第 1 章 2 進数，16 進数，2 の補数

コンピュータの内部では，すべての情報が 0 と 1 の組み合わせで表現されている。本章では，コンピュータが数値を扱う基本的な仕組みである 2 進数，16 進数，2 の補数について学ぶ。これらの知識は，プログラミング，データ構造，コンピュータアーキテクチャなど，情報工学のあらゆる分野の基盤となる。本章の内容を理解することで，コンピュータが内部でどのように数値を処理しているかを把握でき，より深いレベルでのプログラム設計やデバッグが可能となる。

本章の学習目標

- デジタルの概念と，情報・データの違いを説明できる
- 2 進数と 10 進数の相互変換ができる
- 16 進数と 2 進数・10 進数の相互変換ができる
- 2 の補数を用いた負の整数の表現方法を理解する

1.1 デジタルとは

本節では，コンピュータにおけるデジタル表現の基本概念と，情報とデータの違いについて学ぶ。これらはコンピュータを理解するための基礎的な概念である。

1.1.1 デジタルの基本概念

コンピュータでは，すべてのデータとプログラムを **0 と 1（デジタル）**で表現する。文字，画像，音声，動画など，あらゆる種類の情報が最終的には 0 と 1 の列に変換されて処理される。

デジタルの世界では，すべてが「0」と「1」の列で構成される。以下はその一例である。

００１０１１１０１００１０１００１０１００１０１０１００１

このような 0 と 1 の並びにおいて，1 個の「0」または「1」を **1 ビット**と呼ぶ。**ビット**（bit）は「binary digit」（2 進数の桁）の略であり，**情報の最小単位**である。コンピュータが扱うすべてのデータはビットの集まりとして表現される。

1.1.2 情報とデータ

日常的に「情報」と「データ」という言葉は混同されがちであるが，コンピュータサイエンスではこれらを区別して用いる。

情報とは，人間にとって意味のある内容そのものである。例えば，「あの人の電話番号は『123-4567』だ」「明日は晴れだ」といったものが情報である。情報という言葉は，気象情報，個人情報，にせ情報，情報活動，外交情報，情報機関などの用語で使われる。

データとは，コンピュータが処理できる形式に情報を変換したものである。例えば，電話番号を数値として表した 1234567（数値データ）や，天気を文字で表した「晴れ」（文字列データ）などがデータである。データという言葉は，データ通信，データベース，電子メールのデータ，WWW のデータなどの用語で使われる。

このように，人間が理解する「情報」をコンピュータが処理できる形式に変換したものが「データ」である。

1.2 2 進数

本節では，2 進数の基本概念と，2 進数と 10 進数の相互変換方法について学ぶ。2 進数はコンピュータ内部での数値表現の基本である。

1.2.1 2 進数とビット

私たちが日常的に使う 10 進数は，0 から 9 までの 10 個の数字を用いる。これに対し，**2 進数では「0」と「1」だけを使う**。以下は 2 進数の例である。

```
00110101111101110101011
```

2 進数において，**ビット**とは 2 進数の 1 桁のことを指す。例えば，00110101 という 8 桁の 2 進数を考えよう。ビットの位置は通常，最下位（右端）を 1 ビット目として数える。この例では，下から 4 ビット目（右から 4 番目）の値は 0 である。

```
00110101
    ↑
  4 ビット目（値は 0）
```

1.2.2 2 進数と 10 進数の変換

コンピュータの内部では 2 進数が使われるが，人間にとっては 10 進数の方が理解しやすい。そのため，2 進数と 10 進数を相互に変換する方法を理解しておく必要がある。

2 進数から 10 進数への変換

10 進数では，各桁に右から順に 1，10，100，1000，…という重みがある。例えば，10 進数の 246 は，$2 \times 100 + 4 \times 10 + 6 \times 1 = 246$と解釈できる。

2 進数でも同様に，各桁には重みがある。2 進数の場合は右から順に **1，2，4，8，16，32，…**という重みになる。これらは 2 の累乗（$2^0 = 1$，$2^1 = 2$，$2^2 = 4$，$2^3 = 8$，…）に対応している。2 進数を 10 進数に変換するには，**各桁の値（0 または 1）とその重みを掛け合わせ，すべての桁について合計する**。

具体例として，2 進数の 1001 を 10 進数に変換する。まず，各桁の重みを確認する。

```
2 進数:    1    0    0    1
重み:      8    4    2    1
```

各桁の値と重みを掛けて合計する。

$$1001_{(2)} = 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 8 + 0 + 0 + 1 = 9$$

したがって，2 進数の 1001 は 10 進数の 9 である。

10 進数から 2 進数への変換

10 進数を 2 進数に変換するには，**10 進数を 2 の累乗の和で表す**。具体的には，与えられた 10 進数を超えない最大の 2 の累乗を見つけ，それを引くという操作を繰り返す。

具体例として，10 進数の 46 を 2 進数に変換する。まず，2 の累乗を小さい順に並べると，1，2，4，8，16，32，64，…となる。46 を超えない最大の 2 の累乗は 32 である。

変換手順を以下に示す。

1. 46 から 32 を引く：46 − 32 = 14（32 の位は 1）
2. 14 と 16 を比較する：16 は 14 より大きいので使わない（16 の位は 0）
3. 14 から 8 を引く：14 − 8 = 6（8 の位は 1）
4. 6 から 4 を引く：6 − 4 = 2（4 の位は 1）
5. 2 から 2 を引く：2 − 2 = 0（2 の位は 1）
6. 残りが 0 なので終了（1 の位は 0）

これを整理すると，

$$46 = 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 101110_{(2)}$$

したがって，10 進数の 46 は 2 進数の 101110 である。

1.3 16 進数

本節では，16 進数の基本概念と，2 進数・10 進数との相互変換方法について学ぶ。16 進数は 2 進数を簡潔に表現するための記法として，プログラミングやシステム開発で広く用いられている。

1.3.1 16 進数とは

2 進数はコンピュータの内部表現として重要であるが，桁数が多くなると人間には読みにくい。例えば，32 ビットの数値を 2 進数で書くと 32 桁になる。この問題を解決するために 16 進数が用いられる。

16 進数では，16 個の記号 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F を使う。0 から 9 までは 10 進数と同じであるが，10 から 15 に相当する値には，数字の代わりに A, B, C, D, E, F の文字を使用する。以下は 16 進数の例である。

```
0065FDF0
```

この例は 8 桁の 16 進数であり，2 進数で表すと 32 桁になる数値を簡潔に表現している。

1.3.2 2 進数と 16 進数の対応

16 進数の 1 桁は，2 進数の 4 桁に対応する。これは，$16 = 2^4$ という関係に基づいている。2 進数 4 桁で表せる値の範囲（0000〜1111，10 進数で 0〜15）と，16 進数 1 桁で表せる値の範囲（0〜F，10 進数で 0〜15）が一致するためである。

2 進数と 16 進数の対応を以下の表に示す。

16 進数	2 進数	16 進数	2 進数
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

この対応表を用いると，2 進数と 16 進数の変換を素早く行うことができる。

1.3.3 2 進数と 16 進数の関係

前項で示したように，**2 進数 4 桁は 16 進数の 1 桁**に対応する。この関係を利用すると，長い 2 進数を簡潔に表現できる。変換の手順は，2 進数を右から 4 桁ずつ区切り，各グループを対応する 16 進数の 1 桁に置き換えることである。

具体例として，16 桁の 2 進数を 4 桁の 16 進数で表す。

```
2 進数:   0011  0101  1001  1100
           ↓     ↓     ↓     ↓
16 進数:   3     5     9     C
```

各グループを対応表で変換すると，0011→3，0101→5，1001→9，1100→C となる。したがって，2 進数 0011010110011100 は 16 進数で 359C となる。このように，16 桁の 2 進数が 4 桁の 16 進数で表現できる。

1.3.4 10 進数と 16 進数の対応

16 進数と 10 進数の間にも変換が必要になることがある。16 進数の各桁の値と 10 進数の対応を以下の表に示す。

16 進数	10 進数	16 進数	10 進数
0	0	8	8
1	1	9	9
2	2	A	10
3	3	B	11
4	4	C	12
5	5	D	13
6	6	E	14
7	7	F	15

A〜F が 10〜15 に対応するという点が重要である。

1.3.5 10 進数と 16 進数の変換

16 進数から 10 進数への変換

10 進数の各桁の重みが 1，10，100，1000，…（10 の累乗）であるのと同様に，16 進数の各桁には，右から順に **1，16，256，4096，…**という重みがある。これらは 16 の累乗（$16^0 = 1$，$16^1 = 16$，$16^2 = 256$，$16^3 = 4096$，…）に対応している。16 進数を 10 進数に変換するには，**各桁の値とその重みを掛け合わせ，すべての桁について合計する**。

具体例として，16 進数の A0C8 を 10 進数に変換する。まず各桁の値を 10 進数に直すと，A=10，0=0，C=12，8=8 である。次に，各桁の重みを確認する。

```
16 進数:   A      0      C      8
重み:   4096   256     16      1
10 進数:  10      0     12      8
```

各桁の値と重みを掛けて合計する。

$$A0C8_{(16)} = 10 \times 4096 + 0 \times 256 + 12 \times 16 + 8 \times 1$$

$$= 40960 + 0 + 192 + 8 = 41160$$

したがって，16 進数の A0C8 は 10 進数の 41160 である。

10 進数から 16 進数への変換

10 進数を 16 進数に変換するには，**10 進数を 16 の累乗の和で表す**。2 進数への変換と同様に，与えられた 10 進数を超えない最大の 16 の累乗を見つけ，それで割って商と余りを求めるという操作を繰り返す。

具体例として，10 進数の 368 を 16 進数に変換する。16 の累乗は 1，16，256，4096，…である。368 を超えない最大の 16 の累乗は 256 である。

```
368 ÷ 256 = 1 余り 112   （256 の位は 1）
112 ÷ 16 = 7 余り 0      （16 の位は 7）
0 ÷ 1 = 0                （1 の位は 0）
```

これを整理すると，

$$368 = 1 \times 256 + 7 \times 16 + 0 \times 1 = 170_{(16)}$$

したがって，10 進数の 368 は 16 進数の 170 である。

1.4 2 の補数

本節では，コンピュータで負の整数を表現するための 2 の補数について学ぶ。これまで扱ってきた 2 進数は正の整数のみを表現できたが，2 の補数を用いることで負の整数も統一的に扱えるようになる。

1.4.1 2 の補数とは

コンピュータで負の整数を扱う方法はいくつかあるが，**2 の補数**は最も広く用いられている方式である。2 の補数は，負の整数を扱うときに便利であり，加算回路だけで減算も実行できるという利点がある。

2 の補数では，**最上位ビットが符号ビット**となる。符号ビットの意味は以下のとおりである。

- 最上位ビットが **0** の場合 → 正の整数または 0 を表す
- 最上位ビットが **1** の場合 → 負の整数を表す

1.4.2 2 の補数の求め方

ある正の整数に対応する負の整数（符号を反転させた数）を 2 の補数表現で求めるには，以下の手順を用いる。

1. 元の正の整数を 2 進数で表す
2. すべてのビットを反転させる（0 を 1 に，1 を 0 にする。この結果を **1 の補数**という）
3. 1 を加える

例として，8 ビットの 2 の補数表現で-1 を求める。

手順 1：1 を 2 進数で表す → 00000001
手順 2：ビットを反転させる → 11111110（1 の補数）
手順 3：1 を加える → 11111111（2 の補数）

したがって，8 ビットの 2 の補数表現では，**-1 は 11111111** となる。

1.4.3 8 ビットでの表現例

2 の補数表現の具体例として，8 ビットの 2 の補数表現においていくつかの整数がどのように表されるかを以下の表に示す。

10 進数	8 ビット 2 の補数表現	説明
2	00000010	正の数（最上位ビットが 0）
1	00000001	正の数（最上位ビットが 0）
0	00000000	ゼロ
-1	11111111	00000001 のビット反転+1
-2	11111110	00000010 のビット反転+1

この表から，正の整数では最上位ビットが 0 であり，負の整数では最上位ビットが 1 であることが確認できる。

1.4.4 符号ビットの位置

符号ビットは常に最上位ビットに置かれる。8 ビットの整数データの場合，ビット位置は以下のようになる。

```
ビット位置:  8  7  6  5  4  3  2  1
             ↑
          符号ビット
```

2 の補数では，**最上位ビット（この例では 8 ビット目）が符号ビット**である。このビットは，その数が正なのか負なのかを判別するために使われる。**符号ビットが 0 なら正の数（または 0），1 なら負の数**である。

1.4.5 2 の補数の計算例

2 の補数表現が正しく機能することを確認するため，45 と-45 を足すと 0 になることを示す。正しい表現方法であれば，ある数とその符号を反転させた数を足したとき，結果は 0 になるはずである。

まず，-45 の 2 の補数表現を求める。

```
手順 1: 45 を 2 進数で表す     → 00101101（32+8+4+1=45）
手順 2: ビットを反転させる → 11010010（1 の補数）
手順 3: 1 を加える             → 11010011（2 の補数）
```

次に，8 ビットの 2 の補数で 45 と-45 を加算する。

```
 45:  00101101
-45:  11010011
──────────────
     100000000
```

この計算結果は 9 ビットになるが，8 ビットの範囲では最上位の 1 は無視され（オーバーフローとして扱われ），結果は 00000000（つまり 0）となる。

この例からわかるように，2 の補数では**負の数は最上位ビットが 1** となる。また，通常の加算回路をそのまま使って減算（負の数の加算）を実行できることが，2 の補数の利点である。

本章のまとめ

本章では，コンピュータにおける数値表現の基礎について学んだ。以下に各節の要点を整理する。

デジタルと情報・データ： コンピュータでは，すべてのデータとプログラムが 0 と 1 で表現される。情報は人間にとって意味のある内容であり，データはそれをコンピュータが処理できる形式に変換したものである。**ビット**は 2 進数の 1 桁であり，情報の最小単位である。

2 進数： 2 進数は 0 と 1 だけを使う数値表現である。各桁には右から **1, 2, 4, 8, 16, …**という重み（2 の累乗）があり，この重みを用いて 2 進数と 10 進数を相互に変換できる。

16 進数： 16 進数は **0〜9 と A〜F の 16 個の記号**を使う。**2 進数 4 桁が 16 進数 1 桁に対応する**ため，長い 2 進数を簡潔に表現できる。プログラミングやシステム開発では，メモリアドレスや色コードなど，様々な場面で 16 進数が用いられる。

2 の補数： 2 の補数は負の整数を表現するための方式である。**最上位ビットが符号ビット**となり，0 で正または 0，1 で負を表す。負の数を求めるには，「**ビット反転して 1 を加える**」という手順を用いる。2 の補数を用いることで，加算回路だけで減算も実行できる。

章末問題

以下の問題を解いて，本章の理解度を確認する。

問題 1： 2 進数 11010110 を 10 進数に変換せよ。

問題 2： 10 進数 93 を 2 進数に変換せよ。

問題 3： 16 進数 3F7A を 10 進数に変換せよ。

問題 4： 10 進数 500 を 16 進数に変換せよ。

問題 5： 2 進数 10101100 を 16 進数に変換せよ。

問題 6： 8 ビットの 2 の補数表現で，-35 を 2 進数で表せ。

問題 7： 8 ビットの 2 の補数表現 11100101 は，10 進数でいくらか。

章末問題の解答と解説

問題 1 の解答： 214

各桁の重みは右から順に 1, 2, 4, 8, 16, 32, 64, 128 である。

```
2 進数:  1    1    0    1    0    1    1    0
重み: 128   64   32   16    8    4    2    1
```

1 が立っている桁の重みを合計する。

$$11010110_{(2)} = 128 + 64 + 16 + 4 + 2 = 214$$

問題 2 の解答： 1011101

10 進数を 2 の累乗の和で表し，対応する桁に 1 を立てる。

```
93 - 64 = 29   （64 を使う）
29 - 16 = 13   （16 を使う）
13 - 8 = 5     （8 を使う）
5 - 4 = 1      （4 を使う）
1 - 1 = 0      （1 を使う）
```

使った 2 の累乗は 64, 16, 8, 4, 1 である。

$$93 = 64 + 16 + 8 + 4 + 1 = 1011101_{(2)}$$

補足： 8 ビットで表現する場合は，先頭に 0 を補って 01011101 となる。

問題 3 の解答： 16250

16 進数の A〜F は 10〜15 に対応する。各桁の重みは右から順に 1, 16, 256, 4096 である。

```
16 進数:   3      F      7      A
重み:   4096   256     16      1
10 進数:   3     15      7     10
```

$$3F7A_{(16)} = 3 \times 4096 + 15 \times 256 + 7 \times 16 + 10 \times 1$$

$$= 12288 + 3840 + 112 + 10 = 16250$$

問題 4 の解答： 1F4

500 を 16 の累乗で分解する。

```
500 ÷ 256 = 1 余り 244   （256 の位は 1）
244 ÷ 16 = 15 余り 4     （16 の位は 15，すなわち F）
4 ÷ 1 = 4                （1 の位は 4）
```

$$500 = 1 \times 256 + 15 \times 16 + 4 \times 1 = 1F4_{(16)}$$

問題 5 の解答： AC

2 進数を右から 4 桁ずつ区切り，各グループを 16 進数に変換する。

```
2 進数:   1010    1100
           ↓       ↓
16 進数:   A        C
```

1010 は$8 + 2 = 10$で A，1100 は$8 + 4 = 12$で C である。

$$10101100_{(2)} = AC_{(16)}$$

問題 6 の解答： 11011101

2 の補数を求める手順に従う。

手順 1: 35 を 2 進数で表す
```
35 = 32 + 2 + 1 = 00100011
```

手順 2: ビットを反転させる（1 の補数）
```
00100011 → 11011100
```

手順 3: 1 を加える（2 の補数）
```
11011100 + 1 = 11011101
```

間違えないためのポイント： 2 の補数を求める手順は「ビット反転して 1 を加える」である。ビット反転だけで終わると 1 の補数になってしまうので，最後に 1 を加えることを忘れないようにする。

問題 7 の解答： -27

最上位ビットが 1 なので，この数は負の数である。2 の補数表現から元の 10 進数値を求めるには，再度 2 の補数を取って絶対値を求める。

手順 1: ビットを反転させる
```
11100101 → 00011010
```

手順 2: 1 を加える
```
00011010 + 1 = 00011011
```

手順 3: 10 進数に変換する
　　00011011 = 16 + 8 + 2 + 1 = 27

手順 4: 符号を付ける
　　元の最上位ビットが 1 だったので，答えは -27

第 2 章 メモリとメモリアドレス

コンピュータが動作する際，プログラムやデータは一時的にメモリに格納される。メモリの仕組みとアドレスの概念を理解することは，プログラムの動作原理を把握し，効率的なプログラミングやデバッグを行うための基礎となる。本章で学ぶ内容は，システム開発や組込みソフトウェア開発など，幅広い分野で活用される知識である。

本章の学習目標

- メモリの基本的な役割と機能を説明できる
- アドレスの概念とバイト単位の区切りを理解する
- メモリアドレスを 16 進数で表記する理由を説明できる
- メモリの読み出しと書き込みの動作を区別できる

2.1 メモリとは

本節では，メモリの基本的な役割と機能について学ぶ。

コンピュータでプログラムを実行したり，文書を編集したりする際，データはどこかに一時的に保存されている必要がある。この役割を担うのが**メモリ**である。メモリは，データの記憶を行うチップであり，コンピュータの動作に欠かせない部品の一つである。

メモリには，データを書き込んだり，読み出したりする機能がある。**書き込み**とは，データをメモリに記憶させることであり，**読み出し**とは，記憶されているデータをメモリから取り出すことである。プログラムが動作する際には，この書き込みと読み出しが繰り返し行われている。

2.2 メモリとアドレス

本節では，メモリの構造とアドレスの概念について学ぶ。

メモリにはたくさんのデータを格納できるが，それぞれのデータがどこに保存されているかを区別する仕組みが必要である。そのため，メモリは**バイト（8 ビット）単位**に区切られている。第 1 章で学んだように，1 バイトは 8 ビットで構成される。

各バイトには **0 から始まる通し番号**が付けられている。この通し番号のことを**アドレス**という。**番地**と呼ぶこともある。アドレスがあることで，「メモリの何番目にあるデータ」という形で，データの場所を特定できる。

以下の表に，メモリ内のデータとアドレスの対応例を示す。この表では，1 つのマス目が 1 バイトを表しており，上段の数字がアドレス，下段の 2 桁の数字がそのアドレスに格納されているデータである。データは 16 進数で表記されている。例えば，アドレス 0 には 16 進数で「01」というデータが格納されており，アドレス 4 には 16 進数で「02」というデータが格納されている。

アドレス	0	1	2	3	4	5	6	7	8	9
データ	01	00	00	00	02	00	00	00	03	00

2.3 メモリアドレスと 16 進数表記

本節では，メモリアドレスの意味と，16 進数で表記する理由について学ぶ。

前節で学んだように，メモリの各バイトにはアドレスが付けられている。この**メモリアドレス**は，読み書きすべきデータの「場所」を示すものである。

例えば，0065FDF0（16 進数）というメモリアドレスがあったとする。これは，**メモリの先頭から 0065FDF0（16 進数）番目**の位置を意味している。このように，メモリアドレスを指定することで，メモリ内の特定の場所にアクセスできる。

メモリ内のデジタルデータがどのように格納されているかを確認する。メモリ内のデータは，**8 ビットずつ区切られて**，それぞれにメモリアドレスが付けられている。以下に，ビット列が 8 ビット単位で区切られる様子を示す。

```
００１０１１１０　１００１０１００　１０１００１０１
   8 ビット　　8 ビット　　8 ビット
```

このように，どのようなビット列であっても，先頭から 8 ビットずつ区切られ，それぞれの 8 ビットに 1 つのアドレスが割り当てられる。

ところで，メモリアドレスは，ふつう **16 進数で表記**する。なぜ 16 進数を使うのか，その理由を以下に説明する。

メモリアドレスそのものもデジタル，つまり「0」と「1」の列で表現される。しかし，メモリアドレスを「0」と「1」の並びで書くと，**長すぎて人間にとって分かりづらい**という問題がある。以下に 2 進数表記の例を示す。

００１０１１１０１００１０１００１０１００１０１０１００１

このように，2 進数で表記すると長くなり，読み取ることが困難である。そこで，第 1 章で学んだ「**16 進数**」を使う。16 進数を用いると，2 進数 4 桁を 16 進数 1 桁で表現できるため，表記が短くなり，人間にとって扱いやすくなる。

2.4 メモリへの操作

本節では，メモリに対する基本的な操作である読み出しと書き込みについて学ぶ。

メモリの基本機能として述べたように，メモリには**読み出し**と**書き込み**という 2 つの基本的な操作がある。本節では，それぞれの操作が具体的にどのように行われるかを詳しく説明する。

読み出しは，メモリからデータを取得する操作である。読み出しを行う際には，読み出したいデータの**アドレスを指定**する。すると，そのアドレスに格納されているデータがメモリから取り出される。

書き込みは，メモリにデータを格納する操作である。書き込みを行う際には，書き込みたいデータの**アドレスとデータそのもの**を指定する。すると，指定したアドレスにデータが格納される。

2.4.1 読み出しの動作

読み出しの動作について，具体例を用いて説明する。以下の表は，メモリの状態を示している。表中の「??」は，何らかのデータが格納されているが，ここでは具体的な値を示していないことを表している。

アドレス	0	1	2	3	4	5	6	7	8
データ	??	??	??	??	??	??	??	??	??

この状態で，アドレス 4 番地と 5 番地から 2 バイト分を読み出す場合について説明する。2 バイト分とは，連続する 2 つのアドレスのデータを意味するため，アドレス 4 番地のデータとアドレス 5 番地のデータが読み出される。

ここで押さえるべき点は，読み出しの際に**メモリの値は変化しない**ということである。読み出しは，メモリに格納されているデータをコピーして取得する動作であり，元のデータはそのまま残る。何度読み出しても，メモリの内容は同じままである。

なお，メモリの各区画は **1 バイト（16 進数で 2 桁）**である。

2.4.2 書き込みの動作

書き込みの動作について，具体例を用いて説明する。以下では，アドレス 6 番地と 7 番地に「0400」という 2 バイトのデータを書き込む場合について説明する。

「0400」は 16 進数 4 桁のデータである。1 バイトは 16 進数 2 桁で表されるため，「0400」は 2 バイトのデータとなる。この 2 バイトを連続する 2 つのアドレスに書き込む際，先頭の 2 桁「04」がアドレス 6 番地に，次の 2 桁「00」がアドレス 7 番地に格納される。

まず，書き込み前のメモリの状態を示す。表中の「??」は，何らかのデータが格納されていることを表している。

アドレス	0	1	2	3	4	5	6	7	8
データ	??	??	??	??	??	??	??	??	??

この状態で，アドレス 6 番地に「04」，アドレス 7 番地に「00」を書き込むと，メモリの状態は以下のように変化する。

アドレス	0	1	2	3	4	5	6	7	8
データ	??	??	??	??	??	??	04	00	??

書き込みの際に押さえるべき点は，**前の値は消える**ということである。アドレス 6 番地と 7 番地に以前格納されていたデータ（表では「??」と示していた部分）は失われ，新しいデータ「04」と「00」に置き換わる。読み出しとは異なり，書き込みはデータを上書きする動作であるため，元のデータを保持したい場合は，書き込みの前にそのデータを別の場所に退避させておく必要がある。

なお，メモリの各区画は **1 バイト（16 進数で 2 桁）**である。

本章のまとめ

本章では，メモリとメモリアドレスの基本概念について学んだ。以下に要点を整理する。

メモリの役割： メモリはデータの記憶を行うチップであり，書き込みと読み出しという 2 つの基本機能を持つ。データを格納する際には書き込みを行い，データを取得する際には読み出しを行う。

アドレスの概念： メモリに格納されたデータの場所を特定するために，メモリはバイト（8 ビット）単位に区切られており，各バイトには 0 から始まるアドレス

（番地）が付けられている。このアドレスを指定することで，目的のデータにアクセスできる。

16 進数表記： メモリアドレスは通常 16 進数で表記される。これは，2 進数表記が長すぎて人間にとって分かりづらいためであり，16 進数を用いることで表記が短くなり扱いやすくなる。

読み出しと書き込みの違い： 読み出しではメモリの値は変化せず，元のデータがそのまま保持される。一方，書き込みでは前の値が消えて新しい値に置き換わる。この違いを正しく理解することが，メモリを扱う上で必要である。

章末問題

問題 1： メモリの 2 つの基本機能を答えよ。

問題 2： メモリが区切られている単位は何か。また，その単位は何ビットか。

問題 3： アドレスとは何か，簡潔に説明せよ。

問題 4： メモリアドレスを 16 進数で表記する理由を説明せよ。

問題 5： 読み出しと書き込みの動作の違いについて，メモリ内のデータがどうなるかに着目して説明せよ。

問題 6： アドレス 3 番地から 2 バイト分のデータを読み出す場合，どのアドレスのデータが読み出されるか。

問題 7： アドレス 2 番地に「FF」を書き込んだ場合，そのアドレスに以前あったデータはどうなるか。

章末問題の解答と解説

問題 1 の解答： 書き込みと読み出し

解説： 書き込みはデータをメモリに記憶させる機能であり，読み出しは記憶されているデータをメモリから取り出す機能である。

問題 2 の解答： バイト単位である。1 バイトは 8 ビットである。

解説： メモリの最小単位がバイトであること，1 バイト＝8 ビットの関係は基本事項である。

問題 3 の解答： メモリの各バイトに付けられた 0 から始まる通し番号のことである。番地ともいう。

解説： アドレスはデータの「場所」を示すものであり，0 から始まる点に留意する。

問題 4 の解答： メモリアドレスを 2 進数（0 と 1 の並び）で表記すると長すぎて人間にとって分かりづらいため，16 進数を使用する。

解説： 第 1 章で学んだように，16 進数 1 桁は 2 進数 4 桁に対応するため，表記が短くなり扱いやすくなる。

間違えないためのポイント： 「コンピュータが理解しやすいから」ではなく，「人間にとって分かりやすいから」という点に注意する。

問題 5 の解答： 読み出しではメモリの値は変化しない。書き込みでは指定したアドレスの前の値が消え，新しい値に置き換わる。

解説： 読み出しはデータをコピーして取得する動作であり，元のデータは残る。書き込みはデータを上書きする動作であり，元のデータは失われる。

間違えないためのポイント： 読み出しでデータが消えると誤解しやすいが，読み出しではデータは保持される。

問題 6 の解答： アドレス 3 番地とアドレス 4 番地のデータが読み出される。

解説： 「3 番地から 2 バイト分」とは，3 番地を起点として連続する 2 バイト，すなわち 3 番地と 4 番地を指す。

間違えないためのポイント： 「2 バイト分」は「連続する 2 つのアドレス」を意味する。「3 番地と 5 番地」のように飛び飛びのアドレスではない点に注意する。

問題 7 の解答： 以前あったデータは消える（新しいデータで上書きされる）。

解説： 書き込み操作では，指定したアドレスの既存データは失われ，新しいデータに置き換わる。

第 3 章 論理演算と論理ゲート

コンピュータは，0 と 1 という 2 つの値の組み合わせによって，計算やデータの記憶を実現している。本章で学ぶ**論理演算**と**論理ゲート**は，あらゆるデジタル機器の動作原理を支える基盤技術である。これらの概念を理解することで，コンピュータがどのように計算し，記憶するのかという仕組みを把握できる。

本章の学習目標

- 2 進数の基本概念（0 と 1 の二通り）を理解する
- **論理積（AND）** と**論理和（OR）** の定義と違いを説明できる
- 複数ビットに対する論理演算を実行できる
- **AND ゲート**，**OR ゲート**，**NOT ゲート**の動作を理解する
- **半加算器**の仕組みを論理式で表現できる
- コンピュータにおける計算と記憶の実現方法を説明できる

3.1 2 進数の基本

本節では，コンピュータの基礎となる **2 進数**の概念を復習する。第 1 章で学んだ 2 進数の知識をもとに，論理演算への応用について考える。

3.1.1 2 進数とは

2 進数は 0 または 1 の二通りの値をとる。右手の状態を例として考える。

- 右手が下がっている → **0**
- 右手が上がっている → **1**

このように，2 進数では二通りの状態を 0 と 1 で表現する。なお，0 と 1 の割り当てが逆になる場合もある。

3.1.2 変数が 2 つの場合

右手と左手の両方を考えると，組み合わせは **4 通り**になる。

左手	右手
0	0
0	1
1	0

1　　1

n 個の変数がある場合，組み合わせの総数は

$$2^n$$

通りとなる。

3.2 論理積と論理和

本節では，**論理積（AND）** と**論理和（OR）** を学ぶ。これらは論理演算の基本であり，コンピュータの回路設計において重要な役割を果たす。

3.2.1 論理積（AND）

論理積は，**両方とも 1 のときのみ結果が 1** となる演算である。

x	y	論理積
0	0	0
0	1	0
1	0	0
1	1	**1**

3.2.2 論理和（OR）

論理和は，**少なくとも片方に 1 があれば結果が 1** となる演算である。

x	y	論理和
0	0	0
0	1	**1**
1	0	**1**
1	1	**1**

3.2.3 論理和と「選択」の違い

論理和は日常的な「選択」とは異なる。

焼き芋大会への参加を例に考える。「土曜日と日曜日，どちらに参加するか」と聞かれた場合，日常会話では「どちらか一方を選ぶ」ことを期待されることが多い。しかし，論理和の考え方では「両方に参加する」ことも許容される。

- 土曜日の参加：参加しないを 0，参加するを 1 とする
- 日曜日の参加：参加しないを 0，参加するを 1 とする

このとき，土曜日と日曜日の**両方に参加しても構わない**。これが**論理和**の考え方である。

日常会話での「または」は「どちらか一方のみ」という排他的な選択を意味することが多いが，論理演算における論理和は**両方が成立する場合も含む**。

3.2.4 論理積と論理和の真理値表

論理積と論理和を 2 次元の表形式で示す。以下の表では，行が x の値，列が y の値を表す。

論理積（AND）

x ＼ y	0	1
0	0	0
1	0	1

論理和（OR）

x ＼ y	0	1
0	0	1
1	1	1

3.3 複数ビットの論理演算

本節では，複数のビットを一括して処理する論理演算を学ぶ。

3.3.1 ビット単位の論理演算

複数のビットを一括して論理積や論理和を求める場合がある。

4 ビットの例を示す。

- x = 0011
- y = 0101

論理積（AND）の計算：

ビット位置	第 1	第 2	第 3	第 4
x	0	0	1	1
y	0	1	0	1
x AND y	0	0	0	1

結果：0001

論理和（OR）の計算：

ビット位置	第 1	第 2	第 3	第 4
x	0	0	1	1
y	0	1	0	1
x OR y	0	1	1	1

結果：0111

複数ビットの論理演算では，**各ビット位置ごとに独立して**演算が行われる。

3.4 論理ゲート

本節では，論理演算を電子回路として実現する**論理ゲート**を学ぶ。

3.4.1 AND ゲート

AND ゲートは**論理積**を実現する回路素子である。**入力信号 x，y がともに 1 のとき，出力信号 z が 1** となる。

x	y	z
0	0	0
0	1	0
1	0	0
1	1	**1**

3.4.2 OR ゲート

OR ゲートは**論理和**を実現する回路素子である。**入力信号 x，y の少なくとも 1 つが 1 のとき，出力信号 z が 1** となる。

x	y	z
0	0	0
0	1	**1**
1	0	**1**
1	1	**1**

3.4.3 NOT ゲート

NOT ゲートは**否定**を実現する回路素子である。**入力信号 x が 0 のとき出力信号 y は 1** となり，**入力信号 x が 1 のとき出力信号 y は 0** となる。

x	y
0	**1**
1	0

3.5 半加算器

本節では，論理ゲートを組み合わせて算術演算を実現する**半加算器**を学ぶ。

3.5.1 半加算器の動作

半加算器は，2 つの 1 ビット入力 x，y に対して，**和 S（Sum）** と**桁上げ C（Carry）** を出力する回路である。

x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	**1**

x = 1，y = 1 の場合，1 + 1 = 2（10 進数）である。第 1 章で学んだように，これを 2 進数で表すと「10」となり，和 S は 0，桁上げ C は 1 となる。

3.5.2 半加算器の論理式

半加算器の出力を論理式で表すと次のようになる。

和 S は，「x が 1 かつ y が 0」または「x が 0 かつ y が 1」のときに 1 となる。桁上げ C は，「x が 1 かつ y が 1」のときに 1 となる。

3.6 計算と記憶の実現

本節では，コンピュータにおける計算と記憶の仕組みを学ぶ。

3.6.1 計算の実現

計算は，論理ゲート（AND，OR，NOT）の組み合わせで実現可能である。

3.6.2 記憶の仕組み

コンピュータにおける記憶は，以下のような仕組みで実現されている。第 2 章で学んだメモリは，これらの技術によって構成されている。

方式	記憶の原理
DRAM	コンデンサに電荷を蓄えて記憶を行う
SRAM	フリップフロップで記憶を行う
磁気記憶	磁性体金属の皮膜を磁化して記憶を行う
SSD	論理演算の素子の組み合わせで記憶を行う

本章のまとめ

本章では，論理演算と論理ゲートについて学んだ。以下に要点を整理する。

2 進数の基本： 2 進数は 0 または 1 の二通りの値をとる。n 個の変数がある場合，組み合わせの総数は2^n通りとなる。

論理積と論理和： 論理積（AND）は両方とも 1 のときのみ 1 となる。論理和（OR）は少なくとも片方が 1 のとき 1 となる。論理和は「選択」とは異なり，両方が成立する場合も含む。

複数ビットの論理演算： 複数ビットの論理演算は各ビット位置ごとに独立して行われる。

論理ゲート： AND ゲート，OR ゲート，NOT ゲートは論理演算を回路として実現する素子である。

半加算器： 半加算器は論理ゲートの組み合わせで加算を実現する。

計算と記憶： 計算は論理ゲートの組み合わせで実現可能である。記憶は DRAM，SRAM，磁気記憶，SSD などの方式で実現されている。

章末問題

問題 1： 2 進数の変数が 3 つある場合，組み合わせは何通りになるか答えよ。

問題 2： 次の 2 つの 4 ビット値に対して，論理積（AND）と論理和（OR）をそれぞれ求めよ。 - x = 1010 - y = 1100

問題 3： 論理積（AND）と論理和（OR）の違いを，真理値表を用いて説明せよ。

問題 4： 日常会話での「または」と，論理演算における論理和（OR）の違いを説明せよ。

問題 5： 半加算器において，入力 x = 1，y = 1 のとき，和 S と桁上げ C の値をそれぞれ求めよ。また，その理由を説明せよ。

問題 6： コンピュータにおける記憶の仕組みについて，DRAM，SRAM，磁気記憶，SSD の 4 つの方式の特徴をそれぞれ簡潔に説明せよ。

章末問題の解答と解説

問題 1 の解答：

$$2^3 = 8$$

通り

解説： n 個の 2 進数変数がある場合，組み合わせの総数は

$$2^n$$

通りとなる。

問題 2 の解答：

- 論理積（AND）：1000
- 論理和（OR）：1110

解説： 各ビット位置ごとに演算を行う。

ビット位置	第 1	第 2	第 3	第 4
x	1	0	1	0
y	1	1	0	0
AND	1	0	0	0
OR	1	1	1	0

間違えないためのポイント： 各ビット位置を縦に揃えて，1 ビットずつ独立して演算を行う。

問題 3 の解答：

論理積（AND）の真理値表：

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

論理和（OR）の真理値表：

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

違いの説明： 論理積は両方の入力が 1 のときのみ出力が 1 となる。論理和は少なくとも一方の入力が 1 であれば出力が 1 となる。

問題 4 の解答：

日常会話での「または」は，多くの場合「どちらか一方のみ」という選択を意味する。例えば「コーヒーまたは紅茶」と言われた場合，どちらか一方を選ぶことを期待される。

論理演算における論理和（OR）は，**両方が成立する場合も含む**。少なくとも一方が 1 であれば結果は 1 となり，両方が 1 の場合も結果は 1 となる。

問題 5 の解答：

- 和 S = 0
- 桁上げ C = 1

解説： 1 + 1 = 2（10 進数）を 2 進数で表すと「10」となる。下の桁が和 S（0），上の桁が桁上げ C（1）である。

問題 6 の解答：

方式	特徴
DRAM	コンデンサに電荷を蓄えて記憶を行う
SRAM	フリップフロップで記憶を行う
磁気記憶	磁性体金属の皮膜を磁化して記憶を行う
SSD	論理演算の素子の組み合わせで記憶を行う

第 4 章 デジタル画像と画素

デジタル画像は，スマートフォンのカメラ，医療画像診断，自動運転システムなど，現代社会のあらゆる場面で活用されている。本章では，デジタル画像の最も基本的な構成要素である**画素**の概念と，画像情報のコード化方法を学ぶ。これらの知識は，画像処理，コンピュータビジョン，機械学習など，多くの先端技術を理解するための土台となる。

本章の学習目標

- デジタル画像が画素の集まりで構成されていることを理解する
- カラー画像と濃淡画像の違いを説明できる
- 輝度情報のコード化方法を理解する
- カラー画像の成分表現（RGB 方式，輝度・色方式）を説明できる

4.1 画像と画素

本節では，デジタル画像の基本構造と，その最小構成単位である画素について学ぶ。

デジタル画像は，格子状に並んだ小さな点の集まりとして構成されている。この格子の 1 つ 1 つが**画素**（ピクセル）である。画素はデジタル画像を構成する**最小単位**であり，画像の細かさや品質を決定する基本要素となる。

第 1 章で学んだビットが情報の最小単位であるのと同様に，画素はデジタル画像における最小単位である。

4.2 画像の種類

本節では，デジタル画像を，含まれる情報の種類によって分類する方法を学ぶ。

デジタル画像は，含まれる情報の種類によって以下の 2 種類に分類される。

カラー画像

カラー画像は，**輝度と色の情報**を持つ画像である。

濃淡画像

濃淡画像は，**輝度のみの情報**を持つ画像である。モノクロ画像やグレースケール画像とも呼ばれる。

4.3 濃淡画像でのコード化

本節では，濃淡画像における輝度情報を数値として表現する方法を学ぶ。

濃淡画像では，画像の輝度の情報を数値として**コード化**する。第 1 章で学んだように，コンピュータはすべての情報を数値（0 と 1 の組み合わせ）として扱うため，輝度という視覚的な情報も数値に変換する必要がある。

以下に，輝度を 4 段階でコード化する場合の例を示す。

輝度	数値
黒	0
暗い灰色	1
明るい灰色	2
白	3

このようにコード化することで，画像を数値データとして扱うことが可能となる。たとえば，幅 8×高さ 8 の画素で構成される画像の場合，画素数は以下のように計算される。

$$8 \times 8 = 64$$

この 64 個の画素それぞれに 0 から 3 までの数値が割り当てられる。

4.4 カラー画像の成分

本節では，カラー画像を構成する成分の考え方について学ぶ。

カラー画像の表現方法には，主に以下の 2 つの考え方がある。いずれも同じカラー画像を表現するための方法であり，成分の分け方が異なる。

RGB 方式

カラー画像を**R（赤）成分**，**G（緑）成分**，**B（青）成分**の 3 つに分解して考える方法である。

輝度・色方式

カラー画像を**輝度成分**と**色成分**に分解して考える方法である。

4.5 RGB 成分による表現

本節では，RGB 方式によるカラー画像の数値表現を学ぶ。

RGB 方式では，R（赤）成分，G（緑）成分，B（青）成分のそれぞれについて，画素ごとに数値化される。各成分の数値の個数を以下の表に示す。

成分	画素ごとの数値の個数
R（赤）成分	1
G（緑）成分	1
B（青）成分	1
合計	**3**

すべてあわせて，**画素ごとに 3 つの数値**でカラー情報が表現される。

4.6 輝度成分と色成分による表現

本節では，輝度・色方式によるカラー画像の数値表現を学ぶ。

輝度・色方式では，輝度成分と色成分のそれぞれについて，画素ごとに数値化される。各成分の数値の個数を以下の表に示す。

成分	画素ごとの数値の個数
輝度成分	1
色成分	2
合計	**3**

すべてあわせて，**画素ごとに 3 つの数値**でカラー情報が表現される。RGB 方式と同様に，1 つの画素につき合計 3 つの数値が必要となる点は共通している。

本章のまとめ

本章では，デジタル画像の基礎について学んだ。以下に要点を整理する。

画素： デジタル画像は画素（ピクセル）と呼ばれる格子状の点の集まりで構成される。画素はデジタル画像の最小単位である。

画像の種類： 画像はカラー画像（輝度と色の情報）と濃淡画像（輝度のみの情報）に分類される。

コード化： 濃淡画像では，輝度を数値でコード化することで画像をデータとして扱う。

カラー画像の表現： カラー画像の表現には RGB 方式と輝度・色方式がある。いずれの方式でも，カラー画像は画素ごとに 3 つの数値で表現される。

章末問題

問題 1： デジタル画像を構成する最小単位を何と呼ぶか答えよ。

問題 2： カラー画像と濃淡画像の違いを，含まれる情報の観点から説明せよ。

問題 3： 輝度が 4 段階（0〜3）でコード化される濃淡画像において，白と黒はそれぞれどの数値で表されるか答えよ。

問題 4： 幅 10×高さ 10 の画素で構成される画像には，全部で何個の画素があるか答えよ。

問題 5： RGB 方式において，1 つの画素のカラー情報を表現するために必要な数値の個数を答えよ。また，その理由を説明せよ。

問題 6： 輝度・色方式において，輝度成分と色成分はそれぞれ画素ごとにいくつの数値で表されるか答えよ。

章末問題の解答と解説

問題 1 の解答： 画素（ピクセル）

解説： 画素はデジタル画像の最小構成単位である。

問題 2 の解答： カラー画像は輝度と色の情報を持つのに対し，濃淡画像は輝度のみの情報を持つ。

解説： 色の情報を持たない点が，カラー画像と濃淡画像の違いである。

問題 3 の解答： 白＝3，黒＝0

解説： 本章の例では，黒が 0，白が 3 である。

間違えないためのポイント： 黒と白の数値を逆に覚えやすいので注意が必要である。「暗い＝数値が小さい」「明るい＝数値が大きい」と対応づけて記憶するとよい。

問題 4 の解答： 100 個

解説： 画素数は幅×高さで計算される。

$$10 \times 10 = 100$$

問題 5 の解答： 3 つの数値が必要である。理由は，R（赤）成分，G（緑）成分，B（青）成分のそれぞれについて 1 つずつ数値が必要となるためである。

解説： RGB 方式では，赤・緑・青の各成分の強さを個別に数値で表現する。

問題 6 の解答： 輝度成分は画素ごとに 1 つの数値，色成分は画素ごとに 2 つの数値で表される。

解説： 輝度・色方式でも合計 3 つの数値となり，RGB 方式と同じである。

間違えないためのポイント： 「輝度 1 つ＋色 2 つ＝合計 3 つ」と覚える。RGB 方式の「1＋1＋1＝3」と混同しないよう，方式ごとの内訳を区別して理解することが重要である。

第 5 章 画像処理プログラムの基礎

画像処理は，デジタルカメラ，医療画像診断，自動運転など，現代社会のさまざまな分野で活用されている基盤技術である。本章では，画像ファイルからデータを抽出し利用する手順を学ぶ。これらの知識を習得することで，画像処理アプリケーションの開発や，コンピュータビジョン分野への発展的な学習の土台を築くことができる。

本章の学習目標

- PPM 形式の構造（ヘッダと画像データ）を理解し，説明できる
- netpbm コマンド群を用いた画像形式の変換方法を理解できる
- 画像ファイルから RGB 値を抽出する手順を理解できる
- RGB 表色系と YCC 表色系の変換原理を理解できる

5.1 PPM 形式

PPM（Portable PixMap）は，RGB カラー画像を扱うためのフォーマットである。PPM 形式のファイルは**ヘッダ**と**画像データ**で構成される。

5.1.1 ヘッダの構成

ヘッダには以下の情報が含まれる。

- PPM 形式を示す**マジックナンバー**
- 画像の**幅**，**高さ**
- **R**，**G**，**B 値の最大値**

5.1.2 画像データの構成

画像データは以下の特徴を持つ。

- **バイナリ**又は**テキスト形式**で記述される
- 画像の左上のピクセルから始まり，右方向へ進み，1 行分のピクセルが終わると次の行の左端に移る。この順序で，画像の右下のピクセルまで羅列される

5.1.3 PPM 形式の具体例

PPM 形式には，**テキスト形式（P3）**と**バイナリ形式（P6）**の 2 種類がある。

以下に，幅 2 ピクセル，高さ 2 ピクセルの画像をテキスト形式で表現した例を示す。テキスト形式では，空白や改行で値を区切ることができる。

```
P3
2 2
255
255 0 0 0 0 255 0 255 0 0 0 0
```

各部分の意味は以下のとおりである。

部分	内容	説明
1 行目	P3	マジックナンバー（テキスト形式を示す）

2 行目	2 2	幅 2 ピクセル，高さ 2 ピクセル
3 行目	255	R，G，B 値の最大値
4 行目	255 0 0 ...	画像データ

画像データは，各ピクセルについて R の値，G の値，B の値の順で記述される。上記の例では，4 つのピクセルが以下のように格納されている。

ピクセル位置	R	G	B	色
左上	255	0	0	赤
右上	0	0	255	青
左下	0	255	0	緑
右下	0	0	0	黒

5.2 netpbm

netpbm は，各種画像形式の変換を行うコマンド群である。

5.2.1 Windows へのインストール

Windows で netpbm を使用するには，以下の手順でインストールする。

1. GnuWin32 プロジェクトの Web サイトから netpbm のインストーラをダウンロードする
2. ダウンロードしたインストーラを実行し，指示に従ってインストールする
3. インストール先の bin フォルダを環境変数 PATH に追加する

5.2.2 主要なコマンド

コマンド	機能
giftopnm	GIF ファイルを PNM ファイルに変換する
jpegtopnm	JPEG ファイルを PNM ファイルに変換する
pamscale	PNM ファイルの大きさを変える

5.2.3 パイプの使用

これらのコマンドではパイプの使用が可能である。パイプを使用すると，あるコマンドの出力結果を次のコマンドの入力として渡すことができる。以下に使用例を示す。

```
giftopnm in.gif | pamscale -xsize 128 -ysize 128 > out.ppm
```

この例では，GIF ファイルを PNM 形式に変換し，128×128 ピクセルにリサイズして出力ファイルに保存している。

5.3 画像ファイルの操作

画像ファイルからデータを抽出して利用するには，以下の 3 つの処理を順番に行う。

5.3.1 処理 1：画像リストの読み込み

複数の画像ファイルを処理する場合，処理対象となる画像ファイル名をあらかじめテキストファイルにまとめておく。このテキストファイルを**画像リスト**と呼ぶ。プログラムは画像リストを読み込み，記載された画像ファイルを順番に処理する。

5.3.2 処理 2：PPM 形式への変換

画像リストから読み込んだ画像ファイルを PPM 形式に変換する。GIF や JPEG などの形式を PPM 形式に統一することで，以降の処理を共通化できる。

5.3.3 処理 3：画像データの抽出

PPM ファイルから以下の情報を抽出する。

- ヘッダから画像の**幅**と**高さ**を読み込む
- 画像データから **R**，**G**，**B 値**を読み込み，それぞれ別々のリストに格納する

以下に，PPM ファイルを読み込んで RGB 値を抽出する Python プログラムの例を示す。

```
def read_ppm(filename):
    """PPM ファイル（P3 形式）を読み込み，画像情報を返す"""
    with open(filename, 'r') as f:
        # マジックナンバーを読み込む
        magic = f.readline().strip()
        if magic != 'P3':
            raise ValueError('P3 形式の PPM ファイルではありません')

        # コメント行をスキップ
        line = f.readline()
        while line.startswith('#'):
            line = f.readline()

        # 幅と高さを読み込む
        width, height = map(int, line.split())
```

```
    # 最大値を読み込む
    max_val = int(f.readline().strip())

    # 画像データを読み込む
    data = []
    for line in f:
        data.extend(map(int, line.split()))

    # RGB 値を分離してリストに格納する
    col_r = []
    col_g = []
    col_b = []
    for i in range(0, len(data), 3):
        col_r.append(data[i])
        col_g.append(data[i + 1])
        col_b.append(data[i + 2])

    return width, height, col_r, col_g, col_b
```

5.4 画像処理の応用：RGB 表色系から YCC 表色系への変換

PPM 画像ファイルから得られた R，G，B の値を用いてさまざまな画像処理を行うことができる。ここでは，**RGB 表色系から YCC 表色系への変換**を例として示す。

5.4.1 YCC 表色系とは

YCC 表色系とは，色を以下の 3 つの成分で表す表色系である。

- **輝度[Y]**：明るさを表す成分
- **色差[Cb]**：青色方向の色情報を表す成分
- **色差[Cr]**：赤色方向の色情報を表す成分

RGB 表色系が赤・緑・青の 3 色の強さで色を表現するのに対し，YCC 表色系は明るさと色の情報を分離して表現する。画像を輝度 Y のみで表すと，色の情報が除かれ，明るさだけで構成される**モノクロ画像**となる。

第 4 章で学んだ「輝度・色方式」は，この YCC 表色系の考え方に基づいている。

5.4.2 変換式

Y，Cb，Cr の値は R，G，B の値から以下の式で求めることができる。

$$Y = 0.299R + 0.587G + 0.114B$$

$$Cb = -0.172R - 0.339G + 0.511B$$

$$Cr = 0.511R - 0.428G - 0.083B$$

R，G，B が **0〜255** の値のとき，Y は **0〜255**，Cb と Cr は**−128〜127** の値をとる。

5.4.3 実装上の注意点

Cb と Cr は負の値を取りうるため，画像データとして扱う際には値の範囲を調整する必要がある。Cb，Cr の値に **128** を加算することで，値の範囲を 0〜255 に揃えることができる。

以下に，RGB 値を YCC 値に変換する Python プログラムの例を示す。

```python
def rgb_to_ycc(r, g, b):
    """RGB 値を YCC 値に変換する"""
    y = 0.299 * r + 0.587 * g + 0.114 * b
    cb = -0.172 * r - 0.339 * g + 0.511 * b
    cr = 0.511 * r - 0.428 * g - 0.083 * b

    # Cb, Cr に 128 を加算して 0〜255 の範囲に調整
    cb = cb + 128
    cr = cr + 128

    # 値を 0〜255 の範囲にクリップ
    y = max(0, min(255, int(y)))
    cb = max(0, min(255, int(cb)))
    cr = max(0, min(255, int(cr)))

    return y, cb, cr
```

本章のまとめ

本章では，画像処理プログラムの基礎について学んだ。以下に要点を整理する。

PPM 形式： PPM 形式は，RGB カラー画像を扱うためのフォーマットである。ファイルはヘッダ（マジックナンバー，幅，高さ，最大値）と画像データで構成される。

netpbm： netpbm は，各種画像形式の変換を行うコマンド群である。giftopnm，jpegtopnm，pamscale などのコマンドをパイプで連結して使用できる。

画像ファイルの操作： 画像ファイルの操作は，画像リストの読み込み，PPM 形式への変換，画像データの抽出の 3 つの処理で構成される。

RGB 表色系から YCC 表色系への変換： RGB 表色系から YCC 表色系への変換では，輝度 Y と色差 Cb，Cr を計算する。輝度 Y のみを使用するとモノクロ画像を生成できる。

章末問題

問題 1： PPM 形式のヘッダに含まれる 3 つの情報を挙げ，それぞれの役割を説明せよ。

問題 2： PPM 形式のテキスト形式（P3）とバイナリ形式（P6）の違いを説明せよ。

問題 3： 以下の PPM データについて，（a）画像のサイズ（幅×高さ），（b）左下のピクセルの RGB 値を答えよ。

```
P3
3 2
255
255 0 0 0 255 0 0 0 255 128 128 128 64 64 64 192 192 192
```

問題 4： RGB 値（R=150，G=100，B=50）を YCC 表色系に変換したとき，輝度 Y の値を計算せよ。計算過程も示すこと。

問題 5： YCC 表色系において，画像を輝度 Y のみで表すとモノクロ画像となる理由を説明せよ。

章末問題の解答と解説

問題 1 の解答：

PPM 形式のヘッダに含まれる 3 つの情報は以下のとおりである。

1. **マジックナンバー**：ファイルが PPM 形式であること，およびテキスト形式（P3）かバイナリ形式（P6）かを識別するための情報である。
2. **画像の幅と高さ**：画像の水平方向と垂直方向のピクセル数を示す。
3. **R，G，B 値の最大値**：各色成分が取りうる最大値を示す。通常は 255 が指定される。

問題 2 の解答：

テキスト形式（P3）は，RGB 値を 10 進数の数値として記述する。人間が直接読んで内容を確認できる形式である。

バイナリ形式（P6）は，RGB 値をバイナリデータとして記述する。ファイルサイズが小さく，読み込み処理が高速である。

問題 3 の解答：

(a) 画像のサイズ： 幅 3 ピクセル×高さ 2 ピクセル

(b) 左下のピクセルの RGB 値： R=128，G=128，B=128

解説： ヘッダの「3 2」は幅 3 ピクセル，高さ 2 ピクセルを示す。画像データは左上から右へ，上の行から下の行へ順に格納される。

1 行目（上の行）：(255,0,0)，(0,255,0)，(0,0,255) 2 行目（下の行）：(128,128,128)，(64,64,64)，(192,192,192)

左下のピクセルは 2 行目の左端であり，RGB 値は（128，128，128）である。

間違えないためのポイント： 画像データは 3 つの値で 1 ピクセルを構成する。「左下」は最後の行の最初のピクセルである。

問題 4 の解答：

計算式：

$$Y = 0.299R + 0.587G + 0.114B$$

$$Y = 0.299 \times 150 + 0.587 \times 100 + 0.114 \times 50$$

$$Y = 44.85 + 58.7 + 5.7 = 109.25$$

輝度 Y の値は**約 109** である。

問題 5 の解答：

輝度 Y は明るさの情報を表す成分であり，色の情報（色差 Cb，Cr）を含まない。画像を輝度 Y のみで表現すると，各ピクセルは明るさの度合いだけで表され，色の違いは反映されない。明るいピクセルは白に近く，暗いピクセルは黒に近くなるため，白から黒までの濃淡で構成されるモノクロ画像となる。

第 6 章 3 次元画像処理

3 次元画像処理は，CT（Computed Tomography：コンピュータ断層撮影）や MRI（Magnetic Resonance Imaging：磁気共鳴画像法）などの医用画像をはじめ，工業製品の検査，3D ゲーム，VR（Virtual Reality：仮想現実）など，現代社会の様々な分野で活用されている技術である。本章で学ぶ 3 次元画像の扱い方と基本的な処理手法は，これらの応用分野における画像解析の基盤となる知識である。

本章の学習目標

- 2 次元画像から 3 次元画像を構成する概念を理解する
- 3 次元画像のファイル保存形式を理解する
- プログラム中での 3 次元画像データの配列表現を習得する
- Dilation（膨張処理）と Erosion（収縮処理）の原理とプログラム実装を理解する

6.1 3 次元画像の扱い

第 4 章と第 5 章では 2 次元画像処理を学んだ。2 次元画像処理では，写真やスキャン画像のような平面的なデータを扱った。しかし，医療現場で使用される CT や MRI では，人体の内部構造を立体的に捉える必要がある。本節では，2 次元画像の概念をどのように拡張すれば 3 次元画像を表現できるかについて学ぶ。

CT，MRI のように 2 次元画像から 3 次元画像を作るには，2 次元画像を何枚も重ねて厚さを持たせることによって 3 次元画像とする。 CT や MRI では，人体を薄くスライスした断面画像を連続的に撮影する。これらの断面画像を奥行き方向に積み重ねることで，立体的な構造を再現する。

3 次元画像では，従来の 2 次元画像で用いていた **X 座標**（横方向）と **Y 座標**（縦方向）に加えて，奥行き方向を表す **Z 座標**が導入される。

2 次元画像における最小単位は画素（ピクセル）であった。3 次元画像では，これに対応するものとして**ボクセル**と呼ばれる立方体の要素が用いられる。ボクセルは 3 次元空間における最小単位であり，各ボクセルが 1 つの値（輝度値や色情報など）を持つ。

6.2 ファイルへの保存

3 次元画像の概念を理解したところで，次に考えるべきことは，この立体的なデータをどのようにファイルに保存するかである。第 2 章で学んだように，コンピュータのファイルはデータを 1 次元的な並びとして格納するため，3 次元のデータを 1 次元に展開する規則が必要となる。

6.2.1 2 次元画像の場合

まず，2 次元画像のファイル保存形式を確認する。2 次元画像をファイルに保存する際は，画素値を以下の順序で格納する。**Y 座標が同じ画素（同じ行の画素）を左から右へ順に格納し，それを上の行から下の行へ繰り返す。**

```
(x0,y0) (x1,y0) (x2,y0) ・・・ (x255,y0)
(x0,y1) (x1,y1) (x2,y1) ・・・ (x255,y1)
     ・
     ・
     ・
(x0,y255) (x1,y255) (x2,y255) ・・・ (x255,y255)
```

この例では，横 256 画素×縦 256 画素の画像を想定している。ファイルには合計 256×256=65,536 個の画素値が順番に格納される。

6.2.2 3 次元画像の場合

3 次元画像のファイル保存は，2 次元画像の保存形式を拡張したものである。**まず z=0 の断面（2 次元画像）を格納し，続いて z=1 の断面，z=2 の断面と順番に格納していく。** 各 Z 座標における 2 次元画像の格納順序は，前項で説明した 2 次元画像の場合と同様である。

256×256×256 の 3 次元画像の場合，ファイルには合計 256×256×256=16,777,216 個のボクセル値が格納される。3 次元画像は 2 次元画像に比べてデータ量が大幅に増加する。

6.3 プログラム中での扱い

画像データは NumPy 配列を用いて扱う。配列を使用することで，座標を指定して任意のボクセル値に直接アクセスできる。

6.3.1 配列のインデックス順序

3 次元画像では，**image[z][y][x]**のように 3 次元配列を使用する。

重要な点は，**配列のインデックスは[Z][Y][X]の順序**であり，座標の表記(x, y, z)とは順序が逆になることである。例えば，座標(32, 67, 44)のボクセル値を取得するには **image[44][67][32]**と記述する。

6.3.2 カラー画像の配列表現

カラー画像を扱う場合は，RGB（赤・緑・青）の 3 つの色成分を格納するため，4 次元配列 **image[c][z][y][x]**を使用する。第 1 インデックスで色成分を区別する。

- **image[0][z][y][x]** には R（赤）の値
- **image[1][z][y][x]** には G（緑）の値
- **image[2][z][y][x]** には B（青）の値

6.4 Dilation と Erosion

本節では，3 次元画像に対する基本的な処理として，Dilation と Erosion を学ぶ。これらは 2 値画像（ボクセル値が 0 または 1 のみの画像）において物体の形状を変化させる処理であり，ノイズ除去や物体の検出などに用いられる。

6.4.1 Dilation（膨張処理）

Dilation は，各ボクセルについて，そのボクセル自身と周囲 26 個のボクセル（合計 27 個）を調べ，以下の規則で出力値を決定する処理である。

- **27 個のボクセルの中に 1 つでも値が 1 のボクセルがあれば，出力を 1 とする**
- **27 個のボクセルのすべてが 0 であれば，出力を 0 とする**

ここで，**周囲 26 個のボクセル**とは，対象ボクセルを中心とした 3×3×3 の立方体から中心を除いた 26 個のボクセルを指す。この処理により，値が 1 の領域が周囲に 1 ボクセル分拡大する。

以下に Dilation 処理の Python プログラムを示す。

```
import numpy as np

def dilation_3d(image_in):
    """3 次元画像に対する Dilation（膨張処理）を行う"""
    z_size, y_size, x_size = image_in.shape
    image_out = np.zeros_like(image_in)

    for h in range(z_size):
        for i in range(y_size):
            for j in range(x_size):
                # 境界処理
                hm = max(0, h - 1)
                hp = min(z_size - 1, h + 1)
                im = max(0, i - 1)
                ip = min(y_size - 1, i + 1)
                jm = max(0, j - 1)
                jp = min(x_size - 1, j + 1)

                # 3x3x3 の領域を抽出
```

```
                neighborhood = image_in[hm:hp+1, im:ip+1, jm:jp+1]

                # 1 つでも 1 があれば出力は 1
                if np.any(neighborhood == 1):
                    image_out[h, i, j] = 1
                else:
                    image_out[h, i, j] = 0

    return image_out
```

6.4.2 Erosion（収縮処理）

Erosion は，Dilation とは逆の処理である。各ボクセルについて，以下の規則で出力値を決定する。

- **27 個のボクセルのすべてが 1 であれば，出力を 1 とする**
- **27 個のボクセルの中に 1 つでも値が 0 のボクセルがあれば，出力を 0 とする**

この処理により，値が 1 の領域が周囲から 1 ボクセル分縮小する。

以下に Erosion 処理の Python プログラムを示す。

```
import numpy as np

def erosion_3d(image_in):
    """3 次元画像に対する Erosion（収縮処理）を行う"""
    z_size, y_size, x_size = image_in.shape
    image_out = np.zeros_like(image_in)

    for h in range(z_size):
        for i in range(y_size):
            for j in range(x_size):
                # 境界処理
                hm = max(0, h - 1)
                hp = min(z_size - 1, h + 1)
                im = max(0, i - 1)
                ip = min(y_size - 1, i + 1)
                jm = max(0, j - 1)
                jp = min(x_size - 1, j + 1)

                # 3x3x3 の領域を抽出
                neighborhood = image_in[hm:hp+1, im:ip+1, jm:jp+1]

                # すべて1 なら出力も 1
                if np.all(neighborhood == 1):
```

```
                    image_out[h, i, j] = 1
                else:
                    image_out[h, i, j] = 0

    return image_out
```

6.4.3 Dilation と Erosion の比較

両者の違いを以下に整理する。

処理	条件	出力	効果
Dilation	27 個の中に 1 が 1 つでもある	1	物体領域が膨張する
Dilation	27 個すべてが 0	0	
Erosion	27 個すべてが 1	1	物体領域が収縮する
Erosion	27 個の中に 0 が 1 つでもある	0	

6.4.4 組み合わせ処理

Dilation と Erosion を組み合わせることで，より高度な処理が可能となる。

Opening 処理は，Erosion を行った後に Dilation を行う処理である。小さなノイズを除去しつつ，物体の大きさを維持する効果がある。

Closing 処理は，Dilation を行った後に Erosion を行う処理である。物体内部の小さな穴を埋めつつ，物体の大きさを維持する効果がある。

本章のまとめ

本章では，3 次元画像処理の基礎について学んだ。以下に要点を整理する。

3 次元画像の構成： 3 次元画像は，2 次元画像を重ねて厚さを持たせることで構成される。3 次元画像の最小単位は**ボクセル**と呼ばれる。

ファイル保存形式： 3 次元画像のファイル保存では，Z 座標ごとに 2 次元画像を順番に格納する。

配列表現： プログラム中では，3 次元画像を **image[z][y][x]**の 3 次元配列で扱う。配列のインデックスは[Z][Y][X]の順序であり，座標の表記(x, y, z)とは順序が逆である。

Dilation と Erosion： Dilation は物体領域を膨張させ，Erosion は物体領域を収縮させる。これらを組み合わせて Opening 処理や Closing 処理を実現できる。

章末問題

問題 1： 256×256×256 の 3 次元画像をファイルに保存する場合，座標(100, 50, 30)のボクセル値は，ファイルの先頭から何バイト目に格納されるか。ただし，1 ボクセルは 1 バイトとし，座標は 0 から始まるものとする。

問題 2： 3 次元配列において，座標(x, y, z)のボクセル値を取得するための配列アクセスを記述せよ。

問題 3： Dilation の処理において，対象ボクセルとその周囲を含めて参照するボクセルの総数を答えよ。

問題 4： Dilation と Erosion の違いを，処理結果の観点から説明せよ。

問題 5： Opening 処理と Closing 処理の違いを説明せよ。

章末問題の解答と解説

問題 1 の解答：

$$30 \times 256 \times 256 + 50 \times 256 + 100 = 1{,}978{,}980$$

ファイルの先頭を 0 バイト目とすると，座標(100, 50, 30)のボクセル値は **1,978,980 バイト目**に格納される。

解説： 3 次元画像のファイル保存では，Z 座標ごとに 2 次元画像（256×256 バイト）が順番に格納される。計算は Z，Y，X の順序で行う。

1. Z=0 から Z=29 までの 30 枚の断面：30×256×256=1,966,080 バイト
2. Z=30 の断面内で，Y=0 から Y=49 までの 50 行：50×256=12,800 バイト
3. Y=50 の行内で，X=0 から X=99 までの 100 ボクセル：100 バイト

合計：1,966,080+12,800+100=1,978,980 バイト

問題 2 の解答：

```
image[z][y][x]
```

解説： 配列のインデックスは[Z][Y][X]の順序であり，座標の表記(x, y, z)とは順序が逆になる。

問題 3 の解答：

27 個

解説： 対象ボクセルを中心とした 3×3×3 の立方体で，3×3×3=27 個のボクセルを参照する。

問題 4 の解答：

Dilation は，27 個のボクセルの中に 1 つでも 1 があれば出力を 1 とするため，物体領域が膨張する。**Erosion** は，27 個のボクセルがすべて 1 でないと出力が 1 にならないため，物体領域が収縮する。

問題 5 の解答：

Opening 処理は，Erosion を行った後に Dilation を行う処理であり，小さなノイズを除去する効果がある。**Closing 処理**は，Dilation を行った後に Erosion を行う処理であり，物体内部の小さな穴を埋める効果がある。