プログラミングゼミ第二回

指導院生：川崎
担当：阿南　坂田　西村

2000年6月6日

目 次

第1章　ポインタ

1.1　ポインタとは

　Cでは，ポインタは最も重要な概念の一つです．しかし，同時に最も難解な部分でもあります．細かいことは，各自，テキストをじっくり読んでもらうことにして，ここでは，たとえ話でポインタのイメージをつかんでもらおうと思います．

人に何かを説明しようとするとき，何かを指し示すために，指示棒を使ったりします．指示棒は，またの名を「ポインタ」といいます．

今，地図を広げて，ある建物や場所を人に説明しようとしています．たとえば,「たかつガーデン」の建物を示したいとき，その建物を指示棒（指示棒がなければ鉛筆や指を使ったりします）で指せばわかります．このように何かを指し示す働きを持つ物のことをポインタといいます．

それでは、プログラム中でポインタを使うとき，それは何を指そうとしているのでしょう？プログラム中で指したい物は，主に，変数です．ところで，変数には名前が付いています．ちょうど地図上の建物に「たかつガーデン」などという名前があるように，変数にも名前が付いています．変数の名前でその変数を示すだけでは，不足なのでしょうか？

建物の例では,「たかつガーデン」に来てください，と言われたとき,「たかつガーデン」の場所を良く知っている人であれば，それだけで済みますが，場所を知らなければ，名前を言われただけでは困ってしまいます．このように，プログラムの世界でも，名前だけでは済まない状況があるのです．

1.2　ポインタの実現

　以上が，ポインタとは何か，と言う質問に対する例え話的解説ですが，それでは，プログラムの世界では，ポインタをどうやって実現しているのでしょうか？当然，指示棒で指し示す，なんてことはできません．また，建物の例に戻りましょう．

先ほど,「たかつガーデン」を地図上で指し示せば，その場所を教えられると言いましたが，実はもっと別な方法もあります．そう,「たかつガーデン」の住所を知らせる方法です．住所は，またの名を,「アドレス」ともいいます.「たかつガーデン」には,「大阪市天王寺区東高津町 7-11」というアドレスが付いています．実際には住所だけを教えられて，目的の場所に行くことは結構大変なのですが，プログラムの世界（あるいはコンピュータの世界）では，非常に簡単な規則でアドレスが付けられているので，問題ありません．

実際，変数はコンピュータの「メモリ」と呼ばれるデータを記憶しておく場所に置かれています．メモリには，たくさんの仕切があって（普通，1バイト毎に仕切られている），その仕切毎にアドレスが順番に通し番号（すなわち，0以上の整数）で付けられています (図 1.1 参照)．したがって，変数の場所を示すためには，その変数が置かれているアドレスを言えば良いことになります．

当然，ポインタは色々な場所を指せなくてはいけませんので，ポインタが指すアドレスは変化できなくてはなりません．また，アドレスは数字ですから，結局のところポインタとは，変化できる数，す

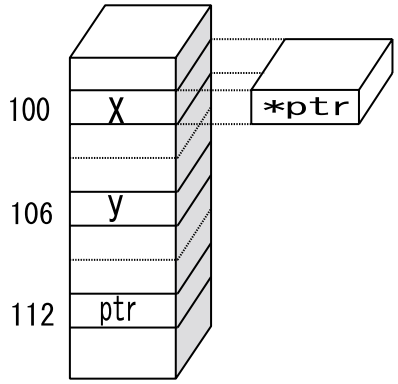

図 1.1: ポインタ変数の説明図

なわち，変数です．ただし，一般の変数との違いは，その保持している数値が，アドレスであるという事です．

最後に，ポインタとは,「変数の配置されているメモリ上のアドレスを保持する変数」です．

1.3 間接演算子・アドレス演算子

主記憶上にあるものはすべて，アドレスというものがついています．一つの番地には，1 バイト（一般的に使われているものとして 8 ビット）入ります．さて，C でアドレスを扱うためのものが，ポインタです．ポインタとは，アドレスを記憶する特別な変数です．ただし変数の前に&をつけると，その変数のアドレスを意味します．次の例を見てみましょう．

```
main()
{
      char c;
      printf("%p\n", &c);
}
```

このプログラムを実行すると変数 c が格納されているメモリのアドレスが表示されますが，この値は C コンパイラや OS がその時にあわせて割りあてるため，実行する環境によって異なります．

4 行目の printf("%p\n",&c); は c のアドレスが 100 番地であったら，100 と表示されます．アドレスを表示させるためには%p を用います．ポインタは，ほかの変数のアドレスを入れておくための変数ですから，もしポインタ変数 p があったとしたら，p=&c と書けば c のアドレスを p に代入できます．ふつうの代入と違うところを注意してください．また*p と書くと，p 番地の内容を取り出すことができます．

char 型の変数を扱うポインタは char *型です．また char *型のポインタでは int 型の変数のアドレスを扱うことはできません．例えば

```
      int n=5;
      char *p=&n;
```

はエラーになります．int 型のアドレスを扱うためには int *型のポインタを使います．

1.4 関数の呼び出しとポインタ

さて，ポインタを使う場面というのはどういう時でしょう．よく使われるのが，関数へのポインタ渡し（call by reference）というものです．

これは，関数を呼び出すときに，変数そのものを渡すのではなく，変数のアドレスを渡すというものです．

変数そのものを渡すものを call by value（値渡し）といいます．ポインタ渡しというものは，アドレスを渡すものですが，なぜそんなことをするかというと，値渡しでは，呼び出した方の値は影響を受けません．これはどういうことかというと，呼び出し側は実引数（argument：アーギュメント）として“ 値 ”を渡し，呼び出される側は仮引数（parameter：パラメータ）として受け取った値の“ コピー ”を使うということです．

```
#include<stdio.h>

void swap(int x,int y)
{
    int temp;

    temp = x;
    x    = y;
    y    = temp;
}

int main()
{
    int a = 5;
    int b = 3;

    swap(a,b);
    printf("A = %d\n",a);
    printf("B = %d\n",b);
    return 0;
}
```

この例は 2 つの int 型の変数を入れ換える関数とそれを使ったプログラムですが，main 関数は a，b という変数の実体を渡すのではなく，その値 5，3 を渡します．呼び出される swap 関数は，その値を x，y の初期値として受け取ります．

swap 関数の中でいくら x と y を交換しても，受け取ったコピーを入れ換えるだけで，本来の a，b を入れ換えるわけではありません．ですから main 関数の a，b は swap 関数を呼び出した後も，それぞれ 5，3 のままです．

それに対してポインタ渡しでは，変更することができます．ふつう変化しない方が，安全なのですが，世の中それだけではできないこともあります．

そのため，アドレスの形で渡してあげると，アドレスそのものは値として呼び出した方を変更できなくても，それが指し示す番地の値は，自由に変えることができるので，結果として，呼び出した方

の変数を変えることができるのです.
それではまず初めにポインタで変数を変更する例を挙げます.

```
#include<stdio.h>

main()
{
    int a = 3;
    int *p = &a;  /*pがaのアドレスを示すようにする*/
        *p = 2;  /*pが指し示す場所を書き換える*/
    printf("a = %d\n",a);
}
```

どんな結果になりましたか？次に簡単な関数のポインタ渡しの例を見てみましょう.

```
#include<stdio.h>

void add_dif(int *add,int *dif){
    int a,b;
    a = *add;
    b = *dif;
   *add = a + b;
   *dif = a - b;
}

int main(){
    int i,j;
    i = 3;
    j = 2;

    add_dif(&i,&j);
    printf("%d %d",i,j);
    return 0;
}
```

まず2行目の `int *add,int *dif` では引数がアドレスなので，ここもアドレスの宣言をします．次に関数の中を順を追って見ましょう.
図 1.2 の一番左の表が関数を呼び出す前（`add_dif` が呼び出される前）の状態としましょう．関数を呼び出すとその隣の状態（`add_dif` に来た直後）になります。0004 番地に`*add` の内容（つまり変数 `i` のアドレス）が格納されます．次にまた隣の表（`a=*add;  b=*dif;` を実行した直後）に移ります．ここで関数内で宣言をした `a` という変数に，`*add` の指す番地の内容（つまり 0001 番地の値）が格納されます．次に`*add = a + b` で `a + b` の値（5）が`*add` の指し示す番地に書き込まれます．最後に関数を抜けて，呼び出した方から `i` の内容を見ると，先ほど書き換えられた 5 という値がでてきます.

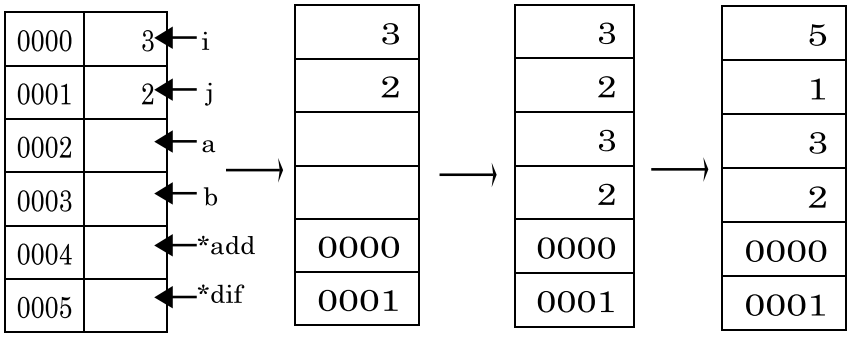

図 1.2: 関数の中身

最後から 3 行目の `add_dif(&i,&j);` で，`i,j` のアドレスを引数にして `add_dif` を呼び出しています．つまり値の 3，2 ではありません．

第2章　ポインタと配列

2.1　ポインタと配列

2.1.1　配列とポインタ

始めに，以下のようなプログラムを実行したらどうなるか考えます．

```
int   a[10];
int  *ptr;
ptr = &a[0];  /* ptr = a も同じ */
```

ptr に a[0] のアドレスを代入するので，ptr は a[0] を指すことになり，*ptr が a[0] のエイリアス[1]となります．

一般に，ポインタ変数 ptr に対して，ptr+i は ptr の指している要素の i 個後ろの要素を示し，ptr-i は ptr の指している要素の i 個前の要素を示します．ここでは，ptr+1 は a[1] を，ptr+2 は a[2] を指すことになります．*はそのポインタ変数が指す変数の実体を表す演算子なので，*(ptr+1) は a[1] のエイリアスとなります．つまり，ptr+i は a[i] のアドレスであり，*(ptr+i) は a[i] のエイリアスです．

また，C 言語では以下のような規則があります．

・配列 a[n] に対して，a[i] を*(a+i)，&a[i] を a+i と書ける

・ポインタ p に対して，*(p+i) を p[i] と書ける

上記において，同じものを表すと以下のようになります．

```
a[0],  *a,       *ptr,       ptr[0]
a[1],  *(a+1),   *(ptr+1),   ptr[1]
    .....
a[9],  *(a+9),   *(ptr+9),   ptr[9]
```

さらにアドレスにおいても同じです．

```
&a[0],  a,    ptr,    &ptr[0]
&a[1],  a+1,  ptr+1,  &ptr[1]
    .....
&a[9],  a+9,  ptr+9,  &ptr[9]
```

このように考えると，ポインタと配列はまったく同じではないかと思えますが，違います．違いとしては，まず変数定義の仕方が以下のように違います．

```
int a[10];   （配列）
int *ptr;    （ポインタ）
```

[1]エイリアス（alias）とは一般的に「別名」「あだ名」という意味の英語で，ここでは，a に対する別のアクセス手段という意味で利用されています．

さらに，ptr は int 変数へのポインタです．また，一般的に配列名 a もポインタと同様に取り扱うことができますが，**aは変数ではなく定数です**[2]．ですから，ptr はアドレスを持ちますが，a は定数なのでアドレスを持ちません．そのため，配列に対して以下のように事は行うことはできません．

```
/* 例1 */
  int a[10];
  int b[10];
      a = b;        /* エラー */

/* 例2 */
  int *p;
  int b[10];
      p = b;        /* 正常に動作する */

/* 例3 */
  int a[10];
  int **pp;
      pp = &a;      /* エラー */

/* 例4 */
  int *p;
  int **pp;
      pp = &p;      /* 正常に動作する */
```

以上の事をまとめると，このようになります．

- 配列 `int a[10];`
  配列は `a[0]`〜`a[9]` の固定的な領域をもつ．`a` は先頭要素 `a[0]` へのポインタ (`&a[0]`) となる．
- ポインタ `int *p;`
  `int` へのポインタ `p` は任意の `int` 型変数を指すことができる．`p` の指す前後の要素を `[]` 演算子を適用することにより，`p[i]` のようなアクセスが可能である．
- C言語では添え字というのは，基準となるポインタからの差，すなわちいくつ離れているかを表す整数である．

2.1.2 関数間の配列の受け渡し

はじめに，以下のプログラムを考えてみます．

[2] 1 と int a に対して，1 は変数 a と同等に取り扱うことができますが，1 は定数です．

```
void func(int *a)
{
    a[3] = 5;
}
int main(void)
{
    int x[10];
    func(x);
    return 0;
}
```

main 関数は，func(x); のように実引数として x を渡しています．x は&x[0] のことです．つまり，配列の先頭要素のアドレスを渡しています．また，void func(int *a) の a は，int 変数へのポインタ型である．a には受け取った値，すなわち x[0] のアドレスが渡され，x[0] を指すことになります．ここで，a[3] に 5 を代入していますが，a[3] は*(a+3) のことです．a は main 中の x のエイリアスとして機能するので，結果的に a[3] は x[3] と等価になります．

次に，次頁の 2 つのプログラムについて考えてみます．例 1，例 2 はそれぞれ関数間の配列の受け渡しを行っていますが，構造が多少違います．これによってどのような違いがあるのでしょうか．結論としては，出力が以下のように違います．

―例１―

```
#include < stdio.h >


void func(int *a)
{
    int i;

    a[0] = 10;
    a[1] = 8;
    a[2] = 12;
    printf("¥nfunc 関数¥n");
    for (i = 0; i < 10; i++)
        printf("% 4d",a[i]);
}



int main (void)
{
    int i;
    int x[10];

    for (i = 0; i < 10; i++)
        x[i] = i;
    printf("¥nfunc 関数を呼び出す前¥n");
    for (i = 0; i < 10; i++)
        printf("% 4d",x[i]);

    func(x);

    printf("¥nfunc 関数を呼び出した後¥n");
    for (i = 0; i < 10; i++)
        printf("% 4d",x[i]);
    return 0;
}
```

―例２―

```
#include < stdio.h >
#include < string.h >

void func(int *a)
{
    int array[10];
    int i;

    memcpy(array, a, sizeof(int) * 10);

    array[0] = 10;
    array[1] = 8;
    array[2] = 12;
    printf("¥nfunc 関数¥n");
    for (i = 0; i < 10; i++)
        printf("% 4d",a[i]);
}

int main (void)
{
    int i;
    int x[10];

    for (i = 0; i < 10; i++)
        x[i] = i;
    printf("¥nfunc 関数を呼び出す前¥n");
    for (i = 0; i < 10; i++)
        printf("% 4d",x[i]);

    func(x);

    printf("¥nfunc 関数を呼び出した後¥n");
    for (i = 0; i < 10; i++)
        printf("% 4d",x[i]);
    return 0;
}
```

例1では main 関数の x は配列であり，func 関数の a はポインタという違いはありますが，a[i] と x[i] は同じモノを指していますので，func 関数で a[0] などの要素を変更すると，main 関数の x[0] も書き換えられてしまいます.

main 関数の配列 x そのものを変更せずに func へ値を渡す場合は，どのようにすればよいのかを例2で示しています．memcpy(array,a,sizeof(int)*10) は，a から array に sizeof(int)*10 個分のメモリをコピーします．sizeof(int) は int 型の変数に必要なメモリのサイズを表しています．そしてそれが 10 個分あるので sizeof(int)*10 個分のメモリをコピーするわけです．この場合も a, array 共にアドレスを表しています.

これと似たものに memmove があります．これは多少 memcpy との違いはありますが，アドレスを指定してコピーを行うという意味では memcpy と同様だと考えていいでしょう.

func 関数を呼び出す前	func 関数を呼び出す前
0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9
func 関数	func 関数
10 8 12 3 4 5 6 7 8 9	10 8 12 3 4 5 6 7 8 9
func 関数を呼び出した後	func 関数を呼び出した後
10 8 12 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9

memcpy	
形式	`void *memcpy(void*dest,const void*src,size_t count);`
プロトタイプ	`<string.h>`
機能	memcpy は, src で指される領域の先頭から dest で指される領域へ count バイトをコピーする．src と dest の領域が重なっている場合の動作は保証されない.
戻り値	dest をそのまま返す.

memmove	
形式	`void *memmove(void*dest,const void*src,size_t count);`
プロトタイプ	`<string.h>`
機能	memmove は, src で指される領域の先頭から dest で指される領域へ count バイトをコピーする．src と dest の領域が重なっていても正常に働く.
戻り値	dest をそのまま返す.

2.2 文字列とポインタ配列

2.2.1 文字配列と文字へのポインタ

文字列とは文字の配列です．例えば,

```
printf("A=%d", a);
```

があるとき, ””で囲まれた部分が文字列で，文字列定数 (string constant) や文字列リテラル (string

literal) と言います.

また, "A=%2d"の式にも値があります. 一般に”xxxx”という式は char へのポインタ型をもち, その文字列の先頭アドレスを値として持ちます.

配列の初期値つきの定義は,

```
char c[]={'a', 'b', 'c'};
```

となります. 初期値つきの配列の定義では, その大きさが自動的に計算され, c の大きさは 3 となります. C 言語では, 文字列には最後にヌル文字 ('\0') をつけて処理します. これにより, 関数が文字列の終わりを見つけることができるようになります. よって上の式では, c には単なる文字 3 個の配列が入力されているだけで, 文字列として扱いたいのであれば,

```
char str1[]={'a', 'b', 'c', '\0'};
```

と明示的にヌル文字を付加しなくてはなりません. しかし, これでは面倒くさいので特別に

```
char str1[]="abc";
```

という表記を認めています. 一方,

```
char *str2="def";
```

という定義がありますが, これは文字列”def”の先頭アドレスである d のアドレスを char 型のポインタ変数 str2 に代入しています. この違いは図 2.1 を参照してください. ここで, 細かいお話をすると, 上の記述は正しくありません. 次のコード

```
(1)   char *p="abcd";
         p[0]='q';
```

は, コンパイルは通ります. つまり, 現行の C コンパイラでは正しいことになります. しかし, 実行時には強制終了 (segmentation fault) させられます. これは, ”abcd”という文字列定数に対し, ポインタ変数 p を用いて参照することはできますが, 書き換えることはできないからです. 一方,

```
(2)   const char *p="abcd";
         p[0]='q';
```

は C コンパイラでは通りません. 文法エラーとなります. const は定数を意味し, これにより, p は char 型定数へのポインタであることを示しています. (1) のように, コンパイルはできてもいざ実行の段階でエラーになってしまうことを未然に防げます. 今後, (1) のようなコードをエラーとするコンパイラが多くなると思われます. また, 上で出てきた segmentation fault とは一体何なのでしょうか. OS の機能の 1 つにメモリ保護があります. OS は, メモリを保護するために, 領域をセグメントで区切り, それぞれに, RW 可能, R 可能, W 可能, RW 不能などの属性を付けています. このうち文字列定数など, 定数は R 可能, つまり Read のみ可能の領域に格納されるので, p[0]='q' のように代入する, つまり Write することはできません. したがって, segmentation fault が生じました.

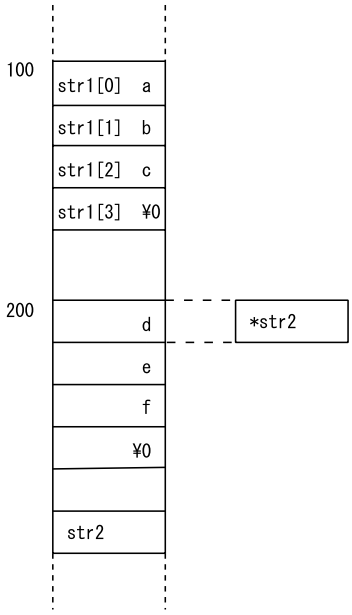

図 2.1: 文字列へのポインタと文字列の配列

column

a[3] と 3[a] はどちらも同じことを意味しています. [] のオペランドの一方は,「ある型 T へのポインタ」であり, 他方は整数型であればいいのです. それでは,

```
int main()
{
    char c, d;
    c="ABC"[0];
    d=1["DEF"+1];
}
```

において, c と d はどんな値を持つのでしょうか？考えてみてください.
ちなみに答えは c=A, d=F です.

2.2.2 文字列と記憶寿命

文字列”ABC”へのポインタを返す関数を作りましたが, うまくいきません. どこがだめなのでしょうか.

```
#include <stdio. h>

char *abc(void)
{
    char p[]="ABC";
    return p ;
}
int main()
{
    printf("%s\n", abc());
    return 0 ;
}
```

答えを言ってしまうと, abc 関数の配列 p は自動的な記憶寿命を持ち, よって, abc 関数の中だけで生きています. その配列の先頭要素へのポインタを返しても, 意味がないわけです. それでは, どうすればいいかというといろいろと方法はあるのですが, 1 つの例として以下のプログラムを参考にして下さい.

```
#include <stdio.h>

char *abc(char *buf)
{
    strcpy (buf, "abc");
    return buf ;
}
int main()
{
    char buf[4];
     printf("%s\n", abc(buf));
    return 0 ;
}
```

2.2.3 ポインタの演算

次に, ポインタの演算についてですが, とりあえず次のプログラムを実行してみました.

```
/* ctest.c */
int main()
{
    int x[200];
    int *p, *q;
    p=&x[0];        /*  p = x;  でも良い  */
    q=p+200;
    printf("q-p=%4d\n", q-p);
    printf("(unsigned)q-(unsigned)p=%4d\n", (unsigned)q-(unsigned)p);
    return 0;
}
```

実行結果は以下のようになりました.

```
hide@zodiac:~$ gcc -o ctest ctest.c
hide@zodiac:~$ ./ctest
sizeof(int)=4
q-p=200
(unsigned)q-(unsigned)p=800
```

なぜこのようになるのか意味を考えてみてください.
ポインタの演算は, そのポインタの指す型の大きさを単位として行われます. よって,

```
char        *cp;
int         *ip;
double      *dp;
```

に対して,

```
  cp++;     /*1 つ後ろの要素を指す.cp の値は sizeof(char) 増える*/
  ip++;     /*1 つ後ろの要素を指す.ip の値は sizeof(int) 増える*/
 dp+=5;     /*5 つ後ろの要素を指す.dp の値は sizeof(double)*5 増える*/
*(ip+6)     /*6 つ後ろの要素を指す. すなわち, ip+sizeof(int)*6 の要素
```

のようになります.

2.2.4 ポインタと配列の比較

以下のプログラムを見てください. chang_a 関数と change_p 関数はどちらも同じことをする関数です.
配列とポインタの表記の違いを見てもらうために 2 つの関数を作りました.
これらの関数は, 入力された文字列の各文字に対して, A → Z, B → Y, C → X,.... と変換する関数です.

```
#include <stdio.h>
#include <string.h>

char *change_a(char *buf, const char *a)
{
int i, n;

n = strlen(a);
for (i = 0; i < n; i++){
if ('A' <= a[i] && a[i] <= 'Z')
           buf[i] = 'Z' - a[i] + 'A';
       else if ('a' <= a[i] && a[i] <= 'z')
           buf[i] = 'z' - a[i] + 'a';
       else
           buf[i] = a[i];
   }
    buf[n] = '\0';
    return buf;
}

char *change_p(char *buf, const char *a)
{
    char *p;

    for (p = buf; *a != 0; a++, p++){
        if ('A' <= *a && *a <= 'Z')
           *p = 'Z' - *a + 'A';
        else if ('a' <= *a && *a <= 'z')
           *p = 'z' - *a + 'a';
        else
           *p = *a;
   }
   *p = '\0';
    return buf;
}

main()
{
    char str[80], out[80];

    printf("moji ? ");
  //scanf("%s", str);
    fgets(str, 80, stdin);
  //printf("%s\n", change_a(out, str));
    printf("%s\n", change_p(out, str));
    return 0;
}
```

ここでは, str[80] に文字列を入力する方法に scanf は使わず, fgets を使っています. これは, 標準入力 (キーボード) からどのような入力がなされるかはわからないので, 予期せぬ入力をされるとコンピュータが落ちる危険があるからです. 具体的には, 80 文字を超えて入力される場合ですが, fgets だと読み取りの最大文字数が指定できるので, 安全に読み取ることができます. また, scanf だと, スペースが入力されると, それを区切りと見てしまうため, スペースを含んだ文字列を入力できません. つまり, `This is a pen.` のような文を入力できないことになります.

2.2.5 malloc と free

とりあえず, 以下のプログラムを見てください.

```
#include <stdio.h>
#define MAX 50000

int main()
{
    int a[MAX], i, size;

    scanf("%d", &size);
    for( i=0; i<size; i++)
        a[i] = 0;
    return 0;
}
```

これは, 単に初期化するプログラムです. ここで注目してほしいのは, 配列 a の最大の要素数は MAX としてマクロで定義されています. しかし, もし size の大きさが MAX を超えてしまうとコンパイル時エラーになってしまいます. また, 結果的に配列の要素の数が, 4 つ, 5 つだけしか必要なかったとしたら, 50000 個も配列要素をとることはメモリを無駄に使ったことになります. ではどうすればいいのでしょうか.

そういった場合, malloc などの関数を使うことによって, 動的 (ダイナミック) に記憶領域を確保することができます. malloc は, ANSI 規格では, `<stdlib.h>`で提供されています. 以下のプログラムを見てください.

```
#include <stdio.h>

int main()
{
    int *a, count;

    printf("int 型の変数がいくつ必要ですか？");
    scanf("%d", count);
    a = (int *)malloc(sizeof(int)*count);
    if(p == NULL)
        printf("確保に失敗しました");
    for (i=0; i<count; i++)
        a[i] = 0;
    free(a);
    return 0;
}
```

ここで, メモリの動的確保を行うのは,

```
p=(int *)malloc(sizeof(int)*count);
```

の箇所です. malloc に必要なメモリ量を第 1 引数として渡すと, 返り値として, 確保されたメモリ領域のポインタが返されます. また, エラー時 (メモリが足りない場合など) には, NULL が返されます. int 型の変数 1 つを記憶するのに必要なメモリ量は sizeof(int) で得ることができます.
従って, count 個の int を記憶するのに必要なメモリ量は,

```
sizeof(int)*count
```

で得られます.

以下の malloc のリファレンスを見てください. malloc 関数が返す型は, `void *`となっていますが, これは, 返す値の型が任意の型へのポインタを示しています.

また, malloc でメモリの確保ができることはわかりましたが, 確保しっぱなしではメモリの容量にも限界があるため, 空きがなくなってしまいます. 確保したものは使わなくなったら返さなくてはなりません. そのための関数が free 関数です. 詳細はリファレンスを参照してください.

malloc	
形式	`void *malloc(size_t`[3]`size);`
プロトタイプ	`<stdlib.h>`
機能	`malloc` は, 大きさが `size` バイトのオブジェクト (記憶領域) の領域を確保する.
戻り値	領域確保が成功したときは, 確保された領域の先頭へのポインタを返す. 領域確保に失敗した場合は, `NULL` を返す.

[3]size_t は sizeof 演算子が返す値の型で, 符号なし整数を示します. 通常は, `unsigned int` と同等の意味です.

free	
形式	`void free(void *p);`
プロトタイプ	`<stdlib.h>`
機能	`free` は p が指す領域を解放する．ただし, p が `NULL` のときは何もしない．p は`malloc` によって, 以前割り当てられた記憶領域へのポインタでないといけない．

文字列ではポインタがよく利用されているので, 参考程度に, 代表的な文字列処理関数を載せておきます.

strcpy	
形式	`char *strcpy(char *s, const char *t);`
プロトタイプ	`<string.h>`
機能	strcpy はポインタ s で示される領域に t で示される文字列の実体をコピーします．転送される文字列の終わりは必ず`'\0'` で終わっていなければなりません．strcpy は`'\0'` も含めてコピーします．したがって，コピー先の領域はコピーされる文字列の長さ+1 以上のサイズが確保されていなければなりません．

strcat	
形式	`char *strcat(char *s, const char *t);`
プロトタイプ	`<string.h>`
機能	strcat はポインタ s で示される領域に格納されている文字列の後に，t で示される文字列をコピーします．s で示される領域は文字列 t を連結できるだけのサイズが確保されていなければなりません．

strcmp	
形式	`int strcmp(const char *s, const char *t);`
プロトタイプ	`<string.h>`
機能	strcpy は文字列 s と t を比較し，その結果で s が t よりも大きければ正，s と t が等しければ 0，s が t よりも小さければ負の値（int 型）を返します．文字列の比較はいわゆる辞書順に大小関係 (`"a" < "ab" < "b" ...`) が決まります．

strlen	
形式	`size_t strlen(const char *s);`
プロトタイプ	`<string.h>`
機能	strcpy は文字列 s の長さ（`\0` は含まない）を返します．