情報工学実験１

「C言語応用」

担当教員：　赤嶺 有平
学籍番号：　０９５７０７Ｂ
氏　　名：　大城 佳明
提出日：　2010年6月8日

目 次

1 LEVE1

1.1 LEVEL 1.1

演習 1，1.2, 1.3 の結果を考察せよ.

1.1.1 演習 1.

Pointer.c として下記のプログラムをコンパイルおよび実行せよ

ソース（Pointer.c）

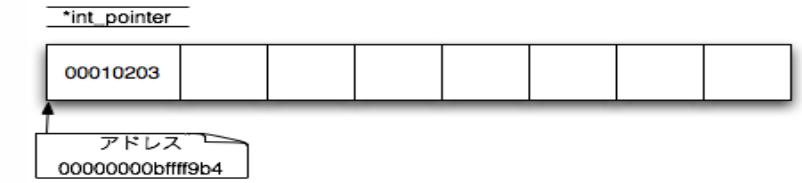
```
01  #include <stdio.h>
02
03  #define ARRAY_SIZE 8
04
05   int main(int argc, char** argv) {
06     int i;
07     int array[ARRAY_SIZE];
08     int *int_pointer = array;
09
10     unsigned char *memory = (unsigned char*)array;
11
12     int\_pointer[0] = 0x00010203;
13
14    //メモリの内容を表示
15     for(i=0; i<sizeof(array); i+=4) {
16         printf("%016lx: %02x %02x %02x %02x \n",
17         (long unsigned int)(memory+i),
18         memory[i], memory[i+1], memory[i+2], memory[i+3]);
19        }
20     return 0;
21  }
```

実行結果（Pointer.c）

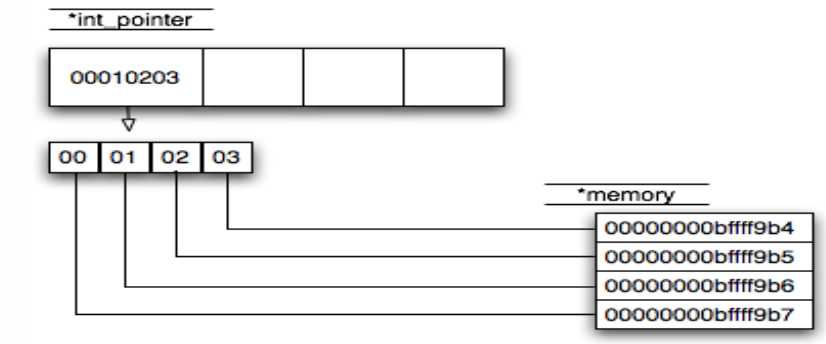
```
01   00000000bffff9b4: 03 02 01 00
02   00000000bffff9b8: c0 04 e0 8f
03   00000000bffff9bc: 0c f5 e2 8f
04   00000000bffff9c0: 04 00 00 00
05   00000000bffff9c4: 00 00 00 00
06   00000000bffff9c8: 00 00 00 00
07   00000000bffff9cc: bc 05 e0 8f
08   00000000bffff9d0: 00 00 00 00
```

＜考察＞

1. Pointer.c の解析を行う
   (a) 03 より ARRAY_SIZE=8 である
   (b) 06〜08 は引数の宣言である
      i. 07 は配列である
      ii. .array のアドレスを int_pointer に入れた
   (c) 10 の「unsigned」とは負を使わないという意味である
   (d) 10 では int 型の array を char 型の array にしてそれを memory にした
   (e) 12 では配列の 0 番目に 1 6 進数で「00010203」を入れている



2. 15〜19 は配列を一つ一つ表示している
   (a) 15 の「i¡sizeof(array)」より 8 回行われることがわかる
   (b) 15 の「i=+4」は int 型は 4 byte なので次の配列を読み込むのに「+4」する必要があるからである
   (c) 17 は memory を int 型のアドレスを表示している
   (d) 18 は memory を順番よく表示している
3. 実行結果より、8 つ表示されているのが分かる
4. 実行結果 01 より、代入した数字が表示されているのが分かる
5. 実行結果より memory は位の順から入れられていることがわかる



6. これをリトルエンディアンという

1.1.2 演習 1.2

Pointer.c に以下のコードを追加し，int 型ポインタの加算が 4 バイト単位である事を確認せよ

ソース（Pointer.c）追加

```
10   int_pointer[0] = 0x00010203;
11   *(int_pointer+1) = 0xFF000001;
12   *(int_pointer+2) = 0xFF000002;
13
14   //メモリの内容を表示
```

実行結果（Pointer.c）追加

```
01 00000000bffff9b4: 03 02 01 00
02 00000000bffff9b8: 01 00 00 ff
03 00000000bffff9bc: 02 00 00 ff
04 00000000bffff9c0: 04 00 00 00
05 00000000bffff9c4: 00 00 00 00
06 00000000bffff9c8: 00 00 00 00
07 00000000bffff9cc: bc 05 e0 8f
```

< 考察 >

1. ソースを解析する
   (a) 11,12 が追加されている
   (b) 11 は int_pointer を＋ 1 したアドレスに「0xFF000001」を入れた
   (c) 12 は int_pointer を＋ 2 したアドレスに「0xFF000002」を入れた
   (d) int_pointer は int 型である
2. 実行結果より 02,03 が変化している事がわかる
3. 「0xFF000001」は最初のアドレス「00000000bffff9b4」より＋ 4 した「00000000bffff9b8」に格納されている
4. 「0xFF000002」は最初のアドレス「00000000bffff9b4」より＋ 8 した「00000000bffff9bc」に格納されている
5. int 型ポインタに＋ n をしたらアドレス＋ 4n されている事が分かる
6. したがって、int 型ポインタの加算は 4 バイトである

1.1.3 演習 1.3

Pointer.c に以下のコードを追加し，unsigned char 型ポインタの加算が 1 バイト単位である事を確認せよ

ソース（Pointer.c）追加

```
12 *(int_pointer+2) = 0xFF000002;
13 *( memory + 5) = 0xee;
14 *( memory + 6) = 0xdd;
15
16 //メモリの内容を表示
```

実行結果（Pointer 追加）

```
01 00000000bffff9b4: 03 02 01 00
02 00000000bffff9b8: 01 ee dd ff
03 00000000bffff9bc: 02 00 00 ff
04 00000000bffff9c0: 04 00 00 00
05 00000000bffff9c4: 00 00 00 00
06 00000000bffff9c8: 00 00 00 00
07 00000000bffff9cc: bc 05 e0 8f
08 00000000bffff9d0: 00 00 00 00
```

< 考察 >

1. l . ソースの解説する
   (a) 13,14 が追加された
   (b) 13 は memory を＋ 5 したアドレスに「ee」を入れた
   (c) 14 は memory を＋ 6 したアドレスに「ff」を入れた
   (d) memory は unsigned char 型である
2. 実行結果より 02 が変化したことがわかる
3. memory の先頭アドレスは「00000000bffff9b4」である
   (a) memory を＋ 5 したところに「ee」が格納されてる
   (b) 「ee」は実行結果より「00000000bffff9b9」に格納されていることがわかる
   (c) memory を＋ 6 したところに「ff」が格納されている
   (d) 「ff」は実行結果より「00000000bffff9ba」に格納されていることがわかる
4. unsigned char 型ポインタに＋ n したらアドレス＋ n されていることがわかる
5. したがって、unsigned char 型ポインタの加算は 1 バイトである

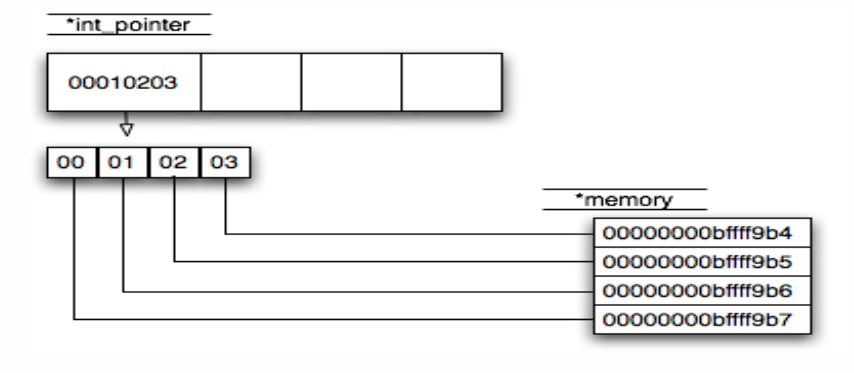
1.2 LEVEL1.2

演習 1，1.2 では，代入した値とメモリの表示内容の順序が逆になっている．それはなぜか．

1. リトルエンディアン
   2 バイト以上のデータ量を持つ数値データを記録したり転送するときには 1 バイトごとに分割するが、これを最下位のバイトから順番に記録/送信する方式。Intel 社のマイクロプロセッサでは、メモリに数値データを格納する際にリトルエンディアンで記録する

2. ビッグエンディアン
   2 バイト以上のデータ量を持つ数値データを記録したり転送するときには 1 バイトごとに分割するが、これを最上位のバイトから順番に記録/送信する方式。Motorola 社のマイクロプロセッサでは、メモリに数値データを格納する際にビッグエンディアンで記録する

3. 演習 1、1.2



4. メモリーの順序が逆になったのは最下位のバイトから順番に記録/送信したからだと考えられる

   したがって、実験に使用したパソコンがリトルエンディアンである

1.3 LEVEL1.3

上記コードを参考に，二つの入力と二つの出力をもち，一回の呼び出しでかけ算と割り算を同時に行う関数及びその呼び出しコードを実装せよ (printf を利用して呼び出し結果を確認できるようにする)

ソース（add_sub.c）

```
01 #include<stdio.h>
02 //基本的構文の関数（二つの入力と一つの出力を持つ）
03 int add(int x, int y)
04 {
05   return x + y;
06 }
07
08 //ポインタを引数にとる関数（二つの入力と二つの出力を持つ）
09 void add_sub(int x, int y, int *res_add, int *res_sub)
10 {
11   *res_add  = x + y;
12   *res_sub  = x - y;
13 }
14
15
16 int main(int argc, char** argv)
17 {
18   printf("add(%d, %d) = %d\n", 2, 3, add(2,3));
19
20  int a,s;
21  add_sub(2, 3, &a, &s);
22
23   printf("add_sub(%d, %d, &a, &s)\n", 2, 3);
24  printf("add = %d\n", a);
25  printf("sub = %d\n", s);
26 }
```

実行結果（add_sub.c）

```
01 add(2, 3) = 5
02 add_sub(2, 3, &a, &s)
03 add = 5
04 sub = -1
```

< 考察 >

1. 1 . ソースを解析する

   (a) 03〜06 は add 関数を宣言してる

i. 03 より外部から int 型の２つの値を取得している
ii. 05 より２つの値を加算した値を return している

(b) 08〜13 は add_sub 関数を宣言している

i.
ii. 09 より int 型の２つの値と int 型の２つのアドレス（ポインタ）を取得してる
iii. 11 では res_add のアドレスに２つの値の和を格納している
iv. 12 では res_sub のアドレスに２つの値の差を格納している

(c) 16〜26 は main 関数である

i. 18 でははは add 関数を呼び出し、printf で出力している
ii. 20 は必要な引数の定義
iii. 21 では add_sub 関数を呼び出し、アドレス a,s に値を入れている
iv. 23 は printf での出力である
v. 24 では加算した結果「a」を出力している
vi. 25 では減算した結果「s」をを出力している

2. 実行結果 01 より add 関数で加算されたことがわかる
3. 03 より add_sub 関数で加算されていることがわかる
4. 04 より add_sub 関数で減算されていることがわかる

2 LEVEL2

2.1 LEVEL2.1

ソース（callback.c）

```
01 #include <stdio.h>
02
03 void DoCallback( int (*cbfunc)(int,int,int) ) //コールバック関数を呼び出す関数
04 //cbfunc は int を返し，三つの int 型引数をとる関数へのポインタ
05 {
06   int ret = cbfunc(0, 1, 2); //コールバック関数を呼び出す
07   printf("callback function returned %d\n", ret);
08 }
09
10 int MyCallbackFunc1(int l, int c, int r)
11 {
12   printf("MyCallbackFunc1 is called\n");
13   return l+c+r;
14 }
15
16 int MyCallbackFunc2(int l, int c, int r)
17 {
18   printf("MyCallbackFunc2 is called\n");
19   return l-c-r;
20 }
21
22 int main()
23 {
24   DoCallback(MyCallbackFunc1);
25   DoCallback(MyCallbackFunc2);
26 }
```

実行結果（callback.c）

```
01 MyCallbackFunc1 is called
02 callback function returned 3
03 MyCallbackFunc2 is called
04 callback function returned -3
```

< 考察 >

1. ソースの解析をする

   (a) 03〜08 は DoCollback 関数の宣言である

      i. 03 より関数ポインタを使い、関数を引数扱いにしている

      ii. 06 はコールバック関数が行われてる

iii. 06 より int 型の ret に cbfunc(0,1,2) の結果の値を代入している

iv. 07 では関数で求めた ret の値を出力している

(b) 10〜14 は MyCollbackFunc1 関数の宣言である

i.

ii. 10 より int 型の整数３つを取得している

iii. 12 は文字の出力

iv. 13 は３つの値の和を return している

(c) C.16〜20 は MyCollbackFunc2 関数の宣言である

i.

ii. 16 より int 型の整数３つを取得している

iii. 18 は文字出力

iv. 19 は３つの値の差を return している

(d) 22〜26 は main 関数である

i.

ii. 24 では DoCallback 関数を使用している。

iii. 24 ではコールバック関数を行うため、MyCollbackFunc1 関数を利用している

iv. 25 では DoCallback 関数を使用している

v. 25 ではコールバック関数を行うため、MyCollbackFunc2 関数を利用している

2. 実行結果 01,02 より MyCollbackFunc1 を使用して加算された値が出力されていることがわかる

3. 実行結果 03,04 より MyCollbackFunc2 を使用して減算された値が出力されていることがわかる

4. 動作を確認するためプログラム一つ一つの間に現在の関数名を入力した

実行結果（callback.c）動作確認

```
01 ///Main 関数///
02 ///DoCallback 関数///
03 ///MyCallbacFunc1 関数///
04 MyCallbackFunc1 is called
05 ///DoCallback 関数///
06 callback function returned 3
07 ///Main 関数///
08 ///DoCallback 関数///
09 ///MyCallbacFunc1 関数///
10 MyCallbackFunc2 is called
11 ///DoCallback 関数///
12 callback function returned -3
13 ///Main 関数///
```

5. 実行結果より考察する

(a) プログラムの流れを書く

i. DoCollback 関数を呼び出す
（01 main 関数）

ii. cbfunc = MyCollbackFunc1 にする
（02 DoCollback 関数）

iii. ソース 06 の時、MyCollbackFunc1 のコールバック関数を呼び出している
（02 DoCollback 関数）

iv. MyCollbackFunc1 で出力、計算が行われる
（03,04 MyCollbackFunc1 関数）

v. int ret に値が入る
（05 DoCollback 関数）

vi. ソース 07 より、出力される
（05 DoCollback 関数）

vii. DoCollback 関数を呼び出す
（06 main 関数）

viii. cbfunc = MyCollbackFunc2 にする
（07 DoCollback 関数）

ix. ソース 06 の時、MyCollbackFunc2 のコールバック関数を呼び出している
（08 DoCollback 関数）

x. MyCollbackFunc1 で出力、計算が行われる
（09,10 MyCollbackFunc2 関数）

xi. int ret に値が入る
（11 DoCollback 関数）

xii. ソース 07 より、出力される
（12 DoCollback 関数）

xiii. 最後に main に戻って return する
（13 main 関数）

(b) DoCallback 関数を読み込むと、MyCallbackFunc 関数も読み込まれていることがわかる

2.2 LEVEL2.2

演習 1.3 のコールバック関数を含むもの含まないものそれぞれのプログラムに,「最初の文字が”#”で始まる」行を抽出するように変更せよ.(#を含む行ではなく，1 文字目が#である行を抽出する）それぞれの変更を行った際に，どのファイルに変更を加えたかに注意し，コードの局所性について考察せよ.

2.2.1 コールバック関数を使っていない

～ 変更前 ～

ソース（callback3Main.c）

```
01 #include<stdio.h>
02 /*
03 標準入力から"printf"が含まれる行を抽出，表示する
04 */
05
06 void searchString(char*);
07
08 int main(void)
09 {
10   searchString("printf");
11
12   return 0;
13 }
```

ソース（callback3Sub.c）

```
01 /*
02  標準入力から指定文字列が含まれる行を抽出，表示する
03  */
04
05 #include <stdio.h>
06 #include <string.h>
07
08 #define BUF_SIZE 256
09
10 void searchString(char* str)
11 {
12   char buf[BUF_SIZE];
13
14   while(fgets(buf, BUF_SIZE, stdin)) {
15     /* strstr(文字列１，文字列２）
16        文字列１に文字列２が含まれていれば,
17        その開始文字を指すポインタを返す.
18        含まれていないときは，NULL を返す.
```

```
19          従って，NULL かどうかを判断する事で
20          文字列 1 が文字列 2 を含むかどうかが判断できる*/
21      if(strstr(buf, str) != NULL) {
22        printf("%s", buf);
23      }
24    }
25 }
```

実行結果（callback3Main.c callback3Sub.c）

```
01 moano
02 n3e
03 printf
04 printf
05 fano
06 joeprintf
07 joeprintf
08 printmoe
```

< 考察 >

1. 1 .callback3Main.c の解析をする
   (a) 06 は使う関数の宣言である
   (b) 08〜13 は main 関数である
   (c) 10 では searchString 関数を利用している
   (d) ここでは ”printf”と同じ文字列なら出力する
2. callback3Sub.c の解析をする
   (a)
   (b) 08 では BUF_SIZE を 256 に定義している
   (c) 10〜25 は searchString 関数である
   (d) 10 では文字列を*char str に取得している。
   (e) 14 では文字列が入力されたら実行される
   (f) 入力された文字列は buf に格納され、最大 256 文字である
   (g) 15〜20 は 21 の説明である
   (h) 22 は入力された文字列に str の文字列が同じならば実行される
   (i) 文字列を表示する
3. 実行結果の 03,04 と 06,07 より ”printf”が含まれる文字は出力されている
4. ”printf”と入力されていなければ出力されない

～ 変更後 ～

ソース（callback3Main.c）変更

```
09  searchString("#");
```

ソース（callback3Sub.c）変更）

```
20        文字列 1 が文字列 2 を含むかどうかが判断できる*/
21     char* s = strstr(buf,str);
22     if(s == buf) {
23       printf("%s", buf);
```

実行結果（callback3Main.c callback3Sub.c）変更

```
01 enova
02 printf
03 no#an
04 #noane
05 #noane
06 m3o#
```

< 考察 >

1. 1 .callback3Main.c は 09 を変更した。
   (a) ”＃”があるかどうかを判断するために callback3Main の 09 を変更する必要がある
   (b) main 関数では変更する事が以上である
2. callback3Sub.c は 21 が追加され、22 が変更された
   (a)
   (b) 「最初の文字が”#”で始まる」より先頭が#であるかどうかの判断をしなければならない
   (c) 21 では buf の str の文字列が同じ部分の先頭アドレスを s に置いた
   (d) 22 では s のアドレスと buf のアドレスが同じかどうか判断している
   (e) 同じなら str の文字列は先頭にあるとこになる
   (f) str の文字が先頭にあれば 23 は実行される

2.2.2 コールバック関数を使っている

～ 変更前 ～

ソース（callback3bMain.c）

```
01 #include <stdio.h>
02 #include <string.h>
03
04 /* 標準入力から 1 行ずつ読み込み
05    第 2 引数として指定したコールバック関数により条件判断を行う.
06    その結果が 1 で有れば標準出力へ出力する
07
08    int (*callback)(char*) は char*型の引数を一つ受け取る関数へのポインタである
09 */
10 void searchWith(int (*callback)(char*) );
11
12 /* 条件判断に使われるコールバック関数
13    line に printf という文字列が含まれていれば 1 を返す
14 */
15 int isContainPrintf(char* line) {
16   return strstr(line, "printf") != NULL;
17   /* 上記は，以下のコードと同じ結果になる.
18      if(strstr(line, "printf") != NULL)
19        return 1;
20      else
21        return 0;
22   */
23 }
24
25 /* コールバック関数 (isContainPrintf) を利用して文字列を検索する */
26 int main(void)
27 {
28   searchWith(isContainPrintf);
29
30   return 0;
31 }
```

ソース（callback3bSub.c）

```
01 #include <stdio.h>
02
03 #define BUF_SIZE 256
04
05 /* 標準入力から 1 行ずつ読み込み
06    引数として与えられたコールバック関数による条件判断を行う.
07    結果が 1 であれば，その行を標準出力へ出力する
```

```
08
09    (int (*callback)(char*) ) は char*型の引数を受け取る関数への
10    ポインタ変数 callback を引数として受け取るという意味
11    従って，呼び出し元が引数として渡した関数ポインタを
12    callback 変数を用いて呼び出す事ができる
13 */
14 void searchWith(int (*callback)(char*) )
15 {
16   char buf[BUF_SIZE];
17   while(fgets(buf, BUF_SIZE, stdin)) {
18     if(callback(buf)) {
19       printf("%s", buf);
20     }
21   }
22 }
```

実行結果（callback3bMain.c callback3bSub.c）

変更前の callback3Main.c と callback3Sub.c の実行結果と同じ

＜考察＞

1. callback3bMain.c の解析をする
   (a) プログラム内にほとんど説明されているので説明されてないところを解析する
   (b) callback3bMain.c には isContainPrintf 関数と main 関数が存在している
   (c) 10 は serchWith 関数の定義を行っている
   (d) 15〜23 は isContainPrintf 関数の宣言である
   (e) 28 では searchWith 関数を isContainPrintf 関数を使って呼び出している
2. callback3bSub.c の解析をする
   (a) プログラム内にほとんど説明されているので説明されてないところを解析する
   (b) callback3bSub.c には serchWith 関数が存在している
   (c) ここでは 18 にコールバック関数が使われている

〜変更後〜

ソース（callback3bMain.c）変更

```
15 int isContainPrintf(char* line) {
16   char* s = strstr(line,"#");
17   return s == line;
18   /* 上記は，以下のコードと同じ結果になる.
```

実行結果（callback3bMain.c callback3bSub.c）変更

変更後の `callback3Main.c` と `callback3Sub.c` の実行結果と同じ

＜考察＞

1. callback3bMain.c の 17 を追加し、18 を変更した
2. ”＃”があるかどうかを判断するために”printf”から”＃”に変更した
3. 16 では line の文字列の”＃”文字を先頭アドレス「s」とおいた
4. 17 では s のアドレスと line のアドレスが等しいか判断している
5. ”＃”が先頭に来ていたら、等しくなる
6. callback3bSub.c は変更しなかった
7. callback3bSub.c は出力する為の関数なので、文字列とは直接的には関係していない

2.3 LEVEL2.3

実際の開発現場では，ライセンスの関係等でライブラリの内部を変更できないことがある．上記の例では，callback4bSub.c をライブラリと考えると，コールバック関数を用いることで抽出時の挙動を実行時に指定することができている．コールバック関数を使わずに，同様のこと（callback4bSub.c に変更を加えずに，実行時に値を見つけたときの挙動を変更する）を実現する方法を考察せよ．

ソース（callback4Main.c）

```
01 #include <stdio.h>
02
03 int hasThree(int item);
04
05 /* メッセージを出力するだけの関数 */
06 void printOnFound(int item)
07 {
08     printf("%d is found.\n", item);
09 }
10
11 /* メッセージではなくアスタリスクの数で値を表す関数 */
12 void indicateOnFound(int item)
13 {
14   int i;
15   putchar('[');
16   for(i=0; i<item; i++) {
17     putchar('*');
18   }
19   puts("]");
20 }
21 /*メイン関数*/
22 int main(void)
23 {
24   int i=0;
25   int j=0;
26   int data1[] = {0,1,2,3,4,5,6,7,8,9,10,-1};
27   int data2[] = {3,6,8,13,5,27,0,6,2,34,63,123,65,-1};
28
29   while(1){
30     int num = data2[i];
31     if(hasThree(num))
32       printOnFound(num);
33     i++;
34     if(num == -1) break;
35   }
36
37   while(1){
38     int num = data1[j];
```

```
39      if(hasThree(num))
40        indicateOnFound(num);
41      j++;
42      if(num == -1) break;
43    }
44
45    return 0;
46 }
```

ソース（callback4Sub.c）

```
01 include <stdio.h>
02
03 //3 の倍数と 3 がつく数字を抜き出すため条件判断を行う関数
04 int hasThree(int item)
05 {
06   return (item % 3 == 0 || item % 10 == 3 || item /10 == 3);
07 }
```

実行結果（callback4Main.c callback4Sub.c）

```
01 3 is found.
02 6 is found.
03 13 is found.
04 27 is found.
05 0 is found.
06 6 is found.
07 34 is found.
08 63 is found.
09 123 is found.
10 []
11 [***]
12 [******]
13 [*********]
```

＜考察＞

1. callback4bMain.c の解析をする

   (a) callback4bMain.c には「printOnFound 関数」、「 indicateOnFound 関数」、「main 関数」が存在している

   (b) 03 は hasThree 関数の宣言である

   (c) 06〜09 は printOnFound 関数である

      i.

ii. 06 より int 型の値を取得してる
iii. 08 より取得した値を出力している

(d) 12〜20 は indicateOnFound 関数である

i.
ii. 12 より int 型の値を取得している
iii. 16〜18 では取得した値の分だけ「*」を出力している

(e) 22〜46 は main 関数である

i.
ii. .24〜27 は必要な引数を宣言している
iii. 29〜35 は pintOnFound 関数で出力するための while 文である
iv. 30 で i 番目の配列の値を num に入れる
v. 31 で num の値が「3 の倍数または 3 がつく数字」であるかを haThree 関数を使用して判定している
vi. 正しいなら 32 が実行され、 printOnFound 関数を使用して出力している
vii. 34 では num（i 番目の配列）の値が「-1」なら while 文を抜ける
viii. 37〜44 は indicateOnFound 関数で出力するために while 文である
ix. 37〜44 は 29〜35 のプログラムと流れは一緒である

2. callback4bSub.c の解析をする

(a) callback4bSub.c は hasThree 関数が存在している
(b) 04〜07 は hasThree 関数である
i. 04 より int 型の値を取得している
ii. 06 より「3 で割った余りが 0」、「10 で割った余りが 3」、「10 で割った答えが 3 」なら 1 を返す
iii. そうでないなら、 0 を返す

3. 上の結果より方法を考える

(a) 変更しない関数は Sub に保存してある
(b) コールドバック関数を使用しない時は関数から関数への受け渡しは「値」または「アドレス」のみである
(c) 関数の中に関数を入れる事が出来ないので、コールドバック関数を使用していた価数は main 関数ですべて行う
(d) Main での関数の宣言を変える必要の時もある

2.4 LEVEL2.4

コールバック関数の有用性について論ぜよ.

Level2.3 より作成したコールドバックを使わない方法では、出力するためのプログラムを増やせば増やすほど、callback4bMain.c の maia 関数のプログラムがとても長くなり、見づらくなる。main 関数内のプログラムでは while 文が似たような動作をしている。これをコールバック関数を使う事によって、while 文を callback4bSub.c に移動する事が出来る。そうすることにより、main 関数内のプログラムが見やすくなり、動作もわかりやすくなる。されに、作成者が main に書く量も大幅に減少する。したがって、関数を多く使う時はコールバック関数を使った方が良い。

2.5 LEVEL2.5

上記の例において，探索条件をコールバック関数として呼び出し側がカスタマイズできるように変更せよ.

ソース（callback4bMain.c）変更

```
01 #include <stdio.h>
02
03 void findNumber(int array[]);
04 void registerCallback(int (*callback)(int) );
05
06
07 int hasThree(int item)
08 {
09   return (item % 3 == 0 || item % 10 == 3 || item /10 == 3);
10 }
11
12 int hasTwo(int item)
13 {
14   return (item % 2 == 0 || item % 10 == 2 || item /10 == 2);
15 }
16
17 /*メイン関数*/
18 int main(void)
19 {
20   /* 配列の終端は-1 である必要がある */
21   int data1[] = {0,1,2,3,4,5,6,7,8,9,10, -1};
22   int data2[] = {3,6,8,13,5,27,0,6,2,34,63,123,65, -1};
23
24
25   registerCallback(hasThree);
26
27   findNumber(data1);
28
29   registerCallback(hasTwo);
```

```
30
31   findNumber(data2);
32
33   return 0;
34 }
```

ソース（callback4bSub.c）変更

```
01 #include <stdio.h>
02
03 static int (*onFoundFuncPointer)(int) = NULL;
04
05 void registerCallback(int (*callback)(int) )
06 {
07   onFoundFuncPointer = callback;
08 }
09
10 void findNumber(int array[])
11 {
12   int i=0;
13   while(array[i] != -1) {
14     if(onFoundFuncPointer(array[i]) ) {
15       printOnFound(array[i]);
16     }
17     i++;
18   }
19   putchar('\n');
20 }
21
22 int printOnFound(int item)
23 {
24   printf("%d is found.\n", item);
25 }
```

実行結果（callback4bMain.c callback4bSub.c）変更

```
01 0 is found.
02 3 is found.
03 6 is found.
04 9 is found.
05
06 6 is found.
07 8 is found.
08 27 is found.
09 0 is found.
10 6 is found.
11 2 is found.
12 34 is found.
13
```

＜考察＞

1. callback4bMain.c の変更したところを示す
   (a) 読み込む関数が int 型なので、04 の　(void (*callback)(int));　→　(int (*callback)(int));
   (b) 07〜10 は探索条件である
   (c) 探索条件を変更できるようにするために Main に移動した
   (d) 12〜15 は新しく作った探索条件である
   (e) 07〜10 の探索条件の数字を変えたプログラムである
   (f) 探索条件を変更するので、25 の　(printOnFound)　→　(hasThree)
   (g) 29〜31 は新しく作った探索条件の出力するためのプログラムである
2. callback4bSub.c の変更したところを示す
   (a) 読み込む関数が int 型なので、03 の　void (*onFoundFuncPointer)　→　int (*onFoundFuncPointer)
   (b) 読み込む関数が int 型なので、05 の　(void (*callback)(int));　→　(int (*callback)(int));
   (c) 探索条件を変更したので、14 の　hasThree(array[i])　→　onFoundFuncPointer(array[i])
   (d) 出力方法は一つだけなので、15 の　onFoundFuncPointer(array[i]);　→　printOnFound(array[i]);
   (e) 22〜25 は出力するための関数を追加した
3. 実行結果より 2 パターンの探索が出来ていることがわかる
4. 変更するプログラムを Main に、変更しないプログラムは Sub に移動するとできる
5. 関数が void 型なのか int 型なのかを注意する必要がある

3 LEVEL3

3.1 LEVEL3.1

glut.c の動作について考察せよ．特に，ユーザの操作に対して GLUT がどのタイミングでどのコールバック関数を呼び出しているのか調べよ．

ソース（glut.c）

```
01 #include <GLUT/glut.h>
02 #include <stdio.h>
03
04 //描画イベントハンドラ
05 void display(void)
06 {
07   printf("display\n");
08 }
09
10 //マウスイベントハンドラ
11 void mouse(int button, int state, int x, int y)
12 {
13   if(state == GLUT_UP) {
14     printf("mouse up\n");
15   }else{
16     printf("mouse down\n");
17   }
18 }
19
20 //キーボードイベントハンドラ
21 void keyboard(unsigned char key, int x, int y)
22 {
23   printf("keyboard(%c)\n", key);
24 }
25
26 int main(int argc, char *argv[])
27 {
28   glutInit(&argc, argv); //glut の初期化
29   glutCreateWindow(argv[0]); //window の作成
30
31   //イベントハンドラの登録
32   glutDisplayFunc(display);  //display コールバック関数の登録
33   glutMouseFunc(mouse);      //mouse コールバック関数の登録
34   glutKeyboardFunc(keyboard);//keyboard コールバック関数の登録
35
36   glutMainLoop(); //イベントループ
37   printf("exit\n");
38   return 0;
39 }
```

実行結果（glut.c）

```
01 display
02 mouse down
03 mouse up
04 keyboard(a)
05 keyboard(0)
06 keyboard(3)
07 )eyboard(
```

< 考察 >

1. glut.c の解析をする
   (a) 05〜09 は display 関数である
      i. ”display ”と表示する
   (b) 11〜18 は mouse 関数である
      i. 13 より受け取った引数が GLUT_UP なら ”mouse up ”と出力される
      ii. そうでないなら ”mouse down ”と出力される
   (c) 21〜24 は keyboard 関数である
      i. 打ち込んだキーを出力する
      ii.
   (d) 26〜39 は main 関数である
      i. コメントにすべて説明されているので省略する
2. 実行結果より考察する
   (a) 01 より glut.c の 32 の関数が呼び出されいてることがわかる
   (b) 02,03 はマウス操作をした
   (c) マウス操作は glut.c の 33 の関数（glutMouseFunc）がが呼び出されている
   (d) 04〜06 はキーボード操作をした
   (e) キーボード操作は glut.c の 34（glutKeyboardFunc）の関数が呼び出されている
   (f) 07 では return(enter) キーを入力した
   (g) return キーが出力されているので見た目が変になってしまっている
   (h) 36 の関数の影響でずっと入力待ち状態になる

3.2 LEVEL3.2

なぜイベント駆動型プログラムが必要とされているのか考察せよ.

1. ユーザーの操作と実行プログラムを密接に関連付けたもので，マウスとアイコンを組み合わせた GUI と相性が良い。
2. イベント駆動型は小規模なモジュールごとにプログラミングできる。
3. 従来のように全体の流れを詳細に分析してからプログラミングする必要はないので，開発の生産性や保守性が向上すると期待されている。
4. プログラムを書く際に必要なイベントハンドラにのみ処理を書けば良いということや、処理の記述をハンドラごとに分けるので、見通しの良いプログラムが期待できる。
5. イベント駆動型プログラミング環境を提供するフレームワーク作者が、フレームワークユーザのプログラムの振る舞いをある程度制御できる。
6. 過度にシステムに負荷を掛ける等の、望ましくないプログラムを減らす効果が期待できる

4　参考文献

1. リトルエンディアン
   http://e-words.jp/w/E383AAE38388E383ABE382A8E383B3E38387E382A3E382A2E383B3.html

2. イベント駆動型プログラミング
   http://itpro.nikkeibp.co.jp/word/page/10000108/
   http://www.weblio.jp/content/イベント駆動型プログラミング