2020年度「プログラミング言語」配布資料 (5)

五十嵐 淳

2022年10月02日

1 C言語

C言語はJavaやOCamlに比べると，プログラムが実行される環境 (ハードウェアやオペレーティングシステム) をより強く意識する必要がある，という意味で**低水準 (low-level)** な言語であると言われる．データのメモリ上の配置について非常に細かく制御できる一方で，プログラムが読み書きすべきでないメモリ領域を読み書きするようなプログラムが容易に書けてしまうという意味でメモリ操作の安全性は低いとされる．一方で，アセンブリ言語に比べれば関数や繰り返しの構文が備わっていたり，ハードウェア間の差異が抽象化されており，低水準ながら移植性が高いプログラムを記述することができる．

1.1 典型的なCプログラムの構造

Cプログラムは，おおまかにいうと，

1. ライブラリ関数を使うための `#include` による**ヘッダファイル** の読み込み．
2. 関数定義と **関数プロトタイプ宣言** の列
3. プログラム実行の開始点となる `main` 関数の定義

で構成される．

以下は，階乗を計算する関数 `fact` を定義し，5の階乗をライブラリ関数 `printf` を使ってディスプレイに出力するためのプログラムである．

```
// C/samples/fact.c

#include <stdio.h>  // for printf

int fact(int n) {
  if (n == 1) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

```
int main(void) {
  printf("fact(5) = %d.\n", fact(5));

  return 0;
}
```

Java と同じように，// 以降行末まで，もしくは，/* と*/ に挟まれた部分はコメントになる．3 行目が，14 行目で呼び出されている printf というライブラリ関数を使うための include ディレクティブ，5〜11 行目が階乗関数 fact の定義，13〜15 行目が main 関数の定義である．構文が Java のメソッドとよく似ていることがわかるだろう[1]．

1.1.1 #include ディレクティブ

#include は，他のファイルに書かれたプログラムを読み込むための記法で，ファイルの内容がそこに展開されたのとほぼ同じ効果が得られる．通常の C プログラムにはファイル名に拡張子.c がつけられるのに対し，拡張子が .h であるファイルはヘッダファイルと呼ばれる．ヘッダファイルには特定のライブラリ関数を使うためのその関数の型の宣言や関連する様々な定数や型の定義が書かれている．

ライブラリ関数のドキュメントにはどのヘッダファイルを include すればよいかが書かれている．stdio.h は standard input and output の略で，入出力に関する関数を使う際に読みこむ．

> # で始まる行 (のほとんど) は，C 言語 (や C++言語) 特有の，**プリプロセッシング (preprocessing)** と呼ばれる前処理で変換される．この処理結果は C コンパイラのコマンドに-E オプションをつけて実行することで見ることができる．

1.1.2 関数定義

関数 fact の定義は Java に慣れた者であれば容易に読めるだろう．(ただし，真の (？)C プログラマは再帰を使わずに繰り返しを使って定義するところだろう．)

C 言語では，ファイルの後ろの方に書かれた関数を何もせずに呼び出す (ちなみにこのようなファイルの後の方に書かれている定義を参照することを (大変紛らわしいが) **前方参照 (forward reference)** という) と，コンパイラに警告される．例えば，main と fact の順番を変更してコンパイルすると，

```
fact.c:4:28: warning: implicit declaration of function 'fact' is invalid in C99
      [-Wimplicit-function-declaration]
  printf("fact(5) = %d\n", fact(5));
                           ^
1 warning generated.
```

これは警告なので，コンパイラは fact を「引数は int ひとつで int を返す」関数だと仮定して処理を進める．後で fact が全く違う型の関数で定義されていたと場合エラーにはなるが，かなり紛らわしい．

ファイルの後の方で定義される関数を呼ぶためには関数の型情報 (返り値型，関数名，引数の型) だけを先に宣言しておく．これを**プロトタイプ宣言 (prototype declaration)** という．

[1] もちろん Java が後発なので，Java が C に似せて作られているのである．

```
int fact(int);
```

を main の前に宣言しておけば，コンパイラも fact が，引数として整数をひとつ取り，整数を返す関数だと理解してコンパイル処理を進めることができる．この例では，単に main を後に置けば済む話だが，例えば複数の関数同士が相互に参照する場合にはプロトタイプ宣言は必須である．

1.1.3 真偽値

C の真偽値は実質的に整数である．両辺が等しいかを比較する == は比較結果が等しければ 1 を，等しくなければ 0 を返すものの，if や while などの条件判定は 0 を偽として扱い，**それ以外の値全て**を真として扱う．この「0 以外なんでも真」であることに依存して書かれたプログラムも多い．比較結果は int 型の整数に格納することもできるが，C99 では，0 と 1 だけが格納できる (1 より大きい値も全て 1 につぶれてしまう) bool 型も (#include <stdbool.h> を書くことによって) 使うことができる．stdbool.h を読み込むと，bool 型だけでなく，true，false という定数も同時に (1, 0 の別名として) 定義される．

1.1.4 main 関数

先に述べたように main 関数は，C プログラムの実行開始地点を示す．プログラム起動時に外から情報をもらわない場合，引数部分には無引数を表す (void) を書く．(これは他の無引数関数も同様である．) プログラムを実行する際のコマンドラインで与えられる引数を文字列としてもらうこともできるが，ここでは扱わない．また返値の型は必ず int である．この返り値を使って，プログラム呼び出し側に，プログラム終了時に「終了ステータス」と呼ばれる情報を返すことができる[2]．(慣習的に正常終了時は 0，異常終了時には 0 以外を返す．)

1.1.5 printf 関数

printf 関数は文字列やデータを整形して (printf の f は format に由来すると思われる) 出力するための関数である．基本的には，第一引数の文字列 (二重引用符で囲まれた部分) を出力するが，% がついている部分は，format directive と呼ばれ，そのまま出力されず，第二引数以降のデータを適宜文字列化して埋め込んで出力する．このプログラムの%d は整数を十進表記 (decimal の d) で整形する，という意味である．format directive には様々な種類があるがここではふれない．

末尾の \n は「バックスラッシュとエヌ」ではなく改行を表す文字の特殊表記である．このようなソースコード上に直接表記できない文字を示すための表記を**エスケープシーケンス (escape sequence)** という．

1.2 構造体

C 言語で OCaml のレコードに相当するものが **構造体 (struct)** である．以下は OCaml で示した 2 次元座標の点の構造体 point，中点を計算する関数 middle を定義し，原点と (3,8) の中点を計算している．

```
// C/samples/struct.c

#include <stdio.h>
```

[2] Java では main メソッドの返値型は void で何も返せない．

```
struct point {
  int x;
  int y;
};

struct point middle(struct point p1, struct point p2) {
  struct point result = {(p1.x + p2.x) / 2, (p1.y + p2.y) / 2};
  return result;
}

int main(void) {
  struct point origin = {0, 0};
  struct point p;
  p.x = 3;
  p.y = 8;

  struct point m = middle(origin, p);

  printf("The middle point between (%d, %d) and (%d, %d) is (%d, %d)\n", origin.x, origin.y, p.x, p.y,

  return 0;
}
```

5〜8 行目が構造体 point の定義である．point は x という名前で読み書きできる整数，y という名前で読み書きできる整数が，この順序で並んだようなデータであることを意味している．ここで宣言される名前 point は **構造体タグ** と呼ばれ，struct と組み合わせることで型の名前として使うことができる．また内部のデータ x, y は (構造体の) **メンバ (member)** と呼ばれる．

構造体のメンバ x にアクセスするためには .x という記法を使う．p1 は構造体を格納した変数であり，このメンバ x は p1.x でアクセスすることができる．p1.x の型は point の定義を見ることで知ることができる．

10〜13 行目が中点を計算する関数 middle である．構造体 struct point の引数をふたつ取り，struct point を返すものとして定義されている．struct point という，struct + 構造体タグ名の表記は，このように関数パラメータや，局所変数の型として使うことができる．変数 result の宣言の右辺は，構造体 (や複数の値から構成されるデータ) を初期化するための表記で，中括弧の中にデータを並べる．この際，(OCaml のレコードと違って) メンバの名前は指定できず，定義の時の順序に従うので，並べる順序が重要である．

初期化構文を使わずに初期化する場合は，18,19 行目のようにメンバアクセスの記法を使って代入を行うことができる．(OCaml とは表記が違うので注意されたし．)

ここで，少しメモリ管理についての説明をしておこう．C 言語でも Java や OCaml と同様，関数呼び出しの際にはフレーム [3]がスタック領域に確保され，その中に関数パラメータや局所変数のための領域が用意される．このようにスタック内に領域が用意される変数を C 言語用語で **auto 変数**と呼ぶ．これは，領域の管理 (確保と解放) が自動的に行われることからこのような名前がついている．これに対し，局所変数宣言に static と

[3]関数呼出しの処理を行うために確保される，関数のパラメータや呼出し元 (戻り先) の情報を記憶するための領域.

いう修飾子をつけると (関数パラメータにはつけることができない)，この変数はスタックとは別の領域に確保される．static 変数の領域はプログラムの実行を通じて固定されているので，それを含む関数の複数の呼び出し間で変数が共有されることになる．例えば，以下の関数 foo を考えてみよう．

```
// C/samples/static.c

int foo(void) {
   static int x = 0;
   x++;
   return x;
}

int main(void) {
   printf("%d\n", foo());
   printf("%d\n", foo());
   printf("%d\n", foo());
}
```

この変数 x はプログラム開始時に 0 に初期化され，foo が呼び出される度に 1 増えていくことになる．(= 0 は最初に一回行われるだけであることに注意．) そのため，このプログラムを実行すると 1 2 3 が順に表示される．一方，static がない場合，初期化は関数が呼び出される度に行われるので，foo() は常に 1 になる．

auto 変数 (関数パラメータ・局所変数) を宣言すると，その型に応じたサイズのメモリ領域が，その関数呼び出しのフレーム中に確保される．int のサイズはハードウェアによって違う可能性があるが，おそらく 4 バイト，struct point は int がふたつ分なので (おそらく) 8 バイトである．origin の初期化や p への代入に際しては，この領域に整数が書き込まれる．middle を呼び出す際には，origin と p に格納されたデータが，その呼び出しフレーム中に確保された p1 と p2 のための領域へとコピーされる．また，return に際しても，変数 result の領域のデータが呼び出し元へとコピーされる．このように，関数を使って構造体をやりとりしてすると，構造体のメンバを構成するデータが全てコピーされる．このコピーには当然時間的にも空間的にもオーバーヘッドが生じるため，(構造体が大きい場合には特に) 避けることも多い．大きなデータ構造のコピーを避けて計算するには—というより，C 言語を理解するには—**ポインタ (pointer)** の概念を理解することが非常に重要である[4]．

1.3 ポインタ

C 言語のポインタを理解するためのいくつかの用語を導入したい．

オブジェクト 連続したメモリ領域のこと．
識別子 型やオブジェクト，構造体などにつけられる名前，もしくは名前として使うことのできる文字列・記号列のこと．
変数 識別子によって名付けられたオブジェクト
左辺値 (L-value) 変数が指すオブジェクトの先頭アドレス

[4] C 言語の構造体と Java のオブジェクトはデータの集まりである点において似ている．しかし，Java オブジェクトの (インスタンス変数の) データは変数のためのメモリ領域とは別の領域に割り当てられていて，変数にはその領域 (オブジェクト) への参照が格納されているだけなので，引数でオブジェクトを渡しているように見えても参照が渡されているだけでオーバーヘッドは小さい．

右辺値 (R-value) 変数が指すオブジェクトに格納されたデータ

`int x;` といった変数宣言は，正確にいうと，メモリに `int` を格納するための領域 (オブジェクト) を確保して，`x` という識別子をその名前とする作業と考えることができる．

構造体のメンバアクセス `p1.y` のような表現は，オブジェクト `p1` の後半部分のオブジェクトを示すためのものである．オブジェクトは使われる文脈によって，その (先頭) アドレスまたは格納されたデータとして解釈される．例えば，

```
x = x + 1;
```

という代入文は「変数 `x` に `x + 1` を代入する」と説明してきたが，より正確には，「`x` という名前がつけられたオブジェクトの先頭アドレスに (オブジェクト `x` から整数データを読み出しそれに 1 を足した整数) を格納せよ」という意味である．変数は，代入文の左辺ではオブジェクトの先頭アドレスとして，右辺ではオブジェクトに格納されたデータとして解釈されるため，それぞれオブジェクトの左辺値，右辺値と呼ばれる．

1.3.1 アドレス演算子&

前置演算子 `&` はオブジェクトの先頭アドレス (左辺値) を返す演算子[5]である．`&` が適用される対象はオブジェクトなので `&1` のように整数などに直接適用することはできない．

以下は，2 変数 `x, y` を宣言して，そのアドレスを出力するプログラム断片である．(C/samples/pointer1.c)

```
int x = 100;
int y = 200;

printf("x and y are allocated at %p and %p\n", &x, &y);
printf("their sizes are %zd bytes.\n", sizeof(int));
```

実行結果は以下のようになる (アドレスの具体的な値は，OS 毎，さらには同じ OS でも各実行毎に違うだろう)．

```
x and y are allocated at 0x7fff52271978 and 0x7fff52271974
their sizes are 4 bytes.
```

プログラム最後の行に現れる `sizeof(int)` は `sizeof` 演算子と呼ばれ型やオブジェクトを引数としてそのサイズをバイト単位で計算する．`int` のデータのサイズは実行環境に依存するがここでは 4 バイトのようである．`&x` と `&y` はどうやら，連続したメモリ領域に確保されているらしいことがわかる．(アドレスを `printf` で表示するためには `%p` を使い，サイズを表す整数には `%zd` を使う．) また，先に宣言した `x` の方が大きいアドレスに格納されている．

この `&` で取得したアドレスは，データとして変数に格納することができる．この「アドレスを格納する変数」がポインタ変数である．ポインタ変数は名前の前に`*` をつけて宣言する．(が，型としては「整数へのポインタ型」`int *` のように扱うので，「名前の前に」という言い方はベストではないかもしれない．)

```
int *px = &x;   // also written: int* px = &x;
int *py = &y;
```

[5] `&` は中置演算子としても使えるが，これはビット毎の論理積を取る演算子で意味が全く違うので注意すること．

px と py にはそれぞれ x と y のアドレスが格納される．px も py も変数なので，そのアドレスを取得することもできる．

```
  printf("px and py are allocated at %p and %p\n", &px, &py);
  printf("their sizes are %zd bytes.\n", sizeof(int *));
```

実行結果:

```
px and py are allocated at 0x7fff52271968 and 0x7fff52271960
their sizes are 8 bytes.
```

px と py も x と y と同様連続した領域に格納されているようだ．(が，px のアドレスに 8 を足しても，...70 で ...74 ではないので y と px の間には 4 バイト分の空白があるようだ．これは**アラインメント (alignment)** といって，データのサイズに応じて，格納場所のアドレスの下位数ビットを 0 に揃えなければいけないという CPU の制約に由来するものである．プロセッサによっては，ロード命令で複数バイト (例えば 4 バイト) を一度に読み出す際に，アドレスが 4 の倍数でないといけないという制約がある．)

さらにこれらのアドレスを別の変数に格納することもできる (この場合，その変数の型は `int **` になる) がここでは行わないでおく．

さて，アドレスに対しては前置演算子 * を使って，そのアドレスに格納されたオブジェクトを参照することができる．例えば*px は px が指す (変数 px に格納された右辺値であるところのアドレスに格納された) オブジェクトになる．そのため x と *px は同じオブジェクトを表す表現として使うことができる．よって x に代入をした後に *px の (右辺) 値を表示すると，x に代入された値が表示される．*px を代入文の左辺に持ってくることもできる．この場合，*px の指すオブジェクトの内容，すなわち x が書き変わる．

```
  printf("px at %p points to x (%d)\n", px, x);
  x = 300;
  printf("px at %p points to x (%d)\n", px, *px);
  *px = *px + 1;  // reads (*px) + 1
  printf("the value of x is %d\n", x);
/* 表示結果
px at 0x7fff52271978 points to x (100)
px at 0x7fff52271978 points to x (300)
the value of x is 301
*/
```

ちなみに ++ 演算子は変数の値を 1 増やすことができるが，`*px = *px + 1;` の場合 `(*px)++;` と括弧をつける必要がある．括弧を付けないと px の値 (つまりアドレス) を増やしてから，(増やす前の元のアドレスを) * で参照する，という全く違った意味になるので注意すること[6]．(「アドレスを増やす」という意味はこの講義では説明しません．)

1.3.2 構造体へのポインタ (C/samples/pointer2.c)

構造体の場合も，同様にして&演算子で構造体が格納された領域の先頭を指すポインタを取得することができる．

[6]C 言語には，表記を簡潔にするための演算子が沢山あるのだが，習熟するまで優先度を間違えやすく，しかも間違えても (コンパイラがエラーを発するわけでもなく) 単に別の意味になるだけのことが多く，エラーの温床となる．

```
  struct point p1 = {500, 100};
  struct point p2 = {300, 400};

  struct point *pp1 = &p1;
  struct point *pp2 = &p2;

  printf("pp1 at %p points to a point (%d, %d)\n", pp1, p1.x, p1.y);
  printf("pp2 at %p points to a point (%d, %d)\n", pp2, p2.x, p2.y);
  printf("their sizes are %zd bytes\n", sizeof(struct point));
/* 実行結果
pp1 at 0x7fff50488960 points to a point (500, 100)
pp2 at 0x7fff50488958 points to a point (300, 400)
their sizes are 8 bytes
*/
```

構造体 point は int ふたつ分の領域を占めるので，サイズは 8 バイトで，p1 と p2 は連続した領域に配置されていることがわかる．

ポインタを通じて，メンバの読み書きをするには，ポインタの参照を行う* と，メンバを選択する ．演算子を用いればできる (. の方が結合が強いので括弧が必要である):

```
  (*pp1).x = 900;   // parentheses needed
```

が，この演算子の組み合わせパターンは頻出なので，それらを組み合わせた->という演算子も用意されている．

```
  pp2->y = 200;     // abbreviation for (*pp2).y
```

このふたつの代入の結果を見てみよう．

```
  printf("pp1 at %p points to a point (%d, %d)\n", pp1, p1.x, p1.y);
  printf("pp2 at %p points to a point (%d, %d)\n", pp2, p2.x, p2.y);

/* 実行結果
pp1 at 0x7fff50488960 points to a point (900, 100)
pp2 at 0x7fff50488958 points to a point (300, 200)
*/
```

見ての通り，p1.x と p2.y を書き変えることができた．

さて，構造体のメンバも記憶領域の一部を占めているオブジェクトであるので，そのアドレスを& で取得することができる．

```
  int *p = &(pp1->y);  // p points to a middle of the object named p2

  printf("p at %p points to %d\n", p, *p);
/* 実行結果
p at 0x7fff50488964 points to 100
*/
```

このようにして，pp1->y (すなわち p1.y) のアドレスを取得することができる．アドレスを見てみると確かに pp1 と pp2 の真ん中である．言い換えると，この p は構造体の途中を指している．この指している先には p1.y の値である 100 があることがわかる．さらに，このポインタを通じて，p1.y の値を書き換えることもできる．

```
  (*p)++;   // equivalent to *p = *p + 1;  incrementing the content of p
            // Do not confuse with *p++;, which is equal to *(p++);.
            // It increments p and read the value pointed to by p (and discards, in this case).
  printf("pp1 at %p points to a point (%d, %d)\n", pp1, p1.x, p1.y);
  printf("pp2 at %p points to a point (%d, %d)\n", pp2, p2.x, p2.y);

/* 実行結果
pp1 at 0x7fff50488960 points to a point (900, 101)
pp2 at 0x7fff50488958 points to a point (300, 200)
*/
```

実行結果で表示される数値が 101 になっていることに注目してもらいたい．

1.3.3 ポインタ渡し (C/samples/swap.c, C/samples/pointer3.c)

ポインタは，単独の変数で使うだけでなく，関数の引数や返り値でやりとりすることもできるし，ポインタをメンバとして持つ構造体を作ることもできる．

関数呼び出しの際，引数に変数の名前を書いたとしても，関数型に渡されるものは，その変数の (右辺) 値である．既に Java や OCaml の関数呼び出しの動作の説明でふれたように，関数側のパラメータ変数については新たに領域が確保されて，渡された値はその変数の初期値になる．よって，以下の novice_swap のように，関数側でパラメータ変数に代入を行っても，その影響を呼び出し側で観察することはできない．

```
void novice_swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  return;
}

int main(void) {
  int x = 2;
  int y = 3;

  novice_swap(x, y);

  printf("x = %d; y = %d", x, y);  // prints "x = 2, y = 3" not "x = 3, y = 2" as one might expect
}
```

しかし，関数に呼び出し側の変数のアドレスを渡してやることで，呼び出し側 (caller) の変数の値を呼び出され側 (callee) から変更することができる．以下は，2 変数の値を入れ替える「正しい」swap 関数の定義とその使用例である (C/sample/swap.c)．

```
#include <stdio.h>

void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
  return;
}

int main(void) {
  int x = 4;
  int y = 100;

  printf("(x, y) = (%d, %d)\n", x, y);
  swap(&x, &y);
  printf("(x, y) = (%d, %d)\n", x, y);

  return 0;
}
```

この関数 swap は，整数へのポインタを受け取る関数となっていて，そのポインタの指す先を入れ替える動作になっている．一方，呼び出し側では，変数のアドレスを& で取得して渡すことで，swap 側で入れ替えてもらっている．

このように，ポインタを渡して関数側で書き換えてもらう，というのは，関数から複数の値を呼び出し側に返したい・伝えたい場合などに使う C 言語で見られるプログラミングパターンで，C 言語のライブラリ関数でもよく見られる．例えば，以下の関数は与えられた整数 x と y の算術平均と逆数和を計算する関数である．

```
bool bar(int x, int y, double* avg, double* sumrecp) {
  bool result = true;

  *avg = (x + y) / 2.0;

  if (x * y == 0) {
    result = false;
  } else {
    *sumrecp = ((double)x + y) / (x * y);
  }

  return result;
}
```

この関数 bar は，算術平均，逆数和，逆数和の計算が成功したかどうかの 3 種類の情報を返していると考えられる．別の方法として double ふたつと bool ひとつからなる構造体を定義して返り値とすることが考えられるが，その代わりに，double へのポインタを引数として，計算結果をそこに書き込むことで呼び出し側に伝

えている．また，x, y のどちらかが 0 であると逆数和が定義できないが，計算が成功したかどうかを真偽値の返り値としている．(ちなみに (double) は整数を浮動小数点数に変換するためのキャスト (型変換，型強制とも呼ばれる) である．この関数は例えば以下のように呼び出す．

```
int a = ...;
int b = ...;
double c;
double d;

if (bar(a, b, &c, &d)) {
  // bar is supposed to write into c and d
  printf("(a+b)/2 = %f;  1/a + 1/b = %f\n", c, d);
} else {
  printf("Either a or b was zero\n");
}
```

結果を書き込んでもらう変数 c, d を用意した上でそのポインタを bar に渡しているところがポイントである．bar の返り値が false だった場合に備えているのは (逆数和が計算できなかった，という) ある種のエラー処理だと考えられる．

1.3.4 ダングリング・ポインタ，malloc と free 関数

さて，ポインタを関数をまたいでやりとりする時には，ポインタの指す先の領域が確保されているかを常に気にする必要がある．上の swap の例であれば，swap の実行中ずっと，x と y の領域が確保されており，a と b は確保された領域を指している．大丈夫でない典型例は，関数側の局所変数へのポインタを返してしまうような関数である．

```
int *foo(int x) {
  int y = x * 2;
  return &y;
}
```

この関数は，局所変数 y に引数の倍の数を入れて，y のアドレスを返している．しかし，スタック上のフレームに確保された y の領域は foo の実行が終わった途端に解放されてしまうので，呼び出し側で foo の返り値の先を参照するのはまずい (未定義動作[7])．このような，解放された領域を指すポインタを**ダングリング・ポインタ (dangling pointer)** と呼ぶ．

```
  int *p = foo(100);
  printf("*p is %d", *p);  // Illegal!
```

つまり，関数の中で新たにメモリ領域を確保して，その領域へのポインタを呼び出し側に返したい場合に局所変数 (正確には auto 変数) を使うことはできない．(ポインタを返すのではなく return y; とするのであれば，

[7] C 言語において「未定義動作 (undefined behavior)」という単語はきちんと定義された用語で，どんな結果も引き起こしうる，という意味である．つまり，エラーメッセージを出力してプログラムの実行を中断してもよいし，そのまま何かしらの動作を続けてもよい．ほとんどの C コンパイラでは効率を重視する (そもそも未定義状態に陥ったかどうかを検知することが難しい) ため，エラー処理などを行わずに何かしらの動作を続けるような機械語を出力することがほとんどである．プログラムによっては，この未定義時の「何かしら」の動作をプログラムに外部から与えるデータで制御できる場合もあり，極端な場合には，任意のコマンドを実行してファイルを全消去したりすることも可能だったりする．C 言語で書かれたソフトウェアのセキュリティーホールの大半はこれである．

単に変数の中身 (右辺値) が呼び出し元にコピーされて返されるので問題はない．もちろん返り値の型は `int *` ではなく `int` にする必要がある．) このような場合には，標準ライブラリの `malloc` 関数を使って，ヒープ領域と呼ばれるスタックとは別のメモリ領域から領域を確保する必要がある．

`malloc` 関数を使うには`<stdlib.h>` を include する．型は，

```
void *malloc(size_t);
```

というもので，確保したい領域のサイズ (型 `size_t` の値，典型的には `sizeof` がこの型を返す) を渡すと確保した領域へのポインタを返してくれるという関数である．(領域の確保に失敗した場合にはどこの領域も指さない—よって`*` で参照した途端に未定義動作になる—null pointer が返ってくる．) `void *` というのは，特殊なポインタ型で，何を指しているのか不明なポインタに対して使われる．`void *` 型のポインタは，通常，**キャスト (cast)** を使って，その指している (べきものの) 先を明示してから使う．キャストは，(型名) 式の形で，式の値を括弧内に書かれた型に「変換」するための機能である．「変換」には，浮動小数点数 `double` から整数 `int` への変換のように，実際に計算を伴うものと，ポインタ型から別のポインタ型のキャストのように何も計算を伴なわないのものがある．ポインタ型のキャストが計算を伴なわないのは，ポインタは，その先に何を指していようとも表現が同じであるためである．つまり，ポインタ型のキャストは，単にコンパイラに型が変わったことを知らせるだけの，ある種の注釈の役割を果たす．どんな型の間ならキャストができるかは C 言語仕様の 6.3 Conversions に詳しい．

さて，`malloc` に戻ると，型 `T` を格納する領域を確保する時には，以下のように呼び出すのが定石である．

```
(T *)malloc(sizeof(T))
```

`malloc` を使うと，上の `foo` は

```
int *foo(int x) {
  int *p = (int *)malloc(sizeof(int));
  *p = x * 2;
  return p;
}
```

と書き換えることができる．本当に行儀のよいプログラムを書きたかったら，以下のように`*p` にアクセスする前に `malloc` の返り値が null pointer でないかどうかを確認する処理を書くべきである．

```
int *foo(int x) {
  int *p = (int *)malloc(sizeof(int));
  if (p == NULL) { printf("malloc failed!!\n"); exit(0); }
  *p = x * 2;
  return p;
}
```

`NULL` が `<stdio.h>` で定義された null pointer を表す定数，`exit` はプログラムの実行をいきなり終了するためのライブラリ関数である．

`malloc` で確保された領域は，対になる `free` というライブラリ関数で解放することができる (解放すべきである)．

```
void free(void *);
```

free 関数は，(malloc で確保された領域の) ポインタを受け取って，その領域を解放する．解放した後にそのポインタの指す先にアクセスしてはいけない (未定義動作)．また，malloc で確保されたわけでもない領域のポインタを渡したり，同じポインタに対し二度 free を行った時の動作は未定義である．(ちなみに free に null pointer を渡した場合には，何もしない，という動作になる．)

free の使用例については後ほど詳しく見ていくが，新しい foo は例えば以下のように使うことができる．

```
  int *p = foo(100);
  printf("*p is %d", *p);  // prints "*p is 200"
  free(p);
  // the contents of p shouldn't be accessed any longer
```

1.3.5 中点関数再び (C/sample/struct-pointer.c)

さてここまでの議論を，中点を計算する関数をポインタ渡しで書き直して復習してみよう．まず，上の bar に倣ったのが以下の定義である．(中点は常に計算できるので返り値型は void としている．)

```
void middle1(struct point *p1, struct point *p2, struct point *result) {
  result->x = (p1->x + p2->x) / 2;
  result->y = (p1->y + p2->y) / 2;

  return;
}
```

中点を計算する対象となる二点 p1, p2 に加え，計算結果を書き込む先を指すポインタ result も引数として受け取る定義になっている．本体では，計算した結果を result->x などを通じて result の指す先のオブジェクトに書きこんでいる．以下の呼び出す側のコードでは，結果を書き込んでもらうための変数 m1 を用意して呼び出している．

```
struct point m1;
middle1(&origin, &p, &m1);
printf("middle1: The middle point between (%d, %d) and (%d, %d) is (%d, %d)\n",
       origin.x, origin.y, p.x, p.y, m1.x, m1.y);
```

構造体を直接渡す定義に比べると，引数を介してやりとりするのがポインタ (これは構造体のサイズに関わらず一定サイズ) なので，構造体が大きな場合には関数呼び出しのコストが低くなる．

計算結果のための領域を関数内で確保する場合，

```
struct point *middle2(struct point *p1, struct point *p2) {
  struct point m;

  m.x = (p1->x + p2->x) / 2;
  m.y = (p1->y + p2->y) / 2;

  return &m;
}
```

と auto 変数で領域を確保してそのポインタを返してしまうと，middle2 の呼び出し後に m の領域は解放されてしまうので，

```
struct point *m2 = middle2(&origin, &p);
// m2 is a dangling pointer (to an already deallocated object)
// So, accessing m2->x and m2->y in the following statement is illegal (undefined)
printf("middle2: The middle point between (%d, %d) and (%d, %d) is (%d, %d)\n",
       origin.x, origin.y, p.x, p.y, m2->x, m2->y);
```

の実行結果は未定義である．

以下のように m を static 変数にすれば，m の領域はプログラムの実行開始時点から最後まで確保されているので，領域が解放されてしまう問題はなくなる．

```
struct point *middle3(struct point *p1, struct point *p2) {
  static struct point m;

  m.x = (p1->x + p2->x) / 2;
  m.y = (p1->y + p2->y) / 2;

  return &m;
}
```

しかし，この方法だと，関数の呼び出しを何度もした時に答えを返すための領域が共有されてしまうという別の問題がある．

```
struct point *m3 = middle3(&origin, &p);
printf("middle3 (1): The middle point between (%d, %d) and (%d, %d) is (%d, %d)\n",
       origin.x, origin.y, p.x, p.y, m3->x, m3->y);
       // prints out
       //   "middle3 (1): The middle point between (0, 0) and (3, 8) is (1, 4)"

struct point *n3 = middle3(&origin, &origin);
printf("middle3 (2): The middle point between (%d, %d) and (%d, %d) is (%d, %d)\n",
       origin.x, origin.y, origin.x, origin.y, n3->x, n3->y);
       // prints out
       //   "middle3 (2): The middle point between (0, 0) and (0, 0) is (0, 0)"
```

正しい答えが出力されているが，実はこの時点で m3 は最早正しい答 (1,4) を格納していない．実際，二度目の答を格納している n3 と，最初の m3 は同じアドレスであり，m3 の答えが破壊されてしまっている．このことは，上のコードに続けて以下のようなコードを実行することで確認できる．

```
printf("m3 and n3 are the same address: %p and %p\n", m3, n3);

printf("middle3 (1'): The middle point between (%d, %d) and (%d, %d) is (%d, %d)!?\n",
       origin.x, origin.y, p.x, p.y, m3->x, m3->y);
       // prints out
```

```
       //   "m3 and n3 are the same address: 0x5617415d8018 and 0x5617415d8018"
       //   "middle3 (1'): The middle point between (0, 0) and (3, 8) is (0, 0)!?"
```

最後に，malloc 関数を使った定義を示す．malloc の定型的パターンで引数には構造体のサイズを渡し，キャスト (struct point *) で型をあわせている．

```
struct point *middle4(struct point *p1, struct point *p2) {
  struct point *m = (struct point *)malloc(sizeof(struct point));

  m->x = (p1->x + p2->x) / 2;
  m->y = (p1->y + p2->y) / 2;

  return m;
}
```

この関数を使うコードは middle2 や middle3 と同様である．

```
struct point *m4 = middle4(&origin, &p);
printf("middle4 (1): The middle point between (%d, %d) and (%d, %d) is (%d, %d)\n",
       origin.x, origin.y, p.x, p.y, m4->x, m4->y);

struct point *n4 = middle4(&origin, &origin);
printf("middle4 (2): The middle point between (%d, %d) and (%d, %d) is (%d, %d)\n",
       origin.x, origin.y, origin.x, origin.y, n4->x, n4->y);

printf("m4 and n4 are different addresses: %p and %p\n", m4, n4);

printf("middle4 (1'): The middle point between (%d, %d) and (%d, %d) is (%d, %d)\n",
       origin.x, origin.y, p.x, p.y, m4->x, m4->y);
```

このコードを実行してみるとわかるように，今度は m4 と n4 のアドレスは異なっており，二度目の middle4 の呼び出しの後も m4 は正しい答えを確保し続けていることがわかる．

また，m4 や n4 の指す先のオブジェクトは，使わなくなったら以下のように free を使って解放するべきである．(参考までに，解放後に m4 を使う**不正な** コードをつけている．Linux の gcc を使ってコンパイルしたところ，実行はできたもののおかしな結果が表示された．)

```
free(m4); free(n4);

printf("middle4 (1''): The middle point between (%d, %d) and (%d, %d) is (%d, %d)\n",
       origin.x, origin.y, p.x, p.y, m4->x, m4->y);
```

1.4 共用体 (C/sample/union.c)

構造体は複数のデータを並べたようなオブジェクトを作るための仕組みであった．これに対し，**共用体 (union)** は，異なる種類のデータを同じ領域に重ねあわせて，ひとつのオブジェクトを異なる型のデータとして扱うた

めの仕組みである．

以下は倍精度浮動小数点数と，int を重ねあわせた共用体 foo の宣言である．

```
union foo {
  int i;
  double d;
};
```

foo は (構造体と同様に) タグと呼ばれ，union foo の形で型として使うことができる．また，i と d は (これも構造体と同様に) メンバと呼ばれる．共用体 foo は，i という名前で読み書きできると，整数 d という名前で読み書きできる浮動小数点数が，同じ領域に重ねあわされているようなデータであることを意味している．「重ねあわされている」というのは，メンバ d を通じてアクセスした時には double として，メンバ i を通じてアクセスした時には int として使えるが，同時には使えない，という感じである．以下の，コード断片は，union foo の変数を用意して，そこに整数 0x12345678 を書き込んでいる．f を整数として扱いたいため，メンバ i を使っている．(文字列の中に現れる 0x は整数定数を 16 進数表記で与える時の接頭辞である．)

```
  union foo f;
  f.i = 0x12345678;

  printf("The size of f is %zd\n", sizeof(union foo));
  printf("f is allocated at %p\n", &f);
  printf("f.i is allocated at %p\n", &(f.i));
  printf("f.d is allocated at %p\n", &(f.d));
  printf("f as integer is %d (decimal) and 0x%x (hexadecimal)\n", f.i, f.i);
/* 実行結果:
The size of f is 8
f is allocatd at 0x7ffea344bcc0
f.i is allocated at 0x7ffea344bcc0
f.d is allocated at 0x7ffea344bcc0
f as integer is 305419896 (decimal) and 0x12345678 (hexadecimal)
*/
```

実行結果にあるように，f のサイズは 8 である．これは int のサイズ 4 と double のサイズ 8 の最大をとった値になっている．共用体はメンバが同じ領域を占めるので，そのサイズは，全メンバの中の最大サイズになる．また，f.i も f.d アドレスは f と同じである．

さて，変数 f には，int か double のどちらか書き込まれているわけだが，**読み出しを行う際には最後に書き込みが行われた型で読み出しを行わなければならない**．上のコードに以下のようなものを続けると，未定義動作になる．

```
  // Reading f.d before writing as f.d is illegal
  printf("f as double is %3.14f\n", f.d);
```

(ただし，おそらく多くのコンパイラでは実行できてしまうだろう．こちらの環境では 0.0 が得られてしまった．)

```
f as double is 0.00000000000000
```

一般に，共用体の中に共用体メンバ，構造体の中に構造体メンバ，構造体の中に共用体メンバが入れ子になるどのパターンも許されている．入れ子になった○○体の宣言をする際には以下のようにまとめて宣言してもよい．

```
union foo {
  int i;
  struct bar {
    double x;
    double y;
  } d;
};
```

また，共用体は OCaml のヴァリアントのように，異なる型の値を混ぜて扱うためにも使う．例えば，2 分探索木のデータを

```
union tree {
  struct leaf {...} l;
  struct branch {...} b;
}
```

のように表現することが考えられるだろう．この場合，ひとつの型 (`union tree`) で，ふたつの種類の値を保持している可能性を表現したい，というのが用途である．しかし，OCaml のヴァリアントと違って，共用体の表す値がどのメンバの値であるかを知る術は用意されていない．つまり，`union tree` 型の変数があっても，それが leaf を表しているか，branch を表しているデータなのかわからないということである．我々は後で 2 分探索木をプログラムする際には，共用体だけでなく leaf/branch の区別をするための情報も付加してデータを設計することになる．

1.5 列挙型

列挙型は，OCaml のヴァリアントの最初の例 (`furikake`) でみたような，いくつかの定数からなる型を定義するための仕組みである．OCaml の

```
type furikake = Shake | Katsuo | Nori;;
```

に対応する型は，

```
enum furikake {
  shake, katsuo, nori
};
```

のように定義できる．構造体・共用体の時と同じく `furikake` はタグと呼ばれ `enum furikake` は型として使える．`{}` の中で宣言された名前は **列挙定数 (enumeration constant)** と呼ばれる．しかし，OCaml とは違って，列挙型は単なる整数型の別名であり，また列挙定数も整数定数の別名である．(左から順に 0，1，2 と割り当てられていく．実は `enum` 宣言の時に `katsuo=10` などと，どの定数なのか指定することもできる．しかも恐しいことに，`=` で指定しない列挙定数の数が 10 に達すると異なる名前に同じ整数が割り当てられる!) 所詮は整数なので，四則演算なども全く問題なく (？) 行うことができる．

(以下は `C/sample/enum.c` より)

```
#include <stdio.h>

enum day {
  monday, tuesday, wednesday, thursday, friday, saturday, sunday
};

int main(void) {
  printf("today is %d\n", monday);

  int x = wednesday * saturday;
  printf("today is %d\n", x);
}
```

1.6 配列*

この先執筆中

- 配列オブジェクト
- 配列アクセス，オーバーラン
- 配列変数の真の意味，配列の引数渡しとポインタ演算
- 多次元配列