第 10 章

ユーザ定義型と構造体

　今までの変数が箱だとすると、構造体とは、いくつかの箱を詰め合わせたダンボール箱のようなものです。ここでは複数のデータを、ダンボール箱に入れるかのようにして、まとめて扱う手法を学びます。

　ここにきて、型の概念がますます重要になります。**変数**はプログラムで定義するものでしたが、**型**も作ることができます。変数には実体がありますが、型だけでは実体がありません。変数か型か、どちらの話かよく区別して読み進めてください。

　構造体は、**オブジェクト指向**にもつながる重要な概念です。それでいて、難しくはありませんので、しっかりと身につけていきましょう。

キーワード

- ユーザ定義型・typedef・struct
- 構造体のメンバ変数
- 値渡し・参照渡し
- 型と関数名を直交させる

→オブジェクト指向

10.1　基本データ型・ユーザ定義型

「**型**」の種類は、次の 2 つに大別されます。

基本データ型 (basic data type)
: int とか double のような、C 言語に最初から備わっている型です。

ユーザ定義型 (user-defined data type)
: プログラムで作った型です。基本データ型を組み合わせて作ります。

ユーザ定義型を作る方法はいくつかあります[*1]。まず typedef 命令の使い方を見てみましょう。このキーワードは type definition（型定義）の意味でしょう。次の文法[*2]で、既に存在する型に、別の名前を与えます。

```
/* 文法 */
typedef 既存の型名 新規の型名;
```

```
/* 実例 */
typedef int myint_t;
```

右の例のようにすると、myint_t が int とまったく同じ働きをするようになります。実質的には何も増えないのですが、ユーザが定義した、新たな型ができたように見えます[*3]。なお、型名には、このように **"_t" の接尾辞**をつける習慣があります。

コラム： typedef の使い道

C 言語の int や long の記憶できる値の範囲は、CPU やコンパイラなどの環境によって異なります（☞ ??項）。int に 16 ビットの領域を割り当てる場合も、32 ビットの場合もあります。

ある数値を扱うのに 32 ビットの領域が必要だとしましょう。int を使うと、環境によっては正しく動きません。そこで、例えば int32_t という型を使うことにして、環境に応じて以下のどちらか片方を有効にして、int か long かを選びます。

```
  typedef long int32_t; // intが16ビットなら
//typedef int  int32_t; // intが32ビットなら
```

このように、**後から型を変更する**のが typedef の典型的な使用例でした。

もっともこれは以前の話で、C99 から int32_t が言語規格に取り込まれ、プログラマが使い分ける必要はなくなりましたが、今でも裏方で typedef が働いています。

*1 enum, union もユーザ定義型を作りますが、本書では扱いません。
*2 マクロの #define と働きが似ているためか、末尾の ; を忘れがちです。そして定義の語順が逆です。
*3 マクロでも近い効果は得られますが、配列やポインタを含む型では不都合があります。

TODO: たい焼きの「型（構造体）」:「魚（変数）」

10.2　構造体

構造体 (structure) は、複数の変数をまとめるユーザ定義型です。例えば、xy 平面上の座標を扱う時には、x 座標と y 座標の 2 つの変数を、常にペアにして使いますから、このような場面で役立ちます。構造体を作るには、次のように `struct` 命令を用います。そして先ほどの typedef と組み合わせるのが簡便です。

```
/* 文法（短縮した慣用句） */
typedef struct {
    型名1 メンバ名1;
    型名2 メンバ名2;
    ...
} 新規の型名;
```

```
/* 実例 */
typedef struct {
    double x; // double x, y;
    double y; // でもよい
} point_t;
```

struct の後の { } の中に、ひとまとめにする変数を列挙します。これらの変数を、構造体の**メンバ** (member) といいます。右上の例では double 型の x と y です。この構造体に、typedef で型名を付けます。例では point_t にしています。

この構造体を使って、**変数**を定義してみましょう。構造体は**型**ですから、「double」のような型名を書くところに「point_t」と書きます。

```
/* 普通の変数の定義と代入 */
double x, y;
x = 1.0;  y = 2.0;
```

```
/* 構造体変数の定義と代入 */
point_t p;
p.x = 1.0;  p.y = 2.0;
```

構造体変数を1つ作ると、含まれるメンバすべてが1組作られます。右の例では、変数 p のメンバの x と y が作られるので、値を代入してみました。構造体の中のメンバを指すには、このように**構造体メンバ参照**の演算子 「`.`」に続けてメンバ名を指定します。メンバ変数も、普通の変数と同じように、代入したり、参照したりできます。

構造体の**初期化**は、1 次元配列と同じで、初期化子をブロックで囲って書き並べます。

```
/* 普通の変数の初期化 */
double x = 1.0, y = 2.0;
```

```
/* 構造体変数の初期化 */
point_t p = { 1.0, 2.0 };
```

コラム： typedef と組み合わせない struct

構造体には、**構造体タグ名** (tag) をつける用法もあります。この場合、型名は「struct 構造体タグ名」の 2 つの単語になり、typedef は使わなくてよくなります。

```
/* 文法 */
struct 構造体タグ名 {
    型名1 メンバ名1;
    型名2 メンバ名2;
    ...
};

/* 変数定義と代入 */
struct 構造体タグ名 変数名;
変数名.メンバ名 = 値;
```

```
/* 実例 */
struct point {
    double x;
    double y;
}; // ←セミコロン忘れに注意

/* 変数定義と代入 */
struct point p;
p.x = 1.0;  p.y = 2.0;
```

この用法には、注意点がいくつかあります。TODO: タグ名→タグ？

- struct のキーワードを記述する場面が多くなり、少々煩雑です。
- 構造体宣言の末尾のセミコロン（;）を忘れがちです。
- C++ との表記の乖離が大きくなります。C++ では、構造体タグ名が、そのまま型名でもあるので、struct を省略しがちです。先に紹介した、typedef との組合せは、C++ と表記を共通にするための手段でもあります。

ただし、メンバに同じ構造体（へのポインタ）を含む**再帰的なデータ構造**には必須の用法です。そして、次のように typedef と組み合わせることも多くあります。

```
/* typedef と分離 */
struct list {
    int data;
    struct list *next;
};
typedef struct list list_t;
```

```
/* typedef と同時 */
typedef struct list {
    int data;
    struct list *next;
    // list_t はここでは未定義
} list_t;
```

右の例でも、list_t はメンバの定義には使えないことに注意してください。

10.2.1　構造体のスコープ

構造体の（変数ではなく、型の）スコープは、宣言したブロックに制限されます。つまり、変数と同じです。しかし構造体は、関数の引数などプログラム全体にわたって使いたいので、スコープを制限する意味はほとんどありません。ですから、トップレベル（関数の外）、しかも関数のプロトタイプ宣言よりも前で構造体を宣言します。

鋭意作成中

10.3 構造体の使用例

構造体を作ったら、同時にメンバを表示する関数も作りましょう。デバッグのためにいずれ必要になります。ソースコード 10.1 を見ていきます。

4–6 行目の構造体の宣言は、**関数定義やプロトタイプ宣言よりも前**にすませる必要があります。順序を間違えると、不可解なコンパイルエラーになります。

10 行目では、printf() で構造体変数 p のメンバの x と y を表示しています。メンバ参照演算子の **.** でメンバの値を一つずつ取り出します。**printf() に構造体を自動的に表示する機能はない**ので、このようにメンバを一つずつ指定して表示する必要があります。

15 行目の関数呼び出しでは、構造体変数 a を引数に渡しています。int 型でも point_t 型でも、**引数への渡し方は同じ**です。

17 行目では、b に a を代入しています。**構造体変数を代入すると、すべてのメンバがコピー**されます。18 行目の表示から、メンバの x, y ともコピーされたことがわかります。

ソースコード 10.1 座標を表示

```
1  #include <stdio.h>
2
3  /* 点を表す構造体 */     // 関数定義やプロトタイプ宣言よりも前に必要
4  typedef struct {
5      double x, y;
6  } point_t;
7
8  /* pを表示する */
9  void point_print(point_t p) {
10     printf("(%g, %g)\n", p.x, p.y); // .(ドット)でメンバーを取り出す
11 }
12
13 int main(void) {
14     point_t a = { 1.1, 2.2 }; // a.x = 1.1;  a.y = 2.2;  [初期化]
15     point_print(a);           // (1.1, 2.2)              [引数渡し]
16     point_t b;
17     b = a;                    // b.x = a.x;  b.y = a.y;  [代入]
18     point_print(b);           // (1.1, 2.2)
19 }
```

10.3.1　構造体を扱う関数を増やす

次に、point_t の点を移動する関数を作ってみましょう。ソースコード 10.2 です。

15 行目の point_move(p, dx, dy) は、点 p を (dx, dy) 方向に移動した点を返します。16 行目で、引数で受け取った p の x, y 座標を変化させて、17 行目で移動後の p を返します。

呼び出し側の main() では、23 行目で point_move() の戻り値を変数 a に代入して、値を更新します。

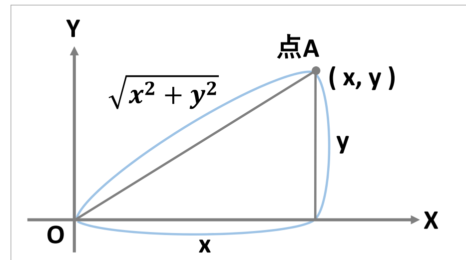
10.3.2　構造体のメンバを増やす

プログラムが一旦完成しても、後になって機能拡張したくなることは、よくあります。例えば点に名前（ラベル）を付けたくなったとしましょう。それには、ソースコード 10.3 のように、構造体のメンバを増やします。

6 行目は、文字列を格納するメンバ変数 label を追加しました。10 行目からの point_print() は、引数はそのままですが、label を表示するための処理を追加しました。21 行目の初期化子にも、label に対応する文字列を追加しました。

ソースコード 10.2　点を移動する

```
1  #include <stdio.h>
2
3  /* 点を表す構造体 */
4  typedef struct {
5      double x, y;
6  } point_t;
7
8
9  /* pを表示する */
10 void point_print(point_t p) {
11     printf("(%g, %g)\n", p.x, p.y);
12 }
13
14 /* pを(dx,dy)方向に移動する */
15 point_t point_move(point_t p, double dx, double dy) {
16     p.x += dx;  p.y += dy;
17     return p;
18 }
19
20 int main(void) {
21     point_t a = { 1.1, 2.2 };
22     point_print(a);            // (1.1, 2.2)
23     a = point_move(a, 1.2, 3.4);
24     point_print(a);            // (2.3, 5.6)
25 }
```



メンバが増えても 15 行目からの point_move() は以前と同じです。22–24 行目の関数呼び出しもそのままです。もし構造体ではなく、x, y, label を別々の引数にしていたら、関数の引数をあちこち修正せねばなりません。このように**データセットを柔軟に拡張**できるのが、構造体のありがたいところです。

ソースコード 10.3　点を移動する（ラベル付き）

```
#include <stdio.h>

/* 点を表す構造体 */
typedef struct {
    double x, y;
    char *label;                                // *label 追加
} point_t;

/* pを表示する */
void point_print(point_t p) {
    printf("点%s(%g, %g)\n", p.label, p.x, p.y); // p.label 追加
}

/* pを(dx,dy)方向に移動する */
point_t point_move(point_t p, double dx, double dy) {
    p.x += dx;  p.y += dy;
    return p;
}

int main(void) {
    point_t a = { 1.1, 2.2, "A" };              // "A" 追加
    point_print(a);                 // 点A(1.1, 2.2)
    a = point_move(a, 1.2, 3.4);
    point_print(a);                 // 点A(2.3, 5.6)
}
```

10.3.3　構造体の代入

これまで見てきたように、通常の int 型の変数と同じように、point_t a, b; と定義した**構造体の変数**は、a=b のような**代入**が行えて、構造体の**すべてのメンバがコピー**されました。関数の**引数渡し**も**戻り値**も、この代入相当の動作が行われていて、すべてのメンバがコピーされました。

この代入は便利そうに思える機能ですが、構造体が大きくなると効率性が問題になります。構造体のメンバに配列を含めることもできますが、動作が保証されるサイズには上限があります[*4]。このため、次に紹介するように、通常は**関数とは構造体をポインタでやりとり**します。

10.4　構造体によるデータ構造

10.4.1　構造体のポインタ

大きな領域のコピーを避けるために、関数とは**構造体をポインタで**受け渡すのが一般的な手法です。ポインタにするためには、次のようにします。

仮引数 受取側の関数では、引数名に * を付加します。

実引数 呼び出し側では、ローカルに定義した構造体変数なら、& を付加して、アドレスに変換します。初めからポインタ型であれば、変数名だけで大丈夫です。

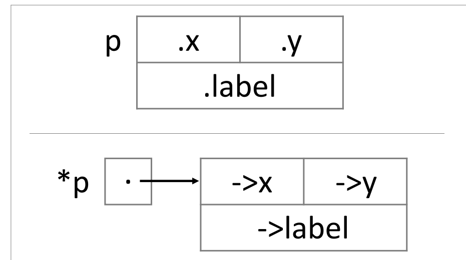
戻り値 構造体を戻り値とする必要がなくなります。関数の型は void にして、代わりに引数にし、関数で書き込みます。

ポインタ変数の point_t *p のメンバを指すには、文法通りには「`(*p).x`」と煩雑な表記になります。ここにカッコが必要なのは、間接参照の * の優先順位が低く、メンバ参照の `.` が先に評価されるからです。しかし、これはよく使う機能ですから**専用の演算子**が用意されていて、**ポインタ用のメンバ参照**の「`->`」を用いて「`p->x`」と簡潔に表記できます。

```
/* 引数が構造体 */
double point_sum_xy(point_t p) {
    return p.x + p.y;
}
```

```
/* 引数が構造体ポインタ */
double point_sum_xy(point_t *p) {
// return (*p).x + (*p).y; // 可能
    return p->x    + p->y;    // 通常
}
```

*4 C99 で 64KB、C89 で 32KB を超えると、動作不良を起こす場合があります。そもそも最初期のコンパイラでは、構造体変数の代入相当の動作自体がサポートされていない場合もありました。



7 ページのソースコード 10.3 を、ポインタ渡しに書き換えたのがソースコード 10.4 です。これで実用的なプログラムになりました。変更点は以下の通りです。

(a) 関数の仮引数は、* を追加してポインタ型にしました。
(b) メンバ参照の演算子を、ポインタ型の場合には . の代わりに -> にしました。
(c) 構造体を戻り値にしているものは、代わりに引数に作用するようにしました。呼び出し側では、戻り値を代入せずとも、実引数が変化します。
(d) 関数の実引数は、ローカルに確保した場合は & でアドレスに変換しました。

ソースコード 10.4 原点からの距離を表示（ポインタ渡し）

```c
#include <stdio.h>

/* 点を表す構造体 */
typedef struct {
    double x, y;
    char *label;
} point_t;

/* pを表示する */
void point_print(point_t *p) {                          // (a)* 追加
    printf("点%s(%g, %g)\n", p->label, p->x, p->y); // (b).を->に変更
}

/* pを(dx,dy)方向に移動する */                          // (c)void に変更
void point_move(point_t *p, double dx, double dy) { // (a)* 追加
    p->x += dx;  p->y += dy;                        // (b).を->に変更
    /* return p; */                                 // (c)return 削除
}

int main(void) {
    point_t a = { 1.1, 2.2, "A" };
    point_print(&a); // 点A(1.1, 2.2)               // (d)& 追加
    /* a = */ point_move(&a, 1.2, 3.4); // (c)a= 削除, (d)& 追加
    point_print(&a); // 点A(2.3, 5.6)               // (d)& 追加
}
```

10.4.2　構造体の配列

int に配列があるように、構造体の配列も作れます。int と同様に、変数定義の変数名に続けて [ ] と、角括弧の中に要素数を書きます。

```
/* 普通の配列 */
double x[3], y[3];
char *label[3];

x[0] = 1.0;  y[0] = 3.0;
label[0] = "A";
```

```
/* 構造体の配列 */
typedef struct {
    double x, y;
    char *label;
} point_t;
point_t p[3];

p[0].x = 1.0;  p[0].y = 3.0;
p[0].label = "A";
```

p[0] のように、添字をつけると構造体の 1 つの変数を表すので、そのメンバを指すには、p[0].x のような表記になります。

初期化子のブロックは、2 次元配列のように 2 重になります。右下の例のように、1 つの構造体のデータが内側のブロックにまとまって好都合です。通常の配列だと、左下の例のように、値が散らばってしまいます。

```
/* 普通の配列 */
double x[3] = { 1.0, 2.0, 3.0 };
double y[3] = { 3.0, 2.0, 1.0 };
char *label[3] = {"A","B","C"};
```

```
/* 構造体の配列 */
point_t p[3] = {
    { 1.0, 3.0, "A" },
    { 2.0, 2.0, "B" },
    { 3.0, 1.0, "C" },
};
```

配列の 1 要素を、ポインタで関数に渡すときには、int でも構造体でも同じで、添字を指定した上で、& でアドレスに変換します。

```
/* 普通の配列要素のポインタ */
for (int i=0; i<3; i++) {
    point_print(&x[i], &y[i]);
}
```

```
/* 構造体の配列要素のポインタ */
for (int i=0; i<3; i++) {
    point_print(&p[i]);
}
```

配列全体を関数の引数として渡すこともできます。（構造体に限らず）配列はポインタと同じで、関数には参照渡しになり、要素数はポインタからはわかならいので、別の引数で渡します。**TODO:** 構造体配列のほうが CPU キャッシュの恩恵を受けやすい

```
/* 普通の配列 */
point_print_array(3, x, y);
```

```
/* 構造体の配列 */
point_print_array(3, p);
```

鋭意作成中

10.4.3 メンバの同じ構造体

構造体は、宣言するたびに新しい型になります。たとえメンバが同じでも、宣言しなおせば、**異なる型**と扱われます。これはありがたい性質です。

double s[2]; と定義した変数なら、例えば、左右の視力でも、左右の握力でも格納できます。視力と握力が**同じ型**ですから、取り違えたおかしなプログラムも作れてしまいます。構造体にしていれば、取り違えるとコンパイルエラーになるので、間違いが未然に防げます。このように、構造体は**型で意味づけを与えて**います。

```
/* double配列 */

  // 準備は不要







/* 視力表示 */
void sight_print(double s[2]) {
    printf("視力 左:%g, 右:%g\n",
           s[0], s[1]);
}

int main(void) {
    // 視力
    double s[2] = { 1.2, 1.5 };
    // 握力(kgf)
    double g[2] = { 15.5, 13.0 };
同じ型
    sight_print(s); // ○
    sight_print(g); // 無意味
}
```

```
/* 構造体 */
typedef struct { // 視力
    double left, right;
} sight_t;

typedef struct { // 握力(kgf)
    double left, right;
} grip_t;                  同じメンバ

/* 視力表示 */
void sight_print(sight_t *s) {
    printf("視力 左:%g, 右:%g\n",
           s->left, s->right);
}

int main(void) {
    // 視力
    sight_t s = { 1.2, 1.5 };
    // 握力(kgf)
    grip_t g = { 15.5, 13.0 };
異なる型
    sight_print(&s); // ○
//  sight_print(&g); // エラー
}
```

10.5　型にまつわる命名の習慣

ユーザー定義型の名前に “_t” の接尾辞をつける習慣があることは、既に述べました。

ユーザー定義型を扱う**関数**の名前には、接頭辞に型の名前（もちろん “_t” を除いて）をつけるとよいでしょう。そして**最初の引数**にその変数を持ってくるとわかりやすいでしょう。プロトタイプ宣言は、例えば以下のようになります。

```
void point_print(point_t *p);
void point_move(point_t *p, double dx, double dy);
```

ところで、型が違っても、似たような動作をする関数があります。例えば point_print() のような画面表示の関数は、どの型でも必ず作ります。同じ動作をする関数名が、型によって hoge_output() とか fuga_dispay() のように違うとややこしいので、(接頭辞の型名を除いて) 統一したいところです。つまり、

型と関数名を直交させる

という方針が沸き起こります。残念ながら C 言語には、関数名を統一するような支援機能はありませんので、プログラマが意識的に揃える必要があります。

×

```
void point_print(point_t *p);
void hoge_output(hoge_t *h);
void fuga_dispay(fuga_t *f);
```

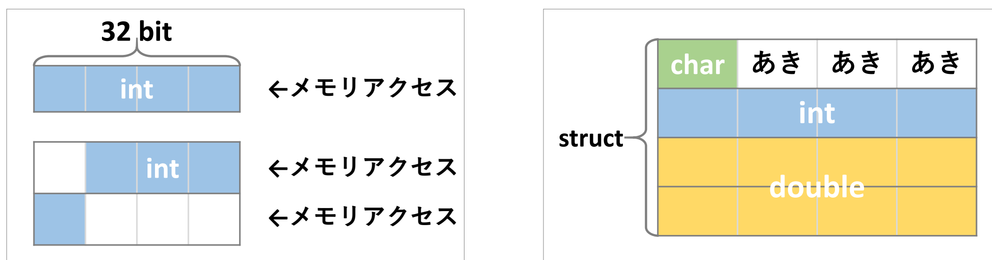
○

```
void point_print(point_t *p);
void hoge_print(hoge_t *h);
void fuga_print(fuga_t *f);
```

コラム： 構造体からオブジェクト指向へ

型（構造体）を中心にして関数を作り始めると、関数を**型ごとに分類**したくなります。その分類のまま「型とそれにまつわる関数」をひとまとまりにしたものが、オブジェクト指向言語の**クラス**に相当します。つまり構造体は**オブジェクト指向**への足がかりになります。

オブジェクト指向言語では、もう関数に型名の接頭辞をつける必要はありません。さらに、クラス（型）が異なっても、メソッド名（≈ 関数名）を自然と揃えたくなる仕組み（クラスの継承）もあります。継承のおかげで、クラス群全体を考えながらクラスを設計することになります。



10.6 構造体のアライメント・パディング（発展的内容）

構造体の大きさは、含まれているメンバ変数の大きさの合計に等しいと思いがちですが、実際にはそれよりも大きくなることがあります。その仕組みを説明します。

まず、通常の変数を考えてみます。32 ビット CPU であれば、1 回のメモリのアクセス（読み書き）を **32 ビット単位**で行います。32 ビットは 4 バイトですから、メモリの番地が 4 の倍数からの 4 バイトなら 1 回でアクセスできます。ところが、途中からの 4 バイトには 2 回のアクセスが必要となり、速度が低下します。これを避けるために、コンパイラは隙間を空けて、次の **4 の倍数の番地**から変数を配置することがあります。このように、倍数に合わせて配置することを**境界調整**とか**アライメント** (alignment) といいます。

構造体のメンバも事情は同じです。例えば、1 バイトの char のあとに 3 バイトの隙間を空けて、続きのメンバを**次の 4 の倍数の番地**に割り当てる、といったことがよく行われています。隙間（詰め物）のメモリのことを**パディング** (padding) といいます。

どのようにアラインするか、いくつパディングを入れるかは、環境（CPU やコンパイラのオプションなど）によって変わります。プログラムから見れば、いくつパディングされていてもあまり影響はありませんが、メモリの使用量を見積もるなど、考慮の必要な場面もあります。実際のパディング量は sizeof 演算子で確かめましょう。

```
struct char_and_int {
    char a;
    int b;
};
printf("%zu\n", sizeof(char));    // 1
printf("%zu\n", sizeof(int));     // 例:4 （環境による）
printf("%zu\n", sizeof(struct char_and_int)); // 例:8 （環境による）
```

上の例では、char が 1 バイト、int が 4 バイトで、これを合わせた構造体が 8 バイトですから、この構造体には $8 - (1 + 4) = 3$ バイトのパディングが含まれているとわかります。

MEMO: memcmp() で比較できない理由でもある

10.7　例題・分数計算

分数の和と差を計算してみましょう。??ページの練習問題 ??. では、和を受け取るのに分子用と分母用に 2 つの関数を使いましたが、構造体を用いると、どのように扱いやすくなるでしょうか。ソースコード 10.5 を見てみましょう。

型宣言 5–8 行目で、分数 $\frac{b}{a}$ を表す構造体を作ります。型名を、fraction（分数）の省略で frac_t としてみます。分子、分母ともに整数でよくて、メンバ名を単純に b と a にします。**分母 a を正**にすると決めておくと、この後の場合分けが少なくなります。

約分 26–30 行目の frac_reduce(*x) は、まず受け取った *x の分子と分母の**最大公約数**を求めます（27 行目）。分子は負の可能性があるので、絶対値をとります。そして分子・分母ともに公約数で割ります（28–29 行目）。**引数の *x を加工**しているので、関数自体は値を返す必要がなく、**void 型**にしています。

加算 33–37 行目の frac_add(*r, *x, *y) では、分数の和 $\frac{b}{a} + \frac{d}{c} = \frac{bc+ad}{ac}$ の計算をします。34–35 行目で、引数の *r の分子と分母に、約分前の計算結果を代入します。そして 36 行目で、**約分のために frac_reduce(r) を呼び出し**て完了です。r はポインタ型ですから、関数呼び出しに & は不要です。

減算 差の計算は、**符号を変えた和の計算**と同じです。40–43 行目の frac_sub(*r, *x, *y) では、41 行目で *y の分子の符号を変えた分数 tmp を作り出して、42 行目で**加算の関数を frac_add(r, x, &tmp) と呼び出し**ます。約分も加算の関数に任せていて簡単です。tmp はポインタ型ではないので、関数呼び出しには & が必要です。

構造体を使わずに、分子と分母の 2 つの関数に分けていたときは、通分のための最大公約数を、それぞれの関数で求めていました。後の処理を、他の関数に任せることもできませんでした。構造体のおかげで、便利になったことを感じてもらえたでしょうか。

ソースコード 10.5　分数の和と差

```
1  #include <stdio.h>
2  #include <stdlib.h> // abs()
3
4  /* b/a の分数を表す型 (a>0) */
5  typedef struct {
6      int b; // 分子
7      int a; // 分母
8  } frac_t;
9
10 // プロトタイプ宣言
11 int gcd(int x, int y);
12 void frac_reduce(frac_t *x);
13 void frac_add(frac_t *r, frac_t *x, frac_t *y);
14 void frac_sub(frac_t *r, frac_t *x, frac_t *y);
```

```
15 void frac_print(frac_t *x);
16
17 /* xとyの最大公約数を返す (x,y>=0 かつ x+y>0 を前提とする) */
18 int gcd(int x, int y) {
19     while (y != 0) { // ユークリッドの互除法
20         int t = x % y;  x = y;  y = t;
21     }
22     return x;
23 }
24
25 /* 約分 */
26 void frac_reduce(frac_t *x) {
27     int d = gcd(abs(x->b), x->a);
28     x->b /= d;
29     x->a /= d;
30 }
31
32 /* 加算 (*r = *x + *y) */
33 void frac_add(frac_t *r, frac_t *x, frac_t *y) {
34     r->b = x->b * y->a + x->a * y->b;
35     r->a = x->a * y->a;
36     frac_reduce(r); // 約分を利用
37 }
38
39 /* 減算 (*r = *x - *y) */
40 void frac_sub(frac_t *r, frac_t *x, frac_t *y) {
41     frac_t tmp = { -y->b, y->a }; // 分子の符号を反転した分数を作る
42     frac_add(r, x, &tmp);         // 加算を利用
43 }
44
45 /* 表示 */
46 void frac_print(frac_t *x) {
47     printf("%d/%d\n", x->b, x->a);
48 }
49
50 int main(void) {
51     frac_t x = { 7, 12 }; // 7/12
52     frac_t y = { -1, 4 }; // -1/4
53     frac_t r;             // 結果の代入先
54     printf("add "); frac_add(&r, &x, &y); frac_print(&r); // 1/3
55     printf("sub "); frac_sub(&r, &x, &y); frac_print(&r); // 5/6
56     printf("0   "); frac_sub(&r, &x, &x); frac_print(&r); // 0/1
57 }
```

$$\frac{x_b}{x_a} = \frac{x_b/d}{x_a/d}$$

$$\frac{r_b}{r_a} = \frac{x_b}{x_a} + \frac{y_b}{y_a} = \frac{x_b y_a + x_a y_b}{x_a y_a}$$

10.8 練習問題

1. **[構造体の関数渡し（☞ 10.3.1 項）]**

   5 ページのソースコード 10.1 に、次の関数を追加せよ。

- `double point_norm(point_t p)` は、原点から点 p までの距離を返す。

2. **[構造体ポインタの関数渡し（☞ 10.4.1 項）]**

   9 ページのソースコード 10.4 に、次の関数を追加せよ。

   - `double point_norm(point_t *p)` は、原点から点 *p までの距離を返す。

3. **[メンバ変数の追加（☞ 10.3.2 項、10.4.1 項）]**

   9 ページのソースコード 10.4 を改造して、point_t 型に z 座標を追加して、3 次元座標上の点に拡張せよ。point_move() の引数には z 座標の移動量 dz を追加し、main() で必要になる z 座標の値は自由に設定せよ。

4. **[名は体を表す（☞ ??節）]**

   ソースコード 10.5 の演算後の**約分**が不要になったとする。次の 3 通りの改修案は、いずれも frac_add() に同じ効果を与えるが、**他人がさらに改修する可能性を考慮**して、避けるべきものを2つ選び、その理由を述べよ。 **(i)** 22 行目を「`return 1;`」にする **(ii)** 28–29 行目をコメントにする **(iii)** 36 行目をコメントにする

5. **[関数の再利用]**

   ソースコード 10.5 では、frac_sub() は frac_add() の**関数を呼び出し**ているが、呼び出さずに関数内の**処理をコピーして符号を反転**しても実現できる。この 2 つの手法のどちらが有利かを、次の 3 通りの評価基準で判断せよ。**(i. 実装効率)** プログラムが作りやすい **(ii. 信頼性)** 動作検証が容易 **(iii. 保守性)** プログラムの改造が容易

6. **[構造体の関数渡し]**

   ソースコード 10.5 に、*x の逆数を *r に代入する関数 frac_inv(*r, *x) を追加せよ。*x の分子が 0 でないことを前提としてよいが、逆数 *r の分母が負になれば、分子分母ともに符号を反転して、**分母を正**に保つようにせよ。

   そして加算と同様の、乗算 frac_mul(*r, *x, *y) と、除算 frac_div(*r, *x, *y) の関数を追加せよ。いずれも、分母が正の、約分した分数を *r に代入せよ。

7. **[構造体の関数群]**

   2 次方程式 $ax^2 + bx + c = 0$ に関して、次の 2 つの構造体（型）を宣言せよ。

   - `quad_t` のメンバは double 型の a, b, c であり、方程式の**係数**を表す。
   - `sol_t` のメンバは double 型の alpha, beta であり、方程式の**実数解**を表す。

   そして、次の関数群を作れ。簡単のため、$a \neq 0$ かつ**判別式は非負**としてよい。

   - `void quad_create(quad_t *q, sol_t *s)` は `*s` を解とする方程式を `*q` に代入する。$(x-\alpha)(x-\beta) = x^2 - (\alpha+\beta)x + \alpha\beta$ を利用してよい。
   - `double quad_disc(quad_t *q)` は `*q` の判別式 $D\,(= b^2 - 4ac)$ を返す。
   - `void sol_solve(sol_t *s, quad_t *q)` は `*q` の実数解 $\dfrac{-b \pm \sqrt{D}}{2a}$ を `*s` に代入する。

8. **[標準ライブラリの構造体を用いる]** ☞ ??項

索引

■ 記号・数字
->（構造体メンバ間接参照）8
.（構造体メンバ参照） 3

■ I
int32_t 2

■ S
struct 3

■ T
typedef 2

■ あ
アライメント **13**

■ か
間接参照 8

■ き
基本データ型 **2**
境界調整 . . . . → アライメント

■ こ
構造体 **3**
構造体タグ名 **4**

■ さ
参照渡し 10

■ し
初期化子 3, 10

■ と
トップレベル 4

■ は
パディング **13**

■ め
メンバ **3**
メンバ参照 3, 8

■ ゆ
ユークリッドの互除法 15
ユーザ定義型 **2**

■ よ
要素数 10