

# ソフトウェア設計及び実験

## 第2回: プログラミング作法

担当: 松本 和幸

# 本日の内容

## 基本的な作法

## ソフトウェア作成のためのツール紹介

## 基本テクニック

- 作成時
  - 命名規則
  - 関数化
  - #define (マクロ)
- デバッグ時
  - assert
  - #ifdef
  - gdb (デバッガ)

# プログラミング作法とは

## ソースコードを書く際の決め事 (プログラミングスタイルともよぶ)

- 開発者の好みや会社の標準
- 決まりを守らなくてもプログラムは動く(動いてしまう)
  - デバッグしにくくなる
- コードの保守には欠かせない
  - 他人 or 自分が(も)読みやすいコード

# 本日の内容

## 基本的な作法

### ソフトウェア作成のためのツール紹介

### 基本テクニック

- 作成時
  - 命名規則
  - 関数化
  - #define (マクロ)
- デバッグ時
  - assert
  - #ifdef
  - gdb (デバッガ)

# 見たい目

## 汚い(見にくい)コード:

```
int i = 0;
for(i=0; i<=100; i++){ printf("%d¥n", i ); if ( i==2 ){ break; } }
for(i=1; i<=101; i++){ printf("%d¥n", i ); if (i==3){ break; } }
```

## 綺麗な(見やすい)コード:

```
int i ,m = 2;
for ( i = 0; i <= m; i++ ){
    printf("%d¥n", i );
    if ( m == 2 && i == 2 ){
        i = 0;
        m = 3;
    }
}
```

変なところで  
インデントしている

変数と演算子の間に  
スペースが無い

## 汚い(見にくい)コード:

```
int i = 0;  
for(i=0; i<=100; i++){ printf("%d¥n", i ); if ( i==2 ){ break; } }  
for(i=1; i<=101; i++){ printf("%d¥n", i ); if (i==3){ break; } }
```

forブロックを1行で  
書いてしまっているため  
流れが理解しづらい

## 綺麗な(見やすい)コード:

```
int i ,m = 2;  
for ( i = 0; i <= m; i++ ){  
    printf("%d¥n", i );  
    if ( m == 2 && i == 2 ){  
        i = 0;  
        m = 3;  
    }  
}
```

# 見たい

## 汚い(見にくい)コード:

```
int i = 0;
for(i=0; i<=100; i++){ printf("%d¥n", i ); if ( i==2 ){ break; } }
for(i=1; i<=101; i++){ printf("%d¥n", i ); if (i==3){ break; } }
```

## 綺麗な(見やすい)コード:

```
int i ,m = 2;
for ( i = 0; i <= m; i++ ){
    printf("%d¥n", i );
    if ( m == 2 && i == 2 ){
        i = 0;
        m = 3;
    }
}
```

定数ではなく  
変数を利用

1つのfor文

# 見たい目

汚い(見にくい)コード:

```
int i = 0;
for(i=0; i<=100; i++){ printf("%d¥n", i ); if ( i==2 ){ break; } }
for(i=1; i<=100; i++){ printf("%d¥n", i ); if ( i==2 ){ break; } }
```

綺麗な

```
int i, j;
for ( i
```

明確な決め事はないが、  
人(自分も含めて)が読みやすい  
コーディングを心がける！

```
    i = 0;
    m = 3;
```

```
}
```

```
}
```



# インデント

## 字下げのこと

## 適切に行うとコードを見やすくできる

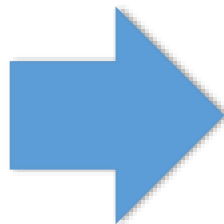
- 構文が複雑な場合(多重ループ, 入れ子構造)

## 字下げを構文に取り入れている言語もある

- Pythonなど

インデントなし

```
if ( i == 1 ){  
if ( k == 2 ){  
printf(“%d¥n”, i );  
}  
}
```



インデントあり

```
if ( i == 1 ){  
    if ( k == 2 ){  
        printf(“%d¥n”, i );  
    }  
}
```

# 空白の扱い

空白(スペース)を入れることで見やすくする

- 変数(定数)と演算子の間

`a+b=10;` ➡ `a + b = 10;`

- かっこと変数(定数)の間

`(a + b) = 10;` ➡ `( a + b ) = 10;`

- 桁の位置合わせ

`c = 1111 + 200;`  
`x = 111 + 200;` ➡ `c = 1111 + 200;`  
`x = 111 + 200;`


# ループ制御

## forを使うか、whileを使うか

```
for ( i = 0; i < 10; i++ ){  
    if ( i == 2 ){  
        printf(“%d¥n”, i );  
    }  
}
```

```
while ( i < 10 ){  
    if ( i == 2 ){  
        printf(“%d¥n”, i );  
    }  
    i++;  
}
```

見落とし  
注意



# 移植性の高いコード(標準化)

## 例) ライブラリ関数を使ったオプション解析

getopt 不使用

```
while ( argc > 1 && argv[1][0] == '-' ){  
    switch ( argv[1][1] ){  
        case 'a':  
            flg = 1;  
            break;  
        case 'b':  
            flg = 2;  
            break;  
        default:  
            usage(); /* 使用方法表示 */  
    }  
}
```

人によって書き方が  
異なってくる

getoptを使用 (unistd.hをインクルード)

```
while ( (ch = getopt(argc, argv, "ab")) != -1 ){  
    switch ( ch ){  
        case 'a':  
            flg = 1;  
            break;  
        case 'b':  
            flg = 2;  
            break;  
        default:  
            usage(); /* 使用方法表示 */  
    }  
    argc -= optind;  
    argv += optind;  
}
```

誰が書いても  
大体同じ書き方

# 本日の内容

## 基本的な作法

## ソフトウェア作成のためのツール紹介

## 基本テクニック

- 作成時
  - 命名規則
  - 関数化
  - #define (マクロ)
- デバッグ時
  - assert
  - #ifdef
  - gdb (デバッガ)

# エディタ

## ソースコードを編集するツール

- Emacs
- gedit
- vi(vim)  
など

Emacsやviは、端末内で実行可能なので便利

- Eclipse（統合開発環境）・・・後期で紹介あり
- Xcode・・・MacOS X用
- Sublime Text・・・Linux, MacOS, Windows用
- Atom・・・Linux, MacOS, Windows用
- meadow・・・Windows用
- Visual Studio Code・・・Windows, MacOS, Linux用

# コンパイラ

## gcc (GNU Compiler Collection)

- C言語以外にC++, Objective-Cなどもコンパイル可能

## コンパイラオプション

- `-o filename`: 実行ファイル名の指定
- `-O2`: 最適化
- `-g`: デバッガ利用(gdb)
- `-lm`: 数学ライブラリ利用
- `-lX11`: X Windowライブラリ利用
- `-Ipath`: インクルードファイルパス指定
- `-Lpath`: ライブラリファイルパス指定
- `-Dmacro-name`: マクロ変数の定義

# 本日の内容

## 基本的な作法

## ソフトウェア作成のためのツール紹介

## 基本テクニック

- 作成時
  - 命名規則
  - 関数化
  - #define (マクロ)
- デバッグ時
  - assert
  - #ifdef
  - gdb (デバッガ)



# 命名規則（色々なルールがある）

## (1) 変数名には意味を持たせる

$A = B * C + D;$

AはB, C を掛け合わせたものにDを足したもの

⇒ 変数がただの記号なので作者にしか何のための計算かわからない

$wage = num\_attendance * day\_rate + bonus;$

賃金(wage)は、出勤日数(num\_attendance)に日給(day\_rate)をかけたものにボーナス(bonus)を足したもの

⇒ 変数名を見れば何の計算式かを推測可能

# 命名規則

## (2) 関数名には意味を持たせる

```
int func1( int a, int b, int c ){  
    return ( a + b + c );  
}
```

**func1** という関数名のみだと、内部でどのような計算が行われるかわからない  
→ 使用する際、いちいち関数の中身を見る必要があり、非効率的

```
int sum( int a, int b, int c ){  
    return ( a + b + c );  
}
```

**sum(summation: 総和)** という関数名から、  
足し合わせる関数であることが分かる

# 命名規則

## 適切な省略を用いる or フルネームを用いる

悪い例:

✓ **SOI**: アイテムのスコア (score of item )

何の略かが分かりづらい...

✓ **school of Internet** (インターネット上の学校)

✓ **sit on it** (少し静かにしなさい)

→ **Item\_score** のほうが意味が分かりやすい

一般的に、どのような省略形が用いられるのかを知る

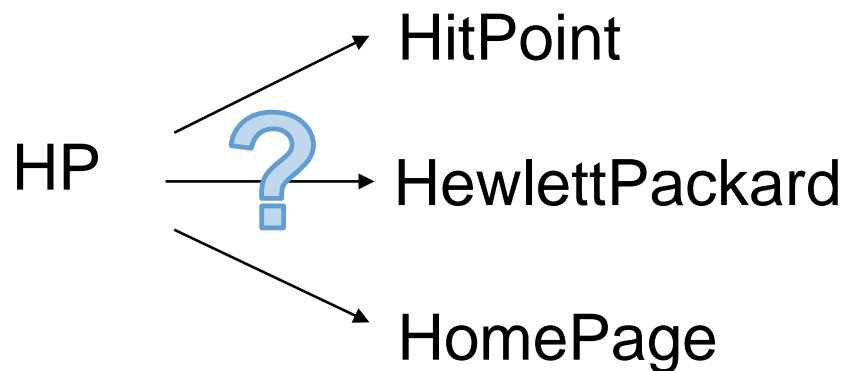
- Image → img
- library → lib

など

- 達人プログラマのソースコードを読んだり自分のコードを他人に読んでもらい評価してもらうなどして身に付ける

# グローバルではわかりやすい名前

- 分担作業でのプログラム開発
  - グローバル変数
  - 意味を持った名前にする
  - 変数名が混同しないように工夫  
(似た変数名や異なる命名規則だとわかりづらい)



# ローカルでは簡潔(短い)名前

- (当面の間)自分だけが読む可能性がある変数
  - 簡潔な命名にすることで編集を容易にする
  - 簡潔でかつ意味が推測できるのがよい

ゲーム開発などでよく使用する変数

hp ... ヒットポイント

life ... ライフ

avg ... 何かの平均値

prob ... 何かの確率

# 命名規則

## ファイル名

### 一般的な構成

- README: プロジェクト概要
- INSTALL: インストール手順
- configure: プラットフォーム設定スクリプト
- Makefile: ビルド

など...

## ファイル拡張子

- .c: Cのソース
- .h: C/C++のヘッダファイル
- .cc, .cpp, .cxx: C++のソース
- .java: Javaソース

など...

# 関数化

main関数にすべての処理を書かない

役割ごとに関数に切り分ける

```
int sum( int a, int b, int c ){  
    return ( a + b + c );  
}
```

```
int mul( int a, int b, int c ){  
    return ( a * b * c );  
}
```

```
int main( int argc, char** argv ){  
    int a, b, c, x, y;  
    a = 1;  
    b = 2;  
    c = 3;  
    x = sum( a, b, c );  
    y = mul( a, b, c );  
    printf(“x = %d, y = %d\n”, x, y );  
    return (0);  
}
```

関数化しないと、全体の見通しが悪くなり、  
プログラムの修正が難しくなる

# define

#defineマクロ を用いて定数を定義

メリット： 後で一括変更できる

```
#include <stdio.h>

#define MIN_SIZE 1
#define MAX_SIZE 50

int main( int argc, char** argv ){
    int m = 0;
    for ( m = MIN_SIZE; m <= MAX_SIZE; m++ ){
        printf( "%d¥n", m );
    }
    for ( m = MAX_SIZE; m >= MIN_SIZE; m-- ){
        printf( "%d¥n", m );
    }
    return (0);
}
```



# #define使用上の注意

```
#define ONE 1  
#define TWO 2  
#define THREE 3
```

このようなマクロ名を使用すると、混乱のもととなる  
⇒ 設定している値の数値が変更になると  
マクロ名まで変更する必要が出てくる



プログラム内における役割を示すマクロ名にする



命名規則

# assert

## エラーが起きそうな箇所に仕掛ける

変数の値のチェックなど

**assert( expression )**

expressionが偽の場合に**エラーメッセージ**を表示して強制終了

```
/* atest.c */
#include <stdio.h>
/* assertを使うため */
#include <assert.h>
int main( int argc, char** argv )
{
    assert( argc > 1 );
    int i;
    for( i = 1; i < argc; i++ ){
        printf(“%s\n”, argv[ i ] );
    }
}
```

## 実行結果:

```
% gcc -o atest atest.c
% ./atest
```

```
atest: atest.c:6: main:
Assertion `argc >1' failed.
アボートしました
```

# assert

マクロ NDEBUG を定義すると、assert() は呼び出されない

→ assert の除去

```
/* atest.c */
#include <stdio.h>
#define NDEBUG
/* assertを使うため */
#include <assert.h>
int main( int argc, char** argv )
{
    assert( argc > 1 );
    int i;
    for( i = 1; i < argc; i++ ){
        printf( "%s\n", argv[ i ] );
    }
}
```

必ず、assert.hの  
インクルード前に記述

## 実行結果:

```
% gcc -o atest atest.c
% ./atest
```

通常は引数が2つ以上あるので、  
assertが呼び出されるはずである  
が、NDEBUGを定義することで、  
assertを無視できる

```
% gcc -o atest atest.c -DNDEBUG
とすることで、コンパイル時にマクロNDEBUGを定義できる
(この場合、ソースコード内で定義しなくてよい)
```

# #ifdef, #ifndef

## #ifdef / #ifndef MACRONAME

MACRONAMEが定義されていれば／されていなければ、その箇所をコンパイル

```
/* deftest.c */
#include <stdio.h>
#define ABC 2
int main( int argc, char** argv )
{
    int i;
#ifdef ABC
    printf(“%d¥n”, ABC );
    for( i = 1; i < argc; i++ ){
        printf(“%s¥n”, argv[ i ] );
    }
#else
    for ( i = 0; i < argc; i++ ){
        printf(“%s¥n”, argv[ i ] );
    }
#endif
}
```

```
/* deftest.c */
#include <stdio.h>
/* #define ABC 2 */
int main( int argc, char** argv )
{
    int i;
#ifdef ABC
    printf(“%d¥n”, ABC );
    for( i = 1; i < argc; i++ ){
        printf(“%s¥n”, argv[ i ] );
    }
#else
    for ( i = 0; i < argc; i++ ){
        printf(“%s¥n”, argv[ i ] );
    }
#endif
}
```

# #ifdef, #ifndef

```
/* deftest.c */
#include <stdio.h>
#define ABC 2
int main( int argc, char** argv )
{
    int i;
#ifdef ABC
    printf(“%d, ”, ABC );
    for( i = 1; i < argc; i++ ){
        printf(“%s, ”, argv[ i ] );
    }
#else
    for ( i = 0; i < argc; i++ ){
        printf(“%s, ”, argv[ i ] );
    }
#endif
}
```



```
% gcc -o deftest deftest.c
% ./deftest a b
2, a, b,
```

```
/* deftest.c */
#include <stdio.h>
/* #define ABC 2 */
int main( int argc, char** argv )
{
    int i;
#ifdef ABC
    printf(“%d, ”, ABC );
    for( i = 1; i < argc; i++ ){
        printf(“%s, ”, argv[ i ] );
    }
#else
    for ( i = 0; i < argc; i++ ){
        printf(“%s, ”, argv[ i ] );
    }
#endif
}
```



```
% gcc -o deftest deftest.c
% ./deftest a b
./deftest, a, b,
```

# #ifdef, #ifndef

% gcc -o deftest deftest.c -DABC=2  
とすることで、コンパイル時にマクロABCを定義できる

```
/* deftest.c */
#include <stdio.h>
#define ABC 2
int main( int argc, char** argv )
{
    int i;
    #ifndef ABC
        for( i = 1; i < argc; i++ ){
            printf("%s, ", argv[ i ] );
        }
    #else
        printf("%d, ", ABC );
        for ( i = 0; i < argc; i++ ){
            printf("%s, ", argv[ i ] );
        }
    #endif
}
```



```
% gcc -o deftest deftest.c
% ./deftest a b
2, ./deftest, a, b,
```

```
/* deftest.c */
#include <stdio.h>
/* #define ABC 2 */
int main( int argc, char** argv )
{
    int i;
    #ifndef ABC
        for( i = 1; i < argc; i++ ){
            printf("%s, ", argv[ i ] );
        }
    #else
        printf("%d, ", ABC );
        for ( i = 0; i < argc; i++ ){
            printf("%s, ", argv[ i ] );
        }
    #endif
}
```



```
% gcc -o deftest deftest.c
% ./deftest a b
a, b,
```

# インクルードガード

## 2重(多重)インクルード

・・・同じヘッダファイルを2回以上インクルードすること

/\* test.h \*/

```
#include "count.h"

void test(void){
    count();
}
```

/\* main.c \*/

```
#include <stdio.h>
#include "count.h"
#include "test.h"

void main(void){
    test();
}
```

/\* count.h \*/

```
void count(void){
    int k = 0;
    while( k != 100 ){
        printf("%d\n", k );
        k++;
    }
}
```

count.h が  
2回インクルード  
↓  
コンパイルエラー

# インクルードガード

## #ifndef と #define を用いて2重インクルードを防止

/\* main.c \*/

```
#include <stdio.h>
#include "count.h"
#include "test.h"

void main(void){
    test();
}
```

/\* test.h \*/

```
#include "count.h"

void test(void){
    count();
}
```

最初にcount.h が  
インクルードされると、  
マクロ\_COUNTが定義される。

2回目以降のインクルード時には  
この部分が読み込まれないので  
コンパイルエラーとはならない

/\* count.h \*/

```
#ifndef _COUNT
#define _COUNT
void count(void){
    int k = 0;
    while( k != 100 ){
        printf("%d¥n", k );
        k++;
    }
}
#endif
```



# デバッガ gdb (GNUデバッガ)

実行時のエラー, 不具合の原因を探る

実行例)

test.c というコードを デバッグ情報を付与 してコンパイル:

```
% gcc -g -O0 test.c
```

(-O0 は、最適化しないという意味)

デバッグ情報を付与した実行ファイル a.out を、gdb上で実行

```
% gdb a.out
```

```
(gdb) break main
```

```
(gdb) run
```

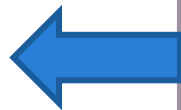
```
....
```

```
(gdb) next
```

```
....
```

1行ごとに実行していく

どこで問題が起きているのかを変数  
の中身などを見ながらチェック



# gdb の起動と使い方の基本

\$ gdb prog      起動 (progをデバッグ)

(gdb) r          実行開始

(gdb) b 10        10行目

(gdb) b func1    関数func1

(gdb) b prog.c:5   prog.cの5行目

← ブレークポイントの設定  
設定した箇所で停止する

(gdb) q          終了

余裕がある人は自分で調べておこう  
(ゲーム開発にはデバッグ技術が重要になります)

# 補足： 文字コード

## ソースファイルに日本語を書く場合

文字コードの種類を確認する

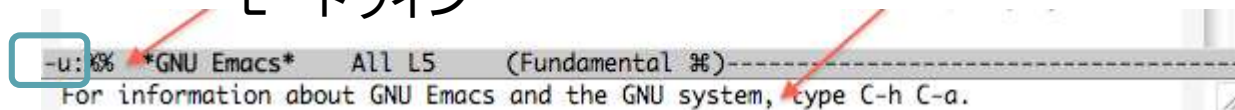
コードが違くと文字化けするので注意

## 文字コードの確認方法

Emacsの場合、モードラインの左端を見る

- E: 日本語EUC
- J: 日本語JIS
- S: 日本語Shift-JIS
- u: Unicode

モードライン



# 補足： 文字コード

## Emacsによる文字コードの変換

**Ctrl+x** [return] **f** [変換先文字コードの種類]

例) shift\_jisのファイルをutf-8で保存

- **Ctrl+x** [return] **f** utf-8

## 文字コードを間違えて開いてしまったとき

**Ctrl+x** [return] **c** [正しい文字コードの種類] [return]  
**M-x** revert-buffer

例) shift\_jisのファイルをutf-8で開いてしまった場合,  
shift\_jis で開きなおす

- **Ctrl+x** [return] **c** shift\_jis [return] **M-x** revert-buffer

# vim での文字コード確認

- 現在の文字コードの確認

```
:set enc?
```

- 既に開いているファイルを別の文字コードで開き直す

```
:e ++enc=[文字コード]
```

Ex.)

```
:e ++enc=utf-8
```

- 保存する文字コードの指定

```
:set fenc=[文字コード]
```

Ex.)

```
:set fenc=euc-jp
```

# 補足 文字コード

テキストファイルの文字コード変換コマンド(ツール)

**nkf**

使い方は、各自manで確認すること。

# レポート課題と提出方法(1/2)

提出締切は次回の講義開始時間とします

1. プログラミング作法についての自分なりの見解を、これまでの経験を踏まえて1000字程度で作文せよ。

# レポート課題と提出方法(2/2)

## 2. サンプルコード“sample\_prog.c”について

**[説明]** サンプルコードは、8x8のサイズの0か1の値が入った行列をファイルから読み込んで、1に挟まれている0を1に変換してテキストファイルに出力するプログラムである。以下の指示に従い、それぞれプログラムを修正せよ。(1)～(3)は別のファイル(カッコ内のファイル名)で提出すること。

- (1). 斜めでも挟めるように修正せよ。(sample\_prog\_kai1.c)
- (2). 縦横両方に挟まれている0のみを1に変換するように修正せよ。(sample\_prog\_kai2.c)
- (3). 最終的に出力される行列内に、1に挟まれている0が残らないように修正せよ。(sample\_prog\_kai3.c)

注意) プログラミング作法の講義で学んだことを取り入れること。

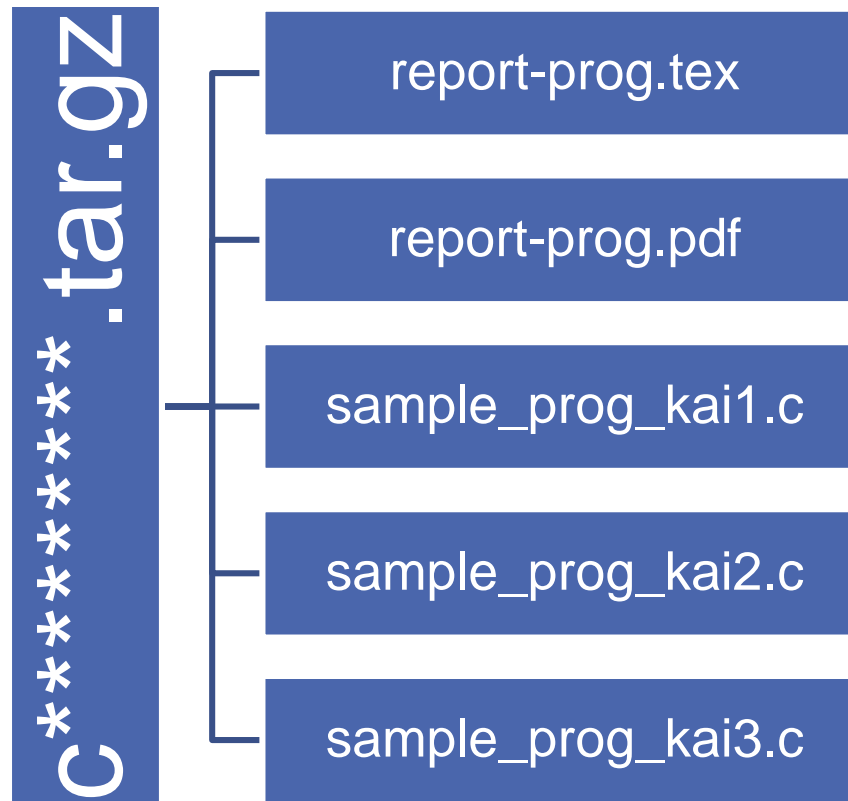
例) 関数化、インデント、コメントの付与, etc.



# 提出方法について

1, 2についてlatexでレポートにまとめ、(2は、どのように修正したのか、動作はどうなったかについてまとめる)、pdfファイル(report-prog.pdf)に変換したものおよび2の(1)~(3)で作成したソースコードと一緒に“ログインアカウント名.tar.gz”というファイル名で圧縮してManabaから提出すること。

提出ファイルの構成



# 次週の予習に関して

次週の講義内容は、「ライブラリ化」です。

レポート提出ができた人は、次週までに以下について予習しておいてください。

- 「分割コンパイル」
  - プログラムを複数ファイルに分割し、コンパイルする方法
- 「make」、「Makefile」
  - makeコマンドの使い方、Makefileの書き方
- 「静的ライブラリ」、「動的ライブラリ」
  - ライブラリの作成、使用方法