**C言語演習**　　　　　　　　　　　　　　　　　　　担当：手塚太郎

本演習ではLinux環境でC言語のプログラミングを行う。

参考書として以下のものを挙げる。テキストの説明が分かりにくかった場合は以下の資料を参考するか、Webページを参照するとよい。

・林晴比古，改訂 新C言語入門 ビギナー編，ソフトバンククリエイティブ，2003.

・林晴比古，改訂 新C言語入門 シニア編，ソフトバンククリエイティブ，2004.

* 平林雅英，ANSI C/C++辞典，共立出版，1996．
* 近藤嘉雪，定本 Cプログラマのためのアルゴリズムとデータ構造，ソフトバンククリエイティブ，1998．

・奥村晴彦，C言語による最新アルゴリズム事典，技術評論社，1991．

・渡邉敏正，データ構造と基本アルゴリズム，共立出版，2000．

・島内剛一他，アルゴリズム辞典，共立出版，1994．

なお、各問題の終わりにある記号「■」はそこまでが問題の範囲であることを表している。

ソースコードは以下の場所にある。

/home/tezuka.taro.gm/lec/c

以下のように実習用のディレクトリを作った上でコピーしてくるとよい。

cd ~

mkdir softeng

cd softeng

mkdir c

cp /home/tezuka.taro.gm/lec/c/\* c

cd c

**はじめに デバグに関して**

プログラムのバグを取り除く方法は多くの言語で共通である。本演習はC言語を対象としているが、以下の点などを参考に、デバグの能力を高めていってほしい。

* エラーメッセージにはエラーが発生した行番号が書かれていることが多いので、その行を見ておかしな部分がないか確認する。
* エラーメッセージの種類はたかだか十数個のため、英語であっても読むようにしていればやがて憶えていける。難しい表現は使われていない。
* エラーメッセージではそのエラーの発生に至るまでの経路（関数の呼び出しの列）まで含めて表示されることがある。その場合、各段落が関数に対応し、その関数名が書かれている。自分が作成した関数名あるいはファイル名のある段落を探し、そこで何というエラーが何行目で生じているかを見ると良い。エラーが一ヶ所だけである場合、それは一番上か一番下の段落にあることが多い。
* はじめてみるエラーメッセージの場合、そのメッセージでWeb検索を行うと、ヒントがつかめることがある。そのエラーメッセージ固有と思われる部分（行番号やファイル名などは取り除いたもの）をフレーズ検索（ダブルクオーツに入れて検索）してみてもよい。
* エラーメッセージにエラー発生の行番号が書かれていない場合、**怪しいと思うところをコメントアウト（行頭に//を付けたり、/\* \*/で囲む）して正常に動くかを確認し、問題箇所を絞り込む**。
* 画面への出力の関数printfを使って変数の値を出力させることで、プログラム中でそれがどのように変化しているかを表示させると、おかしな挙動の理由が分かることがある。しかし**printfの中身が表示されないからといって、それより前の行でエラーが生じたとは限らない**。なぜならprintfの出力はバッファリングといって、ディスプレイに出力される前に一時的に蓄えられ、一定の量がまとめて送られるようにしている。これによってCPUとディスプレイの通信を開く時に掛かるコストを減らしている。バッファの中身がディスプレイに送られる前にエラーが生じてしまうと、処理がprintfの行より先に進んでいるにも関わらず、ディスプレイへの表示が行われないまま止まってしまうということが起きる。これによってあたかもprintfの行より前でエラーが生じたように見えるが、実際はprintfより後でエラーが生じていることがある。
* ソースコードをコピーアンドペースト（コピペ）した場合、**全角スペースや全角文字などが入っていて、エラーを生じさせていることがある**。特にクオーツやコンマは全角と半角が似ているので注意する。
* セグメンテーションフォルト（segmentation fault）というエラーメッセージが出た場合、ポインタや配列関係で生じていることが多い。たとえばポインタを宣言したが、その中身をmallocで確保していないままアクセスした時に生じる。あるいは配列の大きさより大きい添え字で配列にアクセスしようとしているなど。

**1 　C言語の基本**

C言語のソースファイルはRubyと同様、エディタを使って作成できる。本資料ではLinux上で行った結果を示すが、他のOSや開発環境を使用してもよい。

以下のファイルをEmacs等のエディタで作成し、sample1-1.cと名付けて保存せよ。

// sample 1-1

#include <stdio.h>

int main(){

printf("Hello world!\n");

}

C言語のプログラムはRubyのプログラムと違い実行の前にあらかじめコンパイル（compile）という操作を行っておく必要がある。これは人間が書きやすく理解しやすいが冗長な形で書かれたプログラムを計算機が高速に実行できる形に変換する操作である。プログラムに対してコンパイルを行うプログラムをコンパイラ（compiler）と呼ぶ。

Cには多数のコンパイラが存在するが、本演習ではgccを使用する。gccはGNU c compilerの略であり、オープンソースで開発され、無償で使用できる。また、Linuxやmacなどには標準的に搭載されている。

端末上で以下のように打ち込むと、ソースコードsample1-1.cがコンパイルされ、実行可能ファイルprog1が生成される。gccの後に続く-oは出力先（output）を指定するためのオプションである。作成したい実行可能ファイルの名称はこの後に書く。最後の引数はコンパイルの対象となるソースコードである。

gcc -o sample1-1 sample1-1.c

**基本課題1　 プログラムのコンパイル**

-oオプションを付けずに上記の命令を行った場合（すなわちgcc sample1-1.cと打ち込んだ場合）何が起きるか。どのようなファイルが生成されているかを確認せよ。なお、端末ではls –lrtと打つと、現在のディレクトリにあるファイルの一覧が古い順にソートされて出力される。（そのため一番新しいファイルが一番下に表示される）。■

　コンパイルの結果は実行可能ファイルであるため、ファイル名だけで実行できる。

gcc ./sample1-1

sample1-1.c → sample1-1 → 実行

端末（ターミナル）において実行する際には以下のように先頭に「./」を付けることで現在のディレクトリにあるファイルを指定していることを明示し、実行させる。

./sample1-1

以下ではソースコードsample 1-1の説明を行う。

stdio.hはstandard input output header fileの略であり、入出力に関わるライブラリを読み込むための命令である。これによって出力の命令printfが使えるようになる。int main(){ } はmain関数を定義する部分である。Cのプログラムは必ずmain関数から実行される。その他の関数を定義する場合は main(){ } の外に書く。それらの関数はmain関数の中で呼び出すことができる。なお、中括弧（波括弧）で囲まれた部分を**ブロック**（block）と呼ぶ。

　C言語ではすべての変数は**型**（type）を持つ。型はその変数に入れることのできるデータの種類を表している。Cにおける主な型はint, long, float, double, charなどである。int型とlong型の変数には整数が入れられる。float型とdouble型の変数には浮動小数点小数が入る。float型は単精度、double型は倍精度であり、小数点以下何桁まで格納されるかが違う。char型の変数には文字をひとつだけ入れることができる。

Cには（文字ではなく）文字列を表す型が存在しないため、文字列を格納するにはchar型の配列を使用する。sample 1-2はchar型の配列に文字列を格納する例である。

// sample 1-2

#include <stdio.h>

int main(){

char str1[] = "Hello";

char str2[] = "world";

int y = 2;

printf("%s %s! I'm sample %d!", str1, str2, y);

}

**基本課題2　 プログラムのコンパイルと実行**

sample 1-2 をコンパイルし、実行せよ。■

今回のprintfではダブルクオーツで括られた第一引数の他、三つの引数が加わっている。printfのfはformatの略であり、フォーマット（書式）を指定して出力を行うことがひとつの特徴である。第1引数で指定した文字列のうち、%とそれに続く一文字が第二引数以降で渡した変数の値で置き換わる。この場合、最初の%sはstr1の値で、次の%sはstr2の値で、%dはyの値で置き換わる。%とそれに続く一文字はフォーマット指定子と呼ばれ、%sは文字列（string）、%dは整数（decimal）、%fは浮動小数点数（float）、%xは16進数（hexadecimal）というように、変数の値をどのような形で出力させるかを決めている。紛らわしいが、double型は浮動小数点数なので、%dではなく%fによって出力させる。

Cにおいて文字列表示に毎回フォーマット指定を行わなくてはならないのは 、計算機の用途が主に数値の出力であった時代の名残である。printfとループを使えば各行に同じ形で数字が並んでいるような数表を容易に出力できるため、そのような用途に向いている。Cは古くからある言語であるため、その仕様は計算機の過去の用途を反映した部分も多い。

**基本課題3　 文字列の格納と出力**

自分の名前を変数に格納し、それを中に入れて挨拶を出力させるプログラムを作成せよ。

（出力例： こんにちは、太郎さん！） ■

実行時、キーボードから入力した値を変数に代入するのにはscanf()を使用する。次のプログラムではscanfの行でシステムはユーザの入力を待ち、改行が押されるとそれまでに入力された文字列のうち、最初の空白より前の部分を変数firstnameに、空白より後の部分を変数lastnameに格納している。（改行までを読み込むことが決まっているので、printfと違い、scanfの書式指定の中では\nは不要である）。また、char name[20]と宣言することによってnameに20文字までの文字列が格納できるようにしている。

// sample 1-3

#include <stdio.h>

int main(){

char firstname[20];

char lastname[20];

int id;

printf("What’s your full name?\n");

scanf("%s %s", firstname, lastname);

printf("What is your student ID?\n");

scanf("%d", &id);

printf("Welcome, %s (student ID:%d)!\n", firstname, id);

}

scanfで数字を入力させる際、第二引数idの前に&がついている点に注意が必要である。これは関数scanfの内部で変数idの値を操作できるようにするためには、idの値を渡す（教える）だけでは不十分で、idというデータがどこに格納されているか（メモリ上の位置がどこであるか）を教えなくてはならないことに由来している。変数の前に&をつけることで、その変数の値ではなく、住所（アドレス）が得られるため、ここではそれを使っている。それではchar型の配列nameを渡す時にはなぜ&が不要であるのかについては、次章で**ポインタ**（pointer）について学ぶことで明らかになる。（変数nameは配列データの先頭がメモリ上でどの位置にあるかを表すポインタなのである）。

**基本課題4 scanfによる文字列の読み込み**

二つの数字を入力させ、それらの和を出力させるプログラムを書け。■

C言語の基本的な構文、特に制御構造はJavaやRubyによく似ている。if文やwhile文などは同じ意味を持つ。ただしRubyのように多様な書き方はできない。また、変数の型は（forなどの）命令の中ではなくあらかじめ別の行として定義しておかなくてはならない。（たとえば for(int i = 0; i < 10; i++){ といった書き方はできない。なお、i++ はi = i + 1 の略であり、iの値をひとつ増やさせる命令である。（++をインクリメント演算子と呼ぶ。i-- は = i - 1 の略であり、--はデクリメント演算子と呼ばれる）。

// sample 1-4

#include <stdio.h>

int main(){

int i;

for(i = 0; i < 10; i++){

if(i == 7){

printf("Hello world!\n");

}else{

printf(“Hi!”);

}

}

}

**基本課題5 for文によるループ**

好きな名言をchar型の配列に格納し、10回出力させるプログラムを作成せよ。forループを使用すること。■

　for文を使うのと同様に、while文を使ったプログラムも書ける。以下はライプニッツ級数を用いて円周率πを求める（近似する）プログラムである。このプログラムでは数学関数ライブラリ（mathライブラリ）を利用しているため、以下のように-lmをオプションとして付けてコンパイルする必要がある。-lmはgccの直後でなく末尾に付ける必要がある。

gcc –o sample1-5 sample1-5.c -lm

-lは読み込むライブラリ（library）を指定するためのオプションであり、mはmathの略である。

級数とは数列の和である。ライプニッツ級数は以下のように、分母が奇数の分数について加算と減算を交互に行うことで得られる無限級数である。



これは以下のようにも表せる。



この値はarctan 1に収束することが知られている。arctanはtanの逆関数であり、arctan 1はtanの値が1になる角度、すなわち45度（ラジアンで表すとπ/4）である。



このためこの無限級数の計算を途中で打ち切り、4をかけることで、πの近似値が求められる。ただしこの級数の収束は遅く、なかなかπ/4に近付かない。実際のπの計算にはより収束の速い級数が使われる。以下のプログラムを実行し、結果を確認せよ。math.hの関数を使っているため、以下のように-lm オプションを付けてコンパイルすること。

gcc -o sample1-5 sample1-5.c -lm

// sample 1-5

#include <stdio.h>

#include <math.h>

int main(){

int i　= 0;

double arctan1 = 0;

while(i < 10000){

arctan1 += pow(-1,i) / (i\*2 + 1);

if(i % 100 == 0){

printf("pi = %f\n", arctan1 \* 4);

}

i++;

}

}

　ここではヘッダファイルmath.hをincludeすることによって、指数を計算する関数powを使えるようにしている。（-lmによってmathライブラリが読み込まれているが、さらにinclude文でヘッダファイルを読み込ませることでプログラム内で利用できるようになる）。これによって-1のi乗を求めている。これを掛けることで正の項と負の項が交互に現れることを表せる。また、i % 100 はiを100で割った時の余りで、これが0と等しい時にのみif文の中身が実行される。これによってπの近似値が100回に一回だけ表示されるようにしている。なお、+= は左辺の変数に右辺の値を足すことを意味している。すなわちarctan1 += pow(-1,i) / (i\*2 + 1); は以下の略である。

arctan1 = arctan1 + pow(-1,i) / (i\*2 + 1);

**発展課題1 while文によるループ**

自然対数の底（ネイピア数）を求めるプログラムを作成せよ。ただしwhile文を使用すること。計算には以下の公式を使えば良い。



なお、0!は1である。ループは10回まわす程度で比較的良い値が得られる。階乗は急激に大きくなるため、あまりまわし過ぎると分数の値（1/n!）が小さくなりすぎ、精度の限界を超えてしまい、値が求められなくなる。ネイピア数の実際の値は各自調べ、正しい値が求められていることを確認すること。注意する点として、プログラミング言語の多くではaとbが整数の時、a/bを整数の割り算と解釈してしまう。たとえば9/2の結果として4（そして余りが1）を返す。それを避けるためには1.0\*a/bというようにするとよい。今回の場合、階乗がfactに入っているとして、1/factでなく1.0/factと書くとよい。■

これまでにもいくつかの関数（printfやpowなど）を使用してきたが、独自の関数を定義し、呼び出すこともできる。関数の定義はmain関数の定義の外で行う。

関数を定義する際、関数名の前に戻り値の型を書く。以下のsample1-6.cの場合、getSmallestFactor関数は最小の素因子を求める関数であり、整数を返して欲しいので、その戻り値の型はint型である。戻り値を返さない関数の前にはvoidと書く 。voidは空洞という意味である。下記のmain関数の場合は冒頭のvoidが省略されている。関数名に続く小括弧（丸括弧）の中に関数が受け取る引数を書く。これにもそれぞれ型を指定する必要がある。引数は複数でもよい。

sample1-6.cでは入力された数が素数であるかどうかを判定している。実際の判定を行う関数がgetSmallestFactorである。これはもし引数が素数ならば0、そうでなければ引数のもっとも小さい約数（factor）を戻り値として返す。戻り値はreturn文によって指定する。

if(input == 1){ printf("1 is not a prime number.\n"); return;} という行のように、セミコロンで区切られた複数の命令をひとつの行に書いてもよい。上記の行ではinput == 1 だった場合にreturn文によって関数から抜けるようにしている。

// sample 1-6

#include <stdio.h>

int getSmallestFactor(int a){

int i = 2;

while(i\*i <= a){

if(a % i == 0){

return i;

}

i++;

}

return 0;

}

int main(){

int input, factor;

printf("Type in a number: ");

scanf("%d", &input);

if(input == 1){ printf("1 is not a prime number.\n"); return;}

factor = getSmallestFactor(input);

if(factor > 0){

printf("%d is not a prime number. It can be divided by %d.\n", input, factor);

}else{

printf("%d is a prime number.\n", input);

}

}

関数はいくつ入れ子にして呼び出してもよい。たとえばgetSmallestFactor関数の中で自分で定義した別の関数を使ってもよい。

**基本課題6 関数の呼び出し**

getSmallestFactor関数を使い、因数分解を行うプログラムを作成せよ。たとえば120と入力した時、出力が2\*2\*2\*3\*5となるようにせよ。■

**発展課題2 関数の定義**

ライプニッツ級数を計算するsample1-5においてはmath.hのべき乗関数powを使用しているが、独自の関数（名前はpow以外、たとえばpowerにする ）を定義し、math.hを使わずに計算が行えるようにせよ。なお、math.hのpow関数は二つの実数を引数としてとり、0.5乗なども求められるが、ここでは整数乗を求めることができれば十分である。すなわち第二引数はint型でよい。math.hのinclude文を外し、正常に動くことを確認せよ。

なお、独自に定義したべき乗関数の戻り値の型はdouble型にするのがよい。もしint型の場合、pow(-i,i) / (i\*2 + 1) はint型をint型で割る計算、すなわち（小学校で習う）整数の割り算になり、結果は整数になってしまう。（たとえば 7 / 3 は2になってしまう）。ゆえにべき乗関数の戻り値を強制的にdouble型にする必要がある。これには以下のようにdouble型の数1.0を掛けておく方法がある。これはdouble型×int型の結果がdouble型になることを利用している。

arctan1 += 1.0 \* power(-1,i) / (i\*2 + 1);

また、強制的に型を変換する**キャスト**（cast）という機能を使い、以下のように書いてもよい．

arctan1 += (double)power(-1,i) / (i\*2 + 1);

なお、int型の変数をdouble型にキャストすることは可能であるが、double型の変数をint型にキャストすることはできない。■

関数の定義の中で自分自身を使用してもよい。これを**再帰呼び出し**（recursion）と呼び、プログラミングにおいて非常に重要な考え方である。 プログラムで再帰呼び出しを使うには関数を再帰的に定義する。すなわちある関数funcの定義の中でfuncを使用する。

再帰的定義は日常生活でも使われている。たとえば「子孫」という概念を定義する時、「子は子孫」「子の子は子孫」「子の子の子は子孫」……という定義ではいつまで 終わらない。一方、「子は子孫」「子孫の子は子孫」とすれば定義は完了である。ここでは「子孫」を定義するのに「子孫の子」を使っており、再帰的定義が行われている。

数学でも再帰的定義は重要である。「自然数」を定義する時、「0は自然数」「1は自然数」「2は自然数」……という定義ではいつまでも終わらないが、「0は自然数」「自然数に1を足したものは自然数」とすれば定義は完了する。ここでは「自然数」を定義するのに「自然数に1を足したもの」を使っており、再帰的定義が行われている。

**基本課題7 関数の再帰呼び出し**

scanfで入力された数に対し、その階乗を出力するプログラムを再帰的定義を用いて実装せよ。たとえばnの階乗はn-1の階乗にnを掛けたものである。そのためfact(int n)の定義の中でfact(n-1)を使うことが考えられる。また、再帰呼び出しが止まるようにするために、fact(0)は1であることも盛り込む必要がある。すなわちif文を使い、n==0の場合とn>0の場合で処理を分けるとよい。参考として再帰呼び出しによって1からaまでの和を求める関数seriesと、それを使ったプログラムを以下に示す。ただしseriesは引数aが0以下の時は0を返す。

// sample 1-7

#include <stdio.h>

int series(int a){

if(a <= 0){

return 0;

}else{

return a + series(a-1);

}

}

int main(){

int n = 10;

int res = series(n);

printf(“the sum of 1 to %d is %d”, n, res);

}

■

**発展課題3 関数の定義**

引数として与えられた数値xに対し、正弦関数（sin）と余弦関数（cos）を求める関数mysinとmycosを実装し、それをプログラムから呼び出せ。ただし以下の級数を使って計算すること。





main関数の中でmysinとmycosに30度、45度、90度などを代入し、正しい値が求められていることを確認せよ。ただし角度はラジアンで与える必要がある。30度であればラジアンはπ/6であり、45度であればラジアンはπ/4である。すなわち mysin(3.141592653589 / 6) といった形で呼び出せばよい。

なお、この展開式はsinやcosを0のまわりでテイラー展開することによって求められる。

**発展課題4 級数の計算**

πに収束する他の級数（たとえばバーゼル問題）について調べ、πの近似値を求めるプログラムを実装せよ。バーゼル問題とはnの-2乗の和を求める問題であり、オイラーによって以下の結果が得られている。



左辺の級数を途中まで計算し、6をかけて平方根を求めればπの近似値が得られる。平方根を求めるにはmath.hの関数sqrtが使える。 ■

なお、左辺の級数はζ(2)とも表される。一般のζ関数（ゼータ関数）は以下のように定義され、現在も数学において活発な研究の対象である。ただしsは任意の複素数である。たとえば現代の数学においてもっとも有名な未解決問題であるリーマン予想はゼータ関数の零点（値が0になる点、すなわちζ(s) = 0 になる複素数s）が複素平面上でどこに存在するかという問題に対する予想である。



**2　ポインタ**

ポインタはデータの住所（アドレス）を格納した変数である。変数の住所を求めるには&（アンパッサンド）演算子を使う。以下にポインタpにデータ（変数）aのアドレスを格納する例を挙げる。

int a = 10;

int\* p = &a;

これによってpにはデータaの存在する（メモリ空間上の）アドレスが格納される。住所を「指して（point）」いるので、ポインタ（pointer）という名前が付けられている。

int型のデータに対するポインタの型はint\*であり、char型のデータに対するポインタの型はchar\*である。ただし宣言の際、int \*pやchar \*cのように、アスタリスク（\*）を変数の側につけて（空白の後に）書いてもよい。つまりint \*pとint\* pは同じ意味である。

変数aに格納されたデータはaで表され、そのデータのアドレスは&aで表される。一方、ポインタpに格納されたアドレスはpで表され、そのアドレスにあるデータは\*pで表される。つまりポインタ演算子「\*」はアドレス演算子「&」の逆演算子であると考えて良い。実際、 &(\*p) は p と等しく、\*(&a) は a と等しい。

ポインタを使用するとデータそのものではなくそれが存在する場所を参照できるため、様々な便利な形で使用できる。

　ポインタは配列の代用として使えることも重要である。配列arrayを作った時、その要素はメモリ空間上で順番に並んでいる。そのため、たとえばarray[0]のアドレスが分かれば、それに1を加えることでarray[1]のアドレスが分かる。同様にarray[0]のアドレスにiを加えることでarray[i]のアドレスが分かる。すなわちarray[0]のアドレスを通して配列の任意の要素にアクセスできる。

　実は配列arrayにおいて、arrayという変数自体は配列の先頭の要素のアドレスを表している。すなわちarrayは&(array[0])と等しい。このため、たとえば \*(array + i) によって配列のi番目の要素にアクセスできる。（ただし最初の要素を0番目とみなしている）。Cでは文字列はchar型の配列であるため、「strで文字列を表す」というのは実際はstrは文字列の先頭アドレスを表すポインタである。

　printfでフォーマットを%pと指定すると、アドレスが16進法で表示される。16進法では0, 1, 2, ..., 9という10種の記号 に加え、A, B, C, ..., Fという6つの文字も使って数字を表す。繰り上がりは10進法の16になった時に生じる。たとえば16進法でFと書いた場合、それは10進法での15に対応する。16進法で10と書いた場合、10進法での16に対応する。16進法で3Bと書いた時、10進法では (3 \* 16) + 11 = 59 に対応する。

**基本課題8　ポインタの基礎（1）**

以下のプログラムを入力し、コンパイルして実行せよ。実行結果を確認し、表示されたそれぞれの数値が何を意味しているかを理解せよ。

// sample 2-1

#include <stdio.h>

#include <stdlib.h>

int main(){

int a, \*p;

a = 10;

printf("a: %d\n", a);

printf("&a: %p\n\n", &a);

p = &a;

printf("p: %p\n", p);

printf("\*p: %d\n", \*p);

printf("&p: %p\n\n", &p);

a = 20;

printf("a: %d\n", a);

printf("&a: %p\n\n", &a);

printf("p: %p\n", p);

printf("\*p: %d\n", \*p);

printf("&p: %p\n", &p);

}

■

**基本課題9　ポインタの基礎（2）**

以下のプログラムを入力し、コンパイルして実行せよ。実行結果を確認し、表示されたそれぞれの数値が何を意味しているかを理解せよ。また、文字列sに含まれる「w」という文字を（ポインタを通して操作することで）大文字に書き換えるようにプログラムを改変せよ。

#include <stdio.h>

#include <stdlib.h>

int main(){

char c, \*p;

char s[9] = "Hardware";

c = 'A';

p = &c;

printf("c: %c\n", c);

printf("&c: %p\n\n", &c);

printf("p: %p\n", p);

printf("\*p: %c\n\n", \*p);

\*p = 'B';

printf("c: %c\n", c);

printf("&c: %p\n\n", &c);

printf("p: %p\n", p);

printf("\*p: %c\n\n", \*p);

printf("s: %s\n", s);

printf("s[0]: %c\n", s[0]);

printf("s[1]: %c\n", s[1]);

printf("&s[0]: %p\n", &s[0]);

printf("\*s: %c\n", \*s);

printf("\*(s+1): %c\n\n", \*(s+1));

\*(s+3) = 'D';

printf("s: %s\n", s);

}

■

ポインタを使う際、そのポインタが指すアドレスから始まるメモリ領域を確保するために標準ライブラリ関数malloc()を使用する。mallocはmemory allocation（メモリ割り当て）の略であり、以下のように使用する。

　char\* p = (char \*)malloc(sizeof(char)\*size);

sizeofは各データ型のデータサイズをバイト数で返す関数である。たとえばchar型の変数にはASCIIコードが格納され、その大きさは1バイトであるので、sizeof(char)は1になる。一方、sizeof(int)は4になる。実際、大きな数値を格納するには1バイト＝8ビット＝256種では足りないため、int型では4バイトが使われている。単精度浮動小数点数を格納するfloat型は4バイト、倍精度浮動小数点数を格納するdouble型は8バイトを使用する。

上記のコード例ではsizeは確保したい領域の大きさ（配列における要素数）を表すint型の変数であり、プログラム中で設定する必要がある。これによってchar型の値をsize個入れられる領域がメモリ上に一列に並んで確保され、その先頭のアドレスがpに格納される。

配列と比較した時のポインタのメリットとして、保存できる要素数をプログラムの実行時に決められるという点が挙げられる。

int型に限らず、任意の型のポインタについてmallocを行える。

　double\* p = (double \*)malloc(sizeof(double)\*size);

なお、この章のサンプルコードの冒頭でstdlib.hをincludeしているのはmallocを使用するためである。

また、mallocによって確保した領域は使わなくなった段階で解放することでメモリ空間を無駄なく使うことができる。これには関数freeを使用する。RubyやJavaなどの言語ではガーベッジコレクションといって、使わなくなった変数が使用しているメモリ空間を自動で解放していく機能があるが、Cにはそれがないため、明示的に領域の解放を行わなくてはならない。

**基本課題10　ポインタによる配列の実現（int型の配列）**

int\*型のポインタpを定義し、(int \*)malloc(sizeof(int)\*10)によって10個の数字を格納できる領域を確保する．キーボードから入力した10個の整数を、配列要素に順次ポインタpを移動させながら代入し、入力した10個の整数とその要素のアドレスを、同様にポインタpを移動させながら出力するプログラムを作成せよ。

なお、キーボードから入力した値を配列要素に代入するのにはscanf()を用い、第2引数にはポインタpを使用すること。（この際、&pのように&を付ける必要はない）。またポインタを配列の次の要素に移動するためにインクリメント演算子（++）を用いること。また、アドレスの出力ではprintf()における書式指定で%pを用いて出力すること。

　実行例：

　整数値を10個入力して下さい。ひとつ入力するごとに改行を押してください。：

（ここで数字を入力）

配列の中身は以下の通りです。

　値：5 アドレス：0xbf963590

値：7 アドレス：0xbf963594

．．．

値：3 アドレス：0xbf9635b4

(注)　ここで示したアドレスの値は実際の実行結果と異なる可能性がある。

アドレスは16進法で表記されている。数字が4ずつ増えているのは、int型のデータをひとつ入れるために複数のデータ領域（バイト）が必要なためである。この大きさは型によって違う。これは型によって取りうる値の種類の数が違うためである。■

配列の要素p[i]を参照する場合、pの先頭アドレスからi番目を調べるため、内部的に足し算を行っている。実際、p[i]と\*(p+i)は同値であるが、p[i]を参照する場合も\*(p+i)と同様の足し算（pに格納されたアドレスにiだけ足す）が行われていると考えられる。一方、p++によってpの指すアドレスを変化させていく場合、iの値を参照する必要がなくなるため、若干計算の効率化が行われる。

**基本課題11　ポインタによる配列の実現（char型の配列）**

char\*型のポインタpを定義し、(int \*)malloc(sizeof(char)\*10)によって10個の数字を格納できる領域を確保する．キーボードから入力した10個の文字を、配列要素に順次ポインタpを移動させながら代入し、入力した10個の文字とその要素のアドレスを、同様にポインタpを移動させながら出力するプログラムを作成せよ。入力を行う際、アドレスの増え方が前問とどのように違っているかを比較せよ。いずれもp++によってアドレスをインクリメントしているが、pがint\*型であるかchar\*型であるかによって足される値が異なっていることが分かる。■

**基本課題12　ポインタと文字列**

char型のポインタstrと、char型のポインタpを定義し、strから始まる文字列に最大100文字を格納できるできるようにmallocする。scanfを使ってキーボードから入力した任意の文字列をstrに格納した上で、先頭から1文字ずつポインタpを動かす形で文字列を出力し、その後、文字列を1文字ずつ逆順に出力するプログラムを作成せよ。

Cでは文字列の終わりには必ずヌル文字「\0」が入っている。これと一致するまでループさせることで、文字列の長さを求められる。たとえば以下のような関数を作ると、文字列strの長さを戻り値として得られる。このプログラムを改変すればよい。

int strLength(char \*str){

int n = 0;

while( \*str != ’\0’){

str++;

n++;

}

return n;

}

\0はダブルクオーツ（””）ではなくシングルクオート（’’）の中に入れることに注意する。これはダブルクオーツに入れたものは文字列とみなされるため、その後にさらに別の「\0」が入れられてしまうためである。一方、シングルクオートに入れた場合はchar型の値とみなされる。（なお、ヌル文字はASCIIコードにおいて0に対応付けられる文字である。一方、’0’にはASCIIコードでは48が割り当てられている。’\0’や’0’を格納した変数cをprintf(“%d\n”,c) というように%d（10進表記）で出力させると確認できる）。文字列の先頭より前には「\0」が入っているとは限らないので，逆向きに出力させる場合は文字列の先頭のアドレスがポインタstrに格納されているので，末尾から先頭まで動かす時にはpがstrと一致するまでwhileループでpを減らしていく（p--を行う）という方法が考えられる。

while(\*p != ’\0’){

（略）

p++;

}

while(p >= str){

（略）

p--;

}

　実行例：

　100文字以内の文字列を入力して下さい： hello!

　hello!

　!olleh

■

　scanfにおいて数値を読み込ませる場合はscanf("%d", &id);のように&をつけたが、文字列を読み込ませる場合はscanf("%s", name);のように&が不要であったのは、nameが文字列の先頭のアドレスを表すポインタであるためである。

scanfはキーボードに入力された値を変数に格納する関数であるが、scanfから見てデータをどこに書き込んだらよいかを知るためには、scanfにidの値を教えるのでは不十分であり、idという変数が表すデータが保存される場所（アドレス）を教えなくてはならない。そのためポインタを渡さねばならず、&をつけることで変数idのアドレス（ポインタ）を求めているのである。関数にポインタを渡すことはポインタ渡し（またはアドレス渡し）と呼ばれる。ポインタ渡しの対義語は値渡しであり、この場合は変数のアドレスではなく、変数が持つ値が渡されている。

たとえば会社で上司が部下に紙の書類の書き換えを頼む場合を考える。上司は呼び出し元の関数、部下は呼び出される関数である。上司が部下に書類のコピーを渡すのが値渡し、書類の場所を教えるのがポインタ渡しである。前者では元々の書類自体は書き換わらないが、後者では部下は書類の原本を書き換えることができる。

**基本課題13 ポインタ渡し（アドレス渡し）**

二つの整数値をキーボードから入力してそれぞれ変数に代入し、二つの変数の値を入れ換えるプログラムを作成せよ。その際、二つの整数型の変数を指し示すポインタを引数とし、これらの値を入れ換える関数swap()を作成し、これを利用すること。swap()の仕様は以下の通りとする。

void swap(int \*x, int \*y)

機能： 変数xとyの値を入れ換える．

引数x，y： 値を入れ換えるそれぞれの変数

戻り値： なし

実行例：

二つの整数値を入力して下さい：3 9

変数xの値は3，変数yの値は9です．

swap()を呼び出した後の変数xの値は9，変数yの値は3です．

　仮に関数swapにポインタ\*xと\*yではなくxとyを与え、関数内で入れ換えを行った場合、その結果はswapの外にあるxとyに反映されるか？ ■

**発展課題5　ポインタを使ったオリジナルのprintfの実装**

putc(c, stdout)によってchar型の変数cの値を標準出力（画面出力）stdoutに書き出すことができる。printfで文字列を出力するのと同じ挙動を行う関数myprintfを実装し、main関数の中で実際に使用せよ。ただしmyprintfの仮引数は（printfのようなフォーマット指定の引数は入れず）char\*型の変数strのみでよい。また、文字列の末尾には文字「\0」が入っているため、これを使って文字列の末尾に到達したことを判定できる。■

**発展課題6　ポインタを使ったオリジナルのscanfの実装**

c = getc(stdin)によって標準入力（キーボード入力）stdinで得られた一文字をchar型の変数cに格納できる。また、改行が押された際、cには' \n'が入る。scanfで文字列を読み込むのと同じ挙動を行う関数myscanfを実装し、main関数の中で実際に使用せよ。ただしmyscanfの引数は（scanfのようにフォーマット指定の引数は入れず）char\*型の変数strのみでよい。myscanfで読み込まれた結果をprintfを使って出力させることで正しく動いていることを確認せよ。■

**発展課題7　ポインタへのポインタを用いた多次元配列（この課題は難易度が高い）**

ポインタを使って多次元配列（配列の配列）を作成し、データを格納し、内容を表示させよ。たとえば3行4列の行列を格納し、その中身を出力させよ。たとえば以下のように宣言することで、cは「int型のポインタへのポインタ」となり、int型の配列の配列が作れる。ただしint型の変数sizeの値（確保する要素数）は自分で適当な値を決めればよい。

int\*\* c = (int\*\*)malloc(sizeof(int\*)\*size);

なお、\*cや\*(c+1)もポインタになるので、それが指しているアドレスから始まる一定の領域を確保しなければ安全に利用することができない。ゆえに以下のようにmallocする必要がある。

\*c = (int\*)malloc(sizeof(int)\*size);

これを\*cだけでなく\*(c+1)や\*(c+2)等々に対しても行う必要がある。

また、ポインタが指しているアドレスをインクリメントさせる際、ポインタ演算子（\*）の結合順序はインクリメント演算子（++）よりも後なので注意が必要である。すなわち変数cがint\*\*型の場合、\*c++と書くと、\*(c++)という意味になってしまう。つまり\*cが指しているポインタの値を増加させたければ、(\*c)++ と書くこと。

forループなどでポインタpが指しているアドレスを増加させていった場合、そのループの終了後、記憶すべき範囲を超えた値がpに入ったままになっていることがある。そこにアクセスしようとすると、セグメンテーションフォルトが生じてしまう。あとでpを使用するのであれば、ループの前にpのアドレスをあらかじめ別のポインタqに格納しておき、ループを抜けた後でpにq の値をコピーし（p = q; を行い）、先頭アドレスに戻しておくという方法がある。

多次元配列でm×nのデータを格納したい場合、p = q としてから(\*p)++でn回動かして\*\*pにデータを格納していくという方法があるが、p++する前に\*pに格納されているアドレスを（n回動かす前の位置に）戻しておかないと、\*qが\*pに最後に格納されていたアドレスを指しているままなので、そこから(\*q)++で動かして中身を表示させていっても数字がすべて0、あるいはセグメンテーションフォルトを生じさせてしまうことがある。■

**3 構造体**

C言語ではオブジェクト指向言語において使用されるクラスが無いかわりに構造体という機能があり、複数の変数をまとめた新しい「データの入れ物」を作ることができる。（ただしクラスと違い、構造体にはメソッドを定義することはできない）。また、構造体を組み合わせてさらに複雑な別の構造体を作ることもできる。

構造体はstructで始まるブロックを使い、include文とmain関数の間で定義する。

// sample 3-1

#include <stdio.h>

#include <stdlib.h>

struct person {

char name[30];

char address[40];

};

int main(){

int i;

struct person friends[3];

for(i = 0; i < 3; i++){

printf(“What is the name of friend %i?\n”, i);

scanf(“%s”, friends[i].name);

printf(“What is the address of %s?\n”, friends[i].name);

scanf(“%s”, friends[i].address);

}

printf(“\n”);

for(i = 0; i < 3; i++){

printf("Friend %d: %s, ”, i, friends[i].name);

printf("lives in %s\n”, friends[i].address);

}

}

　構造体はint型やchar型と同じように扱えるので、ここではperson構造体の配列friendsを作っている。構造体personの型を持つ変数を宣言する場合、変数名の前にstruct personと書く。

構造体に組み込まれた変数は**メンバ**（member）と呼ばれる。メンバにアクセスするには「.」（ピリオド）を使う。ここではfriends[i].name というようにアクセスしている。

typedefキーワードを使い、構造体宣言の後にそのエイリアス（別名） を書くことで、structを省略できる。ここではpersonDataを同義語としているが、構造体名と同じpersonをエイリアスとしてもよい。typedefは構造体に限らず、任意の型に別名を付けるのに使える。たとえばtypedef int number;とすると、number a = 10;のように、numberによってint型を表せるようになる。

// sample 3-2

#include <stdio.h>

#include <stdlib.h>

typedef struct person {

char name[30];

char address[40];

} personData;

int main(){

int i;

personData friends[100];

}

**基本課題14　構造体・構造体配列**

「商品コード」（10文字未満の文字列）、「商品名」（40文字未満の文字列）、「値段」（整数値）の三つのメンバを持つ構造体を宣言する。その構造体型のデータを最大100個格納できる配列を定義し、goods.txtの内容を配列の先頭の4個の値として設定せよ。goods.txtは以下の内容のファイルである。

ABC-12 chocolate\_cake 600

CDE-34 apple\_pie 500

FGH-56 sparkling\_wine 3000

IJK-78 potato\_chips 120

実行時に構造体の内容が以下の形で画面に出力されるようにせよ。

商品コード：ABC-12

商品名：chocolate\_cake

値段：600

商品コード：CDE-34

商品名：apple\_pie

値段：500

なお、「商品コード：」等のラベルは構造体の値の一部として格納するのでなく、出力時にprintfの中で付け加えるようにせよ。コマンドライン上で以下のようにリダイレクションという機能を使うと、goods.txtの中身をキーボード（標準入力）から打ち込んだように処理させることができる。なお、読み込むデータの数はプログラムの中であらかじめ決めておいてよい。

./prob13 < goods.txt

リダイレクションでファイルを読み込む場合、scanfを使って以下のようにEOFと一致するまでループさせるという方法がある。

while( scanf(“%s %s %d”, id, name, &price) != EOF){ }

EOFはEnd of Fileの略である。ただし以下ではstrに各行の最初の空白までしか入らないので、行全体は読み込めない。■

int型へのポインタやchar型へのポインタを作れるのと同様に、構造体へのポインタを作ることができる。sample3-3はその例である。personData型へのポインタであるfriendsの指している値のメンバにアクセスする場合、(\*friends).nameのように書くことになるが、括弧が多いとプログラムが読みにくくなるため、代わりにfriends->nameと書くことができる。「->」は**アロー演算子**（矢印演算子）と呼ばれる。アロー演算子は単なる略記法である。

// sample 3-3

#include <stdio.h>

#include <stdlib.h>

typedef struct person {

char name[30];

char address[40];

} personData;

int main(){

int i;

personData \*friends, \*addressbook;

friends = (personData\*)malloc(sizeof(personData)\*3);

addressbook = friends;

for(i = 0; i < 3; i++){

printf(“What is the name of friend %i?\n”, i);

scanf(“%s”, (\*friends).name);

printf(“What is the address of %s?\n”, (\*friends).name);

scanf(“%s”, (\*friends).address);

friends++;

}

printf(“\n”);

friends = addressbook;

for(i = 0; i < 3; i++){

printf("Friend %d: %s, ”, i, (\*friends).name);

printf("lives in %s\n”, (\*friends).address);

friends++;

} }

**基本課題15　アロー演算子**

sample3-3において、構造体へのポインタの変数のメンバにアクセスしている部分をアロー演算子を使った形に書き換えよ。

**基本課題16　構造体のネスト**

構造体のメンバの型として他の構造体を使うことができ、**構造体のネスト**と呼ばれる。すでに作成した構造体person型のceo（経営者）、ならびにperson型の配列employee（従業員）をメンバとする構造体companyの型を持つ変数を作成せよ。また、それに値を格納し、その内容を出力させるプログラムを作成せよ。■

　構造体の定義の中に自分自身を使うことができる。すなわち構造体を再帰的に定義できる。これを使ってたとえば**リスト**（list）と呼ばれる非常に重要なデータ構造を作成できる。リストとは一列に並んだデータを格納するデータ構造であり、個々のノードはそのノードに格納されるデータと後続ノードへのポインタから構成される。

ポインタ

データ

ポインタ

データ

ポインタ

データ

ヌル値

データ

　リストはC言語の場合はたとえば以下のように作成する。これは観光ツアーのプラニングで使うような訪問観光地の系列を入れるためのリストである。そのためノードを表す構造体の名称をsite（場所）にしているが、ノードの名称（構造体名）は何であってもよい。nameは観光地の名称、townNameはそれが所在する市町村名を入れるためのメンバである。

struct site {

char name[50];

char townName [20];

struct site\* next;

};

site型の各変数のメンバnextにその次に向かうべきsiteのアドレスを格納でき、データのチェーン（リスト）を作っていける。

**基本課題17　構造体における再帰的定義（リスト）**

リストに具体的なデータを格納した上で（これはファイルを読み込むのでもユーザに入力させるのでもなく、main関数の中にデータを直書きするのでよい）、その中身を出力させるプログラムを実装せよ。その際、再帰的に定義された関数を使ってもよい。実際、再帰的な関数が有益であるのは、再帰的に定義されたデータ構造を扱いやすいことも一因である。再帰的なデータ構造はあらゆる場面で現れるため、再帰的な関数も頻繁に使われる。■

　リストは非常に汎用性の高いデータ構造であり、たとえばLISPというプログラミング言語ではプログラム自体もリストで構成される。

配列と比較した場合のリストのメリットは、ノードの追加や削除が容易なことである。配列ではたとえばi番目の要素とi+1番目の要素の間に追加を行うためにはi+1番目以降の要素をすべてずらさなくてはならないが、リストではi番目のノードのポインタが新しいノードを指すようにし、新しいノードのポインタがi+1番目のノードを指すようにすればよい。配列では削除によって空の要素が生じてしまうが、リストではつなぎ替えによって空なノードを残さずに削除が行える。

**発展課題8　リストへのデータの格納**

ユーザが入力したデータをリストに格納し、終了後、その中身を（リストを辿ることで）すべて出力させるプログラムを作成せよ。たとえばwhile文の中でsite型の変数newSiteを作成し、そのアドレスをひとつ前の変数existingSiteのメンバnextに格納することでリストを繋げていける。なお、site型の変数newSiteではなくsite型のポインタ（site\*型）の変数を作る形でもよい。その場合、while文の中でmallocによって領域を確保することになる。配列では作成時に長さを決めておかなくてはならないが、リストでは任意の長さに伸ばしていくことができることがメリットのひとつである。■

**発展課題9　リストへのデータの追加**

すでに存在するリストに対し、ノードを挿入するプログラムを作成せよ。すなわちあらかじめ（前問のプログラムを使うなどして）リストを用意しておき、ユーザにデータと番号iを入力させ、リストのi番目のノードとしてそのデータを追加するプログラムを作成せよ。ポインタの指す先の書き換えによって実現させること。配列ではその中間に要素を追加した場合、それ以降の要素をすべて動かす（ずらす）必要があるが、リストではその必要がない。■

**発展課題10　構造体における再帰的定義（木）**

ひとつのノードに二つ以上のポインタを持たせることで、枝分かれしていくデータ構造が作成できる。**木**(tree)とは合流の無い連結な有向グラフである。ノードAの持つポインタがノードBを指している場合、AはBの親、BはAの子と呼ばれる。親を持たないノードは根（root）と呼ばれ、子を持たないノードは葉（leaf）と呼ばれる。子を持つノードは内部ノード（internal node）と呼ばれる。木を表す構造体を作成し、生物の系統樹に関わるデータ（知っている範囲でよい）を格納せよ。葉ノードには生物名を格納させること。各ノードが持つ子の数は固定、たとえば2でよい（可変長にしなくてよい）。各ノードが2つの子を持つ木は二分木と呼ばれる。木に格納するデータ自体はユーザに入力させるのではなく、main関数の中に直書きするのでよい。木の深さは2程度でよい。■

4 **マクロとインライン関数**

プログラムで定数（値が変化しない変数）を使う場合、そのために変数を使うと毎回変数の値を参照する必要があり、時間的コストが掛かってしまう。

たとえば物理シミュレーションで重力加速度を使う場合、使うたびに9.80665と書いただけではそれが何を表しているのかが分からず、コメントが必要になってしまう。一方、float grav = 9.80665; と定義してgravを使うようにすると、コードの解釈は容易であるが、gravを使うたびに計算機が変数の値を読みに行く（メモリを参照する）  
マクロはコンパイル前に実行されるプリプロセッサによってその値に置換される変数であり、実行時に変数を用意し格納するという処理が不要になる。たとえば以下のように書くと、GRAVはコンパイル時に9.80665に置換されるため、printfによってg = 9.80665と表示される。

#include <stdio.h>

#define GRAV 9.80665

int main(){

printf("g = %f\n", GRAV);

}

マクロは通常の変数と区別するため、すべて大文字で書くのが慣例である。

**発展課題11　 マクロ**

円周率をマクロに格納しておくことで、ユーザが入力した半径に対して円の面積を返すプログラムを実装せよ。■

5 **ファイル入出力**

ファイル入出力には関数fopenならびにgetcやputcが使用できる。まず、ファイルへのアクセスを可能にするために、ファイルポインタと呼ばれるものを作成する。fopenによってファイルをそのファイルポインタに関連付ける。

以下のプログラムではファイルポインタfpをgetcに与えることで、そこから一文字ずつ文字列を読み込んでいる。なお、getcの戻り値はchar型ではなくint型である点に注意が必要である。また、putcは文字を出力する関数であり、第2引数にファイルポインタを指定するが、ここでstdout（標準出力）を指定することで画面に出力が行われる。

関数feofはfile end ofの略であり、ファイルポインタfpがファイルの終わりに到達した時に1を返す。この時にbreakすることでループから抜け出ている。また、ferrorはfile errorの略であり、ファイル読み込みに際して何らかのエラーが生じた場合に1を返す関数である。

// sample 4-1

#include <stdio.h>

#include <stdlib.h>

int main(){

FILE \*fp;

int c;

fp = fopen("english.txt", "r");

if(fp == NULL){

printf("file could not be opened.\n");

return;

}

while(1){

c = getc(fp);

if(feof(fp)){

break;

}

if(ferror(fp)){

printf("stream error.\n");

break;

}

putc(c, stdout);

}

}

**発展課題12　 ファイル読み込み**

ファイルの中身を1文字ずつ読み込むには以下のように関数getcが使える。getcはファイルの終わりまで来ると-1を返すので、戻り値が-1になるまでwhile文をまわすことでファイルの全体を読み込める。以下ではprintfを使うことで標準入力（stdin）に流し込んだファイルの全体を表示させている。なお、printfの中でcをchar型にキャストしているのはgetcの戻り値がint型であるためである。

int c;

while((c = getc(stdin)) != -1){

printf("%c", (char)c);

}

また、scanfを使って以下のようにEOFと一致するまでループさせるという方法がある。EOFはEnd of Fileの略である。ただし以下ではstrに各行の最初の空白までしか入らないので、行全体は読み込めない。

while( scanf(“%s”, str) != EOF){ }

いずれかを使い、英語の文書ファイルの終わりまで読み込み、文字数を出力させるプログラムを作成せよ。ただし空白や改行文字は文字数に入れても入れなくてもよい。入力する文章のサンプルenglish.txtを用意してあるので、動作確認の際にはリダイレクション（例： ./a.out < english.txt）を用いてenglish.txtを標準入力（stdin）に流し込むとよい。ただしa.outはコンパイルによって得られる実行可能ファイルである。■

**発展課題13 ファイル入出力**

ファイルgoods.txtの内容を1文字ずつ読み出し、goods2.txtという名前のファイルに出力するプログラムを作成せよ。なお、1文字ごとの出力には標準ライブラリ関数putc()が使用できる。foutを出力先ファイルへのファイルポインタとすると、以下のようにして文字cをファイルに書き出せる。

putc(c, fout);

■