プログラミング及び演習
第12回 大規模プログラミング
(2017/07/15)

講義担当
大学院情報学研究科知能システム学専攻
教授　森　健策
大学院情報学研究科知能システム科学専攻
助教　小田　昌宏

本日の講義・演習の内容

- 大きなプログラムを作る
  - 教科書 第12章
    - makeの解説
  - プログラミングプロジェクト
    - どんどんと進めてください
- 講義・演習ホームページ
  - http://www.newves.org/~mori/17Programming
- ところで,
  - プログラミング及び演習もいよいよ終盤です。

プログラムを分割しよう!

- プログラムを作成する上での重要点
  - プログラムの中で互いに関連の深い部分をひとまとめにし、そのまとまりを明確にする
  - プログラムを作成したり、修正・改良したりする際に一括して考慮しなければならない範囲を限定する
- 「明確な思考の単位を作る」
  - 「関数」に分割
  - 例えば入力処理がある関数の中に完全に隔離されていれば、後々入力処理に関して何らかの修正や改良が必要になったとしても、見直さなければならない範囲が限定される

ソースファイルを分割する

- プログラム分割の単位
  - 関数
  - ファイル
- 変数･関数の有効範囲
  - static指定がある場合
    - ファイルの中 (コンパイル単位内) のみで有効
  - static指定がない場合
    - extern宣言をつけることで他のコンパイル単位の変数をアクセス可能
    - 関数の場合にはexternは必要なく他のコンパイル単位の関数をアクセス可能

例　教科書 p. 142

```
/* file 1 */
static int x;
int y;
void f(void)
{
  x=y=0;
}
void main(void)
{
  f();
  g();
  printf("x=%d y=%d¥n",x,y);
}
```

```
/*file 2*/
int x;
extern int y;
void g(void)
{
  x=y=1;
}
```

分割して隠蔽せよ

- プログラムファイルを分割する際の原則
  - 「プログラムに何らかの変更を行う際に、同時に修正しなければならない（可能性の高い）関数を一つのファイルに集める」
- 情報隠蔽
  - 変数・関数の有効範囲は必要最小限度にとどめる
  - 必要な物のみ外部からアクセスできるようにする
- データ構造の抽象化
  - データが内部で具体的にどのように保存されているのかわからないようにする
  - データへのアクセスは関数を通してのみ

情報隠蔽とデータ構造の抽象化

```
/*file 2*/
void main(void)
{
  writeTable(1,20);
  ...
  readTable(1);

}
```

main.c

tableへのアクセスは関数を通じてのみ
tableがどのように実現されているかは外部からはわからない
(配列以外の実装としても呼び出し(この場合はmain関数)側からはわからない)

```
#define TABLESIZE 50
static int table[TABLESIZE];
void writeTable(int i, int value ):
{
  if(i>=0&&i<TABLESIZE)
    table[i] = value;
  else
    exit(1);
}
void readTable(int i)
{
  if(i>=0&&i<TABLESIZE)
    return table[i];
  else
    exit(1);
}
```

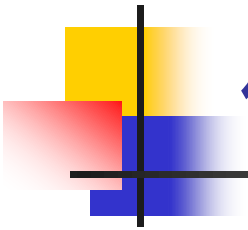
table.c

オブジェクト指向プログラミング言語がもつ3つの特色

- カプセル化
  - プログラムコードとプログラムコードが扱うデータを一体化して外部の干渉や誤用から両者を保護する仕組み
  - 非公開と公開
- ポリモーフィズム
  - 1つの名前を2つまたはそれ以上の関連する目的に使用できるようにする性質
- 継承
  - 1つのオブジェクトが他のオブジェクトの性質を獲得するプロセス

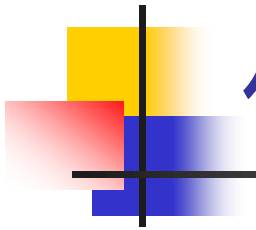
ヘッダファイル

- ヘッダファイル
  - 原型宣言、型宣言、マクロ定義などを複数のソースファイルで共有するために利用される
- 例 - addtree.h
  - #ifndef _ADDTREE_H_
    #define _ADDTREE_H_
    #include "tnode.h"
    struct tnode *addtree(struct tnode *p, char *w);
    #endif
  - addtreeを使うソースファイルではaddtree.hをインクルードする



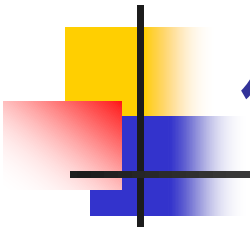
例　lec10-wordsearch.cを分割

- プログラムのメイン
  - main.c
- ノードの定義
  - tnode.h
- ノードのメモリ確保
  - talloc.c
  - talloc.h
- 木構造へのノード追加
  - addtree.c
  - addtree.h
- 木構造の印字
  - treeprint.c
  - treeprint.h
- 入力から1単語取り出す
  - getword.c
  - getword.h



例 addtree.c

```
#include <ctype.h>
#include <string.h>
#include "addtree.h"
#include "talloc.h"
struct tnode*
addtree(struct tnode *p, char *w)
{
  int cond;
  if(p==NULL){
    p = talloc();
    p->word = strdup(w);
    p->count = 1;
    p->left = p->right = NULL;
  }else if ((cond=strcmp(w,p->word))==0){
    p->count++;
  }else if(cond<0){
    p->left = addtree(p->left,w);
  }else{
    p->right = addtree(p->right,w);
  }
  return p;
```



例 addtree.h

```
#ifndef _ADDTREE_H_
#define _ADDTREE_H_

#include "tnode.h"

struct tnode *addtree(struct tnode *p, char *w);

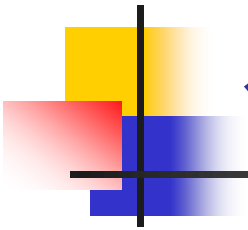
#endif
```

例　main.c

```
#include <ctype.h>
#include <stdio.h>
#include "tnode.h"
#include "addtree.h"
#include "treeprint.h"
#define MAXWORD 100
int main(int argc, char **argv)
{
  struct tnode *root;
  char word[MAXWORD];
  root = NULL;
  while(getword(word,MAXWORD)!=EOF){
    if(isalpha(word[0])){
      root = addtree(root,word);
    }
  }
  treeprint(root);
  return(0);
}
```

addtree.h, treeprint.hの中でもtnode.hをinclude

tnode.hの重複includeを防止する必要有

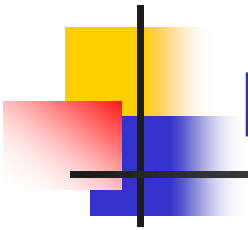


分割コンパイル

- 複数のソースファイルからなるプログラムをコンパイルする場合
  - cc -o prog file1.c file2.c file3.c
- 中間ファイルを生成後結合編集する場合
  - cc -c file1.c
    cc -c file2.c
    cc -c file3.c
    cc -o prog file1.o file2.o file3.o
  - この場合file1を修正したとすれば、1行目と4行目のみを実行すればよい

大規模プログラム開発によくある状況

- 「あなたが開発しているプログラムは複数のプログラムから構成されています．つまりそれらをコンパイルし，リンクすることで1つの実行可能なプログラムが作成されます．複数のプログラムのどれか1つを修正したときは，その修正ファイルのみをコンパイルし直し，新しい実行プログラムを作成します．」
  - C.トンド, A. ネイサンソン, E.ヤント著，"Makeの達人", Pearson Education Japan より



makeを使おう

- make
  - ファイルの更新時間をチェックし，更新されていれば自動コンパイル
  - Makefileと呼ばれるファイルに定義した指示に従って作業を実行
- Makefile の記述例
  - prog: file1.o file2.o
    cc file1.o file2.o -o prog
    file1.o: file1.c file1.h
    cc -c file1.c
    file2.o: file2.c file2.h
    cc -c file2.c
- コンパイル方法
  - make progを生成するための一連の手順を実行
  - make file1.o file1.oを実行するための一連の手順を実行

Makefileを記述する上での注意点

- ターゲット/依存関係行は1桁目から書きはじめる
- コマンド行は字下げ（インデント）する
- 行の内容が複数行に及ぶ場合は，連続した行であることを示すため，行末にバックスラッシュ記号をつける
- コメント行は#で始める

ターゲットと依存関係

- Makefile の記述例
  - prog: file1.o file2.o
    cc file1.o file2.o -o prog
    file1.o: file1.c file1.h
    cc -c file1.c
    file2.o: file2.c file2.h
    cc -c file2.c
- ターゲット
  - prog
  - file1.o
  - file2.o
- 依存関係
  - progはfile1.o file2.oに依存
  - file1.oはfile1.c file1.hに依存
  - file2.oはfile2.c file2.hに依存

マクロを利用したMakefile

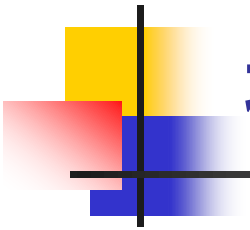
- Makefileにおける変数
- マクロには名前をつけて値を割り当て
- makeがMakefileを実行時マクロを割り当てられた値に展開
- 例
  - ```
    CC = gcc
    CFLAGS = -c -W2 -O
    prog: prog1.o prog2.o
            $(CC) -o prog prog1.o prog2.o
    prog1.o: prog1.c prog1.h
            $(CC) $(CFLAGS) prog1.c
    prog2.o: prog2.c prog2.h
            $(CC) $(CFLAGS) prog2.c
    ```
    ```

依存関係記述にもマクロを利用可

```
CC          = gcc
CFLAGS      = -c -W2 -O
OBJS        = prog1.o prog2.o
TARGET      = prog
$(TARGET): $(OBJS)
        $(CC) -o $(TARGET) $(OBJS)
prog1.o: prog1.c prog1.h
        $(CC) $(CFLAGS) prog1.c
prog2.o: prog2.c prog2.h
        $(CC) $(CFLAGS) prog2.c
```

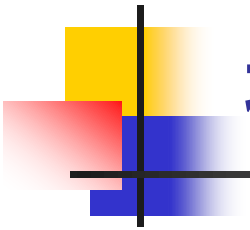
内部マクロ (よく使うもののみ)

- CC
  - Cコンパイラのプログラム名が定義 (cc)
- $@
  - カレントターゲット
  - prog: prog1.o prog2.o
    $(CC) -o $@ prog1.o prog2.o
- $*
  - 拡張子を除くターゲット名
  - prog1.o: prog1.c prog1.h
    $(CC) $(CFLAGS) $*.c



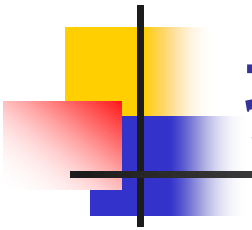
推論規則

- 特定の拡張子を持った依存ファイルから特定の拡張を持つターゲットを生成できるような生成規則
- ファイル名の拡張子に対して作用するためmakeファイル中に記述するコマンドの簡略化に役立つ
- サフィックスルールとも呼ばれる



推論規則

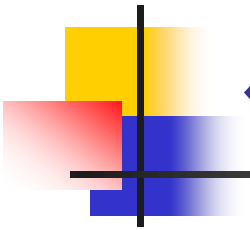
- ピリオド，拡張子，もう一つのピリオド，もう一つの拡張子の順に記述する
- 例1
  - .c.o:
    $(CC) $(CFLAGS) $*.c
  - .cファイルをソースファイルとし.oファイルを生成する
- 例2
  - .c.o:
    $(CC) $(CFLAGS) $<
  - 例1と同様
  - $<は推論規則における依存ファイルを表す



推論規則を用いたMakefile

```
CFLAGS      = -c -W2 -O
OBJS = addtree.o getword.o main.o talloc.o treeprint.o
TARGET      = wordsearch
RM          = rm

$(TARGET): $(OBJS)
        $(CC) -o $@ $(OBJS)
clean: $(OBJS)
        $(RM) -f $(OBJS)
.c.o:
        $(CC) -c $<
```



付録　cursesで色付け

```
int main(int argc, char **argv){
  char text[128];
  initscr();
  start_color();
  noecho();
  cbreak();
  curs_set(0);
  init_pair(1, COLOR_CYAN, COLOR_BLACK);
  init_pair(2, COLOR_RED, COLOR_BLACK);
  init_pair(3, COLOR_YELLOW, COLOR_BLACK);
  init_pair(4, COLOR_WHITE, COLOR_BLACK);
  bkgd(COLOR_PAIR(4));
  keypad(stdscr,TRUE);
  go();
  endwin();
  return 0;
}
```

付録　cursesで色付け

```
void go()
{
  …
  char buffer[256][256];
  char color_buffer[256][256];
  …
  for(j=0;j<256;j++){
    for(i=0;i<256;i++){
      buffer[j][i]=' ';
      color_buffer[j][i] = 0;
    }
  }
 while((ch=getch())!='Q'){
    attron(COLOR_PAIR(4));
    mvaddstr(blocY,blocX,"    ");
    mvaddstr(blocY+1,blocX,"    ");
    mvaddstr(blocY+2,blocX,"    ");
    …
    for(j=0; j<LINES; j++){
      for(i=0; i<COLS/2; i++){
        attron(COLOR_PAIR(color_buffer[j][i]));
        mvaddch(j, i, buffer[j][i] );
      }
    }
    attron(COLOR_PAIR(pattern+1));
     for(jj=0; jj<3; jj++){
       for(ii=0; ii<3; ii++){
         mvaddch(blocY+jj, blocX+ii, blockPattern[pattern][jj][ii]);
       }
     }
     if( collisionBottomWall(blocX, blocY, LINES-3, pattern)){
       for(jj=0; jj<3; jj++){
         for(ii=0; ii<3; ii++){
           buffer[blocY+jj][blocX+ii] = blockPattern[pattern][jj][ii];
           color_buffer[blocY+jj][blocX+ii] = pattern+1;
         }
       }
       beep();
       blocY = 5;
       pattern = (pattern+1)%PATNUM;
     }
     sprintf(msg, "X %03d Y %03d Pat %02d", blocX, blocY, pattern );
     attron(COLOR_PAIR(4));
     mvaddstr(2,COLS-20,msg );
   }
 }
```

さらに工夫するとこんな画面も