第11回

標準的デバッガGDBのすべて

岸 哲夫

内部データの調査とデバッグ

前回の続きで，今回も内部のデータが現在どのような値になっているかを調べる方法について説明する．統合環境の開発ソフトウェアは便利かもしれないが，それ以外使えなくなってしまうという弊害がある．GCC と GDB であれば，たいていの環境で動作するうえに，使い勝手もよいと思う．（筆者）

今回も引き続き GDB を使ったデバッグについて解説します．

プログラマは自分の犯したまちがいには気付きにくい

だれでも自分が犯したまちがいには気が付きにくいものです．これはプログラムを書くうえでの話だけではなく，普遍的な問題ですが，ここではプログラムを書く話に限定します．

プログラマはコンピュータと向き合って仕事をしているだけでは，自分の書いたコードのまちがいに気付きにくいことがあります．

● レビューが大切

そのようなときに大切なのは「レビュー」です．「この処理はこれで良いのだろうか」と他人に確認するだけで，思考が切り替わることがあります．「初期化を忘れていた」，「10 個しかない配列の 11 個目にアクセス」，「メモリ確保を忘れた！」などの基本的なミスは，他人と会話をするだけで，発見されることもあります．

● コードの検証

次に必要なことはコードの検証です．自分の書いたコードが正しいと思い込まずに，確認する作業は重要です．よく見ると，比較条件文で等しいかを確認する際に「=」を 1 個だけ使っていたりするかもしれません．

ユーザ空間で動作するプログラムは，いろいろな方法で監視することができます．GDB などのデバッガを使用し，ステップ実行し，状況を表示させるという方法もありますし，プログラムを分析するためのツールを追加するという手もあります．

- Electric Fence

メモリ・リークの検出に役立ちます．不正なアクセスをすると，core を吐き出して終了します．

- YAMD

動的なメモリ割り当てが正しいかどうかを確認できます．Malloc を使っている場合に有効です．

- MEMWATCH

メモリ・リークやメモリのデータ破壊，二重解放，まちがった解放，未解放メモリ，オーバフロー，アンダフローを検出することが可能です．

もちろん GDB も重要ですが，機会があったら記事中で上記のツールも紹介します．

● システム・エンジニアとプログラムの意思の疎通が図れていない

ある企業がバグに関するレポートを集計し，解析したところ，いちばん多い問題はシステム・エンジニアと，プログラマの意思の疎通が図れていないことだという結論が導き出されました．仕様書が大ざっぱなため，プログラマの意志で決めることが多すぎるわけです．

徹底するならば仕様書に変数の型，桁数まで規定して，プログラマに引き渡すべきでしょうが，そうもいきません．また，それではプログラマではなくコーダです．意思の疎通さえ問題なければ防げるバグも多いようです．もちろん技術的に未熟であるがゆえに発生したバグもあると思います．それは資料を読めば解決する問題です．

GDB の中にある値を参照できるコンビニエンス変数は便利

● コンビニエンス変数の使い方

さて，GDB の機能の話に戻ります．前回（2006 年 3 月号，第 10 回）で説明した「値履歴」は，print コマンドにより表示された値を，GDB の 値履歴に保存するようになっています．このように GDB の中にある値を保持しておいて，それを後で参照できる変数を，コンビニエンス変数と呼びます．

これらの変数は， 実行中の GDB 内でのみ有効です．ユーザ空間で動作しているプログラムの中に存在するものではありません．したがって，コンビニエンス変数を設定してもユーザ・プログラムの実行には直接影響を与えません．つまり，ユーザはこれを自由に使用することができます．

コンビニエンス変数名は，先頭が‘ $ ’で始まります．‘ $ ’で始まる名前はほかの名称と一致しない限り，コンビニエンス変数の名前として使用することができます．

基本的にはユーザ空間で動作しているプログラム中で変数に値を設定することと同じです．

KEYWORD ── GDB，レビュー，デバッガ，Electric Fence，YAMD，MEMWATCH，コンビニエンス変数，値履歴

リスト1　コンビニエンス変数の説明のためにグローバル変数を使用しているソース(test21.c)

```
/*
 *グローバル変数
 */
#include <stdio.h>
void    test1();
int     ix;
int     ix1  =   20;
int main(int argc, char*
argv[])
{
    ix   =   10;
    test1();
    return   0;
}
void    test1()
{
    printf("ix = %d¥n",++ix);
    printf("ix1 = %d¥n",++ix1);
}
```

これは変数なので，代入式を使用してコンビニエンス変数に値を保存することができます．

たとえば，ixが指す値を$hogeに保存するには，以下のようにします．

```
set $hoge = ix
```

リスト1，**図1**，**リスト2**，**図2**のようにインクリメントされるカウンタとして使用することもできます．

図3のようにGDBによって，いくつかのコンビニエンス変

```
$ gdb test21
--起動メッセージ省略--
(gdb) list                          ←①ソース・リストを表示
1       /*
2        *グローバル変数
3        */
4       #include <stdio.h>
5       void    test1();
6       int             ix;
7       int             ix1     =       20;
8       int main(int argc, char* argv[])
9       {
10              ix      =       10;
(gdb)
11              test1();
12              return  0;
13      }
14      void    test1()
15      {
16              printf("ix = %d¥n",++ix);
17              printf("ix1 = %d¥n",++ix1);
18      }
19
(gdb)                               ←②11行目にブレークポイントを設定
Line number 20 out of range; test21.c has 19 lines.
(gdb) break 11
Breakpoint 1 at 0x401084: file test21.c, line 11.
(gdb) run                           ←③実行
Starting program: /home/岸/dbg/test21.exe

Breakpoint 1, main (argc=1, argv=0x4f1d50) at test21.c:11
11              test1();            ←④11行目で停止
(gdb) set $hoge = ix                ←⑤変数hogeにixの値をセット
(gdb) p $hoge                       ←⑥変数hogeを表示
$1 = 10
(gdb) p ix                          ←⑦ixの値を表示(当然hogeと同じ)
$2 = 10
(gdb) p $1                          ←⑧値履歴変数$1を表示(当然hogeと同じ)
$3 = 10
(gdb)
```

図1　コンビニエンス変数を表示するGDBの操作1

リスト2　コンビニエンス変数の説明のために構造体に値を設定しているソース(test2.c)

```
/*
 *構造体
 */
#include <stdio.h>
#include <stdlib.h>
typedef struct sampling_data_area
//構造体の定義
{
  int    k1;
  int    k2;
  int    k3;
} sampling_data,*sampling_dataPtr;
void test(sampling_data     *d);
int main(void)
{
    int  ix;
    sampling_data           gSAMPLING_DATA[10];
        //構造体
    for  (ix=0;ix<10;ix++)
    {
        memset(&gSAMPLING_DATA[ix], 0, sizeof(sampling_data));
        //確保したエリアをクリアしている
    }
    for  (ix=0;ix<10;ix++)
    {
        gSAMPLING_DATA[ix].k1   =    rand()%10+1;
    }
    for  (ix=0;ix<10;ix++)
    {
        gSAMPLING_DATA[ix].k2   =    rand()%10+1;
    }
    for  (ix=0;ix<10;ix++)
    {
        gSAMPLING_DATA[ix].k3   =    rand()%10+1;
    }
    return 0;
}
```

```
(gdb) show convenience              ←①今までに使用されたコンビニエンス変数とその値の一覧を表示する
$ix3 = 10
$ix1 = 10
$ix0 = 14
$bpnum = 1                          ←②最後に設定されたブレークポイントの番号が入っている
(gdb)
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x004011c0 in main at test22.c:33
        breakpoint already hit 1 time
(gdb)
```

図3　コンビニエンス変数を表示するGDBの操作3