

プログラミングII(前期)(2)

担当：電気情報工学科 奥村紀之

試験について(補足)

- 再試験等はやりません。
 - 前期中間、前期期末、後期中間と最終レポートが定期試験の評価
 - 適宜提示する課題で課題点の評価
- 分からないことがあったり、試験の成績が良くなかったなど、単位取得に問題がありそうなきはいつでも質問に来て下さい
 - こちらも精一杯分かるまで教えます
 - 試験の成績は変えられませんが、次の試験の成績を上げるためのお手伝いはできます
- 後期の最終レポートの点数が決まった時点で、1年間の成績を確定します
 - それまでに最大限努力して下さい
 - 課題点が50%あるので、最低でもそちらを100%こなしておけば試験の比重は決して高いものではありません






今日のお題

- Gitによるソースコードの管理
- プログラムを書いて実行するまで

Git-itのダウンロード

- <https://github.com/jlord/git-it-electron/releases>
 - ここからGit-it-Linux-x64.zipをダウンロードして下さい
 - 自分のパソコンを持ってきている人は、Gitをインストールしましょう
 - <https://git-scm.com/book/ja/v1/使い始める-Gitのインストール>

Downloads

 Git-it-Linux-x64.zip	46 MB
 Git-it-Mac-x64.zip	43.9 MB
 Git-it-Win-ia32.zip	118 MB
 Source code (zip)	
 Source code (tar.gz)	

Gitの確認

- Git-itを起動しましょう
 - Macの場合怒られるかもしれません
 - 右クリックで開きましょう
- Gitの設定をするの項目を進めてください
 - 最後にVerifyボタンを押すとここまでの作業を確認してくれます
- アドレスは学校のにしましょう
 - s.akashi.ac.jpのやつです

Step: Gitの設定をする

インストールできたら、**ターミナル**(Bashとか、Shellとか、Promptとか言った別の名前と呼ばれることも多いです)を開いてください。Gitがちゃんとインストールできたか、次のようにして確認できます。:

```
$ git --version
```

このコマンドは、みなさんがいま使っているGitのバージョンを返してくれます。こんな風に表示されるはずです:

```
git version 1.9.1
```

(1.7.10 より上のバージョンであれば問題ありません。)

次に、変更が誰によるものなのかをGitがわかるように設定します:

名前を設定しましょう:

```
$ git config --global user.name "<あなたの名前>"
```

次にEメールアドレスです:

```
$ git config --global user.email "<youremail@example.com(あなたのEメールアドレス)>"
```

VERIFY

Gitの確認

- うまくいくとこんな画面になります
 - 右下のRepositoryに進んでください



リポジトリ作成

- ホームディレクトリの下にこの科目用のディレクトリを作成しましょう
 - `mkdir 2e-programming` とか
- さらにその下に、Git-itの例の通りディレクトリを作ってください
- `git init`を実行したディレクトリをSelect Directoryで指定してVerifyしてみてください

Step: リポジトリを作ろう

新しいフォルダを作って、Gitリポジトリとして使えるように初期化しよう。

簡単にするために、フォルダ名をこれから始めたいプロジェクトの名前としてつけましょう。'hello-world'はどうでしょうか？

ターミナルの中で次のコマンドを1つずつ実行してみましょう。

新しいフォルダを作る:

```
$ mkdir hello-world
```

作ったフォルダに移動する:

```
$ cd hello-world
```

プロジェクトを初期化して新しいGitインスタンスを作る:

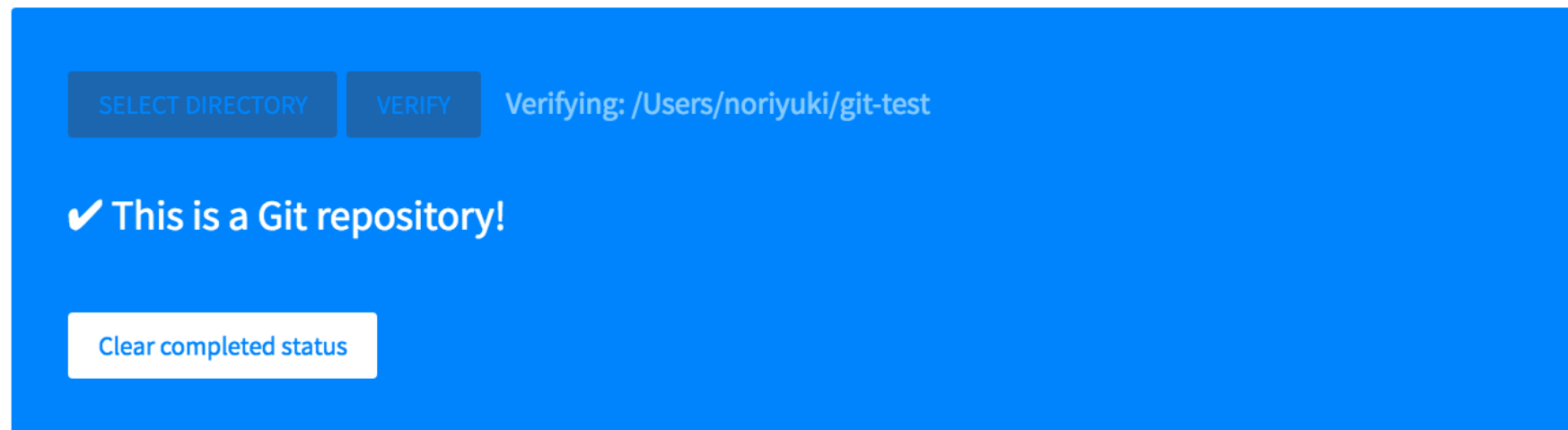
```
$ git init
```

以上です！ コマンド実行の結果はたった一行返ってくるだけです。もしGitリポジトリとしてきちんとできているかどうか再確認したいのであれば、`git status` と入力してみてください。結果、'fatal: Not a git repository...!'と返ってこなければ、成功です。おめでとうございます！

[SELECT DIRECTORY](#)[VERIFY](#)

リポジトリ作成

- うまくできていれば次のような表示が出ます
 - 右下のCommit to itへ進んで下さい



プログラムの作成

- 少しGit-itを離れます
- git initしたディレクトリにターミナルで入って下さい
 - と言わなくても多分そのディレクトリに居るはずですが
- vimでもemacsでも何でも構いませんので、hello.cというファイルを開いて下さい
 - vim hello.c とか emacs hello.c とか

C言語の最小(?)のプログラム

- テキスト (Springs of C) の8ページを開いて下さい

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // この部分に命令を並べる(ほかは毎回同じ)
    return(0);
}
```

と書いてあるはずです

- このコードをhello.cに打ち込んで下さい

とりあえずやってみましょう

- hello.cを保存したら次のコマンドを実行して下さい
 - Linux/Macの場合 `gcc -Wall -o hello hello.c`
 - Windows(Cygwin)の場合 `gcc -Wall -o hello.exe hello.c`
- 何か文句を言われた人は手を挙げて下さい
- 通常、何も問題なければ何も表示されません
 - 次のプロンプトに移るだけです(以下のコマンドを実行してみてください)
 - Linux/Macの場合 `./hello`
 - Windows(Cygwin)の場合 `./hello.exe`
- エラーは残念ながら英語で出る可能性が高いですが、読めますので読んで下さい！

少し解説

- 最小のC言語

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // この部分に命令を並べる(ほかは毎回同じ)
    return(0);
}
```

- C言語には必ずmain文(main関数)が必要です
 - Pythonでは不要でした(作れますが)
- int argc とか char **argv は、main関数の引数です
 - これに関してはいずれ詳しく話します(整数型の変数と文字列型の変数が入る、とだけ覚えておいてもらえばOKです)

中括弧 { }

- C言語の関数などは中括弧で括られます

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // この部分に命令を並べる(ほかは毎回同じ)
    return(0);
}
```

- 括弧開き { から括弧閉じ } まだがひとまとまりになります
 - Pythonの場合、制御しているブロックはインデントで表現された

return と main関数の型

- 型については詳しく後日説明します

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // この部分に命令を並べる(ほかは毎回同じ)
    return(0);
}
```

- main関数は必ず **int(整数)** 型の値を返す
 - 返す、というのが return(0) の仕事
 - この場合、「0」という値が返る(どこに返るかは後の講釈)

セミコロン ;

- C言語では文の末尾にセミコロン ; が必要

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // この部分に命令を並べる(ほかは毎回同じ)
    return(0);
}
```

- セミコロンを見つけると命令文が確定する
 - Pythonでは改行とインデントで文を区別していた

コメント

- // の後ろに書かれているものはコメントとして扱われる

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // この部分に命令を並べる(ほかは毎回同じ)
    return(0);
}
```

- コンパイル作業ではここは読み飛ばされる
 - Pythonでは # がコメントの役割

ヘッダファイル stdio.h

- 標準入出力関数を使うためのヘッダ **stdio.h**

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // この部分に命令を並べる(ほかは毎回同じ)
    return(0);
}
```

- #include <stdio.h> で、標準入出力関数が使えるようになる
 - Pythonで言うところのprint文などが使えるようになる
 - Cクイックリファレンスの索引からstdio.hに関するページを調べて読んでみて下さい

C言語の書き方(あまりよろしくない)

- Pythonのように行やインデントでゴチャゴチャ言われない

少しだけいじりましたが、上と下は全く同じプログラムです

```
#include <stdio.h> int main(int argc, char **argv){/*この部分に命令を並べる(ほかは毎回同じ)*/ return(0);}
```

```
{  
    // この部分に命令を並べる(ほかは毎回同じ)  
    return(0);  
}
```

とりあえずやってみましょう(再掲)

- hello.cを保存したら次のコマンドを実行して下さい
 - Linux/Macの場合 `gcc -Wall -o hello hello.c`
 - Windows(Cygwin)の場合 `gcc -Wall -o hello.exe hello.c`
- 何か文句を言われた人は手を挙げて下さい
- 通常、何も問題なければ何も表示されません
 - 次のプロンプトに移るだけです(以下のコマンドを実行してみてください)
 - Linux/Macの場合 `./hello`
 - Windows(Cygwin)の場合 `./hello.exe`
- エラーは残念ながら英語で出る可能性が高いですが、読めますので読んで下さい！

コンパイルという作業

- Pythonの場合、インタラクティブモードでやっているを入力するたびに命令が実行されていたり、.pyファイルに書いた場合でも、書き終わったら `python .py` とやるだけで動いていた
- 一方、C言語だと
 `gcc -Wall -o hello hello.c`
 みたいな謎のコマンドを打って
 `./hello`
 とやらないと実行できない

コンパイルという作業

- みなさんに書いてもらったhello.cというファイルは「ソースコード」とか「ソースプログラム」と呼ばれます
- ソースは、一見すると人間に読みやすい(かどうかは人によるが…)言語で書かれているが、コンピュータは 0 or 1 しか分らない
 - 人間に分かりやすい言語から、コンピュータに分かりやすい 0 or 1 の並びに変換する作業が「コンパイル」という作業
- Pythonはインタプリタなので、書かれたことをそのまま実行していき必要に応じて勝手に実行してくれる
 - その反面、動作はC言語に比べるとはるかに遅い

少し解説

- gcc -Wall -o hello hello.c では何をやっているか？
- gcc
 - GNU C Compilerとか何とかの略で、C言語のコンパイラ
- -Wall
 - -Wが警告を表示するオプションで、allが引っ付いてあらゆる警告を出すという指定
- -o hello
 - -oは実行ファイルを直後の名前で作れ、という指定(この場合helloという実行ファイル)
- hello.c
 - コンパイル対象になるファイル名(C言語のソースコードは必ず **.c** を付与)

少し解説

- gcc hello.c とやるとどうなるか？
 - a.outという実行ファイルができます(./a.out で実行できる)
- gcc -Wall hello.c とやるとどうなるか？
 - a.outという実行ファイルができて、適宜警告を出してくれる
- gcc -o hello hello.cとやるとどうなるか？
 - helloという実行ファイルができます(./hello で実行できる)

Git-itに戻ります

- git statusとやるとhello.cが作成されたことがわかります
- その後、git add hello.cとやるとcommitできるようになります
 - コメントは、英語で適当に…

Step: Statusを確認し、変更をAddしてCommitしてみよう

次にリポジトリの**status**を使って、ファイルに変更が加わっていないかどうかチェックしてみましょう。ターミナルを使って、'hello-world'フォルダの中で次のコマンドを入力してみましょう。ファイルの変更が表示されると思います。:

```
$ git status
```

次に今作ったファイルを**add** して、変更を**commit** (保存とほとんど同じ意味です) できるようにします。

```
$ git add readme.txt
```

最後に、これらの変更を短い説明文 (commit message) と一緒に**commit**してリポジトリの履歴として保存しましょう。

```
$ git commit -m "<your commit message>"
```

Step: さらに変更してみよう

さらに別の行を一行、`readme.txt` に追加して保存してみましょう。

ターミナルの中で、最後にcommitした内容といまのファイルの内容との差分を確認することができます。これを**diff**といいます。

```
$ git diff
```

さて、そうしたら、先ほど学んだように、この変更もcommitしましょう。

SELECT DIRECTORY

VERIFY

Git-itに変更をcommit

- まずhello.cを次のように書き換えましょう

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello, World. ¥n");
    return(0);
}
```

- Windows系だとバックスラッシュが¥マークになってしまいますが私のスライドで出てくる¥マークはバックスラッシュのことです
 - このあたり、PowerPointで作っている最大の弊害ですね…

同様にcommit

- git diffとやるとhello.cが変更されたことがわかります
- その後、git add hello.cとやるとcommitできるようになります
 - コメントは、違う英語で適当に…

Step: Statusを確認し、変更をAddしてCommitしてみよう

次にリポジトリの**status**を使って、ファイルに変更が加わっていないかどうかチェックしてみましょう。ターミナルを使って、'hello-world'フォルダの中で次のコマンドを入力してみましょう。ファイルの変更が表示されると思います。:

```
$ git status
```

次に今作ったファイルを**add** して、変更を**commit** (保存とほとんど同じ意味です) できるようにします。

```
$ git add readme.txt
```

最後に、これらの変更を短い説明文 (commit message) と一緒に**commit**してリポジトリの履歴として保存しましょう。

```
$ git commit -m "<your commit message>"
```

Step: さらに変更してみよう

さらに別の行を一行、`readme.txt` に追加して保存してみましょう。

ターミナルの中で、最後にcommitした内容といまのファイルの内容との差分を確認することができます。これを**diff**といいます。

```
$ git diff
```

さて、そうしたら、先ほど学んだように、この変更もcommitしましょう。

SELECT DIRECTORY

VERIFY

確認しましょう

- Verifyに成功すると次のようになります
 - GitHubbinに進みましょう

