6 実習案内 GDBデバッガ

文法的に正しいプログラムの虫取り(debug; i.e. 虫, bug, すなわち誤りを取り除くこと)のために、実習室の計算機にはGDB というデバッガが備わっています。 GDBを用いれば 、

- プログラムが異常終了した際に生成される(ことがある)coreファイルを使って異常の起こった場所を突き止めたり、
- C(やFortran, Pascalなどの)プログラムの実行を追跡して実行途中の変数値を調べたり、

できます。

参考文献:

- 小山祐司&斉藤靖&佐々木浩&中込知之「UNIX入門 フリーソフトウェアによる最新UNIX環境」(1996年, トッパン)
- R.M.Stallman&R.H.Pesch「GDB入門」(1999年, アスキー出版局, 1900円+税)

6–1 実行時のエラーについて

プログラム実行時に起こるエラーとしては、次の様なものがあります。

- Segmentation fault（シグナルSIGSEGV）
  プロセスに割り当てられていない領域へのアクセス、または書き込み禁止領域への書き込みを行おうとした。
- Bus error（シグナルSIGBUS）
  ワード境界を無視してメモリアクセスを行った。
- Illegal instruction（シグナルSIGILL）
  不正な(機械語)命令を実行しようとした。
- Stack Overflow
  プログラムが使用する変数領域が大きくなり過ぎた。大抵の場合、無限再帰が原因。
- Floating point exception（シグナルSIGFPE）
  0による除算, オーバーフロー, アンダーフロー等が起こった。

バッファリング：

主記憶 $\Longleftrightarrow$ バッファ $\Longleftrightarrow$ 入出力装置

CPU　　入出力装置

例6．1（バッファリングの影響）

```
[motoki@x205a]$  nl lab-divide-by-0.c
     1 /*-------------------------------------------
     2 /* 出力の効率を上げるためバッファリングが行われているので...
     3 /* 実行時エラーの直前の出力は画面に出力されないことがある..
     4 /*-------------------------------------------
     5 #include <stdio.h>

     6 int main(void)
     7 {
     8   int k;
```

```
 9    printf("Starting\n");
10    printf("Before division... ");     <-- 実行済か？
11    k = 1/0;     /* 0による除算(エラー) */
12    printf("After division\n");
13    return 0;
14 }
[motoki@x205a]$  gcc lab-divide-by-0.c
[motoki@x205a]$  ./a.out
Starting
Floating point exception (core dumped)
[motoki@x205a]$
```

注目点：

プログラム10行目が実行されているにも関わらず

Before division...

という文字列が画面に書き出されていない。

プログラムの出力列の正確な場所にエラーメッセージを入れたい場合は、次の様にこまめにバッファの内容を出力装置に吐き出す。

```
[motoki@x205a]$ nl lab-divide-by-0-fflush.c
     7 #include <stdio.h>

     8 int main(void)
     9 {
    10   int k;

    11   printf("Starting\n");
    12   fflush(stdout);
    13   printf("Before division... ");
    14   fflush(stdout);
    15   k = 1/0;      /* 0による除算(エラー) */
    16   printf("After division\n");
    17   fflush(stdout);
    18   return 0;
    19 }
[motoki@x205a]$ gcc lab-divide-by-0-fflush.c
[motoki@x205a]$ ./a.out
Starting
Before division... Floating point exception (core dumped)
```

```
[motoki@x205a]$
```

6–2 自習 coreファイルを用いたデバッグ

UNIXでは 、プロセスがSegmentation faultなどの原因で異常終了すると core という名前のファイルが作られ、そこに異常終了時のプログラムの状態(メモリイメージ)が保存される様になっています。

このcoreファイルは巨大になりがちなので生成されない様に環境設定されていることもありますが、もしcoreファイルが生成されているなら、これを使ってGDBで異常発生の場所やその時の変数の値を調べることが出来ます。

[但し、GDBを効果的に使うためにはgccコマンドを-gオプション付きで実行して、得られた実行ファイルによってcoreファイルが生成されている必要があります。]

例6．2（coreファイルを用いたデバッグ）coreファイルを用いて異常発生の場所を確かめている様子を次に示す。

注意：
GDBとの会話の中でプログラムの行番号を使うことがある。この時の行番号は空行もカウントしたものなので、GDBを使う際に行番号付きでプログラムを表示させる場合は、-aオプション付きのnlコマンド、または -nオプション付きのcatコマンドを使うべきである。

```
[motoki@x205a]$  cat -n lab-ex02-memory-overflow.c
                 ..................... 空行にも行番号を付けておく
     1 #include <stdio.h>
     2
     3 int main(void)
     4 {
     5   int  i, a[10];
     6
```

```
 7    a[1] = a[2] = 1;
 8    for (i=3; i<=10; ++i)              /* 間違っていても、エラー
 9      a[i] = a[i-1] + a[i-2];          /* にならないこともある.
10
11    printf("a[10]=%d\n", a[10]);  /* a[10] という領域は
12                                      /*本当は確保されていない*/
13
14    printf("a[512]=%d\n", a[512]);/* a[512]へのアクセスは
15
16    for (i=3; i<=1000; ++i)     /* i=513の時になって      */
17      a[i] = a[i-1] + a[i-2];  /* ようやくエラーになる。*/
18    return 0;
19 }
[motoki@x205a]$  ulimit -c 1000
```

..... **core**ファイルの大きさの上限を**1000kB**に設定

補足：
ulimitはbash(Bourne Againシェル)の下で使えるコマンドであって、実習室で標準になっているtcsh(Tenex Cシェル)の下では同じ設定を　limit core 1000k　と書く。

```
[motoki@x205a]$  gcc -g lab-ex02-memory-overflow.c
```
..... デバッグ情報付きの実行形式ファイルを生成

```
[motoki@x205a]$  ./a.out
a[10]=56
a[512]=0
Segmentation fault (core dumped)
[motoki@x205a]$  gdb a.out core
```
..... GDBデバッガをcoreファイル付きで起動

```
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Licens
welcome to change it and/or distribute copies of it under cert
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB.  Type "show warranty'
This GDB was configured as "i386-redhat-linux"...
Core was generated by ‘a.out’.
Program terminated with signal 11, セグメンテーション違反です.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x80484bc in main () at lab-ex02-memory-overflow.c:17
17      a[i] = a[i-1] + a[i-2];      /* ようやくエラーになる。*/
```

.... 異常があった行とその行番号が表示されている。

```
(gdb)  list lab-ex02-memory-overflow.c:17
```

.... **lab-ex02-memory-overflow.cというファイルの17行目付近を表示。**

```
12                                          /* 本当は確保されていない。*/
13
14   printf("a[512]=%d\n", a[512]); /* a[512]へのアクセスは...*
```

```
15
16    for (i=3; i<=1000; ++i)          /* i=513の時になって      */
17      a[i] = a[i-1] + a[i-2];        /* ようやくエラーになる。*/
18    return 0;
19 }
(gdb)  print i ..... 異常発生時の変数 i の値を表示。
$1 = 513
(gdb)  quit ..... GDBデバッガを終了。
[motoki@x205a]$  ls -l core
-rw-------    1 motoki   motoki      61440 Mar  1 16:04 core
[motoki@x205a]$  rm core
            .... 大抵の場合coreファイルは巨大になっているので、
                 削除しておく。
rm: ‘core’ を削除しますか(yes/no)?  y
[motoki@x205a]$
```

coreファイルは普通は巨大なものになっていますから、使った後は必ず削除しておくこと。

6–3 GDBを用いて実行追跡する例

GDBデバッガを用いれば、1ステップずつ実行して時々変数値を観察することによって、プログラムの動作を細かく追跡することも出来ます。

例6．3（1ステップずつ実行・追跡）例2.1の除算&切上げプログラムの場合、GDBを用いて次の様にプログラム実行を1ステップずつ追跡することが出来ます。

```
[motoki@x205a]$  cat -n lab-ex01-ceiling.c
                 ..................... 空行にも行番号を付けておく
     1 #include <stdio.h>
     2 int main(void)
     3 {
     4   int x,y,sum,ceiling;
     5
     6   scanf("%d %d", &x, &y);
```

```
 7    sum=x+y;
 8    ceiling=(x+y-1)/y;          /* x/y の小数点以下切り上げ
 9    printf("%d+%d=%d\n", x, y, sum);
10    printf("ceiling(%d/%d)=%d\n", x, y, ceiling);
11    return 0;
12 }
[motoki@x205a]$  gcc -g lab-ex01-ceiling.c
[motoki@x205a]$  gdb a.out
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Licens
welcome to change it and/or distribute copies of it under cert
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
This GDB was configured as "i386-redhat-linux"...
(gdb)  break main ..... 中断点を関数mainの入口に設定
Breakpoint 1 at 0x80483ff: file lab-ex01-ceiling.c, line 6.
```

```
(gdb)  run ..................................... 実行開始
Starting program: /home/motoki/C-Java2008/Programs-C/a.out

Breakpoint 1, main () at lab-ex01-ceiling.c:6
6    scanf("%d %d", &x, &y);.. 次に実行する行が表示されている。
(gdb)  next  関数scanfの処理を1ステップと見なして1ステップ実行
8 3  ..................................... プログラムへの入力
7    sum=x+y;
(gdb)  print x ..................... この時点でのxの値を表示
$1 = 8
(gdb)  print y ..................... この時点でのyの値を表示
$2 = 3
(gdb)  print sum
$3 = 134513643..... {まだsumには値がセットされていない。}
(gdb)  step ................................... 1ステップ実行
8    ceiling=(x+y-1)/y;              /* x/y の小数点以下切り上げ */
(gdb)  print sum
```

```
$4 = 11
(gdb)  cont  ..... 次の中断点(無い)まで実行
Continuing.
8+3=11
ceiling(8/3)=3

Program exited normally.
(gdb)  quit  ..... GDBを終了
[motoki@x205a]$
```

6–4 GDBデバッガの使い方

GDBを用いてCプログラムの実行追跡をするには普通次の様にします。

（1）-g オプションを指定して gcc（またはcc）コマンドを実行する。

［ -g オプションを指定してコンパイルすると 、変数や関数のデータ型、実行形式コードのアドレスとソースコードの行番号の対応、等のデバッグ情報がオブジェクトファイルの中に格納されます。］

（2）gdb 実行形式ファイル とコマンド入力してGDBを起動する。

［これによって、(gdb)というプロンプトが現われるはずです。この状態で次のコマンドが可能。

help ...GDBコマンド群の簡単な説明一覧
help コマンド群の名前 .そのコマンド群の中のコマンドの簡単な説明
help コマンド名 ...そのコマンドの簡単な説明］

（3）プログラム実行の途中で止まって変数値が意図した通りになっているかどうかをチェックする場所（中断点,breakpoint）を指定する。

［実行時のエラーでプログラムが中断される場合は、これを行わずにプログラムを実行させ、エラーで実行中断されてからその時の変数値等を調べてもよい。］

例えば、次の様な指定ができます。

`break [filename:] linenum`

… （ソースファイル filename の）linenum 行目で実行を中断。

`break [filename:] function`

… （ソースファイルfilenameの）関数 function の呼び出し直後に実行を中断。

`watch exp`

… 式 exp の値が前回と違っていたら実行を中断。（実行速度が著しく低下するので、なるべく使用は避ける。）

（４）(gdb)というプロンプトに対して

run [args] [< file1] [> file2]

とコマンド入力して、GDBの下でプログラムを実行する。

[この後、GDBは最初の中断点でプログラムを一時停止させ、コマンド待ちの状態になる。]

（５）プログラムの実行が終了するまで 次の操作を繰り返し行う。[行う順序は任意。]

- それまでの実行追跡で表示された事柄を吟味する。
- 現在の中断点における変数値等を表示させる。

例えば次の様なコマンド入力 ができます。

print [/format] expr

··· 式 expr の値を表示する。format部(オプション)には次の指定が可能です。

format	意味
t	2進表示
o	8進表示
x	16進表示
d	符号付き10進表示
u	符号なし10進表示
f	浮動小数点表示
c	文字表示
a	アドレス表示

---

x [/nfu] addr

… addr で指定されたアドレスから始まる、n個のデータ(単位u)の内容を書式f で表示。(examineの意)

n部は省略すると1と見なされる。
f 部は print で許される指定に加え次の指定も可能で、初期のデフォルトは x(16進)。

f(ormat)	意味
s	文字列表示
i	命令表示(逆アセンブル)

u部は次の指定が可能。

u(nit)	意味
b	byte
h	half word (2byte)
w	word (4byte,初期のデフォルト)
g	giant word (8byte)

- 現在の中断点に止まる度に変数値等を表示する様に指示する。

例えば 次の様なコマンド入力ができます。

`display [ /format ] expr`

… 式 `expr` の値を表示する。`format`部(オプション)には`print` で許される指定が可能です。

- 中断点からの実行を再開する。

  例えば次の様なコマンド入力ができます。

  cont　… 次の中断点まで実行。

  next　… 次の1行だけ実行して中断。［次が関数呼び出しの時は、関数呼び出しを含む行全体を「次の1行」と考える。］

  nexti　… 次の1機械語命令だけ実行して中断。［次が関数呼び出しの時は、関数呼び出しを「次の1機械語命令」と考える。］

  step　… 次の1行だけ実行して中断。［次が関数呼び出しの時は、関数本体中の最初の1 行を「次の1行」と考える。］

  stepi　… 次の1機械語命令だけ実行して中断。［次が関数呼び出しの時は、関数本体中の最初の1 命令を「次の1機械語命令」と考える。］

- 前回と同じコマンドを実行する。
  (Enter だけを押す。)
- 中断点を (追加) 指定する。
  (上記 (3) の `break` コマンドと `watch` コマンド)
- 現在の追跡状況等を表示する。

例えば 次の様なコマンド入力ができます。

`where`

... 現在の止まっている中断点での関数の呼出し状況(どの関数の何行目で関数が呼ばれ、その関数の何行目でまた別の関数が呼ばれ、... といった情報)を表示する。

`info breakpoint`

... その時点で考慮されている中断点の情報を表示。

`whatis name`

... 識別子 `name` の型を表示。

`ptype type`

... データ型 `type` の定義を表示。

- 中断点の指定を解除／復活する。

例えば 次の様なコマンド入力ができます。

`disable bp-num`
… (info breakpoint コマンドで表示される) 中断点番号 bp-num の中断点を (一時的に) 無効にする。

`enable bp-num`
… 中断点番号 bp-num の中断点を有効にする。

`delete bp-num`
… 中断点番号 bp-num の中断点の登録を抹消する。

`clear position`
… プログラム上の位置 position に設定されている中断点の登録を抹消する。

- ソースプログラムの一部を表示する。

  例えば 次の様なコマンド入力ができます。

  `list`
  … 現在の止まっている中断点付近のソースコードを表示。

  `list [ filename: ] line-num`
  …
  `line-num` 行目付近のソースコードを表示。

- プログラムを無視して、変数の値を強制的に変えてみる。

  例えば 次の様なコマンド入力ができます。

  `set variable var = expr`
  … 式 `expr` の値を変数 `var` に代入する。

- 現在の実行を強制終了させる。
  （Ctrl-c を押す。）

（６）まだ実行追跡を行いたければ（３）または（４）に戻る。

（７）quit と入力してGDBを終了。

6–5 中断点を指定して実行追跡する例

繰り返し構造のあるプログラムの場合は、実行ステップ数が大きく1ステップ実行だけでは実行追跡が煩わしくなります。

こんな場合は、繰り返しループの中に数箇所の中断点を設け、その場所における変数値を観察することによって、プログラムの動作を追跡することが出来ます。

例6．4（ループの中に中断点を設けて実行追跡）繰り返し処理のあるプログラムの場合、GDBを用いて次の様にプログラムの実行追跡を行うことが出来ます。

```
[motoki@x205a]$  cat -n lab-ex03-factorial.c
                 ..........................  空行にも行番号を付けておく
     1 /* 階乗の計算 */
     2
```

```
 3 #include <stdio.h>
 4
 5 int main(void)
 6 {
 7   int   i,n;
 8   float fact;
 9
10   scanf("%d", &n);
11   fact = 1;
12   for (i=2; i<=n; ++i)
13     fact = fact * i;
14   printf("%d! = %.0f\n", n, fact);
15   return 0;
16 }
[motoki@x205a]$  gcc -g lab-ex03-factorial.c
[motoki@x205a]$  gdb a.out
GNU gdb 5.0
```

```
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Licens
welcome to change it and/or distribute copies of it under cert
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
This GDB was configured as "i386-redhat-linux"...
(gdb)  break 13
Breakpoint 1 at 0x8048430: file lab-ex03-factorial.c, line 13.
(gdb)  run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
3 ..................................... プログラムへの入力

Breakpoint 1, main () at lab-ex03-factorial.c:13
13      fact = fact * i;
(gdb)  display i
          ... この中断点に止まる度に変数 i の値を表示する様に指示
1: i = 2
```

```
(gdb)  display fact
2: fact = 1
(gdb)  cont
Continuing.

Breakpoint 1, main () at lab-ex03-factorial.c:13
13      fact = fact * i;
2: fact = 2
1: i = 3
(gdb) ..... Enter のみ打って、前回と同じコマンド実行を指示
Continuing.
3! = 6

Program exited normally.................. プログラムの実行が終了
(gdb)  info breakpoint  .. 現在設定されている中断点の情報を表示
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048430 in main at lab-ex03-fac
```

```
breakpoint already hit 2 times
(gdb)  disable 1 ..... 中断点番号1の中断点を一時的に無効にする
(gdb)  info breakpoint
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x08048430 in main at lab-ex03-fac
breakpoint already hit 2 times
(gdb)  enable 1 ........ 中断点番号1の中断点を有効なものに戻す
(gdb)  info breakpoint
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048430 in main at lab-ex03-fac
breakpoint already hit 2 times
(gdb)  run ............................. 2度目のプログラム実行
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
3 ..................................... プログラムへの入力

Breakpoint 1, main () at lab-ex03-factorial.c:13
13      fact = fact * i;
```

```
2: fact = 1
1: i = 2
(gdb)  cont
Continuing.

Breakpoint 1, main () at lab-ex03-factorial.c:13
13      fact = fact * i;
2: fact = 2
1: i = 3
(gdb) ..................................... Enter のみ
Continuing.
3! = 6

Program exited normally.
(gdb)  quit
[motoki@x205a]$
```

例6．5（自習 幾つかの関数から成るプログラムの実行追跡）プログラムが幾つかのモジュールに階層的に分割されている場合は、関数呼び出し時のパラメータと関数終了(i.e. return)時の主要変数値を観察することによって、プログラムの動作を追跡することが出来ます。

```
[motoki@x205a]$  cat -n lab-ex04-factorial-func.c
     1 /* 関数呼出しによる階乗計算 */
     2
     3 #include <stdio.h>
     4
     5 long factorial(int);
     6
     7 int main(void)
     8 {
     9   int n;
    10
    11   scanf("%d", &n);
```

```
12   printf("%d! = %ld\n", n, factorial(n));
13   return 0;
14 }
15
16 /*************************************/
17 /* 階乗計算の関数                    */
18 /* --------------------------------- */
19 /*   仮引数 n : 非負整数を想定       */
20 /*   関数値   : n!                   */
21 /*************************************/
22 long factorial(int n)
23 {
24   long fact;
25
26   if (n == 0)
27     fact = 1;
28   else
```

```
    29      fact = n * factorial(n-1);
    30
    31    return(fact);
    32 }
[motoki@x205a]$  gcc -g lab-ex04-factorial-func.c
[motoki@x205a]$  gdb a.out
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Licens
welcome to change it and/or distribute copies of it under cert
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty'
This GDB was configured as "i386-redhat-linux"...
(gdb)  break factorial
Breakpoint 1 at 0x8048442: file lab-ex04-factorial-func.c, lir
(gdb)  break 31
Breakpoint 2 at 0x8048469: file lab-ex04-factorial-func.c, lir
```

```
(gdb)  run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
3 ..................................... プログラムへの入力

Breakpoint 1, factorial (n=3) at lab-ex04-factorial-func.c:26
26   if (n == 0)
(gdb)  step
29     fact = n * factorial(n-1);
(gdb)

Breakpoint 1, factorial (n=2) at lab-ex04-factorial-func.c:26
26   if (n == 0)
(gdb)  cont
Continuing.

Breakpoint 1, factorial (n=1) at lab-ex04-factorial-func.c:26
26   if (n == 0)
```

```
(gdb)
Continuing.

Breakpoint 1, factorial (n=0) at lab-ex04-factorial-func.c:26
26   if (n == 0)
(gdb)
Continuing.

Breakpoint 2, factorial (n=0) at lab-ex04-factorial-func.c:31
31   return(fact);
(gdb)  display fact
1: fact = 1
(gdb)  step
32 }
1: fact = 1
(gdb)
```

```
Breakpoint 2, factorial (n=1) at lab-ex04-factorial-func.c:31
31   return(fact);
1: fact = 1
(gdb)  cont
Continuing.

Breakpoint 2, factorial (n=2) at lab-ex04-factorial-func.c:31
31   return(fact);
1: fact = 2
(gdb)
Continuing.

Breakpoint 2, factorial (n=3) at lab-ex04-factorial-func.c:31
31   return(fact);
1: fact = 6
(gdb)
Continuing.
```

```
3! = 6

Program exited normally.
(gdb)  quit
[motoki@x205a]$
```

6–6 GDBを使って変数の内部状態を調べる

GDBのprintコマンドは変数のbit毎の内容をそのまま正確に表示する訳ではない。

指定したアドレスの内容を表示するには x コマンドを用いる。

例題6．6（整数型変数の内部の状態を調べる） **5.3**~ **5.4** 節ではプログラムの中で整数値がどの様に表されているか説明した。これらの事柄を**GDB**のコマンドを用いて実際に自分の目で観察してみよ。 例えば、$-2, -1, 0, 1, 2, 3, 10, 57$ といった整数が実際に内部でどう表されるかを、**GDB**を用いて調べてみよ。

（考え方）　実際に -2，-1，0，1，2，3，10，57 といった整数値を保持したint型変数を用意し、それらの変数を構成する内部のビット列の状態をGDBを用いてのぞき見すれば良い。

GDBにおいて指定したアドレスから始まる領域の内容を2進表示するには x コマンドに t オプションを付けて、

x/t &変数名

という風な実行をすれば良い。

（プログラムとGDB実行）

```
[motoki@x205a]$ cat -n peep-integer-variables.c
     1 #include <stdio.h>
     2
     3 int main(void)
     4 {
     5   int  minus2=-2, minus1=-1, zero =0,
```

```
 6         plus1 = 1, plus2 = 2, plus3=3,
 7         plus10=10, plus57=57;
 8
 9   printf(
10     "minus2=%d minus1=%d zero =%d\n"
11     "plus1 = %d plus2 = %d plus3=%d\n"
12     "plus10=%d plus57=%d",
13     minus2, minus1, zero,
14     plus1,  plus2,  plus3,
15     plus10, plus57);
16   return 0;
17 }
[motoki@x205a]$ gcc -g peep-integer-variables.c
[motoki@x205a]$ gdb a.out
       (GDBからのメッセージ)
(gdb) break 9
Breakpoint 1 at 0x8048406: file peep-integer-variables.c, line
```

```
(gdb) run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out

Breakpoint 1, main () at peep-integer-variables.c:9
9   printf(
(gdb) x/t &zero
0x7ffff7cc: 00000000000000000000000000000000
(gdb) x/t &plus1
0x7ffff7c8: 00000000000000000000000000000001
(gdb) x/t &plus2
0x7ffff7c4: 00000000000000000000000000000010
(gdb) x/t &plus3
0x7ffff7c0: 00000000000000000000000000000011
(gdb) x/t &plus10
0x7ffff7bc: 00000000000000000000000000001010
(gdb) x/t &plus57
0x7ffff7b8: 00000000000000000000000000111001
```

```
(gdb) x/t &minus1
0x7ffff7d0: 11111111111111111111111111111111
(gdb) x/t &minus2
0x7ffff7d4: 11111111111111111111111111111110
(gdb) cont
Continuing.
minus2=-2 minus1=-1 zero =0
plus1 = 1 plus2 = 2 plus3=3
plus10=10 plus57=57
Program exited normally.
(gdb) quit
[motoki@x205a]$
```

(観察結果について)　5.3~5.4節の説明の通りだとすると、例えば

$$-1\times 2^{31} + \sum_{i=0}^{30} 1\times 2^{i} = -1$$

であるので、-1 という整数はコンピュータ内部では 111111$\cdots$111 というビット列によって表されるはずである。 上の観察結果は確かにこれ

と一致している。

□演習6．8（実数型変数の内部の状態を調べる）次のプログラムの中の**float**型変数 zero, plus1, plus2, plus3, minus1, one10th, inf, nan がコンピュータ内部でどういうビット列で表されているか**GDB**を用いて観察してみよ。これらの観察結果は **IEEE**規格**754**による実数の表現方法と一致しているか？

```
 1  #include <stdio.h>
 2
 3  int main(void)
 4  {
 5    float zero  = 0.0,   plus1  = 1.0,
 6          plus2 = 2.0,   plus3  = 3.0,
 7          minus1=-1.0,   one10th= 0.1,
 8          inf   = 1e100, nan    = 0.0/0.0;
 9
10    printf("zero  =%f  plus1 =%f\n"
11           "plus2 =%f  plus3 =%f\n"
12           "minus1=%f  one10th=%f\n"
```

```
13              "inf =%f      nan =%f\n",
14              zero, plus1, plus2, plus3,
15              minus1, one10th, inf, nan);
16  }
```

例題6．7（8bitでの整数表現）　例題5.3では、長さが8のビット列 $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ を入力して、

① このビット列を unsigned char 型データと見た時に表す(非負)整数値 $\sum_{i=0}^{7} b_i \times 2^i$ 、
および

② このビット列を signed char 型データと見た時に表す整数値 $-b_7 \times 2^7 + \sum_{i=0}^{6} b_i \times 2^i$

を出力するCプログラムを示した。このプログラムの場合、計算結果として得られた2つの整数値は、コンピュータ内部では入力したビット列で表されているはずであるが、これをGDBの x コマンドを用いて観察してみよ。

(考え方)　計算結果が得られた直後に中断点を設定した上でGDB上でプログラムを実行し、計算結果の入っている2つの変数の内部のビット列の状態を例題6.6の場合と同じ様にx コマンドを用いて観察するだけである。

(GDB実行)

```
[motoki@x205a]$ cat -n datatype-bit-string-as-char.c
     (例題5.3を参照)
[motoki@x205a]$  gcc -g datatype-bit-string-as-char.c
[motoki@x205a]$ gdb a.out
        (GDBからのメッセージ)
(gdb)  break 38
Breakpoint 1 at 0x8048550: file datatype-bit-string-as-char.c,
(gdb)  run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
Input a bit string of length 8:  0000 1011
```

```
Breakpoint 1, main () at datatype-bit-string-as-char.c:38
38   printf("\n==>The input bit string can be interpreted to\n
(gdb)  x/t &unsigned_val
0xbffff583: 11111111111111111111111100001011
            .............. 最後の8bitだけがunsigned_valの領域。
                           残りは変数iの領域の一部。
(gdb)  x /tb &unsigned_val
0xbffff583: 00001011
(gdb)  x /tb &signed_val
0xbffff582: 00001011
(gdb)  x /2tb &signed_val
0xbffff582: 00001011 00001011
            2つ目の8bitはunsigned_valの内容
(gdb)  cont
Continuing.

==>The input bit string can be interpreted to
```

```
   have a value 11 as a unsigned char data, and
   have a value 11 as a signed char data.

Program exited normally.
(gdb)  run
Starting program: /home/motoki/C-Java2008/Programs-C/a.out
Input a bit string of length 8:  1111 1111

Breakpoint 1, main () at datatype-bit-string-as-char.c:38
38   printf("\n==>The input bit string can be interpreted to\n
(gdb)  x /tb &unsigned_val
0xbffff583: 11111111
(gdb)  x /tb &signed_val
0xbffff582: 11111111
(gdb)  cont
Continuing.
```

```
==>The input bit string can be interpreted to
   have a value 255 as a unsigned char data, and
   have a value -1 as a signed char data.

Program exited normally.
(gdb)  quit
[motoki@x205a]$
```

□演習6．10（IEEE規格754による実数表現）演習5.12では、長さが32のビット列

$$\mathrm{s}\ \mathrm{e_7}\ \mathrm{e_6}\cdots\mathrm{e_1}\ \mathrm{e_0}\ \mathrm{d_1}\ \mathrm{d_2}\ \mathrm{d_3}\cdots\mathrm{d_{23}}$$

を入力して、このビット列の表す実数値(実数表現の仕方はIEEE規格754に従うものと仮定する)

$$\begin{cases} (-1)^{\mathrm{s}}\times(1+\mathrm{M})\times 2^{\mathrm{E}} & \text{if } -127<\mathrm{E}<128 \\ (-1)^{\mathrm{s}}\times\mathrm{M}\times 2^{\mathrm{E}+1} & \text{if } \mathrm{E}=-127 \\ \mathtt{Inf}(\text{無限大}) & \text{if } \mathrm{E}=128,\ \mathrm{M}=0 \\ \mathtt{NaN}(\text{非数},\ \text{Not a Number}) & \text{if } \mathrm{E}=128,\ \mathrm{M}\neq 0 \end{cases}$$

ここで、$\mathrm{M}=\sum_{\mathrm{i}=1}^{23}\mathrm{d_i}\times 2^{-\mathrm{i}}$,

$\mathrm{E}=\sum_{\mathrm{i}=0}^{7}\mathrm{e_i}\times 2^{\mathrm{i}}-127$

を出力するCプログラムを作成することを演習課題とした。

このプログラムの場合、使用した計算機の実数表現方式がIEEE規格754に従っているなら、計算結果として得られた実数値はコンピュータ内部では入力したビット列で表されているはずであるが、これをGDBの x コマンドを用いて調べてみよ。

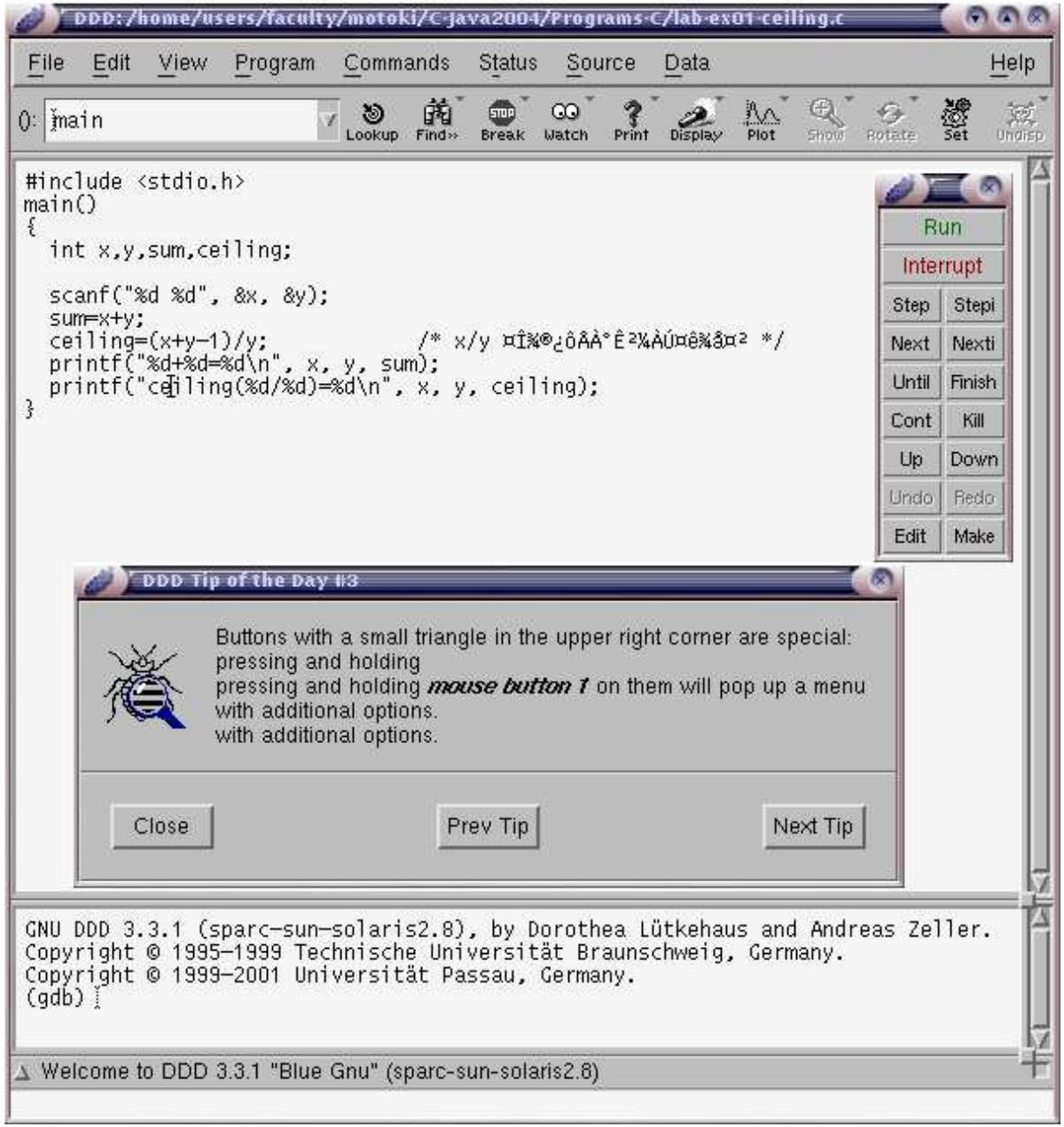
6–7 自習 DDD ---GDBのグラフィカルなフロン

GDBのグラフィカルなフロントエンドとして動作するDDD と呼ばれるツールも存在します。実習室にはインストールされていませんが、

DDDを使えば 、
　GUIベースにGDBを使えるのはもちろん、
　注目したい変数の内容をグラフィカルに表示したり、
　配列内のデータ列をまとめてグラフ表示(プロット表示)したり、
といったことも出来ます。

DDD/GDBを用いてCプログラムの実行追跡をするには ...
(1) -g オプションを指定して gcc (またはcc) コマンドを実行する。

(2) ddd 実行形式ファイル & とコマンド入力してDDD/GDBを起動する。



← メニューバー

← ツールバー

ソースコードウィンドウ

GDB コンソール

← ステータスバー

（3）Tips ウィンドウを閉じる。

（４）様々な操作を行う。例えば、

- DDDというタイトルの縦長のウィンドウ（コマンドツールウィンドウと言う）の中のボタンを押してstep実行等の指示を行う。

- ソースコードの行頭をマウスでクリックした後でツールバーのBreakボタンを押すと、その行がブレイクポイントとして設定される。

- ブレイクポイントとして登録された行頭をクリックするとBreakボタンがClearボタンに変わる。この状態でClearボタンを押すと、ブレイクポイントの設定が解除される。

- ツールバー左端の空欄に関数名を入れてLookupボタンを押すと、指定した関数のソースコードが表示される。

- 注目したい変数を選択しツールバーのDisplayボタンを押すと、その変数の内容がグラフィカルに表示される様になる。

6–8 実行中のプログラムの追跡

GDBは実行中のプロセスも追跡対象にすることが出来ます。具体的には、次の様にします。

```
xcspc70_43%   gdb 実行形式ファイル
    ......................
(gdb)   attach process-ID
    ......................
(gdb)   detach process-ID
    ......................
(gdb)   quit
```

ID番号が process-ID のプロセス(実行中)をGDBに接続

以前にattachしたプロセス(実行中)をGDBから切り離す
