

# ソフトウェア設計及び実験 第3回 ライブラリ化

担当： 松本 和幸

# 本日の講義内容

- プログラムのコンパイル
- プログラムの分割と分割コンパイル
- Makefileの書き方
  - マクロ
  - サフィックスルール
- ライブラリの作成
  - 静的ライブラリ
  - 動的ライブラリ
- レポート課題

# プログラムのコンパイル

- ・ ソースファイルをコンパイルして実行

/\* test.c \*/

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

int main( void )
{
    int x, y, z;
    x = 5;
    y = 2;
    z = pow( x, y );
    printf( "hello: %d¥n", z );
    return 1;
}
```

コンパイル

```
$ gcc -o test test.c -lm
```

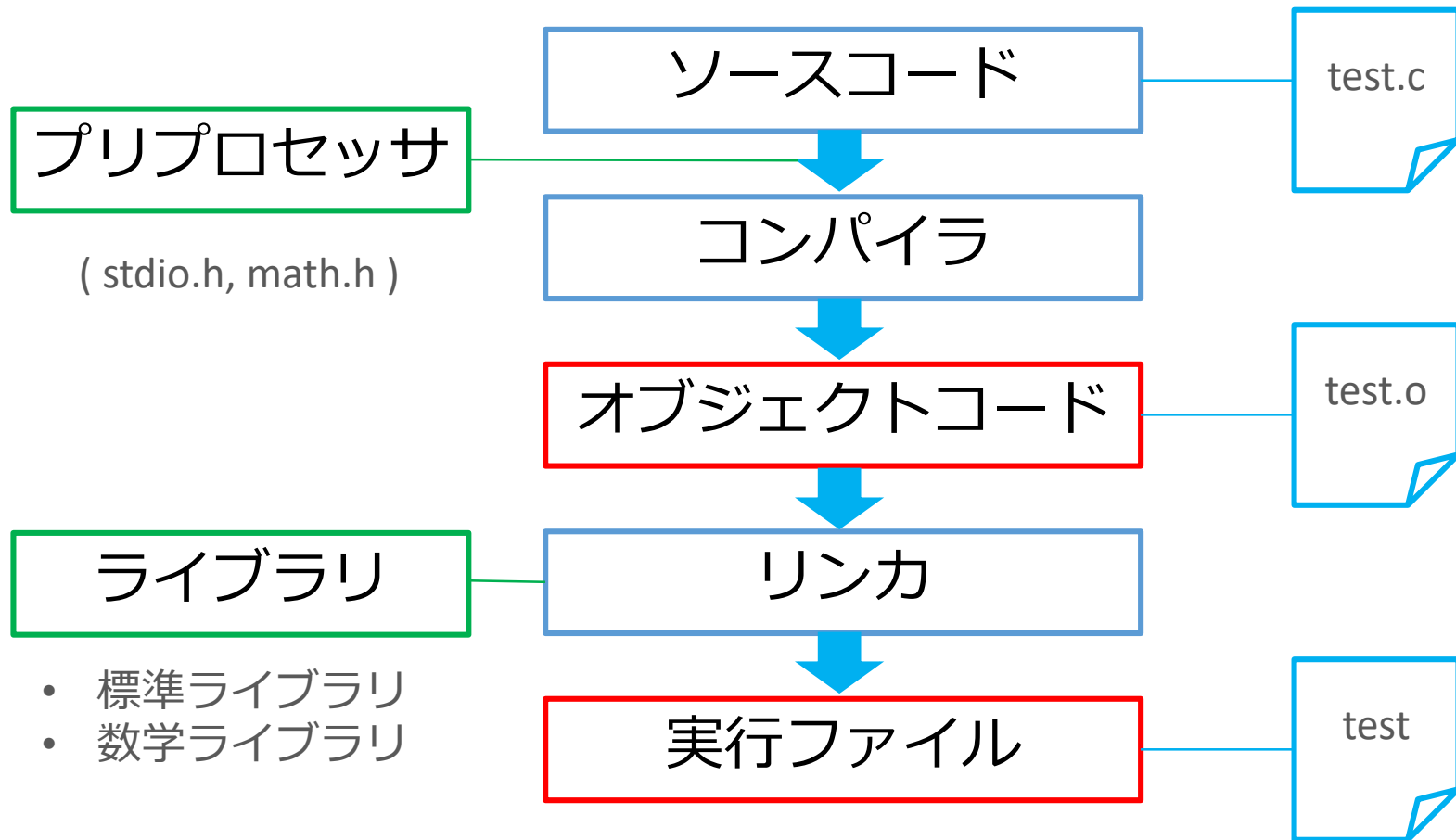


実行

```
$ ./test
hello: 25
```

# プログラム（実行ファイル）はどのようにして生成されるのか？

## ・コンパイルの流れ



# プログラムの分割と 分割コンパイル

# プログラムの分割

- 関数 func1 を、別のプログラムからも使用したい

/\* prog1.c \*/

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

int func1( int x, int y ){
    return pow( x, y * 2 );
}

int main( void )
{
    int x = 5;
    int y = 2;
    int z = func1( x, y );
    printf( "result: %d¥n", z );
    return 1;
}
```

/\* prog2.c \*/

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

int main( void )
{
    int a = func1( 10, 2 );
    printf( "%d¥n", a );
    return 1;
}
```

# プログラムの分割

- 関数 func1 を、別のプログラムからも使用したい

/\* prog1.c \*/

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

```
int func1( int x, int y ){
    return pow( x, y * 2 );
}
```

```
int main( void )
{
    int x = 5;
    int y = 2;
    int z = func1( x, y );
    printf( "result: %d¥n", z );
    return 1;
}
```

/\* prog2.c \*/

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

```
int main( void )
{
```

```
    int a = func1( 10, 2 );
    printf( "%d¥n", a );
    return 1;
}
```

# プログラムの分割

- prog1.c から、関数 func1 を切り出す

/\* prog1.c \*/

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

```
int main( void )
{
    int x = 5;
    int y = 2;
    int z = func1( x, y );
    printf( "result: %d¥n", z );
    return 1;
}
```

/\* func.h \*/

```
#include <stdio.h>
#include <math.h>
Int func1( int x, int y );
```

/\* func.c \*/

```
int func1( int x, int y )
{
    return pow( x, y * 2 );
}
```

```
#include <stdio.h>

int main( void )
{
    int a = func1( 10, 2 );
    printf( "%d¥n", a );
    return 1;
}
```



# 分割コンパイル

- ・プログラムを分割した際のコンパイル方法：

```
$ gcc -o prog1 prog1.c func.c -lm  
$ gcc -o prog2 prog2.c func.c -lm
```

} プログラム作成に必要なソースファイルを指定する

または、

```
$ gcc -c prog1.c prog2.c func.c
```

} オブジェクトファイルの生成

```
$ gcc -o prog1 prog1.o func.o -lm  
$ gcc -o prog2 prog2.o func.o -lm
```

} 実行ファイルの生成  
(オブジェクトファイルのリンク)

# プロトタイプ宣言（復習）

- コンパイルの際に、関数がどのような引数、戻り値をとるのかを、関数の使用前にコンパイラが知っておく必要がある
  - プロトタイプ宣言によりコンパイラに知らせる
- プログラムを分割した場合、違うソースコード内で定義された関数を呼び出すことになる
  - 正しい使われ方をしているかどうかを確認するために、プロトタイプ宣言を一箇所にまとめておく（ヘッダファイルに記述）
  - ソースコードごとにプロトタイプ宣言をしてしまうとバグの原因

# 変数（復習）

- 変数の宣言方法によって有効範囲やふるまいが変わる
- 宣言された場所によって変わるので注意：
  - 関数（ブロック）内で宣言
  - 関数外で宣言
- 変数の宣言に用いるキーワード：
  - `auto` （何もつけないのと同じ）
  - `static`
  - `extern`

分割コンパイル時には必須の知識なので  
使い方を熟知しておくこと

# auto, static, extern キーワード

キーワード	宣言場所	
	関数の内部	関数の外部
キーワードなし(何もつけない)	局所変数。"auto"と同じ意味	大域変数。プログラム内のすべてのファイルの関数から参照可能
auto	スコープはブロック内に制限される	使わない
static	スコープはブロック内に制限される	スコープはファイル内に制限される
extern	使わない	コンパイラに変数の存在を知らせるだけで変数の実体にはならない(関数のプロトタイプ宣言と同様の役割)。

# グローバル変数使用例（分割コンパイル）

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

```
int count; /* グローバル変数 */
void main( void )
{
    printf("hello, world.¥n");
    display(); count++;
    display(); count++;
    display(); count++;
}
```

```
/* func.h */
void display( void );
```

```
/* disp.c */
extern int count;
void display( void )
{
    int i;
    for ( i = 0; i < count; i++ ){
        puts(":::::::::");
    }
    count+=5;
}
```

```
$ gcc -c disp.c
$ gcc -o prog main.c disp.o
```

分割コンパイルをして  
動作を確かめてみよう

# Makefile

# メイクファイル

## Makefile（メイクファイル）とは？

- プログラムのコンパイル作業を**自動化**する
- 分割コンパイルでは、分割したファイルを、それぞれコンパイルしてからリンクする必要があった
- Makefileとは、コンパイルやリンクの作業を一括で行うための**一連の作業を記述したファイル**
- **make**コマンドを実行すると、Makefileに書かれた内容が実行される



# メイクファイルの基本形

- 基本的な書式は以下の通り

[ターゲット名]: 依存関係ファイル名

[TAB] コマンド行

- “prog”という実行形式ファイルを作成する場合

```
prog: prog.c
```

```
gcc -o prog prog.c
```



# 具体的なメイクファイル

```
$ gcc -c prog1.c prog2.c func.c
$ gcc -o prog1 prog1.o func.o -lm
$ gcc -o prog2 prog2.o func.o -lm
```

Makefile で書くと…



```
prog1: prog1.c func.c ← 依存関係 : prog1の作成に必要なファイル
      gcc -c prog1.c func.c
      gcc -o prog1 prog1.o func.o -lm
prog2: prog2.c func.c ← 依存関係 : prog2の作成に必要なファイル
      gcc -c prog2.c func.c
      gcc -o prog2 prog2.o func.o -lm
```

# 具体的なメイクファイル

```
$ gcc -c prog1.c prog2.c func.c  
$ gcc -o prog1 prog1.o func.o -lm  
$ gcc -o prog2 prog2.o func.o -lm
```

Makefile で書くと…



```
prog1: prog1.c func.c  
       gcc -c prog1.c func.c  
       gcc -o prog1 prog1.o func.o -lm  
prog2: prog2.c func.c  
       gcc -c prog2.c func.c  
       gcc -o prog2 prog2.o func.o -lm
```

make方法

```
$ make prog1  
gcc -c prog1.c func.c  
gcc -o prog1 prog1.o func.o -lm
```

```
$ make prog2  
gcc -c prog2.c func.c  
gcc -o prog2 prog2.o func.o -lm
```

# メイクファイルを用いるメリット

- メイクファイル、makeを上手く使うと…

あるオブジェクトファイルや実行ファイルを作成するために必要なソースファイルやヘッダファイルが更新されていた場合に、そのオブジェクトファイルや実行ファイルを再コンパイルしてくれる

- 更新されていない場合は再コンパイルしない  
(コンパイルにかかる時間を節約)

マクロ

# ユーザ定義マクロ

- ・ターゲットがいくつものファイルに依存するため、繰り返しが多くなる
- ・ファイル名の変更があるたびに修正 ➡ 大変

```
prog1: prog.c func1.c func2.c
       gcc -c prog.c func1.c func2.c
       gcc -o prog1 prog1.o func1.o func2.o -lm

prog2: prog.c func3.c func4.c
       gcc -c prog.c func3.c func4.c
       gcc -o prog.o func3.o func4.o -lm
```

マクロを用いて、置き換える（変数のように扱う）

例)

```
SRCS = prog.c func1.c func2.c func3.c func4.c
FUNCS = func1.c func2.c func3.c func4.c
OBSJS = prog.o func1.o func2.o func3.o func4.o
```

# ユーザ定義マクロ

- マクロを用いて置き換えた例

```
SRC = prog.c
FUNCS1 = func1.c func2.c
FUNCS2 = func3.c func4.c
OBS1 = prog.o func1.o func2.o
OBS2 = prog.o func3.o func4.o
LFLAG = -lm

prog1: $(SRC) $(FUNCS1)
    gcc -c $(SRC) $(FUNCS1)
    gcc -o prog1 $(OBS1) $(LFLAG)
prog2: $(SRC) $(FUNCS2)
    gcc -c $(SRC) $(FUNCS2)
    gcc -o prog2 $(OBS2) $(LFLAG)
```

# 内部マクロ

- 内部マクロ（デフォルトマクロ）

あらかじめmakeで定義されているマクロ（CCなどは再定義可能）

```
SRC = prog.c
FUNCS1 = func1.c func2.c
FUNCS2 = func3.c func4.c
OBS1 = prog.o func1.o func2.o
OBS2 = prog.o func3.o func4.o
LFLAG = -lm
```

```
prog1: $(SRC) $(FUNCS1)
    $(CC) -c $^
    $(CC) -o $@ $(OBS1) $(LFLAG)
prog2: $(SRC) $(FUNCS2)
    $(CC) -c $^
    $(CC) -o $@ $(OBS2) $(LFLAG)
```

- CC: Cコンパイラ
- \$^: 依存ファイルリスト
- \$@: ターゲットファイル

# サフィックスルール



# サフィックス（接尾辞）ルール

- コンパイルやアセンブルなどの作業は、どのプログラムでもほとんど同じ手順
  - **拡張子（接尾辞）などの暗黙のルール**を使って記述をより簡略化

例)

C言語のソースファイルは常に .c というサフィックス  
“.c” から “.o” (オブジェクトファイル) が必ず作られる

[名前].c → [名前].o

c.o: → .o というファイルが必要なら .c から作るというルール

**.c.o:**

**\$(CC) -c \$<**

# サフィックスルールの具体例

## • Makefile

```
CC = gcc
TARGET = prog
SRCS = main.c func1.c func2.c
HEADERS = func1.h
OBJS = main.o func1.o
OBJS2 = main.o func2.o
LIB = -lm

all: $(TARGET)

prog: $(OBJS)
    $(CC) -o $@ $(OBJS) $(LIB)
prog2: $(OBJS2)
    $(CC) -o $@ $(OBJS2) $(LIB)
.c.o: $(SRCS)
    $(CC) -c $(SRCS)

clean:
    rm -rf $(OBJS) $(TARGET) *~
```

- オブジェクトファイル生成を一括で記述できる

# サフィックスルールを作る

## • Makefile

```
CC = gcc
SRCS = main.c func1.c
HEADERS = func1.h
OBJS = main.o func1.o
LIB = -lm
TEX = platex
PDF = dvipdfmx
```

**.SUFFIXES:**

**.SUFFIXES:** .tex .dvi .pdf

サフィックスルールに必要な  
接尾辞(サフィックス)の定義

texファイルをコンパイルしてできた  
dviファイルをpdfファイルに変換する

```
prog: $(OBJS)
      $(CC) -o $@ $(OBJS) $(LIB)

.c.o: $(SRCS)
      $(CC) -c $(SRCS)
```

**.tex.pdf:**

```
      $(TEX) $<
      $(PDF) $*.dvi
```

Make実行例:

```
$ make report.pdf
```

```
clean:
      rm -rf $(OBJS) $(TARGET) *~ *.dvi *.pdf
```

# デフォルトマクロ一覧

内部マクロ	意味	使える場所
\$@	ターゲット名	コマンド行
\$%	ターゲットのメンバ名	コマンド行
\$<	依存ファイルの1番最初のファイル名	コマンド行
\$?	依存ファイル中、ターゲットより更新日時が新しいファイルのリスト	コマンド行
\$^	依存ファイルリスト	コマンド行
\$\$@	ターゲット名	依存関係行
\$*	サフィックスを除いたファイル名	サフィックスルール内
<b>\$&lt;</b>	現在処理中のターゲットよりも後で変更された依存ファイルのリスト	<b>サフィックスルール内</b>

# デフォルトマクロ一覧

内部マクロ	意味	デフォルト値
CC	Cコンパイラ	cc
AR	アーカイブコマンド	ar
CXX	C++コンパイラ	g++
TEX	Texコマンド	tex
RM	ファイル削除コマンド	rm -rf
CFLAGS	CCの引数	
LDFLAGS	リンカldの引数	

# makeのその他の機能

# PHONY TARGET

- コンパイルなどのファイル生成以外のコマンドを実行したい場合

```
clean:  
    rm -rf $(OBJDIR) *~ *.dvi *.pdf
```

- 依存するファイルが無く、コマンドを実行するだけ
- . . . clean は、"phony (偽りの) target"
- phonyターゲットを使用する場合、ターゲット名と同じファイルがあると問題が起こるので、以下のように書く

```
.PHONY: clean  
clean:  
    rm -rf $(OBJDIR) *~ *.dvi *.pdf
```

# VPATH

- 大規模プロジェクトでの開発になると…
  - ソースファイルと実行形式ファイルなどのバイナリファイルを別のディレクトリに置く方が望ましい
  - 依存関係のソースコードがどこにあるかを、ディレクトリ構成が変わるたびに書き直すのは面倒
  - → makeには、**ディレクトリサーチ機能** がある
  - カレントディレクトリに依存関係ファイルが無い場合、**VPATH**で指定したパスを探す

```
VPATH = src:../headers
```

**src, ../headers** の順でディレクトリ内を探索する



# VPATH ディレクティブ

- ファイルの種類を指定して検索することができる

例)

- “../headers” というディレクトリ内に、“\*.h” の拡張子を持つヘッダファイルを探しに行く
- “../src” というディレクトリ内に、“\*.c” の拡張子を持つソースファイルを探しに行く

```
vpath %.h ../headers  
vpath %.c ../src
```

# さらに高度なMakefile使用法

- ディレクトリパスの取得、カレントディレクトリの変更など

```
curdir = $(shell pwd)
all:
    echo $(curdir)
    (cd src; make; )
```

- Makefile内ではshell(bash)の文法が使えるが、  
コマンドを1行内に書く必要がある
  - 複数行にわたるスクリプトの場合、バックスラッシュ`\'('¥')を使用

```
files=$(shell ls)
all:
    for f in ${files};
    do (echo $$f; ); ¥
    done
```

複雑な処理になる場合は  
Makefileから分離した方が  
効率が良い

ライブラリ

# ライブラリ

ある機能を持ったプログラムをほかのプログラムから利用できるように部品化したもの（部品を集めたもの=ライブラリ群）

- ソフト実験で使われるライブラリ（一部）：
- **SDL** (Simple DirectMedia Layer)
  - C言語で書かれたクロスプラットフォームライブラリ
    - Perl, Python Perl, Javaなど各種言語からの使用が可能
  - グラフィックス描画, 音楽再生など
- **OpenCV** (Open Source Computer Vision Library)
  - 画像処理、画像解析、機械学習関連のライブラリ
  - 主にコンピュータビジョン向けに利用

# パッケージ、モジュール、コンポーネントとの違い

- プログラミング言語によってはほぼ同等の意味で用いられることも
  - 部品という意味では同じような意味合いで使われることがある
  - ライブラリの方がより広い意味で使われる
- **パッケージ**： オブジェクトや関数の宣言などを、お互い関連するものごとにひとまとめに集めたもの
- **モジュール**： システムの一部を構成するひとまとまりの機能を持った部品で、システムや他の部品への接合部(インターフェース)の仕様が規格化・標準化されていて、容易に追加や交換ができるようなもの
- **コンポーネント**： 特定の機能を持ち単体で完結しているが、単体では使用せず(できず)、他のプログラムから呼び出されたり連結されたりして使われるプログラム部品

# フレームワークとの違い

- ・フレームワーク上で用意されたテンプレートのようなものに自作のプログラムを組み込む形  
(フレームワークが主体)
- ・ライブラリは、プログラマが既存の関数群を使用する  
(部品として扱う) ため、自作部分が主体となる

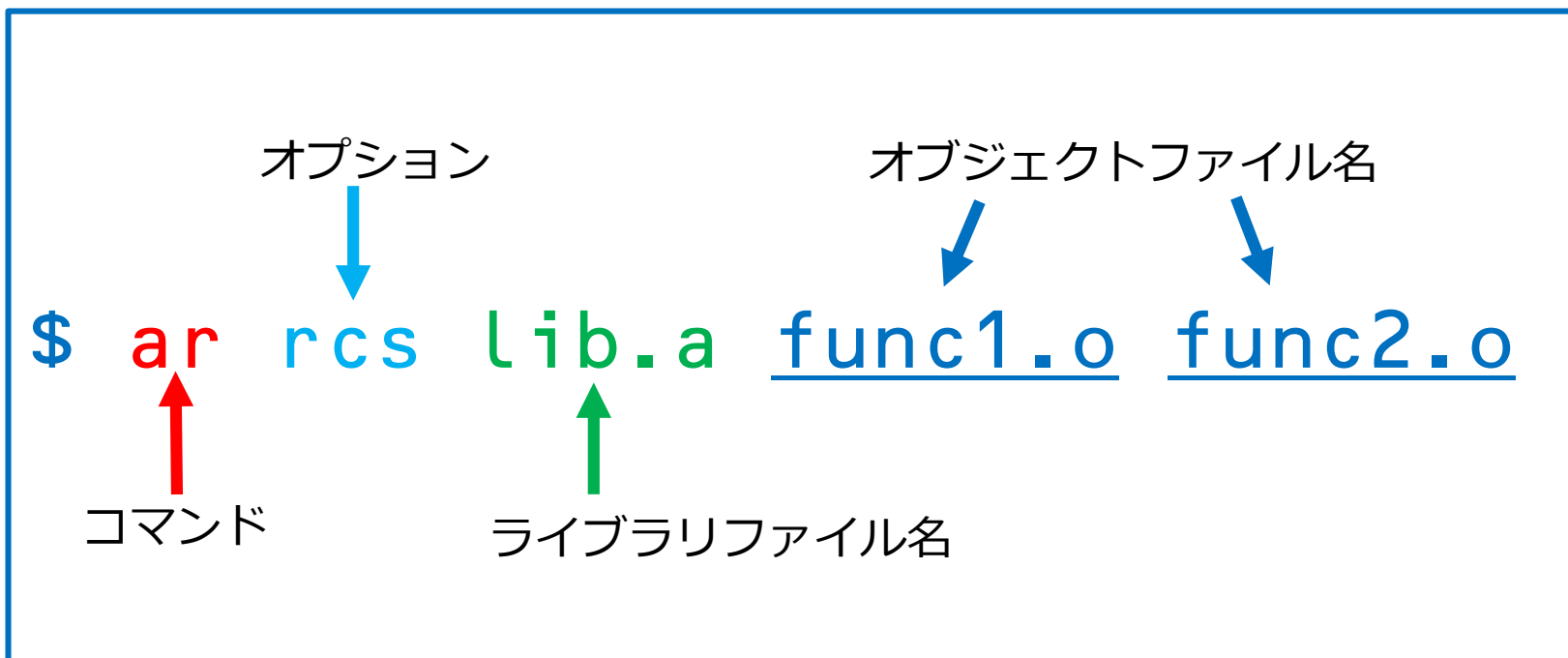
※ 本来の意味で使用されていない場合もあるので注意

# ライブラリの種類

- 静的ライブラリ(Static Link Library)
- 動的ライブラリ(Dynamic Link Library)

# 静的ライブラリの作成

- **.a** という拡張子
- オブジェクトファイルをまとめたもの





# 作成した静的ライブラリの利用法

- Makefile内での記述 (makefile\_sample.tar.gz参考)

```
TARGET=prog
CC=gcc
AR=ar
AROPS=r cs
MAIN=main.c
SRCS=func1.c func2.c
LIB=lib.a
OBJS=func1.o func2.o

$(TARGET): $(LIB)
    $(CC) -o $@ $(MAIN) $^
$(LIB): $(OBJS)
    $(AR) $(AROPS) $@ $^
.c.o:
    $(CC) -c $<
```

# 動的ライブラリ（共有ライブラリ）

- **.so** という拡張子
- プログラムを**実行したときに読み込まれる**
- ある機能を**複数のプログラムで共有**する場合、共有ライブラリが便利
- **機能追加、修正が容易**（ライブラリを更新するだけ）
- 更新した場合でも、古いバージョンを維持できる

# 動的ライブラリの作成

オプション：共有ライブラリ作成に必要

ライブラリファイル名




```
$ gcc -shared -fPIC -o libfunc.so func.c
```


# 動的ライブラリの利用法

- ・作成した動的ライブラ리를 링크

ライブラリファイル名  
(拡張子と先頭の"lib"を除去)



```
$ gcc -I./ -L./ -o prog main.c -lfunc
```



インクルードファイルのあるディレクトリと  
ライブラリファイルのあるディレクトリを指定

makefile\_sample\_so.tar.gz 参照

# リンクしているライブラリの確認

- lddコマンドを使用する

```
$ ldd prog
linux-vdso.so.1 => (0x00007ffccad8b000)
      libfunc.so => ./libfunc.so
(0x00007f3c72e47000)
      libc.so.6 => /lib64/libc.so.6
(0x00007f3c72a66000)
      /lib64/ld-linux-x86-64.so.2
(0x000055ead998b000)
```

# よく使われるライブラリ

- 一般的なもの
  - OpenGL (Open Graphics Library)
  - STL (Standard Template Library)
- ゲーム開発（周辺機器関連）
  - libwiimote … wii リモコンを使用するために利用
- このほかにもいろいろあるので自分で調べてみよう

# レポート課題（1/2）

提出締め切りは次回の講義開始時

1. ライブラリの実用例を1つ調べ、それについて文章でまとめ、自作のプログラムにどのように使用したいか（または、使用したか）について1000字程度で作文せよ
2. Makefileについて、講義で紹介した機能以外のものについて2つ以上調べて説明せよ
3. make以外のビルドツールについて1つ以上調べて、makeとの違い・優れている点などを説明せよ

# レポート課題（2/2）

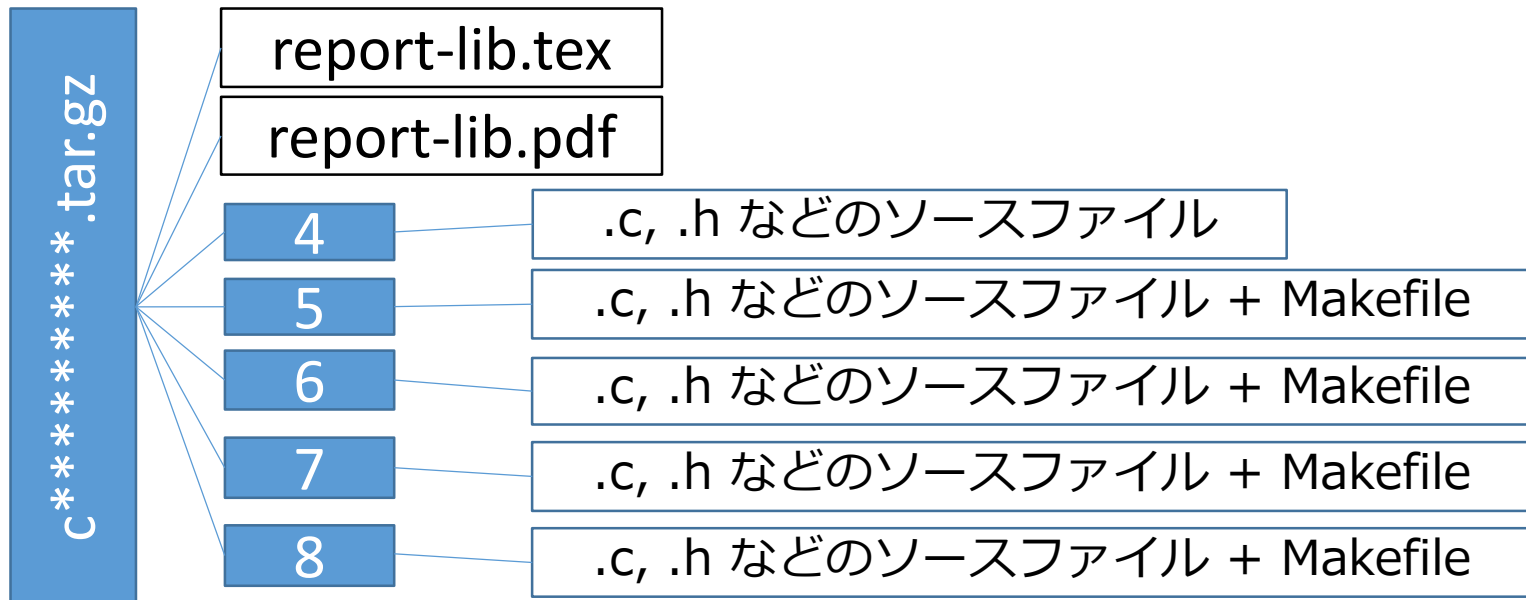
提出締め切りは次回の講義開始時

4. サンプルプログラム（著者当てプログラム）について、以下の機能をmainから分離し、プログラムの分割をせよ
  - ・ ファイル名と著者名などを記載した“database.csv”を読み込む関数
  - ・ 質問する関数
  - ・ 結果と作品名を表示する関数
5. 3.で分割したプログラムをMakefileで一括コンパイルできるようにせよ（マクロ、サフィックスルールも使用すること）
6. 4. のMakefileにライブラリ化（静的）をおこない、リンクする記述を追加せよ（サンプルメイクファイルを参考に）
7. 4.のMakefileに動的ライブラリの作成をおこなう記述を追加せよ（サンプルメイクファイルを参考に）
8. 【発展課題】より正確に著者を当てられるように改良せよ（作品名は固有なのでそのまま質問に含めてはいけない）



# 提出方法、ファイル構成

- ① 1～8についてLaTeXでまとめた（ソースコードの貼り付けは不要）ファイル“report-lib.tex”
- ② ①をpdfファイルに変換した“report-lib.pdf”
- ③ 4～8の課題で作成したソースコード一式を、各課題番号ごとのディレクトリを作成してそのなかに入れる
- ④ cアカウント番号.tar.gz の名前で圧縮してManabaで提出



# 課題のヒント（１）

- データベースをファイルから読み込む部分、質問を出す部分、回答をする部分の３つの部分に分けて関数を作る。その関数をそれぞれ別々のファイルに分割する。
  - 例) `read_database.c`, `question.c`, `answer.c`
- `main.c` を作って、`main`関数から上記の３つの関数を呼び出す
- 複数の関数において共通して使用する変数や構造体は、引数や戻り値、または大域変数として定義する（大域変数とする場合にはヘッダファイルを作ってインクルードするとよい。インクルードガードを使うと便利）。
- マクロや関数の宣言もヘッダファイルに記述する。
- `Makefile`には、サフィックスルールを使ってソースファイル(`.c`)からオブジェクトコード(`.o`)を生成する記述を含める。
- `make`できない場合には、まず、変数の整理からはじめよう。
  - どの関数がどの変数をどこで参照、初期化、更新しているのか確認

## 課題のヒント（２）

- ・ より正確な回答をするためには・・・
  - ・ 元のプログラムはランダムで質問をしている
    - ・ 質問回数上限が来たら終了してしまう
  - ・ すべての人物についてのチェック処理をしていない
  - ・ たとえば、「東京」出身の人物は大勢いるが、ランダムで選択された人物のみがチェックされる仕様になっているので、チェックされないままの人物もある
- ・ 新しいチェックの仕方を検討してみる
  - ・ 毎回の質問の回答結果について、すべての人物と照合してポイント付ける（今回はデータ数が少ないので可能。データが多くなると…）
  - ・ データを読み込んだ時点で、質問ごとにデータをグループ分けしておく