第14回 標準的デバッガGDBのすべて

岸 哲夫

実行中のスタック・フレームに関する情報を見る

プログラムが正しく動くか否かは，内部データのチェックにかかっている．今回は，前々回（本誌2006年6月号掲載）の続きで，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．（筆者）

※本稿で掲載したリスト1～リスト4の全ソースは，本誌付属CD-ROMに収録しました．

「作業言語」の基本

● 作業言語とは

異なるプログラミング言語（以下，言語）であっても，たいていは共通する点がいくつもあります．しかし，その表記法はまったく同様ではなく，言語によって違いがあります．

特定の言語については，GDBにその言語に固有の情報が組み込まれています．これにより，プログラムを記述する際に使用した言語を使ってさまざまな操作を記述したり，構文に従ってGDBに値を出力させることができます．このとき，式を記述するために使用する言語を「作業言語」と呼びます．

● 作業言語の制御方法

作業言語を制御する方法は二つあります．GDBに自動的に設定させる方法と，ユーザが手作業で選択する方法です．

```
set langage
```

で設定状況の確認と変更が行えます．このset langageコマンドを使って手作業で選択する場合については，後で詳しく述べます．

GDBに作業言語を自動的に設定させるには，

```
set language local
```

または，

```
set language auto
```

で指定します．

たとえば，C言語だと変数aのポインタとして*aを定義しますが，Pascal言語では変数aに対してa@となります．

Pascal言語といえば，Macintoshのプログラムを作るためのPascal言語を思い出す人もいることでしょう．筆者は，Pascal言語と聞くと爆弾ダイアログもいっしょに思い出して胃が痛くなりますが（笑），4thDimentionの言語もポインタはこの表記でした．

作業言語の設定と使い方

● ファイル拡張子と言語のリスト

前々回の繰り返しになりますが，ソース・ファイル名が以下のいずれかの拡張子をもつ場合に，GDBはその言語を以下に示すものと推定します．

▶Cソース・ファイルの場合

- .c

▶C++ソース・ファイルの場合

- .C
- .cc
- .cp
- .cpp
- .cxx
- .c++

▶Fortranソース・ファイルの場合

- .f
- .F

▶CHILLソース・ファイルの場合

- .ch
- .c186
- .c286

▶Modula-2ソース・ファイルの場合

- .mod

▶アセンブラ言語のソース・ファイルの場合

- .s
- .S

アセンブラ言語のソース・ファイルの場合，実際の動作はほとんどC言語と同様ですが，ステップ実行時に，GDBは関数呼び出しのための事前処理部をスキップしません．デバッグに必要でなければ，何らかの策を講じるべきです．

● set languageの使い方

GDBに言語を自動的に設定させる場合，ユーザのデバッグ・セッションとユーザのプログラムにおいて，式は同様に解釈されます．もしそうしたければ，言語を手作業で設定することもできます．set languageコマンドでそれが可能になります

前述のように，GDBに作業言語を自動的に設定させるには，

```
set language local
```

または，

```
set language auto
```


KEYWORD ── GDB，作業言語，set language，set language local，set language auto

を使用します．この場合，GDBは作業言語を推定します．

```
set langage
```

と入力すると，GDBがサポートしている言語一覧を表示します(**図1**)．

たとえば，現在デバッグ中の言語をobjective-cと認識させたい場合には，

```
set langage objective-c
```

と入力します．なお，各言語のデバッグについては，この連載の最後に簡単に説明します．

デフォルトでは，作業言語はソース言語の拡張子によってGDBに推定されます．何らかの事情でそれが正しくない場合，手作業で指定すれば問題ありません．

規定の拡張子以外のソース・ファイルの場合には，警告メッセージを出力します．

スタック・フレームに関する情報を見る方法

GCCでは，関数を呼び出すごとにその呼び出し元がわかるように記憶されて処理されています．このとき，この情報はフレーム・ポインタという領域に保存されます．

関数がA→B→C→Dと呼ばれたら，そのアドレスはスタックという考え方で積み上げられます．スタックの各要素はスタック・フレームと呼ばれ，関数の情報が保持されています．それを参照する方法を記します．

ソース・ファイルを**リスト1**に示します．また，生成されたアセンブラ・ソースを**リスト2**に，この実行形式の逆アセンブラを**リスト3**に示します．なお，mapリストは**リスト4**に示します．

```
(gdb) set language
The currently understood settings are:

local or auto    Automatic setting based on source file
ada              Use the Ada language
c                Use the C language
c++              Use the C++ language
asm              Use the Asm language
minimal          Use the Minimal language
fortran          Use the Fortran language
objective-c      Use the Objective-c language
java             Use the Java language
modula-2         Use the Modula-2 language
pascal           Use the Pascal language
scheme           Use the Scheme language
```

図1　GDBがサポートしている言語一覧を表示

実際にスタック・フレームに関する情報を見てみよう

リスト1のソース・ファイルのコンパイルは，次のように行います．

```
gcc test15.c -g -o test15
```

gdb test15でデバッグに入ってから，

```
info frame
```

とコマンドを打鍵すると，

```
no stack
```

と表示されます．

まだ何も**リスト1**のコードを実行していないので，このように表示されることは当然です．

それでは，**リスト1**中の関数内部で，関数呼び出しを行っているところにブレークポイントをセットしてみます

それぞれのブレークポイントで次のコマンドを実行した結果が**図2**(pp.177-178)です．

- f：フレーム情報を表示
- info args：ここでの引き数を表示
- info locals：ここでのローカル変数を表示

なお，関数のアドレスのマッピングについては生成された

リスト1　Cソース・コード(test15.c)

```
#include <stdio.h>
//
// 現在のフレームを見やすくするプログラム
//
void    proc01(int    a,long    b);
void    proc02(int    a,long    b);
void    proc03(int    a,long    b);
void    proc04(int    a,long    b);
void    proc05(int    a,long    b);
void    proc06(int    a,long    b);
void    proc07(int    a,long    b);

main()
{
    proc01(1,100000000);
}
void    proc01(int    a,long    b)
{
    int  proc01_a  =    100;
    char *proc_01achar  =    "abcdefg";
    printf("proc01¥n");
    proc02(2,100000000);
}
void    proc02(int    a,long    b)
{
    int  proc01_a  =    200;
    char *proc_01achar  =    "abcdefg";
    printf("proc02¥n");
    proc03(3,100000000);
}
void    proc03(int    a,long    b)
{
    int  proc01_a  =    300;
    char *proc_01achar  =    "abcdefg";
    printf("proc03¥n");
    proc04(4,100000000);
}
void    proc04(int    a,long    b)
{
    int  proc01_a  =    400;
    char *proc_01achar  =    "abcdefg";
    printf("proc04¥n");
    proc05(5,100000000);
}
void    proc05(int    a,long    b)
{
    int  proc01_a  =    500;
    char *proc_01achar  =    "abcdefg";
    printf("proc05¥n");
    proc06(6,100000000);
}
void    proc06(int    a,long    b)
{
    int  proc01_a  =    600;
    char *proc_01achar  =    "abcdefg";
    printf("proc06¥n");
    proc07(7,100000000);
}
void    proc07(int    a,long    b)
{
    int  proc01_a  =    700;
    char *proc_01achar  =    "abcdefg";
    printf("proc07¥n");
}
```