大島聡史　情報基盤センター助教

C言語速習コース：資料２（後半）「ガウスの消去法」の実装と最適化

後半　実装：ガウスの消去法

- 目標
  - 前半で学んだC言語の基礎知識を元にしてガウスの消去法を実装できるようになる
  - C言語プログラムの最適化（プログラム動作速度の高速化）について理解する

- 内容
  - ガウスの消去法の実装
  - C言語プログラムの最適化
  - ガウスの消去法の最適化実装
  - その他、C言語に関する補足事項など

確認：ガウスの消去法

- $$\begin{cases} a_{00}x_0 + a_{01}x_1 + \cdots + a_{0n}x_n = b_0 \\ a_{10}x_0 + a_{11}x_1 + \cdots + a_{1n}x_n = b_1 \\ \cdots \\ a_{(n-1)0}x_0 + a_{(n-1)1}x_1 + \cdots + a_{(n-1)(n-1)}x_{n-1} = b_{n-1} \end{cases}$$

- n元連立一次方程式をガウスの消去法で解く

いわゆるAx=bのxを求める問題

ガウスの消去法の処理手順

- ガウスの消去法のフローは？
  - （ファイルからの入力）
  - 前進消去ステップ
  - 後退代入ステップ
  - 答えの出力

  - それぞれ関数化してmain関数から順番に実行してやれば良さそう
    - もちろんメイン関数に全部ベタ書きしても動くが、読みやすいように関数として分離させてみよう

ガウスの消去法の実装（1）

- 全体の処理の流れをフローを考えて、プログラムの全体像を書いてみる
  - とりあえず右のプログラムを写す
  - 写しながら、関数の作り方や引数の使い方を再確認する
    - まだ引数や関数の中身は空っぽで良い
  - コンパイルしてエラーが出ないことを確認する
    - gcc -Wall source.c

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

#define N 3

void make_data() {}
void forward_elimination() {}
void backward_substitution() {}
void print_result(){}

int main(int argc, char** argv) {
  double A[N][N], b[N];
  make_data();
  forward_elimination();
  backward_substitution();
  print_result();
  return EXIT_SUCCESS;
}
```

#defineは定数の宣言
コンパイル前に置換される
まずは問題を3x3に限定して考えることにする

ガウスの消去法の実装（2）

- ファイルから行列Aとベクトルbを読み込む関数make_dataを作成する
  - データ形式は
    - 1行目 行列Aの0行目の要素 (スペース区切り)
    - N行目 行列AのN-1行目の要素 (スペース区切り)
    - N+1行目 ベクトルbの要素 (スペース区切り)

例

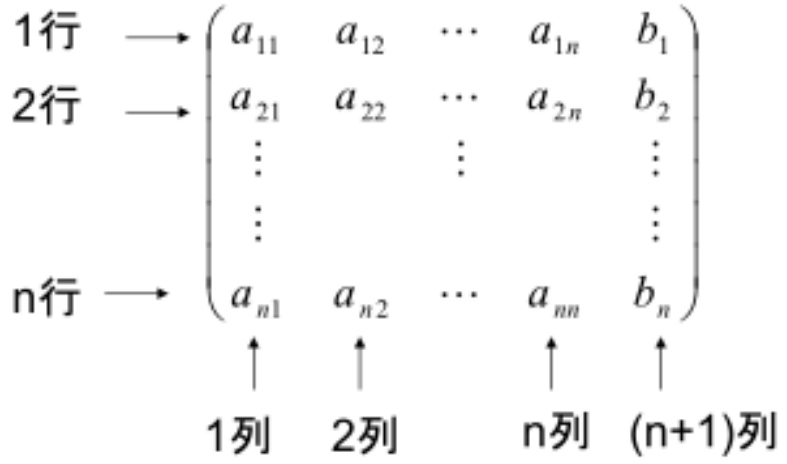
1	-2	1
2	3	1
-1	-4	2
4	1	2

実際には
double型にする

- 行列Aとベクトルbを表示する関数print_resultを作成する
  - 入力ファイルと同じものが画面に表示されるはず
    - 表示位置のズレ（スペースの入れ具合など）は別として
- いずれも、必要な情報は関数の引数として与える
  - サイズ、実際のデータ

前進消去ステップ(1/4)

- forward_elimination関数を作る
- 計算手順を思い出して設計を行う
  - 前進消去ステップ
    - i（最初は1） 行目に注目
      - 各要素をa_{ii}で除算する
      - j=i+1行目に注目
        - » 各要素について、i行目の要素をa_{ji}倍した値を減算する
        - » N行目まで、jをj+1にして繰り返す
      - N行目まで、iをi+1にして繰り返す
    - ※ベクトルについても行列とあわせて計算する必要があることに注意



前進消去ステップ(2/4)

- 反復作業をループとして考える

```
void forward_elimination(……){
  for(i=0; i<N; i++){ // i行目に注目、N行目まで繰り返す
    // i行目の各要素に対する除算
    for(){ // i+1行目に注目、N行目まで繰り返す
      // j行目の各要素に対する減算
    }
  }
}
```

前進消去ステップ(3/4)

- 各行の要素に対する処理もループとして考える必要がある

```
void forward_elimination(……){
  for(i=0; i<N; i++){ // i行目に注目、N行目まで繰り返す
    for(){ } // i行目の各要素に対する除算
    for(){ // i+1行目に注目、N行目まで繰り返す
      for(){ } // j行目の各要素に対する減算
    }
  }
}
```

前進消去ステップ(4/4)

- ループの初期値やベクトルbについても考えてみる

```
void forward_elimination(……){
  for(i=0; i<N; i++){ // i行目に注目、N行目まで繰り返す
    for(j=0; j<N; j++){ A[i][j] = … } // i行目の各要素に対する除算
    b[i] = …
    for(j=i+1; j<N; j++){ // i+1行目に注目、N行目まで繰り返す
      for(k=i; k<N; k++){ A[j][k] = … } // j行目の各要素に対する減算
      b[j] = …;
    }
  }
}
```

ヒント
計算前の値を保存しておかねばならない場合があるのでは？

ガウスの消去法の実装（3）

- 残りの処理を実装する
  - 前進消去を完成させる
  - 後退代入も同様に作ってみる

Tips
前進消去を完成させた時点で実行し、値が正しいかを確認すると、
プログラムのミスを早く見つけられるかもしれない

$$\begin{aligned} 2x_1+3x_2+x_3&=5\\ 2x_1+x_2-2x_3&=1\\ x_1+2x_2+3x_3&=7 \end{aligned} \Rightarrow \begin{pmatrix} 2 & 3 & 1 & 5\\ 2 & 1 & -2 & 1\\ 1 & 2 & 3 & 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3/2 & 1/2 & 5/2\\ 0 & 1 & 3/2 & 2\\ 0 & 0 & 1 & 2 \end{pmatrix} \Rightarrow \begin{aligned} x_3&=2\\ x_2&=2-\frac{3}{2}x_3=-1\\ x_1&=\frac{5}{2}-\frac{3}{2}\times(-1)-\frac{1}{2}\times 2=3 \end{aligned}$$

途中の時点でおかしかったら
最終結果が正しいはずがない！
（仮に正しかったら、さらに別のバグがあるということ）

C言語プログラミングにおけるTips

- 今回の実装を行う上で便利（かもしれない）機能や技術を紹介する
  - デバッグの仕方
  - 出力先の制御
  - 数の表現について（浮動小数点演算の注意点）
  - 算術演算関数の活用
  - 配列の間接参照

デバッグの仕方

- デバッグ：プログラム上の問題（ミス、バグ）を取り除くこと
- デバッグのためのツール（デバッガ）を使う
  - gdb（The GNU Project Debugger)
    - GNUのデバッガ、基本的な使い方であればそれほど難しくはない（が、結構大変）
  - EclipseやVisualStudioなどの統合開発環境には高機能なデバッガが含まれている
    - 一行ずつ変数の値の変化を見ながら実行できるグラフィカルなデバッガなど
  - Valgrind
    - メモリデバッガを中心としたデバッグツール、範囲外参照やmalloc/freeの対応漏れなどを検出できる
- 手軽な対処法：途中の状態を出力してみる
  - 条件式とprintfを組み合わせる
  - 「ここでこの変数はこの値になっているはず」を確認する
  - どこでおかしくなっているかの目安をつける
  - print_result関数のような出力機能を用意しておくと便利

出力先の制御

- よくある問題
  - 実行結果が多い＝画面出力が多いと、結果を遡って確認するのが大変になったり、実行時間（出力時間）が長くなったりする。ファイル出力するように書き換えるのもメンドクサイ。
- 解決方法
  - シェルの機能を使ってファイルに出力すると改善するかもしれない
  - 単純な例：`./a.out > log.txt`
    - 標準出力（画面出力）をlog.txtファイルに書き込む
  - ファイルと画面の両方に出力する例（bash環境の場合）：`./a.out 2>&1 | tee log.txt`

数の表現：浮動小数点演算の注意点

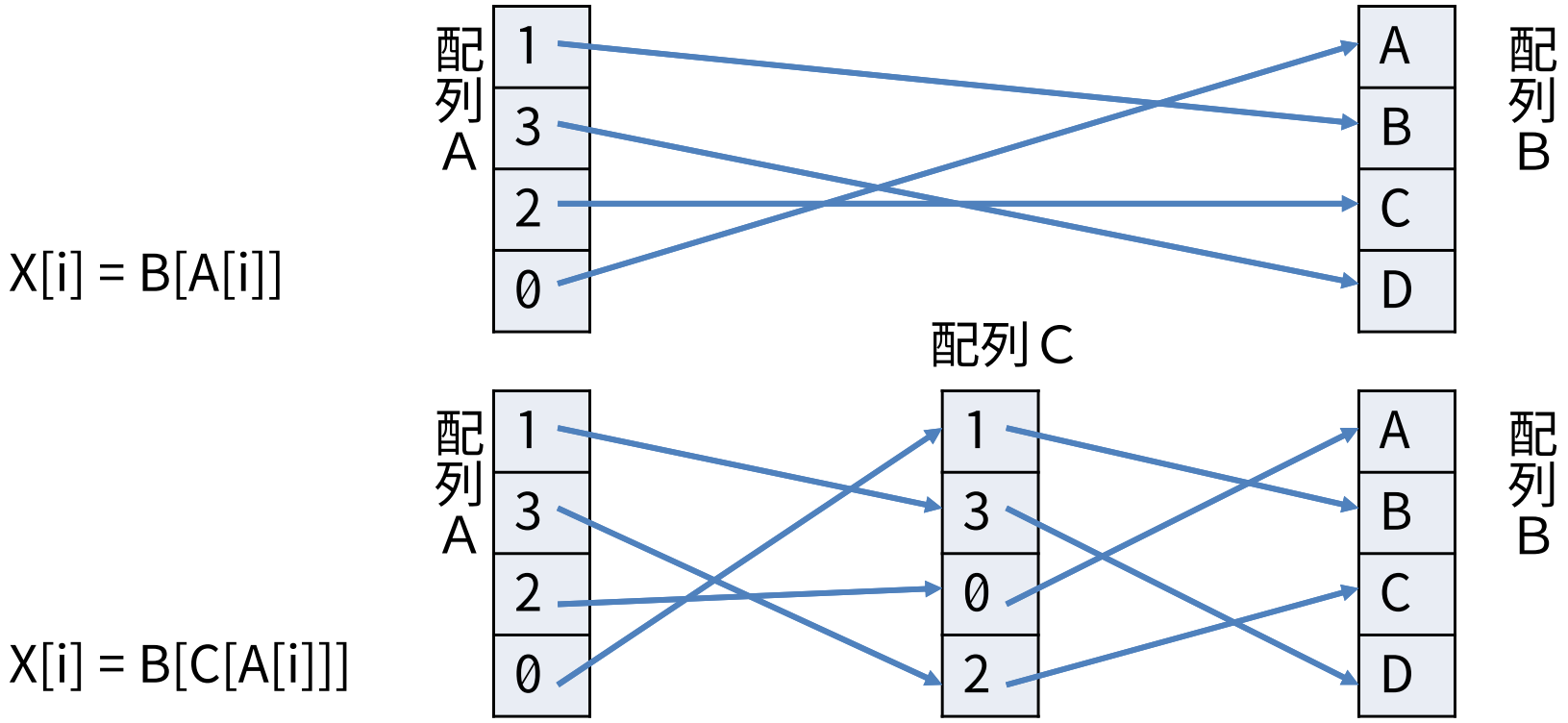
- C言語のfloat型やdouble型は定められたルールに則った方式でデータを保持している
  - $n \times 2^m$形式で保持している（IEEE754形式）
- 扱うデータと計算の内容によってはルールの都合により計算結果がおかしくなることがある
- 手書きでは分数として表現できる数値も実数として持たねばならないため誤差が生じる
  - 無限小数が表現できない
  - 表現可能な値の上限・下限がある
  - 絶対値があまりに大きな値や小さな値は正しく表現・計算できない
  - 絶対値がとても大きく異なる値の加減算を行うと、絶対値の小さい値が無視されてしまう
  - 値がほぼ同じ数値同士の差を取ると誤差が大きくなる
  - etc.
- 出力関数（printf）の都合により差がわからなくなる
  - 出力方法や出力桁数を調整すれば改善することもある

算術演算関数の活用

- C言語そのものにはあまり算術関数が用意されていないが、多くの環境で利用可能なライブラリはある
  - fabs(double f)/fabsf(float f)/fabsl(long double f)
    - 浮動小数変数の絶対値を求める
    - 型によって関数が違う
  - 三角関数、指数・対数関数
  - etc.
- 使うには準備が必要
  - ソースコードの追加：#include <math.h>
  - コンパイル時（正確にはリンク時）：ライブラリのリンク指定
    - `gcc source.c -lm`
    - -lmは数学ライブラリ（libm.a）をリンクする、という意味

配列の間接参照

- 配列を参照する際に別の配列を経由させれば、参照先を変えることができる
  - メモリ使用量が増える、アクセス速度が低下しうる、というデメリットもある



部分ピボット選択の実装

- ピボット選択の手順をプログラムの動作に対応づけて考える

- 新たな行の処理を行う際に、絶対値が一番大きな行を選ぶ処理
  - 大小の比較：if文で判定すれば良い
  - 絶対値の取得：負数の場合は正数に変換する
- 行を入れ替える処理
  - 値を入れ替える：変数の交換、SWAP（前半で扱った）
  - 値を入れ替えない：たくさんの変数を入れ替えるのは処理に時間がかかる→実際には入れ替えずに、入れ替えたような扱いをすると良い

ガウスの消去法の実装（4）

- 部分ピボッティングを行うガウスの消去法を作る

ここで確認と入れかえをすれば良さそう

```
void forward_elimination(……){
  for(i=0; i<n;i++){ // i行目に注目、N行目まで繰り返す
    for(j=0; j<n; j++){ A[i][j] = … } // i行目の各要素に対する除算
    b[i] = …
    for(j=i+1; j<n; j++){ // i+1行目に注目、N行目まで繰り返す
      for(k=i; k<n; k++){ A[j][k] = … } // j行目の各要素に対する減算
      b[j] = …;
    }
  }
}
```

間接参照を使う場合はその後の処理全てに影響するので注意が必要

ガウスの消去法の実装（5）

- 完全ピボッティングを行うガウスの消去法を作る

- 問題サイズを実行時に決められるようにする
  - Nにより大きさが決め打ちされている配列（行列、ベクトル）を変更する
  - malloc/freeを使う
  - main関数内に直接mallocを書くと楽だが、もちろん関数内でmallocするプログラムを書いても良い

高速化のヒント

- 正しい結果を得られるプログラムの書き方は１つではない
- プログラムの作り方次第で性能（実行時間）に大きな差が生じることがある
- 100倍、1000倍以上の差が生じることもあるため、高速化はとても重要
  - １日かかるプログラムが、高速化によって数秒で終わるようになるかもしれない
  - １時間で終わるはずのプログラムが、作り方が悪いと１週間以上かかってしまうかもしれない
- 基本的な高速化手法を幾つか紹介する

CPUが行いやすいような処理にする

- CPUはどんな計算も同じ速度で行えるわけではない（計算式の書き方で性能が変わる）
  - 一般的に、除算はとても重い
  - 加算・減算・乗算はあまり変わらない
  - 論理演算が高速なことがある
- 同じ計算を何度も行う場合はまとめる
  - 一時的な変数に格納しておいて、変数を参照するだけにする
- 分岐処理を減らす
  - 例：多重ループの最内部でifを使わない

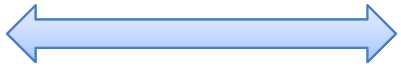
配列アクセスを高速化する

- 同じ回数の配列アクセスでも所要時間に差が生じる
  - 時間的・空間的に近いアクセスは高速に行えることがある
  - 連続した範囲は連続に、同じ要素へのアクセスは長い時間を空けずに行うと、そうでない場合より高速
  - 連続でない（ランダムな、飛び飛びな）アクセスは低速
  - キャッシュのおかげ：メモリの内容を一時的に保存しておき、次回同じメモリにアクセスする際に省略してくれる機構
- 多次元配列のアクセス順序を再確認する
  - ループを入れ替えても結果が変わらない計算でも、メモリアクセス順序が変わって性能が大きく変わることがある
- 同じ配列要素に何度も書き込む場合には、一旦ローカル変数上で計算を行って、最後に配列に書き込んだ方が良いことがある
  - 例：多重ループの最内部における足し込み処理
- 大きな行列積を実装してみるとわかりやすい

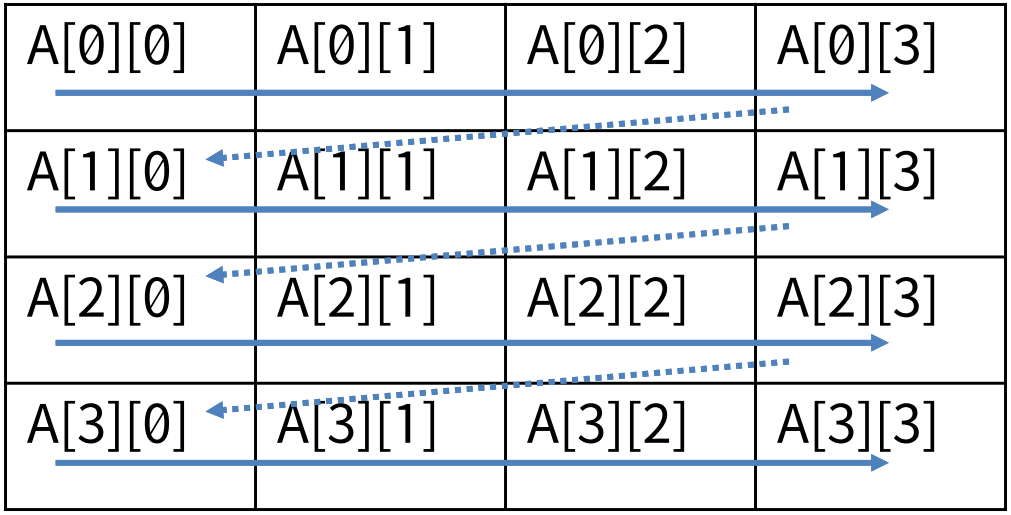
二次元配列への連続アクセスの例

- 配列に順番にアクセスできているかを考える

```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    A[i][j] = ...
  }
}
```




```
for(j=0; j<n; j++){
  for(i=0; i<n; i++){
    A[i][j] = ...
  }
}
```



- これまでに学んできたプログラムの組み方（配列・メモリの確保方法）では、二次元配列はメモリ上で左図矢印の方向に並んで（連続して）いる
- この順序に従うメモリアクセスとそうでないメモリアクセスの速度は大きく異なる

コンパイラに高速化してもらう

- 世の中の多くのコンパイラにはプログラムを最適化する機能が備わっている
- 最適化オプションを使う
  - 例えばgccでは-O1 -O2 -O3（数字が大きい方が強力な最適化を行う）など
  - コンパイラによって異なるため確認が必要
  - 指定しなくてもある程度良いオプションが選ばれていることが多い
- これまでに挙げた高速化技術を適用しなくても、コンパイラによる最適化だけで十分に良い性能が得られることもある
  - 最適化の有無とコンパイラオプションの組み合わせで性能が変わる
- 常にうまく行くわけではない
  - 簡単な（コンパイラにとってわかりやすい）プログラムはうまく行きやすい
  - 手動での最適化も重要

ガウスの消去法の実装（6）

- ガウスの消去法の高速化を行う
  - 計算を置き換えられるところは無いか？
  - 計算順序・配列アクセス順序を変更できないか？
  - コンパイルオプションを変えて性能差を見る
  - 問題サイズが小さいと差がわかりにくいため、大きな問題で比較すると良い
  - 実行時間の測定には time コマンドが便利
    - timeコマンドに続けて測定したいコマンド列を与える
    - time ./a.out data.dat

参考：ライブラリの利用

- それぞれの分野における「よく使うお決まりの処理（関数）」というものが存在する　（例：行列積など）
- プログラミングの得意な人が作った（最適化された）処理を使い回せれば便利
- ソースコードを共有する？
  - プログラムの中身が全部見られてしまう（→オープンソース）
  - コンパイルが必要：大規模問題では大変、オプションの差が性能差になる
- ライブラリを使う
  - 関数などを切り出して別のプログラムからも簡単に使えるようにしたもの
- ライブラリを作る方法
  - コンパイラにオプションを指定、専用プログラムでまとめる
- ライブラリを使う方法
  - 宣言を行う：#include <xxxx.h>
    - 指定したファイルの中に宣言が書かれている
  - コンパイラに使うライブラリを教える
    - gcc source.c -lxxxx という書式で指定する

その他、C言語に関する補足事項など

- C言語を使うにあたって知っておくと良いこと（ガウスの消去法を実装するうえでは気にしなくても良かったことなど）
  - 複数ソースコードにまたがるプログラム
  - 変数のスコープに関する補足事項
  - 再帰関数
  - 非常に大きな配列を扱う際の注意点
  - 構造体
  - 古い記法

複数ソースコードにまたがるプログラム

- プログラムは複数のソースコードに分散していても良い
  - むしろメンテナンス性や部分コンパイルの利便性などを考えると分散していた方が良い
- 例：main関数から呼ばれる関数を別のファイルに移動させてみる

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv){
  int x = 1;
  x = func(x);
  printf( “x = %d￥n” , x);
  return EXIT_SUCCESS;
}
```

func.c

```
#include <stdio.h>
#include <stdlib.h>
int func(int n){
  return n + 1;
}
```

どのようにコンパイルすれば良いのだろうか？

実は gcc main.c func.c でも良い。ただし-Wallをつけると
implicit declaration of function ‘func’
という警告が出る。ただし、実行すれば動く。
（意訳：funcという関数なんて知りません）

警告を消すためにはmain.c側にfunc.c側に存在する関数の情報を入れてやれば良い。
具体的には、main関数より前に
int func(int);
とプロトタイプ宣言を書けば良い。
＝同一ソースコード内の呼び出し位置より下に関数を書く時と同じである。

main.c
```
#include <stdio.h>
#include <stdlib.h>
int func(int);
int main(int argc, char **argv){
  int x = 1;
  x = func(x);
  printf( “x = %d￥n” , x);
  return EXIT_SUCCESS;
}
```

より一般的なコンパイルの方法：オブジェクトファイルを利用する。
-cオプションを付けると中間ファイルが生成される。
gcc -Wall -c main.c　　main.cからmain.oが作られる
gcc -Wall -c func.c　　func.cからfunc.oが作られる
gcc -Wall main.o func.o .oファイルをリンクして実行可能ファイルが作られる。
main.cかfunc.cのどちらか一方のみファイルを更新した場合は、更新したファイルだけ再コンパイルしてリンクすれば良い。

さらに一般的な使い方：プロトタイプ宣言だけのファイル（ヘッダファイル、拡張子は.h）を用意し、呼び出したい元のファイルで#includeする。

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include  “func.h”
int main(int argc, char **argv){
  int x = 1;
  x = func(x);
  printf( “x = %d￥n” , x);
  return EXIT_SUCCESS;
}
```

func.c

```
#include <stdio.h>
#include <stdlib.h>
int func(int n){
  return n + 1;
}
```

func.h

```
int func(int);
```

func.cに実装されている関数が増えても、ヘッダファイルにプロトタイプを追加すれば呼び出し側（main.c）への変更は不要。

一つのグローバル変数を共有したいときは、一箇所以外ではextern節を付ける。（外部のどこかに存在する、ということをコンパイラに教える。）

main.c

```
int global_value;
```

func.c

```
extern int global_value;
```

変数のスコープに関する補足事項

- グローバル(global)変数（大域変数）
  - 関数外にて宣言した変数、プログラム全体で共通
- ローカル(local)変数（局所変数）
  - 関数内やブロック内にて宣言した変数
  - 対象となる関数を再び実行するときには初期状態に戻っている
  - mallocしたものをfreeせずに関数を終了すると残留する、繰り返すとメモリが枯渇する→メモリリーク、プログラムが異常終了する原因となる
- スタティック(static)変数
  - 変数名にstatic接頭辞を付加する　`static int n;`
  - 対象となる関数を再び実行するときには前回の結果が残っている
    - 関数が実行された回数を数えてみよう
  - グローバル変数をstatic化すると、そのファイル内でのみ使える変数になる（extern接頭辞を付けても使えない）

再帰関数

- 関数funcから関数func、つまり自分自身を実行することができる
  - 再帰的な関数実行、再帰関数
- 無限に実行し続けることにならないように、終了条件の設定が必要
- 例：関数func(int n)は、nが1の時は1、それ以外の時はn + func(n-1)を返すとする。func(10)はいくつになるだろうか？

```
int func(int n){
  if(n==1)return 1;
  return n + func(n-1);
}
printf( “%d￥n” , func(10));
```

非常に大きな配列を扱う際の注意点

- あまりに大きなローカル変数は扱えない
  - 例：main関数にて double d[2000][2000];を宣言したところ、プログラム開始直後にエラー終了（セグメンテーション違反）
  - 各関数内で利用可能な資源量（変数・配列の数）には制限があり、この値を溢れるとプログラムが終了してしまう
    - この例では「スタックサイズ」という上限を突破している
    - 環境によって境界は異なるのでいくつか試してみると良い
    - ulimit -s コマンドで制限を緩和できる（できないこともある）
  - 巨大な配列（行列）を扱いたいときはどうすれば良いか？
    - グローバル変数を使うか、動的配列（mallocして使う）を使うと良い（制限が緩くなる）
    - static変数にするという手もある
    - 変数のスコープの違いによる挙動の違いやfreeが必要なことに注意

構造体

- 複数の変数をひとまとめにしたデータ型を作ることができる
- 理解しやすいプログラム作成の助けになる

```
struct product{
  int id;
  int price;
  double margin;
  char producer[0xff];
};

struct product item[N];
item[0].id = 0;
item[0].price = 100;
item[0].margin= 0.3;
strcpy(item[0].producer, “ABC farm” );

for(i=0;i<N;i++){
  printf( “item %d: %d %f %s￥n” ,
    item[i].id, item[i].price, item[i].margin, item[i].producer);
}
```

型の別名を作るtypedefを使った例

```
typedef struct{
  int id;
  int price;
  double margin;
  char producer[0xff];
}product;

product item[N];
item[0].id = 0;
item[0].price = 100;
item[0].margin= 0.3;
strcpy(item[0].producer, “ABC farm” );
```

古い記法

- 古めの教科書に書かれたC言語のプログラムを読んでいると古い書き方のプログラムに出くわすことがある
- 読み方自体は難しくない
- 古い記法の例

```
int func(a, b, n)
double a[][N], b[];
int n;
{
    int i,j, …
```

  - 仮引数に型情報が無い
  - 関数本体の前に変数名と型情報が並んでいる

速習コースに関する問い合わせは担当教員
大島（ohshima at cc.u-tokyo.ac.jp）
までメールでお願いします

（gmail等からの送信でも特に問題ありませんが、
送信者の所属と氏名は明記してください）