です。そして、*target = 2; によって変化するのは target が指し示す先の変数 b です。



ポインタを大域変数とする例

前のプログラムを拡張して、入力の値に応じて数を増やす変数を切替えるプログラムを考えてみます。このプログラムでは、ポインタ変数 target が大域変数となっています。大域変数は、その変数の宣言より後方で定義されるどの関数でも用いることができます。そこで countup() をいう関数を用意して、その関数が呼び出された時点で対象となっている変数を増加させます。一方で、main() 関数は入力に応じて、対象となる変数を切替えています。プログラムの開始時や不正な入力がなされたときには、増加させる変数はありません。そのため、target 変数に nullptr を代入して関数 countup() による誤った更新を防ぎます。

ソースコード 121: ポインタを大域変数として使う例

```
1  // 入力の種類別にカウントするプログラム
2  #include <iostream>
```

```
3  int *target {nullptr}; // 初めは何も対象としない
4
5  void countup()
6  {
7     if (target != nullptr)
8        ++(*target);
9  }
10
11 int main()
12 {
13    int a{}, b{}, c{};
14    for (char ch; std::cin >> ch; ) {
15       switch (ch) {
16       case 'a': target = &a; break;
17       case 'b': target = &b; break;
18       case 'c': target = &c; break;
19       default:
20          std::cout <<"input error: "<< ch <<"\n";
21          target = nullptr;
22          break;
23       }
24       countup();
25    }
26
27    std::cout <<"a = "<< a <<", " <<"b = "<< b <<", " <<"c = "<< c <<"\n";
28    return 0;
29 }
```

大域変数はさまざまな関数で共通にひとつの変数を操作できます。その意味では便利なのですが、この便利さが大きなプログラムでは問題につながります。大きなプログラムでは、不具合が生じたときに、多数の関数のどこで大域変数を更新したかが、把握しづらくなるからです。

ポインタを引数とする例

次の例はポインタの仮引数とすることで問題となる大域変数を排除した例です。関数 countup() の仮引数において int *target とすることで、ポインタを用いています。関数呼び出しの方では、実引数として、&a や&b と指定することで、これらの変数のアドレスを関数に渡しています。

このプログラムでは、関数 countup() における target 変数の nullptr との比較は冗長かもしれません。しかし、この関数を他のプログラムで利用したり、拡張する可能性があるならば、その指定は無駄とはならないでしょう。

ソースコード 122: ポインタを引数として使う例

```
1  // 入力の種類別にカウントするプログラム
2  #include <iostream>
3  void countup(int *target)
4  {
5     if (target != nullptr)
6        ++(*target);
7  }
8
9  int main()
10 {
11    int a{}, b{}, c{};
12    for (char ch; std::cin >> ch; ) {
13       switch (ch) {
14       case 'a': countup(&a); break;
15       case 'b': countup(&b); break;
16       case 'c': countup(&c); break;
17       default:
18          std::cout <<"input error: "<< ch <<"\n";
19          break;
20       }
21    }
22
```

```
23     std::cout <<"a = "<< a <<", " <<"b = "<< b <<", " <<"c = "<< c <<"\n";
24     return 0;
25 }
```

vector 要素のアドレスを実引数とする例

vector の各要素もそれぞれアドレスが与えられています。そのため、& 演算子によってそのアドレスを取得できます。次のプログラムは、前のプログラムと同じ countup() 関数に対して、vector<int>配列の要素のアドレスを実引数として指定しています。&v[0] という指定は&(v[0]) と同じ意味です。配列要素を指定するかぎ括弧の演算子 ([]) は、& 演算子よりも優先されるので、このように括弧を使わなくてもアドレス取得を指定できます。

このプログラムでは、不正な入力に対していちいちエラーを表示せずに、間違った回数を数えることにしました。int や double などの基本データ型を仮引数にすれば、その実引数にリテラル、単一の変数、vector 要素を指定できるのと同じように、int*型のポインタを仮引数にすれば、その実引数は vector 配列の要素であろうと単一の変数であろうと区別なく指定できます。

ソースコード 123: vector 要素のアドレス

```
1  // 入力の種類別にカウントするプログラム
2  #include <iostream>
3  #include <vector>
4  void countup(int *target)
5  {
6     if (target != nullptr)
7        ++(*target);
8  }
9
10 int main()
11 {
12    std::vector v {0,0,0};
13    int x{};
14    for (char ch; std::cin >> ch; ) {
15       switch (ch) {
16       case 'a': countup(&v[0]); break;
17       case 'b': countup(&v[1]); break;
18       case 'c': countup(&v[2]); break;
19       default:
20          countup(&x);
21          break;
22       }
23    }
24
25    std::cout <<"a = "<< v[0] <<", "<<"b = "<< v[1] <<", "
26              <<"c = "<< v[2] <<", "<<"x = "<< x <<"\n";
27    return 0;
28 }
```

13.6 構造体のポインタ引数とアロー演算子

構造体の変数のアドレスもポインタを使った関数に渡すことができます。ポインタは間接演算子 (*) を用いてポインタが指す変数を参照する必要がありました。構造体の場合にはさらにドット演算子 ('.') を使って要素を指定しなければなりません。例えば、ポインタ p に対しては (*p).x と指定します。演算子の結合の強さの関係で括弧が必要です。*p.x と書くと、*(p.x) という意味になってしまうからです。構造体のポインタではこのような見づらい指定が頻繁に必要となるために、これらをまとめて指定する方法が用意されています。例えば、このポインタ変数 p の例ならば p->x と指定できます。この矢印 (->) を**アロー演算子**と呼びます。

次のプログラムでは構造体変数のアドレスを引数に指定しています。update() 関数では、p->x や p->y を左辺値として使うことで、指定された構造体のメンバを更新しています。print() 関数では、p->name などの右辺値としての指定で cout の出力を行っています。この仮引数の const の指定は、仮引数 p の指す先が変更できないことを意味します。p それ自体の変更はできます。また、main() 関数では範囲 for 文を使っています。auto& x とリファレンス指定すると、&x は vector 配列 b の要素のアドレスになります。これにより、update() 関数で vector 配列 b の要素が更新されます。単に auto x とすると、x は一時的な変数であり、&x はその一時変数のアドレスです。つまり、update() 関数を呼び出しても配列は更新されないので注意しましょう。

ソースコード 124: 構造体のポインタ引数の例

```
1  // 関数の引数に構造体へのポインタを使用する例
2  #include <iostream>
3  #include <vector>
4  struct Point {
5      std::string name;
6      int x, y;
7  };
8
9  void update(Point *p) // ポインタ引数で受け取る
10 {
11     if (p != nullptr) {
12        p->x += 2; // (*p).x += 2; と同じ
13        p->y += 3; // (*p).y += 3; と同じ
14     }
15 }
16
17 void print(const Point *p) // 更新なしのポインタ引数で受け取る
18 {
19     if (p != nullptr)
20        std::cout << p->name <<","<< p->x <<","<< p->y <<"\n";
21 }
22
23 int main()
24 {
25     Point a {"X", 1, 2};
26     update(&a);
27     print(&a);
28
29     std::vector<Point> b {{"A", 3, 4}, {"B", 5, 6}};
30     update(&b[0]);
31     print(&b[0]);
32
33     for (auto& x : b) { // &はリファレンス指定
34         update(&x); // x は個々の要素
35         print(&x);
36     }
37     return 0;
38 }
```

13.7 ポインタ引数とリファレンス引数の比較

ポインタを仮引数とする関数は、多くの場合、同じ結果となる関数をリファレンス引数で書き直すことができます。次のプログラムはポインタ引数とレファレンス引数の比較となっています。関数 countup1() はリファレンス引数を用いており、関数 countup2() はポインタ引数を用いています。リファレンスを用いた関数の方が簡潔に表現されています。リファレンス引数では、関数呼び出しで指定した実引数が必ず別名の対象となるため、ポインタのようにチェックが不要になり記述が簡潔です。

しかし、リファレンスの方にも問題があります。それは、リファレンス引数を持つ関数の呼び出しだけを見たときに、値渡しであるか、参照渡しであるかの区別がつかない点です。この例では、main() 関数で countup1(z); としていますが、これだけでは z の渡し方がどちらであるか分かりません。ポイン

タを使う場合には、引数としてアドレスを渡しているので、呼び出した先の関数でアドレスに関連する変数が変更される可能性を認識できます。ところが、リファレンス引数の場合には、値渡しと参照渡しの区別がその呼び出しだけでは分からないので、引数となる変数が変更される可能性があるかどうかが分かりません。

ソースコード 125: ポインタ引数とリファレンス引数

```
1  // 引数の種類による違いの確認
2  #include <iostream>
3  void countup1(int& x) // リファレンスを使った仮引数
4  {
5     ++ x;
6  }
7
8  void countup2(int *y) // ポインタを使った仮引数
9  {
10    if (y != nullptr)
11       ++ (*y);
12 }
13
14 int main()
15 {
16    int z {0};
17    countup1(z);
18    countup2(&z);
19    std::cout << z <<"\n";
20    return 0;
21 }
```

C++言語の元となった C 言語には、リファレンスという機能はありませんでした。リファレンスは演算子の多重定義という C++の拡張機能をサポートするために導入されたため[9]、この例のような通常の関数の引数の指定として 2 種類の方法が使えるようになったのです。したがって、どちらの機能を使うかは、プログラムのどの部分の分かりやすさを強調するかで選択します。

13.8 演算子の意味と優先順位

ポインタを操作するアドレス演算子 (&) と間接演算子 (*) の記号は、その記号が使われる場所によって異なる意味となると説明しました。いくつかの例を見て違いを区別してみましょう。

```
// 基本
int  a { 2 };
int& b { a };      // リファレンス
int *c { &a };     // *はポインタ宣言，&はアドレス演算子
int *x = &a;       // C++98形式，*はポインタ宣言，&はアドレス演算子
a = 2* b;          // 乗算
a = 2 * *c;        // 左側が乗算，右側は間接演算子
b = b & 1;         // ビットごとの論理積
*c = 2;            // 間接演算子
a *= 3;            // a = a * 3;
b &= 3;            // b = b & 3;
```

まず、宣言と式の区別が大切です。宣言では、型名が指定されるのでその点で区別がつきます。ただし、C++98 形式の初期値指定では = の記号を使うので、代入と混同しがちです。さらに、= 記号の右側は式なのでさらなる注意が必要です。

式における & 記号の意味は、& の使用される状況が二項演算であるか単項演算であるかを考えると分かります。a & b や a & 0x0F のような二つのオペランドを持つ演算であるならばビットごとの論理積です。結果のデータ型は char や int などの整数を表す型なので、この点でも確認できるでしょう。&a

[9]「C++の設計と進化」106 ページ

のようにオペランド一つならば、変数 a のアドレス取得です。これらの二つの意味の演算子が一つの式に同時に現れることはそれほどありません。

これに対して式における * 記号は複数の意味で同時に使われることがよくあります。これは間接演算子の結果が int や double になる場合です。* 記号に限らず、四則演算などの数値演算の記号と間接演算子を表す* 記号が混じる場合には、演算子の意味を取り違えないようしなければなりません。次の例は間接演算子と数値演算が混じっている例です。

```
// やや複雑
int x{}, z{}
vector ai {0,2,4,8,16};
int *ip { &ai[0] }; // int *ip { &(ai[0]) };
*ip = *ip + 10;     // (*ip) = (*ip) + 10;
*ip += 10;          // (*ip) += 10;
*ip = x  + *ip;     // (*ip) = x + (*ip);
*ip = x * *ip;      // (*ip) = x * (*ip);
x = ++*ip;          // x = ++(*ip);
z = *ip++;          // z = *(ip++);
```

これらの式を理解するには、演算子を分類して考える必要があります。

- 二項演算: a + b のように項が 2 個
- 前置き単項演算： -5 のように演算子が前に置かれ項が 1 個
- 後置き単項演算：x++のような数式にないプログラム特有のもの。配列の a[1] や関数呼び出しの func() も、後置きの単項演算子と考えてよい。

これらの分類では相互に項と演算子の結び付きの強さが違い、後のものほど強い結び付きとなっています。言い替えれば優先度の高い演算子です。そのため、*ip++という式は、ip と++が先に結び付いて *(ip++) と解釈されます。プログラムを読む場合には上記のルールに従って解釈しなければなりませんが、自分で書いていて、ちょっとでも迷う場合には括弧を付けるとよいでしょう。

13.9 アドレス値の出力

アドレス演算子の結果やポインタ変数を cout に指定することで、変数のアドレスを出力できます。変数のアドレスはコンパイラや OS によって変わるので、アドレスの値自体がどんな値であるかが重要となることはそれほどありません。しかし、アドレスの値から変数がメモリ上でどのように配置されているかを知っておくことは大切です。また、複雑なプログラムでは、アドレスの値を調べることで、プログラムのデバッグを行うこともあります。次のプログラムは、cout を使って、int や double の単体の変数や配列の要素のアドレスを出力しています。

ソースコード 126: アドレスの確認

```
1  // ポインタの値を出力
2  #include <iostream>
3  #include <vector>
4  using std::cout;
5  int ga, gb;
6  double gc, gd;
7  std::vector gv {0,1,2,3};
8  int main() {
9     // 大域変数に対してアドレス演算子の結果をcout で出力
10    cout <<"&ga = "<< &ga <<"\n&gb = "<< &gb <<"\n";
11    cout <<"&gc = "<< &gc <<"\n&gd = "<< &gd <<"\n";
12    for (size_t i = 0; i < gv.size(); i++)
13       cout <<"&gv["<< i <<"]= "<< &gv[i] << '\n';
14
15    // 局所変数に対してアドレス演算子の結果をcout で出力
16    int a, b, c;
17    double d, e, f;
```

```
18      cout <<"&a = "<< &a <<"\n&b = "<< &b <<"\n&c = "<< &c <<"\n"
19           <<"&d = "<< &d <<"\n&e = "<< &e <<"\n&f = "<< &f <<"\n";
20
21      // ポインタ値をcout で出力
22      int *ptr { &a };
23      cout <<"ptr = "<< ptr <<"\n";
24      ptr = &b;
25      cout <<"ptr = "<< ptr <<"\n";
26      double *ptr2 { &d };
27      cout <<"ptr2 = "<< ptr2 <<"\n";
28      ptr2 = &e;
29      cout <<"ptr2 = "<< ptr2 <<"\n";
30
31      // 局所変数としたvector 配列の要素
32      std::vector<int> iv {1,2,3,4,5};
33      for (size_t i = 0; i < iv.size(); i++)
34         cout <<"&iv["<< i <<"]= "<< &iv[i] <<"\n";
35
36      std::vector<double> dv {1.0,2.0,3.0,4.0,5.0};
37      for (size_t i = 0; i < dv.size(); i++)
38         cout <<"&dv["<< i <<"]= "<< &dv[i] <<"\n";
39
40    // char 型変数のアドレスは同じ方法では出力できない
41    char ch { 'a' };
42    // cout <<"&ch = "<< &ch <<"\n"; // ×実行時エラーの可能性あり（コンパイルはできる）
43    cout <<"&ch = "<< static_cast<const void*>(&ch) <<"\n"; // 変換すれば出力可能
44    return 0;
45  }
```

右下は出力結果です。結果のアドレスは 16 進数で示されています。また、これらの値は、前述の通りコンパイラと OS によって変わり、さらには実行するごとに変わるかもしれません。したがって、個々の値よりも相対的な関係が重要です。

すぐに分かるのは、アドレスの値が単一の変数の大域変数と局所変数、そして vector の要素の三種に分かれている点です。これには理由があります。大域変数とはプログラムの開始時から終わりまで使われる変数です。つまり、プログラムの実行開始時にメモリ上のどこかに変数の場所が割り当てられていなければなりません。これに対して、局所変数は、関係する関数が呼び出されている間だけ必要になる変数なので、関数呼び出しの開始時にアドレスが決まります。ただ、main() 関数はプログラムの最初から最後まで動作しているので、この説明では奇異に感じるかもしれませんが、main() 関数もそこから呼ばれる他の関数と同じ仕組みとなっているので、この結果になります。vector 変数は大域変数にも局所変数にもありますが、要素のアドレスは大域変数や局所変数とは異なった値となっています。この理由は vector が push_back() などによって要素数を変更できるためです。実行の状況によって新たな要素の場所が必要になるため、要素数の変更に柔軟に対応できるように単一の変数とは異なった特別な場所が使われるのです。

変数の種類	アドレス値
大域変数	0x406000
局所変数	0x7ffeac189000
vector 要素	0x820000

```
&ga = 0x4061e0
&gb = 0x4061e4
&gc = 0x4061e8
&gd = 0x4061f0
&gv[0]= 0x827eb0
&gv[1]= 0x827eb4
&gv[2]= 0x827eb8
&gv[3]= 0x827ebc
&a = 0x7ffeac18991c
&b = 0x7ffeac189918
&c = 0x7ffeac189914
&d = 0x7ffeac189908
&e = 0x7ffeac189900
&f = 0x7ffeac1898f8
ptr  = 0x7ffeac18991c
ptr  = 0x7ffeac189918
ptr2 = 0x7ffeac189908
ptr2 = 0x7ffeac189900
&iv[0]= 0x828ee0
&iv[1]= 0x828ee4
&iv[2]= 0x828ee8
&iv[3]= 0x828eec
&iv[4]= 0x828ef0
&dv[0]= 0x828f00
&dv[1]= 0x828f08
&dv[2]= 0x828f10
&dv[3]= 0x828f18
&dv[4]= 0x828f20
&ch = 0x7ffeac1898bf
```

もう一つ大切な点は vector 要素のアドレスの変化の仕方です。この実行では、vector<int>は隣の要素のアドレスの差が 4 ずつで、vector<double>は 8 ずつです。これはそれぞれ、sizeof(int) と sizeof(double) の値で、このプログラムの実行環境が定めている値です。vector 要素は、隣り合う要素のアドレスが変数のサイズ分だけずれた値になるように、割り当てられることになっているのです。

最後の部分にも着目しましょう。cout による出力では char*型が特別扱いされます。char 型の変数 ch のアドレスは&ch で得られますが、プログラムの最後の方にあるように、これをそのまま cout で出力できません。その理由は、&ch の型は char*型であり、"abc"のような文字列リテラルと対応するためです。cout で文字列を出力することは非常に多くあるため、char*は他のポインタ型と異なった扱いがなされます。そのため、char 型変数のアドレスを出力するには、この例のように、const void*型に型変換します。

14　配列やポインタを用いた間接参照

これまでの配列の例では、主たる処理対象となるデータだけを配列の要素としてきました。これとは異なり、対象データを管理するための情報を配列にすることもできます。例えば、配列の添字を要素とする配列や、配列要素へのポインタを要素とする配列です。元の配列を全体集合と考えるならば、添字やポインタの配列はその部分集合を表すのに使用できます。部分集合は複数作ることができ、それらを通して間接的に元の全体集合の要素にアクセスするのです。この章では、そのような間接的に配列の要素を指定する方法について見ていきます。

14.1　日付の処理

日付の計算をプログラムで処理するときには、月の取扱いを考えなければならないことがあります。日本では、プログラムで睦月, 如月といった月の名前を使うことはそれほどありませんが、英語圏では Jan, Feb といった名前を頻繁に使います。月の数と月の名前を対応させて考えなければならないときには、月の名前を配列として表現し、その添字を月の数と対応させる方法がよく使われます[10]。

次のプログラムは、1) 日数の少ない月, 2) 授業のある月, 3) 奇数の月といった 1 年のすべての月の部分集合となる月を整数配列で表現し、出力には英語の月名を用いたものです。

ソースコード 127: 日付の処理

```
1  // 配列の添字と月の数字を一致させた例
2  // 月という数の情報とその表示用の文字の情報を分ける
3  #include <iostream>
4  #include <vector>
5  using std::cout, std::vector, std::string;
6  // 月の数字を文字列に変換して出力
7  void print(const vector<int>& idx)
8  {
9     vector<string> montab {"", "Jan","Feb","Mar","Apr","May","Jun",
10                            "Jul","Aug","Sep","Oct","Nov","Dec"};
11    for (size_t i = 0; i < idx.size(); i++) {
12       if (i != 0) cout <<", ";
13       cout << montab[ idx[i] ];
14    }
15    cout <<"\n";
16 }
17
18 int main()
19 {
20    vector sm {2, 4, 6, 9, 11}; // 日数の少ない月
21    print(sm);
22    vector cl {4, 5, 6, 7, 9, 10, 11, 12, 1}; // 授業のある月
23    print(cl);
24    vector om {1, 3, 5, 7, 9, 11}; // 奇数月
25    print(om);
26    return 0;
27 }
```

print() 関数の中の montab[idx[i]] という書き方に注意しましょう。これは、idx 配列の要素を通して間接的に montab 配列の要素を指定しています。間接参照とは関係ありませんが、カンマ区切りで出力する方法も確認しておきましょう。実行結果は右のようになります。

```
Feb, Apr, Jun, Sep, Nov
Apr, May, Jun, Jul, Sep, Oct, Nov, Dec, Jan
Jan, Mar, May, Jul, Sep, Nov
```

[10] プログラミング言語 C, 共立出版

14.2　辞書を用いたコード化

集合を考えた場合には要素の重複はないのですが、集合以外で処理対象となる要素が重複している場合には、この要素の重複を利用する方法が考えられます。その一つが辞書を用いたコード化の手法です。

次のプログラムは、いくつかの英文を整数の並びとして表現しています。まず、英文に現れる単語と句読点を string 型の vector 配列 dic に並べてあります。この配列の要素は ABC 順に並べてあり、重複がないことがすぐに分かります。複数の文は、整数配列の idx に納められています。25, 20 や 1 という数字が何度か現れています。このプログラムでも、for 文の中の dic[idx[i]] という方法によって、整数値を使って間接的に英単語を指定しています。

ソースコード 128: 辞書を用いたコード化

```
// 辞書を使った文の表現（単語のコード化）
// いろいろな長さの文字列をすべて整数として処理できる。
#include <iostream>
#include <vector>
using std::cout, std::vector, std::string;
int main()
{
   // ABC 順に保存した単語の辞書
   vector<string> dic {",", ".", "AFTER", "ARE", "CLOUDS", "COME",
      "COMES", "FAIR", "FALLEN", "FALLING", "FALLS", "FROM", "GO", "HAS",
      "IN", "LIKE", "MAINLY", "ON", "OR", "PLAIN", "RAIN", "SHINE", "SPAIN",
      "TEARS", "THAT", "THE", "TO", "WATER", "WEATHER"};

   // 辞書に対する添字の並びで文を保存
   vector idx {25, 20, 14, 22, 10, 16, 17, 25, 19, 1,
               20, 27, 0, 27, 24, 13, 8, 11, 25, 4, 14, 20, 1,
               23, 3, 9, 15, 20, 1,
               2, 20, 6, 7, 28, 1,
               5, 20, 18, 21, 1,
               20, 0, 20, 0, 12, 26, 22, 1};

   // 辞書を引きながら文を表示
   for (size_t i = 0; i < idx.size(); i++) {
      string s { dic[idx[i]] };
      if (s == ",")
          cout << ",";
      else if (s == ".")
          cout << ".\n";
      else
          cout << " " << s;
   }
   return 0;
}
```

このプログラムを実行すると以下のような英文が現れます。[11]

```
THE RAIN IN SPAIN FALLS MAINLY ON THE PLAIN.
RAIN WATER, WATER THAT HAS FALLEN FROM THE CLOUDS IN RAIN.
TEARS ARE FALLING LIKE RAIN.
AFTER RAIN COMES FAIR WEATHER.
COME RAIN OR SHINE.
RAIN, RAIN, GO TO SPAIN.
```

25 は"THE"、20 は"RAIN"、そして 1 は"."でした。このような辞書を使ったコード化は、うまく応用することで、ファイルの圧縮、通信量の削減、暗号化といったさまざまな用途に応用できます。古いスパイ映画のなかで、ある有名な小説を持ち歩き、ページ数、行番号、左端からの単語の位置を無電で知らせることで、秘密のメッセージを伝えあうシーンを見たことがあるかも知れません。その方法はこのプログラムが行っていることとほぼ同じです。

[11] 一行目は 'My Fair Lady' という映画で使われた文です。

14.3　ポインタによる検索用の索引

インターネットの普及によって、コンピュータによる検索は日常の身近な行為として行われるようになって来ました。そのような検索を行ってみると分かりますが、ある検索条件に対応する結果は複数となることがほとんどです。検索結果が複数となる場合には、検索対象の全体集合から部分集合を作り出すことになるので、これまで見てきた添字の配列が役に立ちます。しかし、添字の配列は基本的に元の配列とセットで考えねばならず、扱う変数が必ず複数となります。そこで、ポインタを使い要素ごとの間接参照を考えていきます。

次のプログラムは社員情報のリストから 20 代や 30 代といった世代別の部分集合を取り出しています。社員一人一人の情報を構造体 Member として、名前と年齢を保存します。実際の情報はその Member の vector 配列にいれておきます。検索によって条件にあった部分集合が作られますが、それは Member*型の vector 配列として表し、原本となる情報は複製しないようにします。

ソースコード 129: ポインタによる部分集合

```
1  // 構造体配列とポインタ配列による検索
2  #include <iostream>
3  #include <vector>
4  using std::vector;
5  struct Member {
6     std::string name; // 名前
7     int age; // 年齢
8  };
9
10 auto find_all(vector<Member>& m, int age) // 条件に合う社員リストを作成
11 {
12    vector<Member*> plist;
13    for (auto& x : m) {
14       if (age <= x.age && x.age < age+10)
15          plist.push_back(&x);
16    }
17    return plist;
18 }
19
20 void print(const vector<Member*>& plist) // 一覧を出力
21 {
22    for (auto p : plist)
23       std::cout << p->name <<" "<< p->age <<"\n";
24    std::cout <<"\n";
25 }
26
27 int main()
28 {
29    vector<Member> members {
30          {"Kasai", 41}, {"Honda",20}, {"Tanaka", 48},
31          {"Suzuki",20}, {"Okuda",30}, {"Sakai", 33} };
32    auto s { find_all(members, 20) }; // 20代を探す
33    print(s);
34    s = find_all(members, 30); // 30代を探す
35    print(s);
36    return 0;
37 }
```

find_all() 関数は第 2 引数 age で与えた年代の社員のリストを作ります。戻り値の型の指定は auto ですが、return 文の式の型から戻り値の型は vector<Member*> となります。関数本体では、範囲 for 文で社員の年齢を一人ずつ確認して、アドレスを plist に push_back() しています。範囲 for 文の x は auto&としてリファレンスにしなければ有効なアドレスが得られないことに注意してください。

```
Honda 20
Suzuki 20

Okuda 30
Sakai 33
```

print() 関数では範囲 for 文を使っています。この変数 p は Member*型で、これは組み込み型です。int や double と同じ扱いなので auto&

にする必要はありません。そして、->演算子を使って、p->name や p->age として Member 型のデータにアクセスしています。このプログラムの実行結果の例は右側のようになります。

14.4 vector 要素のアドレスに関する注意点

これまでに vector 配列の要素のアドレスを使用する例をいくつか見てきました。便利なようですが、知っておくべき重要な注意点があります。

要素数変更後のアドレス

まずは「**いつまでもそこにあると思うな！**」という点に注意しましょう。vector は要素数を後から増やしたり減らしたりできます。それはライブラリ関数を支える裏方のプログラムがいろいろと調整して実現されています。そして、変数とメモリとの関係で言えば、push_back() などで要素数が変動した場合には、要素を保存するのにまったく別のメモリ領域を使う可能性があります。結論として、要素の追加などを行ったらポインタの値は無効になると考えるべきなのです。

次の例は、いくつかの整数値を vector 配列に入力して、それをポインタで間接的に扱おうと思って作り始めたプログラムとその実行例です。

ソースコード 130: 入力と間接参照

```
1  // 入力した値と間接的なポインタ
2  #include <iostream>
3  #include <vector>
4  int main()
5  {
6     std::vector<int> v;
7     std::vector<int*> vp;
8     for (int x; std::cin >> x; )
9        v.push_back(x);
10    for (size_t i = 0; i < v.size(); i++)
11       vp.push_back( &v[i] ); // 要素のアドレスを取得
12
13    for (auto p : vp)
14       std::cout << *p <<"\n";
15    return 0;
16 }
```

```
1 2 3 4 5 6 7 8 9 10   <<----- これを入力後に Ctrl-D を入力
1
2
3
4
5
6
7
8
9
10
```

このプログラムは問題ありません。三つの for 文により、v への入力、vp へのポインタの設定、vp のポインタを使った間接的なアクセスをプログラムでは行っています。

これをあるプログラマが「ループを二つにできそうだ」と考えて修正したのは以下のプログラムです。これはコンパイルと実行ができますが、**誤りを含んでいる**ため、思ったとおりに動作しません

```
// 誤ったプログラムなので真似をしてはならない
#include <iostream>
#include <vector>
int main()
{
   std::vector<int> v;
   std::vector<int*> vp;
   for (int i{0}, x{0}; std::cin >> x; i++) {
      v.push_back(x);       // 要素のアドレスが時折変更になる
      vp.push_back(&v[i]);  // 要素数変更中にこれをやってはならない
   }

   for (auto p : vp)
      std::cout << *p <<"\n";
   return 0;
}
```

このプログラムは vector の v に入力を行いながら同時にそのポインタの配列を作ろうとしています。しかし、push_back() は v の使用するメモリ領域を変更するので、ループが進むにつれて、最初の方で保存したアドレスが無効になっていきます。実行例を見てみましょう。

```
1 2 3 4 5 6 7 8 9 10   <<----- これを入力後に Ctrl-D を入力
34218752
0
34144272
0
5
6
7
8
9
10
```

この実行ではどうやら 5 の入力の後に何かが変わったようです。実行結果は毎回異なるので、このプログラムに何も期待できません。vector 要素のアドレスを利用する場合には、途中で要素数を変更をせずに、すべてが確定してからアドレスを使うようにしましょう。

vector<bool> とアドレス

まったく別の話となりますが、std::vector<bool>は特別な方法で実装されているので要素のアドレスは使えません。指定するとコンパイルエラーになります。bool 型は true または false の二値を表す型なので、メモリ上では 1 ビットあれば十分です。そのため、std::vector<bool>だけはメモリの使用量を最小限に抑えるように作られているのです。結果として、要素ごとのメモリアドレスは取得できません。同様の理由で、要素に対するリファレンスも指定できません。つまり、以下はエラーとなります。

```
// vector<bool>でできないこと
#include <iostream>
#include <vector>
int main()
{
   std::vector<bool> x {true, false, true, false};
   std::cout << &x[1] <<"\n";  // error
   bool& a {x[2]}; // error
}
```

vector 配列の要素のアドレスに関連して覚えておきましょう。

14.5 RGB ファイルの検索

多くの Unix OS では rgb.txt というファイルが使われます。このファイルには色の名前と光の三原色 (赤・緑・青) の関係が書かれています。赤・緑・青はそれぞれ 0 から 255 の整数で表されます。各色が 8 ビットの 256 段階で表されることから 8 **ビットカラー**とも呼ばれます。一方で、対応する色名は英単語で、スペースを含んだ 2 単語以上の場合があります。

次のプログラムは、そのファイルを読み込み、色の名前に関する検索を行います。前節のプログラムと同じで基本的に部分集合をポインタ配列で表しています。

ソースコード 131: RGB ファイルの検索

```
1  // vector を使って rgb.txt を読み込み、色名の検索を行う
2  #include <iostream>
3  #include <iomanip> // setw()
4  #include <fstream>
5  #include <vector>
6  using std::string, std::vector, std::setw;
7
8  struct Color {
9     int r, g, b;
10    string name;
11 };
12
13 auto read_file() // rgb.txt ファイルを読み込む
14 {
15    vector<Color> x;
16    std::ifstream fin("/usr/share/X11/rgb.txt"); // OS によって異なる
17    if (!fin) {
18       std::cerr <<"read_file failed!\n";
19       return x;
20    }
21    // 1行に赤 緑 青 色名がホワイトスペースで区切られて並ぶ
22    for (Color t; fin >> t.r >> t.g >> t.b >> std::ws && getline(fin, t.name); )
23       x.push_back(t);
24    return x;
25 }
26
27 auto find_all(vector<Color>& rgb, string x) // 検索
28 {
29    vector<Color*> clist;
30    for (auto& e : rgb)
31       if (e.name.find(x) != string::npos)
32          clist.push_back( &e );
33    return clist;
34 }
35
36 int main()
37 {
38    auto rgb { read_file() }; // ファイルの読み込み
39    if (rgb.empty()) return 1; // 読み込み失敗
40    for (string cn; std::cin >> cn; ) { // 検索用の色の名を入力
41       for (auto p : find_all(rgb, cn)) { // 対象の絞り込み
42          std::cout
43               << setw(4) << p->r <<" "<< setw(4) << p->g <<" "
44               << setw(4) << p->b <<" "<< p->name <<"\n";
45       }
46    }
47    return 0;
48 }
```

vector<Color>が色情報の原本を保存するための Color 配列であり、vector<Color*>型でポインタを使って部分集合を表しています。

関数 read_file では、rgb.txt ファイルを開き、内容を 1 行ずつ読み取っています。このファイルの各行は次のような形式になっています。

```
245 245 245              white smoke
```

最初の三つの整数が赤・緑・青を表すもので、0 から 255 の範囲です。この整数の後にはタブ文字 ('\t') といったホワイトスペースがいくつか続き、そして、色の名前が現れます。色の名前はこの例のように複数単語になる場合もあります。

ファイルの入力では、まず、Color 構造体の一時変数 t に対して、三原色の整数値を読み込みます。

```
fin >> t.r >> t.g >> t.b
```

この後に読み込み位置は 3 個目の整数のすぐ後となり、色名の文字列の前にあるホワイトスペースが次の入力対象です。しかし、単語の前のホワイトスペースは読み飛ばす方が良いです。そこで std::ws の指定でそのホワイトスペースを読み飛ばします。

```
>> std::ws
```

そして、getline() でスペースを含めて行末までの文字列を変数 t.name に読み込みます。

```
&& getline(fin, t.name)
```

一時変数 t に一行分の情報が入力できたので、配列に追加します。

```
x.push_back(t);
```

find_all() 関数は、第 1 引数の rgb 色情報の中で、第 2 引数の x の文字列を名前に含むものを探し、vector<Color*>型の clist に入れて返します。範囲 for 文の e が rgb 配列の個々の要素 (element) に対応しています。

```
for (auto& e : rgb)
```

その要素のアドレスを登録することが目的なので、auto&によるリファレンス指定が必須です。auto としてしまうと、一時変数のアドレスを登録することになり、うまく動作しません。次に string 型の e.name に x が含まれているかを find() メンバ関数で判定します。

```
if (e.name.find(x) != string::npos)
```

find() メンバ関数は x が e.name に含まれている場合には、string 変数内で x が始まる先頭からの位置を返し、見つからない場合には string::npos で示される特別な値を返します。条件を満たしたならば、その要素のアドレス&e を clist に加えます。この関数は最後に clist を戻り値として返します。複数要素の配列をそのまま返すことになるのですが、vector 配列の場合には処理時間が要素数に比例しないように内部で処理されます。[12]

main() 関数では、read_file() でファイルを読み込み、その結果を rgb 変数の初期値に利用しています。次に読み込み失敗がないかを確認し、色名の入力を得てから検索を行います。find_all() の結果がそのまま範囲 for 文に使われている点に注意してください。これは分解すると以下のように書けます。

```
auto clist { find_all(rgb, s) };
for (auto p : clist) {
  ...
```

find_all() は vector を返し、それを変数 clist の初期値としています。そして、範囲 for 文ではそれをそのまま使用していますが、clist はそれ以外では使用されません。そこで二つをまとめます。

```
for (auto p : find_all(rgb, s) ) {
  ...
```

[12] move 代入と呼ばれる機能が働きます

このように書くことで、find_all() の結果がそのまま範囲 for 文に使われます。結果として変数名を一つ考える必要がなくなり、コードが簡潔になるという効果が生まれます。なお、変数 p はポインタ型でこれは組み込み型であるために、コピーをしても無駄がなく問題もないので、リファレンス指定なしの代入の形としています。実行結果の例は次のようになります。

```
medium                    <<------ これを入力
 123  104  238 medium slate blue
   0    0  205 medium blue
  72  209  204 medium turquoise
 102  205  170 medium aquamarine
  60  179  113 medium sea green
   0  250  154 medium spring green
 199   21  133 medium violet red
 186   85  211 medium orchid
 147  112  219 medium purple
snow                      <<------ これを入力
 255  250  250 snow
 255  250  250 snow1
 238  233  233 snow2
 205  201  201 snow3
 139  137  137 snow4
white                     <<------ これを入力
 248  248  255 ghost white
 245  245  245 white smoke
 255  250  240 floral white
 250  235  215 antique white
 255  222  173 navajo white
 255  255  255 white
```

A　言語リファレンス

A.1　基本の型

基本データ型 :

主に使われる基本データ型は bool・char・int・double の４種です。これらのデータ型の値は、1 個または数個のコンピュータの基本命令で処理されます。基本データ型とそれらのポインタや配列を総称して組み込み型 と呼びます。

void 型 :

情報がないということをあえて示すための型です。単独でこれが変数の型として使われることはありません。戻り値のない関数は void func(); などと指定し、void 関数と呼びます。

配列型 :

同一の型の変数を並べたものです。int a[5]; のようにどの型の要素が何個並ぶかを指定して宣言します。このテキストではこれを使わずに、vector を使ってプログラムを表しました。

ポインタ型 :

変数のメモリ位置を表すアドレスを覚えておくために使用します。

リファレンス型 :

ある変数に別の名前を付けるためのもので、通常、仮引数で使用します。

構造体 :

ユーザが定義できる型で、複数の種類のデータをまとめるために使用します。

クラス :

これもユーザが定義できる型で、構造体を拡張したものです。複数の種類のデータと関数をまとめるために使用します。

A.2　リテラル

リテラル (literal) とは「文字どおりの」という意味で、プログラム中で何らかの値を直接表すための表記を指します。基本データ型のリテラルを次に示します。

- bool リテラル: true false
- 文字リテラル: 'a' 'Z' ' ' '$' '%'
- 整数リテラル:

```
123    245  // 10 進数
0b1010      // 2 進数
0173   0365 // 8 進数は 0 で始まる数字の並び
0x7b   0xf5 // 16 進数は 0x で始まる数字と a から f(または A から F) の並び
```

- 浮動小数点数リテラル:

```
3.1415
314.15e-2       // 314.15 × 10^-2
```

エスケープシーケンス

エスケープシーケンスとは表示できない特殊な文字などを特別な文字の並びで表したもので、文字リテラルや文字列リテラルで使われます。よく使われるのは\n, \t, \0,\\, \', \" です。

文字列リテラル

文字列リテラルは、二つの二重引用符 (") で囲まれた文字の並びです。この指定によって、引用符で囲まれた文字の並びと、その後ろに'\0' が付加されたものが作られます。

```
"hello!"      // {'h', 'e', 'l', 'l', 'o', '!', '\0' }
""            // {'\0'}
```

ASCII 文字コードと文字リテラルの関係

ASCII 文字コードと文字リテラルの関係を表で示します。表中の 0x で始まる数値は C++の整数リテラルと同じ 16 進数で、括弧内が対応する 10 進数を表しています。各文字のコード値は、1 行目の値に左端の列の増分を加えた値です。例えば、'A' のコードは 16 進数で 0x41(10 進数の 65) です。C++であまり使用しない部分は空欄にしてあります。これらの部分を使用する場合には直接に整数値を指定します。

	0x00 (0)	0x10 (16)	0x20 (32)	0x30 (48)	0x40 (64)	0x50 (80)	0x60 (96)	0x70 (112)
+0x0(0)	'\0'		' '	'0'	'@'	'P'	'‘'	'p'
+0x1(1)			'!'	'1'	'A'	'Q'	'a'	'q'
+0x2(2)			'\"'	'2'	'B'	'R'	'b'	'r'
+0x3(3)			'#'	'3'	'C'	'S'	'c'	's'
+0x4(4)			'$'	'4'	'D'	'T'	'd'	't'
+0x5(5)			'%'	'5'	'E'	'U'	'e'	'u'
+0x6(6)			'&'	'6'	'F'	'V'	'f'	'v'
+0x7(7)	'\a'		'\''	'7'	'G'	'W'	'g'	'w'
+0x8(8)	'\b'		'('	'8'	'H'	'X'	'h'	'x'
+0x9(9)	'\t'		')'	'9'	'I'	'Y'	'i'	'y'
+0xA(10)	'\n'		'*'	':'	'J'	'Z'	'j'	'z'
+0xB(11)	'\v'		'+'	';'	'K'	'['	'k'	'{'
+0xC(12)	'\f'		','	'<'	'L'	'\\'	'l'	'|'
+0xD(13)	'\r'		'-'	'='	'M'	']'	'm'	'}'
+0xE(14)			'.'	'>'	'N'	'^'	'n'	'~'
+0xF(15)			'/'	'\?'	'O'	'_'	'o'	

A.3 プログラムの構造

C++プログラムの構造を大雑把に説明します。

- プログラムは、変数宣言, 関数定義, 型定義などが並んだものです。
- 関数定義は、関数宣言と関数の本体からなります。
- 関数本体は複文からなります。
- 複文は 0 個以上の文を{と}という記号でまとめたものです。
- 文は宣言文, 式文, 複文, if 文, switch 文, for 文, while 文などです。

複文の中に複文が入るという入れ子の構造になっている点に注意してください。

A.4 式

式は、それ自身が文になったり、if 文や for 文などの条件に使われます。これは、変数やリテラルそのものであるか、+や-などの演算子を使って組み合わされた形となっています。主に値を表現するために使用されますが、変数の値を変更する式もあります。以下では、基本データ型のオペランドに対する演算子を中心に説明します。

この後の説明では、exp, exp1 といった記号を使いますが、これは変数, 値そのものや演算子で組み合わされた部分式を意味することとします。

算術演算子

```
+exp          // 単項プラス
-exp          // 単項マイナス
exp + exp     // 加算
exp - exp     // 減算
exp * exp     // 乗算
exp / exp     // 除算，int 型の場合、結果の小数部分は切捨て。0 で除算した場合の振る舞いは不定。
exp % exp     // 剰余算 (あまり)。オペランド指定に実数は不可。0 で除算した場合の振る舞いは不定。
```

負の数を含む剰余はコンピュータの種類によって結果が異なるので、プログラム中で二つのオペランドはともに正となるようにしてください。

インクリメントとデクリメント

値の取り出しと更新を同時に行う演算子が++(インクリメント) と--(デクリメント) です。これらは、演算子を前に書くのと、後ろに書くのとで、取り出しと更新の順序が変わります。更新は++が+1, --が-1という意味です。通常は、変数に対してこれらの演算子が適用されます。

```
++ exp    // exp に 1(double なら 1.0) を加えて、結果の値を式の値とする
exp ++    // exp の値を取り出し、式の値としてから、1 を加える
-- exp    // exp に 1(double なら 1.0) を減じて、結果の値を式の値とする
exp --    // exp の値を取り出し、式の値としてから、1 を減じる
```

ビット操作演算子

bool, char, int といった整数系データ型は、値を表現するビット単位の演算が用意されています。

```
~exp          // ビットごとの論理否定を計算する (すべてのビットの反転)
exp | exp     // ビットごとに論理和を計算する
exp & exp     // ビットごとに論理積を計算する
exp ^ exp     // ビットごとに排他的論理和を計算する
exp1 << exp2  // exp1 の値をビット単位で exp2 だけ左にシフトした値が結果となる
exp1 >> exp2  // exp1 の値をビット単位で exp2 だけ右にシフトした値が結果となる
```

1 ビットだけ左にシフトすると整数は 2 倍になり、1 ビットだけ右にシフトすると $\frac{1}{2}$ 倍になります。オペランドが整数系データの場合にはこのようなシフト演算ですが、<<演算子は、左側のオペランドが cout だと出力動作であり、>>演算子は、左側のオペランドが cin だと入力動作です。

比較演算子

二つのオペランドを比べる比較演算子の結果は bool 型です。

```
exp <  exp    // 未満
exp <= exp    // 以下
exp >  exp    // より大きい
exp >= exp    // 以上
exp == exp    // 等しい
exp != exp    // 等しくない
```

bool 型への暗黙の変換

bool 型の値は、上記の比較演算の結果の他に、数値を変換することで得られます。後述の if 文や while 文では、bool 型の値が実行条件として必要です。bool 型の値が必要なところで、数値が得られると、0 以外の値を true に、0 を false に変換することになっています。

bool 型の値を組み合わせる式

bool 型の値を組み合わせる論理演算は注意を要するものがあります。

```
! exp          // 論理否定:true は false に、false は true となる
exp1 || exp2   // 論理和:exp1 が true ならば結果は true，exp1 が false ならば結果は exp2
exp1 && exp2   // 論理積:exp1 が false ならば結果は false，exp1 が true ならば結果は exp2
```

論理積と論理和では、短絡評価により exp1 のみで結果が決まる場合に exp2 を評価しません。しがたって、exp2 に次の代入演算子やインクリメントなどの、変数の値を変更する式が含まれていると、勘違いによるエラーを引き起こしやすいプログラムとなるので、注意してください。

代入演算子

左側のオペランドの値を右側のオペランド値で上書きします。

```
x = y;     // x の値を y の値で上書きする。y の値はそのまま。
```

代入演算子には、=記号と他の演算子を組み合わせた形もあります。これは 次のどれかです。

```
+=, -=, *=, /=, %=, <<=, >>=, |=, &=, ^=
```

例えば、+=演算子では、exp1 += exp2 と指定すると、exp1 が一度だけ計算されることを除いて、exp1 = exp1 + exp2 と同等です。

条件式

条件式は条件によって結果の値を選ぶ式です。

```
exp1 ? exp2 : exp3       // exp1 が true ならば exp2，false ならば exp3 を結果とする
```

条件式の結果は、そのまま代入する場合が多いようです。また、exp1 の部分は比較演算の式が良く用いられます。条件式は exp2 または exp3 のどちらかのみが計算に使われるので、&&や||演算子と同様に、関数を指定する場合には注意が必要です。
条件式を使った代表的な例を以下に示します。

```
max = (a>=b) ? a : b;      // a と b を比較して大きい方を max に代入する
```

sizeof 演算子

変数や型のサイズは sizeof 演算子で知ることができます。サイズとは、対象となる変数がメモリ上で何バイトであるかということです。変数や型のサイズは、コンピュータの種類やコンパイラの種類によって異なるものなので、このような演算子が用意されています。現代のパーソナルコンピュータの多くでは、以下のような結果となるでしょう。

```
sizeof(bool) == 1
sizeof(char) == 1
sizeof(int) == 4
sizeof(double) == 8
```

演算子の優先順位

算術演算の a+b*c が、a+(b*c) を意味するのと同様に、これまで説明した演算子には優先順位が決められています。以下にその優先順位の高い演算子から順に示します。

演算子	結合規則
x++, x--, 関数呼び出し, 配列の添字指定	左
sizeof, ++x, --x, !, -x(単項マイナス), +x(単項プラス)	右
*, /, %	左
+, -	左
<< , >>	左
< , <=, >, >=	左
== , !=	左
&	左
^	左
|	左
&&	左
||	左
?:	右
=, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=	右

同じ順位の演算子が並んだ場合には、結合規則を用います。結合規則が左の場合には左から計算し、(**左結合**、右の場合には右から計算します（**右結合**。例えば、x+y-z は、(x+y)-z であり、x=y=z は、x=(y=z) と解釈されます。[13]。

優先順位と結合規則が決められていますが、一見して混乱を招きそうな式には、これらの規則を当てにせず、() をつけて明確に演算の順序を指定するのが良いプログラミング方法です。

定数式

コンパイル時に値の定まる式を**定数式**と呼びます。

A.5　文

宣言文

変数を使うには使用することを宣言文で宣言します。宣言文は変数が必要になったところで書くことができ、他の文からの前後関係の制約を受けません。宣言文の例を次に示します。

```
int x;                  // int 型の変数 x, 初期値は宣言文を指定した場所による
int y { 0 };            // int 型の変数 y, 初期値は 0
int z { y };            // int 型の変数 y, 初期値は y の保持する値
bool flag { true };     // bool 型の変数 flag, 初期値は true
char ch { 'a' };        // char 型の変数 ch, 初期値は 'a'
```

この例のように、宣言文には初期値を指定する方法としない方法があります。組み込み型で、初期値を指定しない変数宣言では、宣言文の指定場所によって、変数の初期値の決め方が変わります。そのため、可能な限り宣言時に初期値を指定することが望まれます。

[13]厳密に言うと、優先順位と結合規則で説明できない場合があります

const 指定

初期値を設定したら以降の変更ができない変数を宣言できます。これによって、うっかりミスによる値の変更をコンパイラに指摘させることができます。また、配列の宣言ではコンパイル時に値の定まる定数式が必要ですが、const 指定によって定数式に名前をつけることで、明確な意味づけが行えます。

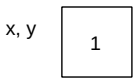
```
const int x { 794 };  // x は変更不可，定数式と同等
int y;                // y を宣言して、
...                   // y の値を変更する
const int z { y };    // y と同じ値で初期化，z は変更不可だが定数式と同等ではない
```

リファレンス指定

ある変数の別名となる変数を宣言できます。

```
int  x { 1 };
int& y { x };    // y は x の別名，x の値の変更は y の値の変更を意味する
```

これは次の図のように、変数という入れ物に二つの名前タグをつけたようなものです。x という指定と y という指定は同じオブジェクトを意味します。



式文と空文

式の後ろにセミコロン (;) をつけると**式文**という文になります。式の最終的な結果は破棄されます。3 ページの例の、cout << "hello!"; は式文です。この式の結果は、cout オブジェクト自身ですが、その結果は使わずに破棄しています。

さらに単純な文には**空文**があります。空文はセミコロン (;) だけの文で、何もしないことを意味します。これは、while 文と組み合わせて使われることの多い文です。

```
while (cin >> x && x == 0)  // 0 以外の入力が得られるまでループ
    ;                       // 空文
```

複文

複数の文を{と}という記号でまとめたものを**複文**と呼びます。複文は、関数の本体として使われる他に、if 文や while 文でも使われます。また、複文は**ブロック**とも呼ばれ、複文の中に複文を含むという入れ子の構造を利用して、変数の有効範囲が決められます。

```
int main()
{                       // 関数本体は複文
    int x { 0 };
    {                   // 入れ子になった複文
        int x { 1 }; // 外側の x を隠す。内側の複文でのみ有効
        int y { 1 };
    }                   // 内側の複文はここまで
     return 0;
}
```

if 文

if 文は条件によって指定した文を実行します。条件には、bool 型の結果となる式を指定します。上述のように、条件に数値を指定した場合にも bool 型に変換されます。以下の 2 種類があります。

```
if ( 条件 ) 文 1
または
if ( 条件 ) 文 1 else 文 2
```

前者は、条件が true ならば文 1 を実行します。後者は、条件が true ならば文 1 を実行し、false ならば文 2 を実行します。

```
// x の絶対値を計算
if (x < 0)
   x = -x;
```

switch 文

switch 文は整数系データ型となる式の結果に応じて、ブロック内部の文の開始場所を決めるものです。通常は、break 文と組み合わせて、ブロック内部の一部の文のみを実行します。

```
switch（条件となる変数）{
case ラベル 1: 文 1; break;
case ラベル 2: 文 2; break;
default:       文 3; break;
}
```

default:には、他の case ラベルで指定した値に一致しない場合の処理を指定します。default:ラベルは省略できますが、「指定以外の処理」はエラー処理が必要となることが多いので省略するかどうかは慎重に考えてください。

switch 文の中で変数を宣言するには、必ず複文を書いてその内側で宣言文を書かなければなりません。

while 文

while 文は繰り返して文を実行するためのものです。指定方法は次の通りです。

```
while（条件）文
```

条件は if 文と同様に bool 型の結果となる式です。条件が true の間だけ文を実行します。

```
while (cin >> x)      // 入力が成功している限り
   cout << x;         // それを出力する
```

for 文

for 文は繰り返して文を実行するためのもので、特に、繰り返し実行の対象となるデータの範囲が事前に指定できるような時に使われます。指定方法は次の通りです。

```
for（初期設定文 条件; 式）文     // これが正式な形式
```

初期設定文は宣言文・式文・空文のどれかで、必ずセミコロン (;) で終わる文です。条件は bool 型の結果となる式です。式は任意の式を指定できますが、通常、代入かインクリメントを使って、繰り返しに関係する更新処理を指定します。

for 文も条件が true である限り、文を実行するという点で while 文と同じですが、これに繰り返しを開始する前に 1 回だけ初期設定文を実行するという点と、次の繰り返しのための条件判定を行う前に、更新のための式評価を行うという 2 点が加わります。つまり、次のようになります。

1. 初期設定文を行う
2. 条件を判定する
3. 条件が true なので文を実行する
4. 式を評価する
5. 条件を判定する
6. 条件が true なので文を実行する
7. 式を評価する
8. ...(上記 3 項目を繰り返す)
9. 条件を判定する
10. 条件が false なので for 文を終了する

次の while 文と for 文例は、同じ結果となる内容ですが、for 文で記述した方が良い場合を示しています。

```
// 1 から 10 まで順に出力する
int i { 1 };
while (i <= 10)
   cout << i++ << "\n";

// 1 から 10 まで順に出力する
for (int j = 1; j <= 10; j++)
   cout << j << "\n";
```

大きく異なるのは変数のスコープです。while 文の前で宣言した変数 i は while 文の後も有功ですが、j は for 文の末尾までがスコープなので、それ以降は使用できません。多くの経験から変数のスコープは不必要に広くしないことが良いとされています。

break 文と continue 文

for 文や while 文などの繰り返し文では、実行している複文を途中で止めるためのジャンプ文を利用できます。

```
break;      // 複文の残りを実行せずにその複文を指定する文を終了する
continue;   // 複文の残りを実行せずに次の繰り返しの処理を続ける
```

前述のように、break 文は for 文や while 文だけでなく、switch 文で指定した複文を終了するためにも使います。for 文の中で continue 文を指定すると、直ちに、更新のための式を行い、条件判定を処理が進みます。

break 文と continue 文は、通常、複文の中で指定されますが、特殊な使用例として、continue 文を単独で指定することで、めだつ空文として使われることがあります。

```
while (cin >> x && x == 0)  // 0 以外の入力が得られるまでループ
    continue;               // 空文と同じ効果
```

A.6　関数定義

関数を定義するには、以下の形式にします。

```
戻り値の型　関数名 ( 仮引数の宣言のリスト ) 複文
```

引数がない場合には仮引数の宣言のリストを指定しませんが、戻り値の型は void を含めて必ず指定しなければなりません。

簡単な関数定義の例を示します。

```
int  max(int x, int y) { return (x < y) ? y : x; }
bool less(int x, int y) { return x < y; }
```

A.7　関数宣言

関数の本体 (複文) を決める前に、関数の外部仕様 (呼び出し方と結果の受け取り方) を決めることができます。これは関数宣言で行います。関数宣言と関数定義では戻り値と仮引数の型が一致しなければなりません。

```
戻り値の型　関数名 ( 仮引数の型のリスト );
int  max(int, int);
bool less(int, int);

// 引数名を書いても良い (無視される)
戻り値の型　関数名 ( 仮引数の宣言のリスト );
int  max(int y, int z);
bool less(int z, int n);
```

A.8　標準ライブラリ

名前空間 std

標準ライブラリは、std という名前空間で宣言されています。これらは、「std::」を指定して使える事前に用意されたクラスや関数群です。例えば、std::cin, std::cout, std::string などです。

これらの名前はプログラム中で頻繁に使われるために、std:: の部分ばかりが目立って、全体が見づらくなります。これを省略には二つの方法があります。

- using ディレクティブ

  例： `using namespace std;`

- using 宣言

  例： `using std::cout, std::string;`

C++98 までは using ディレクティブがよく使われてきました。これは std の名前空間に登録されているものをすべて取り込んで、std::を省略するものです。ところが、C++11 以降では std 名前空間の変数名や関数などが非常に多くなったために、上記のような一括して省略するは良くないと判断されるようになりました。そのために、省略するものだけを指定する using 宣言が推奨されています。

cin と cout

cin >> x という式は文字列を標準入力から読み込み、x の型にあった値に変換します。EOF に到達して読み込むべき文字列がない場合や読み込んだ文字列が値に変換できない場合には、入力が失敗します。その場合には、cin の内部状態が true から false に変わります。式の結果は cin そのものです。cin >> x >> y は (cin >> x) >> y と解釈され、この式の結果もやはり cin そのものです。cin の式が bool 型を必要とするところに指定されると、内部に記録されている状態が式の値となります。

cout << x という式は、逆に、x の値を文字列に変換して、標準出力に書き出します。この式の結果も同様に cout 自身であり、出力の成功と失敗が内部状態に保存されます。

cin >> x の結果の利用としては、次の 2 種類のパターンが良く使われます。

```
if (cin >>x)...     // 単独に入力が成功したかどうかをチェックする場合
while (cin >> x)...// 入力する値の個数が分からない場合
```

マニピュレータ

cin からの入力や cout への出力の振舞を変えるのがマニピュレータです。

cout では、出力の後に改行が必要となることがあります。指定による違いを以下に示します。

```
cout << x;             // 改行なし
cout << x << '\n';     // 改行あり
cout << x << endl;     // 改行＋フラッシュ(強制出力)
cout << x << flush;    // 改行なし＋フラッシュ(強制出力)
```

マニピュレータ endl の指定は "\n"指定と同様に改行の出力を意味しますが、それに加えて、以前に書き込まれた内容を強制的に出力します。出力処理は時間のかかる処理なので、endl が指定されない場合には、出力指定の文字列を一時的に貯めておき、出力のタイミングを遅らせます。延期された内容は、バッファと呼ばれる貯める領域が満杯となるか、プログラムの終了時に出力されます。このため、ループの中で endl が指定されると、貯める動作がなくなり、プログラム実行が遅くなります。問題となるのはプログラムが異常終了する場合で、貯められた内容が出力されずに終わり、プログラムがどこまで進んだのかが分からなくなることです。そのため、何も考えずに endl を指定するのはよくありませんが、プログラムの実行途中で確実に出力を得たい場合には、endl を指定します。

この他に良く使う形式を以下に示します。setw() だけがすぐ後の出力のみに影響を与えて、その他は再度設定変更されるまで有効です。また、setprecision() や setw() といった引数をとるマニピューレータを使用するには、<iomanip>ヘッダファイルをインクルードします。

```
cin >> noskipws >> c;                       // ホワイトスペースを読み飛ばさず入力
cin >> skipws >> i;                         // ホワイトスペースを読み飛ばしてから入力
cin >> ws;                                  // ホワイトスペースのみを読み飛ばす
cin >> oct >> i;                            // 入力文字列を 8 進数と仮定する
cin >> dec >> i;                            // 入力文字列を 10 進数と仮定する
cin >> hex >> i;                            // 入力文字列を 16 進数と仮定する

cout << boolalpha << true;                  // bool 式を true/false の文字で出力
cout << oct << 123;                         // 8 進数の形式で出力  173
cout << dec << 123;                         // 10 進数の形式で出力 123
cout << hex << 123;                         // 16 進数の形式で出力 7b
cout << setprecision(3) << sqrt(2.0);       // 精度 3 桁 1.41
cout << showpoint << 10.0;                  // 小数点以下が 0 でも小数部を出力
cout << noshowpoint << 10.0;                // 小数点以下が 0 なら整数部のみ出力
cout << scientific << 10.00;                // 科学的記法で出力 1.0000e+01
cout << fixed << 10.00;                     // 10 進記法で出力 10.0000
cout << fixed << setprecision(2)<< 10.0/3;// 小数点以下第 2 位までを出力
```

```
cout << setw(4) << 123;                    // 最低でも 4 桁分の幅で出力 (1 回限り)
cout << setw(7) << right << 123;           // 幅を決めて右寄せ出力
cout << setw(7) << left  << 123;           // 幅を決めて左寄せ出力
```

string クラス

string クラスは標準ライブラリで提供されるクラスですが、文字列を扱う組み込み型のごとく使用できます。

```
string s;           // 空文字列で初期化
string s1 {"abc"}; // 文字列リテラルで初期化
s = s1;             // 代入 (s を s1 の中身で置き換える)
s + "abc"           // s と文字列リテラルを連結して、新たな string を返す
s + s1              // s と s1 を連結して、新たな string を返す
s == "xyz"          // s のリテラルが同じ文字列ならば true, それ以外は false を返す
s == s1             // s と s1 が同一文字列ならば true, それ以外は false を返す
s != s1             // s と s1 が異なれば true, それ以外は false を返す
<, <=, >, >=        // ASCII コードによる順序を判定 true または false を返す
```

入出力は次のように指定します。

```
string s;
cin >> s;           // 先頭のホワイトスペースを読み飛ばして文字列を入力
cout << s;          // 出力する
getline(cin, s);    // スペースを含めて行末まで読み込む
```

string クラスでは専用のメンバ関数が多数利用できます。そのいくつかの例を以下にあげます。

```
string s;
if (s.empty())...        // 空文字列なら true, それ以外は false を返す
s.size()                 // s が保持する文字数を返す
int n  {s.find('k') };  // s が'k' を含んでいれば、s の先頭からの位置を返す
                         // 含んでいなければ、string::npos を返す
int n2 {s.find("abc")}; // s が"abc"を含んでいれば、s の先頭からの位置を返す
                         // 含んでいなければ、string::npos を返す
```

次の使用方法には注意が必要です。

```
c = s[n]    // 先頭を 0 とした n 番目の文字を返す. n が負または s.size() 以上だと実行時エラー
s[n] = '*'; // n 番目の文字を書き換える. n が負または s.size() 以上だと実行時エラー
s += "abc"; // s の後ろに"abc"を加える. s.size() の値が変化する
```

Copyright©2006–2019,2020 Shusuke Okamoto, All rights reserved.

この本に対する質問, 疑問, 誤り指摘は電子メールで okam@st.seikei.ac.jp までお願いします。

索 引