

放送大学 オンライン授業

C 言語基礎演習

教科書

目次

第 1 章	プログラミングと C 言語	4
1.1	様々なプログラミング言語	4
1.2	C 言語の特徴	6
1.3	C 言語の学習	6
1.4	この授業の位置づけ	8
第 2 章	C 言語の環境構築	9
2.1	プログラムの作成と実行	9
2.2	テキストエディタとコンパイラの準備	9
第 3 章	はじめの一步	14
3.1	最初のプログラム	14
3.2	当面の定型部分	17
3.3	コーディングスタイル	17
第 4 章	変数	21
4.1	変数と型	21
4.2	変数の宣言と値の代入	22
第 5 章	入出力	28
5.1	関数の役割と書式	28
5.2	画面への出力	29
5.3	キーボードからの入力	36
第 6 章	算術演算	40
第 7 章	演算子と式	44
7.1	演算子	44
7.2	式と評価	44
7.3	演算子の優先順位	45
7.4	型とキャスト	49

7.5	式と文	52
第 8 章	分岐	53
8.1	順次と分岐	53
8.2	真理値	54
8.3	if 文	54
8.4	複合文	60
第 9 章	数の比較	65
9.1	数の比較	66
9.2	数の比較のための演算子	68
第 10 章	論理演算	71
10.1	論理演算	71
10.2	論理演算のための演算子	74
第 11 章	分岐の応用	81
11.1	最大値・最小値	81
11.2	3 つ以上への分岐	88
11.3	論理演算の応用	91
第 12 章	様々な演算子	97
12.1	単項 +、- 演算子	97
12.2	複合代入演算子	98
12.3	インクリメント・デクリメント演算子	100
12.4	条件演算子	102
12.5	コンマ演算子	104
第 13 章	反復	105
13.1	順次・分岐・反復	105
13.2	while 文	106
13.3	反復の制御	111
13.4	for 文	114
13.5	反復のネスト	117
第 14 章	関数の使用	120
14.1	引数と戻り値	120
14.2	ライブラリ関数	120
第 15 章	関数の作成	129

15.1	関数定義	129
15.2	関数呼び出し	132
15.3	関数プロトタイプ宣言	134
15.4	main() 関数、printf() 関数、scanf() 関数	139
第 16 章	引数の受け渡し	141
16.1	値渡し	141
16.2	変数のスコープ	144
第 17 章	コンパイル	148
17.1	コンパイルの流れ	148
17.2	プリプロセス	149
17.3	狭義のコンパイル以降	152
17.4	4 工程の確認	153
付録 A	TeraPad のインストールと使い方	155
A.1	TeraPad のインストール	155
A.2	TeraPad の使い方	157
A.3	TeraPad の便利な設定	160
付録 B	MinGW のインストールと使い方	161
B.1	MinGW のインストール	161
B.2	MinGW を使うための設定	164
B.3	コマンドプロンプトの操作とコンパイル	167
付録 C	コンパイルのエラーメッセージ例	171

第 1 章

プログラミングと C 言語

1.1 様々なプログラミング言語

これからプログラミング言語の 1 つである C 言語を学びます。“1 つ”というからには、プログラミング言語は C 言語だけではありません。どのようなプログラミング言語が思い浮かびますか。

Java や Python など聞いたことがあるかもしれません。Java は、汎用的なプログラミング言語で、様々な種類のソフトウェアを作ることができます。**オブジェクト指向**という概念を学ぶためのプログラミング言語としてもよく使われます。Python は、比較的歴史の長いプログラミング言語ですが、最近人気が高まっています。こちらも汎用的な目的に使えます。データ解析や機械学習などを行いやすいところが人気の理由のようです。

ここでは 2 種類紹介しましたが、ほかにも多くのプログラミング言語があります。では、いくつあるでしょうか。数種類や十数種類ではありません。Wikipedia の“プログラミング言語一覧”^{*1}を見てみましょう。C 言語^{*2}、Java、Python は、もちろん入っています。その他、100 種類以上のプログラミング言語が並べられています。

たくさんの種類がある理由として、プログラミング言語ごとに得意な分野や特徴があることがあげられます。少し長くなりますが、ソフトウェアの種類を考えてみましょう。

PC で皆さんがよく使うソフトウェアは、Web ブラウザや文書作成ソフトウェアなどでしょうか。表計算ソフトウェアやプレゼンテーションソフトウェアもよく使われます。これらのソフトウェアは、大きくくりでは**アプリケーションソフトウェア**、あるいは**デスクトップアプリケーション**などと呼ばれます。アプリケーションソフトウェアと対になる用語は、**基本ソフトウェア**や**システムソフトウェア**です。基本ソフトウェアは、コンピュータのハードウェアを制御したり、アプリケーションソフトウェアが動作するために必要な機能を提供したりします。代表例は、Windows や macOS です。PC で動いているソフトウェアは基本ソフトウェアとアプリケーション

^{*1} <https://ja.wikipedia.org/wiki/%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%9F%E3%83%B3%E3%82%B0%E8%A8%80%E8%AA%9E%E4%B8%80%E8%A6%A7>

入力するのは大変なので、検索してください。

^{*2} 2019 年 1 月現在、単に C と書かれています。原語 (英語) でも、多くの場合 C です。

ンソフトウェアに大別できますが、それだけではありません。マザーボード^{*3}やハードディスクなどの機器の上では、基本ソフトウェアとは独立したソフトウェアが動作しており、これらは**ファームウェア**と呼ばれます。

デスクトップアプリケーションは、**Web システム**、または **Web アプリケーション**の対比として使われることがあります。Web システムとは、Web ブラウザを通して使うサービスを指します。例として、インターネットショッピングやインターネットバンキングがあげられます。これらのサービスを提供するソフトウェアは、遠隔地のサーバで動作しています。このように PC で動作する、または PC から利用できるソフトウェアだけでも多数の種類があります。

スマートフォンやタブレットといった携帯端末も PC と同様の仕組みです。Android や iOS といった基本ソフトウェアが動作しており、その上で**アプリ**などと呼ばれるアプリケーションソフトウェアが動作しています。Web ブラウザを通して Web システムを使うこともできます。携帯端末では、Web システムの対比として、デスクトップアプリケーションではなく**ネイティブアプリケーション**という用語などが使われます。

コンピュータの中ではソフトウェアが動作していますが、PC や携帯端末以外にもコンピュータはたくさんあります。体重計を例に考えてみましょう。アナログの体重計では、目盛りや、目盛りを指す針がくるくる回ります。このような体重計は、ばねの力を使って機械的に体重を量っています。最近主流のデジタルの体重計は、圧力を検知するセンサーで体重を量ります。ついでに体に微弱な電流を流して、体脂肪率や骨格筋率を量ってくれるものもあります^{*4}。このような体重計の内部では、ソフトウェアが動作しています。汎用的なコンピュータ^{*5}ではなく、特定の目的を持った機器の上で動作するソフトウェアを**組み込みソフトウェア**といいます。

このように、様々な種類のソフトウェアがあります。従って、それらを作るプログラミング言語への要求も、おのずと異なるものになります。たとえば、組み込みソフトウェアであれば、リソース^{*6}に制限のあるハードウェア上で高速に動作するソフトウェアを作るプログラミング言語が必要かもしれません。デスクトップアプリケーションであれば、GUI^{*7}を簡単に構築できるプログラミング言語が便利でしょう。また、同じ種類のソフトウェアであっても、短期間で作りたい、高速に動作するものを作りたい、多くの環境で動作するものを作りたい、などプログラミング言語に求められる要求は様々です。すべての要求に応えられるプログラミング言語はありませんので、得意な分野や特徴の異なる多くのプログラミング言語が作られています。

^{*3} コンピュータの様々な部品が接続される基板。

^{*4} 参考: 体組成計の原理 | 健康のつくりかた | タニタ (<https://www.tanita.co.jp/health/detail/37>)

^{*5} PC など。

^{*6} CPU の動作速度やメモリの容量。

^{*7} グラフィカルユーザインタフェース。ユーザが操作する画面。

1.2 C 言語の特徴

C 言語は、前節で紹介したほとんどの種類のソフトウェアの開発に使うことができます。しかし、適しているかどうかは別問題です。たとえば、Web システムを C 言語で開発することはできますが、ほとんどの場合非効率です。Web システムの開発には、Java や Python の方がはるかに適しています。では、C 言語が得意な分野は何でしょうか。多少乱暴ですが、上で紹介した中では、基本ソフトウェア、ファームウェア、組み込みソフトウェアなどが C 言語の得意な分野といえます。つまり、システム全体や、ユーザが使うアプリケーションソフトウェアを支える、裏方で動いているプログラムです。これらのプログラムは、高速な動作、少ないリソース使用量、効率的なメモリ操作などが求められ、C 言語はいずれも満たしています。つまり、他のプログラミング言語で作ったプログラムよりも高速に動作する傾向があり、プログラムのファイルサイズや実行時のメモリ使用量も少ない傾向があります。また、効率的なメモリ操作が行えます。

C 言語の欠点として、バグを含む記述になりやすい、という点があげられます。それがセキュリティ上の問題になることもあります。また、他のプログラミング言語より、記述量が多くなる傾向があります。つまり、生産性は高くありません。C 言語は、プログラマーのスキルによる差が出やすいプログラミング言語といえます。

1.3 C 言語の学習

どのプログラミング言語を学べばよいのかというのは、よく議論されるテーマです。“プログラミング言語 初心者”などで検索すると、初心者が学習するのに適したプログラミング言語についての記事が見つかります。C 言語を推奨する記事もあれば、初心者には適していないとする記事もあります。

C 言語は、使い捨てのプログラムを短時間で書けるようなプログラミング言語でもなければ、きちんと理解せずにサンプルのコードを切り貼りして正しく動いてしまうようなプログラミング言語でもありません。しっかり学習し、注意深く記述する必要があります。コンピュータやプログラミングそのものに対する理解を深められるという点で、学習に適しているといえます。

初心者に適しているかどうかと同様に、C 言語が難しいか簡単かも意見が分かれます。難易度を議論するためには、“原理の理解が難しい”、“覚えることが多すぎる”、“複雑な記述をしなければならない”など難しさの要因を考える必要があります。C 言語では、覚えることは多くありません。ただ、学習の初期の段階に難しい点が多くあります。

まず、環境構築が簡単ではありません。この後で解説しますが、C 言語のプログラムを作って実行できる環境を整えるためには、面倒な作業や、多少のコンピュータの知識が必要です。

次に、簡単なプログラムを理解できるようになるまでに、多くの項目を学習しなければなりません。ソースコード 1.1 は、キーボードから整数を 1 つ読み取って、それをそのまま画面に表示

示するだけの簡単なプログラムです。実行例^{*8}も示します。第 5 章まで読み進めれば書けるようになりますが、実はこのプログラムは、この授業の範囲では完全には理解できません。1 行目の `#include <stdio.h>` の意味がわかるのは、第 17 章です。3 行目の `int`、`main`、`void` と、10 行目の `return 0;` の意味は第 15 章まで進めばわかりますが、完全に理解できるわけではありません。7 行目に出てくる `&` という記号の意味は、この授業の範囲ではわかりません。このように、学習を終える頃にやっと全体が理解できるようなところがあります。最初のうちは、“このようなものだ” と思って書くしかありません。

ソースコード 1.1 簡単なプログラム

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int n;
6
7      scanf("%d", &n);
8      printf("%d\n", n);
9
10     return 0;
11 }
```

実行例

15 ↵

15

また、前節で C 言語の欠点として紹介したとおり、同じ処理を書くのに、他のプログラミング言語と比べてたくさんの量を記述する必要があります。

そして、**ポインタ**の概念や使い方が難しく、初心者がつまづく最大の難所とされています。ポインタはこの授業の範囲外ですが、ポインタを扱えることは C 言語の大きな特徴です。

C 言語は最初につまづきやすいプログラミング言語ですが、それを乗り越えれば比較的簡単に身につけることができます。

C 言語は“学習用”や“初心者向き”のプログラミング言語ではなく、実用的で広く使われています。プログラミング言語の人気ランキングでも常に上位に位置しています。有名なランキングとし

^{*8} 実行例で、ユーザがキーボードから入力する文字は、**青字**で記載します。↵ は、エンターキーを押すという意味です。この実行例では、ユーザが 15 と入力してエンターキーを押すと、プログラムが 15 と表示したことを示しています。

て、TIOBE Index^{*9}や、IEEE Spectrum が発表するものがあります。TIOBE Index では、2019 年 9 月現在、1 位が Java、2 位が C 言語、3 位が Python、4 位が C++ です。IEEE Spectrum の 2019 年のランキング^{*10}では、1 位が Python、2 位が Java、3 位が C 言語、4 位が C++ です。C++ は C 言語を拡張した^{*11}プログラミング言語です。また、Python の標準的な処理系^{*12}は C 言語で書かれています。このように、他のプログラミング言語に影響を与えたり、他のプログラミング言語の実装に使われたりしています。これらのことから、C 言語は学習する価値のあるプログラミング言語といえます。

1.4 この授業の位置づけ

この授業は、C 言語の基礎を学ぶことと、それを通してコンピュータやプログラミングに対する理解を深めることを目的としています。取り扱う範囲は、C 言語の入門の前半部分に相当します。この授業を終えることで、今後 C 言語をさらに学んだり、別のプログラミング言語を学んだりする上での基礎を身につけることができます。配列やポインタといった重要な概念や、実用的なプログラムを書くために必要な文字列処理やファイル処理についてはこの授業の範囲外です。“C 言語の入門を終えた”というためには、さらなる学習が必要です。この授業を終えた後で、是非学習を進めてください。

^{*9} <https://www.tiobe.com/tiobe-index/>

^{*10} The Top Programming Languages 2019.

<https://spectrum.ieee.org/the-top-programming-languages-2019>

^{*11} 大幅に拡張しています。

^{*12} ある言語で書かれたプログラムを解釈し、実行できるようにするためのソフトウェア。

第 2 章

C 言語の環境構築

2.1 プログラムの作成と実行

C 言語でプログラムを作るためには、C 言語の規則に従って命令を記述しなければなりません。命令はテキストで書き、コンピュータ上に保存します。コンピュータが実行できる^{*1}のは、**機械語**で書かれたプログラムだけです。C 言語で書いて保存したファイルはただのテキストファイルです。機械語で書かれたプログラムではありませんので、そのままでは実行できません。言語やその処理系によって異なりますが、C 言語では、命令を記述したファイルを機械語のプログラムに変換し、それを実行する方式が一般的です。記述した命令を**ソースコード**や**ソースプログラム**、それを保存したファイルを**ソースファイル**といいます。また、ソースコード (ソースファイル) を機械語のプログラムに変換することを**コンパイル**といい、コンパイルするためのソフトウェアを**コンパイラ**といいます。変換後のプログラムは実行可能な形式なので、**実行ファイル**とも呼ばれます。

まとめると、C 言語でプログラムを作成し、実行するためには、まずソースコードを記述し、ソースファイルとして保存します。次にコンパイラでソースコード (ソースファイル) をコンパイルし、実行ファイルを作ります。最後に、実行ファイルを実行します。

ソースファイルの実体はただのテキストファイルなので、テキストエディタで記述することができます。つまり、C 言語のプログラムを作成し、実行するためには、**テキストエディタ**と**コンパイラ**が必要です。

2.2 テキストエディタとコンパイラの準備

演習のため、テキストエディタとコンパイラを準備しましょう。

テキストエディタには多数の選択肢があります。コンパイラは、テキストエディタほどは多くありませんが、複数の選択肢から選べます。有償の製品を購入してもよいですし、演習のためであれば、無償のものでも十分です。

テキストエディタは、Windows であればメモ帳でも大丈夫ですが、別途準備することをおすす

^{*1} 正確には、「CPU が実行できる」。

めします。図 2.1 は、C 言語のソースコードを、メモ帳と TeraPad というテキストエディタのそれぞれで開いたときの画面です。TeraPad では、C 言語の文法に応じて、ソースコードの各部分が太字になったり、異なる色になったりしています。これは、**シンタックスハイライト**と呼ばれる機能です。シンタックスハイライト機能を持ったテキストエディタであれば、ソースコードが読みやすいですし、誤りにも気づきやすくなります。

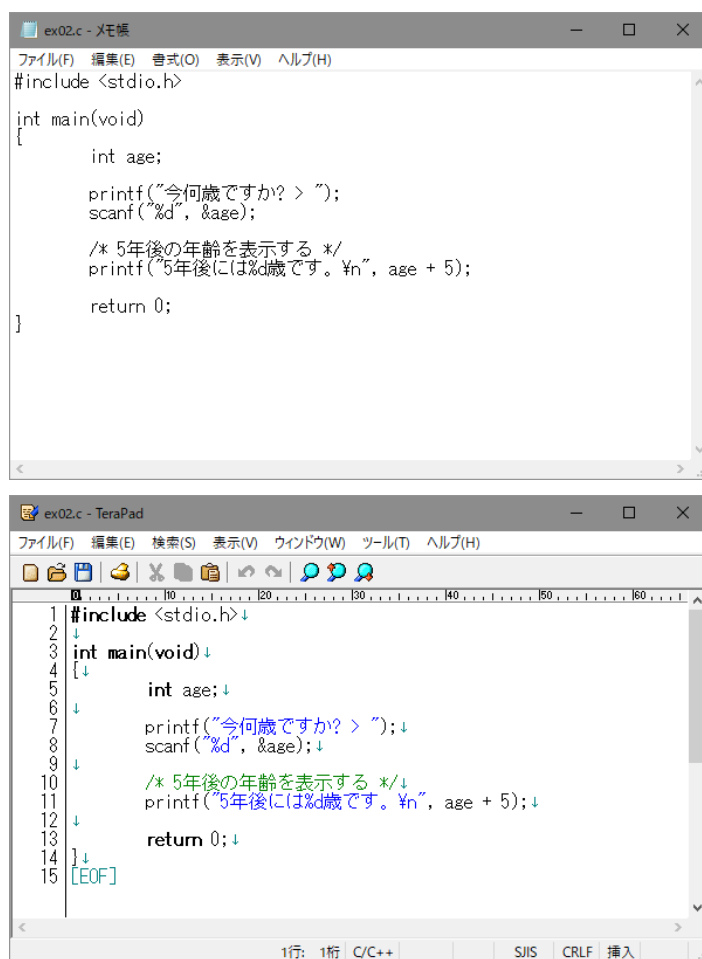


図 2.1 メモ帳(上)と TeraPad(下)でのソースコードの表示

何を使ってよいかわからない場合、Windows であれば、テキストエディタとして TeraPad、コンパイラとして MinGW*2 を使ってください。TeraPad と MinGW のインストール方法や使い方は、それぞれ付録 A と付録 B で説明しています。

それら以外のソフトウェアのインストール方法や使い方の説明は省略します。Web などを参考にインストールし、使い方を学習してください。統合開発環境などと呼ばれる、テキストエディタ、コンパイラ、実行環境などが一体化したソフトウェアを使ってもかまいません。

確認のために、ソースコード 2.1 をコンパイルして実行してください。授業のサイトには、

*2 正確には、MinGW に含まれる GCC。

ex01.c として掲載しています。実行すると、“こんにちは”と表示されます。

ソースコード 2.1 動作確認用プログラム (ex01.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("こんにちは\n");
6
7      return 0;
8  }
```

実行結果

こんにちは

コンパイルできなかったり、実行結果がおかしかったりする場合は、次のような点を確認してください。

コンパイルできない場合

コンパイルのコマンドにパスが通っていますか。Windows のコマンドプロンプトにおける環境変数 PATH の設定方法は、付録 B.2 で説明しています。コマンド名^{*3}だけ入力して、次のように表示される場合、コマンドへのパスが通っていません。

メッセージ

'gcc' は、内部コマンドまたは外部コマンド、
操作可能なプログラムまたはバッチ ファイルとして認識されていません。

その他の環境で、“コマンドが見つかりません”や“command not found”などと表示される場合も同様の原因と考えられます。

コマンドの書式は正しいですか。付録 B.3 でも紹介したとおり、MinGW の gcc の書式は、

```
gcc -o output_file source_file
```

です。-o (小文字のオー) を誤って -0 (数字のゼロ) と入力した場合、次のようなエラーメッセージが表示されます。

^{*3} コンパイルのコマンド名は、コンパイラやインストールした環境によって、gcc、cc、cl、clang など様々です。

gcc のエラーメッセージ

```
gcc: error: ex01.exe: No such file or directory
gcc: error: unrecognized command line option '-0'
```

`output_file`(出力される実行ファイル)と `source_file`(ソースファイル)の順番を逆にしてみても、正しくコンパイルできません。

実行できない場合

意図する名前の実行ファイルができていますか。入力する実行ファイル名が間違えている場合、上で説明したパスが通っていない場合と同様のメッセージが表示されます。

Windows のコマンドプロンプトでは、カレントディレクトリにある実行ファイルは、ファイル名を入力するだけで実行できます。その他の環境では、カレントディレクトリにあるファイルであることを指定しないと動作しないことがあります。たとえば、カレントディレクトリにある `ex01` という名前の実行ファイルを実行する場合、`./ex01` と入力しなければならない環境があります。

実行結果がおかしい場合

“こんにちは”ではなく、文字化けした文字列が表示される場合、ソースファイルの文字コードが正しくない可能性があります。図 2.2 は Shift_JIS で保存すべき環境で UTF-8 で保存した場合、図 2.3 は UTF-8 で保存すべき環境で Shift_JIS で保存した場合の、文字化けの例です。

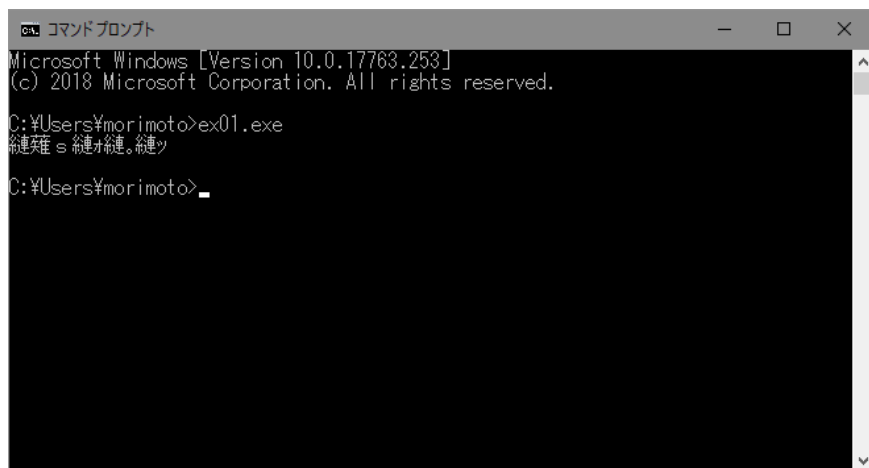


図 2.2 Shift_JIS の環境で UTF-8 を使った場合の文字化け例

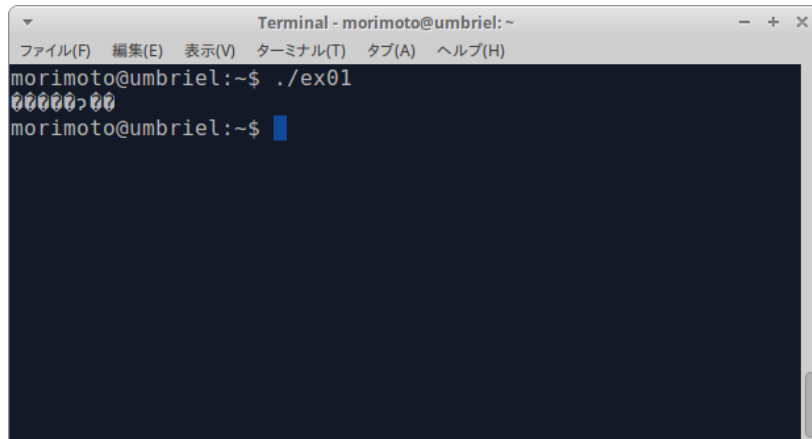


図 2.3 UTF-8 の環境で Shift_JIS を使った場合の文字化け例

【参考】 日本語文字

文字化けに限らず、日本語の文字 (いわゆる全角文字) はトラブルを起こす原因になります。たとえば、Shift_JIS の環境で “こんにちは” とは表示できましたが、“表示” と表示しようとすると、コンパイル時に警告が出たり、実行時に文字化けしたりします。“表” という文字は、十六進数で 95 5c というバイト列です^a。このうち 5c は、特別な役割を持つ \ (バックスラッシュ) の文字コードです。このことから、“表” の一部が \ と認識され、問題が起こります。そのほか、カタカナの “ソ” や漢数字の “十” など、よく使う文字にも 5c は含まれます。このような文字は特別な回避策を行うか、使用を避けるしかありません。コンパイル時のエラーメッセージで日本語部分が文字化けすることもあります。この授業ではわかりやすさのため日本語を使っていますが、正しく日本語を扱うためには、より高度な知識が必要になります。トラブルを避けるためには、日本語を使わない (半角文字だけを使う) ようにしてください。

^a “表” は 2 バイトで表現され、1 バイト目が 10010101 (十六進数で 95)、2 バイト目が 01011100 (十六進数で 5c) です。

第3章

はじめの一步

3.1 最初のプログラム

それでは、最初のプログラムを書いてみましょう。最初に書くプログラムは、ソースコード 3.1 です。

ソースコード 3.1 最初のプログラム (ex02.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int age;
6
7      printf("今何歳ですか? > ");
8      scanf("%d", &age);
9
10     /* 5年後の年齢を表示する */
11     printf("5年後には%d歳です.\n", age + 5);
12
13     return 0;
14 }
```

これをテキストエディタに入力し、ソースファイルとして保存してください。ソースコード 3.1 を保存するソースファイルの名前は、ex02.c とします。授業のサイトに正解例などを示しますので、混乱しないように、できるだけ指定のファイル名で作成してください。初回なので、文字をコピー・アンド・ペーストしたりせず、一から入力してみましょう^{*1}。なお、11 行目の句点 (。) の後

^{*1} これ以降作成するソースコードは、別のソースコードをコピーして必要な箇所だけ書き換えるなど、効率的に作成してください。

ろの記号はバックスラッシュ (\) です。バックスラッシュを入力、または表示すると、円記号 (¥) に見えることがあります。円記号に見えていても問題ありません。

ソースコードが作成できたら、コンパイルし、実行してください。ソースコード 3.1 は、年齢を入力すると、5 年後の年齢を表示するプログラムです。

実行例

```
今何歳ですか? > 42 ↵  
5 年後には 47 歳です。
```

ソースコードは、1 文字でも間違えると動かないことがあります。これだけ短いプログラムでも、一度の記述で正しくコンパイル、実行できることは多くありません。コンパイルでエラーが起こったら、ソースコードを修正して、再度コンパイルしてください。起こしがちな誤りとエラーメッセージの例を付録 C に示します。エラーメッセージには、ソースファイルの行番号が表示されます。その行か、少し上の行に原因があることがほとんどです。

ソースコード 3.1 の各部分の名称や役割を簡単に紹介します。

1 行目のように # から始まる行を**プリプロセッサディレクティブ**と呼びます。ここで使われているのは、プリプロセッサディレクティブの 1 つである `#include` **ディレクティブ**です。プリプロセッサディレクティブについては、第 17 章で説明します。

3 行目から 14 行目まで、つまりこのソースコードのほとんどの部分は `main()` **関数**^{*2}です。4 行目の “{” から 14 行目の “}” までがひとかたまりになっていて、その見出しである `main` が 3 行目に書かれている、と現時点では考えてください。

5 行目では、`int` 型の**変数** `age` を宣言しています。変数については、第 4 章で説明します。

7 行目では `printf()` **関数**、8 行目では `scanf()` **関数**を使っています。`printf()` 関数はプログラム内のデータを出力する役割、`scanf()` 関数はキーボードからデータを読み取ってプログラム内に入力する役割を持っています。どちらも第 5 章で説明します。

7 行目、8 行目、11 行目には 2 つの " の間に文字列^{*3}が入った部分があります。“今何歳ですか? > ” や “%d” の部分です。これを**文字列リテラル**といいます。文字列リテラルを閉じ忘れる、つまり 2 つ目の " を忘れてしまう誤りが多いので気をつけてください。

10 行目の `/*` から始まって `*/` で終わる部分を**コメント**といいます。コメントは人間が読むためのもので、プログラムの動作には影響を与えません。この例では、次の行 (11 行目) で何を行っているかを書いています。自分で書いたプログラムでも、日がたつと意図を忘れてしまうことがあります。ソースコードを読んでも意図がすぐにはわからないような箇所には、なるべくコメントを書くようにしてください。次のように、ソースコードの先頭に、役割や著作権表示などをコメントとして記載することも一般的です。

^{*2} 関数名 (この例では `main`) は後ろに `()` を付ける慣例があります。“`main` 関数” でも不自然ではありませんが、この授業では `()` を付けることにします。

^{*3} 0 文字以上の文字の並び。

ソースコード 3.2 コメントの例

```

1  /*
2   * ex02.c
3   *
4   * キーボードから年齢(整数)を読み取って、
5   * 5年後の年齢を表示するプログラム
6   *
7   * 作成日: 2020年4月1日
8   *
9   * 作成者: 森本
10  */
11
12  #include <stdio.h>
13
14  int main(void)
15  {
16      int age;

```

1 行目の `/*` がコメントの開始、10 行目の `*/` がコメントの終了です。その間には `*/` 以外^{*4}、何を書いてもかまいません。このように複数行にわたって書くこともできます。2 行目から 9 行目は、スペースと `*` で始まっていますが、これはコメントの内部であることをわかりやすくしているだけで、必須ではありません。

ソースコード 3.1 の 13 行目は `return` 文と呼ばれます。`return` 文については、第 15 章で説明します。

^{*4} `*/` と書くと、その部分でコメントが終わったと見なされてしまいます。

3.2 当面の定型部分

当面は、次の記述 (5 行目のコメント以外) を定型部分と考えてください。

ソースコード 3.3 当面の定型部分

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      /* 必要な命令をここから記述する */
6
7      return 0;
8  }
```

4 行目の { の次行から、return 文の前行までの間に必要な命令を記述します。すると、記述した順番で命令が実行されます。定型部分の役割は以降の章で説明します。

3.3 コーディングスタイル

C 言語のソースコードには、改行やスペースを比較的自由に入れられます。ソースコード 3.4 は、ソースコード 3.1 の改行位置やスペースの有無だけを変更したものです。

ソースコード 3.4 読みづらいソースコード

```
1      #include <stdio.h>
2  int main
3  (void){int age;printf(
4      "今何歳ですか? > ");    scanf("%d",
5  &age);/* 5年後の年齢を表示する */
6  printf
7  ("5年後には%d歳です.\n", age
8      + 5
9  );    return 0;}
```

正しくコンパイル、実行でき、全く同じ動作をしますが、とても読みづらいソースコードになっています。これはわざとらしい例としても、改行やスペースの入れ方は、読みやすさに影響します。ソースコードを記述する上での、読みやすさに関する流儀を、**コーディングスタイル**といいます。この授業では一般的と思われるコーディングスタイルで記述しています。ここでは、ソースコードを読みやすくする具体的な方法を 2 つ紹介します。

まず、ソースコード 3.1 では、5 行目から 13 行目の先頭にタブが 1 つ入っています。このように、対になる “{” から “}” の中をタブで字下げすると読みやすくなります。以降の章では、“{ ~ }” の中に “{ ~ }” が入っているソースコードも出てきます。その場合、内側の “{ ~ }” の中は、タブを使ってもう一段階字下げします。そのような例をソースコード 3.5 に、同じソースコードで字下げしていない例をソースコード 3.6 に示します*⁵。字下げをすると、プログラムの構造がわかりやすくなります。

次に、ソースコード 3.1 の 11 行目では、“, age + 5” と書いています。“,” の後ろと “+” の前後にスペースが入っています。スペースを入れない “,age+5” も正しい記述ですが、スペースを入れたほうが読みやすくなります。スペースをどこに入れるべきかは、細かくなりすぎるので説明を省略します。この授業での記述を参考に、読みやすいソースコードを書くように心がけてください。

*⁵ 正しいソースコードですが、あまり意味のない動作をします。また、この授業では学習しない内容を含んでいます。

ソースコード 3.5 字下げあり

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int ia[] = {3, 5, 1, 4, 2};
6      int i;
7
8      for (i = 4; i >= 1; --i)
9      {
10         int max;
11         int j;
12         for (j = 1, max = 0; j <= i; ++j)
13         {
14             if (ia[j] > ia[max])
15             {
16                 max = j;
17             }
18         }
19         if (max != i)
20         {
21             int tmp = ia[max];
22             ia[max] = ia[i];
23             ia[i] = tmp;
24         }
25     }
26
27     return 0;
28 }
```

ソースコード 3.6 字下げなし

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  int ia[] = {3, 5, 1, 4, 2};
6  int i;
7
8  for (i = 4; i >= 1; --i)
9  {
10 int max;
11 int j;
12 for (j = 1, max = 0; j <= i; ++j)
13 {
14 if (ia[j] > ia[max])
15 {
16 max = j;
17 }
18 }
19 if (max != i)
20 {
21 int tmp = ia[max];
22 ia[max] = ia[i];
23 ia[i] = tmp;
24 }
25 }
26
27 return 0;
28 }
```

第 4 章

変数

4.1 変数と型

プログラムの中ではいろいろなデータを扱います。それらを記憶しておくための入れ物を**変数**といいます。変数の中にデータを入れておくと、後から取り出して使うことができます。

プログラムの中で扱うデータには、文字、整数、小数などの種類があります。それらを**型**といいます。C 言語の型はたくさんありますが、この授業では表 4.1 の 3 つだけを使います。char^{*1}は文字の型、int は整数の型、double は小数の型です。C 言語の変数には、入れられるデータの型が決まっています。char 型のデータを入れられる変数を **char 型の変数**といいます。同様に、int 型のデータを入れられる変数が **int 型の変数**、double 型のデータを入れられる変数が **double 型の変数**です。

表 4.1 この授業で使う型

型	データの種類
char	文字
int	整数
double	小数

変数は名前を持っていて、プログラムの中で変数を指定するときにその名前が使われます。つまり、変数は型と名前を持っています。第 3 章で登場した“int 型の変数 age”とは、int 型のデータを入れられる age という名前を持った変数です。図 4.1 は、int 型の変数 age に int 型のデータ (整数) である 42 を入れている様子です。

^{*1} “チャー”と読まれることが多いですが、char は character から来ているので、“キャラ”と読まれることもあります。

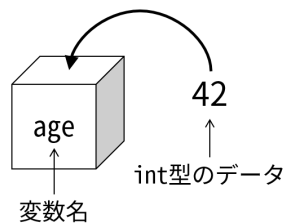


図 4.1 int 型の変数 age

4.2 変数の宣言と値の代入

変数を使うためには、まず、型と名前を指定して変数を作る (データの入れ物を作る) 必要があります。この操作を、変数の**宣言**といいます。

変数宣言の書式は、

```
型名 変数名;
```

です。int 型の変数 age を宣言するには、

```
int age;
```

と書きます。

“{” から “}” までを**ブロック**、または**複合文**といいます*2。変数の宣言は、ブロックの先頭で行う必要があります。複数個の変数を作るためには、変数の宣言を複数回行います。ただし、それらの間に変数宣言以外の文が出てきてはいけません (図 4.2)。

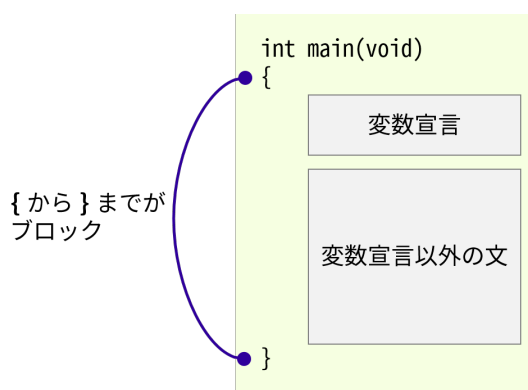


図 4.2 変数の宣言

*2 第 8 章でもう一度説明します。

変数宣言の例をいくつか示します。

ソースコード 4.1 変数宣言の例

```
1 char initial; /* char型変数initialを宣言 */
2 int age;      /* int型変数ageを宣言 */
3 double height; /* double型変数heightを宣言 */
```

同じ型の変数を複数宣言するときは、次のように、変数名を , (コンマ) で区切ることによって 1 文で書くことができます。

ソースコード 4.2 同じ型の変数を複数宣言

```
1 /*
2  * int型の変数xとyとzを1文で宣言する例
3  *
4  * 次の3文と同じ
5  * int x;
6  * int y;
7  * int z;
8  */
9 int x, y, z;
```

以降、変数に入れるデータを“値”といいます。変数の宣言は、値の入れ物を作るだけです。変数に値を入れるには、記号 = を使って

変数名 = 値;

と書きます。= の左側に変数名を、右側に値を書き、最後に ; を付けます。int 型の変数 age に 42 を入れるには、

age = 42;

と書きます。数学で $age = 42$ と書くと、age と 42 が等しいという意味ですが、C 言語の = は、右側の値を左側の変数に入れる役割を持っています。変数に値を入れることを、値を代入するといいます。

変数への代入の例をいくつか示します。

ソースコード 4.3 変数の宣言と値の代入

```
1 char c; /* char型変数cを宣言 */
2 int i; /* int型変数iを宣言 */
3 double d; /* double型変数dを宣言 */
4
5 c = 'A'; /* 変数cに文字 A を代入 */
6 i = 150; /* 変数iに整数 150 を代入 */
7 d = 3.14 /* 変数dに小数 3.14 を代入 */
```

整数と小数は 150 や 3.14 のように、そのまま記述します。文字は、' (シングルクォーテーション) で囲んでください。ソースコード 4.3 の例では、`c = A;` とすると誤りです。また、文字には半角英数字と一部の記号しか使えません。`c = 'あ';` も誤りです。

'A' のように、文字を ' で囲んだものを**文字定数**といいます。第 3 章で学んだ文字列リテラルは、文字列を " で囲んだものです。文字定数と文字列リテラルは異なりますので、注意してください。char 型変数に文字列リテラルを代入することはできません。`c = "A";` は誤りです。文字定数と文字列リテラルの例を、表 4.2 に示します。

表 4.2 文字定数と文字列リテラル

文字定数の例	文字列リテラルの例
'A'	"A"
'z'	"xyz"
'3'	"山田 太郎"
'?'	""

変数を宣言してから値を代入するのではなく、値を指定して変数を宣言することもできます。書式は、

型名 変数名 = 初期値;

です。このように、変数にあらかじめ値 (初期値) を入れておくことを**初期化**といいます*3。初期化を行わない変数宣言と同様、, (コンマ) で区切ることによって、同じ型の変数を複数宣言することができます。

*3 代入に使う = と初期化に使う = は、厳密には異なるものです。

初期値を持った変数を宣言する例をいくつか示します。

ソースコード 4.4 変数の宣言と初期化

```
1  /* char型変数cを宣言し、文字 Z で初期化 */
2  char c = 'Z';
3
4  /*
5   * int型変数i、j、kを宣言
6   *
7   * iは 3 で初期化
8   * jは初期値なし
9   * kは -5 で初期化
10  */
11 int i = 3, j, k = -5;
```

変数は、宣言時に名前を付けました。変数や、以降の章で学習する関数に付けられる名前を**識別子**といいます。識別子として使える文字列は、先頭が英文字 (A~Z、a~z) かアンダースコア (_)、2 文字目以降が英数字 (英文字、0~9) かアンダースコアである必要があります。また、`int` や `return` は、C 言語のソースコード中で特別な意味を持っており、**キーワード**や**予約語**と呼ばれます。キーワードを識別子として使うことはできません。正しくない識別子の例を示します。

ソースコード 4.5 正しくない識別子 (変数名)

```
1  int 12gatsu; /* 先頭に数字は使えない */
2  int my-height; /* ハイフンは使えない */
3  int return; /* キーワードは使えない */
```

識別子は大文字と小文字が区別されます。たとえば `int x, X;` と宣言すれば、`x` と `X` を別の変数として同時に使うことができます^{*4}。変数には、役割がわかるような識別子を付けてください。ただし、一時的に使う変数や、すぐに役割がわかるような変数は `i` や `n` などの簡単な識別子でもかまいません^{*5}。複数の語をつなげて識別子を作る方法として、表 4.3 のような流儀があります。どの方法を使ってもかまいませんが、C 言語ではアンダースコアでつなげる方法が採用されることが多いようです。

C 言語では、変数に何も入っていないという状態はありません。初期化を行わない変数宣言をした直後でも、変数には何らかの値が入っています。ただし、その値は不定です。次章以降で、変数の値を使った処理を行います。変数は、必ず初期化をするか、値を代入してから使う必要があります。

^{*4} 紛らわしいのでおススメはできません。

^{*5} この授業でもそのような識別子を多数使っています。

表 4.3 複数の語をつなげた識別子の作り方

方法	例 1	例 2
アンダースコア (<code>_</code>) でつなげる	<code>my_age</code>	<code>login_user_id</code>
2 つ目以降の語の先頭を大文字にしてつなげる	<code>myAge</code>	<code>loginUserId</code>
すべての語の先頭を大文字にしてつなげる	<code>MyAge</code>	<code>LoginUserId</code>

練習課題 1 (ex03.c)

次のようなプログラムを作ってください。

- `char` 型変数 `initial` を宣言し、文字 `M` を代入する。
- `int` 型変数 `age` を宣言し、整数 `42` を代入する。
- `double` 型変数 `height` を宣言し、小数 `164.5` を代入する。

ソースコードを `ex03.c` というファイル名で保存し^a、コンパイル、実行してください。

^a これ以降の練習課題では、タイトル部分 (この例では「練習課題 1 (ex03.c)」) に記載したファイル名で保存してください。

【発展】 型

int 型は整数であると説明しましたが、どのような整数でも int 型で表せるわけではありません。また、C 言語で整数を表す型は int 型だけではありません。扱える範囲などに応じて、複数の型が用意されています。実は、int 型で扱える範囲は具体的には決められておらず、処理系によって違います。少なくとも $-32,767$ から $32,767$ の範囲の整数を扱えなければならないと決められているだけで、ほとんどの処理系では、これ以上の範囲の整数を扱えます。代表的な処理系では、 $-2,147,483,648$ から $2,147,483,647$ (約 -21 億から 21 億) の範囲の整数を扱えます。さらに大きな数や小さな数を扱いたいときは、long int 型が使える可能性があります。long int 型は、int 型と同じか、より広い範囲の整数を扱えると決められています。授業のサイトに掲載している adv01.c をコンパイル、実行すると、その処理系で扱える int 型と long int 型の範囲が表示されます。

Windows 10 上の MinGW(gcc version 8.2.0) での実行結果

int 型の範囲: $-2147483648 \sim 2147483647$

long int 型の範囲: $-2147483648 \sim 2147483647$

FreeBSD 12.1 上の Clang 8.0.1 での実行結果

int 型の範囲: $-2147483648 \sim 2147483647$

long int 型の範囲: $-9223372036854775808 \sim 9223372036854775807$

Windows 10 上の MinGW(gcc version 8.2.0) では、int 型と long int 型の範囲は同じでした。FreeBSD 12.1 上の Clang 8.0.1 では、long int 型は約 -922 京から 922 京の範囲の整数を扱えます。

long int 型を使うには、単純に int を long int に置き換えればよいわけではありません。この後の int 型に対する説明は、long int 型には当てはまらないこともあります。

第 5 章

入出力

前章の練習課題 1 では、変数を作って、値を代入しました。しかし、内部でデータを処理しただけで、役に立つプログラムではありません。プログラムは、外部からデータを取り込んで、処理をして、その結果を外部に戻すことで役割を果たすことができます。プログラムにデータを入力するには、ユーザにキーボードから入力してもらったり、ファイルから読み取ったりします。プログラム内のデータを外部に出力するには、画面に表示したり、ファイルに書き込んだりします。本章では、画面にデータを表示する方法と、キーボードからデータを読み取る方法を学びます。

5.1 関数の役割と書式

画面にデータを表示するには `printf()` 関数、キーボードからデータを読み取るには `scanf()` 関数を使います。ここでは、`printf()` 関数と `scanf()` 関数を使うために必要な範囲で、関数について簡単に説明します。関数とは、決められた処理を行う命令のまとまりです。`printf()` 関数は画面にデータを表示する処理、`scanf()` 関数はキーボードからデータを読み取る処理を行います。

関数を使うためには関数呼び出しを行います。その書式は、

関数名 (引数₁, 引数₂, 引数₃, ...)

です。引数とは、関数の実行に必要なデータです。関数名の後ろに () を記述し、その中に引数を , (コンマ) で区切りで必要なだけ記述します。どのような引数がいくつ必要かは関数によって異なります。この授業では `printf()` 関数の引数は 1 つ以上、`scanf()` 関数の引数は 2 つ使います。引数₁ を第 1 引数、引数₂ を第 2 引数、のように表現します。

関数呼び出しの例と解釈を、2 つ示します。

例 1: `printf("身長は %dcm、体重は %dkg です。\\n", height, weight)`

関数名は `printf`

引数は次の 3 つ

第 1 引数 … `"身長は %dcm、体重は %dkg です。\\n"`

第 2 引数 … `height`

第 3 引数 … `weight`

例 2: `scanf("%d", &height)`

関数名は `scanf`

引数は次の 2 つ

第 1 引数 … `"%d"`

第 2 引数 … `&height`

どちらの例も、第 1 引数は文字列リテラルです。第 2 引数以降は、変数名や、変数名に記号を加えたものです。

5.2 画面への出力

画面にデータを表示するには、`printf()` 関数を使います。第 1 引数に、表示したい文字列を文字列リテラルとして指定します。“こんにちは”と表示するには、

```
printf("こんにちは");
```

と記述します。前節で示した関数呼び出しの書式には、最後に `;` がありませんでしたが、こちらには付いています。詳しくは第 7 章で説明しますが、`printf()` 関数と次節で説明する `scanf()` 関数を使うときは、最後に `;` を付けてください。

ソースコード 5.1 は、“こんにちは”と表示するだけのプログラムです。

ソースコード 5.1 “こんにちは”と表示するプログラム (ex04.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("こんにちは");
6
7      return 0;
8  }
```

これを、コンパイル、実行してみてください。図 5.1 に Windows のコマンドプロンプトで実行

した場合の例、図 5.2 に別のターミナル*¹で実行した場合の例を示します。

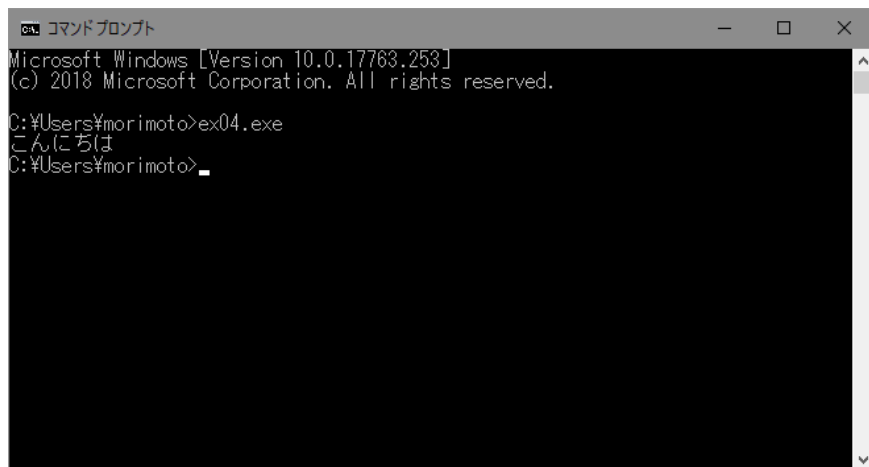


図 5.1 Windows のコマンドプロンプトで実行した場合

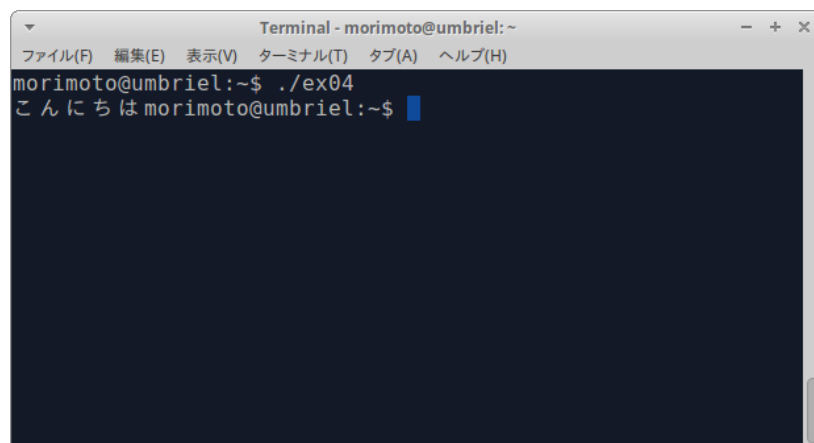


図 5.2 別のターミナルで実行した場合

図 5.2 のターミナルでは、“こんにちは”の後ろに、次のプロンプト*²が続いています。コマンドプロンプトは実行後に自動的に改行が行われますが、図 5.2 のターミナルは自動的な改行は行われません。コマンドプロンプトが特殊で、改行が行われないターミナルがほとんどです。出力する文字列の最後は、明示的に改行するようにします。改行も文字の一種で、改行を表す文字を出力 (表示) することによって改行できます。printf() 関数の第 1 引数として指定する文字列リテラル中の改行したい場所に \n を記述してください。ソースコード 5.2 は、\n による改行の例です。

*¹ ターミナル (ターミナルエミュレータ) とは、コマンドを入力、実行し、その出力を表示できるソフトウェア。コマンドプロンプトは、ターミナルの一種。

*² ユーザにコマンドの入力を促す文字列。この例では “morimoto@umbriel:~\$”。

ソースコード 5.2 改行の出力 (ex05.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("こんにちは\n");
6      printf("寒いですね\n調子はいかがですか\n");
7
8      return 0;
9  }

```

実行結果

```

こんにちは
寒いですね
調子はいかがですか

```

`\n` の場所で改行されていることがわかります。複数行の文字列を出力するときは、`printf()` 関数を連続して書いてもかまいませんし、`printf()` 関数の第 1 引数に改行を複数含む文字列リテラルを指定してもかまいません。出力する文字列の長さなどに応じて使い分けてください。

`\n` は、`\` と `n` の 2 文字ではなく、改行を表す 1 つの文字です。文字列リテラル中に直接改行を書くことができないため、このような特殊な形式を使います。このような表記形式を**エスケープシーケンス**といいます。エスケープシーケンスの例を、表 5.1 に示します。

表 5.1 エスケープシーケンスの例

表記	意味
<code>\n</code>	改行
<code>\t</code>	水平タブ
<code>\'</code>	文字 <code>'</code>
<code>\"</code>	文字 <code>"</code>
<code>\\</code>	文字 <code>\</code> または ¥

文字 ' (シングルクォーテーション) を表す文字定数は、''' と表記することはできません。次のように、エスケープシーケンスを使って '\'' と表記します。

ソースコード 5.3 エスケープシーケンス \'

```
1 /* char型変数c1を文字 ' で初期化 */
2 char c1 = '\'';
3
4 /* 以下は誤り (コンパイルエラー) */
5 char c2 = ''';
```

同様に、文字列リテラルの中に " を直接書くことはできません。エスケープシーケンス \" を使う必要があります。たとえば、「He said "Hello."」と出力したい場合は、次のように記述します。

ソースコード 5.4 エスケープシーケンス \"

```
1 /* 画面に「He said "Hello."」と出力 */
2 printf("He said \"Hello.\"");
3
4 /* 以下は誤り (コンパイルエラー) */
5 printf("He said "Hello.");
```

練習課題 2 (ex06.c)

学生番号と名前を表示するプログラムを作ってください。学生番号の後ろで改行し、名前は次の行に表示します。名前が文字化けする場合は、ローマ字で記述してください^a。

実行例

123-456789-0

森本 容介

^a 名前に漢数字の“十”などが入っていると文字化けすることがあります。第2章末尾の参考記事もご参照ください。

ここまでで、決まった文字列が表示できるようになりました。次は、変数の中身 (値) を表示する方法を説明します。printf() 関数は、第1引数の文字列リテラル中に %d と記述すると、その部分が第2引数に指定された整数に置き換わる、という機能を持っています。たとえば、

```
printf("私は %d 歳です。\\n", 42);
```

と記述したとします。第1引数の文字列リテラル中の %d が、第2引数に指定された整数である

42 に置き換わります。そして、画面に“私は 42 歳です”と表示されます。この例であれば、はじめから

```
printf("私は 42 歳です。\\n");
```

と記述すればよいのですが、第 2 引数には変数を指定できます。たとえば、`int` 型の変数 `age` に 42 が入っているとします。

```
printf("私は %d 歳です。\\n", age);
```

と記述すると、`%d` が変数 `age` の値である 42 に置き換わり、“私は 42 歳です”と表示されます。第 1 引数の文字列リテラル中に、`%d` を複数含めることもできます。`%d` を 2 つ含めた場合は、1 つ目の `%d` が第 2 引数に指定された整数に、2 つ目の `%d` が第 3 引数に指定された整数に置き換わります。3 つ以上含めた場合も同様です。

`%d` を複数使ったプログラムの例をソースコード 5.5 に、この 9 行目で使っている `printf()` 関数の解説を図 5.3 に示します。

ソースコード 5.5 複数の `%d` を使った例 (ex07.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int koku = 75; /* int型変数kokuを 75 で初期化 */
6      int san  = 93; /* int型変数san を 93 で初期化 */
7      int sya  = 68; /* int型変数sya を 68 で初期化 */
8
9      printf("国語は%d点、算数は%d点、社会は%d点です。\\n", koku, san, sya);
10
11     return 0;
12 }
```

実行結果

国語は 75 点、算数は 93 点、社会は 68 点です。

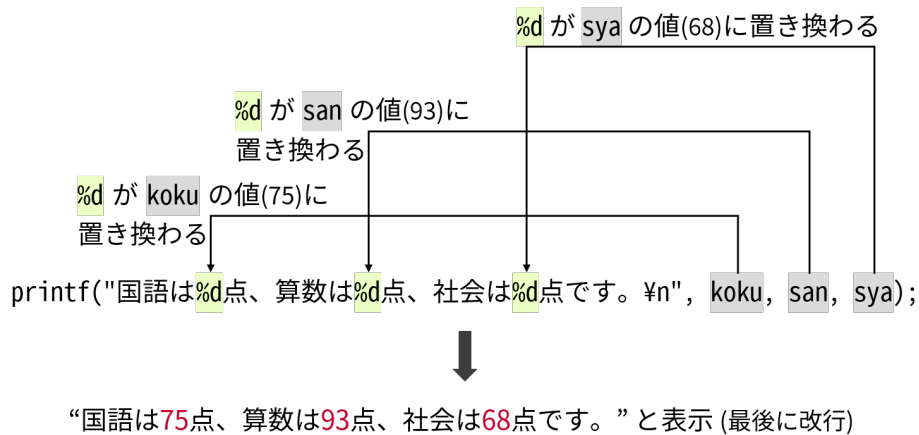


図 5.3 ソースコード 5.5 の 9 行目

`%d` を**変換指定**といいます。変換指定 `%d` は整数に置き換わりました。小数に置き換えるためには `%f`、文字に置き換えるためには `%c` を使います。`%f` で小数に置き換えるとき、小数点以下の桁数は 6 桁になります。たとえば、

```
printf("%f\n", 4.56);
```

では、`4.560000` と表示されます。小数点以下 1 桁までにしたい場合は `%.1f`、2 桁までにしたい場合は `%.2f` のように、`%` と `f` の間に、`.` (ドット) と小数点以下の桁数を記述します。

```
printf("%.1f\n", 4.56);
```

であれば `4.6` (四捨五入)、

```
printf("%.3f\n", 4.56);
```

であれば `4.560` と表示されます。

`printf()` 関数の第 1 引数に指定する文字列リテラルの中では、`%` は特別な役割を持っています。`printf()` 関数で `%` 自体を表示させたいときは、変換指定 `%%` を使います。変換指定 `%%` は、文字 `%` に置き換わります*3。その他の変換指定とは異なり、`%%` には、対応する引数を指定しません。

*3 エスケープシーケンスの役割に似ています。ただし、`%` が特別な役割を持たない文字列リテラル中では、`%` を `%%` にする必要はありません。なお、この授業では文字列リテラルは、`printf()` 関数の第 1 引数と、次節で説明する `scanf()` 関数の第 1 引数にしか使いません。

この授業で使う変換指定を、表 5.2 にまとめます。

表 5.2 printf() 関数の第 1 引数で使う変換指定

変換指定	役割
%c	文字に置き換えられる。
%d	整数に置き換えられる。
%f	小数に置き換えられる。%.1f は小数第 1 位まで、%.2f は小数第 2 位まで。
%%	文字 % に置き換えられる。対応する引数は指定しない。

これらの変換指定を使った例を、ソースコード 5.6 に示します。ソースコードと実行結果を見比べてください。10 行目ではエスケープシーケンス \" を、14 行目では変換指定 %% を使っています。

ソースコード 5.6 様々な変換指定 (ex08.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int score = 85;      /* 得点 */
6      char grade = 'A';    /* 評語 */
7      double pi = 3.14159; /* 円周率 */
8      int chance = 30;     /* 降水確率 */
9
10     printf("得点は%d点、評語は\"%c\"です。\\n", score, grade);
11
12     printf("円周率として%.2fか%.3fを使って計算してください。\\n", pi, pi);
13
14     printf("降水確率は%d%%です。\\n", chance);
15
16     return 0;
17 }
```

実行結果

得点は 85 点、評語は"A"です。
円周率として 3.14 か 3.142 を使って計算してください。
降水確率は 30% です。

練習課題 3 (ex09.c)

練習課題 1(第 4 章) で作った ex03.c をコピーして、ファイル名を ex09.c にしてください^a。ex09.c に文を追加して、変数 initial、age、height を使って次のように表示するプログラムを作ってください。

実行結果 1

M さんの年齢は 42 歳、身長は 164.500000cm です。

“164.500000cm” の部分に不要な “0” が表示されることが気になるようであれば、次のように小数第 1 位まで表示するようにしてください。

実行結果 2

M さんの年齢は 42 歳、身長は 164.5cm です。

^a これ以降も、以前に使ったプログラムを改変する練習課題を出題します。その場合は、以前のプログラムをコピーして書き換えてください。

5.3 キーボードからの入力

キーボードからデータを読み取って変数に格納するには、`scanf()` 関数を使います。この授業では、`scanf()` 関数には引数を 2 つ指定します^{*4}。第 1 引数には、変換指定のみを含む文字列リテラルを記述します。第 2 引数には、& に続けて、値を格納する変数名を記述します。読み取った整数を `int` 型の変数に格納するためには、変換指定 `%d` を使います。たとえば、変数 `age` が `int` 型であるなら、

```
scanf("%d", &age);
```

と記述すると、キーボードから読み取った整数が変数 `age` に格納されます。動作は次のとおりです。プログラムの実行が `scanf("%d", &age);` にさしかかると、プログラムは入力待ちの状態に

^{*4} `printf()` 関数も `scanf()` 関数も多数の機能を持っています。この授業では代表的な使い方だけを説明します。

なります。ユーザは、キーボードから整数を入力してエンターキーを押します。すると、`scanf()` 関数の働きによって入力された整数が読み取られ、変数 `age` に格納されます。その後、プログラムの実行は先に進みます。

この授業で使う変換指定を、表 5.3 に示します。

表 5.3 `scanf()` 関数の第 1 引数で使う変換指定

変換指定	役割
<code>%c</code>	読み取った文字を <code>char</code> 型の変数に格納する。
<code>%d</code>	読み取った整数を <code>int</code> 型の変数に格納する。
<code>%lf</code> (エル・エフ)	読み取った小数を <code>double</code> 型の変数に格納する。

`printf()` 関数と `scanf()` 関数の使い方は似ていますが、次のような点が異なります。

- `scanf()` 関数の第 1 引数の文字列リテラルには、**変換指定だけ**を記述します。
- 小数を表示するための `printf()` 関数の変換指定は `%f` ですが、小数を読み取って `double` 型の変数に格納するための `scanf()` 関数の変換指定は `%lf` です。
- `scanf()` 関数の第 2 引数に指定する変数名は、前に `&` を付けます。

`&` はポインタに関連した記号ですが、この授業の範囲外です。`scanf()` 関数の第 2 引数は、変数名の前に必ず `&` が必要と覚えてください。

ソースコード 5.7 は、名字の頭文字と年齢を入力すると、それらを使った文を表示するプログラムです。

ソースコード 5.7 `scanf()` 関数の使い方の例 (`ex10.c`)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char initial;
6      int age;
7
8      scanf("%c", &initial);
9      scanf("%d", &age);
10
11     printf("%cさんの年齢は%d歳です.\n", initial, age);
12
13     return 0;
14 }
```

これを、コンパイル、実行してみてください。実行すると、プログラムが入力待ちの状態になります。まず、キーボードから文字を1つ入力してエンターキーを押してください。次に、整数を1つ^{*5}入力してエンターキーを押してください。入力した文字と整数を使った文が表示されます。

実行例

M ↵

42 ↵

Mさんの年齢は 42 歳です。

入力した文字と整数が正しく読み取られていることがわかります。ただ、このプログラムは、最初に文字を入力して次に整数を入力することを知っていないと使うことができません。何を入力したらよいかを表示すると親切です。そのように改良したものを、ソースコード 5.8 に示します。

ソースコード 5.8 ソースコード 5.7 の改良 (ex11.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char initial;
6      int age;
7
8      printf("名字の頭文字を入力してください > ");
9      scanf("%c", &initial);
10     printf("年齢を入力してください > ");
11     scanf("%d", &age);
12
13     printf("%cさんの年齢は%d歳です.\n", initial, age);
14
15     return 0;
16 }
```

実行例

名字の頭文字を入力してください > M ↵

年齢を入力してください > 42 ↵

Mさんの年齢は 42 歳です。

^{*5} 数字を1つではなく、整数を1つです。

scanf() 関数を使う前、つまりユーザにデータを入力してもらう前に、printf() 関数を使って、何を入力したらよいかを表示しています。ここでは、入力を促していることをわかりやすくするために、“入力してください”の後ろに > を表示しています。これまでと違い、printf() 関数で最後に改行していない点にもご注目ください。

練習課題 4 (ex12.c)

ソースコード 5.8(ex11.c) を改変して、次のようなプログラムを作ってください。

- 名字の頭文字を入力してもらう。
- 年齢を整数で入力してもらう。
- 身長を小数で入力してもらう。
- 入力されたデータをもとに、“○さんの年齢は○○歳、身長は○○○.○ cm です。”と表示する。身長は小数第 1 位まで表示する。

なお、この授業で説明した方法を使えば、小数で入力されることが期待される箇所に整数で入力されても問題ありません。

実行例 1

名字の頭文字を入力してください > M ↵
年齢を入力してください > 42 ↵
身長を入力してください > 164.5 ↵
M さんの年齢は 42 歳、身長は 164.5cm です。

実行例 2

名字の頭文字を入力してください > S ↵
年齢を入力してください > 20 ↵
身長を入力してください > 170 ↵
S さんの年齢は 20 歳、身長は 170.0cm です。

第 6 章

算術演算

C 言語で加算 (足し算) を行うには記号 `+` を、減算 (引き算) を行うには記号 `-` を使います。7 + 3 の結果は 10 で、7 - 3 の結果は 4 です。7 や 3 のような具体的な数のほかに、変数も使うことができます。int 型の変数 `x` の値が 5、`y` の値が 2 のとき、`x + 3` の結果は 8 で、`y - x` の結果は -3 です。`+` と `-` は数学で使う記号と同じです。乗算 (かけ算) は `×` ではなく `*` (アスタリスク) を使います。7 * 3 の結果は 21 です。除算 (割り算) は `÷` ではなく `/` を使います。C 言語では整数同士の除算の結果は整数という決まりがあります。整数同士の除算を行うと、計算結果の小数点以下が切り捨てられ、商が得られます。7 / 3 の結果は 2 です。整数を整数で割った剰余 (余り) を求めるには、記号 `%` を使います。7 % 3 の結果は 1 です。ここまでの、表 6.1 にまとめます。

表 6.1 算術演算

記号	役割	例
<code>+</code>	加算	<code>7 + 3 → 10</code>
<code>-</code>	減算	<code>7 - 3 → 4</code>
<code>*</code>	乗算	<code>7 * 3 → 21</code>
<code>/</code>	除算 (商)	<code>7 / 3 → 2</code>
<code>%</code>	剰余	<code>7 % 3 → 1</code>

これらの計算結果は、変数に代入したり、`printf()` 関数を使って表示したりできます。ソースコード 6.1 は、整数を 2 つ入力すると加算と減算を行った結果を表示するプログラムです。15 行目では、加算の結果を変数 `sum` に代入しています。18 行目では、`printf()` 関数を使って、加算と減算の結果を表示しています。加算については計算済みの結果が変数 `sum` に入っていますので、その値を表示します。減算の結果は直接表示します。ソースコードと実行例を見比べて、動きを追ってください。

ソースコード 6.1 加算と減算を行うプログラム (ex13.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a, b;
6      int sum; /* 加算の結果を代入する変数 */
7
8      /* 変数aとbにキーボードから整数を入力 */
9      printf("整数 a > ");
10     scanf("%d", &a);
11     printf("整数 b > ");
12     scanf("%d", &b);
13
14     /* 加算の結果を変数に代入 */
15     sum = a + b;
16
17     /* a + b と a - b の結果を表示 */
18     printf("a + b の結果は %d、a - b の結果は %d です.\n", sum, a - b);
19
20     return 0;
21 }

```

実行例

```

整数 a > 4 ↵
整数 b > 7 ↵
a + b の結果は 11、a - b の結果は -3 です。

```

+ と - と * は、計算対象の数として、0、負の数、小数が使えます。どちらか片方でも小数の場合、計算結果も小数になります*¹。

/ と % は、割られる数、割る数のどちらか片方でも負の数を使った場合、結果は処理系に依存します。たとえば、 $-7 \div 3$ が “-2 余り -1” になったり、“-3 余り 2” になったりします。特別な理由がなければ、負の数を使った除算は行わないでください。また、数学と同じく 0 で割ることはできません。

*¹ $0.5 + 0.5$ は整数の 1 になりそうですが、結果は小数の 1.0 です。C 言語では、整数の 1 と小数の 1.0 は違います。

/ は小数を使うことができます。割られる数、割る数のどちらか片方でも小数の場合、整数同士の除算のような切り捨ては行われず、計算結果は小数です。7 / 3 は 2 でしたが、7.5 / 3 は 2.5 になります。

% は整数同士のものにしか使えません。7.5 % 3 は誤りで、コンパイルできません。

ここまで進めば、第 3 章で書いた最初のプログラム (ソースコード 3.1, ex02.c) が、定型部分を除いて理解できます。以下に再掲します。

ソースコード 6.2 最初のプログラム (ソースコード 3.1 の再掲)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int age;
6
7      printf("今何歳ですか? > ");
8      scanf("%d", &age);
9
10     /* 5年後の年齢を表示する */
11     printf("5年後には%d歳です.\n", age + 5);
12
13     return 0;
14 }
```

7 行目で “今何歳ですか? > ” と表示しています。8 行目でキーボードから入力された整数を 1 つ読み取って、5 行目で宣言した int 型の変数 age に格納しています。11 行目で、5 年後の年齢を求めるため age + 5 を計算し、その結果を使って “5 年後には〇〇歳です。” と表示しています。

練習課題 5 (ex14.c)

整数を 2 つ入力するとそれらの加算、減算、乗算、除算 (商と剰余) の結果を表示するプログラムを作ってください。実行例のように、入力された数と、C 言語の記号 (+ や -) を使って、どのような計算結果なのかをわかりやすく表示してください。

実行例 1

整数 1 > 7 ↵

整数 2 > 3 ↵

7 + 3 は 10

7 - 3 は 4

7 * 3 は 21

7 / 3 は 2

7 % 3 は 1

実行例 2

整数 1 > 4 ↵

整数 2 > 9 ↵

4 + 9 は 13

4 - 9 は -5

4 * 9 は 36

4 / 9 は 0

4 % 9 は 4

第 7 章

演算子と式

7.1 演算子

前章では、加算や減算などの計算を表す記号について説明しました。プログラミングの分野では、計算ではなく**演算**と呼ばれることが多いので、以降そのようにします。演算を表す (演算を行う) 記号を**演算子**といいます。+ や - は演算子です。演算の対象となる値や変数を**被演算子**といいます。“age + 5” は、+ が演算子、age と 5 が被演算子です。演算子 + には被演算子が 2 つ必要です。このような演算子を**二項演算子**といいます。被演算子が 1 つ必要な演算子を**単項演算子**といいます。scanf() 関数の第 2 引数で使われる & は単項演算子です。“&age” は、& が演算子、age が被演算子です。ここまでの概念を、図 7.1 にまとめます。

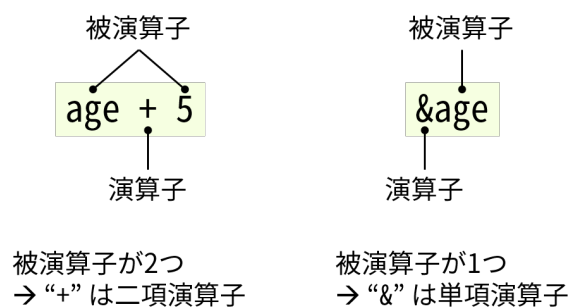


図 7.1 演算子と被演算子

C 言語のほとんどの演算子は単項演算子か二項演算子ですが、三項演算子、つまり被演算子が 3 つ必要な演算子が 1 つだけあります。これは、第 12 章で登場します。

7.2 式と評価

C 言語では、“定数”、“変数”、“演算子と被演算子の組み合わせ”を**式**といいます。**定数**とは、5 や 'A' のように、ソースコード中に直接書いた値のことです。age + 5 や &age のような部分が、

演算子と被演算子の組み合わせです。C 言語の式は型と値を持っており、それを取り出すことを**評価する**といいます。

ソースコード中に書いた定数 5 は、`int` 型と見なされます。この式 (定数) を評価すると `int` 型の 5 が得られます。定数 5.0 は、`double` 型と見なされます。この式 (定数) を評価すると `double` 型の 5(5.0) が得られます。

`int` 型の変数 `a` に 3 が入っているとき、式 (変数) `a` を評価すると `int` 型の 3 が得られます。では次に、式

`a + 5`

を考えます。`a` と 5 はどちらも式で、評価するとそれぞれ `int` 型の 3 と `int` 型の 5 が得られました。二項演算子 `+` は、左辺を評価した値と右辺を評価した値を足します。そして、`+` を使った式を評価すると、加算の結果が得られます。つまり、`a + 5` を評価すると `int` 型の 8 が得られます。

式を評価した値を別の演算に使うことができます。`b` を `int` 型の変数とすると、

`b = a + 5`

を考えます。`=` は右辺を評価した値を左辺の変数に代入する演算子です。つまり、変数 `b` には 8 が代入されます。`b = a + 5` 自体も式です。演算子が `=`、被演算子が `b` と `a + 5` です。代入演算 (`=` を使った演算) を評価すると、代入した値が得られるという決まりがあります。つまり、`b = a + 5` を評価すると `int` 型の 8 が得られます。

このように、式を次々評価することにより、演算が進みます。ここまでの、図 7.2 にまとめます。

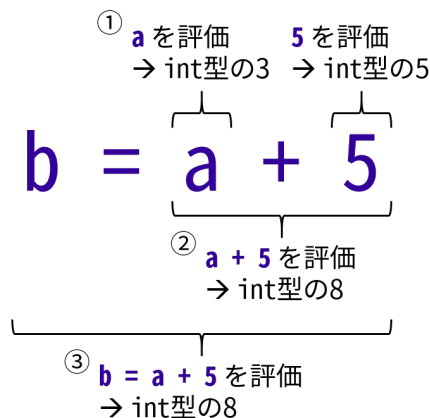


図 7.2 `b = a + 5` の評価

7.3 演算子の優先順位

前節では、式 `b = a + 5` が評価される様子を説明しました。そこでは、`a + 5` を評価してから `b = a + 5` を評価しましたが、`b = a` が評価されてから `b = a + 5` が評価されることはないの

でしょうか。後者の手順を、図 7.3 に示します。int 型の変数 a に 3 が入っているとします。まず、b = a が評価されます。代入演算を評価すると代入した値が得られますので、int 型の 3 が得られます。次に (b = a) + 5、つまり 3 + 5 が評価され、int 型の 8 が得られます。

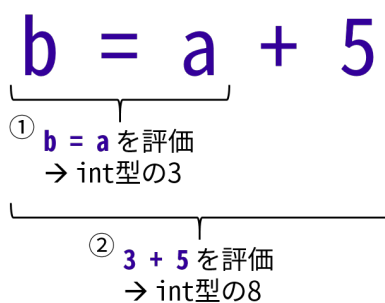


図 7.3 b = a + 5 の評価の別例

図 7.2 の手順では b に 8 が代入されましたが、図 7.3 の手順では b に 3 が代入されました。このように、評価の順序によって、結果が異なります。C 言語に限らず、プログラミング言語では、演算子の優先順位が決められています。C 言語では、= よりも + の方が優先順位が高いと決められていますので、式 b = a + 5 は、図 7.3 ではなく図 7.2 の順に評価されます。

表 7.1 に、この授業に登場する演算子の優先順位を示します。

表 7.1 演算子の優先順位と結合

優先順位	演算子	結合
1	() (関数呼び出し) ++(後置) --(後置)	左
2	++(前置) --(前置) +(単項) -(単項) & !	右
3	() (キャスト)	右
4	* / %	左
5	+(二項) -(二項)	左
6	< <= > >=	左
7	== !=	左
8	&&	左
9		左
10	?:	右
11	= += -= *= /= %=	右
12	,	左

この表の中で、すでに登場した演算子を復習します。

優先順位が 1 の () は、関数呼び出しのための演算子です。printf("こんにちは\n") を使って

いる（と）が該当します。これは演算子のように見えないかもしれませんが、そのようなものだとお考えください。

優先順位が2の`&`は、`scanf()`関数の第2引数に指定する変数名に付ける単項演算子です。

+ と - は、単項演算子と二項演算子があります。単項演算子の方の優先順位は2、二項演算子の方の優先順位は5です。このうち、二項演算子の + と - は、第6章で説明しました。

優先順位が4の * と / と % も、第6章で説明しました。

優先順位が11の = は、代入の演算子です。第4章で説明し、ここまでに何度も使っています。

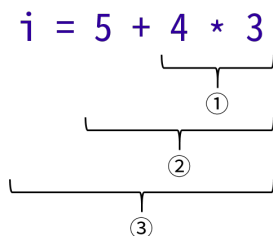
その他の演算子は、これ以降に登場します。

表7.1の中で上位に位置する演算子ほど、高い優先順位を持っています。“結合”列には、“左”、または“右”と書かれています。これは、優先順位が同じ演算子を複数使っている場合の規則を表しています。“左”の場合はより左に書かれている方が、“右”の場合はより右に書かれている方が優先されます。左がより優先されることを**左結合**、右がより優先されることを**右結合**といいます。優先順位を変えたい場合は、()を使います。式を(と)で囲むと、優先して評価されます。

これまでに説明した演算子を使った例を示します。i と j は、int 型の変数とします。

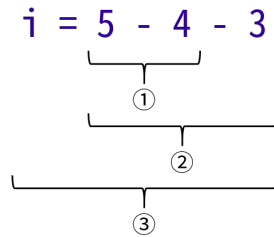
例 1: `i = 5 + 4 * 3`

演算子として、=、+、* の3つが使われています。優先順位の高い順に、*、+、= です。まず、`4 * 3` が評価され、int 型の12が得られます。次に、`5 + 12` が評価され、int 型の17が得られます。最後に、`i = 17` が評価され、変数 i に17が代入されます。



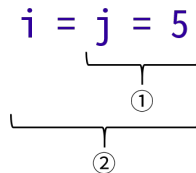
例 2: `i = 5 - 4 - 3`

演算子として、= と、二項の - が使われています。 - は = より高い優先順位を持っています。二項の - は2つ使われていますが、左結合なので、より左にある - が優先されます。つまり、まず `5 - 4` が評価され、int 型の1が得られます。次に、`1 - 3` が評価され、int 型の-2が得られます。最後に、`i = -2` が評価され、変数 i に-2が代入されます。



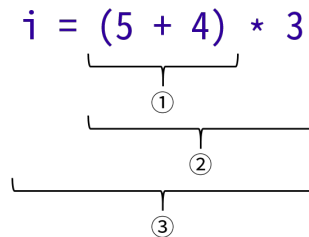
例 3: `i = j = 5`

演算子として、`=` が 2 つ使われています。`=` は右結合なので、より右にある `=` が優先されます。そのため、まず、`j = 5` が評価され、変数 `j` に 5 が代入されます。同時に、`j = 5` からは、`int` 型の 5 が得られます*¹。次に、`i = 5` が評価され、変数 `i` に 5 が代入されます。つまり、変数 `i` にも `j` にも 5 が代入されることになります。



例 4: `i = (5 + 4) * 3`

優先順位の高い順に、`*`、`+`、`=` の 3 つの演算子が使われています。ただし、`()` が使われているので、まず、その中が評価されます。5 + 4 が評価され `int` 型の 9 が得られ、次に 9 * 3 が評価され `int` 型の 27 が得られます。最後に `i = 27` が評価され、変数 `i` に 27 が代入されます。



なお、数学では、 $9 - [8 \times \{(7 + 6) - (5 + 4)\} - 3]$ のように、複数種類の括弧を使い分けることがあります。C 言語では `()` だけしか使えません。この数式の計算結果を変数 `i` に代入する場合は、`i = 9 - (8 * ((7 + 6) - (5 + 4)) - 3)` と記述します。

何らかの計算をした結果を変数に代入することを考えると、`=` の優先順位が低いことは理にかなっています。`=` の優先順位が高かったとすると `i = 3 + 4` ではなく `i = (3 + 4)` のように記述しなければなりません。また、例 3 を見ると、`=` が右結合であることも、理にかなっているこ

*¹ 代入演算を評価すると、代入した値が得られることを思い出してください。例 1 と例 2 でも最後の代入演算を評価すると、それぞれ `int` 型の 17 と -2 が得られます。ただし、その値は使っていません。

とがわかると思います。

では、左結合である二項の `-` が、もし右結合だったとすると、例 2 では変数 `i` に何が代入されるか考えてみてください。正解はこの文の脚注に記載します*2。

優先順位を変える `()` は、使う必要がない場所で使っても問題ありません。`()` を使った方が読みやすくなる場合は、そのようにしてください。

7.4 型とキャスト

ソースコード 5.5(ex07.c) を改変し、平均点を出力するようにしたプログラムを、ソースコード 7.1 に示します。

ソースコード 7.1 平均点の出力 (ex15.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int koku = 75; /* 国語の得点 */
6      int san  = 93; /* 算数の得点 */
7      int sya  = 68; /* 社会の得点 */
8
9      printf("国語は%d点、算数は%d点、社会は%d点です。\\n", koku, san, sya);
10     printf("平均点は%d点です。\\n", (koku + san + sya) / 3);
11
12     return 0;
13 }
```

実行結果

国語は 75 点、算数は 93 点、社会は 68 点です。
平均点は 78 点です。

10 行目の `(koku + san + sya) / 3` で、3 科目の平均点を計算しています。第 6 章で説明したとおり、C 言語では、整数を整数で割ると、整数の商が得られます。つまり、 $\frac{75 + 93 + 68}{3} = 78.6666\dots$ ですが、このプログラムでは、割り算の結果は整数の 78 になります。では、平均点を小数で求めたい場合はどうすればよいでしょうか。

1 つ目の方法は、割られる数と割る数のどちらかでも小数の場合、結果も小数になるという性質

*2 正解は 4 です。

を使います。次のように、割る数を小数にして、 $(\text{koku} + \text{san} + \text{sya}) / 3.0$ とすれば、平均点を小数で求めることができます。

ソースコード 7.2 平均点を小数で出力 (方法 1)

```
10 printf("平均点は%.1f点です.\n", (koku + san + sya) / 3.0);
```

実行結果 (一部)

平均点は 78.7 点です。

3.0 で割るというのは、若干不自然に感じるかもしれません。また、割る数 (3) が `int` 型の変数に入っていると、この方法は使えません。

2 つ目の方法は、**キャスト演算子**を使います。キャスト演算子は、表 7.1 の優先順位が 3 の位置にある `()` で、式の型を強制的に変更する役割を持っています*3。書式は

(型名) 式

です。型を強制的に変更することを、**キャストする**といいます。たとえば、`int` 型の定数 5 を `double` 型にキャストするには、`(double)5` と記述します。これは、5.0 と書くこととほぼ同じです。`double` 型の定数 1.5 を `int` 型にキャストするには、`(int)1.5` と記述します。この場合、小数部が切り捨てられて、1 になります。

次のように、割られる数を `double` 型にキャストすれば、平均点を小数で求めることができます。

ソースコード 7.3 平均点を小数で出力 (方法 2)

```
10 printf("平均点は%.1f点です.\n", (double)(koku + san + sya) / 3);
```

実行結果 (一部)

平均点は 78.7 点です。

次の例は、平均点を小数で出力しようとして、うまくいかない例です。

ソースコード 7.4 誤り例

```
10 printf("平均点は%.1f点です.\n", (double)((koku + san + sya) / 3));
```

実行結果 (一部)

平均点は 78.0 点です。

*3 この章では、3 種類の `()` が登場しました。関数呼び出しの演算子、式の評価の優先順位を変える記号、そしてキャスト演算子です。

この例では、`(koku + san + sya) / 3` を `double` 型にキャストしています。`(koku + san + sya) / 3` は整数同士の除算なので、結果は `int` 型の 78 です。それを `double` 型にキャストしても、小数の 78.0 になるだけで、78.6666…にはなりません。

C 言語には、この授業で使っている `char` 型、`int` 型、`double` 型以外にもたくさんの型が用意されています。どのような型同士でも正しくキャストできるわけではありません。この授業では、`int` 型の整数を小数にするために `double` 型にキャストする場合と、`double` 型の小数を切り捨てて整数にするために `int` 型にキャストする場合のみ、キャスト演算を行うことにします。

ソースコード 7.1 では、平均点を、小数点以下を切り捨てて整数で出力しました。では、四捨五入したい場合はどうすればよいのでしょうか。次のように、`printf()` 関数の第 1 引数で使う変換指定として、`%.0f` を使う方法があります。

ソースコード 7.5 変換指定 `%.0f` による四捨五入

```
10 printf("平均点は%.0f点です.\n", (double)(koku + san + sya) / 3);
```

実行結果 (一部)

平均点は 79 点です。

この方法は、表示の際に四捨五入しているだけで、四捨五入した値を `int` 型の変数に代入することなどはできません。練習課題 6 では、四捨五入した整数を生成します。

練習課題 6 (ex16.c)

ソースコード 7.1(ex15.c) を改変して、小数第 1 位で四捨五入した整数で、平均点を出力するプログラムを作ってください。ただし、`printf()` 関数の第 1 引数で使う変換指定は、`%d` を使ってください。数に 0.5 を足して、切り捨てると、小数第 1 位で四捨五入した整数になります。

実行結果

国語は 75 点、算数は 93 点、社会は 68 点です。
平均点は 79 点です。

練習課題 7 (ex17.c)

身長と体重を入力すると、BMI(Body Mass Index) を出力するプログラムを作ってください。身長は cm で、体重は kg で入力し、どちらも int 型の変数に格納するものとします。BMI は、 $\frac{\text{体重}_{(\text{kg})}}{\text{身長}_{(\text{m})}^2}$ (“体重” ÷ “身長の 2 乗”) で計算できます。入力する身長は cm 単位ですが、BMI の計算では m を使いますので注意してください。

実行例 1

身長を入力してください (cm) > 165 ↵
体重を入力してください (kg) > 55 ↵
BMI は 20.202020 です。

実行例 2

身長を入力してください (cm) > 170 ↵
体重を入力してください (kg) > 70 ↵
BMI は 24.221453 です。

7.5 式と文

これまで曖昧に用語を使ってきましたが、C 言語では、“文” を単位に処理を記述します。

式^{*4}の最後に ; を付けると、**式文**という文になります。 `i = 5` は式で、 `i = 5;` は式文です。C 言語のソースコードには文を書きますので、変数 `i` に 5 を代入するためには、 `i = 5;` と記述します。最後の ; がないと、式が書かれていることになり、コンパイルできません。 `printf("こんにちは\n")` は、関数呼び出し式です。最後に ; を付けて、 `printf("こんにちは\n");` とすると、関数呼び出し文となります。こちらも式文です。

式文以外の文もあります。たとえば、定型部分として `main()` 関数の最後に記述してきた `return` 文も文です。

第 8 章では、`if` 文を説明します。`if` 文を書くためには、“式” と “文” を区別する必要があります。

^{*4} 本章で説明したとおり、“定数”、“変数”、“演算子と被演算子の組み合わせ” が式です。

第 8 章

分岐

8.1 順次と分岐

これまでのプログラムは、上から順に処理が進みました。このようなプログラムの構造を**順次**と呼びます。順次では、記述した文は必ず実行されます。それに対して、何らかの条件判断をして、その結果に応じて処理を実行したりしなかったりすることができます。これを**分岐**と呼びます。順次と分岐の処理の流れを、図 8.1 に示します。

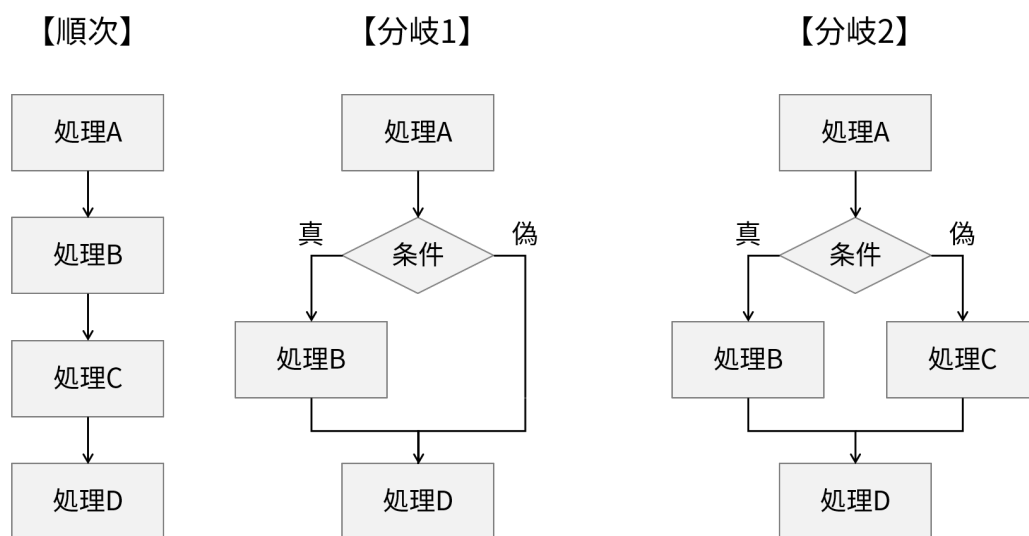


図 8.1 順次と分岐

分岐 1 では、処理 A を終えた後に条件判断を行います。条件が成立していれば、処理 B を行ってから、処理 D を行います。条件が成立していなければ、処理 B は行わずに、処理 D を行います。分岐 2 でも、処理 A を終えた後に条件判断を行います。条件が成立していれば、処理 B を行ってから、処理 D を行います。条件が成立していなければ、処理 C を行ってから、処理 D を行います。C 言語では、どちらのパターンの分岐も記述できます。

なお、プログラムの流れが分かれる部分から、1 つに戻る部分までが分岐です。図 8.1 の分岐 1

と分岐 2 における処理 A と処理 D は、分岐の一部ではありません。

8.2 真理値

条件判断をすると、成立すれば**真**、しなければ**偽**という結果が得られます。このような真か偽かを表す値を**真理値**、**真偽値**、**論理値**などといいます。この授業では“真理値”を使います。プログラミング言語によっては真理値を表す型が用意されています。たとえば Java では、`true`(真) か `false`(偽) を格納できる `boolean` 型があります。C 言語では、**0 以外を真、0 を偽**と見なします。

【発展】 C 言語のバージョン

C 言語のバージョンによっては、真理値を表す `_Bool` 型と、その別名である `bool` 型が使えます。`bool b = true;` と記述すれば、`bool` 型の変数 `b` に `true`(真) が格納されます。多くのプログラミング言語にはバージョンがあり、バージョンによって使える機能や文法が異なります。C 言語の `_Bool` 型・`bool` 型は、C99 というバージョンで定義されたものです。ここまでの説明には、バージョンによっては不正確な記述があります。たとえば、変数の宣言はブロックの先頭で行う必要があると説明しました。しかし、C99 では、変数宣言以外の文の後で変数宣言をすることもできます。また、C99 では、`main()` 関数の最後の `return 0;` は省略可能です。

プログラミング言語の新しいバージョンが策定されても、処理系が対応しなければ使うことはできません。C 言語の比較的新しい機能は、それほど使われていません。この授業では特定のバージョンを想定せず、現在一般的な書き方を紹介します。

8.3 if 文

C 言語で分岐を記述するには、**if 文**を使います。

図 8.1 の分岐 1 を実現する **if 文**の書式は、

if (制御式) 文

です。“制御式”の部分には、式を書きます。制御式を評価した値が真、つまり 0 以外なら“文”が実行されます。偽、つまり 0 なら“文”は実行されません。いくつか制御式の例を見てみます。`i` は `int` 型の変数とします。

例 1: `if (3) ...`

制御式として、定数の 3 が指定されています。3 は 0 ではありませんので、文は必ず実行されます。

例 2: `if (i) ...`

制御式として、定数 `i` が指定されています。`i` の値が 0 でなければ文が実行され、0 であれば文は実行されません。

例 3: `if (i - 3) ...`

制御式として、二項の `-` 演算子を使った式が指定されています。減算の結果が 0 でなければ、つまり `i` の値が 3 でなければ文が実行されます。`i` の値が 3 であれば、文は実行されません。

ソースコード 8.1 に、分岐 1 を実現する `if` 文の例を示します。

ソースコード 8.1 `if` 文の例 1 (ex18.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6
7      printf("正の整数を入力してください > ");
8      scanf("%d", &i);
9
10     if (i % 2)
11         printf("%dは奇数です.\n", i);
12
13     return 0;
14 }
```

10 行目と 11 行目が、1 つの `if` 文です。このように、2 行に分けて、2 行目を字下げすることが一般的です。このプログラムでは、まずキーボードから整数を 1 つ入力してもらい、変数 `i` に格納します。そして、`if` 文によって、`i % 2` が 0 でなければ、`printf("%dは奇数です.\n", i);` が実行されます。`i % 2` は、`i` が偶数であれば 0、奇数であれば 1 です。`i % 2` が 0 でない場合、つまり `i` が奇数の場合に、`printf()` 文が実行され、“`X` は奇数です。”と表示されます*1。

図 8.2 に文の構造を、図 8.3 に処理の流れを示します。

*1 `X` の部分には変数 `i` の値が入ります。

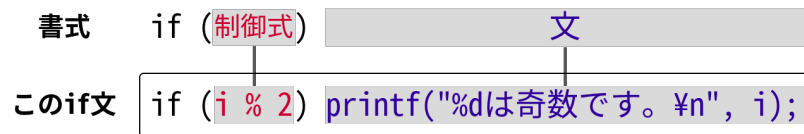


図 8.2 ソースコード 8.1 の if 文の構造

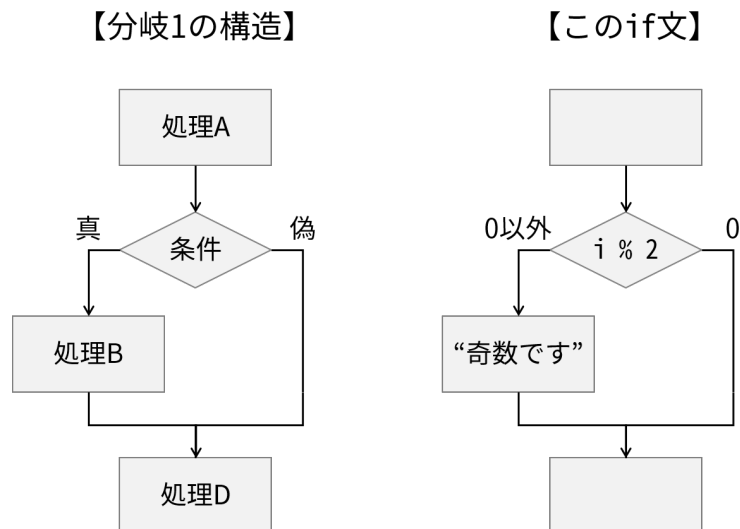


図 8.3 ソースコード 8.1 の if 文の処理の流れ

図 8.1 の分岐 2 を実現する if 文の書式は、

`if (制御式) 文1 else 文2`

です。制御式を評価した値が真、つまり 0 以外なら “文₁” が実行されます。偽、つまり 0 なら “文₂” が実行されます。制御式を評価した結果は、真か偽のどちらかなので、文₁ と 文₂ のうち片方だけが必ず実行されます。

ソースコード 8.2 に、分岐 2 を実現する if 文の例を示します。

ソースコード 8.2 if 文の例 2 (ex19.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6
7      printf("正の整数を入力してください > ");
8      scanf("%d", &i);

```

```

9
10  if (i % 2)
11      printf("%dは奇数です.\n", i);
12  else
13      printf("%dは偶数です.\n", i);
14
15  return 0;
16 }

```

ソースコード 8.1 に 12 行目と 13 行目を加えており、10～13 行目が 1 つの if 文です。i % 2 が 0 でない場合、つまり i が奇数の場合に“奇数です”と表示され、0 の場合、つまり i が偶数の場合に“偶数です”と表示されます。

図 8.4 に文の構造を、図 8.5 に処理の流れを示します。

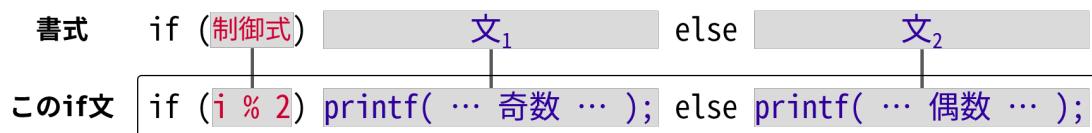


図 8.4 ソースコード 8.2 の if 文の構造

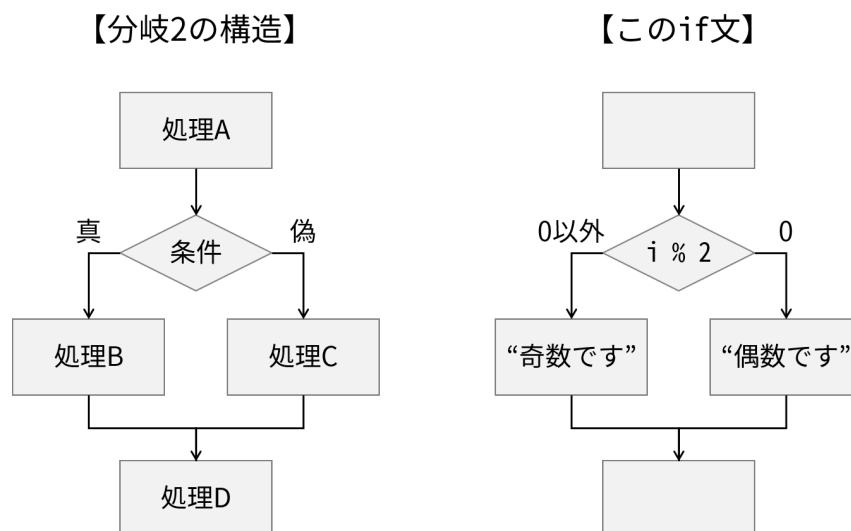


図 8.5 ソースコード 8.2 の if 文の処理の流れ

なお、文₁ と 文₂ の部分には、式ではなく文を記述します。11 行目の最後の ; が無い場合は、文ではなく式 (関数呼び出し式) を書いていることになりますので、文法エラーです。

練習課題 8 (ex20.c)

ソースコード 8.1(ex18.c) を改変して、次のようなプログラムを作ってください。

- 正の整数を入力してもらう。
- 入力された数が 5 で割り切れなければ、“〇〇は 5 で割り切れません。” と表示する。
“〇〇” の部分は、入力された数に置き換える。5 で割り切れれば、何も表示しない。

なお、5 の倍数を 5 で割った余りは 0 です。5 の倍数でない数を 5 で割った余りは 0 以外です。

実行例 1

正の整数を入力してください > 28 ↵
28 は 5 で割り切れません。

実行例 2

正の整数を入力してください > 30 ↵

練習課題 9 (ex21.c)

ソースコード 8.2(ex19.c) を改変して、次のようなプログラムを作ってください。

- 正の整数を入力してもらう。
- 入力された数が 50 以上であれば“〇〇は 50 以上です。”、そうでなければ“〇〇は 50 未満です。” と表示する。“〇〇” の部分は、入力された数に置き換える。

なお、50 以上の数を 50 で割った商は 1 以上です。50 未満の正の数を 50 で割った商は 0 です。

実行例 1

正の整数を入力してください > 45 ↵
45 は 50 未満です。

実行例 2

正の整数を入力してください > 50 ↵
50 は 50 以上です。

実行例 3

正の整数を入力してください > 60 ↵

60 は 50 以上です

【発展】 エラー処理

練習課題 9 の正解例のプログラムに負の数を入力すると、次のようになります。

実行例 a

正の整数を入力してください > -30 ↵

-30 は 50 未満です。

実行例 b

正の整数を入力してください > -60 ↵

-60 は 50 以上です。

実行例 a は正しい出力が得られましたが、実行例 b の出力は誤っています。これは、 $-30 \div 50$ が“0 余り -30”、 $-60 \div 50$ が“-1 余り -10”となったためです。 $-60 \div 50$ の商である -1 は 0 ではありませんので、“50 以上です”と表示されました。なお、第 6 章で説明したとおり、負の数の除算は処理系依存ですので、上の実行例のとおりにならないこともあります。

誤った出力になった原因は、“正の整数を入力してください”と表示しているにもかかわらず、負の数が入力されたためです。不特定のユーザが使うプログラムでは、正しく入力してくれることを期待することはできません。数字ではない文字を入力されるかもしれません。本来であれば、入力された値が適切であるか、この例であれば正の整数が入力されたかを確認してから処理を進める必要があります。また、処理を進める途中でエラーが起こるかもしれません。エラーが起こりそうな箇所では、起こっていないことを確認する必要があります。

実用的なプログラムでは、エラー処理は必須です。ただし、この授業では、ユーザの入力が正しく、エラーも起こらないことを前提として、エラー処理は省略しています。

8.4 複合文

あるスポーツでは、40kg 級、42kg 級、44kg 級、…のように、偶数の体重に対して階級が設定されているとします。体重を入力すると階級を表示する、次のようなプログラムを考えます。

- 体重を整数で入力してもらう。
- 入力された体重が奇数であれば、メッセージを表示した上で 1 を足して偶数に変更する。
- 階級を表示する。

ソースコード 8.3 にプログラム例を示します。

ソースコード 8.3 階級を表示するプログラム例 (誤り)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int weight;
6
7      printf("体重を入力してください > ");
8      scanf("%d", &weight);
9
10     /* weightが奇数であれば1を足して偶数にする */
11     if (weight % 2)
12         printf("%dは奇数です。1を足します。\\n", weight);
13         weight = weight + 1;
14
15     /* 階級を表示する */
16     printf("あなたは%dkg級です。\\n", weight);
17
18     return 0;
19 }
```

11 行目から 13 行目の if 文で、`weight` が奇数であれば、メッセージを表示した上で 1 を足して偶数にしています*2。そして 16 行目で、偶数にした `weight` を使って、階級を表示しています。実行例を 2 つ示します。

*2 `weight = weight + 1` は、右辺の `weight + 1` を評価した値を、左辺の変数 `weight` に代入する代入式です。その結果、変数 `weight` の値が 1 増えます。

実行例 1

体重を入力してください > 55 ↵
55 は奇数です。1 を足します。
あなたは 56kg 級です。

実行例 2

体重を入力してください > 62 ↵
あなたは 63kg 級です。

実行例 1 では、奇数 (55) を入力しました。メッセージを表示した上で 1 が足されて、“56kg 級” と正しく表示されました。実行例 2 では、偶数 (62) を入力しました。if 文の条件を満たさないため ($\text{weight} \% 2$ は 0)、12 行目の `printf()` 文と 13 行目の代入文は実行されないはずですが、`weight` の値は 62 のままなので “62kg 級” と表示されることが期待されますが、実際は “63kg 級” と表示されました。

これは、次のような理由によります。if (制御式) 文 の“文”、および if (制御式) 文₁ else 文₂ の“文₁”と“文₂”には、それぞれ 1 つの文しか書けません。図 8.6、8.7 に示すとおり、11 行目から始まる if 文は 12 行目で終了しており、13 行目は if 文の次の文です。つまり、13 行目の代入文は必ず実行されます。そのため、偶数を入力した場合でも 1 が足されて、奇数になってしまいました。

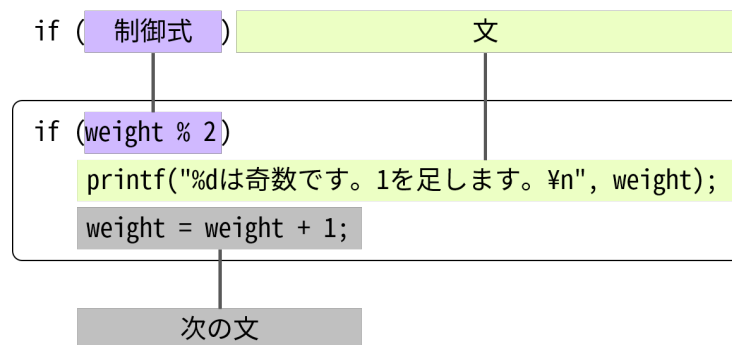


図 8.6 ソースコード 8.3 の 11 行目から 13 行目の文の構造

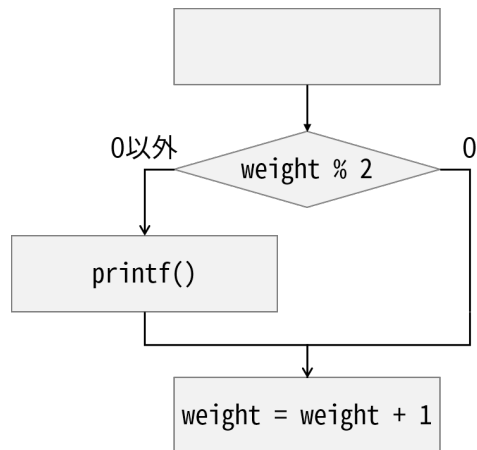


図 8.7 ソースコード 8.3 の 11 行目から 13 行目の処理の流れ

第 4 章で、**ブロック**、または**複合文**について簡単に説明しました。“{” から “}” ままでがブロック、または複合文です。本節では“ブロック”ではなく“複合文”と呼びますが、両者は同じ概念です。複合文には、変数宣言を任意の個数記述し、その後ろに変数宣言以外の文を任意の個数記述します。図 4.2 も参照してください。C 言語では、複合文は 1 つの文と見なされます。つまり、if 文の“文”として複数の文を記述したいときは、それらをまとめて複合文にしてください。

ソースコード 8.3 を正しく動作するように修正したものを、ソースコード 8.4 に示します。

ソースコード 8.4 階級を表示するプログラム例

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int weight;
6
7      printf("体重を入力してください > ");
8      scanf("%d", &weight);
9
10     /* weightが奇数であれば1を足して偶数にする */
11     if (weight % 2)
12     {
13         printf("%dは奇数です。1を足します。\\n", weight);
14         weight = weight + 1;
15     }
16
17     /* 階級を表示する */
  
```

```

18     printf("あなたは%dkg級です.\n", weight);
19
20     return 0;
21 }

```

実行例 1

体重を入力してください > 55 ↵
 55 は奇数です。1 を足します。
 あなたは 56kg 級です。

実行例 2

体重を入力してください > 62 ↵
 あなたは 62kg 級です。

12 行目から 15 行目の複合文が if 文の“文”です。変数 `weight` が奇数であればこの複合文が実行され、偶数であれば実行されません。実行例 2 のとおり、偶数を入力しても正しく動作します。

if 文で実行したい文が 1 つであれば、複合文にする必要はありません。ただし、ソースコード 8.3 のような誤りを犯しやすくなります。文が 1 つであっても、複合文にすることをおすすめします。この授業でも、これ以降そのようにします。ソースコード 8.2 を複合文を使うように変更したものを、ソースコード 8.5 に示します。

ソースコード 8.5 ソースコード 8.2 を複合文を使うように変更

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6
7      printf("正の整数を入力してください > ");
8      scanf("%d", &i);
9
10     if (i % 2)
11     {
12         printf("%dは奇数です.\n", i);
13     }
14     else
15     {

```



```
16     printf("%dは偶数です.\n", i);
17 }
18
19 return 0;
20 }
```

第 9 章

数の比較

前章では、if 文による分岐を学びました。練習課題 9 では、入力された数を 50 で割った商が 0 でないかどうかによって、50 以上であるかを判定しました。もっと直感的に、“変数 *i* の値が 50 以上であれば” のような条件を書けると便利です。また、ソースコード 8.1 では、変数 *i* の値が奇数であれば実行する if 文を紹介しました。では、偶数であれば実行する if 文はどのように書けばよいでしょうか。たとえば、ソースコード 9.1 のように記述することができます。3 行目の ; は、何もしない文で^{くうぶん}空文と呼ばれます。変数 *i* の値が偶数であれば 7 行目の printf() 文が実行され、奇数であれば何も実行されません*1。こちらも“変数 *i* の値を 2 で割った余りが 0 であれば” のような条件を書けると便利です。

ソースコード 9.1 変数 *i* の値が偶数であれば実行

```
1  if (i % 2)
2  {
3      ;
4  }
5  else
6  {
7      printf("%dは偶数です.\n", i);
8  }
```

算術演算だけでは、上の例のような苦し紛れの方法になるか、そもそも意図した条件が書けないことも多くあります。本章と次章では、if 文の制御式によく使われる演算を紹介します。

*1 空文が実行されます。

9.1 数の比較

日常生活においても、“Aさんの年齢とBさんの年齢が同じであれば”や“身長が170cm以上であれば”のように、数同士を比較した結果をもとに処理を行うことがあります。C言語では、if文の制御式として、数同士の比較を表9.1のように記述できます。aとbの部分には式、つまり“定数”、“変数”、“演算子と被演算子の組み合わせ”を記述します。

表 9.1 if 文の制御式における数同士の比較

制御式	意味
if (a == b) 文	a と b が等しければ ($a = b$ であれば) “文” を実行する
if (a != b) 文	a と b が等しくなければ ($a \neq b$ であれば) “文” を実行する
if (a < b) 文	a が b 未満であれば ($a < b$ であれば) “文” を実行する
if (a <= b) 文	a が b 以下であれば ($a \leq b$ であれば) “文” を実行する
if (a > b) 文	a が b より大きければ ($a > b$ であれば) “文” を実行する
if (a >= b) 文	a が b 以上であれば ($a \geq b$ であれば) “文” を実行する

a と b が等しいかを判定するためには $a == b$ を、等しくないかを判定するためには $a != b$ を使います。a が b 未満、a が b 以下、a が b より大きい、a が b 以上を判定するためには、それぞれ $a < b$ 、 $a <= b$ 、 $a > b$ 、 $a >= b$ を使います。いずれも数学の等号、不等号と似ていますが、多少違います。特に、 $==$ と $=$ は間違えやすいので注意してください。a = b と書くと、変数 a に b の値を代入する式になってしまいます。

では、表 9.1 の方法を使って、練習課題 9 の“50 以上であれば”を書き直します。

ソースコード 9.2 練習課題 9 の書き直し (一部)

```
1  if (i >= 50)
2  {
3      printf("%dは50以上です.\n", i);
4  }
5  else
6  {
7      printf("%dは50未満です.\n", i);
8  }
```

$a >= b$ を使い、変数 i が 50 以上かどうかによって、処理を分けています。このような記述にすれば、図 9.1 のように直感的に読むことができます。

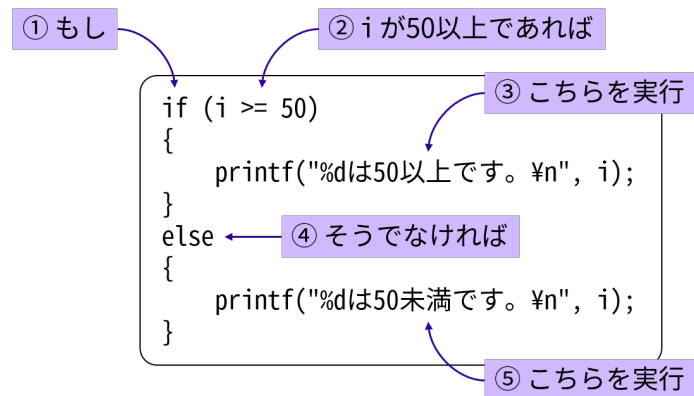


図 9.1 ソースコード 9.2 の読み方

次に、ソースコード 9.1 を書き直します。

ソースコード 9.3 ソースコード 9.1 の書き直し

```

1  if (i % 2 == 0)
2  {
3      printf("%dは偶数です。\\n", i);
4  }

```

ここでは、`a == b` を使いました。a の部分が `i % 2` という式になっています。こちらも、“もし (if)、i を 2 で割った余りが 0 と等しければ 3 行目を実行” と直感的に読むことができます。

練習課題 10 (ex22.c)

年齢を入力してもらい、それが 20 以上であれば“あなたは成人です。”、そうでなければ“あなたは未成年です。”と表示するプログラムを作ってください。

実行例 1

年齢を入力してください > 17 ↵
あなたは未成年です。

実行例 2

年齢を入力してください > 35 ↵
あなたは成人です。

実行例 3

年齢を入力してください > 20 ↵
あなたは成人です。

練習課題 11 (ex23.c)

あるスーパーでは、一の位が 5 の日、つまり 5 日と 15 日と 25 日が特売日です。日を入力すると特売日か否かを表示するプログラムを作ってください。

実行例 1

日を入力してください (1-31) > 13 ↵
13 日は特売日ではありません。

実行例 2

日を入力してください (1-31) > 25 ↵
25 日は特売日です。

9.2 数の比較のための演算子

前節では、if 文の制御式で数同士を比較する方法を紹介しました。しかしそれ以前では、if 文は、制御式の真偽、つまり 0 以外か 0 かに応じて、文を実行したりしなかったりするという説明をしました。数同士の比較を行う制御式でも、この規則は同じです。

$a == b$ などの式は、成立すれば 1、成立しなければ 0 となります。制御式が $a == b$ である if 文 `if (a == b) 文` は、 $a == b$ が成立すれば制御式が 0 以外 (1) なので、文が実行されます。成立しなければ 0 なので、文は実行されません。

$a == b$ の $==$ は演算子、 a と b は被演算子です。式 $a == b$ を評価した値は、 a の値と b の値が等しければ int 型の 1、等しくなければ int 型の 0 です。表 9.1 で使った演算子の正確な説明を、表 9.2 に示します。いずれも二項演算子で、等号や不等号が成立すれば 1、つまり真になり、しなければ 0、つまり偽になります。

$==$ や $<$ は演算子なので、第 7 章での演算子、評価、式などの説明はそのまま当てはまります。ソースコード 9.3 で使った if 文の制御式は $i \% 2 == 0$ でした。この式には、演算子 $\%$ と $==$ が使われています。表 7.1 を見ると、 $==$ よりも $\%$ の方が高い優先順位を持っています。つまり、この式はまず $i \% 2$ が評価され、 i を 2 で割った余りが得られます。次に $==$ が評価され、余りと 0

表 9.2 数の比較のための演算子

演算子	役割
<code>a == b</code>	a と b が等しければ 1、そうでなければ 0
<code>a != b</code>	a と b が等しくなければ 1、そうでなければ 0
<code>a < b</code>	a が b 未満であれば 1、そうでなければ 0
<code>a <= b</code>	a が b 以下であれば 1、そうでなければ 0
<code>a > b</code>	a が b より大きければ 1、そうでなければ 0
<code>a >= b</code>	a が b 以上であれば 1、そうでなければ 0

が等しければ 1、等しくなければ 0 が得られます。

では、表 9.2 の演算子を使った式が、1 か 0 を返すことを確認してみましょう。ソースコード 9.4 をコンパイルして実行してください。授業のサイトには、`ex24.c` として掲載しています。

ソースコード 9.4 数の比較のための演算子の確認用プログラム (`ex24.c`)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("5 == 7 は %d\n", 5 == 7);
6      printf("5 != 7 は %d\n", 5 != 7);
7      printf("5 < 7 は %d\n", 5 < 7);
8      printf("5 <= 7 は %d\n", 5 <= 7);
9      printf("5 > 5 は %d\n", 5 > 5);
10     printf("5 >= 5 は %d\n", 5 >= 5);
11
12     return 0;
13 }
```

実行結果

```

5 == 7 は 0
5 != 7 は 1
5 < 7 は 1
5 <= 7 は 1
5 > 5 は 0
5 >= 5 は 1
```

このプログラムは、式 $5 == 7$ 、 $5 != 7$ 、 $5 < 7$ 、 $5 \leq 7$ 、 $5 > 5$ 、 $5 \geq 5$ を評価した結果をそのまま表示します。等号や不等号が成立すれば 1、しなければ 0 になっていることを確認してください。

第 10 章

論理演算

前章で、分岐の条件として、数同士の比較が書けるようになりました。本章では、“A かつ B であれば”や“A または B であれば”など、複数の条件を組み合わせる方法を学びます。

10.1 論理演算

C 言語での記述方法に進む前に、真理値を対象とした演算である**論理演算**について説明します。

“A かつ B”が成り立つとは、A が成り立ち、かつ B も成り立つことです。“A かつ B”は、A と B の両方が真であれば真になり、片方でも偽であれば偽になります。このような演算を**論理積**といいます (表 10.1)。

表 10.1 論理積

A	B		A かつ B
真	真	\implies	真
真	偽	\implies	偽
偽	真	\implies	偽
偽	偽	\implies	偽

“A または B” が成り立つとは、A と B のどちらか片方でも成り立つことです。“A または B” は、A と B の片方でも真であれば真になり、両方が偽であれば偽になります。このような演算を**論理和**といいます (表 10.2)。

表 10.2 論理和

A	B		A または B
真	真	\implies	真
真	偽	\implies	真
偽	真	\implies	真
偽	偽	\implies	偽

“A でない” が成り立つとは、A が成り立たないことです。“A でない” は、A が真であれば偽、A が偽であれば真になります。このような、真理値を反転させる演算を**論理否定**、または単に**否定**といいます (表 10.3)。

表 10.3 論理否定

A		A でない
真	\implies	偽
偽	\implies	真

具体的な例を考えます。

- “年齢が 28 歳から 32 歳までである” は、“年齢が 28 歳以上である” と “年齢が 32 歳以下である” の 2 つの条件の論理積です。
- “身長が 120cm 未満か年齢が 6 歳以下である” は、“身長が 120cm 未満である” と “年齢が 6 歳以下である” の 2 つの条件の論理和です。
- “年齢が 20 歳未満でない” は、“年齢が 20 歳未満である” という条件の論理否定です。

これらの例を図で表したものを、それぞれ図 10.1、10.2、10.3 に示します*¹。なお、図 10.1 において、2 つの円のどちらにも入らない年齢はありません。28 歳以上 (条件 A) でもなく、かつ 32 歳以下 (条件 B) でもない年齢は存在しないからです。

*¹ このような図をベン図といいます。

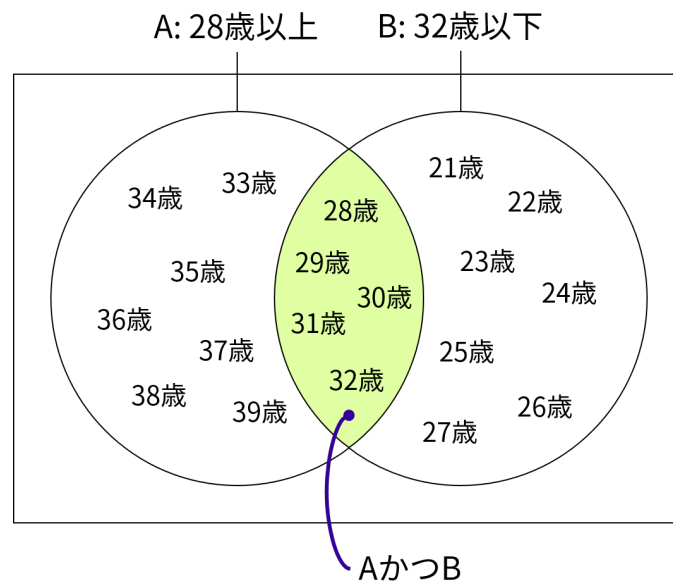


図 10.1 “年齢が 28 歳から 32 歳まで” の図式

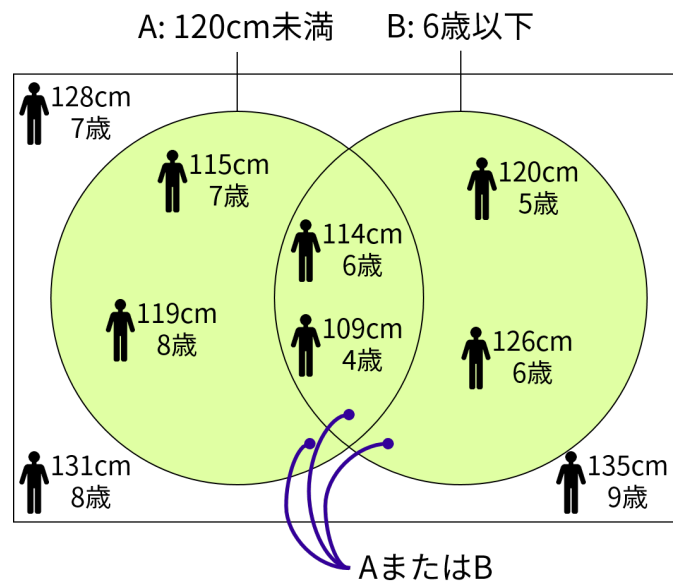


図 10.2 “身長が 120cm 未満か年齢が 6 歳以下” の図式

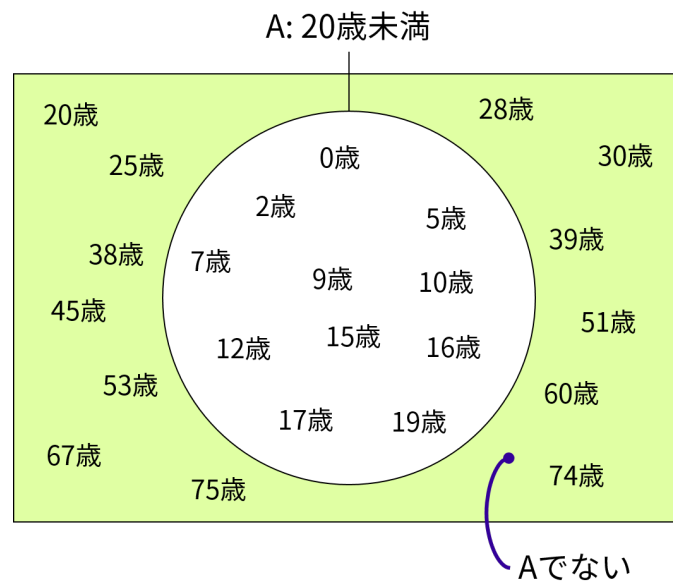


図 10.3 “年齢が 20 歳未満でない” の図式

10.2 論理演算のための演算子

C 言語では、論理積の演算子として `&&`、論理和の演算子として `||`、論理否定の演算子として `!` が用意されています。表 10.4 に、これらの演算子の役割を示します。`&&` と `||` は二項演算子、`!` は単項演算子です。

表 10.4 論理演算のための演算子

演算子	役割
<code>a && b</code>	<code>a</code> と <code>b</code> の両方とも 0 以外であれば 1(真)、そうでなければ 0(偽)
<code>a b</code>	<code>a</code> と <code>b</code> の片方でも 0 以外であれば 1(真)、そうでなければ 0(偽)
<code>!a</code>	<code>a</code> が 0 であれば 1(真)、そうでなければ 0(偽)

ソースコード 10.1 は、`&&` の使用例です。

ソースコード 10.1 `&&` の使用例 (ex25.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int age;
6

```

```

7   printf("年齢を入力してください > ");
8   scanf("%d", &age);
9
10  if (age >= 28 && age <= 32)
11  {
12      printf("あなたは30歳前後です.\n");
13  }
14
15  return 0;
16 }

```

実行例 1

年齢を入力してください > 20 ↵

実行例 2

年齢を入力してください > 31 ↵

あなたは 30 歳前後です。

入力した年齢が 28 歳から 32 歳までであれば“あなたは 30 歳前後です。”と表示します。10 行目の if 文の制御式で && を使っています。&& の被演算子である `age >= 28` は、数の比較のための演算子 `>=` を使った式で、評価すると 0 か 1 になります。`age <= 32` も同様です。`age` の値に対して、各式を評価した値を表 10.5 に示します。

表 10.5 各式を評価した値 (ソースコード 10.1 の if 文の制御式)

age	<code>age >= 28</code>	<code>age <= 32</code>	<code>age >= 28 && age <= 32</code>
20	0	1	0
31	1	1	1
35	1	0	0

`age` が 20 の場合、`age >= 28` は 0、`age <= 32` は 1、`age >= 28 && age <= 32` は `0 && 1` で 0 です。`age` が 31 の場合、`age >= 28` は 1、`age <= 32` は 1、`age >= 28 && age <= 32` は `1 && 1` で 1 です。`age >= 28` と `age <= 32` の両方が 1 の場合、つまり `age` が 28 以上で、かつ 32 以下の場合、`age >= 28 && age <= 32` が 1 となり、12 行目の `printf()` 文が実行されます。

なお、`>=` と `<=` の優先順位は `&&` よりも高いため、

```
age >= 28 && age <= 32
```

に `()` は必要ありません。ただし、

`(age >= 28) && (age <= 32)`

と書いた方がわかりやすいと感じる場合は、そのようにしてください。

なお、`age >= 28 && age <= 32` を `28 <= age <= 32` と書くことはできません。式 `28 <= age <= 32` には、演算子 `<=` が 2 つ使われています。`<=` は左結合なので、まず `28 <= age` が評価され、0 か 1 が得られます。次に `0 <= 32` か `1 <= 32` が評価されます。つまり、`28 <= age <= 32` は、`age` の値にかかわらず必ず 1 になります。

ソースコード 10.2 は、`||` の使用例です。

ソースコード 10.2 `||` の使用例 (ex26.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int height;
6      int age;
7
8      printf("身長を入力してください > ");
9      scanf("%d", &height);
10     printf("年齢を入力してください > ");
11     scanf("%d", &age);
12
13     if (height < 120 || age <= 6)
14     {
15         printf("あなたはこのジェットコースターに乗れません.\n");
16     }
17
18     return 0;
19 }
```

実行例 1

身長を入力してください > 109 ↵

年齢を入力してください > 4 ↵

あなたはこのジェットコースターに乗れません。

実行例 2

身長を入力してください > 126 ↵
年齢を入力してください > 6 ↵
あなたはこのジェットコースターに乗れません。

実行例 3

身長を入力してください > 131 ↵
年齢を入力してください > 8 ↵

入力した身長が 120cm 未満か年齢が 6 歳以下であれば“あなたはこのジェットコースターに乗れません。”と表示します。height と age の値に対して、各式を評価した値を表 10.6 に示します。

表 10.6 各式を評価した値 (ソースコード 10.2 の if 文の制御式)

height	age	height < 120	age <= 6	height < 120 age <= 6
109	4	1	1	1
115	7	1	0	1
126	6	0	1	1
131	8	0	0	0

height < 120 と age <= 6 の片方でも 1 の場合、つまり height が 120 未満か age が 6 以下の場合、height < 120 || age <= 6 が 1 となり、15 行目の printf() 文が実行されます。

ソースコード 10.3 は、! の使用例です。

ソースコード 10.3 ! の使用例 (ex27.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int age;
6
7      printf("年齢を入力してください > ");
8      scanf("%d", &age);
9
10     if (!(age < 20))
11     {
12         printf("あなたは飲酒できます.\n");
13     }
```

```

14
15     return 0;
16 }

```

実行例 1

年齢を入力してください > 18 ↵

実行例 2

年齢を入力してください > 20 ↵
あなたは飲酒できます。

入力した年齢が 20 歳未満でなければ、“あなたは飲酒できます。”と表示します。age の値に対して、各式を評価した値を表 10.7 に示します。

表 10.7 各式を評価した値 (ソースコード 10.3 の if 文の制御式)

age	age < 20	!(age < 20)
18	1	0
20	0	1

age < 20 が 0 の場合、つまり age が 20 未満でない場合、!(age < 20) が 1 となり、12 行目の printf() 文が実行されます。なお、< の優先順位は ! より低いため、!(age < 20) には () が必要です。!age < 20 と書くと、まず !age が評価されるため、意図どおりの処理にはなりません*2。

if 文の制御式を真と偽だけで議論できればわかりやすいのですが、C 言語では 0 以外か 0 かで考える必要があります。そのため、前章と本章で説明した演算子は難しく感じるかもしれません。しかし最初に覚えてしまえば、実際に演算子を使う段階で 0 以外か 0 かを意識することはあまりありません。次のように、直感的に読むことができます。

```
if (age >= 28 && age <= 32) ...
```

もし (if)、age が 28 以上 (age >= 28)、かつ (&&)、age が 32 以下 (age <= 32) であれば、...

```
if (height < 120 || age <= 6) ...
```

もし (if)、height が 120 未満 (height < 120)、または (||)、age が 6 以下 (age <= 6) であれば、...

*2 !age は 0 か 1 なので、!age < 20 は 0 < 20 か 1 < 20 であり、必ず 1 になります。

```
if (!(age < 20)) ...
```

もし (if)、age が 20 未満 (age < 20) でなければ (!)、…

練習課題 12 (ex28.c)

円周率を小数で答えてもらい、それが 3.1 より大きく 3.15 未満であれば“正解です。”そうでなければ“不正解です。”と表示するプログラムを作ってください。

実行例 1

円周率はいくつですか > 3 ↵
不正解です。

実行例 2

円周率はいくつですか > 3.142 ↵
正解です。

実行例 3

円周率はいくつですか > 3.2 ↵
不正解です。

練習課題 13 (ex29.c)

次のようなプログラムを作ってください。

- 出身都道府県の頭文字を、英文字 1 字で入力してもらう。
- それが“e”か“E”であれば“あなたは愛媛県出身ですね。”と表示する。
- “w”か“W”であれば“あなたは和歌山県出身ですね。”と表示する。

なお、char 型変数に格納された文字や、文字定数同士の比較にも、== と != が使えます。たとえば、'a' == 'A' は 0 ('a' と 'A' は等しくない)、'b' != 'c' は 1 ('b' と 'c' は等しくない) です。また、このプログラムでは、if 文を 2 つ使います。

実行例 1

出身都道府県の頭文字を教えてください > e ↵
あなたは愛媛県出身ですね。

実行例 2

出身都道府県の頭文字を教えてください > E ↩

あなたは愛媛県出身ですね。

実行例 3

出身都道府県の頭文字を教えてください > w ↩

あなたは和歌山県出身ですね。

実行例 4

出身都道府県の頭文字を教えてください > T ↩

第 11 章

分岐の応用

分岐や論理演算を学んだことで、記述できるプログラムの範囲が広がりました。本章では、プログラムを読んだり作ったりしながら、分岐や論理演算についての理解を深めます。

11.1 最大値・最小値

ソースコード 11.1 は、2 つの整数を入力すると大きい方の数と小さい方の数を表示するプログラムです。キーボードから入力した 2 つの整数を、`int` 型の変数 `a` と `b` に格納します。そして、大きい方として、`a` が `b` 以上であれば `a` を、そうでなければ `b` を表示します。小さい方も同様です。

ソースコード 11.1 2 つの数の大きい方と小さい方を表示 (ex30.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a, b;
6
7      printf("整数を2つ入力してください。\n");
8      printf("1つ目 > ");
9      scanf("%d", &a);
10     printf("2つ目 > ");
11     scanf("%d", &b);
12
13     if (a >= b)
14     {
15         printf("大きい方は%d、小さい方は%dです。\n", a, b);
16     }
```

```

17     else
18     {
19         printf("大きい方は%d、小さい方は%dです。\\n", b, a);
20     }
21
22     return 0;
23 }

```

このプログラムを次のように改変します。2つの整数を変数 `a` と `b` に格納した後、`a` が `b` 未満であれば、`a` と `b` の値を入れ替えます。すると、必ず `a` が大きい方の数、`b` が小さい方の数になりますので、大きい方として `a`、小さい方として `b` を表示します。この方針でソースコード 11.1 を改変すると、ソースコード 11.2 のようになります。

ソースコード 11.2 変数の値の入れ替え

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a, b;
6
7      printf("整数を2つ入力してください。\\n");
8      printf("1つ目 > ");
9      scanf("%d", &a);
10     printf("2つ目 > ");
11     scanf("%d", &b);
12
13     if (a < b)
14     {
15         /* a と b の値を入れ替える */
16     }
17
18     printf("大きい方は%d、小さい方は%dです。\\n", a, b);
19
20     return 0;
21 }

```

15 行目のコメントの部分に、変数 `a` と `b` の値を入れ替える処理を書けば完成です。そのためにはどうすればよいのでしょうか。 `a` を `b` の値にして、`b` を `a` の値にすればよいのですが、ソースコー

ド 11.3 のように書くことはできません。

ソースコード 11.3 変数の値の入れ替え (誤り)

```
1  if (a < b)
2  {
3      a = b;  /* ① */
4      b = a;  /* ② */
5  }
```

a が 3、b が 5 のときの変数の値の変化を、表 11.1 に示します。

表 11.1 ソースコード 11.3 の変数の値の変化

	a	b
初期値	3	5
①の直後	5	5
②の直後	5	5

①の段階で、変数 a に b の値である 5 を代入してしまい、もともと a に格納されていた 3 が消えてしまいました。変数 a の値を書き換える前に、もとの値を記録しておく必要があります。そのためには、新しい変数を使います。正しく動作するように修正したものを、ソースコード 11.4 に示します。

ソースコード 11.4 変数の値の入れ替え

```
1  if (a < b)
2  {
3      int tmp; /* ① */
4      tmp = a; /* ② */
5      a = b;   /* ③ */
6      b = tmp; /* ④ */
7  }
```

一時的に値を記録しておくための変数 tmp を使います。変数 a の値を tmp に記録 (②) してから、a に b の値を代入 (③) します。そして、b に tmp の値を代入 (④) します。2 行目から 7 行目が複合文となっており、最初に宣言、次に変数宣言以外の文となっていることも確認してください。a が 3、b が 5 のときの変数の値の変化を、表 11.2 に示します。

上のように改変したプログラムを、授業のサイトに、ex31.c として掲載します。

表 11.2 ソースコード 11.4 の変数の値の変化

	a	b	tmp
初期値	3	5	不定値* ¹
②の直後	3	5	3
③の直後	5	5	3
④の直後	5	3	3

【発展】 追加の変数を使わない値の入れ替え

次のようにすれば、一時的に値を記録しておく変数を使わずに、2 つの変数の値を入れ替えることもできます。

```

1  if (a < b)
2  {
3      a = a + b;  /* ① */
4      b = a - b;  /* ② */
5      a = a - b;  /* ③ */
6  }
```

変数 **a** の初期値を *A*、変数 **b** の初期値を *B* とすると、変数の値の変化は次の表のとおりです。

	a	b
初期値	<i>A</i>	<i>B</i>
①の直後	<i>A + B</i>	<i>B</i>
②の直後	<i>A + B</i>	<i>A</i>
③の直後	<i>B</i>	<i>A</i>

変数が 1 つ節約できますが、プログラムはわかりにくくなります。普通は、ソースコード 11.4 のように書きます。

*¹ 第 4 章で説明したように、変数を宣言しただけでは、中にどのような値が入っているかはわかりません。

では次に、3つの整数を入力するとそれらの最大値と最小値を表示するプログラムを作りましょう。ここまでに説明した範囲で作ったプログラム例を、ソースコード 11.5 に示します。

ソースコード 11.5 3つの数の最大値と最小値を表示

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a, b, c;
6
7      printf("整数を3つ入力してください。\n");
8      printf("1つ目 > ");
9      scanf("%d", &a);
10     printf("2つ目 > ");
11     scanf("%d", &b);
12     printf("3つ目 > ");
13     scanf("%d", &c);
14
15     if (a >= b && a >= c)
16     {
17         printf("最大値は%dです.\n", a);
18     }
19
20     if (b > a && b >= c)
21     {
22         printf("最大値は%dです.\n", b);
23     }
24
25     if (c > a && c > b)
26     {
27         printf("最大値は%dです.\n", c);
28     }
29
30     if (a <= b && a <= c)
31     {
32         printf("最小値は%dです.\n", a);
33     }
```

```

34
35     if (b < a && b <= c)
36     {
37         printf("最小値は%dです.\n", b);
38     }
39
40     if (c < a && c < b)
41     {
42         printf("最小値は%dです.\n", c);
43     }
44
45     return 0;
46 }

```

3つの整数を変数 a、b、c に格納した後、それらの大小を比較します。a が b 以上、かつ a が c 以上であれば、a が最大値なので、最大値として a を表示します。次は、b が a 以上、かつ b が c 以上であれば、b を最大値として表示すればよさそうですが、a と b が等しく、かつ a が c 以上のとき、最大値が 2 回表示されてしまいます*2。

ソースコード 11.6 誤り例 (最大値が 2 回表示)

```

1  if (a >= b && a >= c)
2  {
3      printf("最大値は%dです.\n", a);
4  }
5
6  if (b >= a && b >= c)
7  {
8      printf("最大値は%dです.\n", b);
9  }

```

そこで、値が同じ場合、c より b、b より a を優先するようにしています。最小値も同じです。しかし、このプログラムは理解しにくく、記述ミスも起こりやすそうです。そこで、練習課題 14 でこのプログラムを改良します。

2 else を使って if (a >= b && a >= c) { / a が最大値 */ } else if (b >= a && b >= c) { /* b が最大値 */ } else { /* c が最大値 */ } なら正しく動作します。ソースコード 11.5 も、本当は else を使うことが適切です。

練習課題 14 (ex32.c)

次のような方針で、3つの整数の最大値と最小値を表示するプログラムを作ってください。

1. 3つの整数の最大値を格納する変数 `max` と、最小値を格納する変数 `min` を用意する。
2. 3つの整数を、変数 `a`、`b`、`c` に格納する。
3. とりあえず、`a` の値を `max` に代入する。
4. `b` の値が `max` より大きければ、`b` の値を `max` に代入する。
5. `c` の値が `max` より大きければ、`c` の値を `max` に代入する。
6. この時点で、`max` には、`a`、`b`、`c` のうちの最大値が入っている。
7. `min` も同様に求める。
8. 最大値として `max`、最小値として `min` を表示する。

途中まで作ったプログラムを、授業のサイトに `ex32.c` として掲載しています。コメント部分に処理を追加して、正しく動作するプログラムを作ってください。

実行例

整数を 3 つ入力してください。

1 つ目 > 14 ↵

2 つ目 > 8 ↵

3 つ目 > 19 ↵

最大値: 19

最小値: 8

11.2 3つ以上への分岐

条件が成り立つか否かによって、プログラムの流れを2つに分岐することができました。分岐先でさらに分岐すると、3つ以上に分岐することができます。図 11.1 に、4 分岐の例を2つ示します。

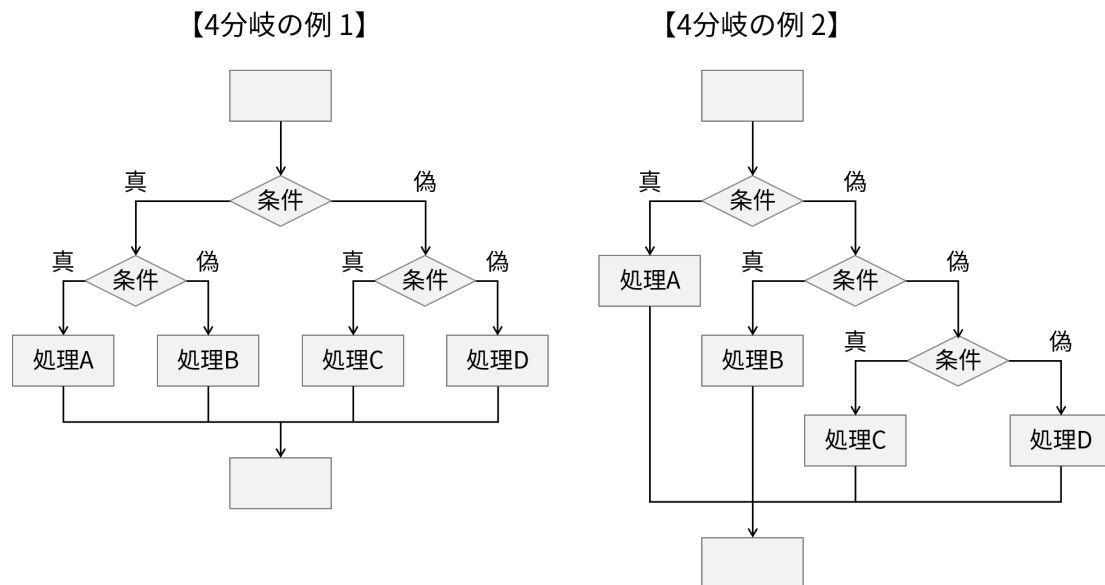


図 11.1 4 分岐の例

例として、3つに分岐するプログラムを考えます。経験年数を入力してもらい、3年未満であれば“新人”、3年以上8年未満であれば“中堅”、8年以上であれば“ベテラン”と表示するプログラムをソースコード 11.7 に示します。

ソースコード 11.7 3 分岐の例

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int y;
6
7      printf("経験年数を教えてください > ");
8      scanf("%d", &y);
9
10     if (y < 3)
11     {
```

```

12     printf("あなたは新人です.\n");
13 }
14 else
15 {
16     if (y < 8)
17     {
18         printf("あなたは中堅です.\n");
19     }
20     else
21     {
22         printf("あなたはベテランです.\n");
23     }
24 }
25
26 return 0;
27 }

```

実行例 1

経験年数を教えてください > 2 ↵
あなたは新人です。

実行例 2

経験年数を教えてください > 7 ↵
あなたは中堅です。

実行例 3

経験年数を教えてください > 8 ↵
あなたはベテランです。

この分岐の構造を、図 11.2 に示します。

外側の `if (制御式) 文1 else 文2` の“文₂”が複合文になっており、別の `if` 文が入っています。この例のように、“文₂”が1つの `if` 文の場合、`{ }` を使って複合文にしないことが普通です。また、`else` の直後の改行と、字下げは行いません。一般的な記述方法に修正したものを、ソースコード 11.8 に示します。

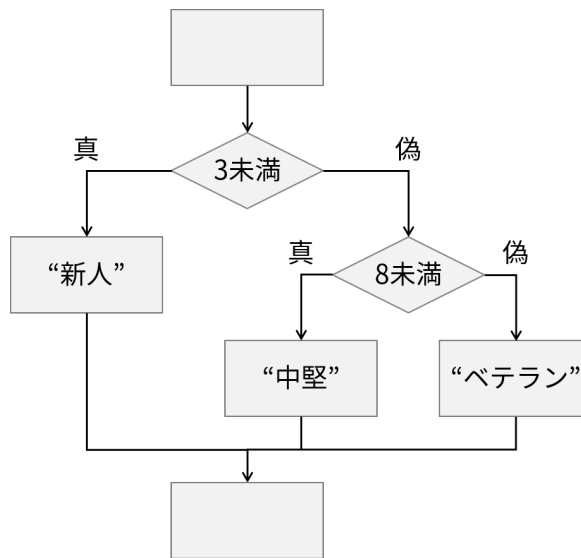


図 11.2 ソースコード 11.7 の分岐の構造

ソースコード 11.8 ソースコード 11.7 の if 文の一般的な記述方法

```

1  if (y < 3)
2  {
3      printf("あなたは新人です.\n");
4  }
5  else if (y < 8)
6  {
7      printf("あなたは中堅です.\n");
8  }
9  else
10 {
11     printf("あなたはベテランです.\n");
12 }
  
```

このように書くと、“もし (if)、y が 3 未満なら、…、そうではなく (else)、もし (if)、y が 8 未満なら、…、そうでなければ (else)、…”と直感的に読むことができます。ソースコード 11.8 のように書いたとしても、構造としてはソースコード 11.7 のとおりであることは理解しておいてください。

練習課題 15 (ex33.c)

次のようなプログラムを作ってください。

1. 0～100 点の得点を整数で入力してもらおう。
2. 90 点以上であれば“S”、80～89 点であれば“A”、70～79 点であれば“B”、60～69 点であれば“C”、60 点未満であれば“D”と表示する。

途中まで作ったプログラムを、授業のサイトに ex33.c として掲載しています。コメント部分に処理を追加して、正しく動作するプログラムを作ってください。

実行例 1

得点を入力してください (0-100) > 80 ↵
あなたの成績は A です。

実行例 2

得点を入力してください (0-100) > 61 ↵
あなたの成績は C です。

11.3 論理演算の応用

“a と b が等しい”の否定は、“a と b が等しくない”です。つまり、ソースコード 11.9 と 11.10 の if 文の制御式は、同じ役割です。

ソースコード 11.9 “等しい”の否定

```
if (!(a == b)) ...
```

ソースコード 11.10 “等しくない”

```
if (a != b) ...
```

同様に、“a が b 未満”の否定は、“a は b 以上”です。ソースコード 11.11 と 11.12 の if 文の制御式は、同じ役割です。

ソースコード 11.11 “未満”の否定

```
if (!(a < b)) ...
```

```
if (a >= b) ...
```

表 11.3 に否定の関係にある演算子の組を示します。論理否定の演算子 `!` を使えば、`==` と `!=` や、`<` と `>=` はどちらか片方だけあればよいことになります。

表 11.3 否定の関係にある演算子の組

<code>==</code>	と	<code>!=</code>
<code><</code>	と	<code>>=</code>
<code>></code>	と	<code><=</code>

【発展】 論理否定の演算子 `!`

`==` と `!` を使えば、`!=` と同じ役割の式を記述できるため、`!=` は冗長であるとも考えられます。`!` も同様です。つまり、これまでに説明した別の演算子を使って `!a` と同じ役割の式を書くことができます。`!a` を評価すると、`a` が 0 以外のとき 0、0 のとき 1 になります。これと同じ役割で、`!` を使わない式を考えてみてください。

⋮

正解は、`a == 0` です。`a == 0` を評価すると、`a` が 0 以外のとき 0、0 のとき 1 になります。つまり、`==` があれば、`!` は必要ありません。

“年齢が 28 歳以上、かつ年齢が 32 歳以下” の否定は、“年齢が 28 歳以上でない、または年齢が 32 歳以下でない” です。一般化すると、“A かつ B” の否定は、“A の否定、または B の否定” です (図 11.3)。“身長が 120cm 未満、または年齢が 6 歳以下” の否定は、“身長が 120cm 未満でない、かつ年齢が 6 歳以下でない” です。一般化すると、“A または B” の否定は、“A の否定、かつ B の否定” です (図 11.4)。これらを、**ド・モルガンの法則**といいます。

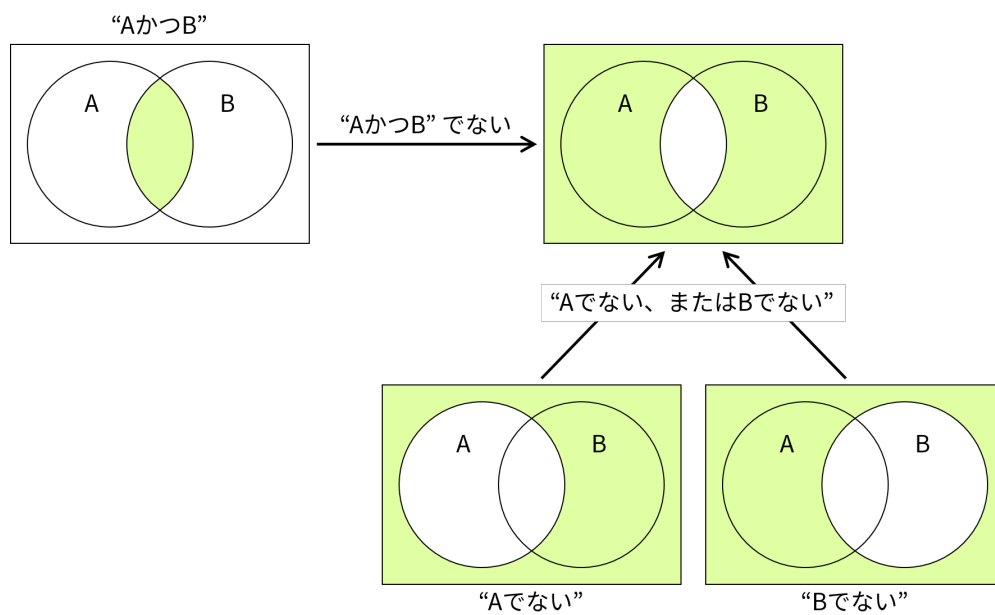


図 11.3 ド・モルガンの法則 1

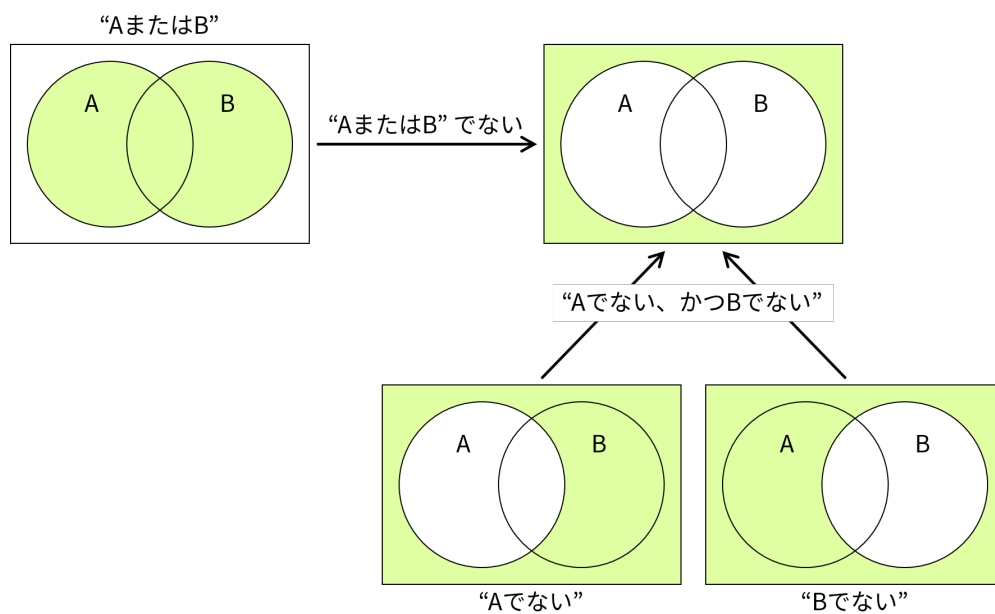


図 11.4 ド・モルガンの法則 2

ソースコード 10.2 の 13～16 行目を、ソースコード 11.13 に再掲します。

ソースコード 11.13 ソースコード 10.2 の 13～16 行目

```
1  if (height < 120 || age <= 6)
2  {
3      printf("あなたはこのジェットコースターに乗れません。\\n");
4  }
```

これは、身長が 120cm 未満、または年齢が 6 歳以下であれば“あなたはこのジェットコースターに乗れません。”と表示するプログラムでした。if 文の条件を反対にして、“あなたはこのジェットコースターに乗れます。”と表示するプログラムに改変するには、次のような方法があります。

ソースコード 11.14 条件の反転 1

```
1  if (!(height < 120 || age <= 6))
2  {
3      printf("あなたはこのジェットコースターに乗れます。\\n");
4  }
```

ソースコード 11.15 条件の反転 2

```
1  if (!(height < 120) && !(age <= 6))
2  {
3      printf("あなたはこのジェットコースターに乗れます。\\n");
4  }
```

ソースコード 11.16 条件の反転 3

```
1  if (height >= 120 && age > 6)
2  {
3      printf("あなたはこのジェットコースターに乗れます。\\n");
4  }
```

“身長が 120cm 未満”を A、“年齢が 6 歳以下”を B とすると、ソースコード 11.13～11.16 の if 文は、表 11.4 のようにまとめられます。

“A かつ B かつ C”など、3 つ以上の条件の論理積、論理和も考えられます。

西暦の年が 4 で割り切れ、かつ 100 で割り切れない年、または 400 で割り切れる年は^{うるうどし}閏年です。西暦の年が“4 で割り切れる”を A、“100 で割り切れない”を B、“400 で割り切れる”を C とすると、閏年の条件は“(A かつ B) または C”です。C 言語の if 文で、閏年かどうかを判定するプログラムを考えます。int 型の変数 year に西暦の年が入っているとすると、条件 A は

表 11.4 ソースコード 11.13～11.16 の if 文

ソースコード 11.13	A または B	⇒	“乗れません”
ソースコード 11.14	(A または B) でない	⇒	“乗れます”
ソースコード 11.15	(A でない) かつ (B でない)	⇒	“乗れます”
ソースコード 11.16	(A でない) かつ (B でない)	⇒	“乗れます”

year % 4 == 0 と書けます。条件 B は year % 100 != 0、条件 C は year % 400 == 0 です。閏年がどうかを判定する if 文の例を、ソースコード 11.17 に示します。

ソースコード 11.17 閏年の判定 1

```

1  if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
2  {
3      printf("%d年は閏年です.\n", year);
4  }
```

この if 文の制御式は (A && B) || C です。&& は || よりも優先順位が高いため、() は必要なく、A && B || C と書くことができます。そのように修正した条件式を、ソースコード 11.18 に示します。

ソースコード 11.18 閏年の判定 2

```

1  if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
2  {
3      printf("%d年は閏年です.\n", year);
4  }
```

さらに、!a と a == 0 が等価である^{*3}ことなどを使えば、ソースコード 11.19 のように簡略化できます。

ソースコード 11.19 閏年の判定 3

```

1  if (!(year % 4) && year % 100 || !(year % 400))
2  {
3      printf("%d年は閏年です.\n", year);
4  }
```

なれてくればソースコード 11.19 が読みやすいと感じるかもしれません。どの記述方法でも大差ありませんので、読みやすいと感じる方法で記述してください。

^{*3} 本章の【発展】で説明しています。

練習課題 16 (ex34.c)

あるジェットコースターは、年齢が 8 歳以上であれば乗ることができます。年齢が 6 歳以上で、かつ身長が 120cm 以上であっても乗ることができます。身長と年齢を入力するとジェットコースターに乗れるか乗れないかを表示するプログラムを作ってください。ただし、使う if 文は 1 つだけです。途中まで作ったプログラムを、授業のサイトに ex34.c として掲載しています。コメント部分を if 文の制御式に置き換えて、正しく動作するプログラムを作ってください。

実行例 1

身長を入力してください > 115 ↵

年齢を入力してください > 8 ↵

あなたはこのジェットコースターに乗れます。

実行例 2

身長を入力してください > 115 ↵

年齢を入力してください > 7 ↵

あなたはこのジェットコースターに乗れません。

実行例 3

身長を入力してください > 125 ↵

年齢を入力してください > 7 ↵

あなたはこのジェットコースターに乗れます。

第 12 章

様々な演算子

表 7.1 に示した演算子の表を、表 12.1 に再掲します。

表 12.1 演算子の優先順位と結合 (表 7.1 の再掲)

優先順位	演算子	結合
1	() (関数呼び出し) ++ (後置) -- (後置)	左
2	++ (前置) -- (前置) + (単項) - (単項) & !	右
3	() (キャスト)	右
4	* / %	左
5	+ (二項) - (二項)	左
6	< <= > >=	左
7	== !=	左
8	&&	左
9		左
10	?:	右
11	= += -= *= /= %=	右
12	,	左

本章では、この表の演算子のうち、ここまでに説明していないものをすべて説明します。

12.1 単項 +、- 演算子

単項の `-` は、被演算子の符号を反転させます。`int` 型の変数 `a` の値が `5` であるとき、`-a` を評価すると `int` 型の `-5` が得られます*¹。単項の `+` は、被演算子の値そのものです。`int` 型の変数 `a` の値が `5` であるとき、`+a` を評価すると `int` 型の `5` が得られます。単項の `+` は、ソースコードの読

*¹ ソースコード中に直接 `-5` と書いた場合、定数の `5` に単項の `-` 演算子を付けた式と見なされます。`-5` という定数ではありません。

みやすさのために使う以外に、ほとんど使い道はありません。単項の `-` を使った `-a` は、二項の `-` を使った `0 - a` とほぼ同じです。

式 `-i - 4` には、単項の `-` 演算子と二項の `-` 演算子が使われています。`i` の前の `-` は、変数 `i` の符号を反転させる単項の `-` 演算子です。`-i` と `4` の間の `-` は、減算を行う二項の `-` 演算子です。

12.2 複合代入演算子

ソースコード 8.4 では、`weight = weight + 1` という式を使いました。これは、右辺の `weight + 1` を左辺の変数 `weight` に代入する式です。つまり、変数 `weight` に 1 を足します。この授業ではまだあまり登場していませんが、変数に何かを足したり引いたりする操作はよく行われます。ソースコード 12.1 にいくつかの例を示します。

ソースコード 12.1 変数への加算、減算 (算術演算子と代入演算子を使用)

```
1 x = x + 3;      /* 変数xに3を足す      */
2 x = x + y;      /* 変数xに変数yの値を足す */
3 x = x - 5;      /* 変数xから5を引く      */
4 x = x - y * 4; /* 変数xから(y * 4)を引く */
```

`+` と `-` ほど多くはありませんが、`*`、`/`、`%` でも同様の操作を行うことがあります。C 言語には、このような算術演算と代入演算をまとめて簡潔に記述できる演算子が用意されています。`a` を変数、`b` を式とすると、`a += b` は、`a = a + b` と同じ意味です。`-`、`*`、`/`、`%` も同様に、`-=`、`*=`、`/=`、`%=` という演算子が用意されています。これらの演算子を、**複合代入演算子**といいます。表 12.2 に、複合代入演算子を示します。左の列の式は、右の列の式と同じ意味です*2。

表 12.2 複合代入演算子

複合代入演算子		算術演算子と代入演算子
<code>a += b</code>	\iff	<code>a = a + b</code>
<code>a -= b</code>	\iff	<code>a = a - b</code>
<code>a *= b</code>	\iff	<code>a = a * b</code>
<code>a /= b</code>	\iff	<code>a = a / b</code>
<code>a %= b</code>	\iff	<code>a = a % b</code>

数を 0 で割ってはいけないことや、`%` は整数同士の剰余しか求められないことなどは、複合代入演算子でも同様です。ソースコード 12.1 の式 (式文) を、複合代入演算子を使って書き直したものを、ソースコード 12.2 に示します。

*2 実は違いが出ることもあります。この授業の範囲では同じとお考えください。

ソースコード 12.2 変数への加算、減算 (複合代入演算子を使用)

```

1 x += 3;      /* 変数xに3を足す      */
2 x += y;      /* 変数xに変数yの値を足す */
3 x -= 5;      /* 変数xから5を引く      */
4 x -= y * 4; /* 変数xから(y * 4)を引く */

```

最後の式 `x -= y * 4` は、演算子の優先順位にも注意して読んでください。

`int` 型の変数 `a` に 5、`b` に 3 が格納されているとき、各演算を行った後の変数の値を、表 12.3 に表示します。いずれも変数 `b` の値は変化しません。

表 12.3 `a` が 5、`b` が 3 のときの、演算による値の変化

演算	演算後の値
<code>a += b</code>	<code>a</code> は 8、 <code>b</code> は 3
<code>a -= b</code>	<code>a</code> は 2、 <code>b</code> は 3
<code>a *= b</code>	<code>a</code> は 15、 <code>b</code> は 3
<code>a /= b</code>	<code>a</code> は 1、 <code>b</code> は 3
<code>a %= b</code>	<code>a</code> は 2、 <code>b</code> は 3

複合代入演算子を使うと、変数を 2 回書くことによる間違いを減らせ、プログラムも読みやすくなります。使える場所では積極的に使うようにしてください。

練習課題 17 (ex35.c)

税抜き価格と年度を入力すると、税込み価格を表示するプログラムを作ります。現実の消費税とは異なりますが、2019 年度以前は税率 5%、2020 年度以降は税率 10% とします。キーボードから入力した税抜き価格を変数 `price` に、年度を変数 `year` に格納します。if 文を使って、2019 年度以前か 2020 年度以降かで分岐します。2019 年度以前であれば、`price` の値を 1.05 倍します。2020 年度以降であれば、`price` の値を 1.1 倍します。そして、税込み価格に変更した変数 `price` の値を表示します。

このような方針で途中まで作ったプログラムを、授業のサイトに `ex35.c` として掲載しています。コメント部分に複合代入演算子を使った式 (式文) を書いて、正しく動作するプログラムを作ってください。

実行例 1

税抜き価格 > 1500 ↵

年度 > 2018 ↵

税込み価格は 1575 円です。

実行例 2

税抜き価格 > 1500 ↵

年度 > 2020 ↵

税込み価格は 1650 円です。

なお、このプログラムでは、税込み価格を、1 円未満の端数を四捨五入して表示します。また、変数 price を double 型ではなく int 型にすると、int 型の変数 (price) に double 型の値 (税込み価格) を代入することになります。その場合でも (四捨五入ではなく切り捨てで) 正しく動作しますが、この授業では正確に説明していないため double 型にしました。

12.3 インクリメント・デクリメント演算子

複合代入演算子を使えば、変数に値を足したり引いたりする処理が記述できました。その中でも特に、変数に 1 を足したり引いたりする処理はよく行われます。++ は変数に 1 を足す単項演算子で、**インクリメント演算子**と呼ばれます。-- は変数から 1 を引く単項演算子で、**デクリメント演算子**と呼ばれます。インクリメント演算子、デクリメント演算子は、変数の前に書くものと、後ろに書くものの 2 種類あります。変数の前に書くものを**前置**、後ろに書くものを**後置**といいます。インクリメント演算子、デクリメント演算子を表 12.4 に示します。

表 12.4 インクリメント演算子とデクリメント演算子

演算子	役割	評価した値
++a	被演算子である変数 a に 1 を足す	インクリメント後
--a	被演算子である変数 a から 1 を引く	デクリメント後
a++	被演算子である変数 a に 1 を足す	インクリメント前
a--	被演算子である変数 a から 1 を引く	デクリメント前

ソースコード 12.3 の 4 つの文は、すべて同じ処理を行います。

ソースコード 12.3 変数 `x` に 1 を足す

```
1  ++x;      /* 変数xに1を足す その1 */
2  x++;      /* 変数xに1を足す その2 */
3  x += 1;   /* 変数xに1を足す その3 */
4  x = x + 1; /* 変数xに1を足す その4 */
```

前置 (`++a`、`--a`) と後置 (`a++`、`a--`) は優先順位が違うほか、評価したときの値が異なります。

前置のインクリメント演算子、デクリメント演算子を使った式を評価すると、インクリメント (1 を足す) 後、デクリメント (1 を引く) 後の値が得られます。つまり、`int` 型の変数 `x` の値が 3 であるとき、式 `++x` を評価すると `int` 型の 4 が得られます。変数 `x` の値が 3 であるとき、式 `--x` を評価すると `int` 型の 2 が得られます。

後置のインクリメント演算子、デクリメント演算子を使った式を評価すると、インクリメント前、デクリメント前の値が得られます。つまり、`int` 型の変数 `x` の値が 3 であるとき、式 `x++` を評価すると `int` 型の 3 が得られます。変数 `x` の値が 3 であるとき、式 `x--` を評価すると `int` 型の 3 が得られます。

変数に 1 が足される、または変数から 1 が引かれる点は、前置も後置も同じです。前置は変数进行操作してからその値を得る、後置は変数の値を得てからその変数进行操作する、と考えることができます。

インクリメント演算子、デクリメント演算子を単独で使う場合は、前置と後置はほぼ同じですが、別の演算子と組み合わせる場合は結果が異なる可能性があります。ソースコード 12.4 と 12.5 で、それぞれ 2 行目の文を実行した直後の、変数 `x` と `y` の値を考えてみてください。

ソースコード 12.4 前置のインクリメント演算子

```
1  int x = 5, y;
2  y = ++x;
```

ソースコード 12.5 後置のインクリメント演算子

```
1  int x = 5, y;
2  y = x++;
```

ソースコード 12.4 では、前置のインクリメント演算子を使っています。`++x` を評価するとインクリメント後の値である 6 が得られ、これが変数 `y` に代入されます。つまり、2 行目の文を実行した直後は、変数 `x` と `y` の値は 6 です。ソースコード 12.5 では、後置のインクリメント演算子を使っています。`x++` を評価するとインクリメント前の値である 5 が得られ、これが変数 `y` に代入されます。つまり、2 行目の文を実行した直後は、変数 `x` の値は 6、`y` の値は 5 です。

ソースコード 12.4 はソースコード 12.6 とほぼ同じ、ソースコード 12.5 はソースコード 12.7 とほぼ同じです。

ソースコード 12.6 ソースコード 12.4 と同じ処理

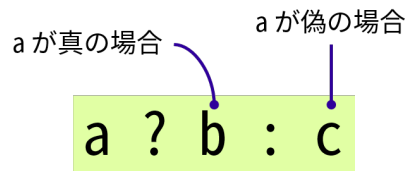
```
1 int x = 5, y;
2 x += 1;
3 y = x;
```

ソースコード 12.7 ソースコード 12.5 と同じ処理

```
1 int x = 5, y;
2 y = x;
3 x += 1;
```

12.4 条件演算子

?: を条件演算子といいます。条件演算子は、被演算子が3つ必要な三項演算子です。1つ目の被演算子を? の左側、2つ目の被演算子を? と : の間、3つ目の被演算子を: の右側に書きます。式 `a ? b : c` を評価すると、a が0以外であればbを、a が0であればcを評価した値が得られます (図 12.1)。



- a が0以外(真)なら、式全体が b
- a が0(偽)なら、式全体が c

図 12.1 条件演算子

キーボードから入力した2つの整数の大きい方を表示するプログラムを、ソースコード 12.8 に示します。

ソースコード 12.8 2つの数の大きい方を表示 (条件演算子を使用) (ex36.c)

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int x, y;
6     int max;
```

```

7
8     printf("整数を2つ入力してください.\n");
9     printf("1つ目 > ");
10    scanf("%d", &x);
11    printf("2つ目 > ");
12    scanf("%d", &y);
13
14    max = x >= y ? x : y;
15
16    printf("大きい方は%dです.\n", max);
17
18    return 0;
19 }

```

14 行目で条件演算子 `?:` を使っています。条件演算子の部分だけを抜き出し、さらに `()` を付けてわかりやすくすると、`(x >= y) ? x : y` です。`a ? b : c` の `a` に相当する部分が `x >= y`、`b` に相当する部分が `x`、`c` に相当する部分が `y` です。`x >= y` が 0 でなければ、つまり `x` が `y` 以上であれば、式全体が `x` となります。`x` が `y` 以上でなければ、式全体が `y` となります。その結果、この式を評価すると、`x` と `y` の大きい方の値が得られます (図 12.2)。それを変数 `max` に代入し、16 行目で“大きい方”として表示しています。14 行目の文は、演算子の優先順位にも注意して読んでください。

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

- `x >= y` が 0 以外(真)なら、式全体が `x`
- `x >= y` が 0(偽)なら、式全体が `y`

図 12.2 ソースコード 12.8 の条件演算子を使った式

なお、このプログラムでは、変数 `max` を使う必要はありません。次のように、`printf()` 関数の第 2 引数に直接式を書くこともできます。

ソースコード 12.9 変数 `max` を使わず、直接出力

```
printf("大きい方は%dです.\n", x >= y ? x : y);
```


練習課題 18 (ex37.c)

練習課題 17 で作った ex35.c を改良します。ex35.c と同じ動作をするプログラムを、if 文を使わずに、条件演算子を使って書いてください。途中まで作ったプログラムを、授業のサイトに ex37.c として掲載しています。コメント部分に条件演算子を使った文を 1 つ書いて、正しく動作するプログラムを作ってください。もちろん、条件演算子だけを使うわけではありません。

12.5 コンマ演算子

表 12.1 の最下行にある , は、コンマ演算子 (カンマ演算子) です。コンマ演算子は、演算子の後に式を置く二項演算子です。[a, b] と書くと、a を評価した後、b を評価します。そして、b を評価して得られる値が、式 a, b を評価して得られる値となります。i を int 型の変数とするとき、次の例を考えます。

ソースコード 12.10 コンマ演算子の使用例 1

```
i = (3, 5);
```

3, 5 の部分がコンマ演算子を使った式です。まず、3 が評価され、次に 5 が評価されます。5 を評価して得られる int 型の 5 が、式 3, 5 を評価した値になります。よって、変数 i には、5 が代入されます。コンマ演算子は、= よりも優先順位が低いいため、() を消して次のように書くと評価順が変わります。

ソースコード 12.11 コンマ演算子の使用例 2

```
i = 3, 5;
```

この場合、まず i = 3 が評価され、変数 i に 3 が代入されます。次に 5 が評価され、int 型の 5 が得られます。式 i = 3, 5 を評価した値は int 型の 5 ですが、その値は使っていません。

コンマ演算子は、不思議な演算子と感じるかもしれません。コンマ演算子を使うと、2 つの式を 1 つの式にすることができます。つまり、式を 1 つしか書けない場所に複数の式を書くためにコンマ演算子を使うことができます^{*3}。そのような例は、次章で登場します。

複合文を使うと、複数の文を 1 つの文にすることができました。複合文とコンマ演算子が見える場面を以下にまとめます。

1 つの文しか記述できない場所に、複数の文を記述したい ⇒ **複合文**

1 つの式しか記述できない場所に、複数の式を記述したい ⇒ **コンマ演算子**

^{*3} コンマ演算子は二項演算子ですが、a, b, c, d のように複数使えば、複数の式を 1 つの式にできます。左結合なので、((a, b), c), d) と見なされます。

第 13 章

反復

13.1 順次・分岐・反復

プログラムの中で、同じ処理を何度か繰り返すことができます。このようなプログラムの構造を**反復**、または**繰り返し**といいます。反復の処理の流れを、図 13.1 に示します。

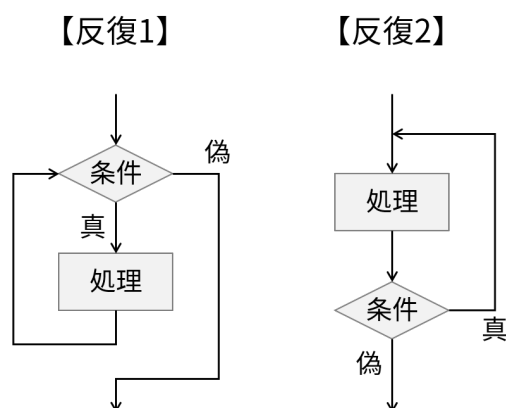


図 13.1 反復

反復 1 では、まず条件判断を行います。条件が成立していれば処理を行い、その後、条件判断に戻ります。条件が成立していなければ、処理は行わず、先に進みます。つまり、“条件が成立していれば処理を行う”ことを繰り返します。

反復 2 では、まず処理を行います。その後、条件判断を行い、成立していればもう一度処理を行います。成立していなければ、先に進みます。こちらも、“条件が成立していれば処理を行う”ことを繰り返します。

反復 1 では、初回の条件判断で条件が成立していなければ、処理は 1 回も行われません。反復 2 では、最低 1 回は処理が行われます。C 言語では、どちらの反復も記述できますが、この授業では、反復 1 に相当する記述方法を 2 種類説明します。

第 8 章では、プログラムの構造として、順次と分岐について学びました。順次、分岐、反復が、

プログラムの構造に関する基本的な要素です。C 言語のプログラムは、これら 3 種類の要素を組み合わせて作ります。分岐の中で分岐を行う例は、第 11 章で紹介しました。分岐の中で反復を行うことや、反復の中で分岐を行うこともできます。反復の次に分岐を行なったとすると、その部分は順次といえます。プログラムの構造の例を、図 13.2 に示します。

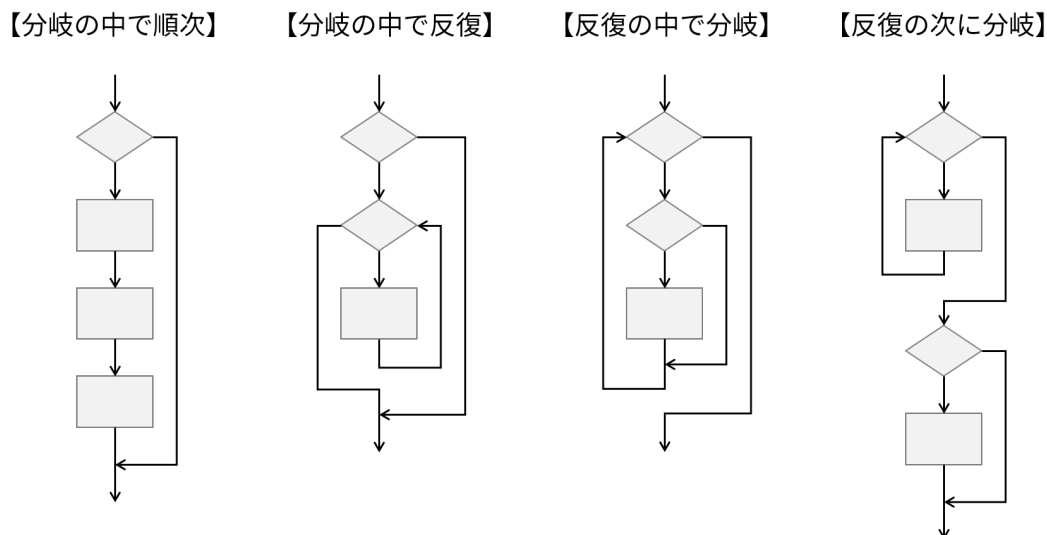


図 13.2 プログラムの構造の例

13.2 while 文

while 文を使うと、図 13.1 の反復 1 が実現できます。while 文の書式は、

while (制御式) 文

です。プログラムの流れが、while 文に到達すると、まず制御式が評価されます。その値が 0 でなければ、“文” が実行されます。0 であれば、“文” は実行されずに、次の処理に進みます。“文” を実行した場合、実行後に制御式の評価に戻ります。その値が 0 でなければ、“文” が再び実行され、また制御式の評価に戻ります。0 であれば、“文” は実行されずに、次の処理に進みます。このように、制御式を評価し、0 でなければ“文” を実行することを繰り返します。これ以降、“文” をループ本体と呼びます。

while 文の使用例をソースコード 13.1 に、このプログラムの構造を図 13.3 に示します。9 行目から 13 行目が while 文です。この while 文の制御式は `i <= 5` です。if 文同様、while 文のループ本体には、1 つの文しか書けません。そこで複合文を使っています。10 行目から 13 行目がループ本体です。

ソースコード 13.1 while 文の使用例 (ex38.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6
7      i = 1;
8
9      while (i <= 5)
10     {
11         printf("%d ", i);
12         ++i;
13     }
14
15     printf("\n");
16
17     return 0;
18 }

```

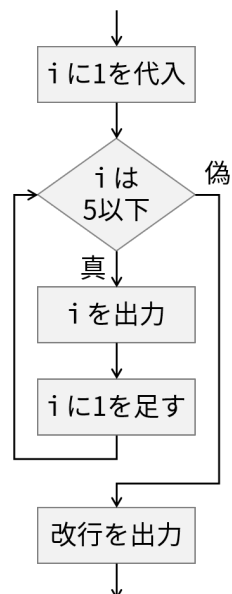


図 13.3 ソースコード 13.1 のプログラムの構造

実行結果

1 2 3 4 5

このプログラムを実行すると、まず、7行目で変数 *i* に 1 が代入されます。次に `while` 文に到達して、制御式 `i <= 5` が評価されます。評価した値は 0 ではない*1)ので、ループ本体が実行されます。その結果、11 行目で “1 ” (数字の後ろにスペースあり) と表示され、12 行目で *i* の値が 2 に変わります。ループ本体の実行が終わったので、再度制御式 `i <= 5` が評価されます。制御式を評価した値は 0 ではない*2)ので、再びループ本体が実行されます。その結果、“2 ” と表示され、*i* の値が 3 に変わります。これを繰り返し、“3 ”、“4 ”、“5 ” と表示されます。“5 ” を表示した後に、*i* の値は 6 になります。その次に制御式を評価すると、値は 0 なので、`while` 文の実行が終了し、先に進みます。そして、15 行目で改行が行われます。最終的に、このプログラムを実行すると、“1 2 3 4 5 ” と表示されることになります。

【発展】 前置、後置のインクリメント演算子

ソースコード 13.1 の `while` 文を再掲します。

```
9      while (i <= 5)
10     {
11         printf("%d ", i);
12         ++i;
13     }
```

11 行目で変数 *i* の値を表示して、12 行目で *i* に 1 を足しました。ここでは 2 文に分けましたが、次のように 1 文で書くこともできます。

```
9      while (i <= 5)
10     {
11         printf("%d ", i++);
12     }
```

11 行目の文の動作を考えます。`while` 文に到達したとき、変数 *i* の値は 1 です。11 行目の `printf()` 関数の第 1 引数には、変換指定 `%d` が含まれています。そこで、第 2 引数である `i++` が評価され、その値で `%d` を置き換えます。`i++` を評価すると *i* の値が 2 に変わりますが、後置のインクリメント演算子を使っているため、評価して得られる値は 1 です。つまり、“1 ” と表示した上で、*i* の値が 2 に変わるようになります。これは、*i* の値を表示してから、*i* に 1 を足す処理と同じです。

*1 変数 *i* の値は 1 なので、`1 <= 5` が評価され、1 が得られます。

*2 これ以降、`2 <= 5`、`3 <= 5`、… が評価されることになります。

また、次のように、`i` を 1 ではなく 0 から始めるようにして、`while` 文の制御式を `++i <= 5` とすることもできます。こちらも動作を考えてみてください。

```
7     i = 0;
8
9     while (++i <= 5)
10    {
11        printf("%d ", i);
12    }
```

ソースコード 13.1 は、1 から 5 まで表示するプログラムでした。次に、1 から、ユーザが指定した数まで表示するプログラムを考えます。ソースコード 13.1 の `while` 文の制御式は `i <= 5` でした。この式の 5 が、表示する最大の数です。そこで、この部分をキーボードから入力した数に置き換えれば、目的のプログラムになります。ソースコード 13.2 にプログラム例を示します。キーボードから入力された整数を変数 `n` に格納し、`while` 文の制御式で使っています。

ソースコード 13.2 1 から指定した数まで表示するプログラム (ex39.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int n, i;
6
7      printf("正の整数を入力してください > ");
8      scanf("%d", &n);
9
10     i = 1;
11
12     while (i <= n)
13     {
14         printf("%d ", i);
15         ++i;
16     }
17
18     printf("\n");
19
20     return 0;
```

21 }

実行例 1

正の整数を入力してください > 3 ↵

1 2 3

実行例 2

正の整数を入力してください > 21 ↵

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

ここから、反復を使ったプログラムを書いていきます。その際、プログラムの誤りによって、反復が無限に繰り返される**無限ループ**になってしまうことがあります。実行したプログラムが終了しなくなった場合は、強制終了させてください。環境によって違いますが、たいてい、Ctrl+C^{*3}で強制終了できます。

練習課題 19 (ex40.c)

正の整数を入力すると、その数から 1 まで、1 ずつ小さくしながら表示するプログラムを作ってください。

実行例 1

正の整数を入力してください > 7 ↵

7 6 5 4 3 2 1

実行例 2

正の整数を入力してください > 18 ↵

18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

練習課題 20 (ex41.c)

練習課題 19 で作った ex40.c を、3 の倍数のみ表示するように改変してください。

^{*3} Ctrl キーを押しながら “C” のキーを押す。

<p>実行例 1</p> <p>正の整数を入力してください > 7 ↵</p> <p>6 3</p>
<p>実行例 2</p> <p>正の整数を入力してください > 18 ↵</p> <p>18 15 12 9 6 3</p>

13.3 反復の制御

`while` 文のループ本体が複合文である場合、複合文の中の文はすべて実行されます。`break` 文と `continue` 文を使うと、ループ本体の実行を途中でやめることができます。`break` 文は、`while` 文自体を終了させます。`break` 文の書式は、

```
break;
```

です。`continue` 文は、ループ本体の残りの処理を飛ばして、次の反復の条件判断に進めます。`continue` 文の書式は

```
continue;
```

です。図 13.4 に、`break` 文と `continue` 文のイメージを示します。

`break` 文と `continue` 文を使ったプログラムの例を、ソースコード 13.3 に示します。

ソースコード 13.3 `break` 文と `continue` 文の使用例 (ex42.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i = 0;
6
7      /* 1から100のうち、3の倍数でも5の倍数でもない数を表示する */
8      while (1)
9      {
10         ++i;
11
12         /* 100より大きくなったらwhile文を終了する */

```



```

13     if (i > 100)
14     {
15         break;
16     }
17
18     /* 3または5の倍数なら以降を飛ばす */
19     if (i % 3 == 0 || i % 5 == 0)
20     {
21         continue;
22     }
23
24     printf("%d ", i);
25 }
26
27 printf("\n");
28
29 return 0;
30 }

```

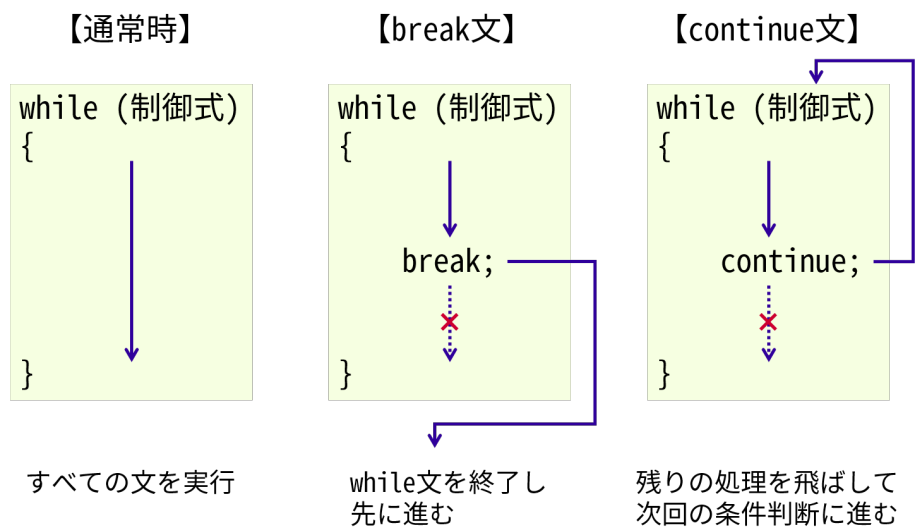


図 13.4 break 文と continue 文

実行結果

```
1 2 4 7 8 11 13 14 16 17 19 22 23 26 28 29 31 32 34 37 38 41 43 44 46
47 49 52 53 56 58 59 61 62 64 67 68 71 73 74 76 77 79 82 83 86 88 89 91
92 94 97 98
```

7 行目のコメントに記載したとおり、このプログラムは、1 から 100 のうち、3 の倍数でも 5 の倍数でもない数を表示します。8 行目から 25 行目までが `while` 文です。この `while` 文の制御式は 1 です。0 ではありませんので、常に条件が成立する無限ループになります。まず、ループ本体のうち、12 行目から 22 行目がない場合の動作を考えます。変数 `i` が 0 の状態で、`while` 文に到達します。ループ本体の先頭 (10 行目) で、`i` に 1 を足しています。そして、24 行目で `i` の値を表示しています。次の反復では 10 行目で `i` の値を 2 にして、24 行目でその値を表示します。同様に、次の反復では 3、さらに次の反復では 4 を表示します。つまり、1 から順に 1 ずつ増やしながら、数を無限に表示します。

次に、12 行目から 22 行目も考えます。表示する数を 1 から 100 までにするためと、3 の倍数でも 5 の倍数でもない数だけを表示するために、`break` 文と `continue` 文を使っています。最初の反復では `i` が 1 の状態で `printf()` 文が実行されます。次の反復では 2、その次の反復では 3 です。`i` が 100 より大きくなったら `while` 文を終了すれば、表示する数を 1 から 100 までに制限できます。そのために、13 行目から 16 行目の `if` 文で条件判断をして、`break` 文を実行しています。また、`i` が 3 の倍数、もしくは 5 の倍数のときに 24 行目の `printf()` 文に到達しなければ、3 の倍数でも 5 の倍数でもない数だけを表示することができます。そのために、19 行目から 22 行目の `if` 文で条件判断をして、`continue` 文を実行しています。

ソースコード 13.3 の `while` 文の動作を、表 13.1 に示します。それぞれの反復回では、`break` 文を実行する条件と `continue` 文を実行する条件のどちらも偽の場合に、`i` の値が表示されます。ループ本体は 101 回実行されることも確認してください。

`break` 文が無条件で実行されると、わざわざ反復にする意味がありません。`continue` 文が無条件で実行されると、それ以降の処理を書く意味がありません。普通はこの例のように、`if` 文などを使って、条件を満たした場合のみ `break` 文や `continue` 文を実行します。

表 13.1 ソースコード 13.3 の `while` 文の動作

反復回	10 行目直後の <code>i</code>	<code>i > 100</code>	<code>i</code> は 3 か 5 の倍数	表示
1 回目	1	偽	偽	1
2 回目	2	偽	偽	2
3 回目	3	偽	真 → <code>continue</code>	
4 回目	4	偽	偽	4
5 回目	5	偽	真 → <code>continue</code>	
		⋮		
98 回目	98	偽	偽	98
99 回目	99	偽	真 → <code>continue</code>	
100 回目	100	偽	真 → <code>continue</code>	
101 回目	101	真 → <code>break</code>		

練習課題 21 (ex43.c)

ソースコード 13.3(ex42.c) を改変して、1 から 100 のうち、3 の倍数か 5 の倍数である数を、降順 (大きい順) に表示するプログラムを作ってください。

実行結果

```
100 99 96 95 93 90 87 85 84 81 80 78 75 72 70 69 66 65 63 60 57
55 54 51 50 48 45 42 40 39 36 35 33 30 27 25 24 21 20 18 15 12 10
9 6 5 3
```

13.4 for 文

ソースコード 13.1 の 7 行目から 13 行目を再掲します。

```

7      i = 1;
8
9      while (i <= 5)
10     {
11         printf("%d ", i);
12         ++i;
13     }
```

7行目は、反復に入る前に一度だけ行う処理です。12行目は、ループ本体の最後に毎回行う処理です。このように、反復に入る前にその前提となる処理を行ったり、各反復回の最後に次の反復回に向けての処理を行ったりしたいことがあります。for文を使うと、そのような処理を行う反復を書けます。for文は、while文同様、図13.1の反復1が実現できます。for文の書式は、

`for (式1; 式2; 式3) 文`

です。while文同様、“文”をループ本体と呼びます。for文に到達すると、式₁が評価されます。次に式₂が評価され、その値が0でなければ、ループ本体が実行されます。0であれば、ループ本体は実行されずにfor文は終了し、次の処理に進みます。ループ本体を実行した場合、実行後に式₃が評価されます。その後式₂が評価され、その値が0でなければ、ループ本体が再び実行されます。つまり、反復に入る前に一度だけ行う処理を式₁に、各反復回が終わった後に行う処理を式₃に書きます。式₂は、while文の制御式に相当し、for文でも制御式と呼びます。break文とcontinue文は、while文と同じ動作をします。

ソースコード13.1をfor文を使って書き直したものを、ソースコード13.4に示します。

ソースコード 13.4 for文の使用例 (ex44.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6
7      for (i = 1; i <= 5; ++i)
8      {
9          printf("%d ", i);
10     }
11
12     printf("\n");
13
14     return 0;
15 }
```

このプログラムのfor文の動作は次のとおりです。

- まず、iを1にする(式₁)。
- iが5以下であれば(式₂)、ループ本体を実行する。
- iに1を足す(式₃)。
- iが5以下であれば(式₂)、ループ本体を実行する。

- i に 1 を足す (式₃)。
 - i が 5 以下であれば (式₂)、ループ本体を実行する。
 - i に 1 を足す (式₃)。
- ⋮

i が 6 になると、式₂ を評価した値が 0 になるため、ループ本体は実行されず、for 文は終了します。この for 文は、 i の値を 1 から 5 まで 1 ずつ増やしながらループ本体を実行する、と考えることができます。

式₁ や 式₃ が不要な場合もあります。式₁、式₂、式₃ は、いずれも省略することができます。式₂ を省略した場合、0 ではないと見なされ、ループ本体が実行されます。いずれかの式を省略した場合でも、`;` は省略できません。

ソースコード 13.3 では、while 文で無限ループを実現するために制御式を 1 にしました。
`for (式1; 1; 式3) 文` や `for (式1; ; 式3) 文` とすれば、for 文で無限ループが実現できます。

while 文と for 文のどちらを使っても、同じような反復が記述できます。図 13.5 の左側の for 文と右側の while 文は同じ処理を行います。ただし、ループ本体で continue 文を使っていない場合のみです。continue した場合、左側の for 文では 式₃ が評価されますが、右側の while 文の 式₃ は評価されません。

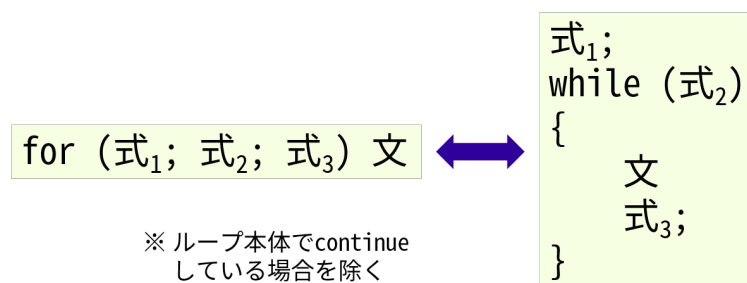


図 13.5 while 文と for 文

1 から 100 のうち、3 の倍数か 5 の倍数である数を数えるプログラムを、ソースコード 13.5 に示します。

ソースコード 13.5 1 から 100 のうち、3 の倍数か 5 の倍数である数を数えるプログラム (ex45.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i; /* 反復用の変数 */
6      int c; /* カウント用の変数 */

```

```

7
8     for (i = 1, c = 0; i <= 100; ++i)
9     {
10         if (i % 3 == 0 || i % 5 == 0)
11         {
12             ++c;
13         }
14     }
15
16     printf("1から100のうち、3の倍数か5の倍数である数は%d個です.\n", c);
17
18     return 0;
19 }

```

実行結果

1 から 100 のうち、3 の倍数か 5 の倍数である数は 47 個です。

このプログラムは、変数 *i* の値を 1 から 100 まで変えながら、*i* が 3 の倍数、または 5 の倍数であるかを確認します。3 の倍数、または 5 の倍数であった場合、変数 *c* に 1 を足します。100 まで確認し終わった後に、*c* に入っている値が求める数となります。そのためには、最初に *c* に 0 を入れておく必要があります。

8 行目から 14 行目が `for` 文です。変数 *i* の値を 1 から 100 まで変えながらループ本体を実行するためには、`for (i = 1; i <= 100; ++i) 文` とします。このプログラムでは、`for` 文の最初で *i* に 1 を代入するだけでなく、*c* に 0 を代入する必要もあります。そこで、式₁ を `i = 1, c = 0` としています。前章で説明したコンマ演算子を思い出してください。式₁ には式を 1 つしか書けませんので、コンマ演算子を使って 2 つの代入式を 1 つの式にまとめています。

13.5 反復のネスト

図 13.2 では、分岐の中に反復が入っている例と、反復の中に分岐が入っている例を紹介しました。反復の中で反復を使うことももちろんできます。このように、分岐や反復の中に別の分岐や反復が入っている構造を、**ネスト**や**入れ子**といいます。

ソースコード 13.6 は、“o” (小文字のオー) を使って、長方形を描くプログラムです。`for` 文を使った反復がネストになっており、外側の `for` 文が 7 行目から 14 行目、内側の `for` 文が 9 行目から 12 行目です。まず、*i* が 0 の状態で内側の `for` 文に到達します。内側の `for` 文は、“o” を 5 回出力して終了します。その後、13 行目で改行を出力し、外側の 1 回目の反復が終了します。その結果、“ooooo” と出力され改行されます。次に *i* が 1 の状態で同じ動作を行います。*i* が 9 まで

同じ動作が行われますので、最終的に“ooooo”の行が10回出力されます。

ソースコード 13.6 長方形を描くプログラム (ex46.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i, j;
6
7      for (i = 0; i < 10; ++i)
8      {
9          for (j = 0; j < 5; ++j)
10         {
11             printf("o");
12         }
13         printf("\n");
14     }
15
16     return 0;
17 }
```

実行結果

```
ooooo
ooooo
ooooo
ooooo
ooooo
ooooo
ooooo
ooooo
ooooo
ooooo
```

練習課題 22 (ex47.c)

ソースコード 13.6(ex46.c) を改変し、次のような 10 段の直角三角形を描くプログラムを作ってください。

実行結果

```
o
oo
ooo
oooo
ooooo
oooooo
ooooooo
oooooooo
ooooooooo
oooooooooo
```

i が 0 の反復では“o”を 1 回出力します。つまり、内側の反復が 1 回繰り返されればよいことになります。 i が 1 の反復では“o”を 2 回出力します。そのために、内側の反復を 2 回繰り返します。 i が x の反復では、内側の反復を $x + 1$ 回繰り返します。これを実現するためには、変数 i を使って、内側の反復の制御式を書く必要があります。

第 14 章

関数の使用

第 5 章で、関数の使い方を学びました。本章と次章では、関数についてさらに詳しく学習します。これまでに登場した関数は、`main()` 関数、`printf()` 関数、`scanf()` 関数の 3 つです。`main()` 関数は特別な関数、`printf()` 関数と `scanf()` 関数は少し変わった関数です。まずは、普通の関数から見ていきましょう。

14.1 引数と戻り値

関数は**引数**を受け取って、**戻り値**を返します。引数とは、関数の実行に必要な値です。戻り値とは、関数を実行した結果の値です。引数を付けて関数呼び出しを行うと、戻り値を得ることができます。

たとえば、引数を 2 乗する関数があるとします。この関数に引数として 5 を渡すと、戻り値として 25 が返ってきます。12 を渡すと 144 が返ってきます。英文字の大文字を小文字に、小文字を大文字にする関数の場合、引数として “E” を渡すと、戻り値として “e” が返ってきます。“r” を渡すと “R” が返ってきます。図 14.1 に、関数のイメージを示します。

引数としてどのような値を与えればよいかは、関数によって異なります。2 乗する関数であれば“数”、大文字小文字を入れ替える関数であれば“英文字”です。また、C 言語の関数の引数は 1 つだけとは限りません。たとえば、英文字を、指定された文字数分、アルファベット順にずらす関数を考えます。“Z” の次は “A” に戻るとします。この関数には、文字と、ずらす文字数の 2 つの引数が必要です。図 14.2 のように、引数として “F” と “7” を渡すと、戻り値として “F” を 7 文字ずらした “M” が返ってきます。“Y” と “4” を渡すと、“C” が返ってきます。

C 言語の関数の戻り値の個数は、0(戻り値なし) か 1 です。

14.2 ライブラリ関数

関数はあらかじめ用意されているものを使うことも、自分で作って使うこともできます。C 言語の規格で定められた、あらかじめ用意されている関数を**ライブラリ関数**といいます。`printf()` 関

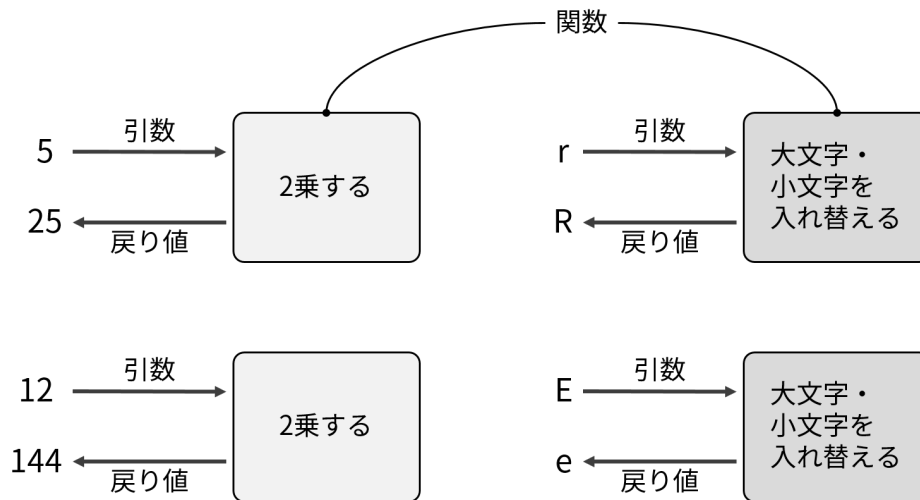


図 14.1 関数のイメージ

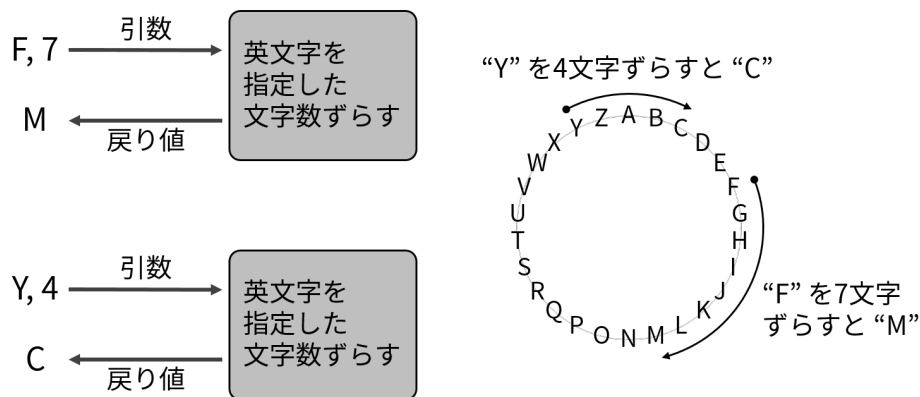


図 14.2 引数が2つ必要な関数

数と `scanf()` 関数は、ライブラリ関数です。

ライブラリ関数を使うためには、**ヘッダ**と呼ばれるものが必要です。それを取り込むために、`#include` ディレクティブを使って、**ヘッダをインクルード**します。`printf()` 関数と `scanf()` 関数に必要なヘッダは、`<stdio.h>`です。

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

と書くことによって、`<stdio.h>`をインクルードできます。ヘッダは、ライブラリ関数を使う前に^{*1}インクルードしなければなりません。普通は、ソースコードの先頭に書きます。これまで定型部分として、`#include <stdio.h>`をソースコードの先頭を書いてきました。これは、`printf()` 関数や `scanf()` 関数を使っていたからです。これらの関数を使わなければ、`#include <stdio.h>`

^{*1} ソースコード中で、ライブラリ関数を使っている場所より上で。

を書く必要はありません。ただ、入力も出力もないプログラムは実行する意味がありません。`<stdio.h>`は、入出力関係の関数に必要なヘッダで、ほぼすべてのプログラムに必須です。

ライブラリ関数の例をいくつか紹介します。

sqrt() 関数 … 平方根を求める

`sqrt()` は、平方根を求める関数です。表 14.1 に使い方を示します。引数として、非負の数 x を与えると、戻り値として x の平方根が返ってきます。引数、戻り値ともに、`double` 型の値です。また、必要ヘッダは、`<math.h>`です。

表 14.1 `sqrt()` 関数

書式	<code>sqrt(x)</code>
引数	非負の数 x (<code>double</code> 型)
戻り値	x の非負の平方根 (<code>double</code> 型)
必要ヘッダ	<code><math.h></code>

`sqrt()` 関数を使ったプログラムの例を、ソースコード 14.1 に示します。`sqrt()` など、`<math.h>`のインクルードが必要な関数を使う場合、コンパイル時に追加のオプションが必要な場合があります。コンパイルに `gcc` を使っていて*2、“`‘sqrt’` に対する定義されていない参照です”や“`undefined reference to ‘sqrt’`”といったエラーが出る場合、次のように、コンパイルのコマンドの最後に `-lm` (ハイフン・エル・エム) を付けてみてください。

```
gcc -o ex48.exe ex48.c -lm
```

ソースコード 14.1 `sqrt()` 関数の使用例 (ex48.c)

```
1  #include <math.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6      double x;
7
8      printf("非負の数を入力してください > ");
9      scanf("%lf", &x);
10
11     printf("%f の平方根は %f です.\n", x, sqrt(x));
```

*2 コマンド名が `cc` で、中身が `gcc` である場合などもあります。

```

12
13     return 0;
14 }

```

実行例 1

非負の数を入力してください > 5 ↵
 5.000000 の平方根は 2.236068 です。

実行例 2

非負の数を入力してください > 9.7 ↵
 9.700000 の平方根は 3.114482 です。

ソースコード 14.1 では、`sqrt()` 関数を使うために、1 行目で `<math.h>` をインクルードしています。また、`printf()` 関数と `scanf()` 関数を使うために、2 行目で `<stdio.h>` をインクルードしています。キーボードから入力された数を、`double` 型の変数 `x` に格納し、11 行目の `printf()` 関数で使っています。この `printf()` 関数の第 1 引数である文字列リテラル中には、変換指定 `%f` が 2 つあります。1 つ目の `%f` は第 2 引数である `x` の値に、2 つ目の `%f` は第 3 引数である `sqrt(x)` の値に置き換えられます。関数呼び出し式を評価すると、その関数が実行され、戻り値が得られます。たとえば、`x` の値が 5 の場合、関数呼び出し式 `sqrt(x)` を評価すると、`sqrt()` 関数が 5 を引数として実行され、その結果である戻り値 2.236… が得られます。2 つ目の `%f` は、2.236… に置き換えられることになります。

ソースコード 14.1 では戻り値を直接表示しましたが、もちろん変数に代入して使うこともできます。戻り値を `double` 型の変数 `y` に代入したければ、`y = sqrt(x);` のように記述します。

abs() 関数 … 絶対値を求める

`abs()` は、絶対値を求める関数です。表 14.2 に使い方を示します。引数として、整数 `j` を与えると、戻り値として `j` の絶対値が返ってきます。引数、戻り値ともに、`int` 型の値です。また、必要ヘッダは、`<stdlib.h>` です。

表 14.2 `abs()` 関数

書式	<code>abs(j)</code>
引数	整数 <code>j</code> (<code>int</code> 型)
戻り値	<code>j</code> の絶対値 (<code>int</code> 型)
必要ヘッダ	<code><stdlib.h></code>

`abs()` 関数を使ったプログラムの例を、ソースコード 14.2 に示します。要点はソースコード

14.1 と同じですので、説明は省略します。

ソースコード 14.2 abs() 関数の使用例 (ex49.c)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int j;
7
8      printf("整数を入力してください > ");
9      scanf("%d", &j);
10
11     printf("%d の絶対値は %d です.\n", j, abs(j));
12
13     return 0;
14 }
```

実行例 1

整数を入力してください > -7 ↵
-7 の絶対値は 7 です。

実行例 2

整数を入力してください > 12 ↵
12 の絶対値は 12 です。

pow() 関数 … べき乗を求める

pow() は、べき乗を求める関数です。表 14.3 に使い方を示します。

表 14.3 pow() 関数

書式	pow(x , y)
引数	数 x と y (いずれも double 型)
戻り値	x の y 乗 (double 型)
必要ヘッダ	<math.h>

【参考】 べき乗

Microsoft Excel では、演算子 \wedge を使ってべき乗が計算できます。5 の 3 乗は、 5^3 です。また、POWER というべき乗を求める関数も用意されており、POWER(5,3) で 5 の 3 乗が計算できます。C 言語では関数 pow() を使ってべき乗を求めますが、プログラミング言語によっては、べき乗の演算子が用意されています。Basic というプログラミング言語のべき乗の演算子は、Excel と同じく \wedge です。Perl というプログラミング言語では、** です。

プログラミング言語	5 の 3 乗
C 言語	pow(5, 3)
Basic	$5 \wedge 3$
Perl	$5 ** 3$

Excel で使われているからか、C 言語でも 5^3 と書いてしまう誤りがあります。C 言語では、 \wedge は別の役割を持った演算子です。この授業の範囲を超えるため説明は省略しますが、C 言語の 5^3 は 6 です。コンパイル、実行はできるのに、計算結果が想定と違うことになります。

練習課題 23 (ex50.c)

$3^{3.5}$ と $0.7^{(-2.5)}$ を計算し、表示するプログラムを作ってください。

実行例

3 の 3.5 乗は 46.765372 です。
0.7 の (-2.5) 乗は 2.439242 です。

toupper() 関数 … 小文字を大文字にする

toupper() は、小文字を大文字にする関数です。表 14.4 に使い方を示します。

表 14.4 toupper() 関数

書式	toupper(c)
引数	文字 c
戻り値	c の大文字 (c が小文字でなければ、c をそのまま返す)
必要ヘッダ	<ctype.h>

練習課題 24 (ex51.c)

文字を入力すると、大文字に変換して表示するプログラムを作ってください。小文字以外が入力された場合は、そのまま表示してください。

実行例 1

文字を入力してください > q ↵
q を大文字にすると Q です。

実行例 2

文字を入力してください > 7 ↵
7 を大文字にすると 7 です。

【発展】 文字の扱いと char 型

コンピュータの中で扱われるデータの最小単位を**ビット**といいます。ビットは、0 と 1 の 2 通りの状態を取ります。コンピュータの中では、すべてのデータがビットの並び (ビット列) として扱われています。各種のデータは、通常、8 ビット (1 バイト) の整数倍の領域に格納されます。

負でない整数は、二進数で表現したときの 0 と 1 を、ビットの 0 と 1 に対応させます。たとえば、十進数の 18 は、二進数で 10010 です。十進数の 18 を 32 ビット (4 バイト) の領域に格納する場合、論理的には 00000000 00000000 00000000 00010010 というビット列になります。int 型が 32 ビットの場合、`int i = 18;` とすると、変数 i の領域として 32 ビットが確保され、上記のようなビット列が格納されます。負の数や小数は省略しますが、どちらも 0 と 1 の並びで表されます。

文字も同様です。たとえば、多くのコンピュータでは、“A” という文字は 01000001 というビット列で表されます。これを整数として読むと 65 です。同じビット列でも、整数として解釈するか文字として解釈するかによって値が異なります。データを保存する側と、取り出して使う側で同じ解釈をしていれば問題は起こりません。

C 言語の char 型には文字が入りますが、char 型は整数の性質を持っています。授業のサイトに掲載している adv02.c をコンパイル、実行してみてください。第 4 章の発展で使った adv01.c に、char 型の範囲を表示する処理を追加したものです。

Windows 10 上の MinGW(gcc version 8.2.0) での実行結果

char 型の範囲: -128 ~ 127

int 型の範囲: -2147483648 ~ 2147483647

long int 型の範囲: -2147483648 ~ 2147483647

FreeBSD 12.1 上の Clang 8.0.1 での実行結果

char 型の範囲: -128 ~ 127

int 型の範囲: -2147483648 ~ 2147483647

long int 型の範囲: -9223372036854775808 ~ 9223372036854775807

どちらの処理系でも、char 型は -128 から 127 の整数が入る型でした。文字と整数は解釈だけの違いであることを理解するために、次のソースコードをコンパイル、実行してみてください。授業のサイトには、adv03.c として掲載しています。

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char c = 'A';
6      char n = 65;
7
8      if (c == n)
9      {
10         printf("c == n は真\n");
11     }
12
13     printf("cを文字として見ると %c、整数として見ると %d\n", c, c);
14
15     printf("nを文字として見ると %c、整数として見ると %d\n", n, n);
16
17     return 0;
18 }
```


実行例

```
c == n は真
c を文字として見ると A、整数として見ると 65
n を文字として見ると A、整数として見ると 65
```

5 行目で、`char` 型の変数 `c` に文字定数 `'A'` を代入しています。6 行目では、`char` 型の変数 `n` に整数 `65` を代入しています。このように、`char` 型の変数に、文字定数ではなく整数を代入しても問題ありません。8 行目から 11 行目の `if` 文で `c == n` が真であることを確認しています。つまり、`'A'` と `65` は同じ値です。変数 `c` にも `n` にも `01000001` というビット列が格納されており、文字として見れば `"A"`、整数として見れば `65` です。解釈による違いを確認しているのが、13 行目と 15 行目の `printf()` 文です。ここでは、同じ変数を変換指定 `%c` を使って文字として表示し、さらに `%d` を使って整数として表示しています。なお、ほとんどの処理系では `"A"` は `65` ですが、そう決まっている訳ではありません。`"A"` が `65` であることなどを前提としたプログラムを書いてはいけません。

さて、表 14.4 の `toupper()` 関数の説明には、混乱を避けるため、あえて引数と戻り値の型を書きませんでした。文字なので `char` 型と思われるかもしれませんが、実はどちらも `int` 型です。C 言語の文字定数は `int` 型です。つまり `'A'` を評価すると、`int` 型の `65` が得られます。

`char c = 'A';` について考えてみます。多くの処理系では、`'A'` は `int` 型の `65` です。代入先の変数の型と代入する値の型が異なる場合でも、代入する値が、代入先の変数が扱える範囲に入っていれば、そのまま代入できます。`65` は `char` 型で表せる範囲に入っていますので、変数 `c` には `65` が入ります。`char c = (char)'A';` のような動作が行われていると考えてください。ソースコード中に直接書いた整数は `int` 型ですので^a、`char n = 65;` も `char n = (char)65;` のような動作が行われていることになります。

^a 第 7 章で説明しました。

第 15 章

関数の作成

何度も行う処理は関数にしておくと便利です。本章では関数の作り方を学びます。

15.1 関数定義

関数を作ることを、関数を**定義**するといいます。関数定義の書式と、関数定義の例を、図 15.1 に示します。例として作った `plus()` 関数は、`int` 型の引数を 2 つ受け取り、それらの和を戻り値として `int` 型で返します。

【関数定義の書式】

```
戻り値の型 関数名(引数の宣言※)
{
    /* 処理 */

    return 式;
}
```

【関数定義の例】

```
int plus(int a, int b)
{
    int c;

    c = a + b;

    return c;
}
```

※ “引数の型 引数の名前”を
コンマ区切りで並べる

図 15.1 関数定義の書式と例

関数定義の最初は戻り値の型です。`plus()` 関数の戻り値は `int` 型なので、`int` と書きます。
次は関数名です。`plus()` 関数の関数名は `plus` なので、`plus` と書きます。
次は引数の宣言です。() の中に、“引数の型 引数の名前”をコンマ区切りで並べます。引数の名前には、識別子として使える任意の名前^{*1}を指定できます。`plus()` 関数には、`int` 型の引数が

^{*1} 第 4 章で説明しました。

2 つ必要です。それぞれの名前を、a、b とする場合、引数の宣言は `int a, int b` となります。

次はブロックになっており、この部分に関数の処理を記述します。関数呼び出しの際に引数として渡された値は、“引数の宣言” に書いた変数に格納され、ブロックの中で使えます。`plus()` 関数を `plus(3, 6)` と呼び出した場合、変数 a と b が用意され、それぞれ 3 と 6 が入った状態になります。

`return 式;` は `return` 文です。`return` 文を実行すると、その関数の実行が終了し、関数呼び出しを行った場所に戻ります。その際、“式” を評価した値が、関数の戻り値となります。

`plus()` 関数では、`int` 型の変数 c を宣言し、`a + b` の値を代入しています。そして、`return` 文で、c の値を返しています。`plus()` 関数を `plus(3, 6)` と呼び出した場合、変数 c の値は 9 になり、戻り値は `int` 型の 9 です。

なお、図 15.1 の `plus()` 関数は、和を保存する変数を使う必要はありません。次のように書いても同じ動作です。

```
1  int plus(int a, int b)
2  {
3      return a + b;
4  }
```

関数によっては引数や戻り値が不要です。引数が不要な場合は、次のように、“引数の宣言”の部分に `void` と書きます。

```
1  int func1(void)
2  {
3      /* 以下略 */
```

戻り値が不要な場合は、次のように、“戻り値の型”の部分に `void` と書きます。

```
1  void func2(int a, int b)
2  {
3      /* 以下略 */
```

戻り値が不要な関数は、`return` 文に式を書かず、`return;` とします。

`return` 文は、関数の最後に書く必要はなく、1 つの関数内に複数あってもかまいません。練習課題 15(ex33.c) では、点数に応じて “S” ～ “D” の評語を表示しました。引数として点数を与えると、戻り値として評語を返す関数 `grade_letter()` の実装例を、ソースコード 15.1 に示します*2。

*2 この関数の戻り値の型は、`char` 型ではなく `int` 型とするのが一般的です。前章の発展を参照してください。

ソースコード 15.1 grade_letter() 関数の実装例 1

```
1 char grade_letter(int score)
2 {
3     if (score >= 90)
4     {
5         return 'S';
6     }
7     else if (score >= 80)
8     {
9         return 'A';
10    }
11    else if (score >= 70)
12    {
13        return 'B';
14    }
15    else if (score >= 60)
16    {
17        return 'C';
18    }
19    else
20    {
21        return 'D';
22    }
23 }
```

このように、関数の途中で `return` 文を書いてもかまいません。ただし、最後に 1 つだけある方がプログラムが読みやすくなることもあります。そのように変更したものを、ソースコード 15.2 に示します。

ソースコード 15.2 grade_letter() 関数の実装例 2

```
1 char grade_letter(int score)
2 {
3     char grade;
4
5     if (score >= 90)
6     {
7         grade = 'S';
```

```

8     }
9     else if (score >= 80)
10    {
11        grade = 'A';
12    }
13    else if (score >= 70)
14    {
15        grade = 'B';
16    }
17    else if (score >= 60)
18    {
19        grade = 'C';
20    }
21    else
22    {
23        grade = 'D';
24    }
25
26    return grade;
27 }

```

自作した関数の定義は、main() 関数の外に書きます。ソースコード中の main() 関数も関数定義です。複数の関数定義は、前後に並べます*3。

15.2 関数呼び出し

自作の plus() 関数を使ったプログラムの例を、ソースコード 15.3 に示します。

ソースコード 15.3 plus() 関数の使用例 (ex52.c)

```

1  #include <stdio.h>
2
3  int plus(int a, int b)
4  {
5      int c;
6
7      c = a + b;

```

*3 main() 関数の中に別の関数定義を書いたりしない、という意味です。

```

8
9     return c;
10 }
11
12 int main(void)
13 {
14     int n1, n2;
15
16     printf("整数 n1 の値 > ");
17     scanf("%d", &n1);
18     printf("整数 n2 の値 > ");
19     scanf("%d", &n2);
20
21     printf("n1 + n2 = %d\n", plus(n1, n2));
22
23     return 0;
24 }

```

実行例

```

整数 n1 の値 > 3 ↵
整数 n2 の値 > 6 ↵
n1 + n2 = 9

```

plus() 関数を main() 関数の上に書いた理由は次節で説明します。C 言語のプログラムは main() 関数から実行が始まります。それ以外の関数は、関数呼び出しを行わないと実行されません。書いた順に関数が実行されるわけではありません。

3 行目から 10 行目で plus() 関数を定義し、main() 関数内の 21 行目で呼び出しています。関数呼び出し式を評価すると、戻り値が得られます。plus(n1, n2) を評価して得られる n1 と n2 の和を、printf() 関数で表示しています。

自作関数の例をもう 1 つ紹介します。ソースコード 15.4 は、自作のべき乗関数 expo() の定義です。

ソースコード 15.4 自作のべき乗関数 expo()

```

1 double expo(double x, int y)
2 {
3     double e;
4

```

```

5      /* 1にxをy回かける */
6      for (e = 1 ; y > 0; --y)
7      {
8          e *= x;
9      }
10
11     return e;
12 }

```

底 (x^y の x) は小数、指数 (x^y の y) は負でない整数とします。それぞれ、`double` 型、`int` 型の引数として受け取ります。`expo()` 関数では、6 行目から 9 行目の `for` 文で、1 に x を y 回かけて、 x^y を求めています。指数が 0 の場合は、この `for` 文の式₁ を評価し、次に式₂ を評価して、`for` 文が終了します。式₁ の評価で、戻り値となる変数 `e` に 1 が代入されますので、指数が 0 の場合も正しく動作します*⁴。

なお、`plus()` 関数と `expo()` 関数は説明のために作っただけで、このような関数を作って使う理由はありません。`plus(3, 6)` は関数を使わずに `3 + 6` と書けばよいですし、`expo()` 関数は、より高機能な `pow()` 関数がライブラリ関数として用意されています。

練習課題 25 (ex53.c)

ソースコード 15.4 に示した自作のべき乗関数 `expo()` を使って、 $(-2.7)^3$ と 1.8^4 を計算し、表示するプログラムを作ってください。`main()` 関数以外を、授業のサイトに `ex53.c` として掲載しています。これに `main()` 関数を追加して、プログラムを完成させてください。なお、`main()` 関数は、`expo()` 関数よりも下にご書いてください。

実行例

-2.7 の 3 乗は -19.683000 です。
1.8 の 4 乗は 10.497600 です。

15.3 関数プロトタイプ宣言

関数は、呼び出す場所よりも前 (ソースコード中で上) に定義されている必要があります。ソースコード 15.3 では、`main()` 関数内で `plus()` 関数を使っていますが、その場所よりも前に `plus()` 関数を定義しています。コンパイルの際、コンパイラは、ソースコードを上から順に読んでいきます。ソースコード 15.3 の 21 行目で `plus()` 関数が出てきたとき、コンパイラは、`plus()` 関数が

*⁴ ただし、0 の 0 乗は 1 となります。

int 型の引数を 2 つ受け取って int 型の戻り値を返す関数であることを知っています。関数定義が後ろに書いてあると、コンパイラが関数呼び出しの部分を読んだ時点では関数の仕様がわからず、正しく処理することができません (図 15.2)。main() 関数の中でライブラリ関数や自作の関数を呼び出したように、自作の関数の中でも別の関数を呼び出すことができます。その場合は、呼び出される関数の定義を前に書きます。

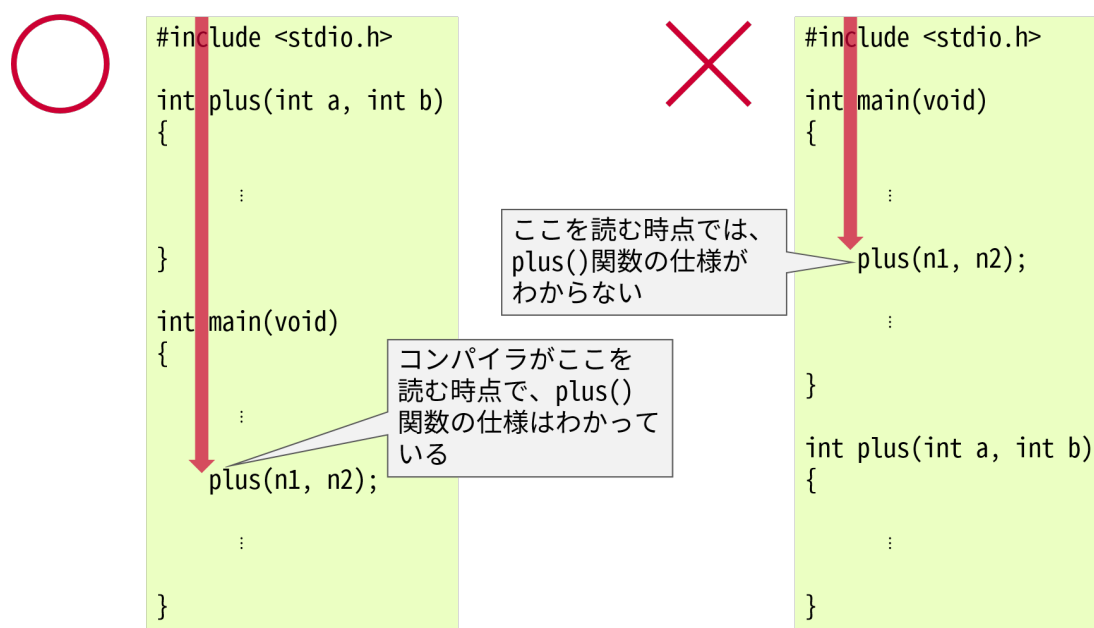


図 15.2 関数の定義順とコンパイラの処理

大きなプログラムでは、関数の呼び出し関係を整理して、順番に書くことは大変です。また、main() 関数は最後に書かれることになりますが、最初に書いた方がわかりやすいこともあります。関数呼び出しの前に**関数プロトタイプ宣言**を行えば、関数呼び出しを関数定義の前に書くことができます。関数プロトタイプ宣言は、関数を使うに当たって、引数の個数と型や、戻り値の型を知らせる役割を持っています。関数プロトタイプ宣言の書式は、

戻り値の型 関数名 (引数の宣言);

です。つまり、関数定義からブロックを削除し、最後に ; を付けます。たとえば、plus() 関数の関数プロトタイプ宣言は、

int plus(int a, int b);

となります。関数プロトタイプ宣言では、変数名は省略することができ、普通は省略されます。つまり、一般的には

int plus(int, int);

と宣言します。

ソースコード 15.3 を、main() 関数を前に書くように変更したものをソースコード 15.5 に、コンパイラの処理のイメージを図 15.3 に示します。

ソースコード 15.5 関数プロトタイプ宣言の使用 (ex54.c)

```
1  #include <stdio.h>
2
3  /* 関数プロトタイプ宣言 */
4  int plus(int, int);
5
6  int main(void)
7  {
8      int n1, n2;
9
10     printf("整数 n1 > ");
11     scanf("%d", &n1);
12     printf("整数 n2 > ");
13     scanf("%d", &n2);
14
15     printf("n1 + n2 = %d\n", plus(n1, n2));
16
17     return 0;
18 }
19
20 int plus(int a, int b)
21 {
22     int c;
23
24     c = a + b;
25
26     return c;
27 }
```

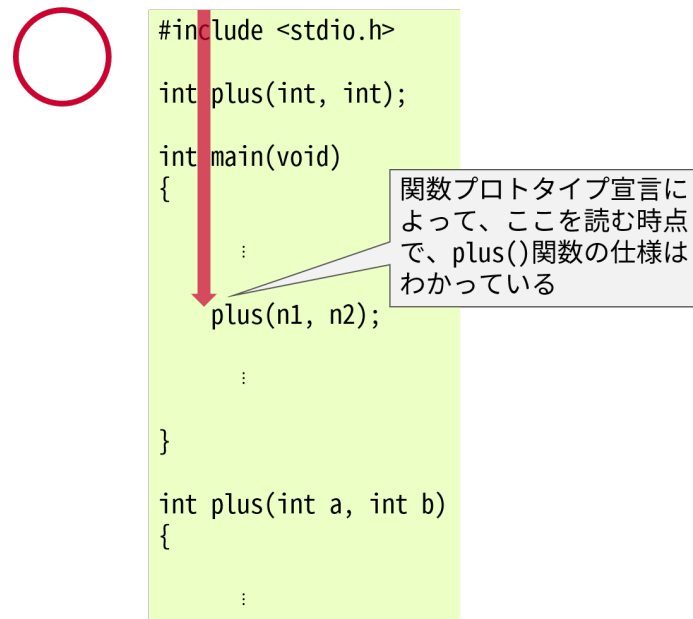


図 15.3 関数プロトタイプ宣言とコンパイラの処理

練習課題 26 (ex55.c)

int 型の引数を 1 つ受け取り、それに 5 を足した数を int 型として返す関数 plus5() を作ってください。plus5() 関数を呼び出す main() 関数と、plus5() 関数の一部を、授業のサイトに ex55.c として掲載しています。コメント部分に plus5() 関数の処理を書いて、正しく動作するプログラムを作ってください。

実行例 1

整数を入力してください > 3 ↵
3 に 5 を足すと 8 です。

実行例 2

整数を入力してください > -8 ↵
-8 に 5 を足すと -3 です。

練習課題 27 (ex56.c)

ライブラリ関数の abs() と同じ働きをする関数 myabs() を作ってください。myabs() 関数は、int 型の引数を 1 つ受け取り、その絶対値を int 型で返します。myabs() 関数を呼

び出す `main()` 関数を、授業のサイトに `ex56.c` として掲載しています。`myabs()` 関数の関数プロトタイプ宣言と定義を記述して、正しく動作するプログラムを作ってください。

実行例 1

整数を入力してください > -5 ↵

-5 の絶対値は 5 です。

実行例 2

整数を入力してください > 3 ↵

3 の絶対値は 3 です。

練習課題 28 (`ex57.c`)

`int` 型の引数を 1 つ受け取り、それが素数であれば 1 を、素数でなければ 0 を返す関数 `is_prime()` を作ってください。`is_prime()` には、3 以上の整数が渡されることを前提にしてください。`is_prime()` 関数を呼び出す `main()` 関数を、授業のサイトに `ex57.c` として掲載しています。`is_prime()` 関数の定義を記述して、正しく動作するプログラムを作ってください。

3 以上の整数 N が素数であるかは、次のようにすれば判定できます。

- 2 で割り切れれば、 N は素数ではない。
- 3 で割り切れれば、 N は素数ではない。
- \vdots
- $N - 1$ で割り切れれば、 N は素数ではない。
- 2 ~ $N - 1$ のいずれでも割り切れなければ、 N は素数である。

実行例 1

3 以上の整数を入力してください > 437 ↵

437 は素数ではありません。

実行例 2

3 以上の整数を入力してください > 439 ↵

439 は素数です。

15.4 main() 関数、printf() 関数、scanf() 関数

main() 関数は特別な関数で、C 言語のプログラムは main() 関数から始まります。main() 関数のないプログラムを実行することはできません。この授業では、次のような main() 関数を使ってきました。

```
1  int main(void)
2  {
3      /* 必要な命令をここに記述する */
4
5      return 0;
6  }
```

この main() 関数は、引数の宣言が void なので、引数を受け取りません。戻り値の型が int で、return 文が `return 0;` であることから、int 型の 0 を返します。この授業ではすべてこの形式でしたが、main() 関数は、引数を受け取ったり、0 以外の値を返したりすることもできます。main() 関数の引数とは何で、戻り値がどこに渡されどのように使われるのかは、この授業では省略します。

また、前章の最初に、printf() と scanf() は少し変わった関数と書きました。どのように変わっているかを説明します。

関数は引数を与えて呼び出します。引数の数とそれぞれの型は、関数ごとに決められています。たとえば、pow() 関数には引数が 2 つ必要で、どちらも double 型です。普通の関数は、このように、引数の個数とそれぞれの型が決められています。printf() 関数は、第 1 引数が文字列リテラルです。ここに変換指定が含まれている場合、第 2 引数以降に対応する式を書きます。printf() 関数は、引数の個数と、第 2 引数以降の型が固定ではありません。この授業では説明しませんでした。scanf() 関数も 3 つ以上の引数を指定することができます。引数が可変個である関数は特殊で、この授業の範囲では、そのような関数を作ることはできません。

関数呼び出しを行うと、その実行結果である戻り値が得られます。実は、printf() 関数と scanf() 関数は、int 型の戻り値を返しています。実用的なプログラムでは、エラーが起こっていないかを確認するため、scanf() 関数の戻り値を使います。printf() 関数は、戻り値が使われることはほとんどありません。printf() 関数は画面にデータを表示する機能が主で、戻り値はおまけ程度です。scanf() 関数はキーボードから読み取ったデータを変数に格納する機能が主です。引数としてデータを渡して、その処理結果を戻り値として受け取るという関数ではない点が特殊といえます。

さらに、scanf() 関数は、第 2 引数に指定した変数の値を書き換えます。キーボードから読み取った値を戻り値として返してくれれば、`int i = scanf(...);` のように変数に代入できます。こちらの方が自然と思えますが、そのような仕様にはなっていません。変数の値を直接書き換

えてしまう関数は、特殊といえます。

第 16 章

引数の受け渡し

16.1 値渡し

前章で作った自作関数 `plus()` を、ソースコード 16.1 に示します。

ソースコード 16.1 自作関数 `plus()` の定義

```
1  int plus(int a, int b)
2  {
3      int c;
4
5      c = a + b;
6
7      return c;
8  }
```

`plus()` 関数は、引数として `int` 型の値を 2 つ受け取ります。受け取った引数は、それぞれ変数 `a`、`b` に格納され、関数内で普通の変数として使うことができます。関数を呼び出す側で指定する引数を^{じつひきすう}実引数といいます。また、呼び出される関数で、実引数を受け取る変数を^{かりひきすう}仮引数といいます。`plus()` 関数を、`plus(3, 6)` と呼び出したとすると、3 と 6 が実引数です。仮引数は、`plus()` 関数の `a` と `b` です (図 16.1)。

この関数呼び出しを行うと、1 つ目の実引数の値である 3 が、仮引数の変数 `a` にコピーされます。同様に、2 つ目の実引数の値である 6 が、仮引数の変数 `b` にコピーされます。実引数に変数である場合も同様です。図 16.2 のように、実引数の変数の値が、仮引数の変数にコピーされます。

変数そのものを渡しているのではなく、変数が持つ値を渡しています。このような引数の渡し方を**値渡し**といいます。C 言語の関数呼び出しは、値渡しです。ソースコード 16.2 をコンパイル、実行して、値渡しが行われていることを確認してみましょう。

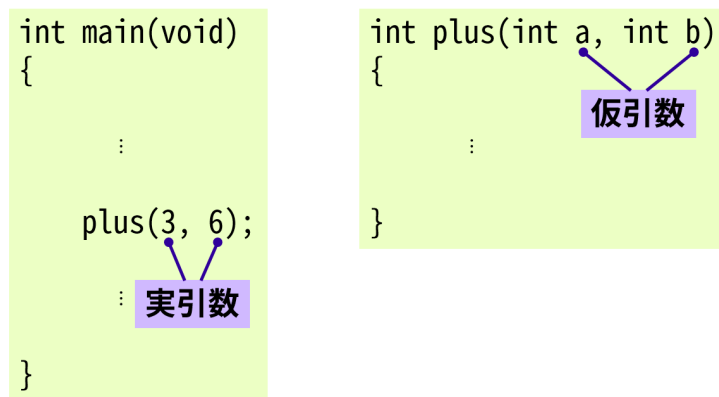


図 16.1 実引数と仮引数

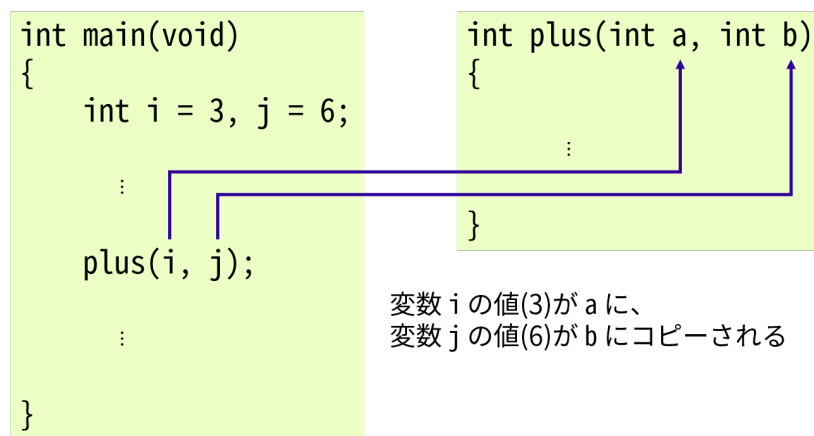


図 16.2 値の受け渡し

ソースコード 16.2 値渡しの確認 (ex58.c)

```

1  #include <stdio.h>
2
3  void func(int);
4
5  int main(void)
6  {
7      int n = 1;
8
9      printf("(1) n = %d\n", n);
10
11     func(n);
12

```

```

13     printf("(4) n = %d\n", n);
14
15     return 0;
16 }
17
18 void func(int a)
19 {
20     printf("(2) a = %d\n", a);
21
22     a = 2;
23
24     printf("(3) a = %d\n", a);
25
26     return;
27 }

```

実行結果

```

(1) n = 1
(2) a = 1
(3) a = 2
(4) n = 1

```

このプログラムの中には `printf()` 文が 4 つあり、それぞれ 1 回ずつ実行されます。最初は、`main()` 関数内にある 9 行目の `printf()` 文です。変数 `n` の値は 1 なので、“(1) n = 1” と表示します。その後、11 行目で自作関数の `func()` を呼び出しています。実引数として `n` を指定し、関数 `func()` の仮引数 `a` で受け取ります。`n` の値は 1 なので、仮引数 `a` には 1 がコピーされます。関数 `func()` 内では、20 行目の `printf()` 文で、`a` の値を使って、“(2) a = 1” と表示します。次に、22 行目の代入文で `a` の値を 2 に変更し、24 行目の `printf()` 文で、`a` の値を使って、“(3) a = 2” と表示します。26 行目の `return` 文で関数 `func()` の実行が終了し^{*1}、`main()` 関数内の 11 行目の次に進みます。この関数呼び出しでは、実引数として `n` を指定しました。それを受け取った仮引数 `a` の値は、2 に変更されています。関数呼び出しでは `n` の値を渡しただけですので、仮引数の値が変更されても、実引数に指定した変数の値が変更されるわけではありません。13 行目の `printf()` 文では、“(4) n = 1” と表示され、`n` の値が変更されていないことを確認できます。

^{*1} 関数の最後 (ブロックを閉じる “}”) まで進むと関数の実行が終了しますので、この場合は `return` 文を書く必要はありません。

16.2 変数のスコープ

これまでのプログラムでは、ソースコード内で変数名が重複しないようにしてきました。しかし、大きなプログラムでは重複しないようにすることは大変です。ソースコード 16.2 では、`main()` 関数内で変数 `n` を、`func()` 関数の仮引数として変数 `a` を使っています。ソースコード 16.3 のように、どちらも `n` にしたらどうなるでしょうか。

ソースコード 16.3 変数名の重複 (ex59.c)

```
1  #include <stdio.h>
2
3  void func(int);
4
5  int main(void)
6  {
7      int n = 1;
8
9      printf("(1) n = %d\n", n);
10
11     func(n);
12
13     printf("(4) n = %d\n", n);
14
15     return 0;
16 }
17
18 void func(int n)
19 {
20     printf("(2) n = %d\n", n);
21
22     n = 2;
23
24     printf("(3) n = %d\n", n);
25
26     return;
27 }
```

実行結果

(1) n = 1
(2) n = 1
(3) n = 2
(4) n = 1

実行結果を見ると、2 行目と 3 行目で `a` が `n` に変わっただけです。`main()` 関数内で定義された変数 `n` と、`func()` 関数の仮引数として定義された変数 `n` は別のものと見なされます。図 16.3 のように、`int` 型の値を入れるための異なる箱が用意され、どちらにも `n` という名前が付けられていると考えてください。

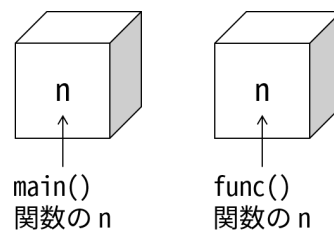


図 16.3 同名の変数

`main()` 関数内で定義された変数 `n` は、`main()` 関数内だけで有効です。`func()` 関数の仮引数として定義された変数 `n` は、`func()` 関数内だけで有効です。

変数は、その有効範囲が決められています。“有効範囲”は原語 (英語) では **スコープ** (scope) で、日本語でも普通はスコープと呼ばれます。

変数のスコープに関する規則は、次のとおりです。

- 変数はブロックの先頭で宣言します。変数のスコープは、宣言されたブロック内です。
- 異なるブロック (スコープ外) からは、変数を参照することができません (図 16.4 左)。
- ブロックが異なれば、同じ変数名を使うことができます (図 16.4 右)。
- ブロックがネストしている場合、内側のブロックもスコープに含まれます (図 16.5 左)。
- 外側のブロックで宣言した変数と同じ名前の変数を、内側のブロックで宣言することができます。その場合、内側のブロックに書いた変数名は、内側のブロックで宣言された変数を指します。内側のブロックからは、外側のブロックで宣言した変数が見えなくなります (図 16.5 右)。
- 関数の仮引数のスコープは、その関数内です。

変数のスコープを確かめるプログラムを、ソースコード 16.4 に示します。このプログラムでは、3 個の異なる変数 `n` を使っており、6 箇所その値を表示しています。実行結果を見る前に、どのような表示になるか考えてみてください。

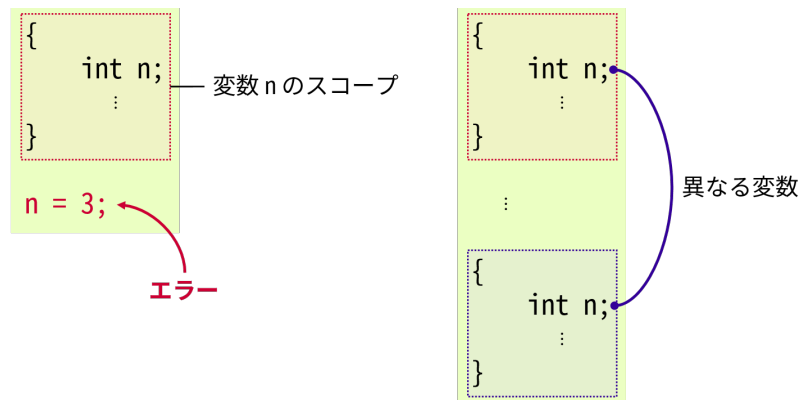


図 16.4 変数のスコープ 1

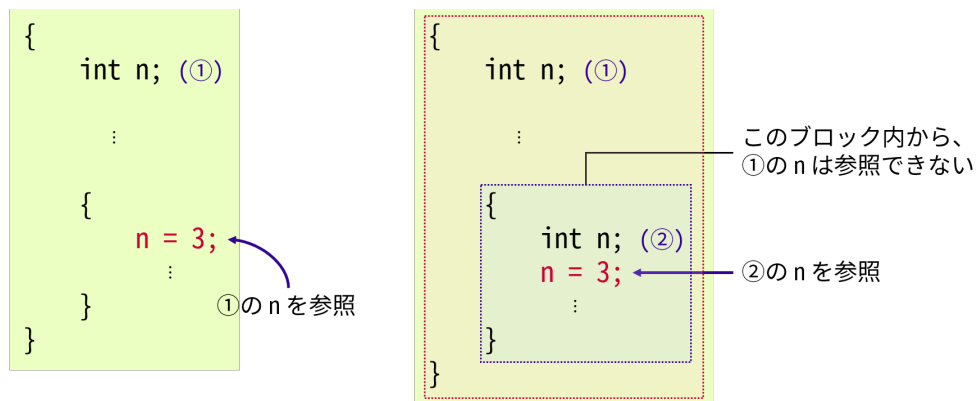


図 16.5 変数のスコープ 2

ソースコード 16.4 変数のスコープの確認 (ex60.c)

```

1  #include <stdio.h>
2
3  void func(int);
4
5  int main(void)
6  {
7      int n = 1;
8
9      {
10         int n = 2;
11         printf("%d\n", n);
12         n = 3;
13     }

```

```

14
15     {
16         printf("%d\n", n);
17         n = 4;
18     }
19
20     printf("%d\n", n);
21
22     func(n);
23
24     printf("%d\n", n);
25
26     return 0;
27 }
28
29 void func(int n)
30 {
31     printf("%d\n", n);
32
33     n = 5;
34
35     printf("%d\n", n);
36
37     return;
38 }

```

⋮

実行結果

```

2
1
4
4
5
4

```

第 17 章

コンパイル

本章では、コンパイラが行う処理を簡単に学びます。

17.1 コンパイルの流れ

コンパイラは、ソースコードをコンパイルし、機械語のプログラムに変換します。一般的なコンパイラの処理は、大きく分けて、図 17.1 に示す 4 つの工程に分けられます。

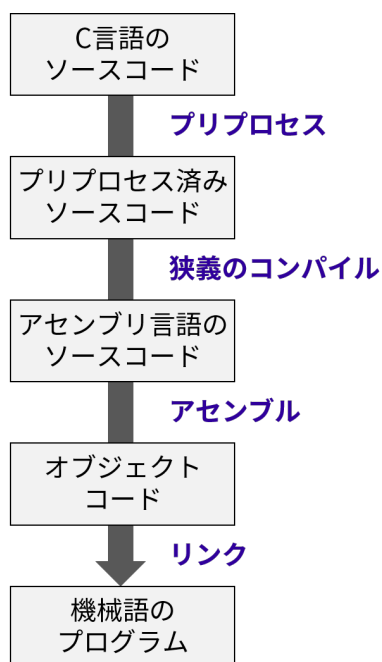


図 17.1 コンパイルの大まかな流れ

1 つ目の工程は、**プリプロセス**です。プリプロセスを行うプログラムを、**プリプロセッサ**といいます。この工程によって、C 言語のソースコードが、プリプロセス済みのソースコードに変換されます。

2つ目の工程は、**狭義のコンパイル**です。狭義のコンパイルを行うプログラムを、(狭義の) **コンパイラ**といいます。この工程によって、プリプロセス済みのソースコードが、アセンブリ言語のソースコードに変換されます。

3つ目の工程は、**アセンブル**です。アセンブルを行うプログラムを、**アセンブラ**といいます。この工程によって、アセンブリ言語のソースコードが、オブジェクトコードに変換されます。

4つ目の工程は、**リンク**です。リンクを行うプログラムを**リンカ**といいます。この工程によって、オブジェクトコードなどをもとに、機械語の実行プログラムが生成されます。

“コンパイル”という用語は、機械語への変換工程全体を指すことがほとんどですが、狭義では1つ目と2つ目の工程か、2つ目の工程のみを指します。“コンパイラ”も同様です。

17.2 プリプロセス

プリプロセスとは、狭義のコンパイルを行う前の、C言語のソースコードに対する前処理です。

たとえば、コメントは人間が読むためのもので、コンパイルには必要ありません。コメントは、プリプロセッサによって削除されます。

から始まる行を**プリプロセッサディレクティブ**と呼ぶことを説明しました。プリプロセッサディレクティブは、プリプロセッサに対する指令 (directive) です。ここでは、**#include** ディレクティブと**#define** ディレクティブについて説明します。

#include ディレクティブ

#include ディレクティブは、ヘッダのインクルードを指示します。プリプロセッサは、**#include** ディレクティブによって指定されたヘッダを、**#include** ディレクティブの書かれた位置に挿入します。ソースコード中の

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

は、プリプロセッサによって、**<stdio.h>**に対応するヘッダに置き換えられます。通常、ヘッダは、ファイルとしてコンピュータ内に格納されています。MinGW を **C:\MinGW** にインストールした場合、**<stdio.h>**に対応するヘッダファイルは、**C:\MinGW\include\stdio.h** です。Unix 系の OS では、**/usr/include/stdio.h** にあることが多いと思います。ヘッダファイルも C 言語のソースファイルで、テキストエディタで開くことができます。

ここで、関数プロトタイプ宣言を思い出してください。関数は、使う前に定義されているか、関数プロトタイプ宣言が行われている必要があります。ライブラリ関数は定義や宣言なしで使えましたが、特別扱いされているわけではありません。実は、ライブラリ関数は、ヘッダの中で、関数が定義されているか関数プロトタイプ宣言が行われています。

MinGW のヘッダファイルは難しく書かれているので、FreeBSD 12.1 上の Clang 8.0.1 のヘッダファイルを見てみます。たとえば、**pow()** 関数を使うには、**<math.h>**のインクルードが必要です。**<math.h>**に対応するヘッダファイルには、次のように書かれています。

ソースコード 17.1 FreeBSD 12.1 の/usr/include/math.h の一部

```

246 double cosh(double);
247 double sinh(double);
248 double tanh(double);
249
250 double exp(double);
251 double frexp(double, int *); /* fundamentally !__pure2 */
252 double ldexp(double, int);
253 double log(double);
254 double log10(double);
255 double modf(double, double *); /* fundamentally !__pure2 */
256
257 double pow(double, double);
258 double sqrt(double);
259
260 double ceil(double);
261 double fabs(double) __pure2;
262 double floor(double);
263 double fmod(double, double);
264
265 /*
266  * These functions are not in C90.
267  */
268 #if __BSD_VISIBLE || __ISO_C_VISIBLE >= 1999 || __XSI_VISIBLE

```

257 行目に `pow()` 関数の関数プロトタイプ宣言があります。そのため、`<math.h>` をインクルードすることによって、`pow()` 関数が使えます。`pow()` 関数を使うためだけであれば、`<math.h>` の中で必要な部分は、`pow()` 関数の関数プロトタイプ宣言だけです。

```
#include <math.h>
```

と書く代わりに

```
double pow(double, double);
```

と書いても動作しますので、試してみてください。ただし、そのようにすると、別の処理系では正しく動作しなくなる可能性があります。たとえば、別の処理系では、ヘッダに関数プロトタイプ宣言が書かれているのではなく、関数定義自体が書かれているかもしれません。通常は、`<math.h>` をインクルードしてください。

printf() 関数や scanf() 関数の定義、または関数プロトタイプ宣言は、<stdio.h>に書かれています。

#define ディレクティブ

#define ディレクティブは、マクロ定義を行います。

```
#define マクロ名 文字の並び
```

と書くと、プリプロセッサによって、ソースコード中の“マクロ名”が“文字の並び”に置き換えられます。たとえば、

```
#define NUMBER 5
```

と書くと、ソースコード中の NUMBER が 5 に置き換えられます。

```
for (i = 0; i < NUMBER; ++i)
```

は、プリプロセッサによって

```
for (i = 0; i < 5; ++i)
```

と書き換えられます。

ソースコード 17.2 に、#define ディレクティブの使用例を示します。

ソースコード 17.2 #define ディレクティブの使用例 (ex61.c)

```
1  #include <stdio.h>
2
3  #define CONSUMPTION_TAX_RATE 10
4
5  int main(void)
6  {
7      double price;
8
9      printf("税抜き価格 > ");
10     scanf("%lf", &price);
11
12     price *= 1 + (double)CONSUMPTION_TAX_RATE / 100;
13
14     printf("税込み価格は%.0f円です.\n", price);
15
16     return 0;
```


17 }

実行例

税抜き価格 > 1500 ↩
税込み価格は 1650 円です。

3 行目の `#define` ディレクティブで、消費税率を

```
#define CONSUMPTION_TAX_RATE 10
```

と定義しています。ソースコード中の `CONSUMPTION_TAX_RATE` は 10 に置き換えられますので、プリプロセッサによって、12 行目は

```
price *= 1 + (double)10 / 100;
```

と書き換えられます。消費税率が変わった場合は、`#define` ディレクティブを書き換えます。この例では `CONSUMPTION_TAX_RATE` を 1 回しか使っていませんが、複数回使っている場合でも、修正は 1 箇所だけですみます。また、`CONSUMPTION_TAX_RATE` を使うことによって、12 行目で税抜き価格から税込み価格に変更していることがわかり、プログラムが読みやすくなります。なお、この行は、複合代入演算子、算術演算における型の規則、キャスト演算子、演算子の優先順位などを理解していないと正しく読めません。難しく感じる場合は、該当箇所を復習してください。

17.3 狭義のコンパイル以降

プリプロセスによって、プリプロセス済みのソースコードが得られます。狭義のコンパイラは、そのソースコードをアセンブリ言語のソースコードに変換します。アセンブリ言語とは、プログラミング言語の一種です。機械語は、人間にとっては意味のないデータの並びで、人間が読み書きすることには適していません。アセンブリ言語は、機械語の命令を、人間が読みやすい形式で書けるようにしたプログラミング言語です。C 言語から機械語に変換するために、アセンブリ言語を経由するのであれば、最初からアセンブリ言語で書けばよいのではないかと思われるかもしれませんが、しかし、アセンブリ言語はコンピュータの動作を理解しないと使えない難しい言語です。長いプログラムの記述にも適していません。また、アセンブリ言語の記述方法は、コンピュータの種類によって違います。C 言語であれば、同じソースコードを異なる種類のコンピュータで使うことができます*1。

次の工程はアセンブルです。アセンブリ言語のソースコードが、アセンブラによって、オブジェクトコードに変換されます。オブジェクトコードとは、実行ファイルの一手手前の状態と考えてください。

*1 コンパイラが、そのコンピュータで動作する機械語のプログラムに変換します。

最後の工程はリンクです。リンカは、オブジェクトコードや、ライブラリと呼ばれるプログラム部品などを結びつけて、実行ファイルを作成します。ここで、`pow()` 関数を使うプログラムを考えます。`<math.h>`に `pow()` 関数の関数プロトタイプ宣言が書かれていれば、プログラム内で `pow()` 関数を使えます。しかし、`pow()` 関数を実行するためには、その実体が必要です。`pow()` 関数の実体は、ライブラリなどの形で用意されており、オブジェクトコードには含まれていません。そこで、実行ファイルに `pow()` 関数の実体を含めるか、`pow()` 関数の実体を呼び出せるようにしなければなりません。リンクでは、そのような作業も行います。第 14 章で、`<math.h>`のインクルードが必要な関数を使う場合、コンパイルのコマンドに `-lm` が必要な処理系があると説明しました。これは、特定のリンカに対するオプションで、“m” という名前を持ったライブラリを使うことを指定します。そのライブラリの中に `pow()` 関数の実体が含まれており、リンカは、実行ファイルがそれを読み出すようにします。

17.4 4 工程の確認

ここまでで、コンパイルの 4 つの工程を簡単に説明しました。GCC は、複数のプログラムを組み合わせてコンパイルを行います。次のように、コンパイルのコマンドに `-v` を付けると、それらのプログラムを読み出す様子を確認できます。

MinGW で `gcc -o ex01.exe ex01.c -v` を実行したときの出力

```
⋮
c:/mingw/bin/./libexec/gcc/mingw32/8.2.0/cc1.exe -quiet ...
⋮
c:/mingw/bin/./lib/gcc/mingw32/8.2.0/././././././mingw32/bin/as.exe -v
...
⋮
c:/mingw/bin/./libexec/gcc/mingw32/8.2.0/collect2.exe -plugin ...
⋮
```

まず、`cc1.exe` でプリプロセスと狭義のコンパイルが行われます。次に `as.exe` でアセンブルが、最後に `collect2.exe` でリンクが行われます。

GCC は、コンパイルを途中の工程で止めることもできます。`ex01.c` のコンパイルを例に、それぞれの工程での生成物を確認してみましょう。なお、以下は MinGW における操作例です。

プリプロセスだけを行うには、

```
gcc -E -o ex01.i ex01.c
```

と入力します。`ex01.i` という名前のファイルが出力されます。これが、プリプロセス済みのソースコードです。テキストエディタで開くと、一番下に `main()` 関数があることがわかります。それ

より上は、`#include` ディレクティブによって`<stdio.h>`がインクルードされた結果です。

`ex01.i` に対して狭義のコンパイルを行うには、

```
gcc -S ex01.i
```

と入力します。`ex01.s` という名前のファイルが出力されます。これが、アセンブリ言語のソースコードです。こちらテキストエディタで開くことができますので、見てみましょう。内容を理解する必要はありませんが、次のような行が書かれていれば注目してください。

```
call    _puts
```

これは、`puts()` というライブラリ関数の呼び出しです。`puts()` 関数は文字列と改行を出力する機能を持っています。`ex01.c` では文字列を出力するために `printf()` 関数を使いましたが、`puts()` 関数を使うコードに書き換えられてしまいました。`puts()` 関数は、`printf()` 関数のような変換指定を行える機能を持っていませんが、より高速に動作します。`ex01.c` で使った `printf()` 関数では変換指定を行っていないので、同等の処理をより高速に実現できる `puts()` 関数を使うように書き換えられました。このように、コンパイラはより高速に動作するように、**最適化**の処理を行っています。

`ex01.s` をアセンブルするには、

```
gcc -c ex01.s
```

と入力します。`ex01.o` という名前のファイルが出力されます。これが、オブジェクトコードが格納されたオブジェクトファイルです。このファイルはバイナリファイルであり、テキストエディタで開くことはできません。実行することもできません。

最後に、`ex01.o` をリンクして実行ファイルを作るためには、

```
gcc -o ex01.exe ex01.o
```

と入力します。`ex01.exe` という名前のファイルが出力されます。これが実行ファイルで、実行すると“こんにちは”と表示されます。

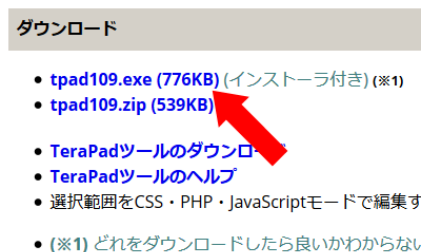
付録 A

TeraPad のインストールと使い方

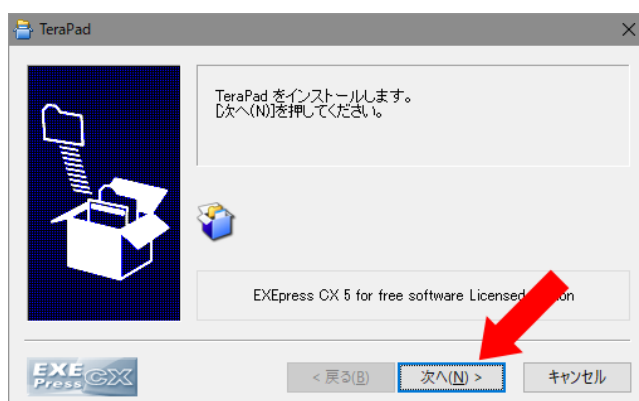
A.1 TeraPad のインストール

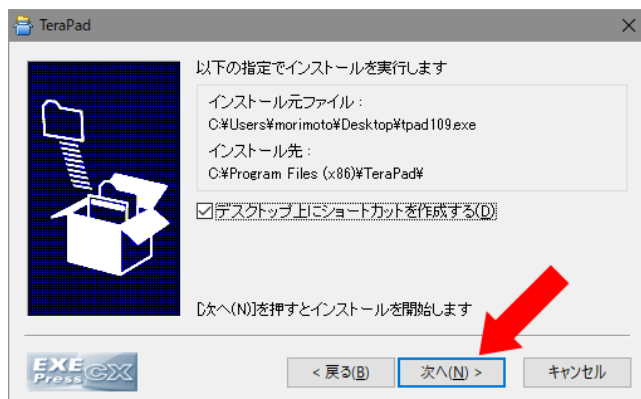
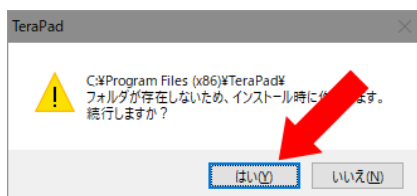
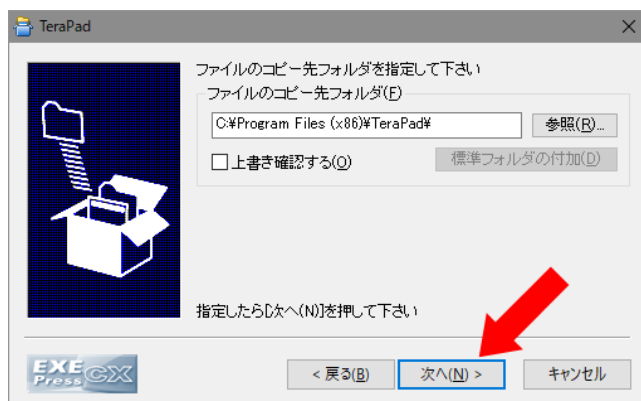
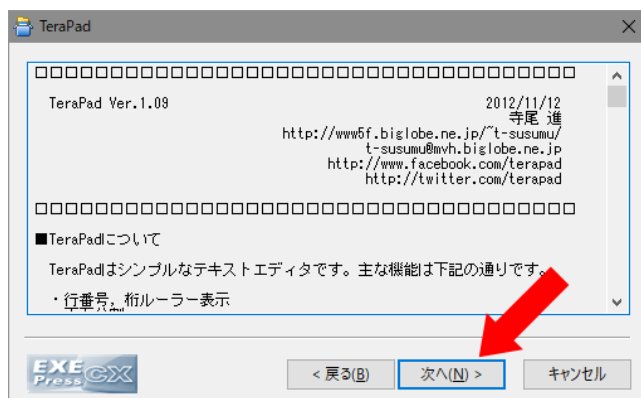
Windows 10 に TeraPad をインストールする手順を紹介します。PC の環境や TeraPad のバージョンによって、画面の表示内容などが多少異なることがあります。インストールには、管理者の権限が必要です。

1. TeraPad の Web サイトである <https://tera-net.com/library/tpad.html> から、インストーラをダウンロードします。下の図では、“tpad109.exe (776KB)” です。



2. ダウンロードしたインストーラを実行します。画面の指示に従って、“次へ (N) >”、または“はい (Y)” を選択して、インストールを進めてください。





これで、TeraPad のインストールは終了です。

A.2 TeraPad の使い方

ここでは、C 言語のソースファイルを編集するための、TeraPad の使い方を紹介します。ファイルの開き方やテキストの編集方法など、一般的な操作の説明は省略します。

C 言語のソースファイルは、テキストファイルです。ただし、拡張子は “.txt” ではなく、“.c” にすることが一般的です。新規に C 言語のソースコードを書き始める場合は、まず編集モードを C 言語用に変更します。メニューバーの“表示 (V)” → “編集モード (M)” を “C/C++(C)” に変更してください (図 A.1)。編集モードが“標準”の場合を図 A.2 に、“C/C++” の場合を図 A.3 に示します。図 A.3 では、C 言語用のシンタックスハイライトが行われていることがわかります。拡張子が .c のファイルを開いた場合は、自動的に編集モードが C/C++ になります。

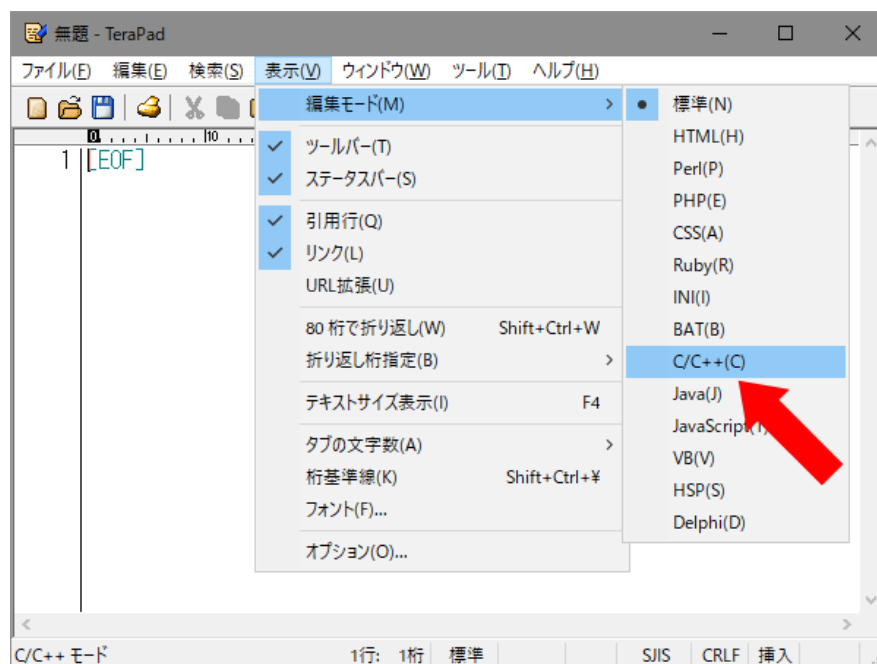


図 A.1 編集モードの変更

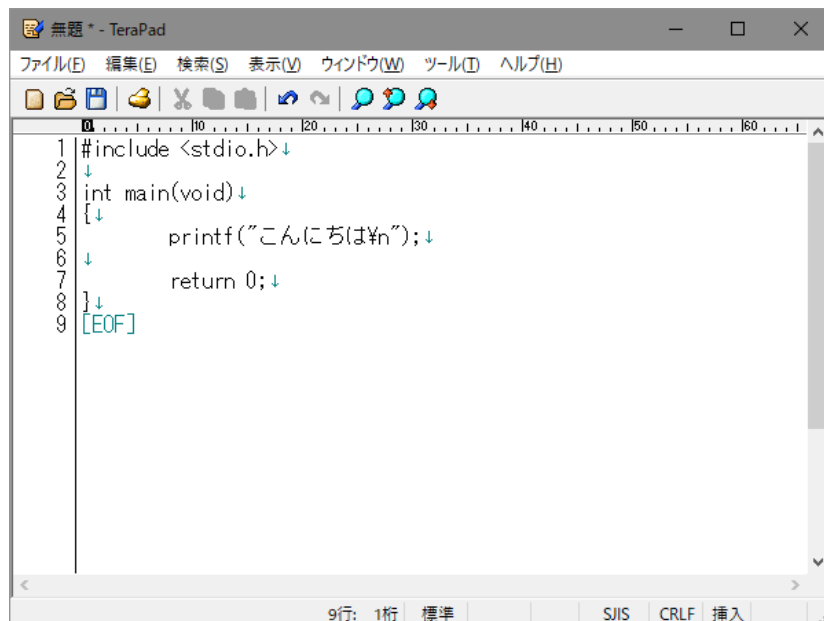


図 A.2 編集モードが“標準”の場合

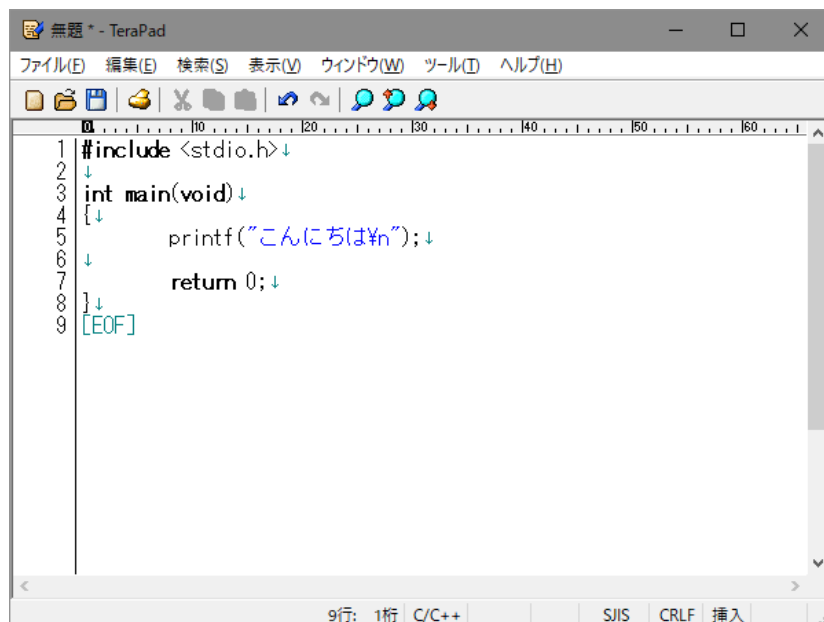


図 A.3 編集モードが“C/C++”の場合

ソースファイルを保存するときは、拡張子を.c にします。図 A.4 のように、ファイルの種類が“C/C++ ファイル (*.c,*.cpp,*.h)” になっていることを確認してください。ファイル名は、英数字だけにしておくが無難です。

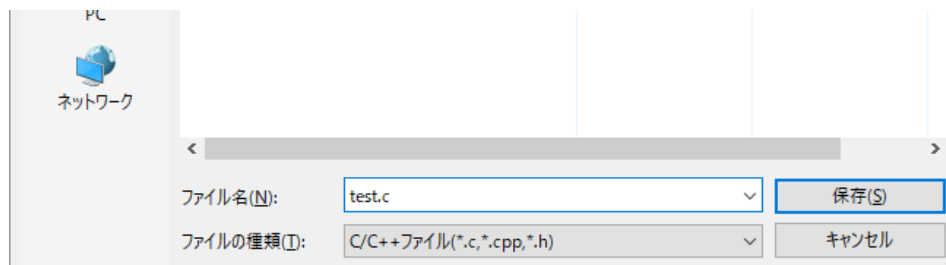


図 A.4 “名前を付けて保存” ダイアログボックス

新規に作成した場合は、正しい文字コードと改行コードを指定して保存する必要があります。TeraPad の標準は、文字コードが Shift_JIS、改行コードが CR+LF です。Windows のコマンドプロンプトを使う場合^{*1}は、標準のままだが正しい設定です。そうでない場合は、メニューバーの“ファイル (F)” → “文字/改行コード指定保存 (K)...” で、正しい文字コードと改行コードを指定して保存してください。ほとんどの環境では、次のいずれかの組み合わせです。

- 文字コードが Shift_JIS、改行コードが CR+LF
- 文字コードが UTF-8(BOM なし)、改行コードが LF

TeraPad の“文字/改行コード指定保存” ダイアログボックスでは、Shift_JIS は“SHIFT-JIS”、UTF-8(BOM なし) は“UTF-8N”と表記されています (図 A.5)。

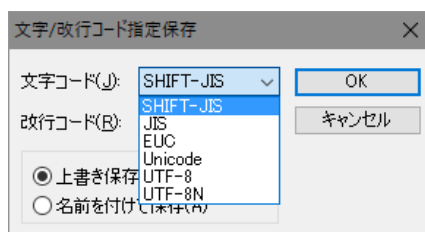


図 A.5 “文字/改行コード指定保存” ダイアログボックス

現在の編集モード、文字コード、改行コードは、ステータスバーで確認できます。図 A.6 の例では、編集モードが C/C++、文字コードが Shift_JIS (“SJIS” と表記)、改行コードが CR+LF です。

^{*1} 付録 B で紹介します。



図 A.6 TeraPad のステータスバー

A.3 TeraPad の便利な設定

メニューバーの“表示(V)” → “オプション(O)...”で、オプションの設定画面が開きます。必要に応じて、表 A.1 の設定を行うと便利です。

表 A.1 TeraPad のオプション

タブ*2	オプション
基本	標準では、タブストップ*3は 8 桁ごとです。C 言語のソースコードを読みやすくするために、行の先頭にタブをいくつか挿入することが一般的です。8 桁の場合スペースが空きすぎると感じる場合は、“タブの文字数 (T)”を 4 程度に変更してください。
表示	タブを挿入しても何も表示されないため、スペースと区別が付きません。タブを挿入した場所にマークを表示したい場合は、“TAB(T)”をチェックしてください。また、ソースコードに全角の空白が紛れ込んでいる場合、コンパイル時にエラーになります。全角の空白も何も表示されませんので、エラーの原因に気づきづらいことがあります。“全角空白 (Z)”をチェックすると、全角の空白の場所にマークが表示されますので、すぐに気づけます。
文字コード	Shift_JIS 以外の文字コード、CR+LF 以外の改行コードで保存する場合、“初期文字コード (C)”、“初期改行コード (E)”を設定してください。保存のたびに、設定する必要がなくなります。

*2 こちらの“タブ”は、表示内容を切り替えるためのタブを意味しています。“オプション”列で使っている“タブ”は、Tab キーで入力できるタブ文字ですので、ご注意ください。

*3 タブを挿入したときに進む位置。

付録 B

MinGW のインストールと使い方

ここで紹介する操作の一部には、管理者の権限が必要です。はじめから管理者のアカウントでサインインして作業することをおすすめします。

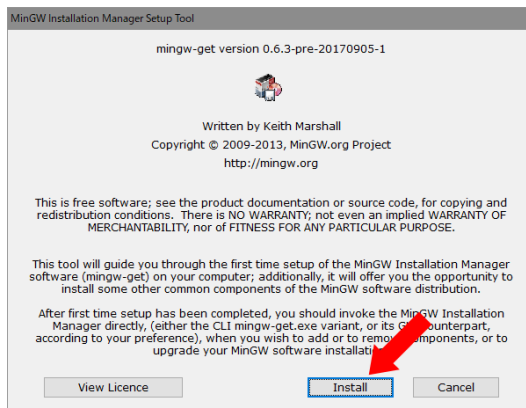
B.1 MinGW のインストール

Windows 10 に MinGW をインストールする手順を紹介します。PC の環境や MinGW のバージョンによって、画面の表示内容などが多少異なることがあります。

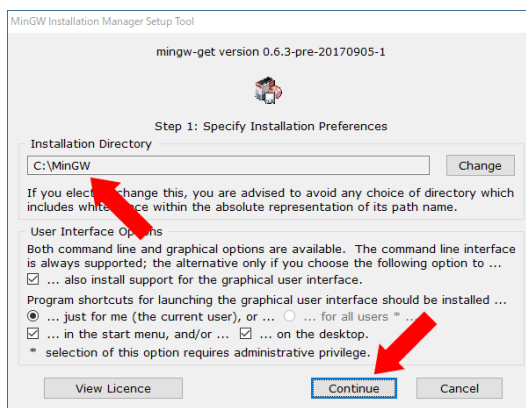
1. OSDN の MinGW プロジェクトサイトである <https://ja.osdn.net/projects/mingw/> にアクセスし、mingw-get-setup.exe をダウンロードします。



2. ダウンロードした mingw-get-setup.exe を実行すると、MinGW Installation Manager Setup Tool が起動します。“Install” をクリックします。

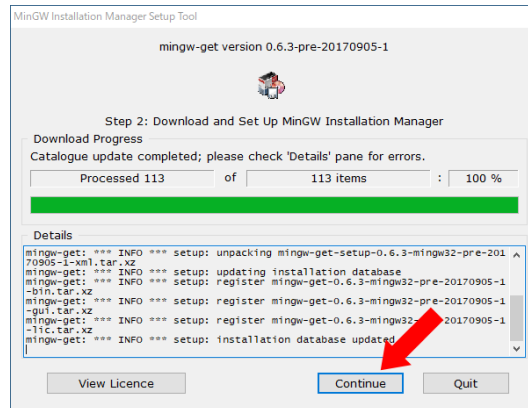


3. インストールディレクトリ (Installation Directory) が表示されますので、記録してください。この例では、C:\MinGW です*¹。これは、C ドライブの中の MinGW ディレクトリ (フォルダ) を意味しています。インストールディレクトリを記録したら、“Continue” をクリックします。

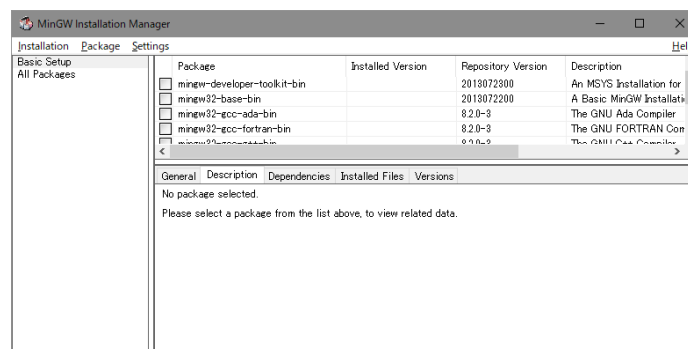


*¹ バックスラッシュ (\) と円記号 (¥) は、見た目が違うだけで意味は同じです。日本語の環境では、普通は C:¥MinGW と表記されます。

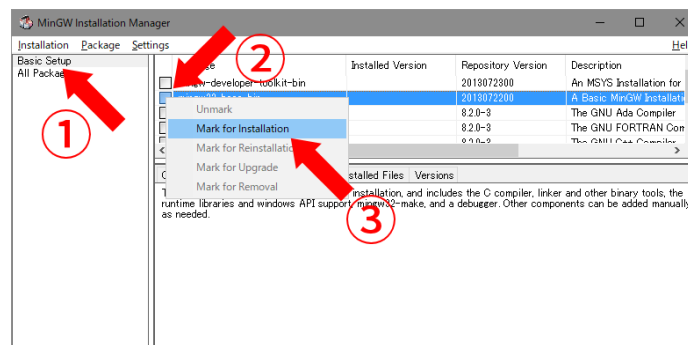
4. MinGW Installation Manager のダウンロードとセットアップが始まります。次の図のようになれば、終了です。“Continue” をクリックします。



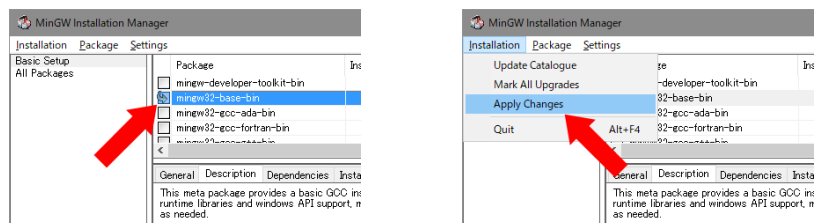
5. 自動的に MinGW Installation Manager が起動します。自動的に起動しない場合や、閉じてしまった場合は、デスクトップにできている MinGW Installer か、スタートメニューに追加された MinGW Installation Manager から起動してください。



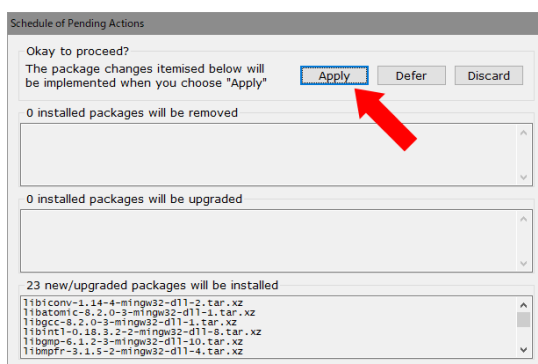
6. MinGW Installation Manager では、必要な機能を選択し、インストールできます。左のペイン (領域) で“Basic Setup” が選択されていることを確認します。右上のペインに表示される“mingw32-base-bin” のチェックボックスをクリックし、“Mark for Installation” をクリックします。



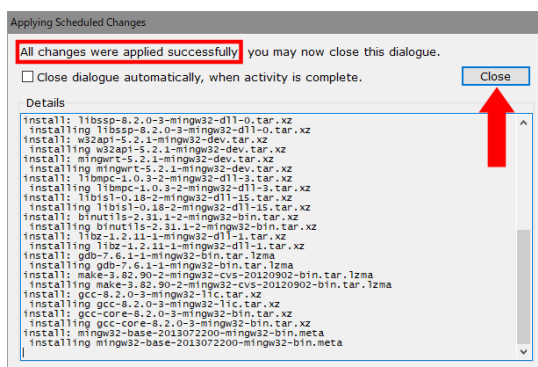
7. チェックボックスの表示が変わったことを確認し、メニューバーの“Installation” → “Apply Changes” をクリックします。



8. 次の図のような確認画面が表示されます。“Apply” をクリックするとインストールが始まります。



9. インストールが終わると、“All changes were applied successfully”と表示されます。“Close” をクリックします。




これで、MinGW のインストールは終了です。MinGW Installation Manager を閉じてください。

B.2 MinGW を使うための設定

MinGW は、コマンドプロンプトから使います。コマンドプロンプトは、次のような方法で起動できます。

- スタートメニューの「“Windows ツール” または “Windows システム ツール”」 → “コマンド プロンプト”
- **Win**+**R***2で “ファイル名を指定して実行” ダイアログを表示し、cmd と入力

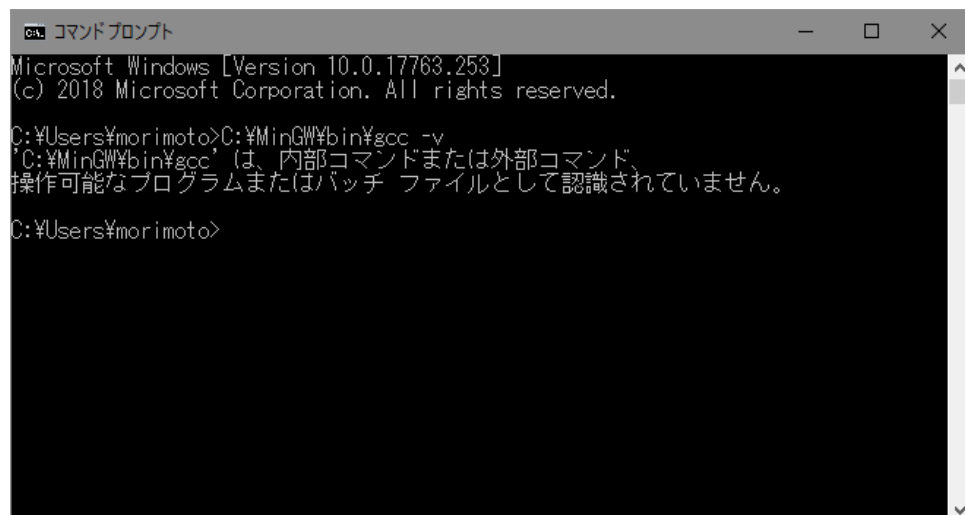
MinGW を C:¥MinGW にインストールした場合、コンパイルに使うコマンドは C:¥MinGW¥bin¥gcc です。コマンドプロンプトに **C:¥MinGW¥bin¥gcc -v** と入力し、エンターキーを押してください。図 B.1 のような出力が得られるはずです。図 B.2 のようになる場合は、入力したコマンドが間違えているか、MinGW が正しくインストールできていません。



```

C:\> gcc -v
Using built-in specs.
COLLECT_GCC=c:\mingw\bin\gcc
COLLECT_LTO_WRAPPER=c:/mingw/bin/./libexec/gcc/mingw32/8.2.0/lto-wrapper.exe
Target: mingw32
Configured with: ../src/gcc-8.2.0/configure --build=x86_64-pc-linux-gnu --host=mingw32 --target=mingw32 --prefix=/mingw --disable-win32-registry --with-arch=i586 --with-tune=generic --enable-languages=c,c++,objc,obj-c++,fortran,ada --with-pkgversion='MinGW.org GCC-8.2.0-3' --with-gmp=/mingw --with-mpfr=/mingw --with-mpc=/mingw --enable-static --enable-shared --enable-threads --with-dwarf2 --disable-sjlj-exceptions --enable-version-specific-runtime-libs --with-libiconv-prefix=/mingw --with-libintl-prefix=/mingw --enable-libstdcxx-debug --with-isl=/mingw --enable-libgomp --disable-libvtv --enable-nls --disable-build-format-warnings
Thread model: win32
gcc version 8.2.0 (MinGW.org GCC-8.2.0-3)
  
```

図 B.1 gcc の出力例



```

C:\> C:¥MinGW¥bin¥gcc -v
' C:¥MinGW¥bin¥gcc ' は、内部コマンドまたは外部コマンド、操作可能なプログラムまたはバッチ ファイルとして認識されていません。
C:\>
  
```

図 B.2 gcc の実行に失敗

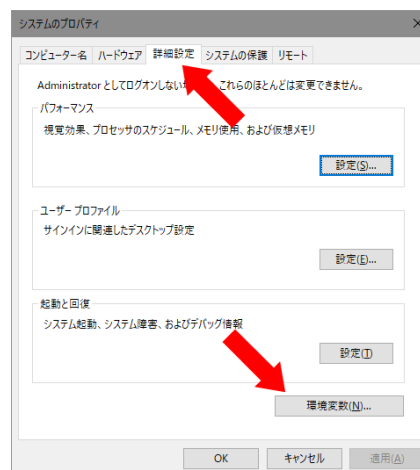
*2 Windows キーを押しながら “R” キーを押し、すぐに両方のキーを離す。

次に、環境変数 PATH の設定を行います。この設定により、`C:\MinGW\bin\gcc` ではなく、`gcc` だけでコマンドを呼び出せるようになります。説明は省略しますが、コマンドの呼び出しを簡単にする以外の役割も持っており、設定は必須です。

1. 次のような方法で、Windows の設定の「バージョン情報」または「詳細情報」を開きます。
 - スタートメニューの「設定」 → 「システム」 → 「バージョン情報」または「詳細情報」
 - スタートメニューの「Windows ツール」または「Windows システム ツール」 → 「コントロール パネル」 → 「システムとセキュリティ」 → 「システム」
 - Win + Pause
 - エクスプローラーの「PC」を右クリック → 「プロパティ (R)」
2. 「システムの詳細設定」をクリックして、システムのプロパティを開きます*3。

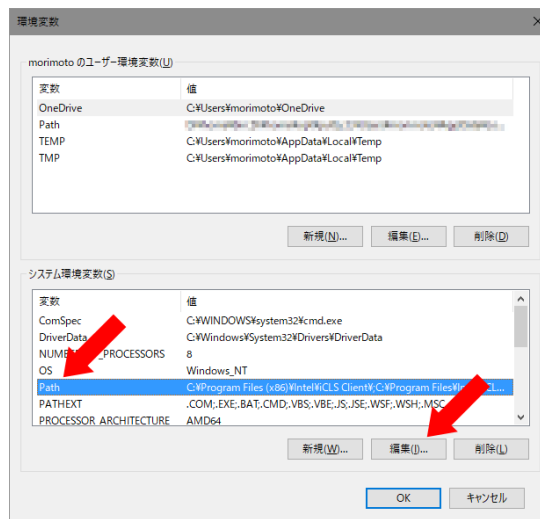
関連設定
[BitLocker の設定](#)
[デバイス マネージャー](#)
[リモート デスクトップ](#)
[システムの保護](#)
[システムの詳細設定](#)
[この PC の名前を変更 \(詳細設定\)](#)

3. 「詳細設定」タブが選択されていることを確認し、「環境変数 (N)...」をクリックします。

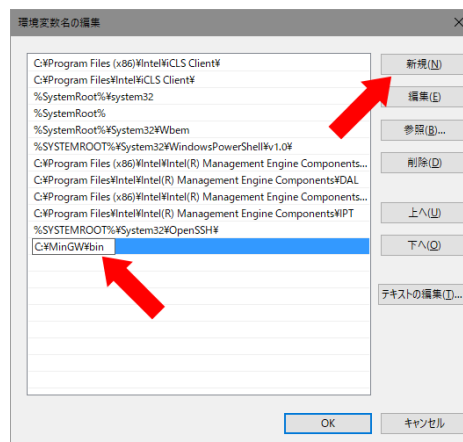


*3 「ファイル名を指定して実行」で `sysdm.cpl` と入力すれば、直接システムのプロパティを開けます。

4. “システム環境変数 (S)” の “Path” の行を選択して、“編集 (I)...” をクリックします。



5. “新規 (N)” をクリックし、MinGW のインストール時に記録したインストールディレクトリの後ろに %bin を加えた文字列を記入します。インストールディレクトリが C:%MinGW の場合は、C:%MinGW%bin と記入します。



“OK” を 3 回押して、システムのプロパティを閉じます。これで設定は完了です。

確認のため、コマンドプロンプトを起動してください^{*4}。 `C:%MinGW%bin%gcc -v` の代わりに `gcc -v` と入力し、図 B.1 のような出力が得られれば、正しく設定できています。

B.3 コマンドプロンプトの操作とコンパイル

MinGW の機能は、コマンドプロンプトにコマンドを入力することによって呼び出します。この授業で使う MinGW のコマンドは `gcc` だけです。

^{*4} 設定が完了する前から起動していたコマンドプロンプトには、設定が反映されません。新しく起動してください。

まずは、コマンドプロンプトの使い方を説明します。

コマンドプロンプトを起動すると、図 B.3 のような画面が表示されます。この例では、`C:¥Users¥morimoto>` と表示され、その後ろに白い四角が点滅しています。`C:¥Users¥morimoto>` の部分を**プロンプト**、白い四角を**カーソル**といいます。プロンプトは、ユーザにコマンドの入力を促す意味を持っています。キーボードから文字を入力すると、カーソルの位置に入力されます。コマンドを入力し、エンターキーを押すと、そのコマンドが実行されます。

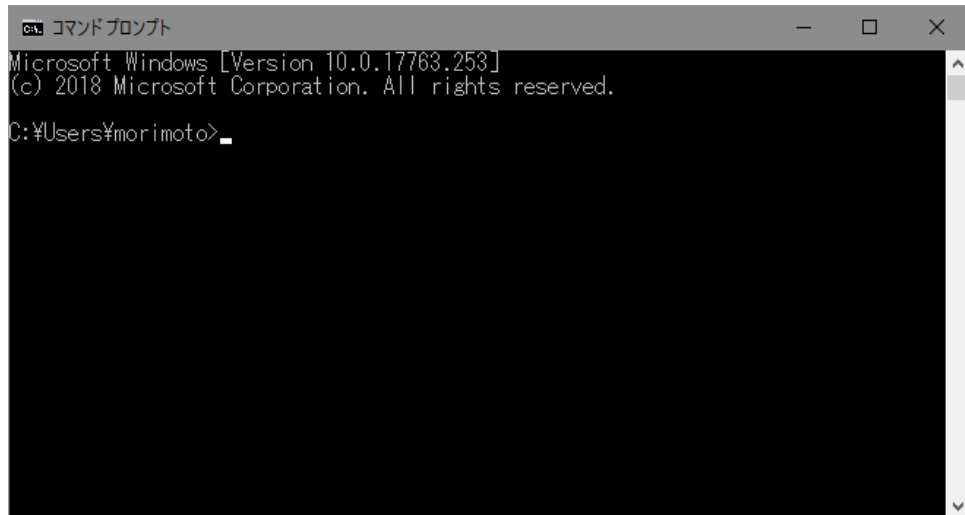


図 B.3 コマンドプロンプト

現在注目しているディレクトリ (フォルダ) を**カレントディレクトリ**といいます。カレントディレクトリは、プロンプトの一部として表示されています。この例では、`C:¥Users¥morimoto`^{*5}です。カレントディレクトリを変更するには、`cd` コマンドを使います。`cd` コマンドの基本的な書式は、

`cd directry`

です。*directry* の部分には、新しいカレントディレクトリを指定します。“`..`” は 1 つ上位のディレクトリ (親ディレクトリ) を表します。表 B.1 にいくつかの例を示します。カレントディレクトリを変更した後に、プロンプトも変わることを確認してください。

演習で使うソースファイルは、1 箇所にまとめておくと便利です。日本語やスペースを含まず、簡単に移動できる `C:¥c.ensyu` などがおすすめです。

カレントディレクトリにあるディレクトリ・ファイルの一覧を表示するには、`dir` コマンドを使います。`dir` と入力するだけです。`dir` コマンドの実行例は省略します。なれないと出力が見づ

^{*5} Windows でディレクトリやファイルを指し示すには、ドライブレターと呼ばれる A から Z の英字 1 文字と、`¥` の後ろに、目的とするディレクトリやファイルにたどり着くまでに経由するディレクトリ名・ファイル名を `¥` 区切りで記します。`C:¥Users¥morimoto` は、C ドライブの中の Users ディレクトリの中の morimoto ディレクトリを表しています。

表 B.1 cd コマンドの例

コマンド	意味
cd test	現在のカレントディレクトリの中にある test ディレクトリに移動する。
cd ..	1 つ上位のディレクトリに移動する。
cd ../..	2 つ上位のディレクトリに移動する。
cd C:¥c_ensyu	C:¥c_ensyu に移動する。
cd /D E:¥home¥morimoto	現在のカレントディレクトリとは異なるドライブに移動する場合、/D を付ける必要がある。この例では、E:¥home¥morimoto に移動する。

らいので、エクスプローラーで代用してもよいと思います*6。

↑キーを押すと、直前に実行したコマンドが再表示されます。同じコマンドを何度も実行する場合に、入力の手間が減って便利です。

以上がコマンドプロンプトの使い方です。ディレクトリの作成やファイルのコピーなどもコマンドで実行できますが、エクスプローラーで作業する方が簡単です。

次に、コンパイルと実行の方法を説明します。

コンパイルするには gcc コマンドを使います。gcc コマンドの基本的な書式は、

```
gcc -o output_file source_file
```

です。source_file の部分にソースファイル名を、output_file の部分にコンパイルしてできる実行ファイル名を指定します。実行ファイル名の拡張子は .exe としてください。ただし、拡張子を省略すると、自動的に .exe が付けられます。また、-o output_file を省略すると、実行ファイル名は a.exe になります。ソースファイルはカレントディレクトリに置かれている必要があります。実行ファイルはカレントディレクトリに作られます。表 B.2 にいくつかの例を示します。

表 B.2 gcc コマンドの例

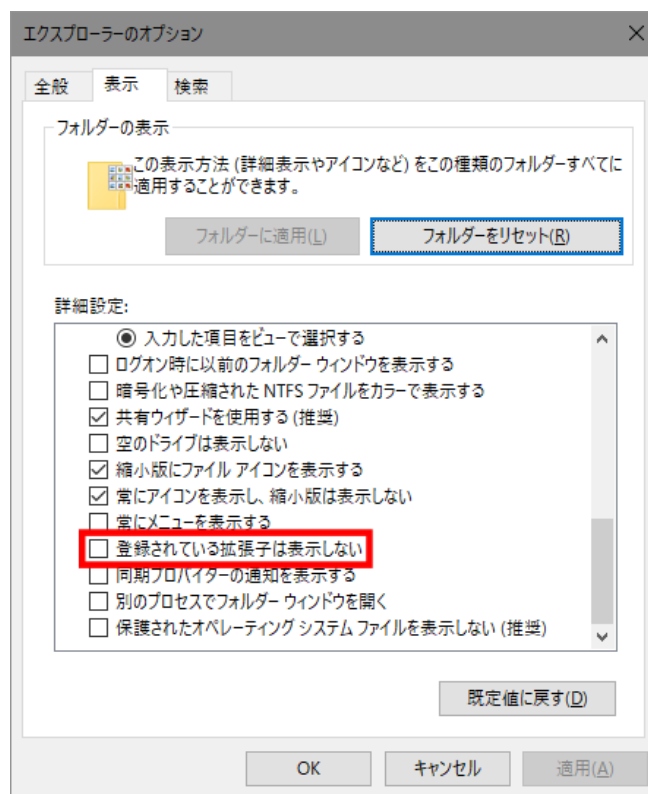
コマンド	意味
gcc -o ex01.exe ex01.c	ex01.c をコンパイルして ex01.exe を作る。
gcc -o ex01 ex01.c	同上。
gcc ex01.c	ex01.c をコンパイルして a.exe を作る。
gcc -o exec.exe source.c	source.c をコンパイルして exec.exe を作る。

*6 カレントディレクトリをエクスプローラーで開くと、当然同じディレクトリ・ファイルが見られます。

実行ファイルを実行するには、コマンドプロンプトにファイル名を入力します。ex01.exe を実行するには、`ex01.exe` と入力してエンターキーを押します。.exe は省略できますので、`ex01` と入力しても ex01.exe が実行されます。

なお、実行ファイルをダブルクリックするなどして直接起動することもできますが、プログラムの終了後すぐにコマンドプロンプトが閉じてしまい、実行結果を確認できません。

実行ファイル名は、ソースファイル名の .c を .exe に変更したものとすることが一般的です。拡張子だけが異なるファイルが存在することになりますので、エクスプローラーで拡張子を表示する設定にすると、より見分けが付きやすくなります。拡張子を表示したい場合、スタートメニューの“Windows システム ツール” → “コントロール パネル” → “デスクトップのカスタマイズ” → “エクスプローラーのオプション” と進み、“表示” タブで、“登録されている拡張子は表示しない”のチェックを外してください。



付録 C

コンパイルのエラーメッセージ例

コンパイルに失敗すると、エラーメッセージが出力されます。MinGW の gcc コマンドが出力するエラーメッセージの例を紹介します。第 3 章で書いた最初のプログラム (ソースコード 3.1, ex02.c) を例にしますので、再掲します。

ソースコード C.1 最初のプログラム (ソースコード 3.1 の再掲)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int age;
6
7      printf("今何歳ですか? > ");
8      scanf("%d", &age);
9
10     /* 5年後の年齢を表示する */
11     printf("5年後には%d歳です.\n", age + 5);
12
13     return 0;
14 }
```

誤り例 1

1 行目の `stdio.h` を `studio.h` と記述してしまった。

ソースコード C.2 誤り例 1

```
1  #include <studio.h>
```

誤り例 1 のエラーメッセージ例

```
ex02.c:1:10: fatal error: studio.h: No such file or directory
#include <studio.h>
~~~~~
compilation terminated.
```

エラーメッセージの先頭に、ex02.c:1 と記述されています。これは、ex02.c の 1 行目という意味です。エラーメッセージには、studio.h というファイル*¹が見つからないと書かれています。

誤り例 2

5 行目の末尾の ; を抜かしてしまった。

ソースコード C.3 誤り例 2

```
5    int age
6
7    printf("今何歳ですか? > ");
```

誤り例 2 のエラーメッセージ例

```
ex02.c: In function 'main':
ex02.c:7:2: error: expected '=', ',', ';', 'asm' or '__attribute__'
before 'printf'
    :
以下略
```

5 行目の誤りですが、エラーメッセージには 7 行目と記述されています (ex02.c:7)。C 言語のほとんどの文は ; で終わる必要があります。5 行目に ; が見つからなかったため、文が継続していると見なされて、7 行目でエラーになりました。このように、誤りの場所とは違う行でエラーになることもあります。

誤り例 3

7 行目の 2 つ目の " を抜かしてしまった。

ソースコード C.4 誤り例 3

```
7    printf("今何歳ですか? > );
```

*¹ ヘッダファイルと呼ばれるファイルです。第 17 章で学習します。

誤り例 3 のエラーメッセージ例

```
ex02.c: In function 'main':
ex02.c:7:9: warning: missing terminating " character
    printf("今何歳ですか? > );
           ^
ex02.c:7:9: error: missing terminating " character
    printf("今何歳ですか? > );
           ~~~~~
:
以下略
```

"今何歳ですか? > " の 1 つ目の " と 2 つ目の " は対になっています。エラーメッセージには、終端の、つまり 2 つ目の " が見つからないと書かれています。シンタックスハイライト機能を持ったエディタであれば、表示がおかしくなるので、誤りに気づくことができます。

なお、誤り例 3 の実際のエラーメッセージは文字化けしています。そのままでは読みにくいため、文字化けしていなければ表示されるはずのメッセージを記載しています。次の誤り例 4 でも日本語文字の部分が文字化けしていますが、こちらはそのまま記載しています。

誤り例 4

8 行目の ; を全角文字で書いてしまった。

ソースコード C.5 誤り例 4 (最後の ; が全角文字)

```
8 scanf("%d", &age);
```

誤り例 4 のエラーメッセージ例

```
ex02.c: In function 'main':
ex02.c:8:19: error: stray '\201' in program
    scanf("%d", &age)mG
                   ^
:
以下略
```

エラーメッセージには、8 行目におかしな文字があると書かれています。

誤り例 5

10 行目の末尾 (* / の後ろ) に全角スペースが紛れ込んでいる。

ソースコード C.6 誤り例 5 (* / の後ろに全角スペース)

```
10  /* 5年後の年齢を表示する */
```

誤り例 5 のエラーメッセージ例

```
ex02.c: In function 'main':
ex02.c:10:29: error: stray '\201' in program
      :
以下略
```

エラーメッセージには、10 行目におかしな文字があると書かれています。

誤り例 4 と誤り例 5 のエラーメッセージには '\201' と記述されています。Shift_JIS の全角セミコロ (;) と全角スペースは 2 バイトで表され、どちらも最初のバイトが八進数で 201 (十六進数で 81) です。コンパイラが先頭からソースコードを読み進め、全角文字が出てきた時点、つまり八進数で 201 のバイトに遭遇した時点でエラーになったことがわかります。

誤り例 6

} を 1 つ余計に入れてしまった。

ソースコード C.7 誤り例 6

```
11  printf("5年後には%d歳です.\n", age + 5);
12
13  }
14  return 0;
15  }
```

誤り例 6 のエラーメッセージ例

```
ex02.c:14:2: error: expected identifier or '(' before 'return'
    return 0;
    ~~~~~
    :
以下略
```

return 0; は、“{” と “}” の間に入っている必要がありますが、4 行目で書いた “{” が 13 行目の “}” で閉じられています。ですので、14 行目の return 0; でエラーが起こっています。

以上のようにエラーメッセージは英語で、内容もあまり親切ではないと感じるかもしれません。しかし、エラーが起こった行か、その少し上の行に原因があることがほとんどです。たくさんエラーが表示される場合は、まず最初のエラーの行番号を確認してください。

C 言語基礎演習 教科書

第 1.6 版

2023/4/1

主任講師 森本容介