I117（28）デバッグ

知念

北陸先端科学技術大学院大学 情報科学研究科
School of Information Science,
Japan Advanced Institute of Science and Technology

デバッグ debug

バグ(プログラム上の欠陥をさす)を取り除く作業
「バグとり」とも呼ぶ

- デバッガを使う
  - バグの場所や理由が想像がつかない場合
- チェックルーチンで確認

1) デバッガを使う程ではない(軽度の)場合
2) デバッガを使えない(重度の)場合

  - assert, printf あるいは特製(後述)

デバッガ

a) デバッガ上で実行
- デバッガが各種の確認しながら実行を進める、実行が遅い

b) 実行・停止後にコアファイルをデバッガに与える
- 実行は早い、ディスクを消費する

どちらも、コンパイル時に -g をつける
このうち、b) はコアファイルのサイズを指定する

デバッガ上で実行

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        printf("%s\n", 32);
}
```

```
% cc -g a4.c
% dbx ./a.out
新機能についての詳細は 'help changes' を参照してください
このメッセージを抑止するには .dbxrc ファイルに 'dbxenv suppr
a.out の読み込み中
ld.so.1 の読み込み中
```

```
libc.so.1 の読み込み中
libdl.so.1 の読み込み中
libc_psr.so.1 の読み込み中
(dbx) run
実行中: a.out
(プロセス id 21296)
シグナル SEGV (フォルトのアドレスにマッピングしていません) 関数
0xff2b455c: strlen+0x0080:      ld        [%o1], %o2
現関数 :main
    4               printf("%s\n", 32);
(dbx)
```

エラー箇所は 4 行目

※ dbx の終了は exit コマンド、quit でも良い

実行後にコアファイルでデバッグ

コアファイルを作るよう設定し、念のため消しておく

```
% limit coredumpsize unlimit
% rm -f core
% cc -g a4.c
% ./a.out
セグメントエラー (coreを出力しました)
```

作られたファイル core も一緒に dbx に与える

```
% dbx ./a.out core
(略)
a.out の読み込み中
```

```
core ファイルハンドラの読み込みに成功しました
(略)
プログラムは シグナル SEGV (フォルトのアドレスにマッピングしてい
0xff2b455c: strlen+0x0080:      ld        [%o1], %o2
現関数 :main
    4             printf("%s\n", 32);
(dbx)
```

この場合も、4行目のエラーを指摘

実行後にコアファイルでデバッグ *(cont.)*

```
#include <stdio.h>
int sub(char *msg) {
        msg[123456]='X';
}
int main() {
        sub("abc");
}
```

コンパイルできるが、実行はエラーとなる

```
% cc -g a5.c
% ./a.out
セグメントエラー (coreを出力しました)
```

実行後にコアファイルでデバッグ *(cont.)*

```
% dbx ./a.out core
(略)
プログラムは シグナル SEGV (フォルトのアドレスにマッピングしてい
現関数 :sub
    3               msg[123456]='X';
(dbx) print msg
msg = 0x20dd8 "abc"
(dbx) print msg[123456]
dbx: アドレス 0x3f018 にアクセスできません
```

アクセスできない領域をアクセスしようとした

実行後にコアファイルでデバッグ *(cont.)*

関数呼び出しの積み重ね中の位置を調べる

```
(dbx) where
=>[1] sub(msg = 0x20dd8 "abc"), 行 3 "a5.c"
  [2] main(), 行 6 "a5.c"
```

上位関数の前後を見てみる

```
(dbx) up
現関数 :main
    6               sub("abc");
(dbx) list 1,10
    1   #include <stdio.h>
    2   int sub(char *msg) {
```

実行後にコアファイルでデバッグ *(cont.)*

```
3               msg[123456]='X';
4       }
5       int main() {
6               sub("abc");
7       }
```

ヒープ領域のデバッグ

近代的なプラットホームでは様々なデバック機構が用意されている

a) コンパイラ
b) 標準ライブラリ
c) デバッグ向けライブラリ
d) 専用デバッガ

ここでは、Solaris の watchmalloc を使う

watchmalloc

- 標準ライブラリに似せたデバッグ向けライブラリ
- 環境変数 LD_PRELOAD や MALLOC_DEBUG を設定

```
LD_PRELOAD=watchmalloc.so.1
export LD_PRELOAD
MALLOC_DEBUG=WATCH
export MALLOC_DEBUG

./a.out
```

free重複

バギープログラム例 (2回free)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
    char *x=NULL;
    x = malloc(5);
    strcpy(x, "ABC");
    free(x);
    free(x); }
```

free重複 *(cont.)*

```
% cc -g a.c
% ./a.out
```

一応動く

```
% ./X.sh
トレーストラップ - コアダンプしました。
```

先程の環境変数を設定(watchmalloc を有効に)して実行するとエラーになる

```
% dbx ./a.out core
(略)
```

free重複 *(cont.)*

```
watchmalloc.so.1 の読み込み中
(略)
プログラムは シグナル TRAP (書き込みアクセスウォッ
0xff3813b0: realfree+0x0044:    st          %
現関数 :main
    9         free(x);
(dbx)
```

9行目で失敗したことが分かる
free の重複が検出できた

初期化不良

```
#include <stdlib.h>
int main() {
    char **x=NULL;
    x = malloc(sizeof(char*)*5);
    free(x[2]);
}
```

```
% cc -g a3.c
% ./a.out
% ./X.sh
セグメント例外 - コアダンプしました。
```

初期化不良 *(cont.)*

```
% dbx ./a.out core
 (略)
プログラムは シグナル SEGV (フォルトのアドレスにマ
0xff381390: realfree+0x0024:    ld          [
現関数 :main
    5         free(x[2]);
(dbx) print x[2]
x[2] = 0xbaddcafe "<不正アドレス 0xbaddcafe>
```

watachmalloc は malloc直後は 0xbaddcafe という値で初期化する、このアドレスが出て来たら初期化不良

その他のヒープ領域デバッグ

- 他のプラットフォームでは環境変数 MALLOC_OPTION や MALLOC_CHECK_ 等で設定
- フリー／オープン
  - Electrict Fence ; D.U.M.A
  - valgrind
- 各種商品

チェックルーチン

- assert — 条件を指定して確認
- printf — 値を表示して確認
- 特性確認ルーチンを用意する

assert

- 条件を満たさない場合にアボート(強制終了)する機構
- 必ず assert.h を読み込む

```
#include <stdio.h>
#include <assert.h>
int main() {
    int x;
    x=17;
    assert(x<9);
}
```

assert *(cont.)*

```
% cc -g a6.c
% ./a.out
Assertion failed: x<9, file a6.c, line 6
アボート (coreを出力しました)
```

6行目で条件を満たしていないことを報告

マクロ NDEBUG を設定すると、無効になる

```
% cc -g -DNDEBUG a6.c
% ./a.out
```

printf デバッグ

人手で動作確認
任意の場所に printf を挿入する

```
printf("x %d", x);
```

突然プログラムがとまると、バッファ中の文字が表示されないことがある
fflush を呼び出してバッファ内容を吐き出す

```
printf("x %d", x); fflush(stdout);
```

プリプロセッサ

コンパイル時に __FILE__ や __LINE__ がファイル名や行番号に置換される

```
#include <stdio.h>
int main() {
    char *x;
    printf("%s:%d\n", __FILE__, __LINE__);
    x++;
    printf("%s:%d\n", __FILE__, __LINE__);
    printf(x);
    printf("%s:%d\n", __FILE__, __LINE__);
}
```

プリプロセッサ *(cont.)*

```
% cc b1.c
% ./a.out
b1.c:4
b1.c:6
セグメントエラー (coreを出力しました)
```

どこまで進んだか確認しやすい

- 6行までは済んだ
- 7行目以降で問題が発生した

プリプロセッサ *(cont.)*

C99 では __func__ に関数名が置換される

```
#include <stdio.h>
int a() {
    printf("%s:%d %s\n", __FILE__, __LINE__,
                  __func__);
}
int main() {
    printf("%s:%d\n", __FILE__, __LINE__);
    a();
}
```

プリプロセッサ *(cont.)*

```
% cc b2b.c ; ./a.out
b2b.c:7
b2b.c:3 a
% gcc b2b.c ; ./a.out
b2b.c:7
b2b.c:3 a
```

※ 以前は __FUNCTION__ などコンパイラの独自マクロだった

syslog

UNIX なら syslog もある
システムのログファイルに書き出す関数

```
...
#include <syslog.h>
...
syslog(LOG_INFO, "x %d", x);
...
```

/var/log/message 等のログファイルに書き出される

特製ルーチン

プログラム内容に応じて用意する

- あらかじめ答えの分かる計算と比較
- 多数繰り返して統計的手法で確認

定石

- 新しいデータ構造を作る際には表示ルーチンを
- バイナリデータは16進表示を

lint — 伝統的なプログラム調査プログラム

用意されていないプラットホームも多い

```
% lint b1.c
(5) 警告: 変数が設定以前に使用されている可能性があります: x
(9) 警告: 関数中に return 文がありません: main

関数が値を返さずに終了しています
    (9) main

関数が返す値は常に無視されます
    printf
```

gcc -Wall

gcc では疑わしい箇所を調査できる
lint とはまた違うチェックができる

```
% cat a4.c
#include <stdio.h>
#include <stdlib.h>
int main() {
        printf("%s\n", 32);
}
% gcc -Wall a4.c
a4.c: In function ’main’:
a4.c:4: 警告: format ’%s’ expects type ’char *’, but →
argument 2 has type ’int’
a4.c:5: 警告: control reaches end of non-void function
```