プログラミングとデバッグ

九州大学大学院 システム情報科学研究院
石田 繁巳 <ishida@f.ait.kyushu-u.ac.jp>

2014/07/03

九州大学
KYUSHU UNIVERSITY

アウトライン

- この講義の目的
- プログラミングの手順
- デバッグに向けた基礎知識
- デバッグ実践
- gdbを使ったデバッグ
- トレーサを使ったデバッグ

この講義の目的

- ソフト開発（プログラミング）手順について学ぶ
  - 企業等での開発と研究での開発の違いを知る
  - 上記を使い分けられるようにする
- 実践を通じてデバッグの考え方を理解する

※ C言語を使います

- 組込み分野ではまだまだ主流なので
- C言語の文法や記法に関してはこの講義の対象外とします（自分で勉強すること）

プログラミングの手順

手順の概要

1. 設計
    - 慣れてきて，小規模なものを1人で作る場合は脳内でやってしまうことが多い
2. ソースを書く
3. ビルド
4. 実行・検証
5. デバッグ
    - 修正 → ビルド → 実行・検証を繰り返す

というのは

半分嘘です

企業での開発と研究での開発

■ 企業での開発

- **「売り物」となるソフト**を開発する
  - 品質を保証することが必須

⇒ 開発プロセスに則った開発

■ 研究での開発

- **動くものを早く**開発する
  - 自分が必要とする正常系が動けばおｋ

※ ただし，正常系の動きはきちんと検証する

⇒ デバッグをしながらの開発

研究での開発手順

1. 全体構成を考える
    - 主要部分の設計に相当する
2. 部分的に作ってみる
3. 予定通りの動きになっているか確認
4. 2, 3を繰り返す

デバッグ能力が極めて重要

5. 全体ができたら期待通りの結果が得られるか確認
    - ダメなら結果について考察し，最初に戻って「考える」ところから再実行

デバッグ

- Debug（De-bug）
  - バグ（bug: 虫）を取り除く（de）作業

- Debugの手順

1. バグの原因を見つける
   - 明確に原因を特定する
2. 対策を考える
3. 修正する
4. 確認する

デバッグに向けた基礎知識

ビルド

- ビルド ＝ コンパイル＋リンク
  - コンパイル
    - ソースコードをrelocatable object（単にobjectとも言う）ファイルに変換する作業
  - リンク
    - relocatable objectファイルと必要なライブラリを1つにまとめて実行可能ファイルを作る作業

⇒ ビルドすることでソースコードから実行可能なファイルが生成される

cf.) make

- make
  - ビルド補助プログラム
    - Unix系OSで一般的に使用されている
  - Makefileというファイルにビルドに必要な情報を書いておき，makeを実行するだけでビルドできる

⇒ ビルドのことをmake（メイク）と呼ぶ場合がある

ライブラリ

- ライブラリ（library）
  - 色々なプログラムで利用するであろう関数群をまとめたrelocatable objectファイル

- 静的ライブラリ（static library）
  - リンク時に実行可能ファイルに書き出される
- 共有ライブラリ（shared library）
  - 実行時に自動的に読み込まれる
  - Windows→xx.dll，Mac→xx.dylib，Linux→libxx.so

ビルドの詳細を見てみよう

- 準備
  - ssh *username*@133.5.151.36
  - cp -r /usr/src/ex ./
  - cd ex/ex01/
- コンパイル
  - gcc -v -c test.c
- リンク
  - gcc -o test -v test.o
- 実行
  - ./test

test.c

```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return(0);
}
```

よく分からない人は

- この辺を見て勉強してね
  - 先輩教えて！プログラミングのabc（第1回）—コンパイルとビルドって何が違うの（上）
    - http://itpro.nikkeibp.co.jp/members/NBY/techsquare/20021008/1/

分割コンパイル

- 開発規模が大きい場合など
  - 関数がたくさん
  - 1つの関数の変更で全体をビルドし直すのは時間のムダ

    ⇒ 分割コンパイル
- 分割コンパイル
  - 各ファイルをコンパイルしてobjectファイルを生成
  - 最後に全部のobjectファイルをリンク

分割コンパイルしてみよう(1) 10分

- 準備
  - `cd ~/ex/ex02/`
- コンパイル
  - `gcc -c welcome.c`
  - `gcc -c main.c`
- リンク
  - `gcc -o welcome main.o welcome.o`
- 実行
  - `./welcome`

分割コンパイルしてみよう(2) 10分

- 書き換え
  - emacs main.c（viでもnanoでもお好きなもので）
    - Flab → the world
- 変更ファイルだけコンパイル
  - gcc -c main.c
- リンク
  - gcc -o welcome main.o welcome.o
- 実行
  - ./welcome

makeしてみよう

- 準備
  - `cd ~/ex/ex03`
- ビルド
  - `make`
- 実行
  - `./welcome`

デバッグ実践

演習1

- 準備
  - cd ~/ex/en01
- 現状
  - makeできません
- 演習問題
  - 原因を特定し，そう判断した理由とともに答えて下さい
  - 修正方法を提示して下さい
    - warningを消す方法も分かれば提示して下さい

演習2

- 準備
  - cd ~/ex/en02
- 現状
  - makeできません
- 演習問題
  - 原因を特定し，そう判断した理由とともに答えて下さい
  - 可能であれば，修正方法を提示して下さい

gdbを使ったデバッグ

gdb

- GNU Debugger
  - プログラムを途中で止めたり，変数の中身を見たりすることができる
- 前提
  - gccで-gオプションを付けてコンパイルしてある
    - 変数名等の情報が実行可能ファイルに埋め込まれる
- 使い方
  - ググってね

gdbを使ってみよう (1)

- 準備
  - cd ~/ex/ex04
  - make debug
- gdbを実行
  - gdb ./main
- ソースの表示
  - (gdb) list arg_handler
- ブレークポイントの設定
  - (gdb) b main.c:55
- ブレークポイントの確認
  - (gdb) info break（i bでも大丈夫）

gdbを使ってみよう (2)

- プログラムの実行
  - (gdb) r -d4 hoge （-d4 hogeは引数）
- 変数の中身を見る
  - (gdb) p argc
  - (gdb) p argv[1]
  - (gdb) x/4c argv[1]
- 続きを実行
  - (gdb) c

トレーサを使ったデバッグ

gdbの欠点

- 再現性の低いバグの発見が困難
  - プログラムを何回も止めて検証するのは非現実的

  ⇒ トレーサを利用
- トレーサ
  - 以下のような情報を画面に表示したりファイルに書き出したりする機能
    - 実行された関数
    - 変数の中身

トレーサを使ってみよう

- 準備
  - cd ~/ex/ex05
- トレーサの追加
  - T_M( )
    - printfとほぼ同じもの
  - T_D( )
    - 16進ダンプを表示する
  - 詳細は trace.h を参照
- ビルド
  - make
- 実行
  - ./main 100

次回のお話

次回に向けて

- このrepositoryをforkして，pull requestの練習をして下さい
  - https://github.com/fukuda-ubi-edu2014/pull-req-ex
  - ダメそうならgit cloneできるようにしておいてください

- 次回はがっつりデバッグに挑戦してもらいます
  - 途中で誰かを指名して解説してもらう形を取ります