

放送大学 オンライン授業
続・C言語基礎演習
教科書

目次

第 1 章	“C 言語基礎演習”からの展開	3
1.1	switch 文	3
1.2	do 文	9
1.3	グローバル変数	15
1.4	論理演算子・条件演算子の短絡評価	16
1.5	C 言語の演算子	17
第 2 章	型とデータ表現	20
2.1	文字と数値を表す型	20
2.2	暗黙の型変換	25
2.3	コンピュータにおけるデータの表現	28
2.4	sizeof 演算子、size_t 型、typedef 宣言	34
第 3 章	ビットを対象とする演算	38
3.1	ビットを対象とする演算の種類	38
3.2	ビットを対象とする演算の演算子	42
3.3	ビットを対象とする演算の利用	45
第 4 章	乱数	57
4.1	乱数の生成	57
4.2	乱数の利用	61
第 5 章	配列	66
5.1	配列の導入	66
5.2	配列の使い方	72
5.3	配列の操作	75
第 6 章	ポインタ	80
6.1	メモリと変数のアドレス	80
6.2	アドレスを格納する変数	82

6.3	ポインタが指すオブジェクト	85
第 7 章	関数の引数におけるポインタ	89
第 8 章	配列とポインタ	95
8.1	配列の要素を指すポインタ	95
8.2	関数への配列の受け渡し	102
8.3	型修飾子 <code>const</code>	109
第 9 章	2 次元配列	112
第 10 章	文字列	120
10.1	文字と文字列	120
10.2	文字列の仕組み	120
10.3	<code>char</code> 型の配列への文字列の格納	127
10.4	文字列の操作	132
10.5	文字列に関するライブラリ関数	136
10.6	文字列の配列	139
第 11 章	構造体	142

第 1 章

“C 言語基礎演習” からの展開

“C 言語基礎演習” では省略した項目があります。それらのうち、`switch` 文と `do` 文を導入します。また、グローバル変数、短絡評価について説明し、C 言語のすべての演算子を紹介します。

1.1 `switch` 文

“C 言語基礎演習” では、分岐を実現する文として `if` 文を学びました。`switch` 文も分岐を実現する文です。`switch` 文の書式を、図 1.1 に示します。

```
switch (制御式)
{
    case 定数1:
        文11
        文12
        ⋮
    case 定数2:
        文21
        文22
        ⋮
    ⋮
    default:
        文x1
        文x2
        ⋮
}
```

図 1.1 `switch` 文の書式

制御式には、評価した値が整数になる式を書きます。`case` の後ろには、整数の定数や文字定数を 1 つと、`:` (コロン) を書きます。その後ろに、必要なだけ文を書きます。必要なければ文がなくてもかまいません。`while` 文や `for` 文と異なり、複数の文を書いた場合でも複合文にする必要はありません。

switch 文の動作は次のとおりです。switch 文に到達すると、まず制御式が評価されます。その値が定数₁と等しければ、プログラムの流れが、case 定数₁: の直後に移動し、文₁₁ から実行されます。定数₂と等しければ case 定数₂: の直後に、定数₃と等しければ case 定数₃: の直後に移動します。いずれの定数とも等しくなければ、default: の直後に移動します。

制御式を評価した値が定数₁と等しい場合は、case 定数₁: の直後に移動して、文₁₁、文₁₂、…を実行した後、case 定数₂: と書かれている部分は無視されて、文₂₁、文₂₂、…も実行されます。その後も同様です。このように、移動した先から switch 文の { } の中の最後まで、すべての文が実行されます。ソースコード 1.1 に例を示します。

ソースコード 1.1 switch 文の使用例 (ex101.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int hour;
6
7      printf("明日何時に起きますか (5-10) > ");
8      scanf("%d", &hour);
9
10     switch (hour)
11     {
12         case 5:
13         case 6:
14             printf("散歩して、");
15         case 7:
16             printf("読書して、");
17             printf("メールを読んで、");
18         case 8:
19             printf("朝食を食べて、");
20         default:
21             printf("歯を磨いてから出勤しましょう。\\n");
22     }
23
24     return 0;
25 }
```

実行例 1

明日何時に起きますか (5-10) > 5 ↩

散歩して、読書して、メールを読んで、朝食を食べて、歯を磨いてから出勤しましょう。

実行例 2

明日何時に起きますか (5-10) > 7 ↩

読書して、メールを読んで、朝食を食べて、歯を磨いてから出勤しましょう。

実行例 3

明日何時に起きますか (5-10) > 8 ↩

朝食を食べて、歯を磨いてから出勤しましょう。

実行例 4

明日何時に起きますか (5-10) > 10 ↩

歯を磨いてから出勤しましょう。

明日何時に起きるかを 5~10 の整数で入力してもらい、int 型の変数 `hour` に格納します。10 行目から 22 行目が `switch` 文です。

- 制御式 `hour` を評価した値、つまり `hour` の値が 5 であれば、`case 5:` の直後に移動します。`case 5:` には対応する文がありませんが、そのまま先に進み、`case 6:` 以降の 5 つの `printf()` 文が順次実行されます。その際、`case 6:`、`case 7:`、`case 8:`、`default:` は無視されます。その結果、

散歩して、読書して、メールを読んで、朝食を食べて、歯を磨いてから出勤しましょう。

と表示されます。

- `hour` の値が 7 であれば、`case 7:` の直後に移動し、

読書して、メールを読んで、朝食を食べて、歯を磨いてから出勤しましょう。

と表示されます。

- `hour` の値が 8 であれば、`case 8:` の直後に移動し、

朝食を食べて、歯を磨いてから出勤しましょう。

と表示されます。

- `hour` の値が 5、6、7、8 のいずれでもなければ、`default:` の直後に移動し、

歯を磨いてから出勤しましょう。

と表示されます。

ここで、`case` 定数: と `default:` の部分をラベルといいます。ラベルは、プログラムの流れ

が移動する先を示す目印の役割を持っています。ここまでの説明では“無視される”と書きましたが、ラベルは文ではありませんので、実行されることはありません。

switch 文では、移動先のラベルより後ろにあるすべての文が実行されます。しかし、分岐をする場合は、分岐先によって別々の処理を実行したいことがほとんどです。break 文を使うと、switch 文を終了させることができます。図 1.2 のように書くと、制御式を評価した値が 定数₁ と等しければ 文₁₁、文₁₂、… だけが実行され、定数₂ と等しければ 文₂₁、文₂₂、… だけが実行されます。いずれの定数とも等しくなければ 文_{x1}、文_{x2}、… だけが実行されます。なお、default: の後ろの break 文を省いても、プログラムの動作は同じです。

```
switch (制御式)
{
    case 定数1:
        文11
        文12
        ⋮
        break;
    case 定数2:
        文21
        文22
        ⋮
        break;
    ⋮
    default:
        文x1
        文x2
        ⋮
        break;
}
```

図 1.2 switch 文における break 文の使用

break 文を使う例を、ソースコード 1.2 に示します。

ソースコード 1.2 switch 文における break 文の使用例 (ex102.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char grade;
6
7      printf("成績を入力してください (S, A, B, C, D) > ");
8      scanf("%c", &grade);
9
```

```

10     switch (grade)
11     {
12         case 'S':
13             printf("大変素晴らしいです。\\n");
14             break;
15         case 'A':
16             printf("素晴らしいです。\\n");
17             break;
18         case 'B':
19             printf("まずまずです。\\n");
20             break;
21         case 'C':
22             printf("もう少し頑張りましょう。\\n");
23             break;
24         case 'D':
25             printf("もっと頑張りましょう。\\n");
26             break;
27         default:
28             printf("入力が正しくありません。\\n");
29             break;
30     }
31
32     return 0;
33 }

```

実行例 1

成績を入力してください (S, A, B, C, D) > A ↵
 素晴らしいです。

実行例 2

成績を入力してください (S, A, B, C, D) > C ↵
 もう少し頑張りましょう。

実行例 3

成績を入力してください (S, A, B, C, D) > E ↩
入力が正しくありません。

キーボードから、“S”、“A”、“B”、“C”、“D” のいずれかの成績 (評語) を入力してもらいます。10 行目から始まる `switch` 文で、入力された文字に応じてメッセージを表示します。想定外の成績を入力した場合は、`default:` の直後に移動し、“入力が正しくありません。”と表示します。

`switch` 文の制御式には、評価した値が整数になる式を書くことができましたが、この例では `char` 型の変数 (`grade`) を使っています。`char` 型の変数の値は整数と見なすことができます。詳しくは、次章で説明します。

`switch` 文では、ほとんどの場合、各ラベルに対応する `break` 文を書くことになります。コンパイラやその設定によっては、ラベルに対応する `break` 文がないと誤りの可能性があると判断して、コンパイル時に警告を行ってくれます。必要がなければ `default:` は書かなくてもかまいません。その場合、制御式の値がいずれの定数とも等しくなければ、何も実行されません。

すでに説明したとおり、`switch` 文の `case` の後ろには整数の定数や文字定数を書きます。このとき、定数の値が重複してはいけません。正しいラベルと誤ったラベルの例を示します。

```
switch (制御式)
{
    case 5:      /* 整数の定数なので正しい */
    case 'a':    /* 文字定数なので正しい */

    /*
     * 以下は誤り
     * (コンパイルできない)
     */
    case 5:      /* 同じ値のラベルが複数 (ここより上に case 5: がある場合) */
    case 1.5:    /* 小数 */
    case "a":    /* 文字列リテラル */
    case i:      /* 変数 */
}
```

練習課題 1 (ex103.c)

入力した月に対応する季節を表示するプログラムを作ってください。

- 月は 1 から 12 の整数で入力します。
- 3 月から 5 月を “春”、6 月から 8 月を “夏”、9 月から 11 月を “秋”、12 月から 2 月を “冬” とします。
- 0 以下、または 13 以上の整数が入力されたときは、エラーメッセージを表示してください。
- 条件分岐には、if 文ではなく switch 文を使ってください。

実行例 1

月 (1-12) > 6 ↵
夏です。

実行例 2

月 (1-12) > 10 ↵
秋です。

実行例 3

月 (1-12) > 15 ↵
入力が正しくありません。

1.2 do 文

図 1.3 に、2 種類の反復を示します。

“C 言語基礎演習” では、反復 1 に相当する while 文と for 文を学びました。ここでは、反復 2 に相当する do 文を説明します。

do 文の書式は、

```
do 文 while (制御式);
```

です。“while” というキーワードが入っていますが、do 文と while 文は異なる文です。while 文や for 文と同様、do 文の “文” も **ループ本体** と呼びます。

プログラムの流れが do 文に到達すると、まずループ本体が実行されます。次に制御式が評価されます。その値が 0 でなければ、再びループ本体が実行され、実行後に再度制御式が評価されます。制御式を評価した値が 0 であれば、do 文が終了し、次の処理 (do 文の次の文) に進みます。

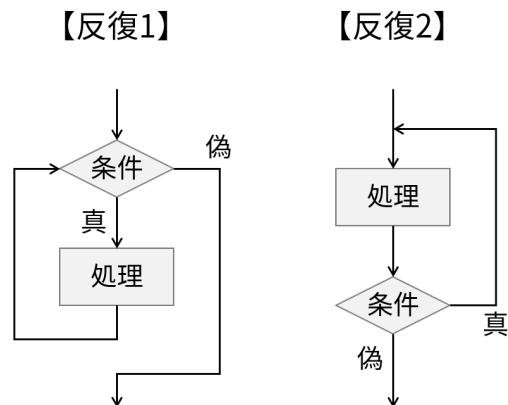


図 1.3 反復

do 文のループ本体が複合文である場合、その中で使う **break 文** と **continue 文** の働きは、while 文や for 文と同じです。つまり **break 文** は do 文を終了させ、**continue 文** はループ本体の残りの処理を飛ばして制御式の評価に進めます。do 文の使用例をソースコード 1.3 に、このプログラムの処理の流れを図 1.4 に示します。

ソースコード 1.3 do 文の使用例 1 (ex104.c)

```

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

実行結果

1 2 3 4 5

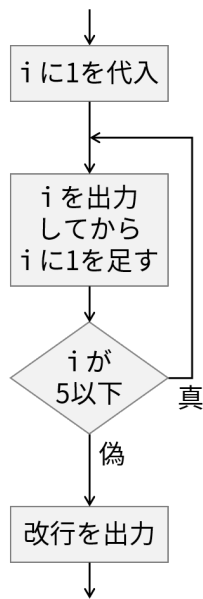


図 1.4 ソースコード 1.3 のプログラムの処理の流れ

このプログラムを実行すると、9 行目の `do` 文に到達した時点で変数 `i` の値は 1 になっています。この `do` 文では、まずループ本体である `printf()` 文が実行されます。その結果、画面に “1 ” (数字の後ろにスペースあり) と表示され、`i` の値が 2 に更新されます。ループ本体の実行が終わったので、制御式である `i <= 5` が評価されます。制御式を評価した値は 0(偽) ではないので、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 に更新されます。これを繰り返し、“3 ”、“4 ”、“5 ” と表示されます。“5 ” を表示した後、`i` の値が 6 に更新されます。その次に制御式を評価すると値は 0(偽) なので、`do` 文の実行が終了します。

ソースコード 1.3 の `do` 文は、`while` 文や `for` 文で記述しても大差ありません。`do` 文の使用が適した例を、ソースコード 1.4 に示します。

ソースコード 1.4 `do` 文の使用例 2 (ex105.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int answer;
6
7      do
8      {
9          printf("3 + 5 = ");
10         scanf("%d", &answer);
11     } while (answer != 8);
  
```

```

12
13     printf("正解です。\\n");
14
15     return 0;
16 }

```

実行例

```

3 + 5 = 7 ↵
3 + 5 = 10 ↵
3 + 5 = 8 ↵
正解です。

```

このプログラムは、 $3 + 5$ の結果を解答させます。解答が正しければ“正解です。”と表示します。不正解であれば再度解答させます。正解を入力するまで繰り返し解答させる部分が、7 行目から始まる `do` 文です。“ $3 + 5 =$ ”と表示し、キーボードから読み取った整数を `answer` に格納する処理を、正解の 8 が入力されるまで繰り返します。

これと同等の処理を、`while` 文を使って書き直した例を、ソースコード 1.5 に示します。

ソースコード 1.5 ソースコード 1.4 の `while` 文での書き換え例 1 (ex106.c)

```

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

```

19 }

ソースコード 1.5 では、while 文のループ本体と同じ処理を、while 文の前にも記述しています。そうすれば、“3 + 5 = ” と表示し、キーボードから読み取った整数を `answer` に格納する処理を少なくとも 1 回は実行できます。しかし、同じ処理が 2 箇所に記述されており冗長です。“3 + 5” を “11 - 3” に直したくなったときは、2 箇所を修正する必要があります。

同じ処理を複数の箇所に書かないように工夫した例を、ソースコード 1.6 に示します。

ソースコード 1.6 ソースコード 1.4 の while 文での書き換え例 2 (ex107.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int answer = 0, is_answered = 0;
6
7      while (!is_answered || answer != 8)
8      {
9          is_answered = 1;
10         printf("3 + 5 = ");
11         scanf("%d", &answer);
12     }
13
14     printf("正解です。\\n");
15
16     return 0;
17 }
```

ソースコード 1.6 では、変数 `is_answered` を導入しています。`is_answered` の値を、ループ本体を 1 回も実行していないとき 0 に、1 回でも実行したとき 1 になるようにします。while 文の制御式は `!is_answered || answer != 8` です。変数の意味も含めて日本語にすると、“ループ本体を 1 回も実行していない、または解答が 8 でない” となります。ループ本体を 1 回も実行していないときは、`answer` の値にかかわらずループ本体を実行して、解答を入力させます。ループ本体を 1 回でも実行すると、`is_answered` の値は 1 になり、`answer` にはキーボードから入力された解答が入っています。制御式の `!is_answered`、つまり“ループ本体を 1 回も実行していない”は成り立たなくなります。その場合、`answer != 8`、つまり“解答が 8 でない”が成り立てば、再びループ本体を実行します。解答が 8 であれば while 文が終了し、14 行目の `printf()` 関数で“正解です。”と表示します。

このソースコードでは、ループ本体を 1 回でも実行したか、という状態を表す変数 (`is_answered`)

を導入しました。何らかの状態や結果を真理値として保持する変数を、**フラグ**と呼ぶことがあります。

ソースコード 1.4 では変数 `answer` を初期化しませんでした。ソースコード 1.6 では 0 で初期化しています。ソースコード 1.4 では、10 行目で変数 `answer` に値が格納された後、11 行目でそれが参照されます。それに対してソースコード 1.6 では、11 行目で変数 `answer` に値が格納される前に、7 行目で参照される可能性があります。変数に値を格納する前に参照することは望ましくありません^{*1}ため、初期化を行いました^{*2}。

なお、`answer` を 0 で初期化しておけば、`is_answered` を使わずに、`while (answer != 8)` だけでも正しく動作します。ただし、その場合は、初期化した意図がわかりづらいと思います。

ここでは、`do` 文を `while` 文で書き直す例を 2 つ示しました。比較すると明らかなように、このプログラムでは `do` 文が適しています。

練習課題 2 (ex108.c)

次の説明に従って、練習課題 1 で作ったプログラムを改変してください。

練習課題 1 では、月として 0 以下、または 13 以上の整数が入力されたときは、エラーメッセージを表示しました。エラーメッセージを表示するのではなく、もう一度入力してもらうように改変してください。その処理には `do` 文を使ってください。`switch` 文でエラーメッセージを表示する処理が実行されることはなくなりますが、残しておいてもかまいません。

実行例 1

月 (1-12) > 6 ↵
夏です。

実行例 2

月 (1-12) > 15 ↵
月 (1-12) > -4 ↵
月 (1-12) > 0 ↵
月 (1-12) > 9 ↵
秋です。

^{*1} 初期化されていない、または値が代入されていない変数の参照はバグの原因になります。

^{*2} 実は、この後説明する論理演算子の短絡評価という規則により、ソースコード 1.6 において `answer` を初期化しなかった場合でも、値の格納前に参照されることはありません。

1.3 グローバル変数

“C 言語基礎演習”では、変数はブロックの先頭で宣言し、スコープはそのブロック内であると説明しました。C 言語の変数は関数の外で宣言することもでき、その場合は、ソースコード内のすべてがスコープになります。このような変数を**グローバル変数**、または**大域変数**といい、これまで使ってきたような変数を**ローカル変数**、または**局所変数**といいます。グローバル変数の動作を確認するプログラムを、ソースコード 1.7 に示します。

ソースコード 1.7 グローバル変数の使用例 (ex109.c)

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

実行結果

```
(1) gvar = 1
(2) gvar = 2
```

3 行目で宣言、初期化している変数 `gvar` は、関数 (`main()` と `inc_gvar()`) の外で宣言されて

いるので、グローバル変数です。gvar は、main() 関数内でも inc_gvar() 関数内でも使えます。main() 関数内の 1 つ目の printf() 関数 (9 行目) で、グローバル変数である gvar の値を表示します。この時点では、gvar の値を初期化時から変更していないので、“1” が表示されます。次に、11 行目で呼び出される inc_gvar() 関数で、gvar の値をインクリメントします。その後、main() 関数内の 2 つ目の printf() 関数 (13 行目) で、gvar の値の “2” を表示します。main() 関数が始まる時点で gvar が用意されていることと、main() 関数と inc_gvar() 関数が参照している gvar が同じ変数であることが確認できます。

1.4 論理演算子・条件演算子の短絡評価

論理積の演算子 && を用いた式 a && b を評価するとき、まず a が評価されます。a を評価した結果が真であった場合、a && b は、b を評価した結果が真なら 1、偽なら 0 です。a を評価した結果が偽であった場合、b の評価結果にかかわらず a && b は 0 です。a を評価した結果が偽であることがわかった時点で a && b の結果が確定しますので、b を評価する必要はありません。そこで、C 言語では、b の評価を省略します。

論理和の演算子 || を用いた式 a || b を評価するときも、まず a が評価されます。その結果が真であることがわかった時点で、a || b の結果が 1 と確定しますので、b の評価は省略されます。

条件演算子 ?: を使った式 a ? b : c を評価するときは、まず a が評価されます。その結果が真であれば、b だけが評価され、c は評価されません。偽であれば、c だけが評価され、b は評価されません。

以上のような評価方法を、**短絡評価**といいます。短絡評価を行うことにより、無駄な処理をなくしています。

短絡評価の性質を使って、if 文を使わずに条件分岐を書くことができます。

x が 1 なら y に 2 を代入するという処理を if 文を使って書くと、次のようになります。

```
if (x == 1)
{
    y = 2;
}
```

これと同等の処理を、&& を使って書くと、次のようになります*3。

```
(x == 1) && (y = 2);
```

x == 1 が真の場合、y = 2 が評価、つまり実行されます。x == 1 が偽の場合、y = 2 は評価されません。これにより、x が 1 のときだけ、y に 2 を代入する処理が行われます。

|| 演算子を使えば、次のように書けます。

*3 == の優先順位は && よりも高いため、x == 1 を () で囲む必要はありません。

```
(x != 1) || (y = 2);
```

`x != 1` が偽、つまり `x` が 1 のときだけ、`y` に 2 を代入する処理が行われます。

`x` と `y` のうち大きい方を `max` に、小さい方を `min` に代入する処理を `if` 文を使って書くと、次のようになります。

```
if (x > y)
{
    max = x;
    min = y;
}
else
{
    max = y;
    min = x;
}
```

これと同等の処理を、`?:` を使って書くと、次のようになります。

```
x > y ? (max = x, min = y) : (max = y, min = x);
```

`x > y` が真の場合は `(max = x, min = y)` のみが、偽の場合は `(max = y, min = x)` のみが評価されます。1つの式しか記述できない場所 (条件演算子の2つ目と3つ目の被演算子) に複数の式を記述するため、コンマ演算子を使っています。

短絡評価の性質を使った条件分岐は、プログラミング言語によっては多用されます。ただし、C言語で使われることはほとんどありません。短絡評価によって評価されない可能性がある部分に代入等の処理を記述するときは、その性質を理解して、注意して記述してください。

1.5 C 言語の演算子

C 言語のすべての演算子を、表 1.1 に示します。

このうち、“C 言語基礎演習”で登場していない演算子を見てみましょう。

優先順位 1 の `[]` は、**配列**に関する演算子で、第 5 章で登場します。

優先順位 1 の `.` と `->` は、**構造体**に関する演算子で、第 11 章で登場します。

優先順位 2 の `*`(単項) は**ポインタ**に関する演算子です。優先順位 2 の `&`(単項) は“C 言語基礎演習”に登場しましたが、説明は省略していました。これも**ポインタ**に関する演算子です。ポインタは第 6 章で説明します。

優先順位 2 の `sizeof` は次章で説明します。

表 1.1 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) ~ &(単項) *(単項) ! sizeof	右
3	()(キャスト)	右
4	*(二項) / %	左
5	+(二項) -(二項)	左
6	<< >>	左
7	< <= > >=	左
8	== !=	左
9	&(二項)	左
10	^	左
11		左
12	&&	左
13		左
14	?:	右
15	= += -= *= /= %= <<= >>= &= ^= =	右
16	,	左

優先順位 2 の ~ (チルダ)、6 の << と >>、9 の &(二項)、10 の ^ (ハット)、11 の | は、**ビットを対象とする演算**を行う演算子です。ビットを対象とする演算は、第 3 章で説明します。二項演算子である <<、>>、&、^、| には複合代入演算子が用意されています。それぞれ <<=、>>=、&=、^=、|= で*4、優先順位は 15 です。

+ と - は、単項演算子のものと二項演算子のものがあることを、“C 言語基礎演習”で説明しました。役割は似ていると感じるかもしれませんが、単項演算子の +、- と二項演算子の +、- は異なる演算子です。& と * にも、単項演算子のものと二項演算子のものがあります。いくつか例を示します。

例 1: `scanf("%d", &i)`

`scanf()` 関数の呼び出しの第 2 引数で使われている `&` は、第 6 章で説明する単項演算子です。

例 2: `printf("%d\n", a & b)`

`printf()` 関数の呼び出しの第 2 引数で使われている `&` は、第 3 章で説明する二項演算子です。

*4 たとえば、`a = a << b` は `a <<= b` と書けます。

例 3: $*x = *y * 3$

1つ目と2つ目の $*$ は第6章で説明する単項演算子、3つ目の $*$ は乗算を行う二項演算子です。

第 2 章

型とデータ表現

2.1 文字と数値を表す型

“C 言語基礎演習”では、文字、整数、小数を格納するために、それぞれ `char` 型、`int` 型、`double` 型の変数を使いました。また、プログラム中に直接書いた整数の定数 (3、8 など) は `int` 型であること、小数の定数 (0.5、2.4 など) は `double` 型であることなどを説明しました。ここでは、表 2.1 に示す新しい型を導入します (`char` 型、`int` 型、`double` 型も含めています)。

文字型

`char` 型、`signed char` 型、`unsigned char` 型は**文字型**と呼ばれます。文字型という名称ですが、整数を表す型とを考えてください。コンピュータ上のデータは、文字も数値もビット (0 と 1) の並びで表されています。データ自体に、これは文字でこれは数値といった意味づけがされているわけではありません。文字か数値かは、**データの解釈の仕方が違うだけです**。`char` 型の変数 `c` に、文字定数 `'A'` を代入すると、ほとんどの環境では `c` のデータは `01000001` になります。図 2.1 に示すように、これを文字として読むと文字 “A” になり、整数として読むと 65 になります*2。変数 `c` の値を `printf()` 関数で表示する場合、変換指定として `%c` を使うと “A”、`%d` を使うと “65” と表示されます。

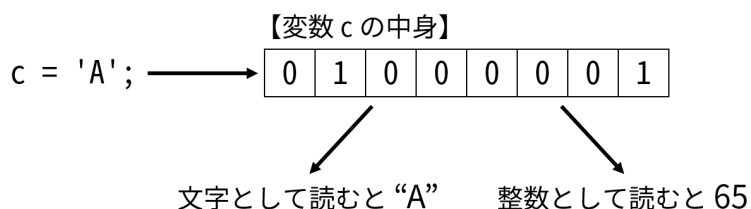


図 2.1 ビット列の解釈

*1 十進数として出力する場合。

*2 “C 言語基礎演習”で発展として説明しました。

表 2.1 文字と数値を表す型

種類	型	型の別名	printf() 関数の変換指定*1
文字型	char		%d
	signed char		%d
	unsigned char		%d
整数型	signed short int	short int signed short short	%d
	unsigned short int	unsigned short	%u
	signed int	int signed	%d
	unsigned int	unsigned	%u
	signed long int	long int signed long long	%ld
	unsigned long int	unsigned long	%lu
浮動小数点型	float		%f
	double		%f
	long double		%Lf

数を表すときのプラスとマイナスの符号を、英語で signといいます。signed は**負の整数も表せる**という意味で、unsigned は**0 と正の整数のみを表せる**という意味です。つまり、signed char 型は負の整数、0、正の整数を表すことができ、unsigned char 型は 0 と正の整数を表すことができます。signed を“符号付き”、unsigned を“符号なし”と呼びます。signed も unsigned も付かない char 型は、signed char 型か unsigned char 型のどちらかと同じです。どちらと同じであるかは処理系によって違います。signed char 型と unsigned char 型の変数に値を入れて、printf() 関数で表示する例を、ソースコード 2.1 に示します。

ソースコード 2.1 signed char 型と unsigned char 型の変数の使用例 (ex110.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      signed char sc = -4;
6      unsigned char uc = 6;

```

```

7
8     printf("sc = %d\n", sc);
9     printf("uc = %d\n", uc);
10
11     return 0;
12 }

```

実行結果

```

sc = -4
uc = 6

```

整数型

signed short int 型、unsigned short int 型、signed int 型、unsigned int 型、signed long int 型、unsigned long int 型は、整数を表す型です*3。signed、unsigned の 2 種類と、short int、int、long int の 3 種類の組み合わせで、計 6 種類あります (図 2.2)。

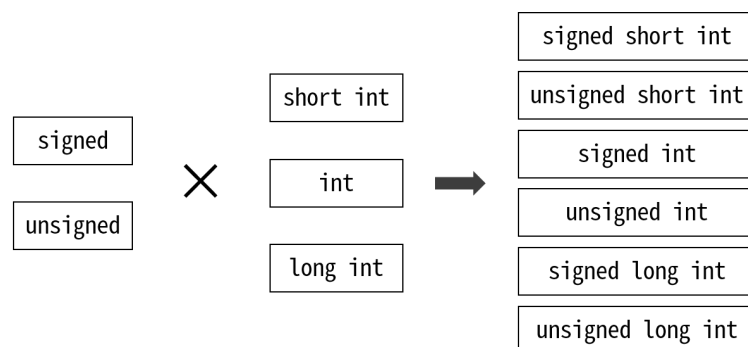


図 2.2 6 種類の整数型

signed と unsigned は、文字型と同様に、符号付きと符号なしを表しています。short int と int と long int は、表現できる範囲が違います。int は、short int と同じか、より広い範囲の整数を表現できます。long int は、int と同じか、より広い範囲の整数を表現できます。この性質は、signed、unsigned のいずれについても成り立ちます (図 2.3)。

ここまでに登場した型の範囲を調べてみましょう。授業のサイトに掲載している ex111.c をコンパイルして実行してみてください。

*3 char 型、signed char 型、unsigned char 型も整数を表す型です。

signed short int型 が表せる範囲 $\subseteq$ signed int型 が表せる範囲 $\subseteq$ signed long int型 が表せる範囲

unsigned short int型 が表せる範囲 $\subseteq$ unsigned int型 が表せる範囲 $\subseteq$ unsigned long int型 が表せる範囲

$A \subseteq B \cdots B$ はAを含む、またはAとBは同じ

図 2.3 short int、int、long int が表現できる範囲

Windows 11 上の MinGW-w64(gcc version 13.2.0) での実行結果

文字型

char 型 : -128 ~ 127
signed char 型 : -128 ~ 127
unsigned char 型 : 0 ~ 255

整数型

signed short int 型 : -32768 ~ 32767
unsigned short int 型 : 0 ~ 65535
signed int 型 : -2147483648 ~ 2147483647
unsigned int 型 : 0 ~ 4294967295
signed long int 型 : -2147483648 ~ 2147483647
unsigned long int 型 : 0 ~ 4294967295

FreeBSD 13.1 (amd64) 上の Clang 13.0.0 での実行結果

文字型

char 型 : -128 ~ 127
signed char 型 : -128 ~ 127
unsigned char 型 : 0 ~ 255

整数型

signed short int 型 : -32768 ~ 32767
unsigned short int 型 : 0 ~ 65535
signed int 型 : -2147483648 ~ 2147483647
unsigned int 型 : 0 ~ 4294967295
signed long int 型 : -9223372036854775808 ~ 9223372036854775807

`unsigned long int` 型 : 0 ~ 18446744073709551615

どちらの処理系でも、`char` 型は `signed char` 型と同じでした。1 つ目の処理系では `long int` は `int` と同じでしたが、2 つ目の処理系ではより広い範囲の整数を表現できました。このように、値の範囲は処理系によって違います。

整数型の型名は、`signed` と `int` を省略できるという規則があります。たとえば `signed short int` は、`signed` を省略すれば `short int`、`int` を省略すれば `signed short`、両方省略すれば `short` です。ただし、`signed int` は両方省略すると名前がなくなってしまうので、片方しか省略できません。これまで使ってきた `int` 型は、`signed int` 型の別名でした。一般的に使われる型名を、表 2.2 に示します。

表 2.2 一般的に使われる型名

長い型名	一般的に使われる型名
<code>signed short int</code>	<code>short</code>
<code>unsigned short int</code>	<code>unsigned short</code>
<code>signed int</code>	<code>int</code>
<code>unsigned int</code>	<code>unsigned</code> または <code>unsigned int</code>
<code>signed long int</code>	<code>long</code>
<code>unsigned long int</code>	<code>unsigned long</code>

なお、前述のとおり、`char` は `signed char` の略ではありませんので注意してください。

浮動小数点型

`float` 型、`double` 型、`long double` 型は、小数を表す型です。文字型、整数型と違って、`signed` と `unsigned` はありません。`float` 型、`double` 型、`long double` 型のいずれも、負の小数、0.0(小数の 0)、正の小数が表せます。

これら 3 つの型は、表現できる範囲が違います。ただし、整数型のように、いくつからいくつまで、と簡単には説明できません。“浮動小数点”の意味と合わせて、後ほど説明します。

`scanf()` 関数の変換指定

非負整数を読み取って `unsigned int` 型の変数に格納するための `scanf()` 関数の変換指定は、`%u` です(表 2.3)。その他の型に対応する変換指定は、この授業では省略します。

表 2.3 scanf() 関数の第 1 引数で使う変換指定 (“C 言語基礎演習” の表 5.3 に %u を追加)

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

【発展】 C 言語の型

C 言語の char 型は、処理系によって signed char 型のこともあれば、unsigned char 型のこともあります。また、char 型、short 型、int 型、long 型などが何ビットなのか (0 と 1 が何個入れられるか) は決められていません。具体的なビット数に依存しないようにプログラミングする必要があります。C99^aでは int32_t(32 ビットの符号付き整数)、uint64_t(64 ビットの符号なし整数) など、ビット数が決められた型が用意されています。この授業の範囲では、次章で説明するビット単位の演算を行うときに、型のビット数を意識する必要があります。その他の箇所ではそれほど気にする必要はありません。

なお、Java にも short 型、int 型、long 型がありますが、こちらはそれぞれ 16、32、64 ビットと決められています。

^a “C 言語基礎演習” の “8.2 真理値” で、発展として説明しました。

2.2 暗黙の型変換

キャスト演算子を使えば、型を変更することができます。“C 言語基礎演習” 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に、キャスト演算を行いました。

ここで、ソースコード 2.2 について考えます。

ソースコード 2.2 自動的な型変換の例 (ex112.c)

```

1  #include <math.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6      short s1, s2;
7      float f1, f2;
8  }
```

```

9      s1 = 5;
10     printf("s1 = %d\n", s1);
11
12     s2 = 100000;
13     printf("s2 = %d\n", s2);
14
15     f1 = 2.5;
16     f2 = sqrt(f1);
17     printf("f2 = %f\n", f2);
18
19     return 0;
20 }

```

Windows 11 上の MinGW-w64(gcc version 13.2.0) での実行結果

```

s1 = 5
s2 = -31072
f2 = 1.581139

```

MinGW-w64 ではコンパイル時に警告が表示されますが、コンパイルと実行は行えます。

9 行目では、`short` 型の変数 `s1` に、`int` 型の値 5 を代入しています。この場合、`int` 型の 5 が、自動的に `short` 型に変換されます。

```
s1 = (short)5;
```

のような動作が行われていると考えてください。

12 行目では、`short` 型の変数 `s2` に、`int` 型の値 100000 を代入しています。`s1` への代入と同様に、

```
s2 = (short)100000;
```

のような動作が行われます。しかし、13 行目の `printf()` 関数で `s2` の値を表示すると、`-31072` となっています。前節で紹介した `ex111.c` の実行結果では、`short` 型 (`signed short int` 型) で表せる最大の数は 32767 でした。そのような処理系では、`short` 型の変数に 100000 を入れることはできません。このような場合に、型変換後の値 (`int` 型の値 100000 を `short` 型に変換した値) がどうなるかは、処理系に依存します。

15 行目では、`float` 型の変数 `f1` に、`double` 型の値 2.5 を代入しています。これまで同様、

```
f1 = (float)2.5;
```

のような動作が行われ、f1 に 2.5 が代入されます。16 行目では、sqrt() 関数の呼び出しを行っています。sqrt() 関数の引数は 1 つで、引数、戻り値ともに、double 型です。この場合、float 型の変数 f1 の値が sqrt() 関数に渡される際に、自動的に double 型に変換されます。また、sqrt() 関数からの戻り値が変数 f2 に代入される際に、自動的に float 型に変換されます。つまり、

```
f2 = (float)sqrt((double)f1);
```

のような動作が行われています。

このように、代入文における = の左の変数と右の式の値や、関数呼び出しにおける実引数と仮引数で型が異なる場合、**自動的に型が変換される**ことがあります。その際、s2 = 100000 の例のように、変換前の値が変換後の型で表現しきれないことがあります。その場合の動作は、型の組み合わせによって、C 言語の規格で決められていることもあれば^{*4}、処理系に依存することもあります。この授業では細かい規則は省略します。

算術演算では、int 型同士の演算の結果は int 型に、double 型同士の演算の結果は double 型になります。3 + 2 は、int 型と int 型の加算です。結果は int 型の 5 です。3.5 - 2.5 は、double 型同士の減算です。結果は double 型の 1.0 です。int 型の 1 ではないことに注意してください。

では、3 * 2.5 はどうでしょうか。3 は int 型、2.5 は double 型です。算術演算で int 型と double 型の演算を行う場合、int 型の値が double 型に変換され、結果も double 型になります。3 * 2.5 が計算される様子を、図 2.4 に示します。

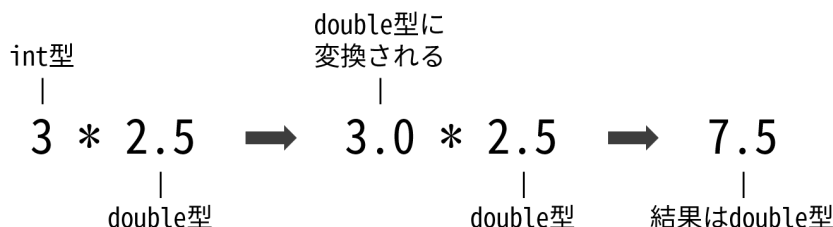


図 2.4 3 * 2.5 の計算

算術演算における型変換については、int 型と double 型以外の組み合わせなど、細かい規則は省略します。

ここまでに説明したような自動的な型変換を**暗黙の型変換**と呼ぶことがあります。

^{*4} たとえば、int 型の負の値 x を unsigned int 型に変換すると、(unsigned int 型で表現できる最大値) + x + 1 になることが規格で決められています。unsigned int 型で表現できる最大値が 4,294,967,295 であれば、(unsigned int)-5 は 4,294,967,295 + (-5) + 1 = 4,294,967,291 になります。

2.3 コンピュータにおけるデータの表現

コンピュータの中でデータがどのように表現されているかを知っていると、ビットを対象とする演算、ポインタ、文字列など、以降で扱う内容の理解が深まります。そこで、一度 C 言語から離れて、コンピュータにおける数値データと文字データの表現方法を学びます*5。

コンピュータ内では、すべてのデータがビット (0 と 1) の並び (ビット列) で表現されています*6。

符号なし整数 (非負整数) の表現

コンピュータで符号なし整数を表現する場合は、その数を二進数で表したときの 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 44 を二進数で表すと 101100 です。コンピュータ内では、十進数の 44 は 101100 というビット列で表されます。ただし、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。符号なし整数を表すために 32 ビットの領域を使うなら、十進数の 44 は、00000000 00000000 00000000 00101100 というビット列で表されます。

符号なし整数を表すために x ビットの領域を使う場合、表現できる最小値と最大値は次のようになります。

- 最小値は、すべてのビットが 0 の場合で、その値は 0 です。
- 最大値は、すべてのビットが 1 の場合で、その値は $2^x - 1$ です。

表 2.4 に、符号なし整数に使うビット数と、表現できる数の範囲を示します。

表 2.4 符号なし整数に使うビット数と、表現できる数の範囲

ビット数	表現できる数の範囲
8 ビット (1 バイト)	0~255
16 ビット (2 バイト)	0~65,535
32 ビット (4 バイト)	0~4,294,967,295 (約 43 億)
64 ビット (8 バイト)	0~18,446,744,073,709,551,615 (約 1845 京)

符号付き整数の表現

符号付き整数を表現する場合は、最上位のビット (Most Significant Bit, **MSB**) で正負を表し、残りのビットで大きさを表します。MSB は、非負整数なら 0、負の整数なら 1 にします。

*5 本節は、“情報理論とデジタル表現 ('19)” の印刷教材の第 2 章と第 11 章に記述した内容とほぼ同じです。

*6 データを、コンピュータで扱える形式であるビット列にすることや、ビット列にしてコンピュータ内に記録することを“表現する”や“表す”と呼んでいます。

非負整数の MSB 以外のビットは、符号なし整数と同じです。たとえば、符号付き整数を表すために 8 ビットの領域を使うなら、十進数の 44 は、00101100 というビット列で表されます。先頭の 0 が負ではないことを表しており、残りの 0101100 が大きさ (44) を表しています。

負の整数の MSB 以外のビットの表し方には何通りかありますが、最も広く使われている 2 の補数表現を紹介します。負の整数の 2 の補数表現は、絶対値が同じ正の数のビット列を反転させて、1 を足します。ビット列の反転とは、0 を 1 に、1 を 0 にする操作です。符号付き整数を表すために 8 ビットの領域を使うなら、 -44 の 2 の補数表現は、次の手順で求められます。

1. 絶対値が同じ正の数のビット列を求めます。
 -44 と絶対値が同じ正の数、つまり 44 のビット列は、00101100 です。
2. 各ビットを反転します。
00101100 のビットを反転すると 11010011 です。
3. ビット列を二進数と考えて、1 を足します。
11010011 に 1 を足すと 11010100 です。

つまり、 -44 の 2 の補数表現は 11010100 です。

符号付き整数を表すために x ビットの領域を使う場合、表現できる最小値と最大値は次のようになります。

- 最小値は、MSB が 1 で残りのビットがすべて 0 の場合で、その値は -2^{x-1} です。
- 最大値は、MSB が 0 で残りのビットがすべて 1 の場合で、その値は $2^{x-1} - 1$ です。

表 2.5 に、符号付き整数に使うビット数と、表現できる数の範囲を示します。

表 2.5 符号付き整数に使うビット数と、表現できる数の範囲

ビット数	表現できる数の範囲
8 ビット (1 バイト)	$-128 \sim 127$
16 ビット (2 バイト)	$-32,768 \sim 32,767$
32 ビット (4 バイト)	$-2,147,483,648 \sim 2,147,483,647$ (約 -21 億 ~ 21 億)
64 ビット (8 バイト)	$-9,223,372,036,854,775,808 \sim 9,223,372,036,854,775,807$ (約 -922 京 ~ 922 京)

表 2.4 と 2.5 を比較してみてください。同じビット数で表現できる最大値が、符号付き整数は符号なし整数の約半分になっています。その代わり、符号付き整数では、正の数とほぼ同じ個数の負の数も表現できます。

なお、“符号なし整数”とは非負整数のみを表現できる方式で、“符号付き整数”とは負の数も含めた整数を表現できる方式です。0 と正の数は“符号なし整数”を用いて、負の数は“符号付き整数”を用いるという意味ではないので注意してください。

小数の表現

十進数における“10”や、二進数における“2”を**基数**といいます。基数を明記するには、たとえば十進数なら $5678_{(10)}$ 、二進数なら $101_{(2)}$ のように、数の後ろに括弧付きの基数を下付き文字で表記します。

コンピュータで小数を表現するときには、**浮動小数点数**が用いられます。浮動小数点数では、数値を

$$(\text{符号})(\text{仮数}) \times (\text{基数})^{(\text{指数})}$$

の形式 (指数形式) で表現します。ここでの符号とは、正負の記号 (+ または -) です。たとえば、基数を 10、仮数を 1 以上 10 未満とすると、123.4 は $+1.234 \times 10^2$ 、 -0.000987 は -9.87×10^{-4} と表せます (表 2.6)。

表 2.6 指数形式の例

数	符号	仮数	基数	指数
$123.4 = +1.234 \times 10^2$	+	1.234	10	2
$-0.000987 = -9.87 \times 10^{-4}$	-	9.87	10	-4

コンピュータ上では、基数として 2 を用います。たとえば、十進数の $9.625_{(10)}$ を二進数にすると $1001.101_{(2)}$ です。仮数の整数部が 1 になるように小数点の位置を決めると、 $1001.101_{(2)}$ は、 $+1.001101_{(2)} \times 2^3$ と表せます*7。基数は 2 と決まっていますので、符号 (+)、仮数 ($1.001101_{(2)}$)、指数 ($3_{(10)}$) の組で、小数を表すことができます。

浮動小数点数を 32 ビットで表現する場合の内部表現の例*8を、図 2.5 に示します。MSB が符号を表し、残りのビットが指数を格納する部分と仮数を格納する部分に分けられています。浮動小数点数の内部表現は複雑なので詳細は省略しますが、指数、仮数ともに、対応する値を加工して*9格納します。

浮動小数点数に対して、私たちが普段使っているような小数が**固定小数点数**です。表 2.7 に、いくつかの正の数をそれぞれ十進数の固定小数点数と浮動小数点数で表した例を示します。

固定小数点数は、小数点の左側 (整数部) と右側 (小数部) をそれぞれ記録します。表 2.7 の一番上の例では 0 (整数部) と 125 (小数部)、二番目の例では 3 (整数部) と 14 (小数部) を記録します。

固定小数点は小数点の位置が 1 の位の右側に固定されている、浮動小数点は小数点の位置が固定されていない、と考えることができます。

*7 仮数は二進数、基数と指数は十進数で書いていますので、注意してください。

*8 単精度浮動小数点数と呼ばれる形式です。

*9 たとえば、仮数の整数部は必ず 1 なので、記録しないことにより 1 ビット節約しています。

各ビットの役割

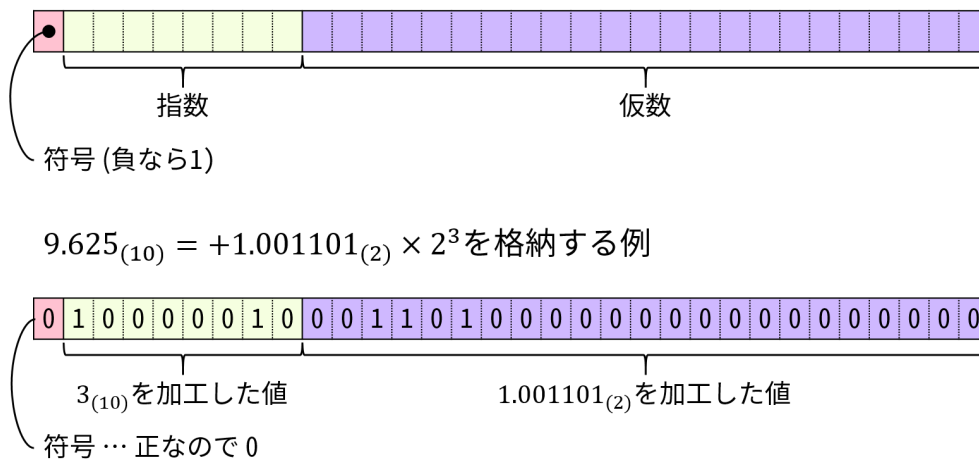


図 2.5 浮動小数点数の内部表現の例

表 2.7 固定小数点数と浮動小数点数 1

固定小数点数	浮動小数点数
0.125	1.25×10^{-1}
3.14	3.14×10^0
164.5	1.645×10^2
2000	2×10^3

表 2.8 に、大きい数と小さい (0 に近い) 数をそれぞれ固定小数点数と浮動小数点数で表した例を示します。

表 2.8 固定小数点数と浮動小数点数 2

[illegible]

表 2.8 の一番下の数は、固定小数点数では、0(整数部) と 00000000000000000000000000612(小数部) を保持する必要があります。これに対して、浮動小数点数では、6.12(仮数) と -27 (指数) を保持するだけです。このように、浮動小数点数を用いると、絶対値が非常に大きい数や非常に小さい数を、少ない桁数で表すことができます。ここでは十進数で説明しましたが、コンピュータ上で

使われる二進数でも同じです。

整数型が表せる範囲は、最小値と最大値で説明できました。浮動小数点型が表せる範囲は、仮数と指数に割り当てられた領域で表現できる範囲、となります。

【発展】 小数の精度

コンピュータで小数を扱うときは、精度に関する問題が起こることがあります。その代表的な例である丸め誤差と情報落ちを紹介します。

十進小数の $0.1_{(10)}$ を二進数で表すと、 $0.000110011\cdots_{(2)}$ で、“0.0”の後ろに“0011”が無限に続く循環小数になります。十進小数を二進小数に変換すると、ほとんどの場合循環小数になります。このとき、どこかの桁で端数処理をする必要がありますが、端数処理をした後で十進小数に変換しても、元の値には戻りません。たとえば、 $0.1_{(10)} = 0.000110011\cdots_{(2)}$ をコンピュータで扱うとき、二進数の小数第 10 位以下を切り捨て、 $0.000110011_{(2)}$ にします。これを十進小数に戻しても $0.099609\cdots_{(10)}$ であり、 $0.1_{(10)}$ には戻りません。このような“丸め”による誤差を**丸め誤差**といいます。

十進数の浮動小数点数で、仮数は小数点以下 3 桁までしか表せないとしします。123000 は 1.23×10^5 、0.045 は 4.5×10^{-2} で、いずれも小数点以下 3 桁の範囲に収まっています。しかし、この 2 つの数値を足すと $123000 + 0.045 = 123000.045 = 1.23000045 \times 10^5$ となり、仮数が小数点以下 3 桁を超えてしまいます。加算の結果から小数第 4 位以下を切り捨てると、 1.230×10^5 となり、0.045 の情報が消えてしまいます。このような現象を**情報落ち**といいます。ここでは十進数で説明しましたが、二進数でも同じです。図 2.5 の例では、仮数を格納する領域は 23 ビットでしたが、十進数にすると 7 桁程度の精度しかありません。授業のサイトに掲載している `adv11.c` と `adv12.c` は、それぞれ丸め誤差と情報落ちを確認するプログラムです。

`adv11.c` は、100 万分の 1 を 100 万回足すプログラムです。丸め誤差のために、結果は 1 にはなりません。

adv11.c の実行結果例

100 万分の 1 を 100 万回足すと 1.009039

`adv12.c` は絶対値の大きい数と小さい数を足すプログラムです。 $123000 + 0.045$ の結果は、123000.046875 になり、小さい数である 0.045 の精度が落ちています。123000 を 10 倍した $1230000 + 0.045$ の結果は、1230000.000000 になり、0.045 の情報がなくなってしまいました。

adv12.c の実行結果例

```
f1 = 123000.000000
f2 = 0.045000
f1 + f2 = 123000.046875

f1 = 1230000.000000
f2 = 0.045000
f1 + f2 = 1230000.000000
```

なお、adv11.c と adv12.c は、処理系によって結果が異なる可能性があります。ここでは小数の格納に float 型を使いましたが、double 型や long double 型でも同じような現象が起こります。

文字の表現

文字に対応するビット列を文字コードといいます。たとえば、文字“A”の文字コードは 01000001 です。コンピュータ上に“A”を記録すると、1 バイトの領域に 01000001 と格納されます。

文字とビット列を対応させる体系も文字コードと呼ばれます。この意味での文字コードには、ASCII コード、Shift_JIS、UTF-8 などがあります。

ASCII コードにおけるビット列と文字の対応を、図 2.6 に示します。

ASCII コードは、7 ビットで文字を表す文字コードです。文字“A”のビット列は、上位 3 ビットが 100、下位 4 ビットが 0001 で 1000001 です。これを 1 バイトの領域に格納するために、先頭に 0 を付けて 01000001 とします。上位 3 ビットが 010、下位 4 ビットが 0000 の位置にある SP はスペースです。上位 3 ビットが 011 の列には、“0”～“9”が格納されています。これは、数字 (“0”～“9”の文字) であって、数値ではありません。

上位 3 ビットが 000 または 001 である 32 文字と、一番右下の DEL には背景色を付けています。これらの文字は、コンピュータの出力装置や通信装置を制御するための特殊な文字で、**制御文字**と呼ばれます。これまで、printf() 関数で改行を出力するために、エスケープシーケンス \n を使ってきました。\\n は、制御文字 LF を表します。

ASCII コードが定義している文字は、英数字と一部の記号だけです。Shift_JIS は、ASCII コードで定義された文字に加え、大半の日本語の文字を扱えます。UTF-8 は、世界中の文字を表現できます。

ある文字に対応するビット列 (前者の意味での文字コード) は、(後者の意味での) 文字コードによって異なることがあります。文字“A”のビット列は、ASCII コード、Shift_JIS、UTF-8 のいずれも 01000001 です。文字“あ”のビット列は、Shift_JIS なら 10000010 10100000、UTF-8

		上位3ビット							
		000	001	010	011	100	101	110	111
下位4ビット	0000	NUL	DLE	SP	0	@	P	`	p
	0001	SOH	DC1	!	1	A	Q	a	q
	0010	STX	DC2	"	2	B	R	b	r
	0011	ETX	DC3	#	3	C	S	c	s
	0100	EOT	DC4	\$	4	D	T	d	t
	0101	ENQ	NAK	%	5	E	U	e	u
	0110	ACK	SYN	&	6	F	V	f	v
	0111	BEL	ETB	'	7	G	W	g	w
	1000	BS	CAN	(	8	H	X	h	x
	1001	HT	EM	)	9	I	Y	i	y
	1010	LF	SUB	*	:	J	Z	j	z
	1011	VT	ESC	+	;	K	[	k	{
	1100	FF	FS	,	<	L	\	l	
	1101	CR	GS	-	=	M	]	m	}
	1110	SO	RS	.	>	N	^	n	~
	1111	SI	US	/	?	O	_	o	DEL

図 2.6 ASCII コードのビット列と文字の対応

なら 11100011 10000001 10000010 です*¹⁰。UTF-8 を使って “あ” を記録すると、3 バイトの領域に 11100011 10000001 10000010 と格納されます。0 と 1 の数が多くなると読みづらくなるため、十六進数で表記することも一般的です。十六進数で表すと、文字 “あ” のビット列は、Shift_JIS なら 82 A0、UTF-8 なら E3 81 82 になります。

2.4 sizeof 演算子、size_t 型、typedef 宣言

sizeof 演算子を使うと、値や型の大きさが得られます。sizeof 演算子の書式は、

sizeof(型名)

または

sizeof 式

です。たとえば、sizeof(int) で int 型の大きさが、sizeof 3.5 で double 型の定数である 3.5 の大きさが得られます。ここでの “大きさ” とは、バイト数と考えてください。sizeof 演算子で得られる値の型は size_t 型です。size_t 型は、符号なし整数を表す型 (unsigned int や unsigned long) の別名として定義されています。具体的にどの型の別名であるかは、処理系によって異なります。size_t 型の値を printf() 関数で表示するための変換指定は、size_t 型が

*¹⁰ 読みやすいように、8 ビットごとにスペースを入れています。

`unsigned int` 型の別名の場合は `%u`、`unsigned long` 型の別名の場合は `%lu` です。ただし、これでは処理系によって変換指定を変える必要があり、不便です^{*11}。そのため、この授業では `int` 型にキャストして `%d` を使うことにします。授業のサイトに掲載している `ex113.c` は、様々な型や値の大きさを表示するプログラムです。コンパイルして、実行してみてください。ここでは、MinGW-w64 ではなく、派生元の MinGW での実行結果を示します。

Windows 11 上の MinGW(gcc version 9.2.0) での実行結果	
<code>sizeof(char)</code>	<code>= 1</code>
<code>sizeof(short)</code>	<code>= 2</code>
<code>sizeof(int)</code>	<code>= 4</code>
<code>sizeof(long)</code>	<code>= 4</code>
<code>sizeof(float)</code>	<code>= 4</code>
<code>sizeof(double)</code>	<code>= 8</code>
<code>sizeof(long double)</code>	<code>= 12</code>
<code>sizeof(size_t)</code>	<code>= 4</code>
<code>sizeof 'A'</code>	<code>= 4</code>
<code>sizeof 3</code>	<code>= 4</code>
<code>sizeof 3.0</code>	<code>= 8</code>
<code>sizeof "A"</code>	<code>= 2</code>
<code>sizeof "ABC"</code>	<code>= 4</code>
<code>sizeof (3 + 5)</code>	<code>= 4</code>
<code>sizeof (3.0 + 5)</code>	<code>= 8</code>

この実行結果では、`char` 型は 1 バイト、`short` 型は 2 バイト、`int` 型と `long` 型は 4 バイトでした。この処理系では、`int` 型と `long` 型は同じで、格納できる値の最小値は $-2,147,483,648$ 、最大値は $2,147,483,647$ です^{*12}。

`float` 型は 4 バイト、`double` 型は 8 バイト、`long double` 型は 12 バイトです。この処理系では、`float` 型は、図 2.5 に示した形式をしています。

`size_t` 型は 4 バイトです。この処理系では、`size_t` は `unsigned long` の別名として定義され

^{*11} C99 という C 言語の規格では、`size_t` 型のための変換指定 `%zu` が導入されました。2023 年時点の MinGW-w64 では、コンパイル時にオプション `-D__USE_MINGW_ANSI_STDIO` を指定すると、`%zu` が使えます。

^{*12} 表 2.5 も参照してください。

ており、`sizeof(long)` の結果と同じです*¹³。

文字定数の型は `int` 型です*¹⁴。`sizeof 'A'` は、`int` 型の大きさである 4 になります。3 は `int` 型の定数なので、`sizeof 3` は 4 です。3.0 は `double` 型の定数なので、`sizeof 3.0` は 8 です。

文字列リテラルの大きさは、第 10 章を学習すれば理解できます。`sizeof "A"` は 2、`sizeof "ABC"` は 4 です。

`3 + 5` を評価すると `int` 型の 8 で、`3.0 + 5` を評価すると `double` 型の 8.0 です。このことから、`sizeof (3 + 5)` が 4、`sizeof (3.0 + 5)` が 8 になることが理解できます。

次に、異なる処理系での実行結果を示します。

FreeBSD 13.1 (amd64) 上の Clang 13.0.0 での実行結果	
<code>sizeof(char)</code>	= 1
<code>sizeof(short)</code>	= 2
<code>sizeof(int)</code>	= 4
<code>sizeof(long)</code>	= 8
<code>sizeof(float)</code>	= 4
<code>sizeof(double)</code>	= 8
<code>sizeof(long double)</code>	= 16
<code>sizeof(size_t)</code>	= 8
<code>sizeof 'A'</code>	= 4
<code>sizeof 3</code>	= 4
<code>sizeof 3.0</code>	= 8
<code>sizeof "A"</code>	= 2
<code>sizeof "ABC"</code>	= 4
<code>sizeof (3 + 5)</code>	= 4
<code>sizeof (3.0 + 5)</code>	= 8

`long` 型が 4 バイトでなく 8 バイト、`long double` 型が 12 バイトでなく 16 バイト、`size_t` 型が 4 バイトでなく 8 バイトである点が、1 つ目の処理系との違いです。この処理系でも、`size_t` は `unsigned long` の別名ですが、`unsigned long` 型の大きさが違うため、`size_t` 型の大きさも

*¹³ 処理系によらず、`int` 型と `unsigned int` 型の大きさは同じで、`long` 型と `unsigned long` 型の大きさは同じです。

*¹⁴ “C 言語基礎演習” で発展として説明しました。

違います。

なお、`sizeof` 演算子で得られる大きさの領域が、値の表現にすべて使われるとは限りません。たとえば、`ex113.c` の 1 つ目の実行結果では `long double` 型の大きさは 12 バイトでしたが、そのうち 80 ビット (10 バイト) しか使いません。

前述のとおり、`size_t` 型は既存の型の別名として定義されています。型に別名を付けるには、次の書式の `typedef` 宣言を行います。

```
typedef 型名 別名;
```

`size_t` 型の `typedef` 宣言はヘッダに記載されており、`<stdio.h>` をインクルードすれば取り込まれます。適当なソースコードから、プリプロセス済みのソースコードを生成してご覧ください^{*15}。MinGW でプリプロセスを行ったソースコードには、

```
typedef unsigned int size_t;
```

と記載されています。ここでは、`size_t` を `unsigned int` の別名として定義しています。MinGW-w64 でプリプロセスを行ったソースコードには、この授業では扱わない記述があって読みにくいですが、`unsigned long long`^{*16} の別名として定義されています。

^{*15} “C 言語基礎演習” の “17.4 4 工程の確認” を参照してください。

^{*16} C99 で導入された型です。

第3章

ビットを対象とする演算

3.1 ビットを対象とする演算の種類

C 言語では、整数を表すビット列に対して、6 種類の演算を行うことができます。

ビット単位の論理積

ビットの 1 を真、0 を偽と考えると、2 つのビットの**論理積**は、表 3.1 のとおりです。

表 3.1 2 つのビットの論理積

b_1	b_2	b_1 と b_2 の論理積
1	1	$\implies$ 1
1	0	$\implies$ 0
0	1	$\implies$ 0
0	0	$\implies$ 0

ビット単位の論理積は、2 つのビット列の対応するビットごとに論理積を求めた結果を返します。
ビット単位の論理積の例を、図 3.1 に示します。

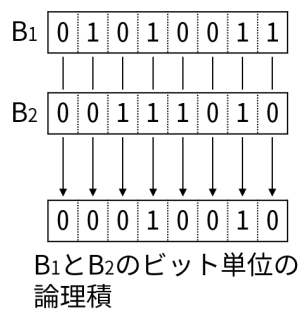


図 3.1 ビット単位の論理積の例

B_1 のビット列は 01010011、 B_2 のビット列は 00111010 です。 B_1 と B_2 のビット単位の論理積は、00010010 となります。

ビット単位の論理和

2 つのビットの論理和は、表 3.2 のとおりです。

表 3.2 2 つのビットの論理和

b_1	b_2	b_1 と b_2 の論理和
1	1	1
1	0	1
0	1	1
0	0	0

ビット単位の論理和は、2 つのビット列の対応するビットごとに論理和を求めた結果を返します。ビット単位の論理和の例を、図 3.2 に示します。

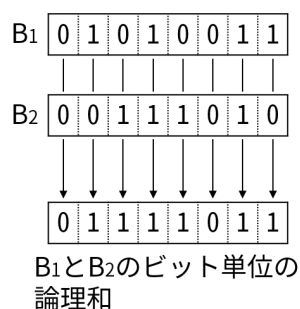


図 3.2 ビット単位の論理和の例

ビット単位の排他的論理和

2 つの真理値の排他的論理和 (exclusive or, **XOR**) とは、片方が真でもう片方が偽のときに真、両方とも真、または両方とも偽のときに偽となる論理演算です (表 3.3)。2 つのビットの排他的論理和は、表 3.4 のとおりです。

ビット単位の排他的論理和は、2 つのビット列の対応するビットごとに排他的論理和を求めた結果を返します。ビット単位の排他的論理和の例を、図 3.3 に示します。

表 3.3 排他的論理和

A	B		A XOR B
真	真	$\Rightarrow$	偽
真	偽	$\Rightarrow$	真
偽	真	$\Rightarrow$	真
偽	偽	$\Rightarrow$	偽

表 3.4 2つのビットの排他的論理和

b_1	b_2		b_1 と b_2 の排他的論理和
1	1	$\Rightarrow$	0
1	0	$\Rightarrow$	1
0	1	$\Rightarrow$	1
0	0	$\Rightarrow$	0

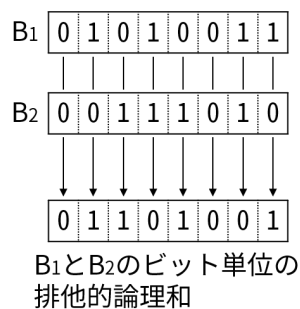


図 3.3 ビット単位の排他的論理和の例

ビットの反転

1 ビットを**反転**させると、0 なら 1 に、1 なら 0 になります (表 3.5)。

表 3.5 1 ビットの反転

b		b の反転
1	$\Rightarrow$	0
0	$\Rightarrow$	1

ビットの反転は、ビット列のすべてのビットを反転させた結果を返します。ビットの反転の例を、図 3.4 に示します。

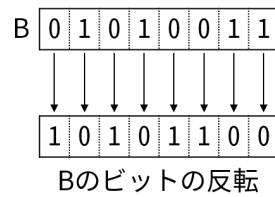


図 3.4 ビットの反転の例

ビットの反転を行った結果を、元の値の**1の補数**と呼びます。1の補数に1を足したものが、負の整数の2の補数表現のもとになっている**2の補数**です。二進数で考えると、元の値と1の補数を足すと、元の値と同じ桁数ですべての桁が1になります。元の値と2の補数を足すと、元の値より桁数が1増えて、最上位の桁が1、その他の桁が0になります。元の値に1の補数および2の補数を足す例を、図 3.5 に示します。

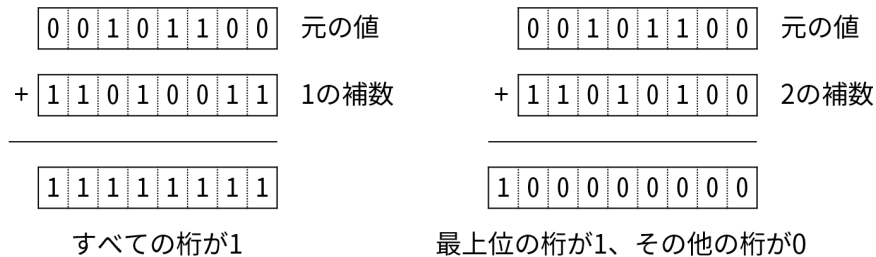


図 3.5 元の値と、1の補数および2の補数の加算

左シフト

これまでの、ビットという語が、0か1の値を表すことも、その値の入れ物を表すこともありました。左シフトと次の右シフトの説明では、入れ物をビット、ビットに入っている0または1を(ビットの)値と呼びます。**左シフト**は、すべてのビットの値を、指定された数だけ上位のビットに移動する演算です。その際、上位のあふれた値を捨て、下位の空いたビットには0を入れます。01010011を3ビット左シフトする様子を、図 3.6 に示します。

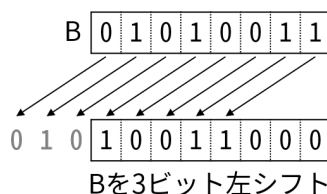


図 3.6 左シフトの様子

値を3ビットずつ上位にずらした結果、先頭の010があふれて捨てられます。下位の3ビット

には 0 が格納され、演算結果は 10011000 になります。

右シフト

右シフトは、すべてのビットの値を、指定された数だけ下位のビットに移動する演算です。その際、下位のあふれた値を捨てます。上位の空いたビットに入れる値は、次の 2 通りの方法があります。

論理シフト (論理右シフト) では、上位の空いたビットに 0 を入れます。01010011 と 10101100 をそれぞれ 3 ビット論理右シフトする例を、図 3.7 に示します。いずれも値が 3 ビットずつ下位に移動し、あふれた値は捨てられ、上位の 3 ビットには 0 が格納されます。

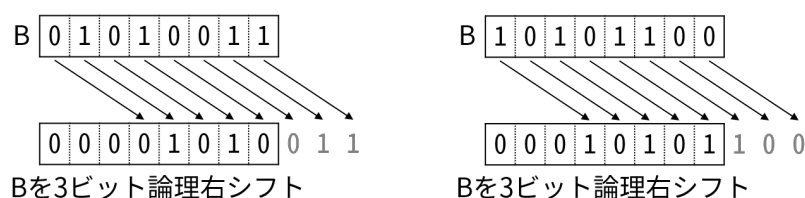


図 3.7 論理右シフトの様子

算術シフト (算術右シフト) では、上位の空いたビットに、元の値の最上位ビットと同じ値を入れます。つまり元の値の最上位ビットが 0 なら 0 を、1 なら 1 を入れます。図 3.7 の例と同じビット列をそれぞれ 3 ビット算術右シフトする例を、図 3.8 に示します。01010011 は最上位ビットの値が 0 なので、演算結果は論理右シフトと同じです。10101100 は最上位ビットの値が 1 なので、3 ビットずつ下位に移動した結果空いた上位の 3 ビットには、1 が格納されます。

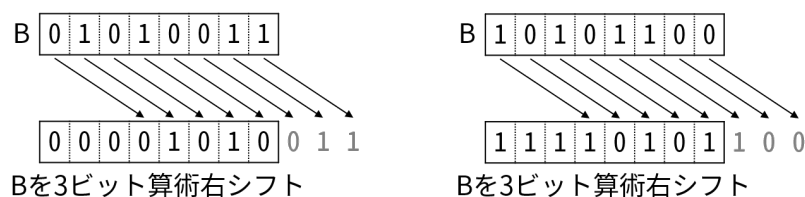


図 3.8 算術右シフトの様子

3.2 ビットを対象とする演算の演算子

ビットを対象とする演算を行う演算子を、表 3.6 に示します。

ビット単位の排他的論理和の演算子は $\wedge$ (サーカムフレックス, ハット)、ビットの反転の演算子は $\sim$ (チルダ) です。被演算子である a と b は、整数を表す型 (文字型や整数型) である必要があります。負でない数に対する右シフトは論理シフトで、負の数に対する右シフトが論理シフトか算術

表 3.6 ビットを対象とする演算の演算子

演算子	役割
<code>a & b</code>	a と b のビット単位の論理積
<code>a b</code>	a と b のビット単位の論理和
<code>a ^ b</code>	a と b のビット単位の排他的論理和
<code>~a</code>	a のビットの反転 (1 の補数)
<code>a << x</code>	a を x ビット左シフト
<code>a >> x</code>	a を x ビット右シフト

シフトかは処理系に依存します*¹。

`int` 型の変数 `a` と `b` に対してこれらの演算子を使った例を、ソースコード 3.1 に示します。

ソースコード 3.1 ビットを対象とする演算の例 (ex114.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 12, b = 10;
6
7      printf("a      = %d\n", a);
8      printf("b      = %d\n", b);
9      printf("a & b  = %d\n", a & b);
10     printf("a | b  = %d\n", a | b);
11     printf("a ^ b  = %d\n", a ^ b);
12     printf("~a     = %d\n", ~a);
13     printf("a << 3 = %d\n", a << 3);
14     printf("a >> 2 = %d\n", a >> 2);
15
16     return 0;
17 }
```

*¹ `unsigned int` 型などの符号なし整数は、MSB(最上位ビット) が 1 でも負の数ではないので、論理シフトになります。

実行結果	
a	= 12
b	= 10
a & b	= 8
a b	= 14
a ^ b	= 6
~a	= -13
a << 3	= 96
a >> 2	= 3

int 型が 32 ビットで、負の整数が 2 の補数表現である場合の、各式の値のビット列と、それを十進数として読んだときの値を、表 3.7 に示します。

表 3.7 ソースコード 3.1 の各式の値

式	ビット列	十進数
a	00000000 00000000 00000000 00001100	12
b	00000000 00000000 00000000 00001010	10
a & b	00000000 00000000 00000000 00001000	8
a b	00000000 00000000 00000000 00001110	14
a ^ b	00000000 00000000 00000000 00000110	6
~a	11111111 11111111 11111111 11110011	-13
a << 3	00000000 00000000 00000000 01100000	96
a >> 2	00000000 00000000 00000000 00000011	3

~a の値は 11111111 11111111 11111111 11110011 です。これを 2 の補数表現の符号付き整数として読むと、その値は十進数で -13 です。

十進数 n を x 桁左シフトすると、 $n \times 10^x$ になり、 x 桁右シフトすると、 $n \times 10^{-x}$ の小数部を切り捨てた値になります (図 3.9)。これと同様に、非負整数 a を x ビット左シフトすることは、 a を 2^x 倍することに相当し、 x ビット右シフトすることは、 a を 2^{-x} 倍し、小数部を切り捨てることに相当します。 $a \ll 3$ では、12 を 3 ビット左シフトしているので、結果は $12 \times 2^3 = 96$ です。 $a \gg 2$ では、12 を 2 ビット右シフトしているので、結果は (12×2^{-2}) の整数部 = 3 です。

~a、 $a \ll x$ 、 $a \gg x$ は、 a の値を変更するものではありません。式を評価すると、 a に対して、それぞれビットを反転、 x ビット左シフト、 x ビット右シフトした結果が得られます。

二項演算子である &、|、^、<<、>> にはそれぞれ複合代入演算子 &=、|=、^=、<<=、>>= が用意されています。

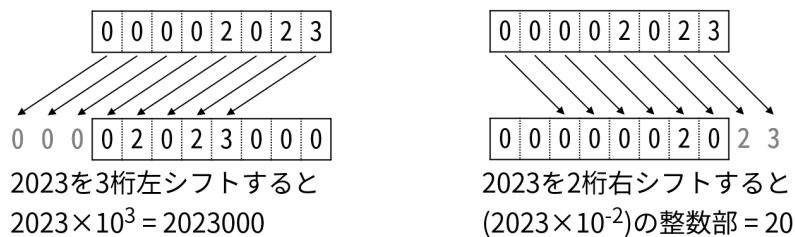


図 3.9 十進数のシフト

3.3 ビットを対象とする演算の利用

整数 1 の内部表現は 0000...001 です。整数 1 を左シフトすることによって、特定のビットが 1 で、それ以外のビットが 0 の値を作り出せます。これと演算子 & を使って、任意のビットの値を調べるプログラムを、ソースコード 3.2 に示します。

ソースコード 3.2 int 型の変数における任意のビットの値を調べるプログラム (ex115.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int num, x;
6
7      printf("num > ");
8      scanf("%d", &num);
9      printf("位置 > ");
10     scanf("%d", &x);
11
12     printf("numの下位から%dビット目の値は %c です。 \n", x, num & 1 << (x - 1)
13         ? '1' : '0');
14
15     return 0;
16 }
```

実行例 1

```

num > 13 ↵
位置 > 2 ↵
```

num の下位から 2 ビット目の値は 0 です。

実行例 2

num > 13 ↵

位置 > 3 ↵

num の下位から 3 ビット目の値は 1 です。

このプログラムにおける“位置”とはビットの位置で、最下位のビットの位置を 1、その 1 つ上位のビットの位置を 2、…、とします。“num > ”で任意の整数を、“位置 > ”で 1 以上の整数を入力すると、変数 num における指定した位置のビットの値を出力します。

12 行目の printf() 関数の第 3 引数 `num & 1 << (x - 1) ? '1' : '0'` に注目してください。

まずは、`1 << (x - 1)` について考えます。入力した位置、つまり x の値が 1 のとき、`1 << (x - 1)` は、1 を 0 ビット左シフトした値です。これはシフトをしないのと同じで、最下位のビットが 1 で、それ以外のビットが 0 である値になります。入力した位置が 2 のとき、`1 << (x - 1)` は、1 を 1 ビット左シフトした値です。下位から 2 ビット目が 1 で、それ以外のビットが 0 である値になります。同様に考えると、`1 << (x - 1)` は、下位から x ビット目の値が 1 で、それ以外のビットの値が 0 である値になります*2。

次は、`num & 1 << (x - 1)` を考えます。ビット b_1 とビット b_2 の論理積は、 b_2 の値が 1 なら b_1 の値と同じで、 b_2 の値が 0 なら 0 です (表 3.8)。よって、`num & 1 << (x - 1)` の値は、下位から x ビット目が num の下位から x ビット目と同じで、それ以外のビットが 0 になります (図 3.10)。`num & 1 << (x - 1)` の結果のビット列を整数として読むと、num の下位から x ビット目の値が 1 なら 0 以外、0 なら 0 です。

表 3.8 ビット b_1 とビット b_2 の論理積

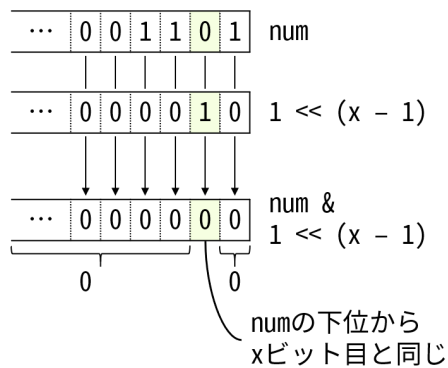
b_1	b_2		論理積	
1	1	$\Rightarrow$	1	b_2 の値が 1 なら b_1 の値と同じ
0	1	$\Rightarrow$	0	
1	0	$\Rightarrow$	0	b_2 の値が 0 なら b_1 の値によらず 0
0	0	$\Rightarrow$	0	

最終的に、`num & 1 << (x - 1) ? '1' : '0'` を評価すると、num の下位から x ビット目が 1 なら文字 '1'、0 なら文字 '0' になります。

num が 13、x が 3 の場合、`num & 1 << (x - 1)` の値は 1 ではなく、ビット列で 0000…000100、十進数で 4 になることに注意してください。

*2 二項の - は << よりも優先順位が高いため、() は必要ありません。

numが13、xが2の場合



numが13、xが3の場合

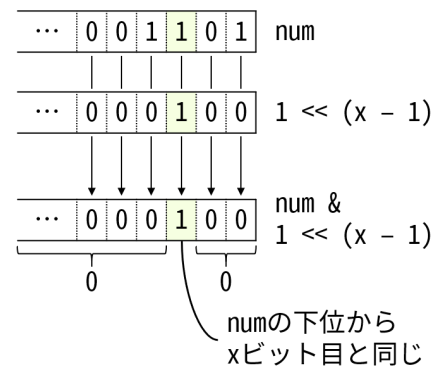


図 3.10 $\text{num} \& 1 \ll (x - 1)$ の値

評価した結果が 0 以外か 0 になるのではなく、1 か 0 になる式を作れば、条件演算子 `?:` を使わずにそのまま出力できます。そのような工夫を行ったプログラムを、ソースコード 3.3 に示します。

ソースコード 3.3 ソースコード 3.2 の改良版 (ex116.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int num, x;
6
7      printf("num > ");
8      scanf("%d", &num);
9      printf("位置 > ");
10     scanf("%d", &x);
11
12     printf("numの下位から%dビット目の値は %d です。\\n", x, num >> (x - 1) & 1)
13     ;
14     return 0;
15 }
```

このソースコードでは、`x` を左シフトするのではなく、`num` を右シフトしています。`num` が負の場合、負の数を右シフトすることになりますが、`num` に元々あったビットの値しか見ていないので、論理シフトか算術シフトかを気にする必要はありません。`printf()` 関数の変換指定として、

%c ではなく %d を使っている点も注意してください。

練習課題 3 (ex117.c)

キーボードから入力した整数を `int` 型の変数 `num` に格納し、`num` のビット列を表示するプログラムを作ってください。 `int` 型が 32 ビットであることを前提にしてかまいません。途中まで作ったプログラムを、授業のサイトに `ex117.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。

ex117.c では、int 型のビット数を表すマクロ INT_BIT を定義しています。ほとんどの処理系で int 型は 32 ビットですが、そうでない場合はこのマクロの値を書き換えてください。

実行例 2 では、負の整数が 2 の補数表現であることが確認できます。

実行例 1

```
num > 22 ↩
```

num のビット列は 0000000000000000000000000000000010110 です。

实行例 2

```
num > -22 ↩
```

`num` のビット列は 11111111111111111111111101010 です。

ソースコード 1.6 では、ループ本体を 1 回でも実行したかを表すフラグとして、int 型の変数 (`is_answered`) を用いました。フラグは 2 通りの値を表すことができればよいので、1 ビットで実現できます。ビットを対象とする演算の応用として、変数のビットをフラグとして使うプログラムを作ってみましょう。そのためには、次のような操作が必要です。

- 特定のビットの値を調べる。
- 特定のビットの値を 1 にする。
- 特定のビットの値を 0 にする。
- 特定のビットの値を反転する。

特定のビットの値を調べる

特定のビットの値を調べる方法はすでに学びました。

特定のビットの値を 1 にする

ビット列 A とビット列 B を用いて、ビット列 A の特定のビットの値を 1 にする方法を考えます。b₁ と b₂ の片方が 0 でもう片方が 1 とします。表 3.9 のような結果になる b₁、b₂ の組と演算が見つければ、ビット単位の演算を用いてビット列 A の特定のビットの値だけを 1 にしたビット列を生成できます。

表 3.9 特定のビットの値だけを 1 にする演算

	A のビット	B のビット		演算結果	
特定のビット	1	b ₁	⇒	1	A のビットが 1 になる
	0	b ₁	⇒	1	
それ以外のビット	1	b ₂	⇒	1	A のビットは変わらない
	0	b ₂	⇒	0	

b₁ が 1、b₂ が 0、演算が論理和のとき、表 3.9 の演算結果を満たします (表 3.10)。

表 3.10 論理和を用いて特定のビットの値だけを 1 にする

	A のビット	B のビット		論理和	
特定のビット	1	1	⇒	1	A のビットが 1 になる
	0	1	⇒	1	
それ以外のビット	1	0	⇒	1	A のビットは変わらない
	0	0	⇒	0	

つまり、A の下位から x ビット目を 1 にするには、B の下位から x ビット目を 1、それ以外を 0 にして A と B のビット単位の論理和を求め (図 3.11)、それを A に代入します。

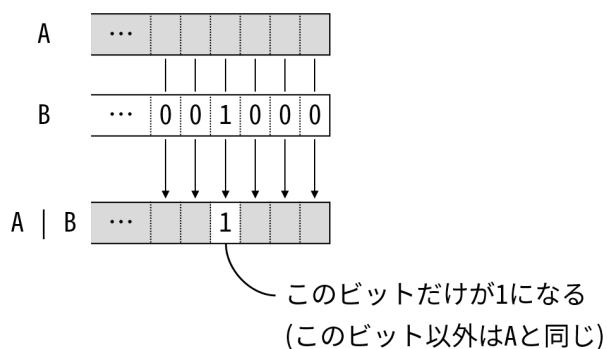


図 3.11 特定のビットの値を 1 にする操作

特定のビットの値を 0 にする

特定のビットの値を 0 にするにはどうすればよいでしょうか。表 3.11 のような結果になる b_1 、 b_2 の組と演算を考えてみてください。

表 3.11 特定のビットの値だけを 0 にする演算

	A のビット	B のビット		演算結果	
特定のビット	1	b_1	$\Rightarrow$	0	A のビットが 0 になる
	0	b_1	$\Rightarrow$	0	
それ以外のビット	1	b_2	$\Rightarrow$	1	A のビットは変わらない
	0	b_2	$\Rightarrow$	0	

b_1 が 0、 b_2 が 1、演算が論理積のとき、表 3.11 の演算結果を満たします (表 3.12)。

表 3.12 論理積を用いて特定のビットの値だけを 0 にする

	A のビット	B のビット		論理積	
特定のビット	1	0	$\Rightarrow$	0	A のビットが 0 になる
	0	0	$\Rightarrow$	0	
それ以外のビット	1	1	$\Rightarrow$	1	A のビットは変わらない
	0	1	$\Rightarrow$	0	

A の下位から x ビット目を 0 にするには、B の下位から x ビット目を 0、それ以外を 1 にして A と B のビット単位の論理積を求め (図 3.12)、それを A に代入します。

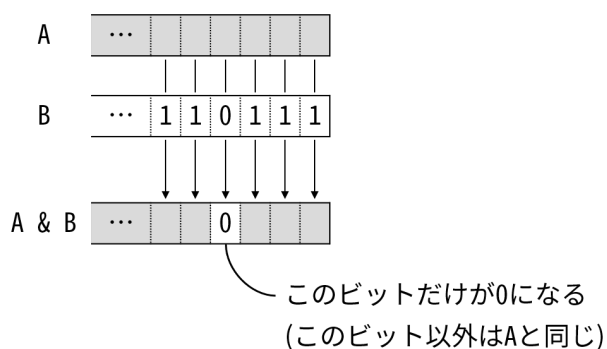


図 3.12 特定のビットの値を 0 にする操作

特定のビットの値を反転する

最後に、特定のビットの値を反転する方法を考えます。ビットの値を調べて 1 であれば 0 に、0 であれば 1 にする方法でも実現できますが、もっと簡単な方法があります。表 3.13 のような結果

になる b_1 、 b_2 の組と演算を考えてみてください。

表 3.13 特定のビットの値を反転する演算

	A のビット	B のビット	演算結果	
特定のビット	1	b_1	$\Rightarrow$ 0	A のビットが反転する
	0	b_1	$\Rightarrow$ 1	
それ以外のビット	1	b_2	$\Rightarrow$ 1	A のビットは変わらない
	0	b_2	$\Rightarrow$ 0	

正解は、 b_1 が 1、 b_2 が 0、演算が排他的論理和です (表 3.14)。

表 3.14 排他的論理和を用いて特定のビットの値を反転する

	A のビット	B のビット	排他的論理和	
特定のビット	1	1	$\Rightarrow$ 0	A のビットが反転する
	0	1	$\Rightarrow$ 1	
それ以外のビット	1	0	$\Rightarrow$ 1	A のビットは変わらない
	0	0	$\Rightarrow$ 0	

A の下位から x ビット目を反転するには、B の下位から x ビット目を 1、それ以外を 0 にして A と B のビット単位の排他的論理和を求め (図 3.13)、それを A に代入します。

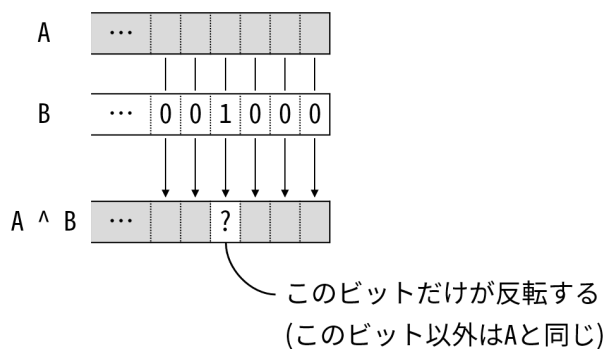


図 3.13 特定のビットの値を反転する操作

ここまでで、特定のビットの値を調べる、1 にする、0 にする、反転する、の 4 つの操作ができるようになりました。これらを使って、変数のビットをフラグとして使うプログラムを作ります。途中まで作ったプログラム (ex118.c) の前半部分を、ソースコード 3.4 に示します。

ソースコード 3.4 ex118.c の前半部分

```

1  #include <stdio.h>
2
3  int bf = 0; /* すべてのフラグがOFF */
4
5  /* フラグを調べる */
6  int is_set(int);
7
8  /* フラグをONにする */
9  void set(int);
10
11 /* フラグをOFFにする */
12 void clear(int);
13
14 /* フラグを反転する */
15 void toggle(int);
16

```

このプログラムでは、int 型のグローバル変数 bf の下位 4 ビットをフラグとして使います。図 3.14 のように、最下位ビットをフラグ [1]、下位から 2 ビット目をフラグ [2]、3 ビット目をフラグ [3]、4 ビット目をフラグ [4] とします。

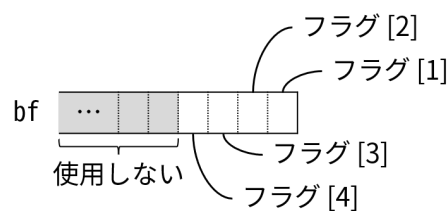


図 3.14 変数のビットをフラグとして使用

図 3.15 のように、ビットの値が 1 ならフラグが ON、0 ならフラグが OFF とします。

bf	...	0	0	0	0
					すべてのフラグがOFF
bf	...	0	0	1	0
					フラグ [2] がON
bf	...	1	0	1	0
					フラグ [2] と [4] がON

図 3.15 ビットとフラグ

ソースコード 3.4 には、4つの関数 `is_set()`、`set()`、`clear()`、`toggle()` のプロトタイプ宣言が書かれています。それぞれの関数の想定動作は次のとおりです。

<code>is_set()</code>	引数でフラグの番号を受け取り、そのフラグが ON なら 0 以外を、OFF なら 0 を返す。
<code>set()</code>	引数でフラグの番号を受け取り、そのフラグを ON にする。
<code>clear()</code>	引数でフラグの番号を受け取り、そのフラグを OFF にする。
<code>toggle()</code>	引数でフラグの番号を受け取り、そのフラグを反転する。

次に、`ex118.c` の後半部分を、ソースコード 3.5 に示します^{*3}。

ソースコード 3.5 `ex118.c` の後半部分

```
17 int main(void)
18 {
19     while (1) {
20         int fn, i;
21         char op;
22
23         /* 現在のフラグの状態を表示する */
24         printf("フラグ: [1] [2] [3] [4]\n状態: ");
25         for (i = 1; i <= 4; ++i)
26         {
27             printf(" %s", is_set(i) ? "ON " : "OFF");
28         }
29         printf("\n\n");
30
31         /* 操作対象フラグの番号を入力させる */
32         printf("対象フラグ (1-4, その他: 終了) > ");
33         scanf("%d%c", &fn);
34         if (fn < 1 || fn > 4)
35         {
36             /* 1~4以外が入力されたら反復を終了する */
37             break;
38         }
39
40         /* フラグに対する操作を入力させる */
```

^{*3} 27 行目と 47 行目では、`printf()` 関数の第 1 引数で変換指定 `%s` を使っています。`%s` は第 10 章で説明します。

```

41     printf("操作 ('p': 確認, 's': ON, 'c': OFF, 't': 反転) > ");
42     scanf("%c", &op);
43
44     switch (op)
45     {
46         case 'p':
47             printf("\nフラグ [%d] は%sです。 \n\n", fn, is_set(fn) ? "ON"
: "OFF");
48             break;
49         case 's':
50             set(fn);
51             printf("\nフラグ [%d] をONにしました。 \n\n", fn);
52             break;
53         case 'c':
54             clear(fn);
55             printf("\nフラグ [%d] をOFFにしました。 \n\n", fn);
56             break;
57         case 't':
58             toggle(fn);
59             printf("\nフラグ [%d] を反転しました。 \n\n", fn);
60             break;
61     }
62 }
63
64 return 0;
65 }

```

main() 関数は、while 文を使った無限ループになっており、次の動作を繰り返します。

1. 現在のフラグの状態を表示する。(23～29 行目)
2. キーボードから操作対象のフラグの番号を入力させ、変数 `fn` に格納する。入力されたフラグの番号が 1～4 でなければ、無限ループを終了する。(31～38 行目)
3. キーボードからフラグに対する操作を入力させ、変数 `op` に格納する。(40～42 行目)
4. `fn` が表すフラグに対して、`op` が表す操作を行う。(44～61 行目)

フラグの操作は、確認する (ON か OFF かを表示する)、ON にする、OFF にする、反転する、の 4 種類です。文字 `p`、`s`、`c`、`t` として入力された操作指示に応じて、44 行目から始まる `switch` 文でフラグの操作を行っています。

33 行目の `scanf()` 関数の第 1 引数は、"`%d`" ではなく "`%d%c`" となっています。詳細は省略しますが、42 行目の `scanf()` 関数で文字を読み取るために、この変換指定が必要とお考えください*4。また、フラグの番号として入力できるのは、整数だけです。数字以外を入力してしまった場合は、プログラムを強制終了させてください。

このプログラムの想定実行例は次のとおりです。

想定実行例

フラグ: [1] [2] [3] [4]

状態: OFF OFF OFF OFF

対象フラグ (1-4, その他: 終了) > 2 ↵

操作 ('p': 確認, 's': ON, 'c': OFF, 't': 反転) > s ↵

フラグ [2] を ON にしました。

フラグ: [1] [2] [3] [4]

状態: OFF ON OFF OFF

対象フラグ (1-4, その他: 終了) > 2 ↵

操作 ('p': 確認, 's': ON, 'c': OFF, 't': 反転) > s ↵

フラグ [2] を ON にしました。

フラグ: [1] [2] [3] [4]

状態: OFF ON OFF OFF

対象フラグ (1-4, その他: 終了) > 4 ↵

操作 ('p': 確認, 's': ON, 'c': OFF, 't': 反転) > t ↵

フラグ [4] を反転しました。

フラグ: [1] [2] [3] [4]

状態: OFF ON OFF ON

*4 エンターキーで入力される改行も文字として認識されます。33 行目の `scanf()` 関数で `%d` を使った場合、42 行目の `scanf()` 関数では、33 行目で読み残していた改行を読み取ってしまいます。これまでには数値を連続して読み取るプログラムもありましたが、数値 (`%d`) の場合は改行を読み飛ばしますので、うまくいっていました。

対象フラグ (1-4, その他: 終了) > 2 ↩

操作 ('p': 確認, 's': ON, 'c': OFF, 't': 反転) > c ↩

フラグ [2] を OFF にしました。

フラグ: [1] [2] [3] [4]

状態: OFF OFF OFF ON

対象フラグ (1-4, その他: 終了) > 4 ↩

操作 ('p': 確認, 's': ON, 'c': OFF, 't': 反転) > p ↩

フラグ [4] は ON です。

フラグ: [1] [2] [3] [4]

状態: OFF OFF OFF ON

対象フラグ (1-4, その他: 終了) > 0 ↩

この例では、フラグ [2] を ON にする、フラグ [2] を ON にする*⁵、フラグ [4] を反転する、フラグ [2] を OFF にする、フラグ [4] の状態を確認する、の 5 つの操作を行っています。

練習課題 4 (ex118.c)

ex118.c に対して、`is_set()`、`set()`、`clear()`、`toggle()` の各関数を実装し、プログラムを完成させてください。

*⁵ ON の状態から ON にしても変わらないことを確認するため。

第 4 章

乱数

さいころを振って出る目は、1 以上 6 以下です。振ってみるまで、どの目が出るかはわかりません。**乱数**とはこのような性質を持った数です。また、乱数の数列を**乱数列**といいます。さいころを何度も振って、出た目を並べると、乱数列になります。

以降で作成するプログラムには、乱数を使うものがあります。本章では、C 言語における乱数の使い方を説明します。

4.1 乱数の生成

表 4.1 に示すライブラリ関数 `rand()` は、0 以上 `RAND_MAX` 以下の乱数を返します。

表 4.1 `rand()` 関数

書式	<code>rand()</code>
引数	なし
戻り値	0 以上 <code>RAND_MAX</code> 以下の乱数 (<code>int</code> 型)
必要ヘッダ	<code><stdlib.h></code>

`RAND_MAX` は、ヘッダ `<stdlib.h>` で定義されるマクロで、その値は処理系によって異なります。次のプログラムで確認してみましょう。

ソースコード 4.1 `RAND_MAX` の値を確認するプログラム (`ex119.c`)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      printf("%d\n", RAND_MAX);
7  }
```

```

8     return 0;
9 }

```

Windows 11 上の MinGW-w64(gcc version 13.2.0) での実行結果

32767

FreeBSD 13.1 (amd64) 上の Clang 13.0.0 での実行結果

2147483647

MinGW-w64 では、RAND_MAX の値は 32767 でしたので、rand() 関数は 0 以上 32767 以下の乱数を返します。

rand() を呼び出すたびに、乱数列から値が 1 つずつ返されます。rand() を 5 回呼び出して、返された値を表示するプログラムを、ソースコード 4.2 に示します。

ソースコード 4.2 乱数を 5 回表示するプログラム (ex120.c)

```

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

```

実行結果例

41
18467
6334
26500

何回か実行してみてください。毎回同じ値が表示されるはずですが、`rand()` が使う乱数列は計算によって求めており、初期状態が同じであれば同じ乱数列になります。つまり、乱数のように見えるだけで、計算方法を知っていれば、次に出る数がわかります。このような乱数、乱数列は、真の乱数、乱数列と区別して、**擬似乱数**、**擬似乱数列**と呼ばれます。

`rand()` の初期状態を変える関数が、表 4.2 に示す `srand()` です。

表 4.2 `srand()` 関数

書式	<code>srand(seed)</code>
引数	非負整数 <i>seed</i> (<code>unsigned int</code> 型)
戻り値	なし
必要ヘッダ	<code><stdlib.h></code>

`srand()` 関数の引数として、擬似乱数列を生成するための^{たね}種を指定します。種が同じであれば、同じ擬似乱数列が生成されます。`srand()` 関数の呼び出しを省略した場合は、種として 1 が指定されたと見なされます。`srand()` 関数は、`rand()` 関数を呼び出す前に、一度だけ呼び出します*¹。

プログラムの実行ごとに異なる擬似乱数列を用いるには、種を変える必要があります。種を乱数で与えたいところですが、`srand()` 関数を呼び出す前に `rand()` 関数を用いても、実行ごとに値を変えることができません。そこで、現在時刻をもとにした数値を種として使うことが一般的です。そうすれば、実行ごとに種が異なり、全く違う擬似乱数列が生成できます。“現在時刻をもとにした数値”として、具体的には表 4.3 に示す `time()` 関数の戻り値を使います。

表 4.3 `time()` 関数

書式	<code>time(NULL)</code>
引数	この授業では“NULL”を指定
戻り値	現在時刻を表す値 (<code>time_t</code> 型)
必要ヘッダ	<code><time.h></code>

`time()` 関数の引数には“NULL”と書いてください。“NULL”については、第 10 章で簡単に説明します。戻り値の型である `time_t` 型は、数値を表す既存の型の別名として、`<time.h>` で `typedef` 宣言されています*²。“現在時刻を表す値”が何かは規格では決められていませんが、多くの処理系では、協定世界時*³で 1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数*⁴を返します。たとえば、日

*¹ `srand()` 関数を呼び出すたびに、`rand()` 関数で得られる値が、擬似乱数列の先頭の値に戻ってしまいます。

*² 第 2 章で学んだ `size_t` 型も、`typedef` 宣言で作られた型でした。

*³ UTC。日本標準時 (JST) より 9 時間遅れています。

*⁴ この値は Unix 時間 (Unix time) やエポック秒 (Epoch time) と呼ばれます。

本標準時で 2023 年 4 月 1 日 0 時 0 分 0 秒であれば、1680274800 になります。これを `unsigned int` 型にキャストして、`srand()` 関数の引数にします。

ソースコード 4.2 を改変し、プログラムの実行ごとに異なる値が表示されるようにしたものを、ソースコード 4.3 に示します。

ソースコード 4.3 実行ごとに異なる乱数を 5 回表示するプログラム (ex121.c)

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

実行結果例

```
23033
9093
5154
29881
11549
```

プログラムで乱数として 0 以上 `RAND_MAX` 以下の整数を使いたいということはほとんどなく、“1 以上 10 以下の整数”や“0 以上 5 以下の実数”など、`rand()` 関数の戻り値とは異なる範囲や型の値が必要になります。その場合、`rand()` 関数の戻り値を加工して、必要な範囲、型の乱数を作り出します。`int` 型の変数 `r` に `rand()` 関数の戻り値が代入されている場合、次の式で 1 以上 10 以下の整数を作り出せます。

$$1 + r \% 10$$

$r \% 10$ は r を 10 で割った余りで、取りうる値は 0 以上 9 以下の整数です。1 にこの値を足すことによって、1 以上 10 以下の整数を作っています*⁵。0 以上 5 以下の実数は、次の式で作ります。

$$r * 5.0 / \text{RAND_MAX}$$

$r * 5.0$ の取りうる値は、0 以上 $\text{RAND_MAX} \times 5$ 以下の実数です*⁶。これを RAND_MAX で割ることによって、0 以上 5 以下の実数を作っています。

4.2 乱数の利用

乱数は、ゲームやシミュレーションのプログラムで利用されます。ここでは、乱数を使うプログラムを 2 つ作ります。

簡易カードゲーム

10 枚のカードがあり、それぞれ 1~10 の整数が書かれています。プレイヤーは、数が見えない状態で 1 枚を引き、出た数を記録した後カードを戻します。出た数の合計が 23 を超えず、できるだけ 23 に近づくまでカードを引くことを繰り返します。出た数の合計が 23 を超えなかった場合はその数が得点になり、超えた場合は 0 点になります。

このゲームを実装してみましょう。想定実行例は次のとおりです。

想定実行例 1

現在の得点: 0

カードを引きますか (はい: 1, いいえ: 0) > 1 ↩

カードは 4 でした

現在の得点: 4

カードを引きますか (はい: 1, いいえ: 0) > 1 ↩

カードは 1 でした

現在の得点: 5

カードを引きますか (はい: 1, いいえ: 0) > 1 ↩

*⁵ RAND_MAX が 32767 の場合、32768、32769 がない分、厳密には 9 と 10 が他の数値に比べて出にくくなりますが、無視できる程度です。

*⁶ int 型の値 (r) と double 型の値 (5.0) の算術演算なので、結果は double 型になります。

カードは 10 でした

現在の得点: 15

カードを引きますか (はい: 1, いいえ: 0) > 1 ↩

カードは 6 でした

現在の得点: 21

カードを引きますか (はい: 1, いいえ: 0) > 0 ↩

あなたの得点は 21 点です

想定実行例 2

現在の得点: 0

カードを引きますか (はい: 1, いいえ: 0) > 1 ↩

カードは 8 でした

現在の得点: 8

カードを引きますか (はい: 1, いいえ: 0) > 1 ↩

カードは 10 でした

現在の得点: 18

カードを引きますか (はい: 1, いいえ: 0) > 1 ↩

カードは 10 でした

23 点を超えました

あなたの得点は 0 点です

途中まで作ったプログラム (ex122.c) を、ソースコード 4.4 に示します。

ソースコード 4.4 簡易カードゲーム (ex122.c)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main(void)
6 {
7     int point = 0;
```

```

8      int cont;
9
10     srand((unsigned int)time(NULL));
11
12     do
13     {
14         printf("現在の得点: %d\n", point);
15         printf("カードを引きますか (はい: 1, いいえ: 0) > ");
16         scanf("%d", &cont);
17         if (cont)
18         {
19
20
21             /* ここに処理を記述 */
22
23
24         }
25         printf("\n");
26     } while (cont);
27
28     printf("あなたの得点は %d 点です\n", point);
29
30     return 0;
31 }

```

7行目で宣言している変数 `point` には、得点を格納します。プレイヤーは、カードを引くなら 1 を、引かないなら 0 をキーボードから入力します。入力された値を整数として取り込み、8 行目で宣言している変数 `cont` に格納します。`cont` を真理値として用い、12 行目から始まる `do` 文の反復の判定に使用します。

練習課題 5 (ex122.c)

ex122.c の “/* ここに処理を記述 */” と書いた部分に必要な処理を記述して、プログラムを完成させてください。

くじ引きのシミュレーション

1 ポイントを使って参加すると、5% の確率で 10 ポイントが、15% の確率で 3 ポイントが当たるくじがあります。このくじに何度も参加したときに得られるポイントをシミュレーションしてみましょう。

想定実行例は次のとおりです。

想定実行例 1

くじを引く回数 > 100 ↗

100 回くじを引き、10 ポイントに 8 回、3 ポイントに 12 回当たりました。

使ったポイントは 100、得たポイントは 116 です。

想定実行例 2

くじを引く回数 > 5000 ↗

5000 回くじを引き、10 ポイントに 256 回、3 ポイントに 701 回当たりました。

使ったポイントは 5000、得たポイントは 4663 です。

途中まで作ったプログラム (ex123.c) を、ソースコード 4.5 に示します。

ソースコード 4.5 くじ引きのシミュレーション (ex123.c)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main(void)
6  {
7      int trials; /* くじを引く回数 */
8      int w1 = 0; /* 10ポイントに当たった回数 */
9      int w2 = 0; /* 3ポイントに当たった回数 */
10     int i;
11
12     srand((unsigned int)time(NULL));
13
14     printf("くじを引く回数 > ");
15     scanf("%d", &trials);
16
```

```

17
18     /* ここに処理を記述 */
19
20
21     printf("%d回くじを引き、10ポイントに%d回、3ポイントに%d回当たりました。\\n",
22           trials, w1, w2);
23     printf("使ったポイントは%d、得たポイントは%dです。\\n", trials, w1 * 10 + w2
24           * 3);
25
26     return 0;
27 }

```

くじ引きに参加する回数は、キーボードから入力し、変数 `trials` に格納します。くじ引きのシミュレーションを `trials` 回行い、10 ポイントに当たった回数を変数 `w1` に、3 ポイントに当たった回数を変数 `w2` に格納します。そして、21 行目と 22 行目で、シミュレーションの結果を表示します。くじ引きを繰り返している途中では、10 ポイントと 3 ポイントに当たった回数だけを記録しておき、最終的に得たポイントは 22 行目の `printf()` 関数の第 3 引数 (`w1 * 10 + w2 * 3`) で計算しています。

5% や 15% の事象が起こった (10 ポイントや 3 ポイントに当たった) かを、乱数を使って決めるには、次のような方法が考えられます。

- 0 以上 1 以下の実数の乱数を発生させて、0.05 未満なら 5% の事象が、0.05 以上 0.2 未満なら 15% の事象が起こったとする。
- 0 以上 19 以下の整数の乱数を発生させて、0 なら 5% の事象が、1～3 なら 15% の事象が起こったとする。

練習課題 6 (ex123.c)

ex123.c の “/* ここに処理を記述 */” と書いた部分に必要な処理を記述して、プログラムを完成させてください。

第 5 章

配列

5.1 配列の導入

5 種類の商品の価格を入力し、最高額の商品の番号と価格を表示するプログラムを、ソースコード 5.1 に示します。

ソースコード 5.1 最高額の商品の番号と価格を表示するプログラム (ex124.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int price1, price2, price3, price4, price5;
6      int max_id;    /* 最高額の商品の番号 */
7      int max_price; /* 最高額の商品の価格 */
8
9      printf("商品 1 の価格 > ");
10     scanf("%d", &price1);
11     printf("商品 2 の価格 > ");
12     scanf("%d", &price2);
13     printf("商品 3 の価格 > ");
14     scanf("%d", &price3);
15     printf("商品 4 の価格 > ");
16     scanf("%d", &price4);
17     printf("商品 5 の価格 > ");
18     scanf("%d", &price5);
19
20     max_id = 1;
```

```

21     max_price = price1;
22     if (price2 > max_price)
23     {
24         max_id = 2;
25         max_price = price2;
26     }
27     if (price3 > max_price)
28     {
29         max_id = 3;
30         max_price = price3;
31     }
32     if (price4 > max_price)
33     {
34         max_id = 4;
35         max_price = price4;
36     }
37     if (price5 > max_price)
38     {
39         max_id = 5;
40         max_price = price5;
41     }
42
43     printf("最高額の商品の番号は %d、価格は %d です。\\n", max_id, max_price);
44
45     return 0;
46 }

```

実行例

商品 1 の価格 > 150 ↩

商品 2 の価格 > 120 ↩

商品 3 の価格 > 210 ↩

商品 4 の価格 > 80 ↩

商品 5 の価格 > 150 ↩

最高額の商品の番号は 3、価格は 210 です。

変数 price1～price5 に、それぞれ番号 1～5 の商品の価格を格納します。次に、ひとまず商品

1 を最高額と考えて、最高額の商品の番号を格納する変数 `max_id` に 1 を、最高額の商品の価格を格納する変数 `max_price` に `price1` の値を代入します。そして、商品 2~5 の価格が `max_price` を超えていないか順に確認し、超えていれば `max_id` と `max_price` を更新します。

このプログラムは正しく動作します*¹が、次のような欠点があります。

- 9 行目から 18 行目までと、22 行目から 41 行目までは同じような処理が何度も書かれています。商品の種類がこれ以上増えるとソースコードが読みづらくなり、間違いも混入しやすくなります。
- 商品の番号 (`max_id`) から直接価格を得ることができないため、`max_id` と `max_price` の 2 つの変数を使う必要があります。

これらの欠点は、**配列**を使うと解決できます。このプログラムは後で改善することにして、まずは配列について説明します。

配列とは、**同じ型の変数**を並べて、それぞれに値を格納できるようにしたものです。配列に含まれる 1 つ 1 つの変数を、**要素**といいます。配列自体も変数です。配列を使うためには宣言が必要で、その書式は

要素の型名 配列の変数名 [要素数];

です。要素の型が `int` 型で、要素数が 5 の配列 `ia` の宣言は、

`int ia[5];`

です。要素の型が `double` 型で、要素数が 3 の配列 `da` の宣言は、

`double da[3];`

です。この授業では、要素数 (上の例では、5 と 3) として定数を使うことにします*²。

配列の各要素には、^{添字}**添字**と呼ばれる 0 から始まる番号が与えられています。

配列の変数名 [添字]

という書式で、各要素を表します。添字を囲む `[]` は、表 1.1 に示した優先順位 1 の演算子です*³。たとえば、`int ia[5];` と宣言すると、`ia[0]`、`ia[1]`、`ia[2]`、`ia[3]`、`ia[4]` の 5 つの要素を、それぞれ `int` 型の変数として使えます。**`ia[5]` は使えない**ことに注意してください。

詳しくは第 8 章で学習しますが、配列は、要素が一行に並んでいます。要素数 5 の `int` 型の配列 `ia` のイメージを、図 5.1 に示します。

*¹ 最高額の商品が複数ある場合は、その中で番号が一番小さい商品が選ばれます。

*² 従来の C 言語では定数しか使えませんでした。現在の処理系は、条件を満たした変数を使える場合もあります。

*³ 配列の宣言で使った `[]` は演算子ではありません。

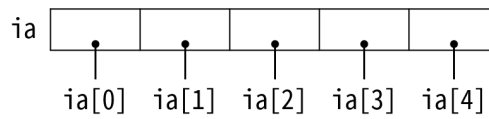


図 5.1 配列のイメージ

ソースコード 5.2 は、“C 言語基礎演習” で使った、変数の値を入れ替えるプログラムです。

ソースコード 5.2 変数の値の入れ替え (“C 言語基礎演習” の ex31.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         int tmp;
16         tmp = a;
17         a = b;
18         b = tmp;
19     }
20
21     printf("大きい方は%d、小さい方は%dです。 \n", a, b);
22
23     return 0;
24 }
```

このプログラムでは、キーボードから入力された 2 つの整数を、int 型の変数 a と b に格納します。そして、a が b より小さければ、int 型の変数 tmp を使って、a と b の値を入れ替えます。

このプログラムで使っている 3 つの int 型の変数を、要素数が 3 の配列の要素に置き換えたものが、ソースコード 5.3 です。

ソースコード 5.3 ソースコード 5.2 を配列を使うように変更 (ex125.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int n[3];
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", &n[0]);
10     printf("2つ目 > ");
11     scanf("%d", &n[1]);
12
13     if (n[0] < n[1])
14     {
15         n[2] = n[0];
16         n[0] = n[1];
17         n[1] = n[2];
18     }
19
20     printf("大きい方は%d、小さい方は%dです。 \n", n[0], n[1]);
21
22     return 0;
23 }
```

実行例

整数を 2 つ入力してください。

1 つ目 > 3 ↵

2 つ目 > 5 ↵

大きい方は 5、小さい方は 3 です。

ソースコード 5.3 では、要素数が 3 の int 型の配列 `n` を用意し、`a` の代わりに `n[0]`、`b` の代わりに `n[1]`、`tmp` の代わりに `n[2]` を使っています。

関数内で必要な変数のうち同じ型のものをまとめて配列とすれば、宣言する変数が減ってよいと感じるかもしれません。しかし、このような配列の使い方は好ましくありません。ソースコード 5.3 は、配列の要素が普通の変数のように使えることを示す例と同時に、配列の不適切な使い方を

示す例でした。ソースコードのわかりやすさの観点から、年齢を格納する変数は `age`、体重を格納する変数は `weight` など、異種の値には異なる変数名を使うべきです。配列は、10 人の体重を要素数 10 の配列に格納するなど、連続する同種の値をまとめることに適しています。

配列の添字には変数も使え、`for` 文などを使って連続して処理することができます。`for` 文を使った配列の操作の例を、ソースコード 5.4 に示します。

ソースコード 5.4 `for` 文を使った配列の操作の例 (ex126.c)

```
1  #include <math.h>
2  #include <stdio.h>
3
4  #define NUM 6
5
6  int main(void)
7  {
8      double da[NUM];
9      int i;
10
11     /* 代入 */
12     for (i = 0; i < NUM; ++i)
13     {
14         da[i] = sqrt((double)(i + 1));
15     }
16
17     /* 表示 */
18     for (i = 0; i < NUM; ++i)
19     {
20         printf("√%d = %.2f\n", i + 1, da[i]);
21     }
22
23     return 0;
24 }
```

実行結果

```
√ 1 = 1.00
√ 2 = 1.41
√ 3 = 1.73
```

```
√ 4 = 2.00
√ 5 = 2.24
√ 6 = 2.45
```

このプログラムでは、8 行目で要素数が 6 の `double` 型の配列 `da` を宣言しています。そして、12 行目から始まる `for` 文で、`da[0]~da[5]` に、それぞれ 1~6 の非負の平方根を代入しています。ループ本体の 1 回目では `da[0] = sqrt((double)1);`、2 回目では `da[1] = sqrt((double)2);` に相当する処理が行われることになります。次に、18 行目から始まる `for` 文で、`da[0]~da[5]` に格納された 1~6 の非負の平方根をわかりやすく表示しています。さて、ソースコード 5.1 の欠点としてあげた 2 点は、配列を使うと次のように解決できます。

- 変数 `price1~price5` の代わりに、要素数 5 の配列が使えます。商品の価格を入力する処理と、最高額の商品を求める処理は、反復を用いることによって簡潔に記述できます。
- 商品の番号から配列の要素が特定できます。最高額の商品を求める処理では、商品の価格を保持する必要はありません。

練習課題 7 (ex127.c)

ソースコード 5.1(ex124.c) を、次のように改変してください。

- 商品の価格を、変数 `price1~price5` ではなく、配列 `price` に格納する。
- 変数 `max_price` は使わない。

変数 `max_id` には最高額の商品の番号を格納していましたが、必要であれば使い方 (格納する値の意味) を変えてください。

5.2 配列の使い方

配列を使うために必要な知識をまとめて説明します。

これまでの変数同様、配列を宣言した直後の要素の値は不定です。初期化^{*4}をするか、値を代入してから使う必要があります。

配列を初期化するには、次のように、要素の初期値を、(コンマ) 区切りで並べ、全体を `{ }` で囲みます。

```
1 /* ia[0]~ia[4]を、1~5で初期化 */
2 int ia[5] = { 1, 2, 3, 4, 5 };
```

^{*4} 宣言と同時に変数に値 (初期値) を入れることを初期化といいます。

```

3
4  /* ca[0]を'A'、ca[1]を'B'、ca[2]を'C'で初期化 */
5  char ca[3] = { 'A', 'B', 'C' };

```

配列を初期化する場合、要素数は省略できます。要素数を省略すると、{ } の中に書いた初期値の数が要素数と見なされます。

```

1  int ia[] = { 1, 2, 3, 4, 5 }; /* iaの要素数は 5 */
2  char ca[] = { 'A', 'B', 'C' }; /* caの要素数は 3 */

```

{ 1, 2, 3, 4, 5 } のような書式で値を指定できるのは、初期化のときだけです。また、配列に配列を代入することはできません。

```

1  int a1[5] = { 1, 2, 3, 4, 5 }, a2[5];
2
3  a2 = { 5, 4, 3, 2, 1 }; /* エラー。初期化の書式での代入は不可。 */
4  a2 = a1; /* エラー。配列に配列の代入は不可。 */

```

配列を別の配列にコピーしたい場合は、次のように 1 要素ずつ代入する必要があります。

```

1  int a1[5] = { 1, 2, 3, 4, 5 }, a2[5], i;
2
3  for (i = 0; i < 5; ++i)
4  {
5      a2[i] = a1[i];
6  }

```

配列の要素数を、プログラムの途中で変えることはできません。ですから、プログラムで使う十分な要素数で宣言する必要があります。

配列の変数名に対して `sizeof` 演算子を適用すると、**配列の大きさ**が得られます。

$$\text{配列の大きさ} = \text{要素 1 つの大きさ} \times \text{要素数}$$

です。ia が要素数 5 の `int` 型の配列であるとき、`sizeof ia` は、`int` 型の大きさの 5 倍です。逆に、配列の要素数は、

$$\text{要素数} = \text{配列の大きさ} \div \text{要素 1 つの大きさ}$$

で求められます。このとき、要素 1 つの大きさとして、要素の型の大きさ (`sizeof(int)`)、または添字が 0 である要素の大きさ^{*5}(`sizeof ia[0]`) を使えます。これらを確認するプログラムを、ソースコード 5.5 に示します。

^{*5} 配列には、添字が 0 の要素が必ずあります。

ソースコード 5.5 配列の大きさを確認するプログラム (ex128.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int ia[5];
6      double da[7];
7
8      printf("int型の大きさ: %d\n\n", (int)sizeof(int));
9
10     printf("ia[0]の大きさ: %d\n", (int)sizeof ia[0]);
11     printf("iaの大きさ: %d\n", (int)sizeof ia);
12     printf("iaの要素数: %d\n\n", (int)(sizeof ia / sizeof(int)));
13
14     printf("double型の大きさ: %d\n\n", (int)sizeof(double));
15
16     printf("da[0]の大きさ: %d\n", (int)sizeof da[0]);
17     printf("daの大きさ: %d\n", (int)sizeof da);
18     printf("daの要素数: %d\n", (int)(sizeof da / sizeof da[0]));
19
20     return 0;
21 }
```

Windows 11 上の MinGW-w64(gcc version 13.2.0) での実行結果

int 型の大きさ: 4

ia[0] の大きさ: 4

ia の大きさ: 20

ia の要素数: 5

double 型の大きさ: 8

da[0] の大きさ: 8

da の大きさ: 56

da の要素数: 7

int 型の大きさが 4 なので、int 型の配列 ia の要素の大きさも 4 です。ia の要素数が 5 なので、ia の大きさは $4 \times 5 = 20$ になります。

このプログラムでは ia の要素数が 5、da の要素数が 7 であることはわかっていますが、配列の大きさから要素数が計算できるかを確認しています。ia の要素数は、sizeof ia を int 型の大きさ sizeof(int) で割ることによって、5 と求まりました。da の要素数は、sizeof da を 1 番目の要素の大きさ sizeof da[0] で割ることによって、7 と求まりました。このようにして求めた配列の要素数を使えば、定数を記述する箇所を減らすことができます。また、次のように要素数を指定せずに配列を初期化した場合は、プログラムの中に要素数を書く必要がなく、要素数を変えときの修正漏れを回避できます。

```
1  int ia[] = { 1, 2, 3, 4, 5 };
2  int len, i;
3
4  len = (int)(sizeof ia / sizeof ia[0]);
5  for (i = 0; i < len; ++i)
6  {
7      printf("%d\n", ia[i]);
8  }
```

練習課題 8 (ex129.c)

第 1 項と第 2 項が 1 で、第 n 項 ($n \geq 3$) が第 $(n - 2)$ 項と第 $(n - 1)$ 項の和である数列をフィボナッチ数列といいます。int 型の配列 fibo に、フィボナッチ数列の第 1 項から第 15 項を格納し、表示するプログラムを作ってください。fibo[0] と fibo[1] には定数 1 を代入し、fibo[2] ~ fibo[14] には反復を使って値を代入してください。fibo のすべての要素に値を代入した後、それらの値を表示します。途中まで作ったプログラムを、授業のサイトに ex129.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。

実行結果

フィボナッチ数列は 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

5.3 配列の操作

ここでは、配列の扱いに慣れるために、いくつかのプログラムを作ります。

ソースコード 5.6 は、要素数 6 の `int` 型の配列 `ia` の要素に、0 以上 9 以下の整数の乱数を代入して表示するプログラムです。

ソースコード 5.6 配列の要素に乱数を代入して表示するプログラム (`ex130.c`)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define NUM 6
6
7  int main(void)
8  {
9      int ia[NUM];
10     int i;
11
12     srand((unsigned int)time(NULL));
13
14     /* ia[0]～ia[NUM - 1]に0以上9以下の乱数を代入 */
15     for (i = 0; i < NUM; ++i)
16     {
17         ia[i] = rand() % 10;
18     }
19
20     /* iaの要素の値を表示 */
21     for (i = 0; i < NUM; ++i)
22     {
23         printf("[%d] ", i);
24     }
25     printf("\n");
26     for (i = 0; i < NUM; ++i)
27     {
28         printf(" %d ", ia[i]);
29     }
30     printf("\n");
31
32     return 0;
33 }
```

実行結果例 1

[0]	[1]	[2]	[3]	[4]	[5]
7	1	2	1	3	9

実行結果例 2

[0]	[1]	[2]	[3]	[4]	[5]
4	5	7	6	8	2

以降の 3 つの練習課題では、`ex130.c` に処理を追加します。`ex130.c` では、マクロ `NUM` で要素数を定義し、1 箇所の修正で要素数を変えられるようにしています。練習課題で作るプログラムも、要素数として 6 ではなく `NUM` を使ってください。また、新しい関数は作らずに、すべて `main()` 関数内で処理するようにしてください*⁶。

練習課題 9 (ex131.c)

`ex130.c` に処理を追加して、要素の値の最大値を表示するプログラムを作ってください。

実行結果例 1

[0]	[1]	[2]	[3]	[4]	[5]
6	4	8	6	0	8

最大値は 8 です

実行結果例 2

[0]	[1]	[2]	[3]	[4]	[5]
5	7	1	6	1	5

最大値は 7 です

*⁶ 配列を関数の引数にするには、第 8 章まで読み進める必要があります。

練習課題 10 (ex132.c)

ex130.c に処理を追加して、次のようなプログラムを作ってください。

- 要素の値の並びを逆順にする。
- 逆順にした後、要素の値を表示する。

NUM が奇数 (たとえば 7) の場合でも正しく動作するようにしてください。途中まで作ったプログラムを、授業のサイトに ex132.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。

実行結果例 (NUM が 6 の場合)

[0]	[1]	[2]	[3]	[4]	[5]
0	7	9	2	5	9

逆順にしました

[0]	[1]	[2]	[3]	[4]	[5]
9	5	2	9	7	0

実行結果例 (NUM が 7 の場合)

[0]	[1]	[2]	[3]	[4]	[5]	[6]
7	3	1	5	9	5	6

逆順にしました

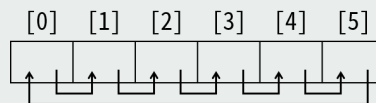
[0]	[1]	[2]	[3]	[4]	[5]	[6]
6	5	9	5	1	3	7

練習課題 11 (ex133.c)

ex130.c に処理を追加して、次のようなプログラムを作ってください。

- 末尾以外の要素の値を 1 つずつ後ろに移動する。
- 末尾の要素の値を先頭に移動する。
- 移動後に、要素の値を表示する。

各要素の値の移動先は、次のとおりです。



実行結果例 1

[0]	[1]	[2]	[3]	[4]	[5]
4	7	9	4	5	3

移動しました

[0]	[1]	[2]	[3]	[4]	[5]
3	4	7	9	4	5

実行結果例 2

[0]	[1]	[2]	[3]	[4]	[5]
5	6	5	9	0	7

移動しました

[0]	[1]	[2]	[3]	[4]	[5]
7	5	6	5	9	0

第 6 章

ポインタ

ここでは、C 言語の大きな特徴であると同時に、最大の難所とされているポインタを学習します。演算子や型名として使う記号が紛らわしく混乱することがありますが、基本的な仕組みは単純です。

6.1 メモリと変数のアドレス

コンピュータでプログラムを実行するためには、SSD^{*1}などの補助記憶装置に保存されたプログラムファイルを読み出し、プログラムの命令やプログラムに必要なデータを、**メモリ上に配置**する必要があります。**変数の領域**もメモリ上に確保されます。プログラムで変数の値を使うことは、その変数に割り当てられたメモリ領域の値を使うことと同義で、変数の値を変えることは、そのメモリ領域の値を変えることと同義です。メモリには 1 バイトごとに**アドレス** (番地) が振られています。char 型の変数 c と int 型の変数 i を用いるプログラムの実行時における変数の領域のイメージを、図 6.1 に示します。

ここでは、int 型が 4 バイトであることを前提にしています。図のように、メモリ上に、変数 c が使う 1 バイトの領域と、変数 i が使う 4 バイトの領域が確保されます^{*2}。変数 c の領域は 511 番地に、変数 i の領域は 504~507 番地に割り当てられています。先頭アドレスは、それぞれ 511 番地と 504 番地です。

変数名に対して、単項演算子 & を適用すると、その変数の領域の**先頭アドレス**が得られます (表 6.1)。

図 6.1 の例では、&c で 511 が、&i で 504 が得られます。

^{*1} ソリッドステートドライブ。不揮発性の (電源を切ってもデータが消えない) 半導体メモリを用いた、代表的な補助記憶装置。

^{*2} 図の例からもわかるとおり、宣言した順に変数の領域が割り当てられたり、隙間なく割り当てられたりするとは限りません。

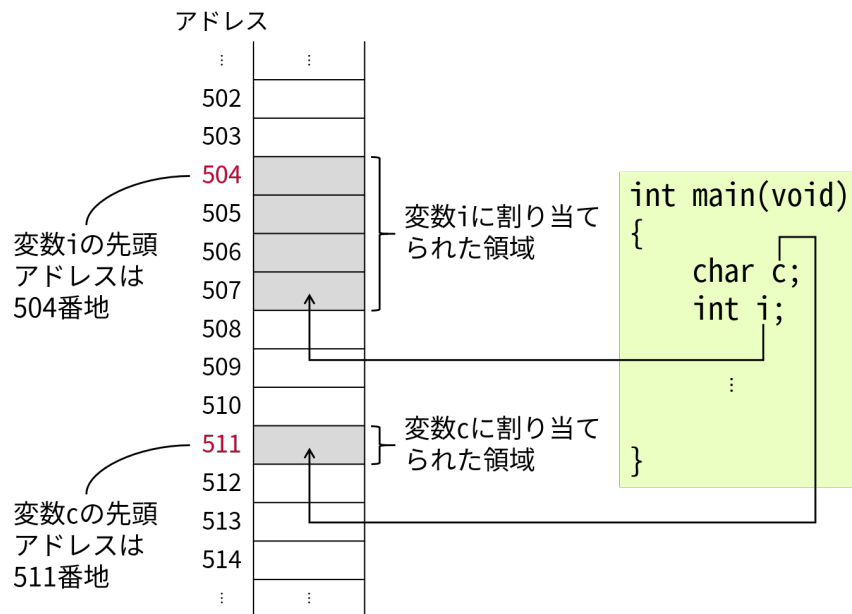


図 6.1 メモリ上の変数の領域のイメージ

表 6.1 単項 & 演算子

演算子	役割
&a	a の先頭アドレス

以降、変数のアドレスといえば、先頭アドレスを指すことにします。`printf()` 関数でアドレスを表示するための変換指定は、`%p` です。`%p` による出力形式は、処理系によって異なります。ソースコード 6.1 は、`main()` 関数内で `char` 型の変数 `c` と `int` 型の変数 `i` を宣言し、それぞれのアドレスを表示するプログラムです。

ソースコード 6.1 変数のアドレスの表示 (ex134.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      char c;
6      int i;
7
8      printf("変数cのアドレス: %p\n", &c);
9      printf("変数iのアドレス: %p\n", &i);
10
11     return 0;

```

12 }

Windows 11 上の MinGW-w64(gcc version 13.2.0) での実行結果例

変数 c のアドレス: 000000CE82DFCCCF

変数 i のアドレス: 000000CE82DFCC8

FreeBSD 13.1 (amd64) 上の Clang 13.0.0 での実行結果例

変数 c のアドレス: 0x7fffffff91b

変数 i のアドレス: 0x7fffffff914

変数 c のアドレスを &c、変数 i のアドレスを &i で取得し、printf() 関数の変換指定 %p を使って表示しています。1 回目の実行結果例では、変数 c のアドレスは 000000CE82DFCCCF 番地、変数 i のアドレスは 000000CE82DFCC8 番地でした*3。2 つの実行結果例を見比べると、%p による出力形式が異なることがわかります。

6.2 アドレスを格納する変数

ソースコード 6.1 では、printf() 関数の引数にアドレスを取得する式を記述しました。次は、アドレスを一度変数に格納してから表示することにします。アドレスを格納する変数は、そのアドレスに格納されている値の型名の後ろにスペースと * を付けた型にします。たとえば、char 型の変数のアドレスは char * 型の変数に、int 型の変数のアドレスは int * 型の変数に格納します。

アドレスを変数に格納してから表示するプログラムを、ソースコード 6.2 に示します。

ソースコード 6.2 アドレスの変数への格納と表示 (ex135.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char c;
6      int i;
7      char *cp;
8      int *ip;
9
10     cp = &c;
```

*3 十六進数で表示されています。なお、OS が提供する仮想記憶と呼ばれる仕組みによって、プログラムから見えるこのアドレスは、仮想的なアドレスです。

```

11     ip = &i;
12
13     printf("変数cのアドレス: %p\n", cp);
14     printf("変数iのアドレス: %p\n", ip);
15
16     return 0;
17 }

```

Windows 11 上の MinGW-w64(gcc version 13.2.0) での実行結果例

変数 c のアドレス: 000000580C1FFA5F
 変数 i のアドレス: 000000580C1FFA58

7 行目で、char *型の変数 cp を宣言しています。型名が char *で、変数名が cp です。*cp という変数名ではありません*⁴。10 行目の `cp = &c;` で、cp に変数 c のアドレスを格納しています。そして、13 行目で、cp に格納された変数 c のアドレスを表示しています。

10 行目と 13 行目は、表 6.2 のような誤りをしないように注意してください。

表 6.2 アドレスを格納する変数に関する誤り

操作	正しい式	誤った式
アドレスの代入 (10 行目)	<code>cp = &c</code>	<code>*cp = &c</code>
アドレスの取り出し (13 行目)	<code>printf("%p", cp)</code>	<code>printf("%p", &cp)</code>

ソースコード 6.2 の ip は cp と同様なので、説明は省略します。

単項 & 演算子で得られるアドレスは**ポインタ**と呼ばれ、ポインタを格納する変数は**ポインタ変数**と呼ばれます。ソースコード 6.2 では、&c で得られるポインタを、ポインタ変数 cp に代入しています (図 6.2)。

ポインタとは、指し示すものという意味です。&c で得られるポインタは変数 c を指しています。ポインタは、**アドレスの情報**に加えて、**指す先の型に関する情報**も持っていると考えることができます。たとえば、ソースコード 6.2 の実行結果例では、&c によって char *型の 000000580C1FFA5F が得られました。000000580C1FFA5F 番地という場所の情報に加えて、その場所が char 型の値が格納される領域であるという情報も持っています。同様に、&i によって int *型の 000000580C1FFA58 が得られました。000000580C1FFA58 番地という場所の情報に加えて、その場所から始まる 4 バイトが int 型の値が格納される領域であるという情報も持っています。

*⁴ 型名と変数名の間にスペースを入れて、`char * cp;` としてもかまいません。ただし、スペースは入れないことが一般的と思われます。

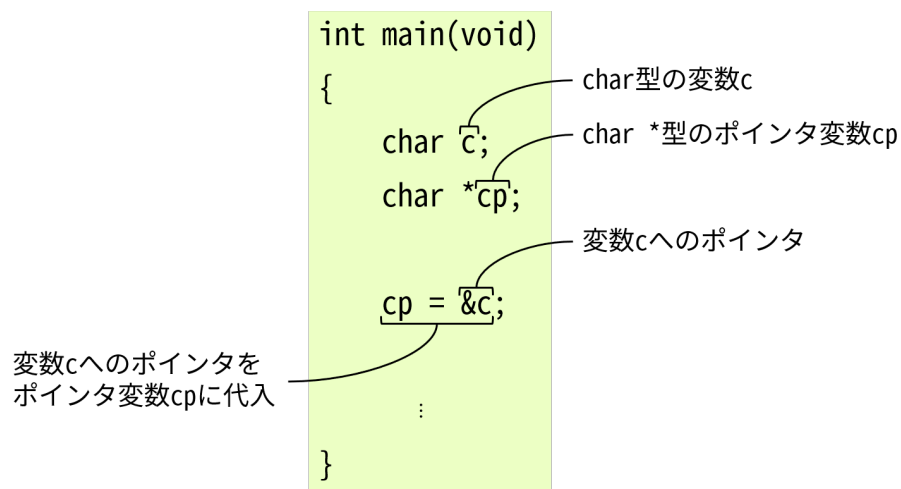


図 6.2 ポインタ変数へのポインタの代入

ポインタ変数 cp は、格納している値 (指している変数) に着目して **c へのポインタ** と呼ばれたり、指している領域の型に着目して **char へのポインタ** と呼ばれたりします。

ポインタ変数も変数であり、メモリ上に領域が割り当てられます。図 6.2 で、cp に c へのポインタを代入したときのメモリのイメージを、図 6.3 に示します。

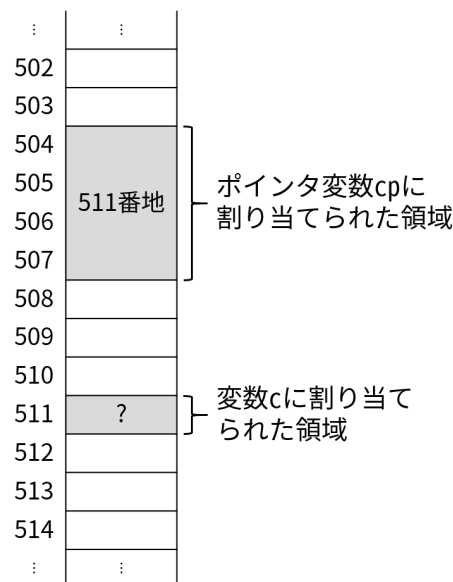


図 6.3 ポインタ変数の領域のイメージ

ポインタ変数の大きさは処理系によって異なりますが、ここでは 4 バイトであることを前提にしています。変数 c のアドレスが 511 番地であるとすれば、`cp = &c` によって、cp に 511 が代入されます。

型が同じ複数のポインタ変数をまとめて宣言するときは注意が必要です。

```
int *x, y;
```

は、`int *`型の変数 `x` と `int` 型の変数 `y` を宣言したと見なされます。つまり、`x` はポインタ変数ですが、`y` はそうではありません。`x` も `y` も `int *`型にするには、

```
int *x, *y;
```

と宣言する必要があります。

```
int x, *y;
```

は、`int` 型の変数 `x` と `int *`型のポインタ変数 `y` の宣言です。

6.3 ポインタが指すオブジェクト

何らかの値を表すメモリ上の領域を**オブジェクト**といいます。変数はオブジェクトで、変数にはメモリ上の領域が対応しています^{*5}。ポインタに対して、単項演算子 `*` を適用すると、そのポインタが指すオブジェクトが得られます (表 6.3)。

表 6.3 単項 `*` 演算子

演算子	役割
<code>*a</code>	ポインタ <code>a</code> が指すオブジェクト

たとえば次のソースコードの 4 行目のように、`int *`型のポインタ変数 `ip` に、`int` 型の変数 `i` のアドレスを格納したとします。

```
1 int i = 3, j;  
2 int *ip;  
3  
4 ip = &i;  
5  
6 j = *ip; /* ipが指すオブジェクトの値、つまり変数iの値をjに代入 */  
7 *ip = 5; /* ipが指すオブジェクト、つまり変数iに5を代入 */
```

このとき、`*ip` は、`ip` が指すオブジェクト、つまり変数 `i` を表します。`*ip` を評価すると変数 `i` の値が得られます。また、`*ip` を代入演算子 `=` の左に書くと、右に書いた値が変数 `i` に代入されます。

単項演算子 `*` の使用例を、ソースコード 6.3 に示します。

^{*5} オブジェクトに付けた名前が変数名です。

ソースコード 6.3 単項演算子 * の使用例 (ex136.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x = 3, y = 5;
6      int *ip;
7
8      printf("x = %d\n", x);
9
10     ip = &x;
11     *ip = 7;
12
13     printf("x = %d\n", x);
14
15     ip = &y;
16     x = *ip;
17
18     printf("x = %d\n", x);
19
20     return 0;
21 }

```

実行結果

```

x = 3
x = 7
x = 5

```

このプログラムでは、`int` 型の変数 `x` と `y` を、それぞれ初期値を 3 と 5 として宣言します。8 行目で `x` の値が初期値の 3 であることを確認します。10 行目で、`int` へのポインタ変数 `ip` に `x` へのポインタを代入します。

11 行目の式は、`*ip = 7` です。`ip` が指すオブジェクト、つまり `x` に 7 を代入します。`x` の値が 7 に変わったことを、13 行目で確認します。

15 行目で、それまで `x` へのポインタが格納されていたポインタ変数 `ip` に、`y` へのポインタを代入します。これによって、`ip` は `y` を指すようになります。16 行目で、`x` に `ip` が指すオブジェクトの値、つまり `y` の値を代入します。`x` の値が `y` の値である 5 に変わったことを、18 行目で確認

します。

このプログラムの実行時のメモリのイメージを、図 6.4 に示します。変数 x のアドレスが 504 番地、変数 y のアドレスが 512 番地であることを想定しており、10 行目、11 行目、15 行目、16 行目の実行直後のメモリの状態を表しています。図中の矢印は、ポインタ変数 ip が指すオブジェクトを表します。

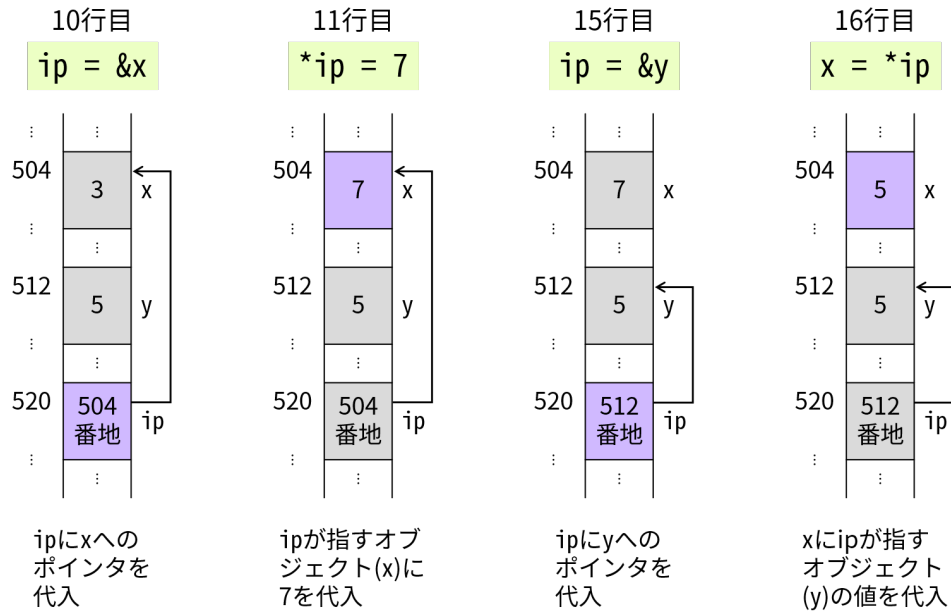


図 6.4 ex136.c の実行時のメモリのイメージ

単項演算子 $\&$ はオブジェクトからポインタを得て、単項演算子 $*$ はポインタからオブジェクトを得ることから、ちょうど逆の役割を持っています (図 6.5)。 a を変数とすると、 $\&a$ は“ a へのポインタ”が指すオブジェクトなので、 a そのものです。

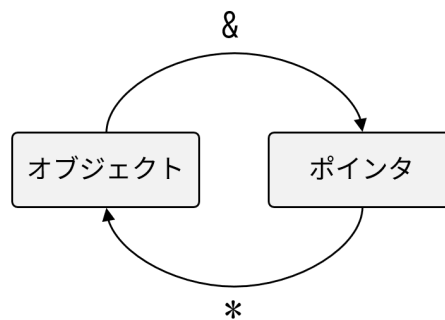


図 6.5 単項演算子 $\&$ と $*$

なお、ポインタ変数の宣言に使う $*$ と単項 $*$ 演算子は異なるものです。混乱しないようにしてください。

練習課題 12 (ex137.c)

次のようなプログラムを作ってください。

1. `int` 型の変数 `i` を初期値 3 で、`double` 型の変数 `d` を初期値 1.5 で宣言する。
2. `i` へのポインタを格納するためのポインタ変数 `ip` と、`d` へのポインタを格納するためのポインタ変数 `dp` を宣言する。
3. `ip` に `i` へのポインタを、`dp` に `d` へのポインタを代入する。
4. `i` を使わずに (`ip` を使って)、`i` の値を 2 増やす。
5. `i` と `d` を使わずに (`ip` と `dp` を使って)、`d` の値を `i` 倍する。
6. `i` の値と `d` の値を表示する。

`i` の値は $3 + 2 = 5$ 、`d` の値は $1.5 \times 5 = 7.5$ になります。途中まで作ったプログラムを、授業のサイトに `ex137.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、上の説明どおりに動作するプログラムを作ってください。

実行結果

```
i = 5
d = 7.500000
```

第 7 章

関数の引数におけるポインタ

ソースコード 7.1 をご覧ください。実行したときどのような出力になるか予測できると思います。

ソースコード 7.1 値渡しの確認 (ex138.c)

```
1  #include <stdio.h>
2
3  void decuple1(int);
4
5  int main(void)
6  {
7      int x = 5;
8
9      printf("x = %d\n", x);
10
11     decuple1(x);
12
13     printf("x = %d\n", x);
14
15     return 0;
16 }
17
18 void decuple1(int y)
19 {
20     y *= 10;
21 }
```

実行結果

```
x = 5
x = 5
```

main() 関数内である 11 行目で、`x` を実引数として関数 `decuple1()` を呼び出します。`decuple1()` は、仮引数 `y` で `x` の値を受け取ります。そして、20 行目で `y` の値を 10 倍に更新します。引数として 5 を受け取るので、`y` の値が 5 から 50 に更新されることになります。

main() 関数で `decuple1()` を呼び出す前後での `x` の値を表示します。`decuple1()` を呼び出す前には、`x` の値は初期値の 5 です。`decuple1()` では仮引数 `y` の値を更新していますが、呼び出し側の実引数 `x` の値には影響を与えません。`decuple1()` を呼び出した後の `x` の値も 5 です。

関数呼び出しでは、**実引数の値が仮引数の変数にコピー**されます。このような引数の渡し方を**値渡し**と呼ぶことを“C 言語基礎演習”で学習しました。

ソースコード 7.2 で、main() 関数の `x` と `decuple1()` 関数の `y` のアドレスを確認してみましょう。

ソースコード 7.2 `x` と `y` のアドレスの確認 (ex139.c)

```
1  #include <stdio.h>
2
3  void decuple1(int);
4
5  int main(void)
6  {
7      int x = 5;
8
9      printf("変数xのアドレス: %p\n", &x);
10
11     decuple1(x);
12
13     return 0;
14 }
15
16 void decuple1(int y)
17 {
18     printf("変数yのアドレス: %p\n", &y);
19
20     y *= 10;
21 }
```

Windows 11 上の MinGW-w64(gcc version 13.2.0) での実行結果例

変数 x のアドレス: 0000003CD01FF6DC

変数 y のアドレス: 0000003CD01FF6B0

実行結果から、x と y は異なる領域が割り当てられた異なる変数であることが確認できます。y の値を変更したとしても、x の値には影響がありません。

decuple1() の仮引数の名前を y から x に変更しても、main() 関数の x と decuple1() 関数の x は異なる変数です。このことは、“C 言語基礎演習”で変数のスコープとして説明しました。それぞれの変数に割り当てられた領域 (アドレス) から異なる変数であることが確認できます。

変数の値が呼び出し先の関数で書き換えられないことは値渡しの特長である一方、書き換えられると便利なこともあります。次のようにすれば、呼び出し元の関数で宣言された変数の値を、呼び出し先の関数で書き換えられます。

1. 呼び出し先の関数に、変数の値ではなく、変数へのポインタを渡す。
2. 呼び出し先の関数は、渡されたポインタが指すオブジェクトの値を書き換える。

ソースコード 7.1 を改変して、main() 関数の x の値を呼び出し先の関数で書き換えるようにしたプログラムを、ソースコード 7.3 に示します。

ソースコード 7.3 呼び出し先の関数での変数の値の書き換え (ex140.c)

```
1  #include <stdio.h>
2
3  void decuple1(int);
4  void decuple2(int *);
5
6  int main(void)
7  {
8      int x = 5;
9
10     printf("x = %d\n", x);
11
12     decuple1(x); /* 変数xの値を渡す */
13
14     printf("x = %d\n", x);
15
16     decuple2(&x); /* 変数xへのポインタを渡す */
17
18     printf("x = %d\n", x);
```

```

19
20     return 0;
21 }
22
23 /* 仮引数yで変数xの値を受け取る */
24 void decuple1(int y)
25 {
26     y *= 10;
27 }
28
29 /* 仮引数ipで変数xへのポインタを受け取る */
30 void decuple2(int *ip)
31 {
32     *ip *= 10;
33 }

```

実行結果

```

x = 5
x = 5
x = 50

```

ソースコード 7.1 に関数 `decuple2()` を追加し、`main()` 関数から呼び出した上で呼び出し前後の `x` の値を表示します。`decuple2()` には、`int` 型の変数へのポインタを渡します。`decuple2()` の仮引数は、`int *`型です。`decuple2()` 関数内の 32 行目の式 `*ip *= 10` は、仮引数で受け取ったポインタが指すオブジェクトの値、つまり `main()` 関数の変数 `x` の値を 10 倍に更新します。

ソースコード 7.3 における `decuple1()` と `decuple2()` の処理のイメージを、それぞれ図 7.1 と図 7.2 に示します。

このように、関数の引数としてポインタを渡すことによって、呼び出し元の変数の値を間接的に書き換えることができます。

decuple1() の `y *= 10` はローカル変数 `y` の値を更新

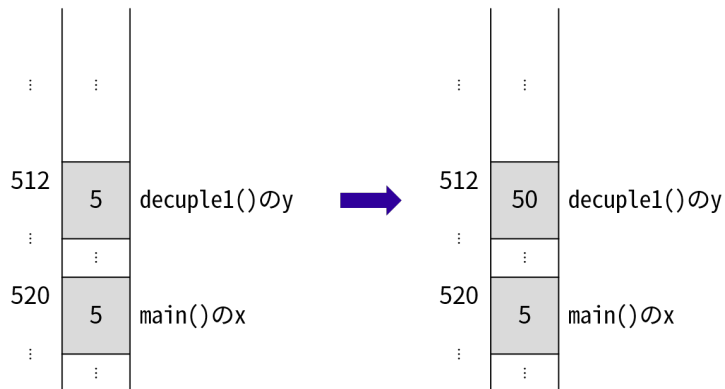


図 7.1 `decuple1()` の処理のイメージ

`decuple2()` の `*ip *= 10` は
`ip` が指す `main()` の `x` の値を更新

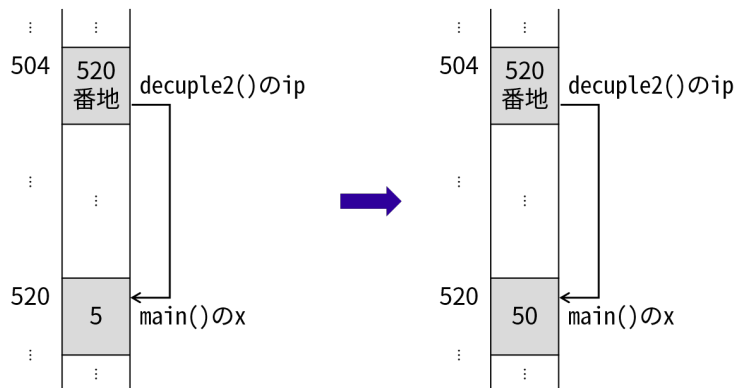


図 7.2 `decuple2()` の処理のイメージ

練習課題 13 (ex141.c)

`double` へのポインタを受け取り、そのポインタが指す値を、非負の平方根に書き換える関数 `sqrt2()` を作ってください。ポインタが指す値は、非負の数であることを前提にしてください。たとえば、`double` 型の変数 `d` の値が 5 であるとき、`d` へのポインタを引数にして `sqrt2()` を呼び出すと、`d` の値が $\sqrt{5} = 2.236\dots$ に書き換えられます。
`sqrt2()` 関数内では、次のライブラリ関数 `sqrt()` を使ってください。

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

途中まで作ったプログラムを、授業のサイトに `ex141.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

0 以上の数 > 5 ↵
平方根は 2.236068 です。

練習課題 14 (`ex142.c`)

`int` へのポインタを 2 つ受け取り、それぞれのポインタが指す値を交換する関数 `swap_int()` を作ってください。たとえば、`int` 型の変数 `x` の値が 3、`int` 型の変数 `y` の値が 5 であるとき、`x` へのポインタと `y` へのポインタを引数にして `swap_int()` を呼び出すと、`x` の値が 5、`y` の値が 3 に書き換えられます。

途中まで作ったプログラムを、授業のサイトに `ex142.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

`x` の値 > 3 ↵
`y` の値 > 5 ↵

`x` = 5
`y` = 3

“C 言語基礎演習”では、`scanf()` 関数の第 2 引数に指定する変数名の前に `&` を付けると説明しました。ここまで読み進めれば、第 2 引数として変数へのポインタを指定していたことがわかります。`scanf()` 関数では、キーボードからの入力に応じて、呼び出し元の変数の値を書き換えます。変数の値を渡されても書き換えることができません。ポインタを渡してもらうことにより、呼び出し元の変数の値を書き換えています。

第 8 章

配列とポインタ

8.1 配列の要素を指すポインタ

配列は、添字が小さい要素から順に、メモリ上の連続した領域が割り当てられます。要素数が 5 の `int` 型の配列 `ia` のイメージを、図 8.1 に示します。

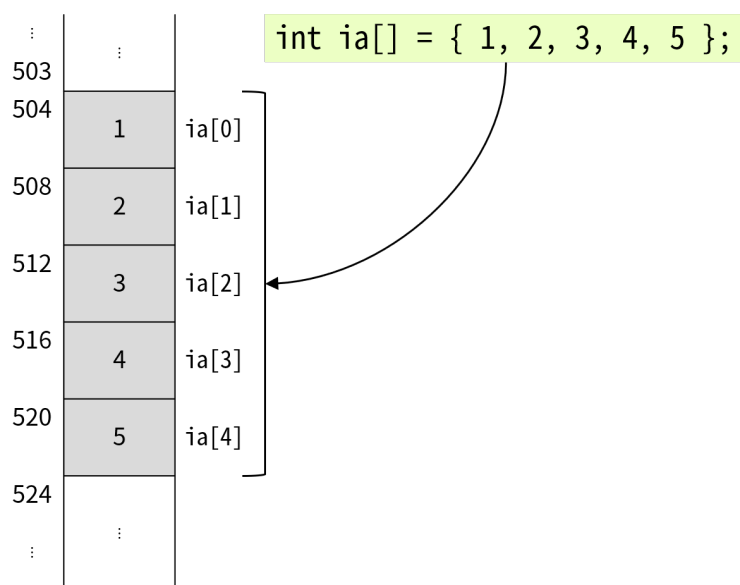


図 8.1 配列の領域のイメージ

図では、`int` 型が 4 バイトであることを前提にしており、`ia` の領域として 504 番地から 523 番地が割り当てられています。`ia[0]` のアドレスは 504 番地、`ia[1]` のアドレスは 508 番地です。

配列の変数名を評価すると、その配列の先頭要素へのポインタが得られます。図の例では、`ia` は `&ia[0]` と同義で、その値は `int *` 型の 504 です。

ここまですを確認するプログラムを、ソースコード 8.1 に示します。

ソースコード 8.1 配列とメモリ上の領域 (ex143.c)

```

1  #include <stdio.h>
2
3  #define NUM 5
4
5  int main(void)
6  {
7      char ca[NUM];
8      int ia[NUM];
9      int i;
10
11     /* char型の大きさを表示 */
12     printf("sizeof(char) = %d\n\n", (int)sizeof(char));
13
14     /* caの各要素のアドレスを表示 */
15     for (i = 0; i < NUM; ++i)
16     {
17         printf("&ca[%d] = %p\n", i, &ca[i]);
18     }
19
20     /* caを評価した値を表示 */
21     printf("\nca      = %p\n\n", ca);
22
23     /* int型の大きさを表示 */
24     printf("sizeof(int) = %d\n\n", (int)sizeof(int));
25
26     /* iaの各要素のアドレスを表示 */
27     for (i = 0; i < NUM; ++i)
28     {
29         printf("&ia[%d] = %p\n", i, &ia[i]);
30     }
31
32     /* iaを評価した値を表示 */
33     printf("\nia      = %p\n", ia);
34
35     return 0;

```

Windows 11 上の MinGW(gcc version 9.2.0) での実行結果例

```
sizeof(char) = 1

&ca[0] = 0061FF17
&ca[1] = 0061FF18
&ca[2] = 0061FF19
&ca[3] = 0061FF1A
&ca[4] = 0061FF1B

ca = 0061FF17

sizeof(int) = 4

&ia[0] = 0061FF00
&ia[1] = 0061FF04
&ia[2] = 0061FF08
&ia[3] = 0061FF0C
&ia[4] = 0061FF10

ia = 0061FF00
```

要素数 5(マクロ NUM で定義) の `char` 型の配列 `ca` と `int` 型の配列 `ia` を宣言し、次の値を表示します。

- `char` 型の大きさ
- `ca` の各要素のアドレス
- `ca` を評価した値 (`ca` の先頭要素のアドレス)
- `int` 型の大きさ
- `ia` の各要素のアドレス
- `ia` を評価した値 (`ia` の先頭要素のアドレス)

実行結果の読みやすさのため、ここでは MinGW-w64 ではなく、MinGW での実行結果例を示しました。`ca` の要素のアドレスは、`ca[0]` が 0061FF17、`ca[1]` が 0061FF18、…、`ca[4]` が 0061FF1B です。`char` 型の大きさが 1 なので、`ca` の要素は添字の順に隙間なく並んでいることがわかります。`ia` の要素のアドレスは、添字の順に 0061FF00、0061FF04、…、0061FF10 です。

int 型の大きさが 4 なので、ia の要素も添字の順に隙間なく並んでいることがわかります。また、ca と &ca[0] が同じであり、ia と &ia[0] が同じであることも確認できます。

配列の要素を指すポインタに整数 a を足すと、**a 個後ろの要素を指すポインタ**になります。また、整数 a を引くと、**a 個前の要素を指すポインタ**になります。int へのポインタなら sizeof(int) 単位で、double へのポインタなら sizeof(double) 単位で、アドレスが加減算されます。1 バイト単位ではありませんので、注意してください。このことを確認するプログラムを、ソースコード 8.2 に示します。

ソースコード 8.2 ポインタへの加減算 (ex144.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int ia[] = { 1, 2, 3, 4, 5 };
6      int *ip;
7
8      /* ipはia[2]を指す */
9      ip = &ia[2];
10
11     /* ipの値とipが指すオブジェクトの値を表示 */
12     printf("ip = %p\n", ip);
13     printf("*ip = %d\n\n", *ip);
14
15     /* (ip + 2)の値と(ip + 2)が指すオブジェクトの値を表示 */
16     printf("ip + 2 = %p\n", ip + 2);
17     printf("*(ip + 2) = %d\n\n", *(ip + 2));
18
19     /* (ip - 1)の値と(ip - 1)が指すオブジェクトの値を表示 */
20     printf("ip - 1 = %p\n", ip - 1);
21     printf("*(ip - 1) = %d\n\n", *(ip - 1));
22
23     /* ipをインクリメント。ipはia[3]を指す */
24     ++ip;
25
26     /* ipの値とipが指すオブジェクトの値を表示 */
27     printf("ip = %p\n", ip);
28     printf("*ip = %d\n", *ip);
```

```
29
30     return 0;
31 }
```

Windows 11 上の MinGW(gcc version 9.2.0) での実行結果例

```
ip = 0061FF10
*ip = 3

ip + 2 = 0061FF18
*(ip + 2) = 5

ip - 1 = 0061FF0C
*(ip - 1) = 2

ip = 0061FF14
*ip = 4
```

ia は要素数 5 の int 型の配列で、要素の値は先頭から順に 1、2、3、4、5 です。ip は int へのポインタで、9 行目で ia[2] へのポインタを代入します。

12 行目で ip の値 (アドレス) を表示し、13 行目で ip が指すオブジェクトの値、つまり ia[2] の値を表示します。

16 行目で ip + 2 の値を表示します。実行結果例では、ip の値は 0061FF10 で、ip + 2 の値は 0061FF18 です。2 を足すことによって、int 型の大きさ 2 つ分の 8 増えていることがわかります。ip + 2 は、ia[2] の 2 つ後ろの要素である ia[4] を指すポインタです。17 行目で、ip + 2 が指すオブジェクトの値を表示し、このことを確認します。

20 行目と 21 行目では、ip - 1 で同様の処理を行います。実行結果から、ip - 1 が、ip が指す要素の 1 つ前の要素、つまり ia[1] を指すポインタであることがわかります。

24 行目では、`++ip;` を実行します。++ip は ip = ip + 1 と同じなので、ip が 1 つ後ろの要素、つまり ia[3] を指すようになります。27 行目と 28 行目で ip の値と ip が指すオブジェクトの値を表示することによって、このことを確認します。

ソースコード 8.2 の実行時におけるメモリのイメージを、図 8.2 に示します。

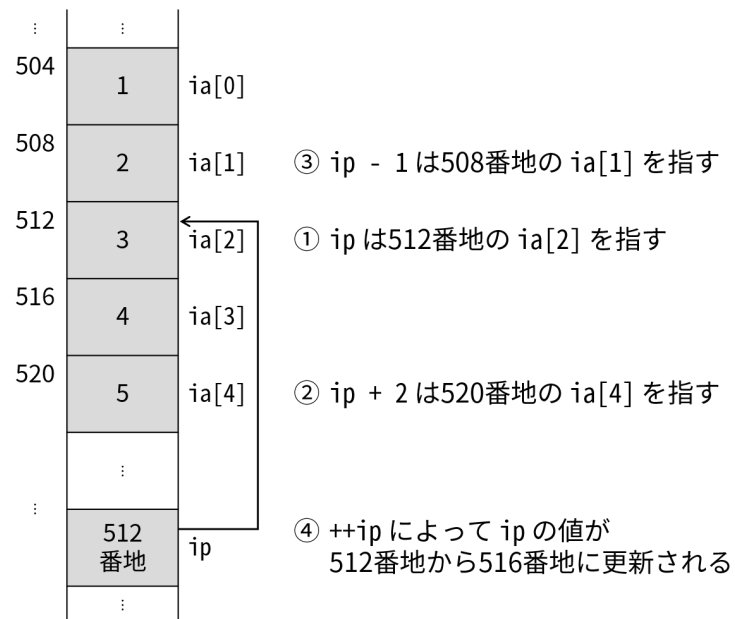


図 8.2 ソースコード 8.2 の実行時におけるメモリのイメージ

次は、ソースコード 8.3 の動作を考えます。

ソースコード 8.3 ポインタによる配列の操作 (ex145.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int ia[] = { 1, 2, 3, 4, 5 };
6      int *ip;
7      int len;
8      int i;
9
10     ip = ia; /* ipはiaの先頭要素を指す */
11     len = (int)(sizeof ia / sizeof ia[0]); /* lenはiaの要素数 */
12
13     /* iaの要素の値を2倍する */
14     for (i = 0; i < len; ++i)
15     {
16         *(ip + i) *= 2;
17     }

```

```

18
19     /* iaの要素の値を表示する */
20     for (i = 0; i < len; ++i)
21     {
22         printf("%d\n", *ip++);
23     }
24
25     return 0;
26 }

```

実行結果

```

2
4
6
8
10

```

初期値が先頭から順に 1、2、3、4、5 である int 型の配列 `ia` に対して、すべての要素の値を 2 倍した後、それらを表示するプログラムです。10 行目で `ip` に `ia` の先頭要素へのポインタを代入し、11 行目で `len` に `ia` の要素数を代入します。14 行目から始まる `for` 文で、`ia` のすべての要素の値を 2 倍します。

20 行目から始まる `for` 文で、`ia` のすべての要素の値を表示します。この `for` 文は、22 行目の `printf()` 関数を `len` 回実行します。`len` は 5 なので、この `for` 文は、

```

printf("%d\n", *ip++);
printf("%d\n", *ip++);
printf("%d\n", *ip++);
printf("%d\n", *ip++);
printf("%d\n", *ip++);

```

と同じです。`printf()` 関数の第 2 引数である `*ip++` の動作は次のとおりです。

この式を評価する前に `ip` が指している要素を、“現在の要素”と呼ぶことにします。この式は、`ip` に対して、単項 `*` 演算子と後置のインクリメント演算子が適用されています。後置のインクリメント演算子の方が優先順位が高いため*1、`*(ip++)` と解釈されます。`ip++` を評価すると、`ip` は、現在の要素の 1 つ後ろの要素を指すポインタになります。ただし、後置なので、`ip++` を評価して得られる値は、現在の要素へのポインタです。この値 (ポインタ) に対して単項 `*` 演算子を適

*1 表 1.1 を参照してください。

用することで、現在の要素の値が得られます。まとめると、`*ip++` は、`ip` が指している現在の要素の値を取り出した上で、`ip` が指す要素を 1 つ後ろの要素に更新します。20 行目から始まる `for` 文のループ本体の動作は、表 8.1 のようになります。

表 8.1 ソースコード 8.3 の 20 行目から始まる `for` 文のループ本体の動作

反復回	処理
1 回目	<code>ip</code> が指す 1 つ目の要素の値を表示した上で、 <code>ip</code> が指す要素を 2 つ目の要素に更新する。
2 回目	<code>ip</code> が指す 2 つ目の要素の値を表示した上で、 <code>ip</code> が指す要素を 3 つ目の要素に更新する。
	⋮
5 回目	<code>ip</code> が指す 5 つ目の要素の値を表示した上で、 <code>ip</code> が指す要素を 1 つ後ろに更新する ^{*2} 。

配列や第 10 章で学習する文字列を扱うプログラムでは、ポインタに対して、単項 `*` 演算子とインクリメント演算子を適用する式がよく使われます。

8.2 関数への配列の受け渡し

練習課題 9 では、配列の要素の値の最大値を求めるプログラムを作りました。正解例では、`main()` 関数の中に最大値を求める処理を記述しました。ここでは、最大値を求める処理を関数にすることを考えます。そのためには関数に配列を渡す必要があります。しかし、C 言語では配列自体を渡すこと、つまり関数の仮引数に、実引数の配列をコピーすることはできません。

そこで、配列の先頭要素へのポインタを関数に渡すようにします。ソースコード 8.4 は、練習課題 9 の正解例を、最大値を求める関数を使うように変更したものです。

ソースコード 8.4 配列の先頭要素へのポインタの受け渡し (`ex146.c`)

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define NUM 6
6
7  int array_max(int *);
8

```

^{*2} `ip` は存在しない 6 つ目の要素を指すようになります。この状態で、`ip` が指すオブジェクトを参照しないようにしてください。

```

9  int main(void)
10 {
11     int ia[NUM];
12     int i;
13
14     srand((unsigned int)time(NULL));
15
16     /* ia[0]～ia[NUM - 1]に0以上9以下の乱数を代入 */
17     for (i = 0; i < NUM; ++i)
18     {
19         ia[i] = rand() % 10;
20     }
21
22     /* iaの要素の値を表示 */
23     for (i = 0; i < NUM; ++i)
24     {
25         printf("[%d] ", i);
26     }
27     printf("\n");
28     for (i = 0; i < NUM; ++i)
29     {
30         printf(" %d ", ia[i]);
31     }
32     printf("\n");
33
34     printf("\n最大値は %d です\n", array_max(ia));
35
36     return 0;
37 }
38
39 int array_max(int *ip)
40 {
41     int max, i;
42
43     for (max = *ip, i = 1; i < NUM; ++i)
44     {
45         if (*(ip + i) > max)

```

```

46     {
47         max = *(ip + i);
48     }
49 }
50
51 return max;
52 }

```

練習課題 9 と同じ動作なので、実行結果例は省略します。このプログラムでは、配列の要素の値の最大値を求める関数 `array_max()` を定義しています。`array_max()` は、仮引数 `ip` で `int` 型の配列の先頭要素へのポインタを受け取ります。そして、`ip` が指す要素を後ろにずらしながら、`NUM` 個の要素の値を調べ、その最大値を返します。

34 行目で `array_max()` 関数を呼び出すときの実引数は、配列 `ia` の先頭要素へのポインタである `ia` です。実引数を `&ia[0]` にしても正しく動作しますが、配列名だけ (`ia`) の方が表記が簡単です。

`array_max()` 関数では、渡される配列の要素数が `NUM` であることを前提にしています。次は、`array_max()` 関数を、任意の要素数の配列に対応させることを考えます。

第 5 章で、`int` 型の配列 `ia` の要素数が、

```
sizeof ia / sizeof(int)
```

または

```
sizeof ia / sizeof ia[0]
```

で求められることを学びました。しかし、`array_max()` 関数の `ip` に対してこの方法は使えません。なぜなら、`sizeof ip` は、“`ip` が指す要素を先頭要素とする配列の大きさ”ではなく、`ip` 自体の大きさ、つまり `int *`型の大きさになるからです。このことを、ソースコード 8.5 で確認してみましょう。

ソースコード 8.5 配列の要素へのポインタと `sizeof` 演算子 (ex147.c)

```

1  #include <stdio.h>
2
3  void func(int *);
4
5  int main(void)
6  {
7      int ia[] = { 1, 2, 3, 4, 5 };
8
9      printf("sizeof ia = %d\n", (int)sizeof ia);

```

```

10
11     func(ia);
12
13     return 0;
14 }
15
16 void func(int *ip)
17 {
18     printf("sizeof ip = %d\n", (int)sizeof ip);
19 }

```

実行結果例

```

sizeof ia = 20
sizeof ip = 4

```

実行結果例の処理系では、`int` 型の大きさは 4 で、`int *`型の大きさも 4 です。`main()` 関数内の `sizeof ia` は、配列 `ia` の要素 1 つの大きさ (4) の要素数 (5) 倍で、20 になります。`func()` 関数内の `sizeof ip` は、`int *`型の大きさの 4 になります。

`ia` は配列の先頭要素へのポインタなのに、`sizeof ia` で“`ia` の先頭要素へのポインタの大きさ”ではなく配列全体の大きさが得られることに疑問を持たれたかもしれません。実は、配列の変数名がその先頭要素へのポインタであるという規則は、配列の変数名が `sizeof` 演算子の被演算子である場合は適用されません。`sizeof 配列の変数名` は“配列の先頭要素へのポインタの大きさ”ではなく、配列全体の大きさになります。

`array_max()` のような関数を任意の要素数の配列に対応させるには、引数として要素数も受け取るようにします。ソースコード 8.4 をそのように変更したものの主要部分をソースコード 8.6 に、ソースコード 8.6 における関数呼び出しのイメージを図 8.3 に示します。

ソースコード 8.6 ソースコード 8.4 の改変 (ex148.c)

```

1  /* 前略 */
2
3  #define NUM 6
4
5  int array_max(int *, int);
6
7  int main(void)
8  {
9      int ia[NUM];

```

```

10
11     /* 中略 */
12
13     printf("\n最大値は %d です\n", array_max(ia, NUM));
14
15     return 0;
16 }
17
18 int array_max(int *ip, int len)
19 {
20     int max, i;
21
22     for (max = *ip, i = 1; i < len; ++i)
23     {
24         if (*(ip + i) > max)
25         {
26             max = *(ip + i);
27         }
28     }
29
30     return max;
31 }

```

```

int main(void)
{
    :
    array_max(ia, NUM)
    :
}

int array_max(int *ip, int len)
{
    :
}

```

図 8.3 ソースコード 8.6 における `array_max()` 関数の呼び出しのイメージ

ソースコード 8.6 では、ソースコード 8.4 からの変更箇所の下線を引いています。array_max() の第 2 引数で、配列の要素数を受け取るように改変しました。main() 関数における array_max() 関数の呼び出しでは、ia の要素数として NUM を使っています。NUM の代わりに (int)(sizeof ia / sizeof ia[0]) を使うこともできます。

なお、array_max() 関数で、配列の要素数を受け取る仮引数は int 型にしました。C 言語では、このような大きさ (長さ) を表す値には、int 型ではなく size_t 型を使うことが一般的ですが、その場合は int 型との演算などで考慮しなければならない点があります。そこで、この授業では簡略化のために int 型を使います。

練習課題 15 (ex149.c)

int 型の配列の先頭要素へのポインタと要素数を受け取り、配列のすべての要素の値を 10 倍にする関数 decuple_array() を作ってください。
途中まで作ったプログラムを、授業のサイトに ex149.c として掲載しています。コメント部分に decuple_array() 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。

実行結果

```
x[0] = 10
x[1] = 20
x[2] = 30

y[0] = 90
y[1] = 80
y[2] = 70
y[3] = 60
```

【発展】 配列風の表記とポインタ風の表記

第 5 章では、配列の各要素を、**配列の変数名 [添字]** という書式で表すことを学びました。ここで使っている [] は二項演算子です。[] の被演算子は、一方が配列の要素へのポインタで、もう一方が整数です。前者を E1、後者を E2 とします。式 E1[E2] を評価すると、E1 が指す要素の E2 個後ろの要素の値が得られます。E2 が負の数の場合は、前の要素になります。E1[E2] は $*(E1 + (E2))$ と等価です。

配列 ia とポインタ変数 ip が次のような状態であるとしします。

```

1  int ia[] = { 1, 2, 3, 4, 5 }, *ip;
2  ip = ia;

```

このとき、`ia[3]` の被演算子は、`ia` と `3` です。`ia` は配列 `ia` の先頭要素を指すポインタです。`ia[3]` を評価すると配列 `ia` の先頭要素の 3 つ後ろの要素、つまり 4 つ目の要素の値である 4 が得られます。`*(ia + 3)` も、配列 `ia` の先頭から 4 つ目の要素の値である 4 になります。`ip` は、配列 `ia` の先頭要素を指しているので、`ip[3]` は `ia[3]` と同じです。以上より、`ia[3]`、`*(ia + 3)`、`ip[3]`、`*(ip + 3)` はすべて同じです。`ia[3]` と `ip[3]` は配列風の書き方、`*(ia + 3)` と `*(ip + 3)` はポインタ風の書き方といえます。

次のように、`ip` に `ia[2]` へのポインタを格納したとします。

```

1  int ia[] = { 1, 2, 3, 4, 5 }, *ip;
2  ip = &ia[2];

```

`ip` が指す要素の 1 つ前の要素の値は、ポインタ風を書けば `*(ip - 1)` で、配列風を書けば `ip[-1]` です。どちらも値は 2 です。

関数に配列を渡すには、配列の先頭要素へのポインタを渡すことを学びました。たとえば、ソースコード 8.4 では、関数 `array_max()` の定義における仮引数の宣言を `int *ip` としました。配列の先頭要素へのポインタを `ip` で受け取ります。`ip` は `int` へのポインタですが、これを `int ip[]` と宣言してもよいことになっています。次のように、ポインタ風の宣言と、配列風の宣言が可能です。

```

1  /* ポインタ風の仮引数の宣言 */
2  int array_max(int *ip)
3  {
4      /* 略 */
5  }

```

```

1  /* 配列風の仮引数の宣言 */
2  int array_max(int ip[])
3  {
4      /* 略 */
5  }

```

書き方が違うだけで、どちらの宣言でも `ip` は `int` へのポインタです。配列風の宣言では、実引数の配列が値渡し (仮引数にコピー) されるように見えますが、そうではありません。渡された関数内でポインタが指す先の内容を書き換えると、呼び出し側の配列も書き換えられます。

このように配列の変数とポインタ変数は、同じような使い方ができます。混乱を避けるため、配列として宣言した変数は配列風に、ポインタとして宣言した変数はポインタ風に使うことをおすすめします。

8.3 型修飾子 const

変数の宣言の際に、型名の前に `const` を付けると^{*3}、その変数の値は変更できません。例を以下に示します。

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      const int x = 1;
6      const int y[] = { 1, 2, 3, 4, 5 };
7
8      printf("x = %d\n", x); /* 参照は可。 */
9      printf("y[1] = %d\n", y[1]); /* 参照は可。 */
10
11     x = 2; /* エラー。変数xは書き換え不可。 */
12     y[1] = 6; /* エラー。配列yの要素は書き換え不可。 */
13
14     return 0;
15 }
```

`x` も `y` も、宣言時に初期値を入れた後は、値を書き換えることはできません。このように、プログラム中で値を書き換える必要がない、もしくは書き換えてはいけない場合に `const` を使うことができます。`const` は、型に付加的な制約を付ける**型修飾子**の1つです。

“C 言語基礎演習”のソースコード 17.2(ex61.c)では、消費税率を表すマクロ `CONSUMPTION_TAX_RATE` を定義して使いました。`const` を付けた変数を使うように書き換えたものを、ソースコード 8.7 に示します。

ソースコード 8.7 型修飾子 `const` の使用例 (ex150.c)

```
1  #include <stdio.h>
2
```

^{*3} 後ろに付けることもできますが、この授業では前に付けることにします。

```

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

```

初期値が 10 である変数 `consumption_tax_rate` を宣言して使っています。宣言した後に `consumption_tax_rate` に値を代入することはできません。マクロ定義と異なり、`const` を付けた変数はスコープを限定できることが利点になりえます。

ポインタ変数の宣言において、型名の前に `const` を付けると、ポインタが指すオブジェクトの値を書き換えることができません。その例を以下に示します。

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i = 3;
6      const int *ip;
7
8      ip = &i; /* 代入は可。 */
9
10     printf("i = %d\\n", *ip); /* 参照は可。 */
11
12     *ip = 5; /* エラー。ipを介したオブジェクトの書き換えは不可。 */
13
14     return 0;
15 }

```

ポインタ変数 `ip` にポインタを代入する操作や、`ip` が指すオブジェクトの値を得る操作は行えま

すが、`ip` を介してオブジェクトを書き換えることができなくなります。

前節では、配列の先頭要素へのポインタを受け取る関数を扱いました。その際、受け取った関数内で呼び出し元の配列の要素を書き換えることができます。ポインタを受け取る仮引数の宣言に `const` を付けておけば、関数内で配列の要素を書き換えることを防止できます。

ソースコード 8.6 の `array_max()` 関数は、引数として受け取る配列の要素は書き換えません。そこで、ソースコード 8.8 のように `const` を付けることができます。

ソースコード 8.8 ポインタを受け取る仮引数における `const` の使用 (`ex151.c`)

```
1  /* 前略 */
2
3  int array_max(const int *, int);
4
5  /* 中略 */
6
7  int array_max(const int *ip, int len)
8  {
9      int max, i;
10
11     for (max = *ip, i = 1; i < len; ++i)
12     {
13         if (*(ip + i) > max)
14         {
15             max = *(ip + i);
16         }
17     }
18
19     return max;
20 }
```

ソースコード 8.6 との違いは、仮引数の宣言に `const` を付けただけです。こうしておけば、`array_max()` 関数が配列の要素を書き換えないことが明確になります。自分が作った関数ではない場合は、`const` が付いていることを確認することで、副作用なく呼び出せることがわかります。

なお、`array_max()` 関数は、第 2 引数の `len` も書き換えません。そのため、

```
int array_max(const int *, const int);
```

とすることもできますが、このような使い方はあまり一般的ではありません。仮引数における `const` は、ポインタが指すオブジェクトの値を書き換えないことを示すために使われます。

第 9 章

2 次元配列

配列は、型が同じ要素を並べたデータ構造でした。C 言語では、配列の要素を配列にすることもできます。たとえば、“要素数 3 の int 型の配列”を要素とする要素数 4 の配列などが実現できます。この配列を、図 9.1 に示します。

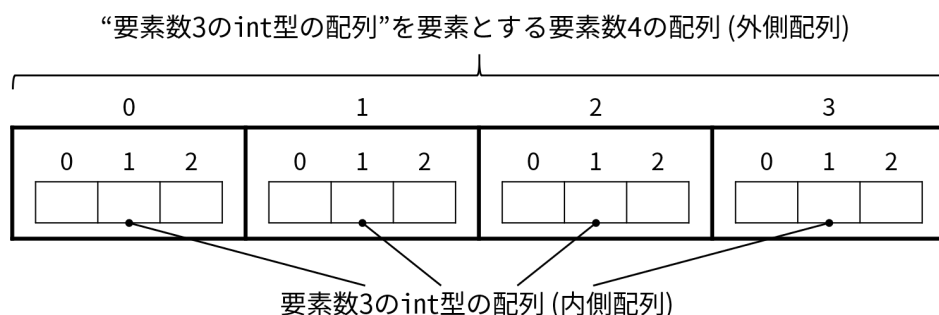


図 9.1 “要素数が 3 の int 型の配列”を要素とする要素数 4 の配列

このような配列を **2 次元配列**といい、これまでの配列を **1 次元配列**といいます*1。この授業では、配列を要素とする配列を**外側配列**、外側配列の要素である 1 次元配列を**内側配列**と呼ぶことにします*2。

2 次元配列の宣言の書式は、

内側配列の要素の型名 外側配列の変数名 [外側配列の要素数] [内側配列の要素数];

です。たとえば、図 9.1 のような構造を持つ 2 次元配列の変数 `ta` は、内側配列の要素の型が `int`、外側配列の変数名が `ta`、外側配列の要素数が 4、内側配列の要素数が 3 で、その宣言は

`int ta[4][3];`

となります。内側配列の要素は、

*1 要素が 2 次元配列である配列は 3 次元配列です。3 次元以上の配列はこの授業では扱いません。

*2 一般的な呼び方ではありません。

外側配列の変数名 [外側配列の添字] [内側配列の添字]

で表せます。配列 `ta` の内側配列の要素は、`ta[0][0]`、`ta[0][1]`、`ta[0][2]`、`ta[1][0]`、…、`ta[3][1]`、`ta[3][2]` の 12 個です (図 9.2)。

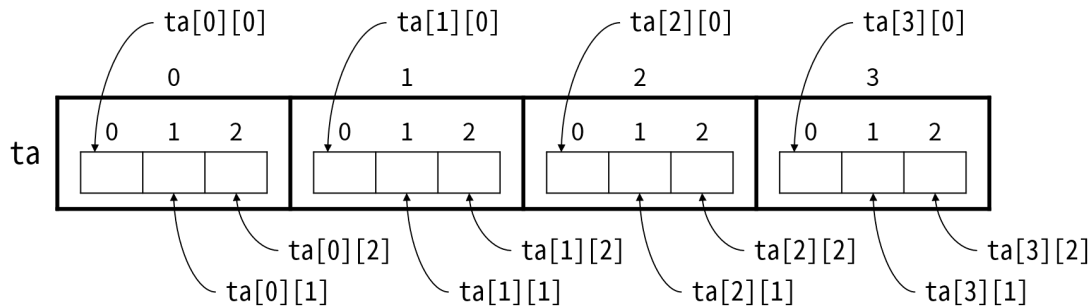


図 9.2 2次元配列 `ta` の内側配列の要素

2次元配列の内容を表すために、図 9.3 の左下のような表がよく使われます。

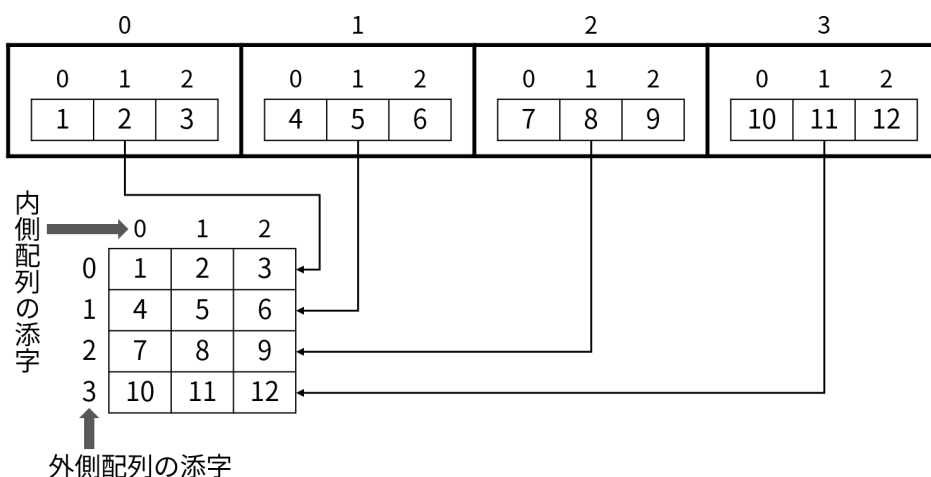


図 9.3 2次元配列の表現

外側配列の要素である内側配列を、それぞれ 1 行で表します。表の行数は外側配列の要素数、列数は内側配列の列数です。図 9.3 の 2次元配列は表で表すと 4 行 3 列になるので、“4 行 3 列の配列”のように呼ばれることもあります。その場合、外側配列の要素数を**行数**、内側配列の要素数を**列数**と呼びます。

2次元配列は要素が配列である配列ですが、行と列から構成される表形式に要素を並べたデータ構造と考えることもできます。その場合、2次元配列の宣言の書式は、

要素の型名 配列の変数名 [行数] [列数];

で、2次元配列の要素は、

配列の変数名 [行の添字] [列の添字]

とを考えます。この考え方に従って、“int 型の配列” の配列は、int 型の 2 次元配列といいます。以降では、**行**、**列**という表現も使います。

2 次元配列は、次のように初期化します。

```
1  int ta[4][3] = {  
2      { 1,  2,  3 },  
3      { 4,  5,  6 },  
4      { 7,  8,  9 },  
5      { 10, 11, 12 }  
6  };
```

1 次元配列の初期化に使う { 初期値₁, 初期値₂, ... } を行の数だけ , (コンマ) 区切りで並べ、全体を { } で囲みます。この書式は、1 次元配列同様、初期化のときだけしか使えません。この例では読みやすさのためにスペースと改行を入れていますが、これらは必須ではありません。初期化をする際は、次のように行数を省略できます。

```
1  /* 行数は 4 */  
2  int ta[][3] = {  
3      { 1,  2,  3 },  
4      { 4,  5,  6 },  
5      { 7,  8,  9 },  
6      { 10, 11, 12 }  
7  };
```

行数を省略した場合、初期値の数から計算されます。上の例では、行数は 4 になります。列数を省略することはできません。

1 次元配列は、for 文などを使って連続して処理することができました。2 次元配列も同様です。3 行 2 列の 2 次元配列の要素に値を代入した後、それらを表示するプログラムを、ソースコード 9.1 に示します。

ソースコード 9.1 for 文を使った 2 次元配列の操作の例 (ex152.c)

```

1  #include <stdio.h>
2
3  #define NUM_R 3 /* 行数 */
4  #define NUM_C 2 /* 列数 */
5
6  int main(void)
7  {
8      int ta[NUM_R][NUM_C];
9      int i, j;
10
11     /* taの要素に値を代入 */
12     for (i = 0; i < NUM_R; ++i)
13     {
14         for (j = 0; j < NUM_C; ++j)
15         {
16             printf("ta[%d][%d] = ", i, j);
17             scanf("%d", &ta[i][j]);
18         }
19     }
20     printf("\n");
21
22     /* taの要素の値を表示 */
23     for (i = 0; i < NUM_R; ++i)
24     {
25         for (j = 0; j < NUM_C; ++j)
26         {
27             printf(" %4d", ta[i][j]);
28         }
29         printf("\n");
30     }
31
32     return 0;
33 }

```

実行例

```
ta[0][0] = 9 ↵  
ta[0][1] = 57 ↵  
ta[1][0] = -61 ↵  
ta[1][1] = 8 ↵  
ta[2][0] = -14 ↵  
ta[2][1] = 42 ↵
```

```
  9   57  
-61   8  
-14  42
```

行の添字のために変数 *i* を、列の添字のために変数 *j* を使っています。27 行目の `printf()` 関数では、変換指定 `%4d` を使っています。これは、整数を少なくとも 4 桁に置き換えます。4 桁に満たない場合は右詰めで、空いた桁にはスペースが入ります。たとえば、`printf("%4d", 12)` の出力は、“ 12” (スペース・スペース・1・2) となります。

練習課題 16 (ex153.c)

9 行 9 列の 2 次元配列 `ta` を九九の表に見立てて値を代入した後、出力するプログラムを作ってください。1 列目と 1 行目 (下図の背景色が紫の部分) には 1~9 を代入し、残りの要素 (下図の背景色が白の部分) には 1 列目と 1 行目の値を使って計算した結果を代入してください^a。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
[0]	1	2	3	4	5	6	7	8	9
[1]	2	4	6	8	10	12	14	16	18
[2]	3	6	9	12	15	18	21	24	27
[3]	4	8	12	16	20	24	28	32	36
[4]	5	10	15	20	25	30	35	40	45
[5]	6	12	18	24	30	36	42	48	54
[6]	7	14	21	28	35	42	49	56	63
[7]	8	16	24	32	40	48	56	64	72
[8]	9	18	27	36	45	54	63	72	81

途中まで作ったプログラムを、授業のサイトに `ex153.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。

実行結果									
1	2	3	4	5	6	7	8	9	
2	4	6	8	10	12	14	16	18	
3	6	9	12	15	18	21	24	27	
4	8	12	16	20	24	28	32	36	
5	10	15	20	25	30	35	40	45	
6	12	18	24	30	36	42	48	54	
7	14	21	28	35	42	49	56	63	
8	16	24	32	40	48	56	64	72	
9	18	27	36	45	54	63	72	81	

^a 添字を使って計算することもできますが、この練習課題では1列目と1行目の値を使うことにします。

1次元配列同様、2次元配列もメモリ上の連続した領域が割り当てられます。図9.4のように、外側配列の添字が小さい要素順に、内側配列が並びます。

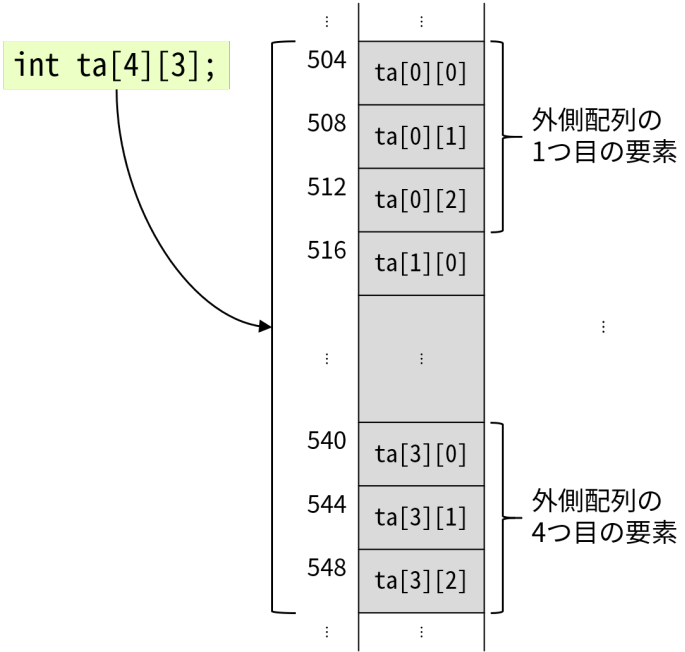


図 9.4 2次元配列の領域のイメージ

2次元配列 `ta` において、`ta[0]` は外側配列の1つ目の要素である1次元配列を、`ta[1]` は外側配列の2つ目の要素である1次元配列を表します (図9.5)。

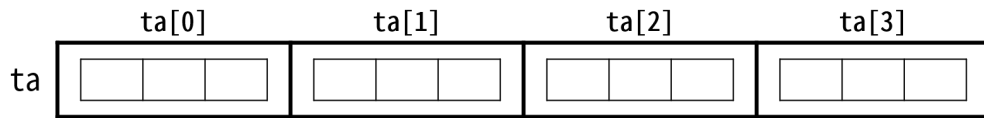


図 9.5 4 行 3 列の 2 次元配列 `ta`

`ta[x]` (x は添字) を評価すると `ta[x]` の先頭要素へのポインタが得られます^{*3}。つまり、`ta[x]` は `&ta[x][0]` と同義で、`*(ta[x] + 2)` は `ta[x][2]` の値です。これを、ソースコード 9.2 で確認します。

ソースコード 9.2 外側配列の要素の評価 (ex154.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int ta[4][3] = {
6          { 10, 20, 30 },
7          { 40, 50, 60 },
8          { 70, 80, 90 },
9          { 100, 110, 120 }
10     };
11
12     printf("ta[1] は &ta[1][0] と同義\n");
13     printf("  ta[1]: %p\n", ta[1]);
14     printf("  &ta[1][0]: %p\n\n", &ta[1][0]);
15
16     printf("*(ta[3] + 2) は ta[3][2] の値\n");
17     printf("  *(ta[3] + 2) = %d\n", *(ta[3] + 2));
18     printf("  ta[3][2] = %d\n", ta[3][2]);
19
20     return 0;
21 }
```

^{*3} 1 次元配列の変数名を評価すると、その配列の先頭要素へのポインタが得られることを思い出してください。たとえば、1 次元配列の配列名を `a` とすると、`a` は `&a[0]` と同義です。

Windows 11 上の MinGW-w64(gcc version 13.2.0) での実行結果例

```
ta[1] は&ta[1][0] と同義
  ta[1]: 00000018F33FFABC
  &ta[1][0]: 00000018F33FFABC

*(ta[3] + 2) は ta[3][2] の値
  *(ta[3] + 2) = 120
  ta[3][2] = 120
```

2次元配列の変数名 (これまでの例では `ta`) を評価するとポインタが得られますが、この授業では扱わないことにします^{*4}。

^{*4} `ta` を 4 行 3 列の配列とすると、`ta` の型は “要素数が 3 の `int` 型の配列” へのポインタとなります。

第 10 章

文字列

10.1 文字と文字列

文字列や文字列リテラルについて、簡単に復習します。文字列とは、0 文字以上の文字の並びです。プログラム中に値として文字を書くときは、1 文字を ' (シングルクォーテーション) で囲んだ**文字定数**を使います。文字列を書くときは、文字列を " (ダブルクォーテーション) で囲んだ**文字列リテラル**を使います。表 10.1 に、“C 言語基礎演習”の表 4.2 を再掲します。

表 10.1 文字定数と文字列リテラル (“C 言語基礎演習”の表 4.2)

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

一般的なプログラムでは、“文字”を扱うよりも“文字列”を扱う方が多いと思われます。ここまで文字列がほとんど出てこなかったことが不自然に感じられたかもしれません^{*1}。C 言語で文字列を扱うには、配列とポインタの理解が欠かせません。これらを学んだことで、やっと文字列に進むことができます。

10.2 文字列の仕組み

C 言語の文字列は、**文字列を構成する文字をメモリ上の連続した領域に並べ、最後に文字コードが 0 の文字を付けたものです**。ソースコード中に (たとえば `printf()` 関数の第 1 引数に) 文字列リテラルとして `"string"` と書いたときのメモリのイメージを図 10.1 に示します。メモリ上の連続する 7 バイトの領域に、先頭から順に `'s'`、`'t'`、`'r'`、`'i'`、`'n'`、`'g'`、0 が格納されます。

^{*1} ここまでに文字列リテラルを使った箇所の大半は、`printf()` 関数の第 1 引数と `scanf()` 関数の第 1 引数です。

⋮	⋮
502	
503	
504	's'
505	't'
506	'r'
507	'i'
508	'n'
509	'g'
510	0
511	
512	
⋮	⋮

図 10.1 メモリ上の文字列リテラルのイメージ

文字列を扱うときに、文字列中の文字は文字定数で、末尾の 0 は数値で書くと統一感がありません。そこで、0 の代わりに '\0' という表記を用います。'\0' は、値 (文字コード) が 0 の文字を表します。'\0' は、**ヌル文字** (ナル文字) と呼ばれます*2。

文字列リテラルを評価すると、先頭文字へのポインタが得られます。このことを、ソースコード 10.1 で確認します。

ソースコード 10.1 文字列リテラルとポインタ (ex155.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      char *cp = "string";
6      int i;
7
8      printf("cp = %p\n\n", cp);
9
10     for (i = 0; i < 6; ++i)
11     {
12         printf("(cp + %d) = '%c'\n", i, *(cp + i));
13     }
14
15     return 0;
16 }
```

*2 図 2.6 の、ビット列 00000000 の文字 NUL が該当します。

Windows 11 上の MinGW-w64(gcc version 13.2.0) での実行結果例

```
cp = 00007FF6BA354000
```

```
*(cp + 0) = 's'  
*(cp + 1) = 't'  
*(cp + 2) = 'r'  
*(cp + 3) = 'i'  
*(cp + 4) = 'n'  
*(cp + 5) = 'g'
```

5 行目でポインタ変数 `cp` を、文字列リテラル `"string"` を評価した値、つまり先頭文字である `'s'` へのポインタで初期化します。8 行目に表示するアドレスは、`'s'` が格納されたメモリのアドレスです (図 10.1 では 504 番地に相当します)。10 行目から始まる `for` 文で、`cp` を通して、文字列リテラル中の文字を 1 文字ずつ表示します。配列の要素を指すポインタ同様、`cp + i` は `cp` から `i` 文字だけ後ろの文字 (`sizeof(char) × i` だけ先のアドレスに格納された `char` 型の値) を指すポインタになります。

ソースコード 10.1 では、`"string"` の文字数が 6 文字であることがわかっていたので、`*(cp + 0)` から `*(cp + 5)` までの 6 文字を表示しました。ヌル文字に到達したかを判定すれば、ソースコード中に文字数を書く必要はありません。そのように修正したものを、ソースコード 10.2 に示します。

ソースコード 10.2 ヌル文字の判定による文字列の走査 (ex156.c)

```
1  #include <stdio.h>  
2  
3  int main(void)  
4  {  
5      char *cp = "string";  
6  
7      while (*cp != '\0')  
8      {  
9          printf("%c\n", *cp++);  
10     }  
11  
12     return 0;  
13 }
```

実行結果

's'
't'
'r'
'i'
'n'
'g'

cp がヌル文字を指すようになるまで、9 行目の printf() 関数を使った文を繰り返し実行します。この文は cp が指す文字を表示した上で、cp を、現在指している文字の 1 文字後ろの文字を指すように更新します。この動作が難しく感じるようであれば、ソースコード 8.3 の解説を復習してください。

第 2 章の ex113.c では、sizeof "A" が 2、sizeof "ABC" が 4 になることを確認しました。sizeof "A" は 'A' とヌル文字で 2、sizeof "ABC" は 'A'、'B'、'C'、ヌル文字で 4 になります。0 文字の文字列 "" にもヌル文字がありますので、sizeof "" は 1 になります (図 10.2)。

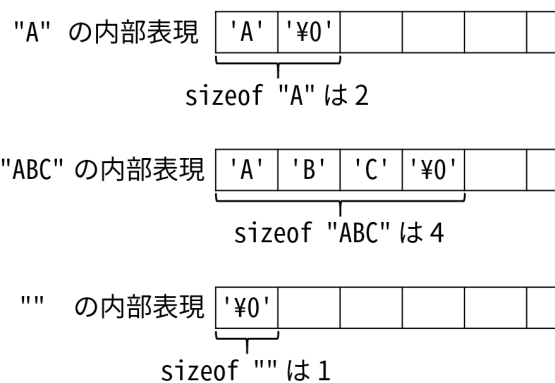


図 10.2 文字列の大きさ

文字列リテラルがメモリ上に配置されているなら、次のようにすれば文字を書き換えられそうですが、このような操作は許容されていません。

ソースコード 10.3 文字列中の文字の書き換え (誤り)

```
1 char *cp = "string";
2
3 /* cpの2文字目を'p'に書き換え、"spring"に変更 */
4 *(cp + 1) = 'p';
```

文字列リテラル中の文字を書き換えてはいけません^{*3}。そのため、文字列リテラルを指すポインタ変数は、次のように型修飾子 `const` を付けることが一般的です。

ソースコード 10.4 文字列リテラルを指すポインタ変数の宣言

```
const char *cp = "string";
```

これまで、`printf()` 関数と `scanf()` 関数の第 1 引数の型を説明していませんでした。いずれの型も `const char *` です^{*4}。つまり、文字列 (の先頭文字) へのポインタを受け取ります。また、そのポインタを通して文字列を書き換えることはありません。

表 10.2 に示すライブラリ関数 `strlen()` は、文字列の長さを返します。

表 10.2 `strlen()` 関数

書式	<code>strlen(s)</code>
引数	文字列 <code>s</code> (<code>const char *</code> 型)
戻り値	<code>s</code> の長さ (<code>size_t</code> 型)
必要ヘッダ	<code><string.h></code>

ここでの文字列の長さとは、ヌル文字を除いた文字の個数です^{*5}。`strlen()` の動作を確認するプログラムを、ソースコード 10.5 に示します。

ソースコード 10.5 ライブラリ関数 `strlen()` (`ex157.c`)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      const char *cp;
7
8      cp = "ABC";
9      printf("\n%s\ の長さは %d\n", cp, (int)strlen(cp));
10
11     cp = "Yamada Taro";
12     printf("\n%s\ の長さは %d\n", cp, (int)strlen(cp));
13
```

^{*3} メモリには、プログラムで書き換えられる領域と書き換えられない領域があります。一般的に、文字列リテラルは書き換えられない領域に配置されます。もちろん、変数は書き換えられる領域に配置されます。

^{*4} 処理系によっては、`const` 以外の型修飾子が付いていることもあります。

^{*5} 1 文字が 2 バイト以上のマルチバイト文字を使った場合は文字の個数ではありません。この授業では、マルチバイト文字を含む文字列の操作は扱わないことにします。

```

14     cp = "";
15     printf("\'%s\' の長さは %d\n", cp, (int)strlen(cp));
16
17     return 0;
18 }

```

実行結果

```

"ABC" の長さは 3
"Yamada Taro" の長さは 11
"" の長さは 0

```

9、12、15 行目の `printf()` 関数の第 1 引数では、文字列に置き換えるための変換指定 `%s` を使っています (表 10.3)。`%s` に対応する引数には、文字列へのポインタを指定します。文字列リテラルで `"` を表すエスケープシーケンス `\` を使っていることにも注意してください。

表 10.3 `printf()` 関数の第 1 引数で使う変換指定 (“C 言語基礎演習” の表 5.2 に `%s` を追加)

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

ソースコード 10.5 の 3 つの `printf()` 関数は、それぞれ文字列 `"ABC"`、`"Yamada Taro"`、`"` の長さを表示します。変数 `cp` を使わずに、

```
printf("\'ABC\' の長さは %d\n", (int)strlen("ABC"));
```

や

```
printf("\'%s\' の長さは %d\n", "ABC", (int)strlen("ABC"));
```

でも動作します。

なお、このプログラムでは、`cp` が指す文字列を途中で変えています。型修飾子 `const` を付けたポインタ変数は、その変数を通してオブジェクトの値を書き換えることができません。しかし、変数が指すオブジェクトは、プログラムの途中で変えることができます。

練習課題 17 (ex158.c)

ライブラリ関数の `strlen()` と同じ働きをする関数 `mstrlen()` を作ってください。ただし、戻り値の型は `size_t` 型ではなく `int` 型にしてください。
途中まで作ったプログラムを、授業のサイトに `ex158.c` として掲載しています。コメント部分に `mstrlen()` 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。

実行結果

```
"ABC" の長さは 3
"Yamada Taro" の長さは 11
"" の長さは 0
```

ここまで見てきたように、C 言語では、ポインタが指す文字からヌル文字の直前の文字までが文字列です。次のプログラムの実行結果を考えてみてください。

ソースコード 10.6 文字を指すポインタと文字列 (ex159.c)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      const char *cp = "string";
7
8      printf("%c\n", *(cp + 2));
9      printf("%s\n", cp + 2);
10     printf("%d\n", (int)strlen(cp + 2));
11
12     return 0;
13 }
```

⋮
⋮
⋮

正解は、次のとおりです。

実行結果

```
r
ring
4
```

`cp + 2` は、"string" の 3 番目の文字 'r' を指すポインタです。8 行目の `printf()` 関数は、ポインタが指す文字である `r` を表示します。9 行目の `printf()` 関数は、ポインタが指す文字から始まる文字列である `ring` を表示します。その長さである 4 を、10 行目の `printf()` 関数で表示します。

10.3 char 型の配列への文字列の格納

文字列リテラルによってメモリ上に配置される文字列を書き換えてはいけないことを学びました。文字列を書き換えたい場合は、文字列を **char 型の配列に格納** します。char 型の配列に格納した文字列を書き換える例を、ソースコード 10.7 に示します。

ソースコード 10.7 char 型の配列に格納した文字列の操作 (ex160.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char str[] = { 's', 't', 'r', 'i', 'n', 'g', '\0' };
6
7      printf("%s\n", str);
8
9      str[1] = 'p';
10     printf("%s\n", str);
11
12     str[4] = '\0';
13     printf("%s\n", &str[1]);
14
15     return 0;
16 }
```

5 行目では、char 型の配列 `str` を宣言しています。要素数は 7 で、初期値は、's'、't'、'r'、'i'、'n'、'g'、'\0' です。

配列にはメモリ上の連続した領域が割り当てられるため、`str` は文字列 "string" を表しています (図 10.3 の①)。7 行目の `printf()` 関数では、`str(str の先頭要素へのポインタ)` が指す文字か

ら始まる文字列、つまり "string" を表示します。

9 行目で、`str` の 2 番目の要素 (`str[1]`) に、文字 'p' を代入します。これによって、`str` が表す文字列が "string" から "spring" に書き換えられました (図 10.3 の②)。

12 行目で、`str` の 5 番目の要素 (`str[4]`) に、ヌル文字 '\0' を代入します。13 行目の `printf()` 関数では、`str[1]` から始まる文字列である "pri" を表示します。先頭文字からヌル文字の直前までが文字列であることが確認できます (図 10.3 の③)。`printf()` 関数の変換指定 `%s` に対応する引数は、文字列 (の先頭文字) へのポインタです。`str[1]` に単項演算子 `&` を付けていることにも注意してください。

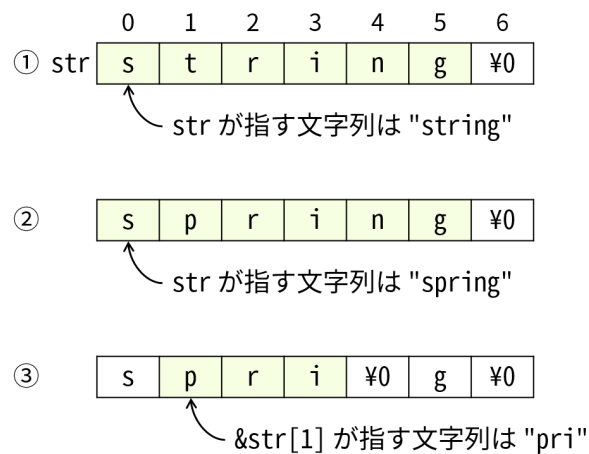


図 10.3 ソースコード 10.7 の `str`^{*6}

ソースコード 10.7 では、要素の値を 1 つずつ指定して、`str` を初期化しました。`char` 型の配列を文字列で初期化するときは、

```
char str[] = { 's', 't', 'r', 'i', 'n', 'g', '\0' };
```

の代わりに、文字列リテラルのような形式で、

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

と書けます。後者の書き方では、'g' の後ろにヌル文字も格納されます。後者の書き方は文字列リテラルを代入しているわけではなく、初期化のときだけに使える特別な書き方です。

ソースコード 10.7 では、文字列の途中にヌル文字を入れて、文字列の長さを短くしました。ヌル文字を通常の文字に書き換え、それより後ろにヌル文字を追加すれば、文字列の長さを長くすることができます。文字列の長さを長くする例を、ソースコード 10.8 に示します。

^{*6} これ以降は、図中の文字を ' で囲まないことにします。

ソースコード 10.8 文字列の操作例 (ex161.c)

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      char str[10];
6
7      str[0] = 'a';
8      str[1] = 'b';
9      str[2] = 'c';
10     str[3] = '\0';
11
12     printf("%s\n", str);
13
14     str[3] = 'd';
15     str[4] = 'e';
16     str[5] = 'f';
17     str[6] = '\0';
18
19     printf("%s\n", str);
20
21     return 0;
22 }
```

実行結果

```

abc
abcdef
```

char 型の配列 `str` を宣言し、7 行目から 10 行目で文字列 "abc" を格納します (図 10.4 の①)。`str[3]` にヌル文字を代入していることに注目してください。12 行目の `printf()` 関数で、この文字列を表示します。

14 行目から 17 行目で、`str` が表す文字列を "abc" から "abcdef" に変更します (図 10.4 の②)。`str[3]` のヌル文字を 'd' に更新し、新たに `str[6]` にヌル文字を代入しています。19 行目の `printf()` 関数で、変更後の文字列 "abcdef" を表示します。

このように、char 型の配列に格納された文字列は、通常の配列と同じように操作できます。

	0	1	2	3	4	5	6	7	8	9
① str	a	b	c	¥0	?	?	?	?	?	?
②	a	b	c	d	e	f	¥0	?	?	?

図 10.4 ソースコード 10.8 の `str`

ソースコード 10.8 では、`str` を要素数 10 の配列として宣言しましたが、たとえば

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

と宣言すると、`str` の要素数は 4 になります。この場合、文字列の長さを長くすることができません。ですから、文字列の操作を行う場合は、十分な要素数を指定して宣言する必要があります。

配列を宣言時に初期化する場合、値を指定しない要素は 0 になります*⁷。つまり、

```
char str[10] = "abc";
```

と宣言すると、`str` の要素数は 10 で、初期値は先頭から順に 'a'、'b'、'c'、'\0'(0)、0、0、0、0、0、0 になります。

ソースコード 10.9 は“C 言語基礎演習”で使ったプログラムで、名字の頭文字と年齢を入力してもらい、それらを表示します。

ソースコード 10.9 名字の頭文字と年齢の入力と表示 (“C 言語基礎演習”の 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;

```

*⁷ 初期化しない場合、要素の値は不定です。

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。

これを、名字の頭文字ではなく、名字を入力するように改変します。

文字列を読み取って char 型の配列に格納するための scanf() 関数の変換指定は %s です (表 10.4)。

表 10.4 scanf() 関数の第 1 引数で使う変換指定 (表 2.3 に %s を追加)

変換指定	役割
%c	読み取った文字を char 型の変数に格納する。
%d	読み取った整数を int 型の変数に格納する。
%u	読み取った非負整数を unsigned int 型の変数に格納する。
%lf (エル・エフ)	読み取った小数を double 型の変数に格納する。
%s	読み取った文字列を char 型の配列に格納する。

変換指定 %s を使う場合、第 2 引数には配列変数名 (文字列の格納先を表すポインタ) を指定します。

名字を入力するように改変したプログラムを、ソースコード 10.10 に示します。

ソースコード 10.10 scanf() 関数による文字列の入力 (ex162.c)

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char name[20];
6      int age;
7
8      printf("名字を入力してください > ");
9      scanf("%s", name);
10     printf("年齢を入力してください > ");
11     scanf("%d", &age);
12
13     printf("%sさんの年齢は%d歳です。\\n", name, age);
```

```
14
15     return 0;
16 }
```

実行例

名字を入力してください > Morimoto ↵

年齢を入力してください > 45 ↵

Morimoto さんの年齢は 45 歳です。

変換指定 %s を使うと、キーボードから入力した文字に加えて、ヌル文字も格納されます。第 2 引数には、ヌル文字を含めて文字列が格納できる大きさの配列を指定する必要があります。ソースコード 10.10 では、name を要素数 20 の配列として宣言していますので、名字として 19 文字まで入力できます。

不特定のユーザが使うプログラムでは、どのような長さの文字列が入力されるかわかりませんので、このような scanf() 関数の使い方は適切ではありません。また、変換指定 %s では、文字列の途中に (ソースコード 10.10 では名字の途中に) スペースが入っていると、その直前の文字までしか読み取りません。実用的なプログラムではこれらへの対応が必要です。これ以降のプログラムも、簡略化のため、ユーザが想定外の入力をしないことを前提にしているものがあります。

10.4 文字列の操作

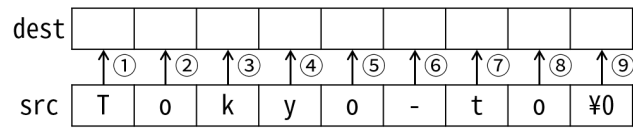
ここでは、文字列を扱うプログラムを作って、文字列の理解を深めます。

文字列のコピー

通常の配列同様、char 型の配列に演算子 = を使って文字列を代入することはできません。

```
1 char s1[] = "string";
2 char s2[7];
3
4 s2 = s1; /* エラー。配列に配列の代入は不可。 */
```

そこで、char 型の配列に文字列を格納する関数 str_copy() を作ります。str_copy() は、第 2 引数 src に指定される文字列を、第 1 引数 dest に指定されるポインタが指す char 型の配列にコピーします。図 10.5 のように、src の文字を 1 文字ずつ、dest が指す配列に代入することになります。ヌル文字も代入することを忘れないでください。



①～⑨まで1文字ずつコピー

図 10.5 `str_copy()` による文字列のコピー

練習課題 18 (ex163.c)

ex163.c は、`str_copy()` を使って、キーボードから読み取った文字列を、別の配列にコピーして表示するプログラムです。コメント部分に処理を記述して、`str_copy()` を完成させてください。

プログラムを実行する際は、`s1` としてスペースを含まない文字列を指定してください。

文字列 `s1` > Tokyo-to ↵

`s1` を `s2` にコピーしました。

`s2` に格納されている文字列は Tokyo-to です。

文字列からの文字の探索

次の仕様を持つ関数 `locate_char()` を作ります。

- 第1引数 `s` に指定された文字列の中から、第2引数 `c` に指定された文字を探します。
- `s` の中に `c` が見つければ、その位置を返します。位置は、`s` の先頭を 0 とします。
- `s` の中に `c` が複数ある場合は、一番先頭に近い `c` の位置を返します。
- `s` の中に `c` が見つからなければ、-1 を返します。

`s` が "Tokyo-to" の場合の戻り値は、`c` が 't' なら 6、'o' なら 1、'i' なら -1 です (図 10.6)。

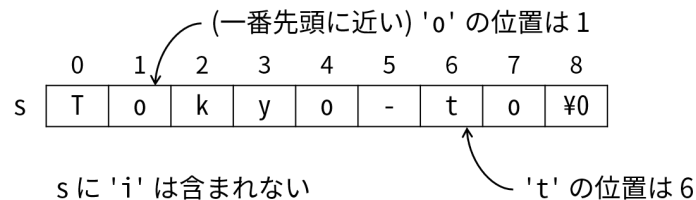


図 10.6 "Tokyo-to" の中の文字

練習課題 19 (ex164.c)

ex164.c は、キーボードから読み取った文字列 `s` と文字 `c` を引数として、`locate_char()` を呼び出すプログラムです。コメント部分に `locate_char()` の定義を記述してください。
ex164.c の 11 行目で、`scanf()` 関数の第 1 引数が `"%s%c"` となっているのは、3.3 節の脚注*4 での説明と同じ理由からです。`s` には、スペースを含まない文字列を指定してください。

実行例 1

文字列 `s` > Tokyo-to ↵

文字 `c` > o ↵

Tokyo-to の中の o の位置は 1 です。

実行例 2

文字列 `s` > Tokyo-to ↵

文字 `c` > i ↵

Tokyo-to の中に i は含まれません。

文字列の比較

2 つの文字列が等しいかを、演算子 `==` を使って判定することはできません。

```
1 char s[] = "string";
2
3 /*
4  * 以下の制御式は誤り。
5  * sの先頭要素のアドレスが、文字列リテラル"string"の先頭文字が格納
6  * されたアドレスと等しいかを判定しており、常に成り立たない。
7  */
8 if (s == "string")
9 {
10     printf("s に格納されている文字列は \"string\" です。\\n");
11 }
```

そこで、2 つの文字列が等しいかを判定する関数 `str_comp()` を作ります。`str_comp()` は、第 1 引数 `s1` に指定される文字列が、第 2 引数 `s2` に指定される文字列と等しければ 0 を、等しくなければ 1 を返します。次の方法で、2 つの文字列が等しいかを判定することにします。

先頭から順に、2つの文字列の同じ位置にある文字同士を比較します。

- 文字が等しくなければ、2つの文字列は等しくないと判定します。
- 文字が等しく、かつそれがヌル文字であれば、2つの文字列は等しいと判定します。
- 文字が等しく、かつそれがヌル文字でなければ、次の文字の比較に進みます。

この方法で比較する様子を、図 10.7 に示します。

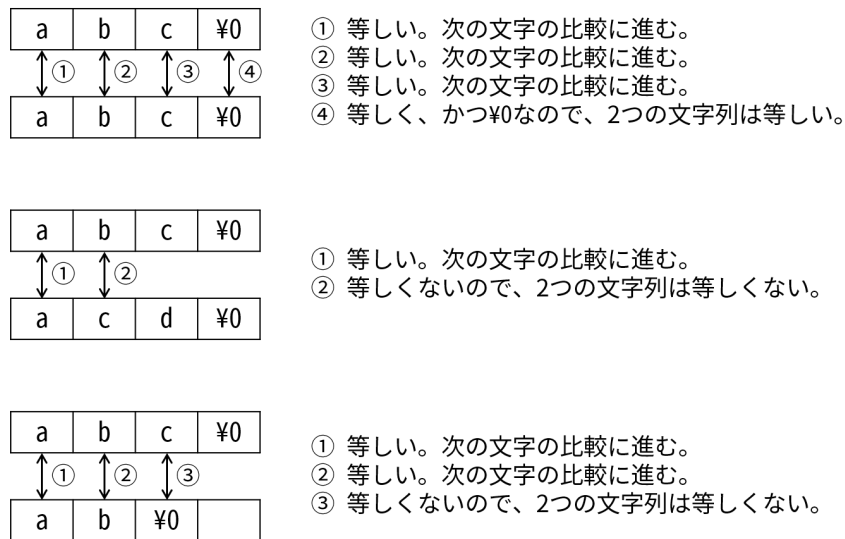


図 10.7 2つの文字列の比較

図 10.7 の 1 つ目と 3 つ目の例のように、ヌル文字も普通の文字のように比較します。

練習課題 20 (ex165.c)

ex165.c は、キーボードから読み取った文字列 s1 と s2 を引数として、`str_comp()` を呼び出すプログラムです。コメント部分に `str_comp()` の定義を記述してください。

実行例 1

文字列 s1 > abc ↵

文字列 s2 > abc ↵

abc と abc は等しいです。

実行例 2

文字列 s1 > abc ↵

文字列 s2 > acd ↵

abc と acd は等しくありません。

実行例 3

文字列 s1 > abc ↵

文字列 s2 > ab ↵

abc と ab は等しくありません。

10.5 文字列に関するライブラリ関数

練習課題 18～20 では、それぞれ `str_copy()`、`locate_char()`、`str_comp()` を実装しました。これらと同じような役割を持つ関数が、それぞれライブラリ関数 `strcpy()`、`strchr()`、`strcmp()` として用意されています。

`strcpy()` 関数の仕様を表 10.5 に、`strcpy()` を使って練習課題 18 と同じ動作を行うプログラムをソースコード 10.11 に示します。

表 10.5 strcpy() 関数

書式	<code>strcpy(dest, src)</code>
引数	コピー先の配列 <i>dest</i> (<code>char *</code> 型) と文字列 <i>src</i> (<code>const char *</code> 型)
処理	<i>src</i> が指す文字列を、ヌル文字を含め、 <i>dest</i> が指す配列にコピーする。
戻り値	<i>dest</i> の値 (<code>char *</code> 型)
必要ヘッダ	<code><string.h></code>

ソースコード 10.11 ライブラリ関数 `strcpy()` の使用 (ex166.c)

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(void)
5 {
6     char s1[100], s2[100];
```

```

7
8     printf("文字列s1 > ");
9     scanf("%s", s1);
10
11     strcpy(s2, s1);
12
13     printf("s1をs2にコピーしました。 \ns2に格納されている文字列は %s です。 \n",
14           s2);
15
16     return 0;
17 }

```

strcpy() は第 1 引数に指定したポインタをそのまま返します。ソースコード 10.11 では、この戻り値を使っていません。

strchr() の仕様を表 10.6 に、strchr() を使って練習課題 19 と同じ動作を行うプログラムをソースコード 10.12 に示します。

表 10.6 strchr() 関数

書式	strchr(s, c)
引数	文字列 s(const char *型) と文字 c(int 型)
戻り値	s に c が含まれていれば、s の中で一番先頭に近い c へのポインタ(char *型)。含まれていなければ NULL。
必要ヘッダ	<string.h>

ソースコード 10.12 ライブラリ関数 strchr() の使用 (ex167.c)

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      char s[100], c;
7      char *ret;
8
9      printf("文字列s > ");
10     scanf("%s%c", s);

```

```

11     printf("文字c > ");
12     scanf("%c", &c);
13
14     ret = strchr(s, c);
15
16     if (ret == NULL)
17     {
18         printf("%s の中に %c は含まれません。\\n", s, c);
19     }
20     else
21     {
22         printf("%s の中の %c の位置は %d です。\\n", s, c, ret - s);
23     }
24
25     return 0;
26 }

```

表 10.6 の“戻り値”に登場した `NULL` は、マクロとして定義されている、値が 0 の定数です。`NULL` は、ポインタが何も指していないことを表すために使われます。ソースコード 10.12 の 16 行目の `if` 文の制御式では、`strchr()` の戻り値 `ret` が `NULL` であるかを判定しています。

ソースコード 10.12 の 22 行目では、文字列 `s` の中の文字 `c` の位置を、`ret - s` で求めています。第 8 章では、配列の要素を指すポインタに整数 `a` を足すと、`a` 個後ろの要素を指すポインタになることを学びました。後者のポインタから前者のポインタを引くと、`a` が求まります。`ret - s` により、`c` の位置、つまり `c` が `s` の先頭の文字から何文字後ろであるかが計算できます。

`strcmp()` の仕様を表 10.7 に、`strcmp()` を使って練習課題 20 と同じ動作を行うプログラムをソースコード 10.13 に示します。

表 10.7 `strcmp()` 関数

書式	<code>strcmp(s1, s2)</code>
引数	文字列 <code>s1</code> と <code>s2</code> (いずれも <code>const char *</code> 型)
戻り値	<code>s1</code> が <code>s2</code> より大きければ 0 より大きい整数、等しければ 0、小さければ 0 より小さい整数 (<code>int</code> 型)。文字列の大小は、文字コードを基準にした辞書順。
必要ヘッダ	<code><string.h></code>

ソースコード 10.13 ライブラリ関数 `strcmp()` の使用 (ex168.c)

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      char s1[100], s2[100];
7
8      printf("文字列s1 > ");
9      scanf("%s", s1);
10     printf("文字列s2 > ");
11     scanf("%s", s2);
12
13     printf("%s と %s は%s。 \n", s1, s2, strcmp(s1, s2) == 0 ? "等しいです" :
14           "等しくありません");
15
16     return 0;
17 }
```

`<string.h>`は、ここで紹介した3つの関数や `strlen()` など、文字列に関するライブラリ関数を提供しています。

10.6 文字列の配列

`char` 型の2次元配列を用いれば、**文字列の配列**が実現できます。2次元配列の1行に1つの文字列を格納します。文字列の配列を操作する例を、ソースコード 10.14 に示します。

ソースコード 10.14 `char` 型の2次元配列を用いた文字列の配列 (ex169.c)

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      /* 3人分の名前(9文字以下)を格納する2次元配列 */
7      char name[3][10];
8
9      int i;
10 }
```

```

11     /* 1人目は "Morimoto" */
12     strcpy(name[0], "Morimoto");
13
14     /* 2人目と3人目はキーボードから入力 */
15     printf("2人目 > ");
16     scanf("%s", name[1]);
17     printf("3人目 > ");
18     scanf("%s", name[2]);
19
20     printf("\n");
21
22     /* 3人の名前を表示 */
23     for (i = 0; i < 3; ++i)
24     {
25         printf("%d人目: %s\n", i + 1, name[i]);
26     }
27
28     printf("\n");
29
30     /* 1人目の名前を "Morimoto" から "Mori" に変更して表示 */
31     name[0][4] = '\0';
32     printf("%s\n", name[0]);
33
34     return 0;
35 }

```

実行例

```

2人目 > Yamada ↵
3人目 > Tanaka ↵

```

```

1人目: Morimoto
2人目: Yamada
3人目: Tanaka

```

```

Mori

```

12 行目では、`name[0]` が指す配列に文字列 "Morimoto" をコピーしています。`name[0]` は、“`name` の最初の要素である `char` 型の 1 次元配列” の先頭要素へのポインタであることに注意してください。31 行目では、`name[0][4]` にヌル文字を代入することで、`name[0]` に格納した文字列を "Morimoto" から "Mori" に変更しています (図 10.8)。

char型の2次元配列name

	0	1	2	3	4	5	6	7	8	9
0	M	o	r	i	m	o	t	o	¥0	?
1	Y	a	m	a	d	a	¥0	?	?	?
2	T	a	n	a	k	a	¥0	?	?	?

↓ name[0][4]に '¥0' を代入

	0	1	2	3	4	5	6	7	8	9
0	M	o	r	i	¥0	o	t	o	¥0	?
1	Y	a	m	a	d	a	¥0	?	?	?
2	T	a	n	a	k	a	¥0	?	?	?

図 10.8 ソースコード 10.14 の 31 行目の操作

第 11 章

構造体

人の名前、年齢、身長を扱うことを考えます。名前は最大 9 文字として、要素数 10 の `char` 型の配列に格納します。年齢は `int` 型の変数に、身長は `double` 型の変数に格納します。次のソースコードでは、Yamada さんと Tanaka さんの 2 人について、データを格納する変数を宣言し、値を代入しています。

```
1 char name_yamada[10], name_tanaka[10];
2 int age_yamada, age_tanaka;
3 double height_yamada, height_tanaka;
4
5 /* Yamadaさんのデータ */
6 strcpy(name_yamada, "Yamada");
7 age_yamada = 35;
8 height_yamada = 170.5;
9
10 /* Tanakaさんのデータ */
11 strcpy(name_tanaka, "Tanaka");
12 age_tanaka = 51;
13 height_tanaka = 158.5;
```

`name_yamada` と `age_yamada` と `height_yamada` が Yamada さんのデータ、`name_tanaka` と `age_tanaka` と `height_tanaka` が Tanaka さんのデータです。3 つの変数の組で 1 人分のデータを表していますが、今のままでは、3 つの変数が同じ人のデータであることが変数名からしかわかりません。**構造体**を使えば、1 人分の名前、年齢、身長をまとめることができます。

構造体とは、複数種類のデータをまとめて格納できる型です。構造体に含まれるデータ (変数) を**メンバ**と呼びます。構造体を使うためには、まず、**構造体宣言**をして型を作る必要があります。構造体宣言の書式と例を、図 11.1 に示します。

書式の中の“**タグ名**”は、構造体の名前です。“メンバ宣言”の部分には、これまでの変数宣言

【構造体宣言の書式】

```
struct タグ名 {
    メンバ宣言1
    メンバ宣言2
    :
};
```

【構造体宣言の例】

```
struct person {
    char name[10];
    int age;
    double height;
};
```

図 11.1 構造体宣言の書式と例

のように、メンバの型と名前を記述します。メンバ宣言の末尾と構造体宣言の末尾には、; (セミコロン) が必要です。図 11.1 の右は、要素数 10 の char 型の配列 `name`、int 型の `age`、double 型の `height`、の 3 つのメンバを持つ、タグ名が `person` である構造体の宣言です。この授業では、構造体の宣言は関数の外側、つまりグローバル変数や関数プロトタイプ宣言と同じ場所を書くことにします。

構造体宣言は、構造体の型を作るだけです。構造体に値を入れて使うには、通常の変数と同じように、型を指定して変数を宣言します。構造体の型名は“`struct タグ名`”です。たとえば、図 11.1 の右で宣言した構造体の型名は、“`struct person`”です。構造体の型から変数を作って値を格納するイメージを、図 11.2 に示します。

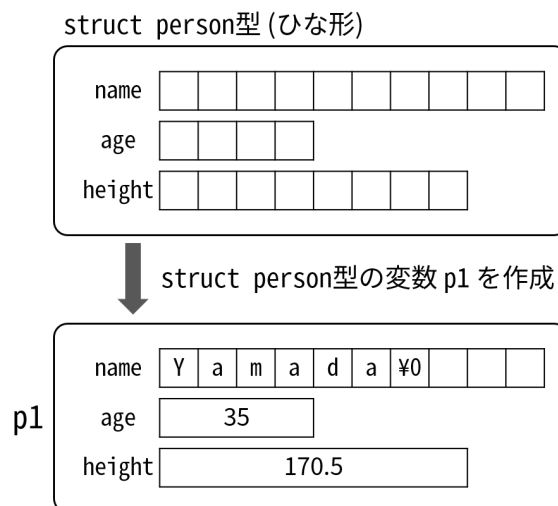


図 11.2 構造体の型と変数

構造体のメンバは、演算子 `.` を使って、“**構造体変数名.メンバ名**”と表します (表 11.1)。

表 11.1 演算子 `.`

演算子	役割
<code>a.b</code>	構造体 <code>a</code> のメンバ <code>b</code>

構造体を使う例を、ソースコード 11.1 に示します。

ソースコード 11.1 構造体の使用例 (ex170.c)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct person {
5      char name[10];
6      int age;
7      double height;
8  };
9
10 int main(void)
11 {
12     struct person p1, p2;
13
14     /* p1はYamadaさんのデータ */
15     strcpy(p1.name, "Yamada");
16     p1.age = 35;
17     p1.height = 170.5;
18
19     /* p2のデータはキーボードから入力 */
20     printf("名前 > ");
21     scanf("%s", p2.name);
22     printf("年齢 > ");
23     scanf("%d", &p2.age);
24     printf("身長 > ");
25     scanf("%lf", &p2.height);
26
27     printf("\n");
28     printf("%sさんの年齢は%d歳、身長は%.1fcmです。 \n", p1.name, p1.age, p1.height);
29     printf("%sさんの年齢は%d歳、身長は%.1fcmです。 \n", p2.name, p2.age, p2.height);
30
31     return 0;
32 }
```

実行例

名前 > Tanaka ↵

年齢 > 51 ↵

身長 > 158.5 ↵

Yamada さんの年齢は 35 歳、身長は 170.5cm です。

Tanaka さんの年齢は 51 歳、身長は 158.5cm です。

4 行目から 8 行目が、タグ名が `person` である構造体の宣言です。12 行目で `struct person` 型の変数 `p1` と `p2` を宣言した後、それぞれの変数のメンバに値を格納して表示しています。15 行目の `p1.name` は、`p1` のメンバ `name`(`char` 型の配列) の先頭要素へのポインタです*¹。23 行目の `&p2.age` は、`&` より `.` の方が優先順位が高いため、`&(p2.age)` と解釈されます。この式を評価すると、`p2` のメンバ `age` へのポインタが得られます。

構造体の型名は“`struct タグ名`”ですが、`typedef` 宣言をして `struct` を含まない型名にすることも一般的です。たとえば次のようにして、`struct person` の代わりに `Person` を使います。

```
1 struct person {
2     char name[10];
3     int age;
4     double height;
5 };
6 typedef struct person Person;
```

構造体宣言と `typedef` 宣言は、次のようにまとめて書くことができます。

```
1 typedef struct person {
2     char name[10];
3     int age;
4     double height;
5 } Person;
```

その場合は、次のように構造体のタグ名 (`person`) を省略できます。

```
1 typedef struct {
2     char name[10];
3     int age;
4     double height;
```

*¹ 構造体のメンバも、通常の変数同様、メモリ上に領域が確保されます。

```
5 } Person;
```

構造体の配列を作ることができます。ソースコード 11.1 を、構造体を `typedef` 宣言し、構造体の配列を使うように改変したプログラムを、ソースコード 11.2 に示します。

ソースコード 11.2 ソースコード 11.1 の改変 (ex171.c)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  typedef struct {
5      char name[10];
6      int age;
7      double height;
8  } Person;
9
10 int main(void)
11 {
12     Person p[2];
13
14     /* p[0]はYamadaさんのデータ */
15     strcpy(p[0].name, "Yamada");
16     p[0].age = 35;
17     p[0].height = 170.5;
18
19     /* p[1]のデータはキーボードから入力 */
20     printf("名前 > ");
21     scanf("%s", p[1].name);
22     printf("年齢 > ");
23     scanf("%d", &p[1].age);
24     printf("身長 > ");
25     scanf("%lf", &p[1].height);
26
27     printf("\n");
28     printf("%sさんの年齢は%d歳、身長は%.1fcmです。 \n", p[0].name, p[0].age, p[0].height);
29     printf("%sさんの年齢は%d歳、身長は%.1fcmです。 \n", p[1].name, p[1].age, p[1].height);
30
```

```

31     return 0;
32 }

```

ソースコード 11.2 では、構造体の別名として `Person` を定義し、`Person` 型の配列 `p` を作っています。

構造体はしばしばポインタを介して扱われます。たとえば、ポインタ `a` が構造体を指しているとき、そのメンバ `b` は `(*a).b` と表せます*²。これを簡潔に記述できる演算子 `->` が用意されています (表 11.2)。

表 11.2 演算子 `->`

演算子	役割
<code>a -> b</code>	<code>a</code> (構造体へのポインタ) が指す構造体のメンバ <code>b</code>

構造体変数には、構造体を代入することができます。しかし、構造体同士の比較をすることはできません。

```

1  Person p1, p2;
2
3  p2 = p1; /* 代入は可能 */
4
5  /* 比較は不可 (コンパイルエラー) */
6  if (p1 == p2)
7  {
8      /* 略 */
9  }

```

配列は、別の配列に代入できませんでした。

```

1  char ca1[10] = "Morimoto", ca2[10];
2
3  ca2 = ca1; /* 代入は不可 (コンパイルエラー) */

```

しかし、構造体がメンバとして配列を持つ場合は、構造体の代入によって、メンバの配列の中身もコピーできます。

```

1  Person p1, p2; /* Personはメンバとして配列nameを持つ */
2
3  p2 = p1; /* p2.nameにp1.nameがコピーされる */

```

*² `*(単項)` より `.` の方が優先順位が高いため、優先順位を変える `( )` が必要です。`*a.b` は `*(a.b)` と解釈されます。

構造体を初期化するには、次のように、メンバの宣言順に初期値を , (コンマ) 区切りで並べ、全体を { } で囲みます。

```
1  typedef struct {
2      char name[10];
3      int age;
4      double height;
5  } Person;
6
7  /* メンバname、age、heightの初期値は、それぞれ"Yamada"、35、170.5 */
8  Person p = { "Yamada", 35, 170.5 };
9
10 /* Personの配列の初期化 */
11 Person pa[] = {
12     { "Yamada", 35, 170.5 },
13     { "Tanaka", 51, 158.5 }
14 };
```

練習課題 21 (ex172.c)

次のメンバを、この順に持つ構造体を作ってください。

- 要素数 10 の char 型の配列 name (名前を格納)
- int 型の height (cm 単位の身長を格納)
- int 型の weight (kg 単位の体重を格納)

typedef 宣言をして、この構造体に PDATA という名前を付けてください。

PDATA を使うプログラムを、授業のサイトに ex172.c として掲載しています。コメント部分に PDATA の宣言を追加して、正しく動作するプログラムを作ってください。

実行結果

Takahashi さんの身長は 166cm、体重は 54kg です。

練習課題 22 (ex173.c)

練習課題 21 で使った PDATA の配列 `pd` に格納された 5 人分のデータから、最も身長が高い人の名前と身長を表示するプログラムを作ってください。

途中まで作ったプログラムを、授業のサイトに `ex173.c` として掲載しています。コメント部分に変数宣言や処理を追加して、正しく動作するプログラムを作ってください。配列の大きさを 1 番目の要素の大きさで割ることによって要素数を求めるテクニックは、構造体の配列でも使えます。

実行結果

Sato さん: 182cm

練習課題 23 (ex174.c)

練習課題 21、22 で使った構造体 PDATA へのポインタを受け取り、BMI を計算して返す関数 `calc_bmi()` を作ってください。

BMI は、 $\frac{\text{体重}_{(\text{kg})}}{\text{身長}_{(\text{m})}^2}$ (“kg 単位の体重” ÷ “m 単位の身長の 2 乗”) で計算できます^a。

途中まで作ったプログラムを、授業のサイトに `ex174.c` として掲載しています。コメント部分に `calc_bmi()` の定義を記述して、正しく動作するようにしてください。

実行結果

Ito さんの BMI は 24.6 です。

Sato さんの BMI は 22.3 です。

Takahashi さんの BMI は 19.6 です。

Hasegawa さんの BMI は 25.2 です。

Yano さんの BMI は 23.5 です。

^a “C 言語基礎演習” の練習課題 7 では、cm 単位の身長と kg 単位の体重から BMI を求めるプログラムを作りました。

続・C 言語基礎演習 教科書
第 1.2 版
2024/4/1
主任講師 森本容介