デバッグの手間を減らす: “printf デバッグ” と GDB の使い方

小野孝男

2010 年 10 月 25 日

リスト 1 のプログラム target.c を使い, デバッグ手法としての “printf デバッグ” と “デバッガ GDB の簡単な使い方” を説明する.

リスト 1　対象とするプログラム target.c

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  void sub1(char *str, int i)
5  {
6      str[i] = toupper(str[i]);
7      printf("%s\n", str);
8  }
9
10 int main()
11 {
12     char str[] = "hello";
13
14     sub1(str, 2); /* This works well */
15
16     sub1(0, 1); /* Oops */
17
18     return 0;
19 }
```

1　まずコンパイルと実行

このプログラムを普通にコンパイルして実行すると

```
> gcc -o target target.c
> ./target
heLlo
Segmentation fault
```

となる. ここで, 最後に “Segmentation fault” とあるように, このプログラムにはバグがある. どこがおかしいかを調べるため “printf デバッグ” を行ったり GDB を使ったりする.

2　printf デバッグ

printf デバッグとは, “プログラム中に printf を埋め込んで表示させる手法” である. つまり, 実行されているかどうかあやしいところに printf を入れておき, どこまで表示されるかを見ることによって “停止した

ところ” を見付ける手法である. また, 同時に “あやしい変数” の値を表示させるころもできる.

今の例では 13 行目や 15 行目に printf を入れればきちんと表示されるが, 17 行目に入れると printf は実行されない. つまり “16 行目があやしい” とあたりをつけることができる.

なお, printf デバッグでは確実に出力させるため

```
printf(表示内容); fflush(stdout);
```

のように関数 fflush を実行するか, あるいは

```
fprintf(stderr, 表示内容);
```

のように stderr に出力する.

3 GDB によるデバッグ

3.1 GDB でデバッグするためにコンパイル

GDB でデバッグするためには -g オプションを付けてコンパイルする. このオプションはプログラムの動作とは無関係なので, もちろんこのオプションを付けてコンパイルしてもエラーは出る:

```
> gcc -g -o target target.c
> ./target
heLlo
Segmentation fault
```

3.2 GDB の起動

ここで, デバッグ対象となるプログラムを指定して GDB を起動する:

```
> gdb ./target
(ライセンス文章など)
(gdb)
```

(gdb) が GDB のプロンプトである. この状態で GDB にコマンドを入力することができる.

3.3 GDB 管理下でプログラムを実行: run

さて, GDB の管理下でプログラムを実行するために **run** コマンドを実行する:

```
(gdb) run
Starting program: /home/onotakao/debug/target
heLlo

Program received signal SIGSEGV, Segmentation fault.
0x080483b9 in sub1 (str=0x0, i=1) at target.c:6
6     str[i] = toupper(str[i]);
```

やはり “Segmentation fault” で停止してしまった. しかし, いくつかの情報が追加されている. つまり,

```
0x080483b9 in sub1 (str=0x0, i=1) at target.c:6
```

という行によって “プログラムがどこで停止したか” がわかる. 今の場合

- アドレス 0x080483b9 で止まった[*1]
- これは関数 sub1 の中である
- ソースファイル target.c の 6 行目で
- 関数に対する引数は第 1 引数の str が 0x0, 第 2 引数の i が 1

ということを意味する. その次の行はソースファイルの 6 行目をそのまま表示している.

なお, run を実行するときに

```
(gdb) run arg1 arg2
```

のように引数を与えれば, デバッグしたいプログラムに引数を渡すことができる.

3.4 値の表示: print

現在のスタックフレーム (関数) にある変数の値は **print** コマンドで表示できる:

```
(gdb) print str
$1 = 0x0
(gdb) print i
$2 = 1
```

これは, 変数 str の値が 0x0, i の値が 1 であることを示している.

なお, print で表示する値は変数だけでなく, 関数を呼び出さなければ C で有効な式を使うことができる. つまり, 配列に対して添字付けをしたり, メンバーアクセス演算子の . や -> で構造体のメンバーを調べることもできる.

3.5 関数の実行履歴の表示: bt

ちょっと大きなプログラムでは, 複数の関数が呼び出されることになる. そのようなときに “関数がどのように呼び出されてきたか” を知るために **bt** コマンドを使う:

```
(gdb) bt
#0  0x080483b9 in sub1 (str=0x0, i=1) at target.c:6
#1  0x08048425 in main () at target.c:16
```

現在実行中の関数が最も上に表示され, それぞれの関数を呼び出した関数が順次表示される. 今の例だと現在実行している関数は sub1 で, target.c の 6 行目にいること, この関数は main の中, target.c の 16 行目

[*1] この値は知ってもあまり役に立たないので無視してもいい.

で呼び出されていることがわかる.

3.6 スタックフレームの移動: up, down

C では, “関数にローカルな変数” をその関数以外で見ることができない. そこで, 別の関数で定義された変数の値を調べるときには **up**, **down** といったコマンドでスタックフレームを移動する:

```
(gdb) up
#1  0x08048425 in main () at target.c:16
16     sub1(0, 1);
(gdb) down
#0  0x080483b9 in sub1 (str=0x0, i=1) at target.c:6
6     str[i] = toupper(str[i]);
```

up で呼出元の関数に, down で呼出先の関数に移動することができる.

3.7 GDB の終了: quit

GDB を終了するときには **quit** コマンドを使う:

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
>
```

実行中のプログラムが存在するときには “本当に終了してよいか” どうかを聞いてくる. y を入力するとデバッガは終了する.

4 終わりに

いずれにしても, デバッグ手法に王道は存在しない. printf デバッグや GDB によるデバッグによっても見付からない (あるいは見付けにくい) バグというのはありえる. 最初に挙げた target.c でも 12 行目が

```
12 char *str = "hello";
```

となっていたら簡単ではない (**文字列リテラルを変更してはいけない**ことを意識していないと難しい). また, 特にポインタまわりのバグでは “printf を入れると動作する” とか “GDB ではちゃんと動く” など, デバッグが困難な場合もある.

プログラム作成におけるデバッグのコストは (それそのものは何も生み出さないにもかかわらず) 非常に大きなものとなっている. 従って, **バグがなるべく入らないように**, また**入ったとしてもバグが見付けやすいように**プログラムを作成することが重要である.