プログラミング2　第7回

講義計画（暫定）

- ~~第1回(11/28(金))　ガイダンス、Cプログラミングの基礎（データ型、変数、四則演算、制御構文）~~
- ~~第2回(12/05(金))　ビット演算、配列、関数~~
- ~~第3回(12/12(金))　配列と関数、文字列、ソート~~
- ~~第4回(12/19(金))　再帰関数、ポインタ(1)~~
- ~~第5回(12/26(金))　ポインタ(2)~~
- ~~第6回(01/ 9(金))　構造体・共用体、リスト~~
- 第7回(01/23(金))　Cプログラミング実用上必須のもの（defineマクロ、ファイル操作、複数ソースコンパイルなど）
- 第8回(01/30(金))　期末試験（**出席必須！**）

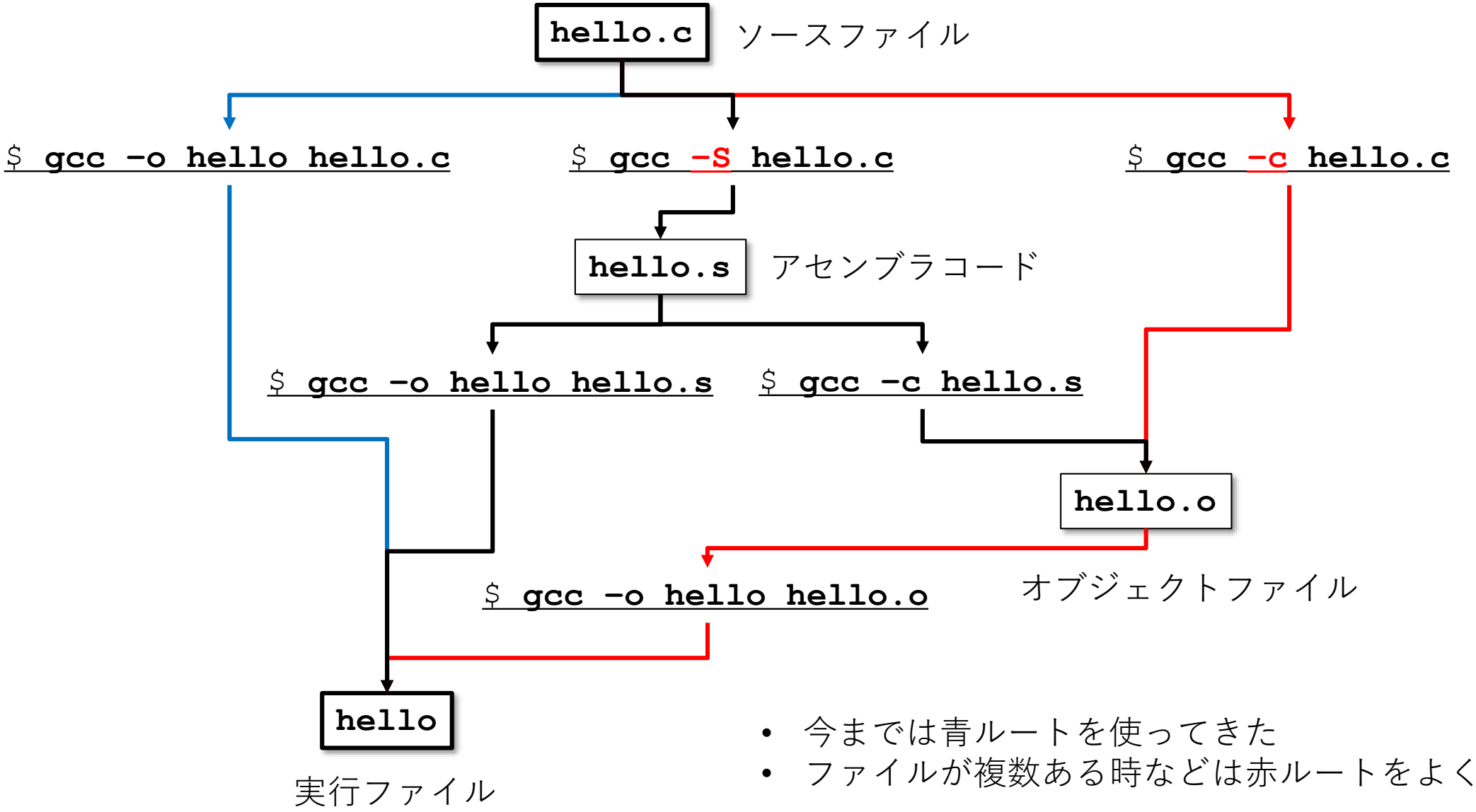
期末試験について

- 必ず現地で参加してください
  - 当日やむを得ない理由で来られなくなった場合は連絡してください
- その場でプログラムを書いてもらいます
  - プログラムを実行できる環境を用意してください
- Web検索無し、講義資料の閲覧もなし
- これまでの演習で作ったプログラムは閲覧OK
  - 演習と似た問題を一問以上出します
- テストの満点は60点だが、50点を超えた分は切り捨てて、成績への反映は最大50点とする
  - 演習も同様に最大50点。足し合わせた点数が最終成績になる
    - ただし、成績(A+, A, B, C, C-）への変換は、標準的な換算方法**ではなく**、相対評価で行う
      - 一昨年度実績： A+: 100, A: 85-99, B: 70-84, C: 65-69, C-: 60-64
      - 昨年度実績：標準的な換算法と同じ

コンパイラの仕組み（gccの例）

- gcc コマンドだが、実は以下の3ステップに分けられる
  1. プリプロセッサ(preprocessor) & コンパイラ(compiler)
     - コマンド名 cc1
     - （実質0ステップ目）プリプロセスとして“#include …”などの#から始まる部分を処理
     - コンパイルしてアセンブラコード(assembler code) (*.s)を出力
  2. アセンブラ(assembler)
     - コマンド名 as
     - アセンブラコードをオブジェクトファイル(object file) (*.o)に変換
  3. リンカ(linker)
     - コマンド名 collect2
     - リンカコマンドldはcollect2から呼び出し
     - オブジェクトファイルやライブラリを結合（リンク）して実行ファイルを出力

gccによるCプログラムコンパイルの流れ



- 今までは青ルートを使ってきた
- ファイルが複数ある時などは赤ルートをよく使う（後述）

アセンブラコードの取得

- -S を付けてコンパイルすると、第1段階で止まり、
  アセンブラコードが拡張子*.sファイルで出力される

```
$ gcc -S hello.c
$ ls
... hello.s ...
```

- アセンブラコード（ファイル）を表示

```
$ less hello.s
```

アセンブラコードの例 →

```
.file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello, World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
（以下省略）
```

プリプロセッサディレクティブ

- プリプロセッサ（preprocessor）
  - コンパイルに先立ってソースファイルに前処理を行うプロセッサ
  - プリプロセッサへの命令はディレクティブ（directive）と呼び、シャープ(#)から始まる
- #includeプリプロセッサディレクティブ
  - 山括弧（< >）やダブルクォーテーション（" "）で囲って指定した外部ファイルの内容をその場所に取り込む
  - ヘッダーファイルの取り込みで頻繁に用いる
- #defineプリプロセッサディレクティブ
  - ある文字列を別の文字列に変換
  - 定数の定義や簡単な演算の関数形式での表現などに用いる
- その他のディレクティブ
  - #if, #elif, #else, #ifdef, #ifndef, #endif, #undef, #error, #line, #pragma

#defineプリプロセッサディレクティブ

- プログラム中の文字列をプリプロセッサ（コンパイルの前）で置換する機能
- オブジェクト形式マクロ（object-like macros）
  - オブジェクト形式マクロの一般的な形式

```
#define マクロ名 置換する文字列
```

  - 例

```
#define N (10)
int a[N];
```

括弧はなくても良いが、
あると安全

プリプロセッサによって
int a[(10)];
に置換

- 関数形式マクロ（function-like macros）
  - 関数形式マクロの一般的な形式

```
#define マクロ名(引数リスト) 置換する処理
```

  - 例

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
c = MAX(x, y);
```

プリプロセッサによって
c = ((x) > (y) ? (x) : (y));
に置換

プリプロセスの結果

- gccに-Eオプションをつけると、プリプロセスの結果を確認できる

```
#include <stdio.h>

#define N 20
#define MUL_BAD(X,Y) X * Y
#define MUL_OK(X,Y) (X) * (Y)

int main(){

  int n = N;
  int m = N*2;

  int mul_bad = MUL_BAD(n+1,m+1);
  int mul_ok = MUL_OK(n+1,m+1);

  return 0;
}
```



```
/*
  stdio.h のファイルを展開した結果が
  記述される。すごく長い。
  printfのプロトタイプ宣言などが書かれている
*/



int main(){

  int n = 20;
  int m = 20*2;

  int mul_bad = n+1 * m+1;
  int mul_ok = (n+1) * (m+1);

  return 0;
}
```

Nが展開されて20に

X,Yがそのまま展開されるので、定義するときに()で囲んでおかないと、意図せぬ展開が行われる

ヘッダーファイルの中身

- lessでstdio.hを表示
  - stdio.hファイルの場所は環境によって異なる
  - 例えば以下のような場所にある

```
$ less /usr/include/stdio.h
```

  - 標準的な入出力（standardなinput/output）に関する様々なものが定義されている

stdio.h の一部

```
...
char *fgets(char * restrict, int, FILE *);
...
int printf(const char * restrict, ...) __attribute__((__format__ (__printf__, 1, 2)));
...
int scanf(const char * restrict, ...) __attribute__((__format__ (__scanf__, 1, 2)));
...
```

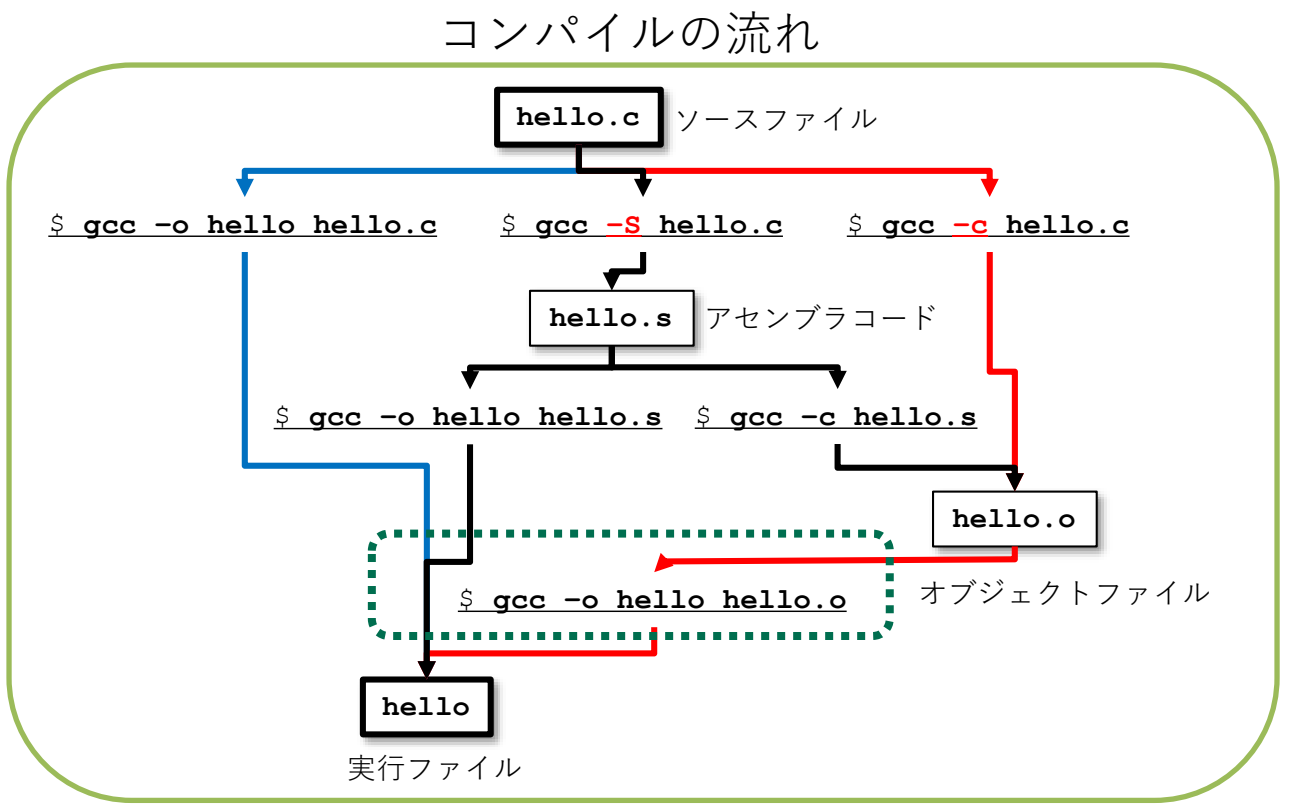
標準Cライブラリと数学関数

- 標準Cライブラリ
  - C言語処理系では、言語規格で定められた標準Cライブラリが提供されている
  - 入出力(stdio), 一般ユーティリティ(stdlib), 文字列操作(string), 整数型の大きさ(limits), 浮動小数点型の特性(float), 複素数(complex), 診断(assert), 数学(math), 日付・時間(time), など
  - どんな環境でも同様に利用できるはずなので積極的に使って良い
  - CPUの違いやOSの違いなどにより、同じように使えるが全く同じ結果が得られるとは限らないものもある点には注意

- 数学関数math
  - 数学的な演算を行う関数
  - 正弦sin, 余弦cos, 平方根sqrt, 自然対数log, 指数関数exp, べき乗pow, 浮動小数点数の絶対値fabs, など
  - 使用するにはヘッダーファイルmath.hをインクルードする
  - gccでのコンパイル時に –lm を追加で指定する
    - 実際はコンパイルの流れにおける最後のリンク時に必要となる

ライブラリの役割と使い方

- C言語におけるライブラリは、変数や定数が書かれたヘッダファイルの流用と、オブジェクトファイルの流用をスムーズに行う仕組みだと思えば良い
- ヘッダファイルには流用する価値のある変数や定数、オブジェクトファイルに収められた関数のプロトタイプ宣言が書かれている
- ライブラリファイルにはオブジェクトファイルの中身そのもの、またはその参照情報が書かれている
  - 静的ライブラリと動的ライブラリで異なるが、詳細は省略
- ヘッダファイルをincludeし、ライブラリファイルをリンクすれば、ライブラリの機能を利用できる
  - ヘッダファイルのincludeのみで完結するライブラリの場合はリンクは不要



分割コンパイル

- これまでは単一のソースファイルにプログラム全てを記述
  - コード量が大規模になった場合に様々な問題が生じる
- 複数のソースファイルに分けて分割コンパイルすることができる
  - 関数を機能ごとにまとめて複数のソースファイルに記述
  - 編集作業・プロジェクト管理が容易になる
- 複数ソースファイルの注意点
  - main()関数は全体で1つ
  - グローバル変数はファイルにまたがって有効
  - 複数ファイルになっても上から下へ読んでいくのが原則
    - include先も含めて、上に存在しない関数や変数は使えない
  - **自作のヘッダファイル経由**で、プロトタイプ宣言やグローバル変数を共有する

分割ソースファイルの例

- <>で囲んだファイルは標準ライブラリ用のディレクトリからインクルードされる
- ""で囲んだファイルはソースファイルと同じ場所から優先的にインクルードされる

main.c

```
#include <stdio.h>
#include "myfunc.h"

int global_val = 1;
int main(){
  myfunc();
  printf("hello %d\n", global_val);
  return 0;
}
```

自作関数用のヘッダーファイルをインクルード

グローバル変数の宣言

myfunc.cの関数を呼び出し

グローバル変数を出力

myfunc.c

```
#include <stdio.h>
#include "myfunc.h"

void myfunc(){
  printf("hello %d\n", global_val);
  global_val++;
}
```

自作関数用のヘッダーファイルをインクルード

main関数で定義されたグローバル変数を出力

グローバル変数をインクリメント

myfunc.h

```
extern void myfunc();
extern int global_val;
```

extern宣言（後述）

分割ソースファイルの例（ヘッダファイル展開後）

main.c

```
#include <stdio.h>
extern void myfunc();
extern int global_val;

int global_val = 1;
int main(){
  myfunc();
  printf("hello %d\n", global_val);
  return 0;
}
```

ここではメモリ確保されない

ここでメモリ確保

myfunc.c

```
#include <stdio.h>
extern void myfunc();
extern int global_val;

void myfunc(){
  printf("hello %d\n", global_val);
  global_val++;
}
```

どこかで定義されると宣言するだけ

- このようにヘッダファイルなしで書いても意味は同じ
- extern宣言
  - 宣言だけ行い、定義しない（つまりメモリ上に確保しない）方法
    - 通常の変数宣言は、宣言時にメモリ上に確保してしまうため、プログラム中に複数回書けない
    - myfunc.h はmain.cでもmyfunc.cでも展開されるので、`global_val`変数の多重定義を防ぐため、extern宣言の必要がある
    - 一方、関数のプロトタイプ宣言は元々宣言だけでメモリ上に確保しないため、extern宣言しなくても問題は起こらない
  - 定義はプログラム中のどこか一箇所で行われる

複数ソースファイルのコンパイル方法

- 方法1 （コマンド1回）

```
$ gcc myfunc.c main.c
```

- 方法2 （1つずつコンパイルして最後にリンク）

```
$ gcc -c myfunc.c
$ gcc -c main.c
$ gcc myfunc.o main.o
```

→ myfunc.oが生成される
→ main.oが生成される

ファイル数が増えてきた時、Makefile（今回は説明しない）にコンパイルルールを記述して、makeコマンドでコンパイルすることが一般的

- 実行結果

```
$ ./a.out
hello 1
hello 2
```

`global_val`がファイルを跨いで共有されている

インクルードガード（二重インクルードの防止）

- ヘッダファイルの二重読み込みが問題になるケースがある

```
struct student {
  char *name;
  int id;
  int grade;
};
```

st_student.h

ヘッダファイル内で構造体studentの宣言

```
#include "st_student.h"
struct class{
  struct student *students;
  int num_of_students;
};
```

st_class.h

ヘッダファイル内でst_student.hをインクルード、構造体classの宣言

```
#include "st_student.h"
#include "st_class.h"
int main(){
  struct student taro;
  struct class cs;
```

main.c

st_student.hとst_class.hを両方インクルードした結果、studentが二重に宣言される

- これを防止するために、”インクルードガード”が使われる

```
#ifndef _ST_STUDENT_H
#define _ST_STUDENT_H
struct student {
  char *name;
  int id;
  int grade;
};
#endif
```

- #ifndef ○○：○○が未定義（#defineされていない）なら#endifまでのコードが読み込まれる。定義済みなら無視される。
- 一回目の読み込み時は_ST_STUDENT_Hが未定義なので内部のコードが読み込まれるが、二回目の読み込み時には_ST_STUDENT_Hが定義済みなので無視される。

ファイル入出力

- ファイル入出力はストリームとして扱う

- ストリーム（stream）
  - ファイル（file）などの入出力デバイスにアクセスする際に、その種類に左右されずにアクセスするための抽象レベル
    - ディスクファイル，画面，キーボード，メモリ，ポートなど様々な入出力デバイスをファイルと考える
    - ＝ファイル以外も同じようなプログラム（コード、関数）で制御できる
  - テキストストリーム（text stream）とバイナリストリーム（binary stream）が存在
  - 現在位置（current position）の概念が存在
    - 例：50バイト読み込んだあとで、さらに読み込みを行うと、51バイト目から読み込まれる

ファイルシステムの基礎

- ファイルにアクセスする手順
  - ファイルを開く（open）
    - **fopen()**を利用
    - ファイルを読み書きするためのストリームが生成される
    - 読み込みか書き込みか、テキストかバイナリかなどのモード指定を行う
  - ストリームに対して読み書きを実行する
    - **fprintf(), fscanf(), fputs(), fgets()**などを利用
    - 原則としてファイルの先頭から順番に読み書きする
  - ファイルを閉じる（close）
    - **fclose()**を利用
    - ストリームが廃棄される

ファイルを開く

- ファイルを開いてストリームと結びつける
- fopen()の一般的な形式

```
FILE *fopen(char *ファイル名, char *モード);
```

  - 戻り値はFILE型オブジェクトを指すFILE*であり、利用者は内部構造を直接操作せず、ストリームとして扱う。
  - ファイルを開く操作に失敗するとNULL （ヌルポインタ）を返す
  - 主なモード
    - テキストでの読み込み("r")，書き込み("w")，追加書き込み("a")
    - バイナリでの読み込み("rb")，書き込み("wb")，追加書き込み("ab")

よくある使い方の例

```
char fname[] = "file.dat";
FILE *fp;

if((fp = fopen(fname, "r")) == NULL){
  printf("Failed to open file %s¥n", fname);
  exit(EXIT_FAILURE);
}
```

```
if(  ( fp = fopen(fname, "r") ) == NULL  ) {
```

- fpにfopenの結果を代入し、その結果がNULLと等しいかどうかを判定している
- 代入と比較を一度に記述するという、C言語プログラムでよく使う書き方

モード一覧

※{r, w, a}と{b, +}の組み合わせ

モード	意味	ファイルが存在しない場合の動作
r	テキストでの読み込み	NULLを返す
w	テキストでの書き込み	新規に作成
a	テキストでの追加書き込み	新規に作成
rb	バイナリでの読み込み	NULLを返す
wb	バイナリでの書き込み	新規に作成
ab	バイナリでの追加書き込み	新規に作成
r+	テキストでの読み書き	NULLを返す
w+	テキストでの読み書き	新規に作成
a+	テキストでの追加読み書き	新規に作成
r+b	バイナリでの読み書き	NULLを返す
w+b	バイナリでの読み書き	新規に作成
a+b	バイナリでの追加読み書き	新規に作成

- r 系だけは対象ファイルがあらかじめ存在している必要がある
- w系はファイルが存在していると中身が消去される

- r+系もw+系も読み書き両方可能であるが、あらかじめファイルが必要か、既にあるファイルがどうなるか、が異なる

ファイルを閉じる

- ストリームに結び付けられているファイルを閉じて、ストリームを廃棄する
- fclose()の一般的な形式

```
int fclose(FILE *ストリーム);
```

  - 引数はfopen()の戻り値を指定する
  - 戻り値は成功すると0、エラーが起こるとEOF（End of File）

  - 正常終了時には処理系が後始末することが多いが、バッファされた出力や書き込みエラーを正しく処理するため、明示的にfclose()し、戻り値も確認する。

1文字単位の入出力

- 読み込みfgetc()
  - ストリームから次のバイトをunsigned char型の値として読み込み、int型の値として返す
  - エラーもしくはファイルの最後に到達したときはEOFを返す
  - 一般的な形式

```
int fgetc(FILE *ストリーム);
```

- 書き込みfputc()
  - ストリームにint型（char型でも良い）の文字をunsigned char型の値として書き込む
  - 一般的な形式

```
int fputc(int 文字, FILE *ストリーム) ;
```

一文字ずつファイル書き込みの例

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char str[] = "School of Informatics";
  char fname[] = "test.txt";
  FILE *fp;
  char *p;

  if((fp = fopen(fname, "w")) == NULL){
    printf("Failed to open file %s\n", fname);
    exit(EXIT_FAILURE);
  }

  p = str;
  while(*p){
    if(fputc(*p, fp) == EOF){
      printf("Failed to write file %s\n", fname);
      exit(EXIT_FAILURE);
    }
    p++;
  }
  fclose(fp);
  return 0;
}
```

ファイルを書き込みモードで開く

fopen()が失敗していたら失敗終了

文字列strの先頭アドレスをpに代入

pが'¥0'になるまで繰り返し処理

fputc()で1文字出力、失敗したら終了

pが指すアドレスを1文字分進める

これでstrに格納された文字列を全て出力できる

一文字ずつファイル読み込みの例

```
char fname[] = ”test.txt";
FILE *fp;
int ch;

if((fp = fopen(fname, "r")) == NULL){
  printf("Failed to open file %s¥n", fname);
  exit(EXIT_FAILURE);
}

while((ch = fgetc(fp)) != EOF){
  putchar(ch);
}

fclose(fp);
```

ファイルを読み込みモードで開く

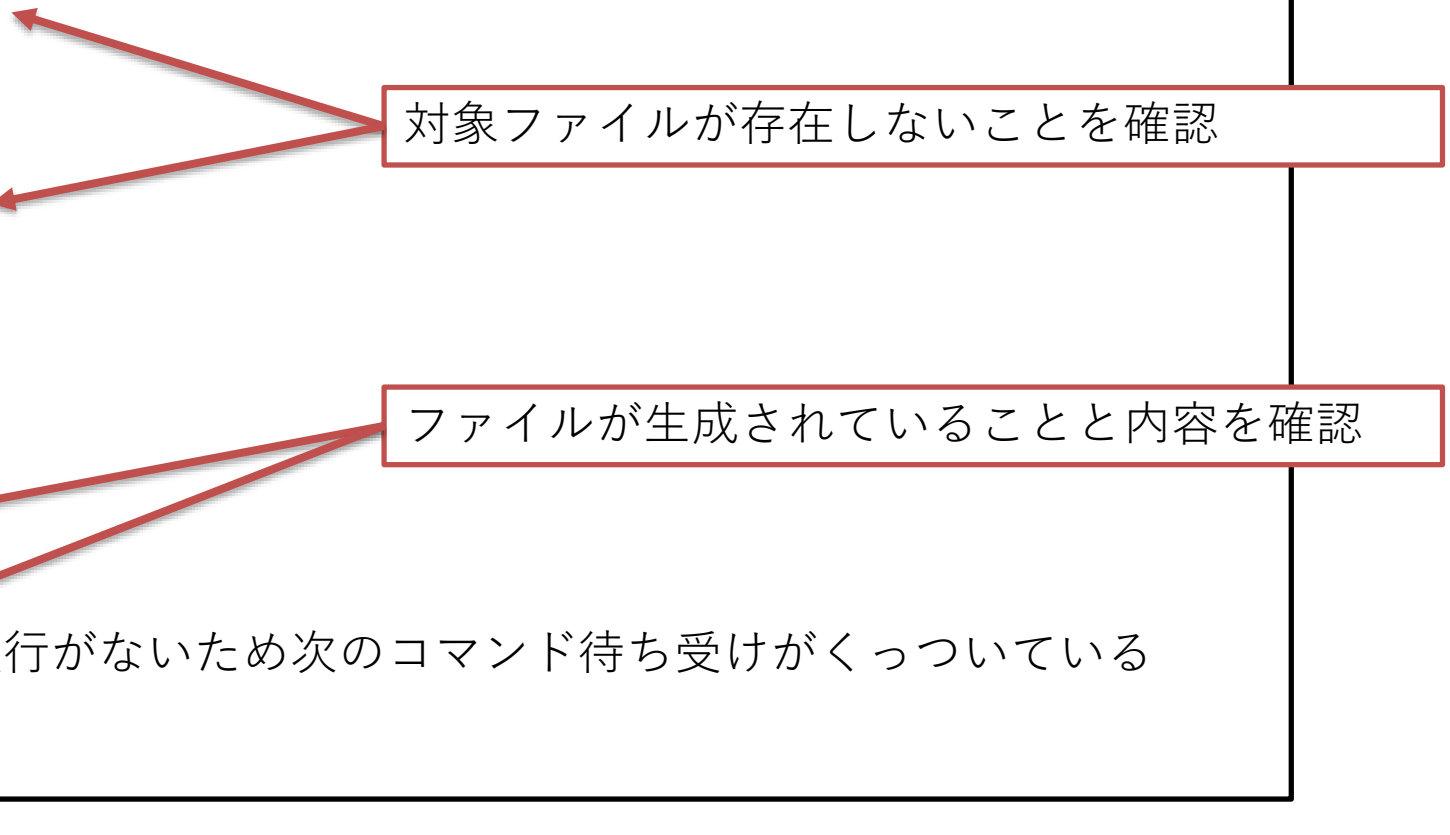
fopen()が失敗していたら失敗終了

fgetc()で1文字入力し変数chに代入、失敗したら終了

putchar()で画面に文字を出力

実行結果例

```
$ gcc -o write write.c
$ gcc -o read read.c
$ cat test.txt
cat: test.txt: そのようなファイルやディレクトリはありません
$
$ ./read
Failed to open file test.txt
$
$ ./write
$ cat test.txt
School of Informatics$
$
$ ./read
School of Informatics$
```



feof()とferror()

- fgetc()の問題点
  - エラーもしくはファイルの最後に到達したときに戻り値としてEOFを返すが、どちらなのか判断できない
  - バイナリファイルから読み込んだ値がEOFと同じビット列である場合に区別できない
- feof()とferror()
  - feof()はファイルの終わりに到達していれば0以外、そうでなければ0を返す
  - ferror()はエラーが発生していれば0以外、そうでなければ0を返す
  - 一般的な形式

```
int feof(FILE *ストリーム);
int ferror(FILE *ストリーム);
```

高レベルテキスト関数

- fgets()とfputs()
  - 標準入力からのgets()、標準出力へのputs()と同等だが、微妙に動作が異なる点がある
    - ※gets() はC11で標準ライブラリから削除されており、使用してはいけない。
  - 一般的な形式

```
char *fgets(char *文字列, int 最大読み込み文字数, FILE *ストリーム);
int fputs(char *文字列, FILE *ストリーム);
```

- fscanf()とfprintf()
  - 標準入力からのscanf()、標準出力へのprintf()と同等の操作をファイルに対して行う
  - 一般的な形式

```
int fscanf(FILE *ストリーム, char *制御文字列, 変数リスト);
int fprintf(FILE *ストリーム, char *制御文字列, 変数リスト);
```

バイナリデータの入出力

- fread()とfwrite()
  - テキスト形式で数値を読み書きする場合、コンピュータ内部での表現とASCIIテキストとのデータ変換が必要になる
    - データ変換によって厳密に同じ値とならない場合がある　→　できるだけ「生」の状態で読み書きしたい
    - 読み書きの時間や生成されるファイルのサイズなどはバイナリ形式の方が効率的
    - ただしバイナリ形式は、別環境でそのまま読める保証はない
  - 一般的な形式

```
size_t fread(void *バッファ, size_t サイズ, size_t 数値, FILE *ストリーム);
size_t fwrite(void *バッファ, size_t サイズ, size_t 数値, FILE *ストリーム);
```

  - 利用例

```
int buf[10];
fp = fopen("file.dat", "rb");
fread(buf, sizeof(int), 10, fp);
fclose(fp);
```

```
double buf[20];
fp = fopen("file.dat", "wb");
fwrite(buf, sizeof(double), 20, fp);
fclose(fp);
```

ランダムアクセス，その他の関数

- fseek()とftell()
  - 大きなファイルを取り扱う場合など、ファイルの任意の箇所からアクセスしたい場合に使う
  - int fseek(FILE *ストリーム, long int オフセット値, int 基準位置定数);
    - 基準位置からオフセット値分だけ移動させる
    - SEEK_SET：ファイルの先頭、SEEK_CUR：現在位置、SEEK_END：ファイルの末尾
  - long ftell(FILE *ストリーム);
    - 現在位置の取得

- その他の関数
  - int rename(char *旧ファイル名, char *新ファイル名);
  - int remove(char *ファイル名);
  - void rewind(FILE *ストリーム);　※位置を先頭に戻す
  - int fflush(FILE *ストリーム);

標準ストリーム

- システムが提供する特殊なストリーム
  - stdin　　標準入力（standard input）
  - stdout　　標準出力（standard output）
  - stderr　　標準エラー（standard error）
  - プログラム起動時に開かれているため、プログラム中で明示的にオープンする必要はない（できない）
  - 入出力関数で標準ストリームを指定するとコンソール入出力関数と同じ意味になる
  - 例

```
printf("Mt. Fuji is %d meters high¥n", h);
```

```
fprintf(stdout, "Mt. Fuji is %d meters high¥n", h);
```

ファイルI/Oとバッファ

- 今回扱っているファイルI/O関数群（f~）は「高水準入出力関数」と呼ばれ、ファイルの読み書きにバッファを用いている
- 操作関数が実行される度に毎回ディスクにアクセスが行われるとは限らず、まとめて行うことで性能を向上させている
  - メモリ上にデータを蓄積し、読み込み命令に対してメモリ上の内容を返したり、バッファが一杯になるまではディスクへの書き込みを行わないことで、ディスク操作回数を減らして実行時間を短縮させている
- タイミングがシビアなプログラムの実行時や、エラー終了時のログ確認などでトラブルの元となることがある
- setvbufでバッファモードを変更したり、適宜fflushを行うことで、バッファの影響を小さくすることができる

昨年の期末テスト（提出不要）

（ただし似た問題を一問以上出します）

プログラミングテスト

- スケジュール
  - TACTに普段の演習課題と同様の方法で提出する
  - テストの時間は15:00までとする
    - 15:00までしか提出できません！
  - 終わった人は早く退出して休憩時間にして良い
  - 15:15からテストの解説を行う

注意事項１

これは去年のもので、提出不要です。

2024年度期末テスト

- TACTのリソースの「期末テスト」から、問題のPDF及び雛形をダウンロードして使う
  - 採点の都合上、**必ず雛形を使うこと**
  - **指定された関数以外の改変は不可**
    - デバッグのためにprintf文などを入れた場合は、提出前にコメントアウトすること
- TACTの課題のページの「期末テスト」に普段のレポートと同様にソースコードをアップロードする
  - ソースファイルの名前は　学籍番号-8-{問題番号}.c　とすること　例：12345678-8-4.c
  - 結果のファイルは不要
- 自身のPCでコード作成や実行をしても問題はない（はず）
  - 環境に依存する問題は出していないつもり
  - 何か環境問題が発生した場合にはその時に対応を考えます
- gccの拡張などは使用しないこと
  - 関数内関数などは禁止

注意事項２

- 講義資料やWebの閲覧なし
- これまで作ったプログラムは閲覧OK
- テストの満点は60点だが、50点を超えた分は切り捨てて、成績への反映は最大50点とする
  - 演習も同様に最大50点。足し合わせた点数が最終成績になる
    - ただし、成績(A+, A, B, C, C-）への変換は、標準的な換算方法**ではなく**、相対評価で行う
      - 昨年度実績： A+: 100, A: 85-99, B: 70-84, C: 65-69, C-: 60-64

問題1：Fizz Buzz遊び（6/60点）

- 以下の仕様に基づいて、言葉遊びである「Fizz Buzz」を模擬するプログラムを作成せよ。
  - コマンドライン引数でnの値を入力する
    - nの値はintの表現可能範囲の正整数であることを仮定していい
  - 1からnまでの数字を画面に表示する、ただし……
  - 3で割り切れる場合は数字の代わりに Fizz と表示
  - 5で割り切れる場合は数字の代わりに Buzz と表示
  - 3と5の両方で割り切れる場合は数字の代わりに Fizz Buzz と表示(FizzとBuzzの間はスペース)
  - 毎回改行すること
- 雛形ex_01.cを利用する
  - FizzBuzz関数以外はいじらないこと

実行例

```
$ ./a.out 15
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
Fizz Buzz
```

問題2：逆さ言葉（6/60点）

- 以下の仕様に基づいて、入力された文字列を反転させるEmordnilap関数を作成せよ
  - コマンドライン引数で文字列を入力する
    - 文字列はアルファベットであることを仮定していい
  - char* Emordnilap(char *src)関数は、引数の文字列srcを反転させて返す関数
  - 逆さにするのは終端文字の手前までで、返す文字列の最後に終端文字を入れること
  - 以下は長さ10の文字列を確保するmallocの例
    - char* str = malloc(sizeof(char)*10);
- 雛形ex_02.cを利用する
  - Emordnilap関数以外はいじらないこと
    - 従って、string.hはインクルードできず、strlen関数などは利用できないことに注意すること

実行例

```
$ ./a.out emordnilap
before: emordnilap, after: palindrome
```

問題3：2進数から10進数への変換（10/60点）

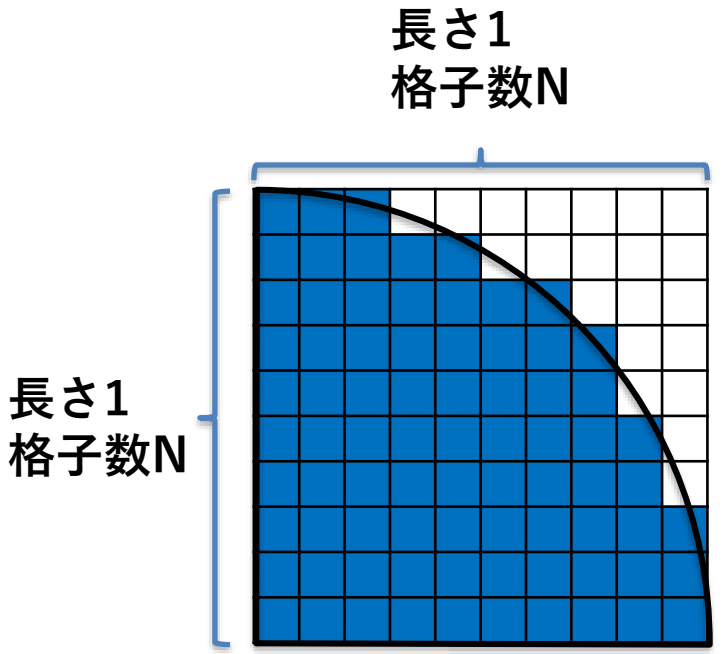
- 2進数を10進数に変換するプログラムを作成せよ
  - コマンドライン引数で2進数を入力する
    - 入力される2進数は、0, 1からなる31桁以下の文字列を仮定していい
    - 用意されたint ctoi(char c)関数は、実装内容を確認した上で利用していい（使わなくてもいい）
- 雛形ex_03.cを利用する
  - convert2to10関数以外はいじらないこと

実行例

```
$ ./a.out 1111111111
input: 1111111111, output: 1023
```

問題4：円周率の計算（10/60点）

- 円周率の近似値を求めるpi関数を作成せよ
  - コマンドライン引数で一辺の格子数Nの値を入力する
    - Nは10^4以下の正整数を仮定していい
  - 実装するdouble pi(int n) は、一辺の格子数を引数とし、計算したπの値を返す関数
- 大まかなアルゴリズム
  - 長さ1 × 1の領域をN × Nの格子に区切り、格子の**中心が**扇形に入っているかどうかを判定する
  - 図形に入っている格子数の割合が、扇形の面積の近似値になる
  - 扇形の面積 = 半径^2 * π / 4
- 雛形ex_04.cを利用する
  - pi関数以外はいじらないこと
    - 従って、math.hのインクルードはできない



実行例

```
$ ./a.out 10000
N = 10000, pi = 3.14159388
```

問題5：組み合わせ数を求めよ（12/60点）

- 組み合わせ数に関する以下の関数3つを実装をせよ
  - **式１：**なおN個の物の中からM個を選ぶ組み合わせの数$_nC_m$は次のように表される
    - $_nC_0 = 1$
    - $_nC_n = 1$
    - $_nC_m = {}_{n-1}C_{m-1} + {}_{n-1}C_m$
  - **式２：**または、次の漸化式で求められる
    - $_nC_0 = 1$
    - $_nC_m = \frac{n-m+1}{m} {}_nC_{m-1}$

1. 再帰を用いて、式1に基づき、組み合わせの数$_nC_m$求める関数combination1を実装せよ（4点）
   - コマンドライン引数でnとmの値を入力する
   - long long combination1(int n, int m) は、組み合わせの数$_nC_m$を返す関数として実装する
   - n = 30以下程度を想定して良い

次ページに続く

問題5：組み合わせ数を求めよ（12/60点）

2. 再帰を用いて、式2に基づき、組み合わせの数${}_nC_m$求める関数combination2を実装せよ（4点）
   - コマンドライン引数でnとmの値を入力する
   - long long combination2(int n, int m) は、組み合わせの数${}_nC_m$を返す関数として実装する
   - n = 30以下程度を想定して良い
3. 再帰を**使わず**、式2に基づき、組み合わせの数${}_nC_m$求める関数combination3を実装せよ（4点）
   - コマンドライン引数でnとmの値を入力する
   - long long combination3(int n, int m) は、組み合わせの数${}_nC_m$を返す関数として実装する
   - n = 30以下程度を想定して良い

- 雛形ex_05.cを利用する
  - combination1~3の関数以外いじらないこと。

```
./a.out 20 10
A1: 20C10 = 184756
combination1 time =   0.00202107 [sec]
A2: 20C10 = 184756
combination2 time =   0.00000000 [sec]
A3: 20C10 = 184756
combination3 time =   0.00000000 [sec]
```

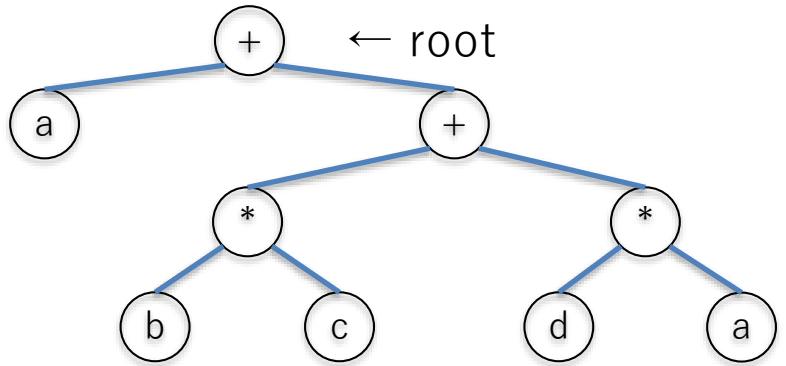
※各関数の実行時間を出力している。目安なので気にしなくて良い。実装次第では計測関数の精度上0.0秒と表示されるが、問題ない。

問題6：構文木を作成せよ（16/60点）　※他の問題ができてからやること

- コマンドライン引数に式を与える。以下の3つの関数を実装せよ。
  - 式は、小文字アルファベット1文字の変数と、足し算と掛け算の”+”, ”*” からなる
    - 引数の例外処理は考えなくて良い。
  - 引数の例
    - ./a.out “a+b*c+d*a”　※二重引用符で囲む必要あり

1. 引数の式を木構造で保存する関数create_astを作成せよ（6点）
   - struct node* create_ast(char *str, int length) は、引数の式 str から対応する木構造リストを作り、root にあたるノードのポインタを返す関数。lengthは文字列の長さ。
   - 構造体 node のメンバ char cに変数名(a, b, c, …, z)や演算子(+, *)を保存する
     - 演算子の場合、２つの子ノードを持ち、メンバleftとrightに子ノードへのポインタを保存する
   - create_astから呼び出される、追加の関数を定義してもよい

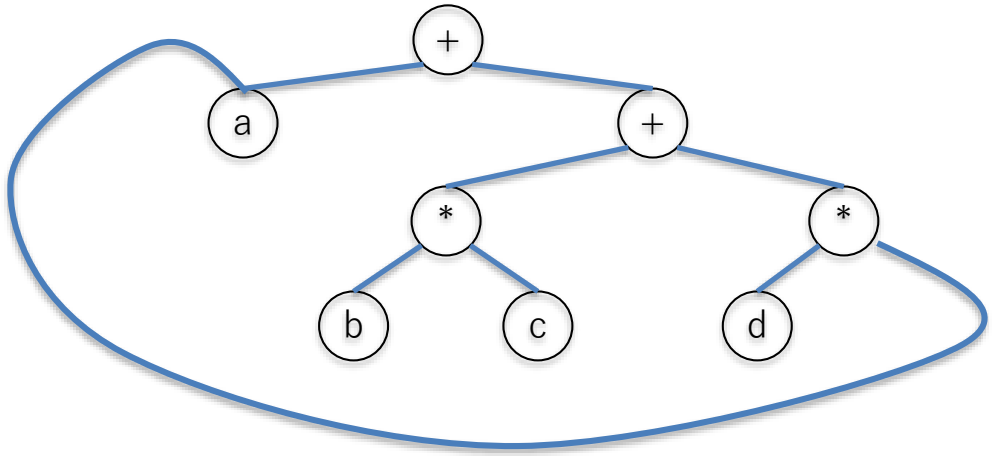
```
struct node{
  char  c;
  int   num;
  struct node *left;
  struct node *right;
};
```



次ページに続く

問題6：構文木を作成せよ（16/60点）　※他の問題ができてからやること

2. 木構造に保存されている各変数に値を取り込む、register_num関数を実装せよ（5点）
   - 具体的には、変数(a,b,c…,z)が保存されているnodeのメンバnumに、入力された値を取り込む
     - 採点の都合上、あらかじめ用意されているscan_variable関数を使って入力された値を取り込むこと
     - int scan_variable(char c)は、値を取り込む変数名を引数とし、変数名の表示、scanfによる入力（整数値）受付をし、入力された整数値を返す関数
   - ひとつの変数につき、一度だけ入力を求めること
     - 例えば “a+b*c+d*a” は式中に二回 “a”が出てくるが、入力を求めるのは一度だけ
     - 木構造中に出てくるノードについては、同じ変数名のノードが複数あっても構わない
   - register_numから呼び出される、追加の関数を定義してもよい

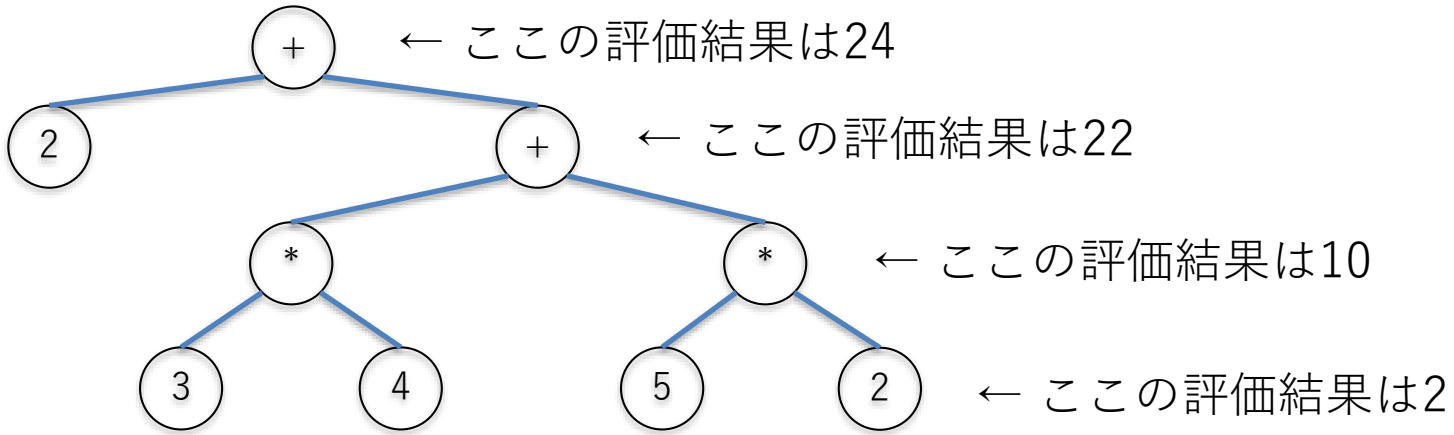


← 木構造中のaを一つにするためにこのように実装してもいいが、**ここまでは求めない（加点もしない）**

次ページに続く

問題6：構文木を作成せよ（16/60点）　※他の問題ができてからやること

3. 木構造から式の計算結果を導出する、calc_equation関数を実装せよ（5点）
   - int calc_equation(struct node *p)は、与えられたnode以下の計算式を評価し、計算結果を返す関数
   - calc_equationから呼び出される、追加の関数を定義してもよい



- 雛形ex_06.cを利用する
  - **自作関数を足して良い**。ただし、main関数などはいじらないこと。

次ページに続く

問題6：実行例など

実行例

```
$ ./a.out "a+b*c+d*a"
ex6.1: (a+((b*c)+(d*a)))
a=2
b=3
c=4
d=5
ex6.2: (2+((3*4)+(5*2)))
ex6.3: (2+((3*4)+(5*2))) = 24
```

← 左から読み進めて、”+” の左側(a)と右側(b*c+d*a)に分離する。同様に、右側部分を”+”の左側(b*c)と右側(d*a)に分離する… というふうに考えるとうまくいくかも。

- mallocの書き方
  - struct node *tmp = malloc(sizeof(struct node));
- あらかじめ用意されているprint_astの実装を参考にすること
- 本来、確保したメモリは解放するべきだが、しなくていい

提出前最終確認

- コンパイルができることを確認すること
- デバッグに使ったprintf文などは、コメントアウトすること
- 実行ファイルでなく、ソースファイル(.c)を添付すること