



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

目次

1	はじめに	1
1.1	実行環境	1
2	今回の構文	1
2.1	printf 関数	1
2.2	変数宣言・定義	2
3	演習課題 1	2
3.1	コードリスティング	2
3.2	実行結果	3
4	演習課題 2	3
4.1	コードリスティング	3
4.2	実行結果	3
5	演習課題 3	3
5.1	コードリスティング	4
5.2	実行結果	4
6	演習課題 4	4
6.1	コードリスティング	4
6.2	実行結果	5
A	付録	6
A.1	Deep Dive - 変数の舞台裏	6

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 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` (ソースコードに含める) ことで利用できる。[3]

printf 関数

```
1 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]

```
1 "%d" // 整数
2 "%f" // 浮動小数点
3 "%c" // 文字 (単一)
4 "%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]

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

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。

2.2 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

```
1 int a; // 整数型の変数の宣言
2 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。[2]

C99 以降の変数宣言・代入

```
1 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
```

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

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("C言語");
5     return 0;
6 }
```

3.2 実行結果

```
information-processing-1_2nd-class/programs on ㇿ main [!?] via C v14.2.1-gcc
> gcc -Wall prog1.c -o prog1

information-processing-1_2nd-class/programs on ㇿ main [!?] via C v14.2.1-gcc
> ./prog1
C言語
information-processing-1_2nd-class/programs on ㇿ main [!?] via C v14.2.1-gcc
> 
```

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

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("C\n");
5     printf("言\n");
6     printf("語\n");
7
8     return 0;
9 }
```

4.2 実行結果

```
information-processing-1_2nd-class/programs on ㇿ main [!?] via C v14.2.1-gcc
> gcc -Wall prog2.c -o prog2

information-processing-1_2nd-class/programs on ㇿ main [!?] via C v14.2.1-gcc
> ./prog2
C
言
語
information-processing-1_2nd-class/programs on ㇿ main [!?] via C v14.2.1-gcc
> 
```

5 演習課題 3

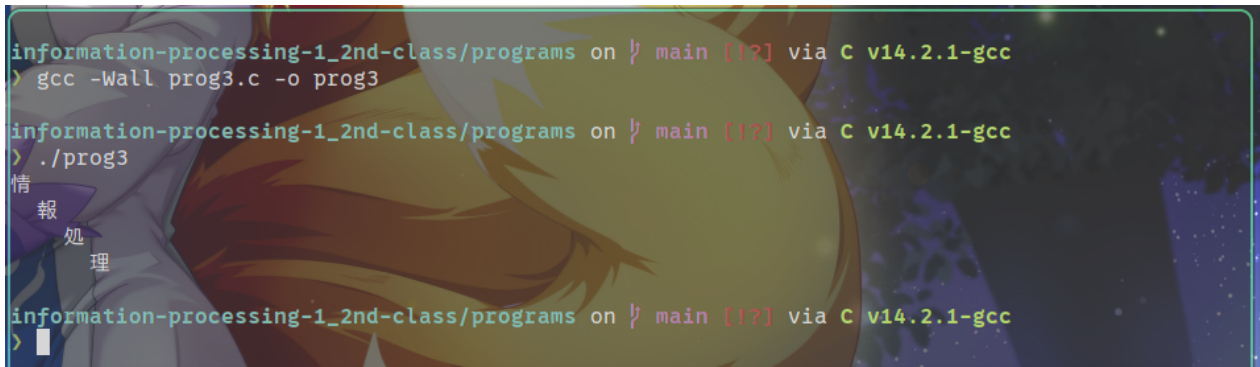
出力に「情報処理」を右に向かって斜めに表示するプログラム

5.1 コードリスティング

演習課題 3

```
1  #include <stdio.h>
2
3  int main(void) {
4      printf("情\n");
5      printf("  報\n");
6      printf("    処\n");
7      printf("      理\n");
8
9      return 0;
10 }
```

5.2 実行結果



```
information-processing-1_2nd-class/programs on 主 main [!?] via C v14.2.1-gcc
> gcc -Wall prog3.c -o prog3
information-processing-1_2nd-class/programs on 主 main [!?] via C v14.2.1-gcc
> ./prog3
情
報
処
理
information-processing-1_2nd-class/programs on 主 main [!?] via C v14.2.1-gcc
> 
```

6 演習課題 4

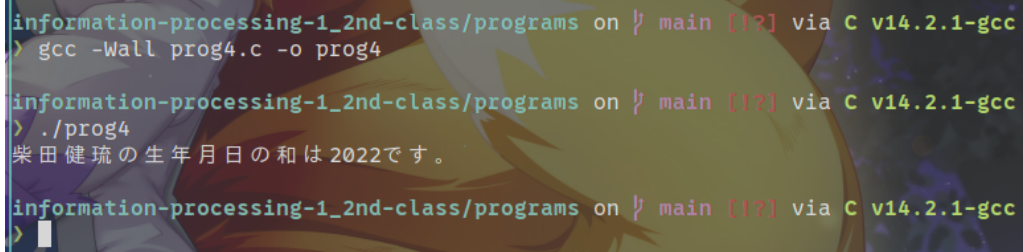
3つの int 型整数 y,m,d を宣言し, y は誕生年, m は誕生月, d は誕生日で初期化し, y+m+d の値を「○○○○の生年月日の和は****です.」と表示するプログラム

6.1 コードリスティング

演習課題 4

```
1  #include <stdio.h>
2
3  int main(void) {
4      int y = 2007;
5      int m = 8;
6      int d = 7;
7
8      printf("柴田健琉の生年月日の和は%dです. \n", y+m+d);
9
10     return 0;
11 }
```

6.2 実行結果



```
information-processing-1_2nd-class/programs on 主 menu [?] via C v14.2.1-gcc
> gcc -Wall prog4.c -o prog4

information-processing-1_2nd-class/programs on 主 menu [?] via C v14.2.1-gcc
> ./prog4
柴田 健琉の生年月日の和は2022です。

information-processing-1_2nd-class/programs on 主 menu [?] via C v14.2.1-gcc
> 
```

A 付録

A.1 Deep Dive - 変数の舞台裏

2.2 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。なお、この付録を読むのに特別な知識は必要なく、必要に応じて説明する。

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、解説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。[4]

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も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。[4]

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説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 Binutils²⁾ 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 -Intel -d demo
```

このコマンドの `-Intel` 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:      ファイル形式 elf64-x86-64
```

セクション `.text` の逆アセンブル：

```
0000000000401000 <main>:
```

```
401000: 55                push    rbp
401001: 48 89 e5          mov     rbp, rsp
401004: c7 45 fc ff 00 00 00 mov     DWORD PTR [rbp-0x4], 0xff
40100b: 90                nop
40100c: 5d                pop     rbp
40100d: 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) が指している場所へ書き込む。
- 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 つ分移動させる。この時の 4 は単位が 1 バイトであるため $4 \times 8 = 32$ 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 `movl` 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。

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。

以上より、[2.2](#) 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。

References

- [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]Cubbi et al. *Declarations*. 01/2025. URL: <https://en.cppreference.com/w/c/language/declarations> (visited on 04/17/2025).
- [3]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).
- [4]Contributors of GCC. *Top (Using the GNU Compiler Collection(GCC))*. 07/2024. URL: <https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/> (visited on 04/17/2025).

Last Compiled(UNIX Time): 1744901578