プログラミング通論'20 ＃2 – アドレスとポインタ

久野 靖 (電気通信大学)

2020.2.21

今回は次のことが目標となります。

- アドレスとポインタの概念、およびC言語におけるそれらの扱いを理解する。
- 配列や構造体とポインタの関係について理解し扱えるようになる。

アドレスとポインタの概念
アドレスと左辺値・右辺値

コンピュータがプログラムを実行するときの主要な構成要素は、データを格納する**主記憶**(memory、メモリ)と、命令を実行し演算をおこなう**CPU**(central processing unit)です。そして図1のように、メモリには**番地**(address、アドレス)が割り振られていて、番地を指定することでデータの取り出しや格納が行なえます。この図では番地は4飛びに(16進で)表記してありますが、これは整数1個が4バイト(32ビット)で、番地は1バイト単位でつけてあるCPUが多いのでそれにならっています。

アセンブリ言語や高水準言語でプログラムするときは、番地を直接書いていたら繁雑ですから、適当な名前(AとかBとか)をつけて扱っています。この図の例では3つのアセンブリ言語命令がありますが、それは次のような意味になります(番地は適当な例示で16進表記です)。

アドレスと左辺値・右辺値

- A の番地 (1C04) から整数値を取り出し、レジスタ eax に転送する。
- B の番地 (1C08) から整数値を取り出し、レジスタ eax に現在入っている値と加算する (レジスタの値がその加算の結果になる)。
- レジスタ eax に入っている値を、C の番地 (1C0C) に格納する。

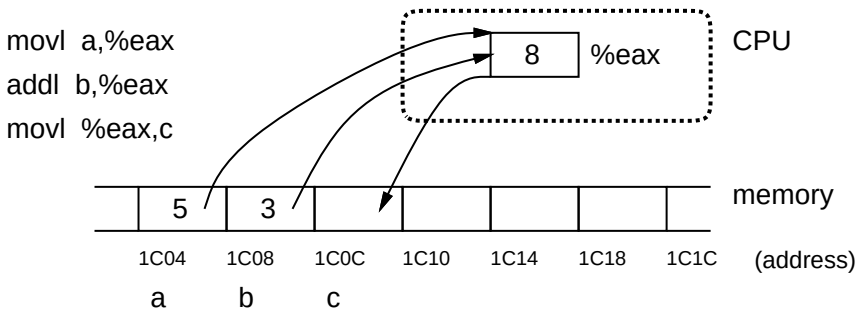

図 1: CPU 命令とメモリ番地

アドレスと左辺値・右辺値

C言語の科目なのでCに戻るとして、これをCでは次のように書くわけです。

```
c = a + b;
```

ここで良く見て欲しいのですが、「=」の左と右では変数の意味するところが違います。右では「aに入っている値」「bに入っている値」を意味しますね。これを**右辺値** (right value、rvalue) と呼びます。しかし、左では…「c = ...」というのは「cの番地」つまり図でいうと1C0C番地に (右辺で計算した) 値を格納する、という意味になります。つまり、代入の左に書いた場合はそれは番地 (アドレス) なのです。これを**左辺値** (left value、lvalue) とも呼びます。

アドレス取得演算子・間接参照演算子

普段はこのようにアドレスと値(右辺値)を使い分けているのですが、C 言語の特徴として、任意の変数のアドレスを取得して値として扱える、ということがあります(ややこしくなる原因でもあります)。たとえば次のコードを見てみましょう。

```
int a, b = 10, *p;
p = &a; *p = b;
```

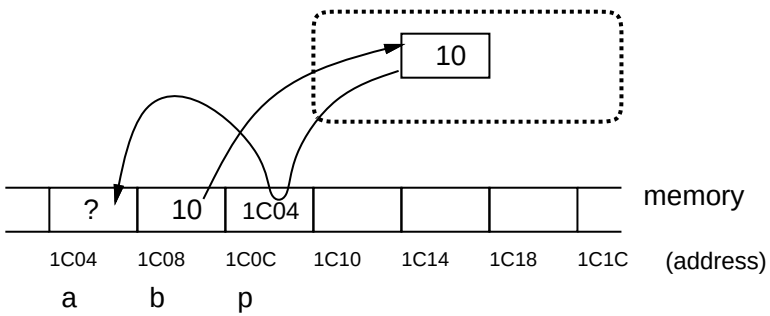

図 2: アドレスの取得と間接参照

アドレス取得演算子・間接参照演算子

これは、1行目の宣言/初期化で、変数aとbは整数型、pは「整数を指すポインタ型」と宣言しています。ポインタ型とは要するに「アドレスを表す型」のことです。さらに、bには初期値10を入ています。次に2行目で、pに「aのアドレスを」入れています(「&」は「アドレス取得演算子」です)。そして3行目はbに入っている値を取り出し、「pに入っているアドレスに」格納します。pには変数aのアドレスが入っているわけですから、結果として変数aに10が入ることになります(図2)。[1]

「*p = b;」の「*」は「参照たどり演算子」ないし「**間接参照演算子** (indirect reference operator)」と呼ばれ、ポインタ型の値にだけ使えます。その意味は、左辺値として使う場合は「そのポインタ値が表すアドレスに格納」、右辺値として使う場合は「そのポインタ値が表すアドレスに格納されている値」となります。上記は左辺値に現れる例でしたが、たとえば続いて「b = *p + 1;」のように書くと、こんどは「aに入っている値に1を足してbのアドレスにに格納」になります。

[1]今日のシステムではアドレスは 64 ビット長が多いのですが、図が繁雑になるので図ではアドレスも整数と同じ 32 ビットの長さとして描いてあります。さらに、32 ビットのアドレスは 16 進表記で 8 桁になりますが、見にくいので 16 進 4 桁で例示しています。

アドレス取得演算子・間接参照演算子

なぜ「間接」かというと、「a = b」や「b = a」のように書いた場合は「aの番地に代入」「aの値を取り出し」のように「直接」指定しているのに対し、ポインタ変数pを経由している場合は「いちどpに入っている番地を取り出し、その番地への代入/番地に格納されている値を参照」となるからです。面倒なだけに思えるかも知れませんが、このように「間接」にすることで、pに格納する番地を変更することでさまざまな場所にある値を扱えるという柔軟性が得られるのです。

ここで、C言語の宣言の書き方と型の書き方について説明しておきます。「int i;」と書いた場合、変数iは整数型、というのは普通に使ってきました。では「int *p;」は? これは「変数p に間接参照演算子を適用した*pが整数である」ことを意味します。ということは、pそのものは「整数へのポインタ型」となります。たとえば「int **q;」であれば、2回間接参照すると整数になるので、qは「整数へのポインタのポインタ型」です。

アドレス取得演算子・間接参照演算子

そして、**キャスト演算** (cast operation)「(型指定) 式」のために型指定を書くときは、変数宣言から宣言される変数名を取り除いたものを書きます。ですから、`(int)` は整数型へのキャスト、`(int*)` は整数のポインタ型へのキャスト、`(int**)` は整数のポインタへのポインタ型へのキャストになります。C 言語のここの構文は大変わかりづらいのですが、そういうことになっています。

配列とポインタ演算

次は**配列** (array) です。C 言語では配列を確保するということは、単にその配列の各要素を格納するのに必要な領域を用意することと同じです。そして、変数宣言で配列を確保した場合、その宣言した名前は… 「領域の先頭のアドレス」になります。次を見てください。

```
int a[5] = {1,3,5,7,9}, i, *p; p = a;
```

このようにすると、メモリ上にはまず整数5個ぶんの領域が用意され、配列 a となります。そして整数変数 i、ポインタ変数 p の領域が取られます。次の「p = a;」は? 配列の名前 a は、配列の先頭のアドレスつまり 1C04 を意味するので、それが p に入ります。

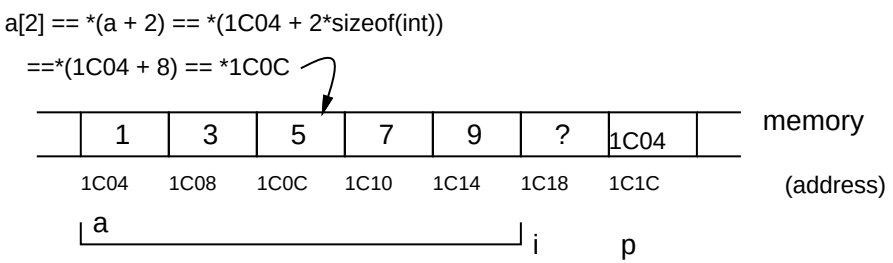

図 3: 配列の領域とポインタの関係

配列とポインタ演算

そして、配列は添字をつけてアクセスしますが、C言語では「a[i]」は「*(a + i)」と同じ意味である、と定められています。この足し算は「**ポインタ演算**(pointer calculation)」と呼ばれ、片方がポインタ値、もう片方が整数である必要があります。

そして、アドレスに直接足すのでなく、ポインタが指す要素のバイト数(整数なら32ビットなので4)を掛けて足すと定められています。これはつまり、「ポインタが指している要素(例: 整数)のi個ぶん先の要素のアドレス」となります。そして、外側の*演算子があるので、そのアドレスに入っている値を参照したり(右辺値の場合)、そのアドレスに格納したり(左辺値の場合)できるわけです。

上の例では配列名aに対してポインタ演算していましたが、ポインタ変数pを使っても同じようにできます。そして上の例ではpには配列aのアドレスが入っていますから、どちらでも同じことです。たとえば「a[2]」でも「p[2]」でもどちらも「5」の入っている場所がアクセスできます。

配列の入力・出力・比較

それでは実践に進むことにして、まず配列を読み込む/出力する関数(これらは既習だと思いますが)、そして2つの配列が等しいかどうか調べる関数を作ってみます。これの関数の名前やパラメタを決める必要があるので、それらのプロトタイプ宣言を示します。

- `void iarray_read(int *a, int n);` — 配列`a`に`n`個の値を読み込む。
- `void iarray_print(int *a, int n);` — 配列`a`の`n`個の値を出力。
- `bool iarray_equal(int *a, int *b, int n);` — 二つの配列`a`、`b`の先頭から`n`個の値が等しいなら`true`、そうでなければ`false`を返す。

配列の入力・出力・比較

ではこれらを使うコードを見てみましょう。

```
// iarray_demo.c --- array input/output and equality demo.
#include <stdio.h>
#include <stdbool.h>
void iarray_read(int *a, int n) {
  for(int i = 0; i < n; ++i) {
    printf("%d> ", i+1); scanf("%d", a+i);
  }
}
void iarray_print(int *a, int n) {
  for(int i = 0; i < n; ++i) { printf(" %2d", a[i]); }
  printf("\n");
}
bool iarray_equal(int *a, int *b, int n) {
  for(int i = 0; i <= n; ++i) {
    if(a[i] != b[i]) { return false; }
```

```
  }
  return true;
}
char *bool2str(bool b) { return b ? "true" : "false"; }
int main(void) {
  int a[4], b[4];
  iarray_read(a, 4); iarray_print(a, 4);
  iarray_read(b, 4); iarray_print(b, 4);
  printf("equal: %s\n", bool2str(iarray_equal(a, b, 4)));
  return 0;
}
```

配列の入力・出力・比較

bool2str とは? C 言語では bool 型といっても整数なので、普通に出力すると 0 か 1 になりますが、true、false と出力したいので「論理値を"true"、"false" の文字列 (C 言語では文字ポインタ) に変換」します。実行例は次の通り。

```
% ./a.out
1> 4
2> 3
3> 2
4> 1
  4  3  2  1
1> 4
2> 3
3> 2
4> 1
  4  3  2  1
equal: false
%
```

配列の入力・出力・比較

あれれ、同じ内容を打ち込んだのに「等しくない」という結果になっていますね? iarray_equal にバグがあるようです。ここですぐ iarray_equal を熟読してバグを取ろうとする方、ちょっと待ってください。このようにバグがあると分かればそれを探して除去できますが、たとえば上の実行例で「2 つの配列が違う場合」だけ試してしまうと、この例のように「等しいのに等しくないと答える」バグはあると気づかず見過ごされます。ではどうするのがよいのでしょう?

単体テストとテストケース

array_equalが正しいかどうかを確認するのに、上のように「気分で打ち込んで」様子を見るのはよい方法ではありません。もっと系統的に「確認するためのデータ」を用意して調べるべきです。調べる対象である単純な機能に対してその正しさをチェックするテストは**単体テスト** (unit test)、テストに含まれる「試行データと想定正解の組」を**テストケース** (test cases) と呼びます。具体例を見てみましょう。

```
// test_array_equal.c --- unit test of array_equal
#include <stdio.h>
#include <stdbool.h>
(iarray_read, iarray_print, iarray_equal, bool2str here)
void expect_bool(bool b1, bool b2, char *msg) {
  printf("%s %s:%s %s\n", (b1==b2)?"OK":"NG",
         bool2str(b1), bool2str(b2), msg);
}
int main(void) {
  int a[] = {0, 1, 2, 3, 4, 5}, b[] = {9, 1, 2, 3, 4, 6};
```

```
  expect_bool(iarray_equal(a, b, 5), false, "01234 : 91234");
  expect_bool(iarray_equal(a+1, b+1, 5), false, "12345 : 12346");
  expect_bool(iarray_equal(a+1, b+1, 4), true, "1234 : 1234");
  expect_bool(iarray_equal(a+1, b+1, 3), true, "123 : 123");
  expect_bool(iarray_equal(a, b, 0), true, "[] : []");
  return 0;
}
```

単体テストとテストケース

新たに作った関数expect_boolというのは、テストしようとする関数の結果型がboolのとき使うもので、結果値b1と想定される結果b2が等しければOK、そうでなければNGと表示し、さらに2つの値とメッセージ(どのテストケースかが分かるような文字列)を表示します。ついでなので、整数用のexpect_intと実数用のexptct_doubleも示しておきます。

```
void expect_int(int i1, int i2, char *msg) {
  printf("%s %d:%d %s\n", (i1==i2)?"OK":"NG", i1, i2, msg);
}
void expect_double(double d1, double d2, char *msg) {
  printf("%s %g:%g %s\n", (d1==d2)?"OK":"NG", d1, d2, msg);
}
```

単体テストとテストケース

話題を戻すと、ここでは先頭と最後の値が違う2つの配列を用意して、先頭や末尾を含んだり含まなかったりするさまざまな範囲でiarray_equalを呼び、結果を想定正解と比べてチェックします。実行例を見ましょう。

```
% ./a.out
OK false:false 01234 : 91234
OK false:false 12345 : 12346
NG false:true 1234 : 1234
OK true:true 123 : 123
NG false:true [] : []
%
```

単体テストとテストケース

こうして見るとやはり、「等しいはずなのに等しくないという結果」があります。「1234 : 1234」は等しくなくて、「123 : 123」は等しいというのは、指定された範囲の先まで比べているからかもと思えます。長さ0の「[] : []」も等しくないという結果なのでその予想は正しそうです。そこでコードを熟読すると、`iarray_equal`の中のfor文の条件「`i <= n`」は正しくなく、「`i < n`」であるべきだとわかります。

どうでしょう? そんな面倒なことをしなくてもよく見たらバグは見付かる? この程度ならそうでしょうけれど、複雑なコードの中のどこかで`iarray_equal`を利用しているとしたら、そのコード全体からバグを探すのは大変です。だから単体テストを用意してそれぞれの部品をチェックしておくことは大切なのです。さらに、どの部品も将来的に機能を追加・修正する可能性があります。そのときに「壊れて」いないかどうか確認するために、このテストケースをとっておいて再度実行するべきです。これを**回帰テスト** (regression test) と呼びます。

単体テストとテストケース

この後の演習で配列を結果とするものに対して単体テストを書くため、expect_iarrayも示します。2つの配列とサイズとメッセージを受け取り、中では(もちろんバグを修正した)iarray_equalを下請けに使います。

```
bool iarray_equal(int *a, int *b, int n) {
  for(int i = 0; i < n; ++i) {
    if(a[i] != b[i]) { return false; }
  }
  return true;
}
void expect_iarray(int *a, int *b, int n, char *msg) {
  printf("%s %s\n", iarray_equal(a, b, n)?"OK":"NG", msg);
  iarray_print(a, n); iarray_print(b, n);
}
```

単体テストとテストケース

演習 1 array_equal.c、test_array_equal.cの例題を動かして動作を確認しなさい。納得したら以下のものを作ってみなさい。必ず単体テストを動かし、またテストケースは増やしてみること。

a. 配列の最大値を求める関数`int iarray_max(int *a, int n)`を作成する。

```
int a[] = {9,0,0,1,2,3}, b[] = {-1,-3,-2,-4,-1};
expect_int(iarray_max(a, 6), 9, "9 0 0 1 2 3");
expect_int(iarray_max(a+1, 5), 3, "0 0 1 2 3");
expect_int(iarray_max(b, 5), -1, "-1 -3 -2 -4 -1");
```

b. 配列の並び順を逆順にする関数`void iarray_revese(int *a, int n)`を作成する。

```
int a[] = {8,5,2,4,1}, b[] = {1,4,2,5,8};
iarray_reverse(a, 5); expect_iarray(a, b, 5, "85241 -> 14258");
```

単体テストとテストケース

c. 何らかの整列アルゴリズムで配列を昇順に整列する関数`void iarray_sort(int *a, int n)` を作成する。

```
int a[] = {8,5,2,4,1}, b[] = {1,2,4,5,8};
iarray_sort(a, 5); expect_iarray(a, b, 5, "85241 -> 12458");
```

d. 2 つの配列 (長さは同じ) を受け取り、2 番目の各要素の値を 1 番目の配列の各要素に足し込む関数 `void iarray_add(int *a, int *b, int n)` を作成する。

```
int a[] = {8,5,2,4,1}, b[] = {1,1,2,2,3}, c[] = {9,6,4,6,4};
iarray_add(a, b, 5); expect_iarray(a, c, 5, "85241+11223 -> 96464");
```

単体テストとテストケース

e. 配列に加えて整数2つを受け取り1つを返す関数へのポインタを受け取り、それを用いて配列の値を集約した結果を返す関数`int iarray_inject(int *a, int n, int (*fp)(int, int))`を作成する。さらにそれを用いて、「配列の合計値」「配列の最大値」を求める。[2]

```
int a[] = {8,5,2,4,1};
expect_int(iarray_inject(a, 5, iadd), 20, "8+5+2+4+1");
expect_int(iarray_inject(a, 5, imax), 8, "max(8,5,2,4,1)");
```

[2] inject(注入) とは、たとえば列「1,2,3」に対して「+」を注入するといった場合、個々の要素の間に「+」を挿入するような意味です。その結果は「1+2+3」となり、合計を意味します。

単体テストとテストケース

最後の設問には新しい概念が出て来ているので説明しておきます。C 言語では変数のポインタのほかに**関数のポインタ** (function pointer) も扱うことができます。たとえば次のような感じです。

```
int iadd(int x, int y) { return x + y; }
int imax(int x, int y) { return (x > y) ? x : y; }
int (*fp)(int,int);
fp = iadd;  // or fp = imax;
```

変数 fp の型は何でしょうか。先に説明した宣言の読み方を援用すると、「fp を間接参照して、さらに整数を 2 つパラメタとして渡して呼び出すと、整数になる」型、つまり「整数を 2 つ受け取り 1 つ返す関数へのポインタ」型となります。呼び出す時は「(*fp)(1, 2)」のように間接参照が先に実行されるようかっこで囲む必要があります。あと、関数ポインタの変数への代入時には「&」が不要なことに注意してください。単独の関数名はそれ自体が「関数へのポインタ」を表します。

動的メモリ管理と構造体の情報隠蔽

mallocとfreeによる動的メモリ管理

ここまでに学んだ内容だと、配列の大きさは配列を宣言する時に定数で指定し、実行時には変更できません。たいていのプログラムでは扱うデータの量が分かるのでそれにいくらか余裕を持たせた大きさで宣言すればよいのですが、配列を多数使うような場合、全部に余裕を持たせていると「使わない無駄」が多かったりして好ましくありません。

そこで別の方法として、「使う時にサイズを指定してメモリを割り当てる」「使い終わったら返却する」という機能を活用するやり方があります。これを「**動的メモリ管理**(dynamic memory management)」と呼び、Cの標準ライブラリにはこのために、次の2つの関数が含まれています。これらは(関連する型も含めて)`stdlib.h`で宣言されています。

- `void *malloc(size_t size);` — 領域の**割り当て**(allocation)
- `void free(void *ptr);` — 領域の**返却**(deallocaiton)

mallocとfreeによる動的メモリ管理

見慣れない型が出てきますが、まず**void***というのは「何らかのポインタ」を表す型で、実際に使う時は`int*`など必要なポインタ型にキャストして使います。次に**size_t**というのは「メモリのサイズを扱う時に使う整数型」で、システムの必要に応じて`unsigned int`や`unsigned long`に対応させられますが、使う時は要するに整数を渡すと思っておけばよいです。

たとえば、1000要素の実数配列を使いたい時には次のようにすればよいわけです。

```
double *a = (double*)malloc(1000 * sizeof(double)); // 割り当て
... a[i] を使用する ...
free(a);  // 返却
```

sizeofは任意のデータ型の1要素のバイト数を指定するのに使えます。要素数はここでは1000としていますが、実行時にいくつ必要か計算して、その値で割り当てるのが実際の使い方なわけです。

mallocとfreeによる動的メモリ管理

ところで、freeで返却しなかったらどうなるの、という疑問があるかも知れません。freeで返却した領域は、後で別のmalloc呼び出しがあったときに再度利用されます。ですから、長時間動くプログラムに「freeし損ない」が含まれていると、徐々に使わないメモリ領域がふくらんできて性能低下する原因になります。これを「**メモリリーク** (memory leak)」と呼びます。

ただし、プログラムがもう終わるというところではmallocももう使わないわけですから、freeしない、という流儀もあり得ます。また、今日のシステムではメモリは潤沢にあるので、長時間動くプログラムでなく、あまり大量のデータを使わないなら、途中で一切freeしないという方針でも動くでしょう。

そして、込み入ったプログラムだとどれとどれを使っていてどこで使わなくなるのかを把握するのが難しいこともあります。その結果、間違って(つまり実はまだ使うのに)freeしてしまう)と、その領域を他の部分で書き換えることになり、原因の分かりにくいバグになります。

mallocとfreeによる動的メモリ管理

このようにfreeには多くの問題があるので、現在は使わなくなった領域を自動的に把握して回収する「**ごみ集め** (garbage colleciton)」機能を搭載し、freeを使わない言語が主流です。

今回は、行儀よく「自分で割り当てたものは自分で解法する」流儀で記述していますが、次回からは簡単のため「プログラムが終了する際には開放は省略する」方針を採ります。ただし、ずっと動き続けるようなプログラムで割り当てを繰り返す場合は、解放も必要である (そうしないとメモリが不足する可能性がある) ことを覚えておいてください。

可変長配列

malloc を使って実行時に自由な長さの領域を割り当てられるのはいいのですが、割り当てた領域の長さは自分で把握している必要があります。その繁雑さを避けるのに、長さも一緒に記録しておく方法があります。とくに整数の配列なら、図 4 のように「先頭にデータが何個入っているか記録しておく」ことができます。そして、長さ 2 と 3 の列をくっつけて長さ 5 の列を作り出す、みたいなことができるようにするわけです。

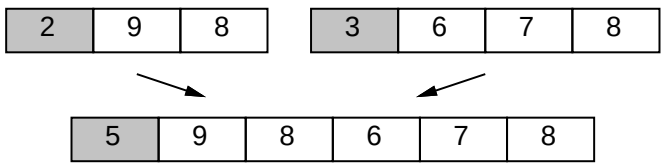

図 4: 先頭に長さを入れた整数の列

実際にやってみましょう。指定した長さの列を作る ivec_new、列の内容を読み込む ivec_read、内容を打ち出す ivec_print、そして 2 つの列をくっつける ivec_concat を作成し、main からこれらを呼び出します。

可変長配列

```
// ivec_demo.c --- int vector demonstration.
#include <stdio.h>
#include <stdlib.h>
void iarray_read(int *a, int n) {
  for(int i = 0; i < n; ++i) {
    printf("%d> ", i+1); scanf("%d", a+i);
  }
}
void iarray_print(int *a, int n) {
  for(int i = 0; i < n; ++i) { printf(" %2d", a[i]); }
  printf("\n");
}
int *ivec_new(int size) {
  int *a = (int*)malloc((size+1) * sizeof(int));
  a[0] = size; return a;
}
```

```
void ivec_read(int *a) { iarray_read(a+1, a[0]); }
void ivec_print(int *a) { iarray_print(a+1, a[0]); }
int *ivec_concat(int *a, int *b) {
  int *c = ivec_new(a[0]+b[0]);
  for(int i = 1; i <= a[0]; ++i) { c[i] = a[i]; }
  for(int i = 1; i <= b[0]; ++i) { c[i + a[0]] = b[i]; }
  return c;
}
int main(void) {
  int *a, *b, *c;
  a = ivec_new(3); ivec_read(a);
  b = ivec_new(2); ivec_read(b);
  c = ivec_concat(b, a); ivec_print(c);
  free(a); free(b); free(c);
  return 0;
}
```

可変長配列

参考までに、ivec_concatの単体テストも示しておきます。

```
// test_ivec_concat.c --- unit test for ivec_concat.
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
(ivec_new, ivec_concat, iarray_equal, iarray_print, expect_iarray)
int main(void) {
  int a[] = {3,1,2,3}, b[] = {2,4,5}, c[] = {5,1,2,3,4,5};
  int *p = ivec_concat(a, b);
  expect_iarray(p, c, 6, "[1,2,3]+[4,5]=[1,2,3,4,5]");
  return 0;
}
```

可変長配列

演習 2 上の例題をそのまま動かせ。長さを変更して動かしてみてもよい。その後、次の関数を作ってみよ。単体テストも作ること (どのようにテストするかは自分で決めてよい)。

a. 2 つの列を受け取り、両方の列の内容を交互に並べた列を返す (例: 「1, 2, 3」「4, 5, 6」→「1, 4, 2, 5, 3, 6」。長さが異なる場合の扱いは好きに決めてよい)。

b. 1 つの列を受け取り、その内容を逆順にした列を返す (例: 「1, 2, 3」→「3, 2, 1」)。

c. 1 つの列を受け取り、その内容を昇順に整列した列を返す (例: 「3, 1, 4」→「1, 3, 4」)。

d. 2 つの昇順に整列ずみの列を受け取り、両方の列をマージした昇順の列を返す (例: 「3, 5, 9」「1, 4, 6」→「1, 3, 4, 5, 6, 9」)。

e. その他自分が面白いと思う列の操作を行なう。

構造体による情報隠蔽

情報隠蔽 (information hiding、encapsulation) とは、ひとまとまりの機能を実装するデータ構造を実装するコードだけから参照可能にし、それ以外からは見えなくすることを言います。

Ruby などクラス機能を持つ言語ではこれはクラスを使うことで自然に実現できますが、C 言語では工夫して実現する必要があります。前回の `eps.c` のように、ファイルを分けてファイル内だけの変数 (**static** 変数) を使うことも 1 つの方法ですが、その変数群が 1 セットしか用意できないという弱点があります。ここではその弱点がない方法として、次の方法を使います。

(1) データ構造を表す変数群一式が 1 つの構造体 (レコード) のフィールドになるような構造体を定義し、それを用いて実装する。データ一式ごとの領域を動的に割り付ける。

(2) 外部から API を呼び出すためのヘッダファイルには構造体定義は書かず、API の関数群には構造体のポインタを渡す。

(3) 外部のファイルはヘッダファイルを取り込み、API の関数を呼び出すことで機能を使う。

構造体による情報隠蔽

具体例がないと分からないと思うので、「等差数列の項を次々に取り出して来る」機能を作ってみましょう。API を定義するヘッダファイルを次のようにします (cdseq.h)。

```
// cdseq.h --- constant difference sequence API.
struct cdseq *cdseq_new(int s, int d);
int cdseq_get(struct cdseq *r);
void cdseq_free(struct cdseq *r);
```

構造体の定義が書かれていなくても、構造体のポインタ型は自由に使うことができます。それは、ポインタのビット数は指す先がどのような型であっても同じだからです (メモリアドレスなので当然ですが)。

これを用いて「1 から始まる階差 2 の等差数列と、0 から始まる階差 3 の数列を 2:1 で混ぜて 15 個出力する」プログラムを作ってみます。

構造体による情報隠蔽

```
// cdseq_demo.c -- cdseq demonstration.
#include <stdio.h>
#include "cdseq.h"
int main(void) {
  struct cdseq *s1 = cdseq_new(1, 2);
  struct cdseq *s2 = cdseq_new(0, 3);
  int i;
  for(i = 0; i < 6; ++i) {
    printf(" %2d", cdseq_get(s1));
    printf(" %2d", cdseq_get(s1));
    printf(" %2d", cdseq_get(s2));
  }
  printf("\n"); cdseq_free(s1); cdseq_free(s2);
  return 0;
}
```

構造体による情報隠蔽

動かしたようすは次の通り。

```
% ./a.out
1  3  0  5  7  3  9 11  6 13 15  9 17 19 12 21 23 15
```

構造体による情報隠蔽

では、実装部分 cdseq.c を見ていただきます。

```
// cdseq.c -- cdseq implementation.
#include <stdlib.h>
#include "cdseq.h"
struct cdseq { int value, diff; };
struct cdseq *cdseq_new(int s, int d) {
  struct cdseq *r = (struct cdseq*)malloc(sizeof(struct cdseq));
  r->value = s; r->diff = d; return r;
}
int cdseq_get(struct cdseq *r) {
  int v = r->value; r->value += r->diff; return v;
}
void cdseq_free(struct cdseq *r) {
  free(r);
}
```

構造体による情報隠蔽

最初に構造体の定義があります。等差数列なので、初項と公差を覚えることとしています。

cdseq_newでは、まず構造体の領域を割り当てます。本当は返されたポインタ値がNULLだったらメモリ不足のエラーなのですが、まず起きないので当面省略します。つぎに、返された構造体の領域の初項と公差に値を入れ、その後ポインタ値を返します。ここで使っているアロー演算子について復習しておきます。

```
struct cdseq s; s.value = 1; s.diff = 3;  // 通常変数
struct cdseq *r = &s;
  (*r).value = 1; (*r).diff = 3;  // ポインタ
  r->value = 1; r->diff = 3; // アロー演算子
```

上の1行目のように、構造体のフィールドは「s.value」のように「.」に続けてフィールド名を指定しますが、もし通常変数ではなくポインタだったとすると、2行目のように「(*r).value」と、まず間接参照する必要があります。C言語ではこれをよく書くため、もっと見た目がよいように同じことを「r->value」と書いてもよくなっています。

構造体による情報隠蔽

さて cdseq_get ですが、「次の値」は value フィールドにあるのでそれを変数 v に保存します。そして、value フィールドは diff だけ増やすことで、次の値を用意しておきます。最後に保存してあった v を返します。最初に return v; としたくなりますが、return するとその後の文は一切実行されないので、このように一旦覚えておき return は最後にする必要があります。

最後の cdseq_free ですが、これはレコードのポインタを free するだけです。だったら呼ぶ側で直接 free を呼べばよい? この場合はそれでもよいですが、レコードの中にさらに動的メモリ割り付けした結果のポインタを保持したいときは、それも返却しなければなりませんから、このようにそれぞれの構造ごとに後始末の関数を用意するほうがよいのです。

構造体による情報隠蔽

こちらも課題の前に単体テストの例を示します。今度は複数回 get を呼ぶごとに値をそれぞれテストしていることに注意。

```
// test_cdseq_1.c --- unit test for cdseq.
#include <stdio.h>
#include "cdseq.h"
void expect_int(int i1, int i2, char *msg) {
  printf("%s %d:%d %s\n", (i1==i2)?"OK":"NG", i1, i2, msg);
}
int main(void) {
  struct cdseq *s = cdseq_new(2, 3);
  expect_int(cdseq_get(s), 2, "2+3*0 = 2");
  expect_int(cdseq_get(s), 5, "2+3*1 = 5");
  expect_int(cdseq_get(s), 8, "2+3*2 = 8");
  return 0;
}
```

構造体による情報隠蔽

演習 3 上の等差数列生成 API の例題をそのまま動かせ。動いたら次のような機能を同様のやりかたで実現してみよ。いずれも単体テストを作ること (単体テストのやりかたは自分で決めてよい)。

a. 上の `cdseq.c` に、等差数列を初項に戻す機能 `cdseq_reset` を追加してみよ。

b. 上の例では `get` を呼ぶたびに次の値が出て来たが、`get` だけでは値が進まず、`cdseq_fwd` を呼ぶと次の値に進むように変更してみよ。さらに、現在が何番目 (最初が 0 とする) の項かを返す機能 `cdseq_num` を追加してみよ。

c. 初項と公比を与えて等比数列 (実数値) を生成するような機能を実現してみよ。詳細は好きに決めてよい。

d. 複数の値を `put` でき、いつの時点でもそれまでの数値のうちの最大値と最小値が `get` できるような機能を実現してみよ。詳細は好きに決めてよい。

e. その他、構造体と情報隠蔽のしくみを使って、何か面白いと思う機能を実現してみよ。

トレーニング課題

2019年度の実施で、極めて基本的なC言語プログラムでの練習が必要な学生さん「も」いるということが明らかになったので、今年度から「トレーニング課題」を追加します。トレーニング課題は基本練習の易しい課題で、「トレーニング課題10個で」通常の課題1個の代わりとして提出することができます。B課題は通常の課題2個でも、通常1個トレーニング10個でも、トレーニング20個でも構いません。A課題のレビュー課題はトレーニング課題以外を用いてください。もしトレーニング課題のみで出す場合は、レビュー課題は10個見てもらってください。

また、トレーニング課題であっても、各々に単体テストは必須ですので注意してください。

トレーニング課題

演習 T4 整数配列 a とそのサイズ n、値 v を受け取り、先頭から n 要素を v に設定する関数 `void iarray_fill(int a[], int n, int v)` を作り単体テストする例題を示す。

```
#include <stdio.h>
#include <stdbool.h>
void iarray_fill(int a[], int n, int v) {
  for(int i = 0; i < n; ++i) { a[i] = v; }
}
(iarray_print, iarray_equal, expect_iarray here)
int main(void) {
  int a[] = { 0,1,2,3,4,5 }, b[] = { 6,6,6,3,4,5 };
  iarray_fill(a, 3, 6);
  iarray_print(a, 6);
  expect_iarray(a, b, 6, "[0,1,2,3,4,5] -> [6,6,6,3,4,5]");
  iarray_fill(a+3, 0, 7);
```

```
  iarray_print(a+3, 3);
  expect_iarray(a+3, b+3, 3, "[3,4,5] -> [3,4,5]");
  return 0;
}
```

これを参考に(しなくてもよいが)、次の関数を作り単体テストせよ。関数の名前は好きに決めてよい。パラメタ a, n, v は例題と同じとする。

a. 先頭から n 要素を v 増やす。
b. 先頭から n 要素について、値が 0 だったときそれを v に変更する。
c. 先頭から n 要素について、値が v に等しかったきそれを 0 に変更する。
d. 先頭から n 要素について、値が v より大きかったらそれを v に変更する。
e. 先頭から n 要素について、値が v より小さかったらそれを 1 減らす。
f. 先頭から n 要素について、絶対値が v より大きかったらそれを 0 にする。
g. 先頭から n 要素について、絶対値が v より小さかったら符号を反転する。

トレーニング課題

演習 T5 整数配列 a, b とサイズ n を受け取り、a の内容を反転して (逆順にして)b にコピーする関数`void iarray_revcopy(int a[], int b[], int n)`を作り単体テストする例題を示す。

```
#include <stdio.h>
#include <stdbool.h>
void iarray_revcopy(int a[], int b[], int n) {
  for(int i = 0; i < n; ++i) { b[i] = a[n-i-1]; }
}
(iarray_print, iarray_equal, expect_iarray here)
int main(void) {
  int a[] = { 0,1,2,3,4,5 }, b[6], c[] = { 5,4,3,2,1,0 };
  iarray_revcopy(a, b, 6);
  iarray_print(b, 6);
  expect_iarray(b, c, 6, "[0,1,2,3,4,5] -> [5,4,3,2,1,0]");
  iarray_revcopy(a, b, 2);
```

```
  iarray_print(b, 2);
  expect_iarray(b, c+4, 2, "[0,1] -> [1,0]");
  return 0;
}
```

これを参考に(しなくてもよいが)、次の関数を作り単体テストせよ。関数の名前は好きに決めてよい。パラメタa, b, nは例題と同じとする。指定されていないbの内容は変更されないままにすること。nは偶数であるものとしてよい。

a. aの前半をbの後半、aの後半をbの前半にコピー。
b. aの前半をbの後半に逆順でコピー。
c. aの後半をbの後半に逆順でコピー。
d. aの前半をbの0、2、4、…番に1つおきにコピー。
e. aの前半をbの1、3、5…番、後半を0、2、4…番にコピー。
f. aの0番目と1番目の和、2番目と3番目の和…をbの後半にコピー。

本日の課題 2a

「演習 1」～「演習 3」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. プログラムどれか 1 つのソースと「簡単な」説明。単体テストと実行例を必ず含めること。
3. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
4. 以下のアンケートの回答。

   Q1. アドレス、ポインタについて納得しましたか。

   Q2. 「構造体を用いた情報隠蔽」について納得しましたか。

   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 2b

「演習 1」～「演習 3」(ただし 2a で提出したものは除外、以後も同様) の小課題全体から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ演習 2 からも選ぶこと。レポートは次回授業前日 23:59 を期限とします。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。単体テストと実行例を必ず含めること。
3. 2 つ目の課題についても同様。

次回までの課題 2b

4. 以下のアンケートの回答。

   Q1. アドレス、ポインタを使うプログラムで注意すべきことは何だと思いますか。

   Q2. ここまでのところで、プログラムを作るときに重要だが自分で身に付いていないと思うことは何ですか。

   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。