岐阜工業高等専門学校

第 2 回課題

電子制御工学科: 情報処理 I

担当教員：岡崎 憲一

柴田 健琉 (15 (2 年生))

令和 7 年 04 月 17 日

Made with LaTeX + vim

目次

1　はじめに

1.1　実行環境

この課題のプログラムは以下の環境で動作することが確認されている：

- OS: Arch Linux
- CPU アーキテクチャ: x86_64
- C コンパイラ: gcc（GCC） 14.2.1 20250207
- C コンパイラオプション: -Wall <ソースコード名> -o <プログラム名>[1)]

2　今回の構文

2.1　#include マクロ

コンパイラはソースコードをコンパイルする前にプリプロセッサを実行し、ファイル内に定義されたマクロを展開する。マクロは普通の文と違い、「;」(セミコロン) が文末に無く、マクロ名の先頭に「#」(ハッシュ記号) が付く。#include はマクロの 1 つであり、主に関数・変数・定数・型などの宣言が羅列されているヘッダーファイルの内容をマクロが書かれた場所に展開するもので、大抵の場合ソースコードの最上部に書かれている。[7]

よく使用されるヘッダーファイルの一部には以下の物がある [2]：

- stdlib.h：標準ライブラリのこと、メモリ管理や頻出アルゴリズムなどに関する関数などが宣言・定義されている
- stdio.h：標準入出力のこと、ファイル操作などに関する関数が宣言・定義されている
- string.h：文字列操作によく使用される関数を宣言・定義されている
- wchar.h：日本語のようなマルチバイトの文字を扱うための型や関数が宣言・定義されている
- math.h：三角関数や対数関数などの数学関数が宣言・定義されている

#include マクロ

```
#include <ヘッダーファイル.h> // 1) コンパイラ・システムが指定した
                              // ディレクトリ(フォルダパス)内でファイルを
                              // 探す。
                              // Linuxならば/usr/includeの場合が多い。
                              // また、GCCでは引数に "-I <ディレクトリ>" を
                              // 渡すことで探索パスを実行時に追加できる。[6]

#include "ヘッダーファイル.h" // 2) 書かれているソースコードと
                              // 同じディレクトリの中でファイルを探す。
                              // 見つからなければ 1) と同じ挙動を取る。
```

1)　-Wall はコンパイル時に「凡ミス」や書式に関する全ての警告を表示するフラグである。[6]

2.2 main 関数

プログラムが呼び出される際に最初に実行される関数。この関数の戻り値は整数型であるが、これはプログラムの終了が正常であるかどうかを示めすものであり、Linux では 0 が正常終了、1 が異常終了などである。この異常終了判定に使用される値は OS によって異なる。なので stdlib.h で提供される exit( ) 関数や EXIT_SUCCESS, EXIT_FAILURE 定数を使うのが望ましい。[4]

main 関数

```c
int main(void) {
    // プログラム実行時に何も引数を渡す必要がない場合

    // 文

    return 0;
}

int main(int argc, char** argv) {
    // プログラム実行時に引数を渡す場合、argc は呼び出しプログラム名を含む引数の数、
    // argv は呼び出しプログラム名を含む引数の値(文字列型)の配列である。

    // 文

    return 0;
}
```

2.3 printf 関数

printf とは Print Format のことで、第一引数に書式 (文字列)、第二以降の引数に表示したいデータ (変数、定数、リテラル) を羅列する。この関数は stdio.h を include することで利用できる。[5]

printf 関数

```c
printf("<書式>", <データ1>, <データ2>, ...);
```

書式には以下のような物がある：

printf の書式 (一部)[1]

```c
"%d" // 整数
"%f" // 浮動小数点
"%c" // 文字(単一)
"%s" // 文字列
```

更に、書式には文字の表示を制御できる特殊文字 (escape sequence) があり、一部の文字はソースコードと干渉するためそれらを使用しないと表示できない。特殊文字はバックスラッシュで始まり、ASCII 文字 1 つが続く。

特殊文字 (一部)[1]

```
"\n" // 改行
"\\" // バックスラッシュ
"\?" // 疑問符
"\"" // ダブルクウォーテーションマーク
"\'" // シングルクウォーテーションマーク
```

2.4　変数宣言・定義

変数はコンピュータのメモリ上にある値が入る箱のような物である。この箱には整数や小数値、文字などが入るが、メモリ上ではすべて 1 と 0 で表現されている。この一次元な 1 と 0 の海の中でどうやってプログラムが値の種類を決定しているのかというと、それはコンパイラが機械語に変換する際に変数名に添えられた型から読み出す時の必要な 1 と 0 の列の長さを実行ファイルに書き込んでいるのだ。

C 言語で変数を扱うには二つの操作が必要である。1 つは変数の宣言、もう 1 つは変数の定義である。変数の宣言とは文字通り変数の存在を宣言することである。具体的にはその変数を格納する場所をメモリ上に作ることである。そして変数の定義は値の代入と言い替えることができ、宣言で作られた場所に値を書き込む操作である。

ソースコードでは次のようになる:

変数の例

```
int a;  // 整数型の変数の宣言
a = 17; // 変数名 a に 17 という値を書き込む
```

上記の書き方は C99 などの標準規格が制定される前のソースコードによく見られるが、C99 以降からは 1 行で宣言と代入が出来るようになっている。[3]

C99 以降の変数宣言・代入

```
int a = 17; // 整数型の変数 a の宣言と同時に 17 という値を書き込む
```

型には一部として以下の物がある [1]：

- `int`：整数型
- `char`：文字型
- `float`：32 ビット単精度浮遊少数型
- `double`：64 ビット倍精度浮遊少数型

なお、一部の型が占有するバイト数はコンパイラや CPU によって異なるが、`x86_64` の場合は `int` 型が 4 バイト、`char` 型が 1 バイトである。

3　演習課題 1

出力に「C 言語」と表示するプログラム

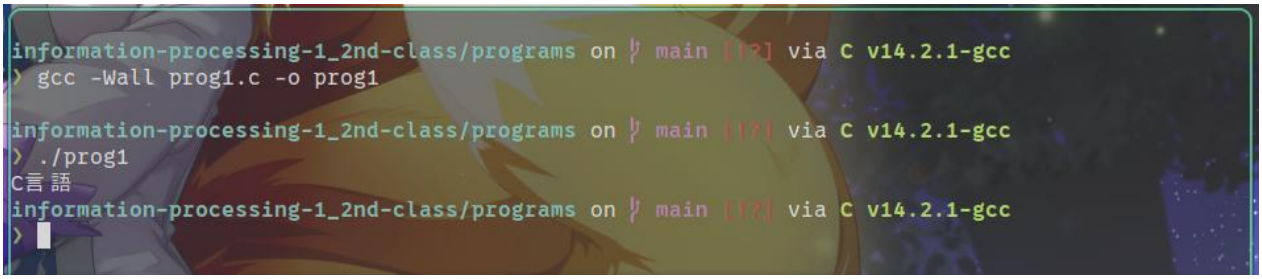
3.1　コードリスティング

演習課題 1

```c
#include <stdio.h>

int main(void) {
    printf("C言語");
    return 0;
}
```

3.2　実行結果



4　演習課題 2

出力に「C 言語」を縦書きで表示するプログラム

4.1　コードリスティング

演習課題 2

```c
#include <stdio.h>

int main(void) {
    printf("C\n言\n語\n");

    return 0;
}
```

4.2　実行結果

5　演習課題 3

出力に「情報処理」を右に向かって斜めに表示するプログラム

5.1　コードリスティング

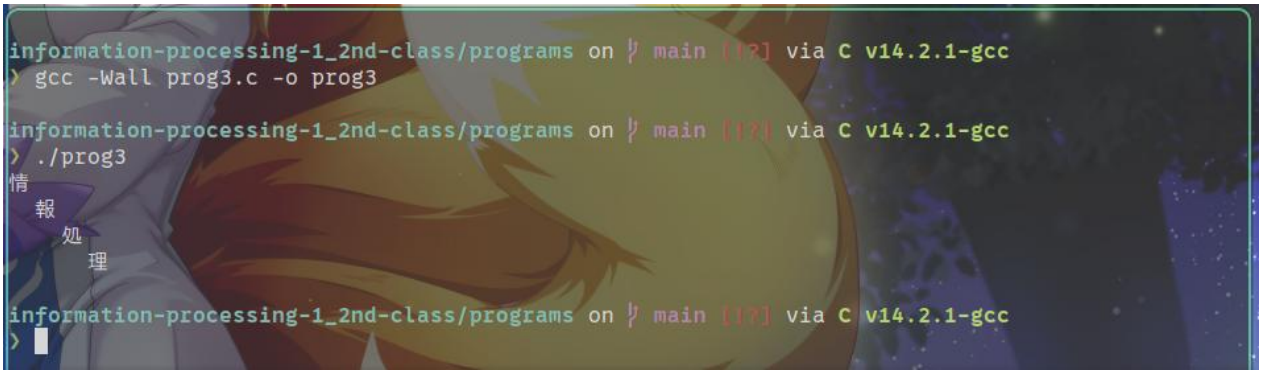
演習課題 3

```c
#include <stdio.h>

int main(void) {
    printf("情\n  報\n    処\n      理\n");

    return 0;
}
```

5.2　実行結果



6　演習課題 4

3 つの int 型整数 y,m,d を宣言し，y は誕生年，m は誕生月，d は誕生日で初期化し，y+m+d の値を「〇〇〇〇の生年月日の和は＊＊＊＊です.」と表示するプログラム

6.1　コードリスティング

演習課題 4

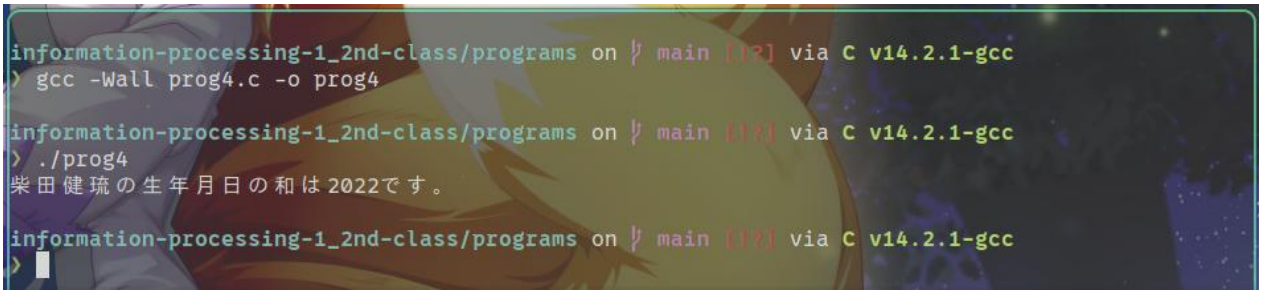
```c
#include <stdio.h>

int main(void) {
    int y = 2007;
    int m = 8;
    int d = 7;

    printf("柴田健琉の生年月日の和は%dです。\n", y+m+d);

    return 0;
}
```

6.2　実行結果



A　付録

A.1　Deep Dive - 変数の舞台裏

2.4 節にてプログラムが値の種類を判別するためにコンパイラが型情報に基づいて読み出すべきビット長を実行ファイルに書き込まれると書いたが、実際はどうであろう。ここではコンパイラの生成物である実行ファイルをアセンブリに分解して検証していく。

まずは検証用のソースコードを用意する。今回は以下の物を用意した：

検証用コード

```
1 void main(void) {
2     int a;
3     a = 255;
4
5     return;
6 }
```

なお、実行環境は 1.1 節のものとする。

このコードをコンパイルするにあたって、解読を容易にするために少しフラグを変更する必要がある。Linux での `gcc` コンパイラは特に指定されていなければコンパイラの機能を使用するための `libgcc` と `libc` という C 言語の標準的な関数 (`printf`, etc.) などが宣言・定義されたライブラリをリンクします。これによりソフトウェアの日常的な機能の再開発を避けれるが、人間が読めるアセンブリに直すとその部分のコードが析出してしまい、目的のコードが埋もれてしまう。なので、これらライブラリの自動リンクをやめる必要がある。そこで今回は`-nostdlib` フラグと`-nolibc` フラグを追加する。[6]

最近のコンパイラは非常に賢く、無意味なコードを取り除いて最適化しようとする。だが今回の検証用コードはただ単に変数を宣言・定義するだけで、実際に使用されないのでそのままではコンパイラに無意味と見なされ、勝手に変数の宣言・定義のコードを削除してしまう。なのでここにコンパイラによる最適化を無効にするために`-O0` フラグも追加する。[6]

最終的なコンパイルコマンドはこのようになる：

```
gcc -Wall -nostdlib -nolibc -O0 demo.c -o demo
```

上記のコマンドを実行するとリンカからエントリーポイント：プログラムの始まりが定義されていないとエラーを吐くが解読に問題はないのでそのままでよい。

次に生成物である `demo` をデコンパイル (実行ファイルからアセンブリに戻す作業) を行うために GNU Binutils[2] の `objdump` を利用する。

2） https://www.gnu.org/software/binutils/

以下がデコンパイルコマンドとなる：

```
objdump -Mintel -d demo
```

このコマンドの-Mintel フラグはデコンパイル時にアセンブリを私が個人的に読みやすい Intel 記法で表記すると objdump に命令できる。

上記のコマンドを実行すると次のような出力になる：

```
demo-x86_64:     ファイル形式 elf64-x86-64

セクション .text の逆アセンブル:

0000000000001000 <main>:
    1000:       55                      push   rbp
    1001:       48 89 e5                mov    rbp,rsp
    1004:       c7 45 fc ff 00 00 00    mov    DWORD PTR [rbp-0x4],0xff
    100b:       90                      nop
    100c:       5d                      pop    rbp
    100d:       c3                      ret
```

これで demo のデコンパイル結果が出た。[3)]

このアセンブリの口語訳は以下となる：

- rbp レジスタ (CPU 内にあるごく小さな変数) の内容をスタック (メモリ上にある最初に入れたデータは最後に出る仕組みを持つデータ保持の構造) の最上部に積む。
- rsp レジスタの内容を rbp レジスタに書き込む。
- 16 進数値 0xff(10 進数で 255) を rbp レジスタの内容から 0x4 を引いたアドレス (メモリ上の場所を表す住所のような数値) が指している場所から DWORD(4 バイト) 分の領域に書き込む。
- なにもしない。
- スタックの最上部にある値を取り、それを rbp レジスタに読み込む。
- 呼び出し元の関数に制御を戻す。

rbp レジスタは主に関数内でのみ有効なローカル変数を格納するレジスタである。また、rsp レジスタはスタックポインタといい、現在実行されている関数に関するデータの読み書きに使用される。

今回の例では、まず rbp レジスタの内容をスタックに退避させ、現在のスタックポインタが指しているアドレスを rbp レジスタにコピーする。次に、rbp が指しているアドレスから 4 つ分移動させ、スタックの 4 バイト分の領域を確保する。1 バイトは 8 ビットであるため $4 * 8 = 32$ ビット分の空を確保している。これはまさに宣言そのもので、int 型は 32 ビットのサイズを持つのでちゃんと当てはまる。更にこの空いた場所に mov 命令を使用して 16 進数値 0xff、10 進数の 255 を移動させている。これが定義であり、宣言した場所に値を代入するという C 言語の a = 255; と一致する。

x86_64 の様な CISC(複雑命令セットコンピュータ) は宣言と定義を一つの命令で行っている。

3)　コンパイル時に-nostdlib -nolibc を指定しないとデコンパイル結果が 100 行以上になる。

では、小さく単純な命令セットを持つ RISC(縮小命令セットコンピュータ) ではどうだろうか。
コンパイラを RISC-V 64 ビットアーキテクチャ用のツールチェーンに変更して再度検証してみる。
まずは、以下のコマンドでコンパイルする：

```
riscv64-none-elf-gcc-14.2.1 -Wall -nostdlib -nolibc -O0 demo.c -o demo-riscv
```

そして以下のコマンドでデコンパイルする：

```
riscv64-none-elf-objdump -d demo-riscv
```

デコンパイル結果は以下の通りである：

```
demo-riscv:     ファイル形式 elf64-littleriscv

セクション .text の逆アセンブル:

00000000000100e8 <main>:
   100e8: 1101                addi    sp,sp,-32
   100ea: ec06                sd      ra,24(sp)
   100ec: e822                sd      s0,16(sp)
   100ee: 1000                addi    s0,sp,32
   100f0: 0ff00793            li      a5,255
   100f4: fef42623            sw      a5,-20(s0)
   100f8: 0001                nop
   100fa: 60e2                ld      ra,24(sp)
   100fc: 6442                ld      s0,16(sp)
   100fe: 6105                addi    sp,sp,32
   10100: 8082                ret
```

このアセンブリの口語訳は以下となる [8]：

- sp(スタックポインタ) レジスタに sp レジスタの値 + (−32) の加算結果を代入する。
- ra レジスタから sp レジスタが示めしているアドレス + 24 のアドレスが指している場所に 64 ビット値をコピーする。
- s0 レジスタから sp レジスタが示めしているアドレス + 16 のアドレスが指している場所に 64 ビット値をコピーする。
- s0(スタックポインタ) レジスタに sp レジスタの値 + 32 の加算結果を代入する。
- a5 レジスタに 10 進数値 255 を代入する。
- a5 レジスタから sp レジスタが示めしているアドレス + (−20) のアドレスが指している場所に 32 ビット値をコピーする。
- なにもしない
- sp レジスタが示めしているアドレス + 24 のアドレスが指している場所から ra レジスタに 64 ビット値をロードする。

- sp レジスタが示めしているアドレス + 16 のアドレスが指している場所から s0 レジスタに 64 ビット値をロードする。
- sp(スタックポインタ) レジスタに sp レジスタの値 + 32 の加算結果を代入する。
- 呼び出し元の関数に制御を戻す。

やはり、命令の種類が少ない RISC では命令の数が多くなっている。だが最初の 4 つと最後の 4 つは関数のスタックフレームに関する命令で無視してよい。見るべきものは `100f0` と `100f4` である。
そこでは 32 ビットの値 255 を a5 レジスタに記憶し、スタックにコピーしている。ここでは 64 ビット長のレジスタに 32 ビット値を代入することで変数の宣言と定義を同時にしている。しかしレジスタは一時的なデータの保持に使用されるので関数の寿命の間、いつでも参照できるようにスタックに移動させているのである。x86_64 との違いは読み出すバイト数が命令引数のキーワードとして明示されているか否かで、RISC-V では読み出すサイズによって命令が分けられている (eq. SB: 1 バイト、SH: 2 バイト、SW: 4 バイト)[8]。

比べてみると、CISC と RISC ではやることは同じである：スタックに変数のサイズ分の場所を作り、値を入れる。

以上より、2.4 節で示した値の種類の判別にビット長を実行ファイルに埋め込むということがらが実証された。

参考文献

[1]Adam Bard et al. *Learn C in Y Minutes*. Japanese. Trans. by 柴 田 健 琉. 03/2025. URL: https://learnxinyminutes.com/ja/c/ (visited on 04/17/2025).

[2]Bazzy et al. *C Standard Library headers*. 02/2025. URL: https://en.cppreference.com/w/c/header (visited on 04/18/2025).

[3]Cubbi et al. *Declarations*. 01/2025. URL: https://en.cppreference.com/w/c/language/declarations (visited on 04/17/2025).

[4]Cubbi et al. *Main function*. 07/2023. URL: https://en.cppreference.com/w/c/language/main_function (visited on 04/18/2025).

[5]Eendy et al. *printf, fprintf, sprintf, snprintf, printf_s, fprintf_s, sprintf_s, snprintf_s*. 01/2025. URL: https://en.cppreference.com/w/c/io/fprintf (visited on 04/17/2025).

[6]Contributors of GCC. *Top (Using the GNU Compiler Collection(GCC))*. Inc., Free Software Foundation. 07/2024. URL: https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/ (visited on 04/17/2025).

[7]P12 et al. *Source file inclusion*. 09/2023. URL: https://en.cppreference.com/w/c/preprocessor/include (visited on 04/18/2025).

[8]SHAKTI Development Team. *RISC-V ASSEMBLY LANGUAGE Programmer Manual Part I*. 01/14/2021, pp. 12–13, 29–37, 61–62, 76, 80–81. URL: https://shakti.org.in/docs/risc-v-asm-manual.pdf.

Last Compiled(UN*X Time): 1745241536