● 今後の予定

【7月7日】

- 再帰的関数呼び出し

【7月14日】

- ファイル入出力
- main の引数の意味

【7月21日？】

- （予備）
- その他
- 学期末課題の説明

なお，７月２１日は「教員採用試験がある」とのことなので，可能ならば他の「補講日」に授業を行います．**７月２３日（金）午後**または**７月２８日（水）午前**を考えています．（どちらがよいか，意見を下さい．）

● 今日の実習

【サンプルプログラム】

★ ex12-1.c **最も単純かつ基本的なポインタの例**.

1. “int *pn, *pm” で pn と pm という２つの int 型オブジェクトへのポインタを作成．（以後簡単のため，「int 型へのポインタ」呼ぶ.）
2. オブジェクトに対して & をつけると，そのオブジェクトのアドレスを取り出すことができる．この & を「**アドレス演算子**」と呼ぶ.
3. ポインタに対して * をつけると，ポインタの指し示す先の値を参照できる．この * を「**間接演算子**」と呼ぶ.
4. “pn = &n” で pn が n を指し示すように，pn に n のアドレスを代入.
5. “*pn = 2” によって，pn の指し示す先の値に 2 を代入する.
   この時点では，pn は n を指し示しているので，n に 2 を代入したことになる.
6. “pm = pn” によって，pm には pn が指し示すもののアドレスが代入される.
   この時点では，pn は n を指し示しているので，pm には n のアドレスが代入され，pm は n を指し示している.
7. printf 関数で “%p” を指定すると，対応するポインタのアドレスが表示される．（具体的な値には意味はほとんど無いと考えてもよい.）

★ ex12-2.c **（１次元）配列とポインタとの対応**.

C 言語の書籍には「**C では配列とポインタは同じもの**」と書いてあることが多いが，その意味を考えてみよう.

1. “int a[10]”, “char s[11]” で，それぞれ int 型，char 型の要素を持つ配列を作成．また，“int *pa”, “char *ps” で，それぞれ int 型，char 型へのポインタを作成.
2. “ps = s” によって ps に配列 s の先頭のアドレスを代入.
   ex12-1.c では，ポインタにアドレスを代入するためにアドレス演算子を利用したが，**配列の場合には状況が異る**.
   「**オブジェクトが配列の場合には，識別子はその配列の先頭アドレスをあらわす**」ので，“s” そのものには配列 s の先頭のアドレスが入っている．それは，s[0] のアドレスそのものである.
   したがって，“ps = s” と “ps = &s[0]” は等価な操作である.

3. char 型のポインタ ps に対して ps+1 は「**次のアドレス**」をあらわす.
   したがって，ps が s[0] を指し示しているとき，ps+1 は s[1] のアドレスをあらわす．特に “*(ps+i)” は “s[i]” の値を参照している.
   逆に char 型へのポインタ ps を用いて，ps が char 型の配列 s を指し示しているときには，*(ps+i) の代わりに ps[i] と書いてもよい.
4. int 型のポインタ pa に対して pa+1 は「**次のアドレス**」をあらわす.
   と，ここまでは char 型と同じであるが，ポインタが示す型が異ると「**次のアドレス**」の中身が異ってくる．すなわち，「**次のアドレス**」とは，そのポインタが示す型のオブジェクトの意味で「次」を意味する.
   したがって，pa が a[0] を指し示しているとき，pa+1 は a[1] のアドレスをあらわす．特に “*(pa+i)” は “a[i]” の値を参照している.
   つまり，pa+1 はそれが示す型のバイト数先のアドレスである.
5. 実際には配列の要素参照 a[i] は内部では *(a+i) として実現される．この意味で「**配列とポインタは同じもの**」である.

★ ex12-3.c **ポインタを引数にする関数**

1. 最初に定義している変数は，大域変数 a と main 関数内の局所変数 b である.
2. 通常 “int foo_0(int a)” によって関数内部で実引数の値を変更しても，呼び出し側に戻ったとき，その変更は反映されない.
   実際，呼び出し側（main）と foo_0 での変数のアドレスをみると，全く異ったものであることがわかる.
3. ところが “int foo_1(int *a)” によって関数内部で仮引数の値を変更すると，呼び出し側に戻ったとき，その変更が反映されている.
   実際，呼び出し側（main）と foo_1 での変数のアドレスをみると，呼び出し側での実引数のアドレスと関数内部での仮引数のアドレスが一致している．（当たり前のことである.）
   このことを「**関数副作用**」と呼ぶ.
4. なお，大域変数と局所変数は，それらが格納されるアドレス空間が全く異っていることに注意しよう.

★ ex12-4.c **配列を関数に渡す（配列要素（文字列）のコピー）**

1. ここでは，文字列 a をコピーする数種類の方法を考える.
2. 単純に “for(i=0;i<len;i++) *(b+i) = *(a+i) ;” によって文字列長 len の文字列 a を b にコピーすることができる.
3. 文字列は “Null Terminate” であることを用いれば，“while((*(c+i) = *(a+i))) i += 1 ;” によっても文字列 a を c にコピーすることができる.
4. “char *_strcpy(char *t, char *s)” で定義した関数には，その実引数として char 型の配列を渡すことができる．なぜなら，char 型の配列 a を実引数に用いることにより，関数には a の先頭アドレスが渡されるため，関数側からは，実引数が配列であってもポインタであっても，実際には「アドレスが入っている」という意味では区別はない.
5. “char *_strcpy(char *t, char *s)” で定義した関数内部で，
   “while((*(t+i) = *(s+i))) i += 1 ;” とすれば，文字列 s を t にコピーすることができる.
6. 次に，“char *strcpy(char *t, const char *s)” で定義した関数内部では，
   “while((*t++ = *s++)) ;” によって文字列 s を t にコピーすることができる.

ここで “t++” とは t の示すアドレスをインクリメントすることであり, 実際には, t の「次のアドレス」を t に代入することとなる. （これが文字列コピーの最も標準的なコードである）

7. これらの関数は「コピーした文字列の先頭アドレス」という char 型へのポインタを戻り値にしている.
8. “const char *s” の “const” とは, 「定数」を意味し, これがついたオブジェクトは変更が不可能になる. この場合は指し示すポインタの中身の変更ができなくなる. しかし, ポインタ自身の変更は可能である.
9. ところが, “char a[N], e[N]” と定義した関数内で “while((*e++ = *a++)) ;” とすると「文法エラー」が発生する. これがなぜかを次の例で考えてみよう.

★ ex12-5.c **ポインタと配列の違い**

1. ここでは,

```
char  a[] = "message" ;
char  *p  = "message" ;
```

という２つの方法で文字列を定義している. これらの定義の違いを調べる.

2. p に関しては,

```
for(i=0;*(p+i);i++) printf("%s\n", p+i) ;
for(;*p;p++) printf("%s\n", p) ;
```

のいずれの操作も可能である.

3. ところが a に関しては,

```
for(i=0;*(a+i);i++) printf("%s\n", a+i) ;
```

は可能であるが,

```
for(;*a;a++) printf("%s\n", a) ;
```

は “a++” で「文法エラー」となる.

4. これらの２つの定義の違いは以下の通り.

```
char  a[] = "message" ;
```

は「要素数 8 の char 型の配列」を定義しているので, a で示されるアドレスから 8 バイトを使って文字列 “message” が格納される. この a で示されたアドレスは「**固定された**場所」である.
一方, p は

```
char  *p  = "message" ;
```

と定義されているが, これは, char 型へのポインタ p を定義して, 「ある場所に格納された文字列リテラル（この場合は “message”）のアドレス」を p に代入する（このアドレスで初期化する）という意味を持つ文である.

5. これらの定義の違いにより, p のインクリメントは可能であるが, a のインクリメントが許されない理由は明らかであろう. この意味で「**C では配列とポインタは同じ**」という言葉を真に受けるとハマってしまうことになる.

★ ex12-6.c **値の入れ替えを行う関数**

ここでは, ポインタを使った有名な例として, 「値の入れ替えを行う関数」“swap” を作ってみよう.

1. 変数に格納された値を入れ替えるためには, 次のようなコードを用いることが多い.

```
c = a ; a = b ; b = c ;
```

（a, b, c ともに同じ型のオブジェクトと仮定している.）

2. この例を単純に関数に入れたものが

```
void not_swap_char(char a, char b)
```

である. このままでは, ex12-3.c で見たように, 変数の値の入れ替えは実現できない.

3. char 型の変数の値を入れ替えるには, 仮引数として char 型へのポインタを用いなければならない.

```
void swap_char(char *a, char *b)
```

4. 次に int 型の変数の値を入れ替えるためには, 仮引数として int 型へのポインタを用いなければならない.

```
void swap_int(int *a, int *b)
```

5. この例で, “swap_char” を用いて int 型の変数の値を入れ替えてみよう. その呼び出し方法は

```
swap_char((char *)&n, (char *)&m) ;
```

となる. ここで, n, m は int 型のオブジェクトであるので, 単純に &n, &m とすると, これらは int 型のオブジェクトのアドレスであるため, 値の型としては int 型へのポインタとなる. そのため, 仮引数定義との型の不整合が生じる. これを回避するため, 明示的に (char *) によって char 型へのポインタに型変換している.
なお, どのような型へのポインタであっても, そのオブジェクトは**アドレスを代入するために必要かつ十分な長さの領域**となるため, ポインタの型変換は自由に行うことができる.
しかし, インクリメントやポインタ演算を行うため, 型を指定しなければいけない.

6. さて, “swap_char” を用いて int 型の変数の値を入れ替えると, 正しい結果を得ることができない.
これは, “swap_char” 内部で *a が char 型へのポインタとなっているため, c = *a の代入では「右辺値」として a の示すアドレスの先頭バイトのみが参照されるためである.
7. 逆に “swap_int” を用いて char 型の変数の値を入れ替えると, 「実行時エラー」(Bus Error) が発生する可能性がある. （いつでもそうなるかどうかはわからない.）
これは, （難解な話だが）ワード境界以外の境界から複数バイトを読み出そうとする場合に生じるエラーである.
8. しかし, 「型が違っても変数の値の入れ替えくらいはできなきゃ大変！」と思うのが自然であり, それを解決するための方法が次の例である.

★ ex12-7.c **汎用データポインタ**

1. 「型が違っても変数の値の入れ替えくらいはできなきゃ大変！」ということで, これを解決するためには, 関数に対して入れ替えたい変数へのポインタだけでなく, その変数のサイズも渡してしまえばよい.
2. 具体的には

```
void swap(void *a, void *b, size_t size)
```

という形で関数を作成する.
ここで出てきた「void *」を**汎用データポインタ**と呼び, 「どんな型へのポインタも受け取ることができるポインタ」である.

3. 汎用データポインタを使う際には, 関数側で「インクリメント」がどのような意味があるかを知るために, char 型へのポインタ（最もサイズの小さなオブジェクトへのポインタ）に変換しなければならない.

★ ex12-8.c sizeof **について**

1. sizeof 演算子は, 被演算子に「型名」を取るときには, その型のオブジェクトが利用するバイト数を返す.
   被演算子に「オブジェクト」を取るときには, そのオブジェクトが利用するバイト数を返す.
   これを用いると, 配列の要素数を得ることができる.
2. しかし, 一旦関数に渡ってしまうと, 実引数は配列であっても, 仮引数定義がどのように書いてあっても, 関数内部からはポインタにしか見えないため, sizeof で配列の要素数を得ることはできない.

★ ex12-9.c **ポインタの配列とポインタへのポインタ**

1. “char *str[3]” と定義すると, 「char 型へのポインタ」型の要素数 3 の配列が定義できる.
2. “char **pstr” と定義すると, 「char 型のポインタ」へのポインタが定義できる.
3. 上で定義した char *str[3] の str[0] は char 型へのポインタであるので, str は「char 型へのポインタ」へのポインタとなる. すなわち, char **pstr は「文字列への列」と理解することができる.
4. “char *str_array[] = {"abc", "defg", "hijkl"}” と定義すると, 「char 型のポインタ」型の配列が定義できる. この時, “str_array[i]” はこの配列の i 番目の要素である文字列へのポインタとなる.
5. main 関数の仮引数定義にある argv は「コマンドライン引数」へのポインタであり, その要素数が argc に格納されている.

★ ex12-10.c **文字列に含まれる文字の最初の位置を見つける**

1. 関数 “char *strchr(const char *s, int c)” は標準関数に含まれるもので, 文字列 s の中から c に一致する最初の文字のポインタを返す.
2. もし, 一致するものが無いときには NULL を返している. “NULL” とは「何も指し示さない」という特別なポインタである.
3. 関数の戻り値が “char *” となっているので, “p = strchr(s,'i')” として得られた p をそのまま printf 関数に使うことができる.
4. “p-s” という「謎」の「ポインタの減算」は, p の位置が s の先頭から何文字目かを表している.
   「ポインタの減算」は「それらのポインタが同じオブジェクト内を指し示しているときに限り」意味を持つ.

【課題】 以下の課題では, 必要に応じて関数を作ってプログラムを書くこと. また, 「標準関数」は自由に利用してかまわないが, 当然, その問題自身の目的になっている標準関数は利用してはならない. （必要なものは stdio.h, ctype.h, strings.h に含まれるものだけである.）

なお, exercise12-1 から exercise12-8 までの難易度は各自で判断してください. （多少難しいものも含まれています.）

★ exercise-12-1 次の仕様をみたす関数をつくりなさい.

【形式】

```
int ext_gcd(int a, int b, int *x, int *y)
```

【機能説明】

２つの正の整数 a, b に対して, 拡張されたユークリッドの互除法を用いて

$$ax + by = \gcd(a, b)$$

をみたす x, y を, $xy \neq 0$ をみたすもので, 一組求めます. 戻り値は a と b の最大公約数です.

★ exercise-12-2 unsigned long 型の値 n を 1970 年 1 月 1 日 午前 0 時 0 分 0 秒からの累積秒数と考えて, n を与えたときに, それが year 年 month 月 day 日 hour 時 minute 分 second 秒となる year, month, day, hour, minute, second を求める関数を書きなさい. ただし, hour は 24 時間表示とします.

★ exercise-12-3 unsigned long 型の値 n を 1970 年 1 月 1 日 午前 0 時 0 分 0 秒からの累積秒数と考えて, n を与えたときに, 次の形式の「日付文字列」へのポインタを返す関数を書きなさい. 形式は “WWW␣MMM␣DD␣HH:MM:SS␣YYY” ただし,

- DD は日付を示す２桁の数値. （01 などとします. 以下同様）
- HH は時間を示す２桁の数値.
- MM は分を示す２桁の数値.
- SS は秒を示す２桁の数値.
- YYYY は年を示す４桁の数値.
- MMM は月を示す３桁の文字. １月からそれぞれ Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec とします.
- WWW は曜日を示す３桁の文字. 日曜日からそれぞれ Sun Mon Tue Wed Thr Fri Sat とします.

曜日を計算するために 1970 年 1 月 1 日は木曜日であることを利用してかまいません.

【ヒント１】 「日付文字列」へのポインタを返すためには, このフォーマットに従う文字列の長さが 24 文字であることを利用して, 関数内部で定義する静的変数

```
static char str[25] ;
```

へのポインタを返せばよい.

【ヒント２】 「日付文字列」を作成するためには, 標準関数 sprintf を使うのがよい.

★ exercise-12-4 以下の標準関数を書きなさい.

☆ exercise-12-4-1 strcat

☆ exercise-12-4-2 strncat

☆ exercise-12-4-3 strncpy

☆ exercise-12-4-4 strcmp

☆ exercise-12-4-5 strncmp

☆ exercise-12-4-6 strspn

☆ exercise-12-4-7 strcspn

☆ exercise-12-4-8 strstr

☆ exercise-12-4-9 strpbrk

【注意１】 多分「易しい順」で「依存関係のある順」に並んでいるはず.

【注意２】 これらを書くためには, 前のものを利用すると簡単になるはず.

【注意３】 それぞれの標準関数の仕様は「オンラインマニュアル」を参照のこと.

★ exercise-12-5 次の仕様をみたす関数をつくりなさい.

【形式】

```
int strrstr(const char *s1, const char *s2)
```

【機能説明】

文字列 s1 の中で文字列 s2 と最後に一致する部分の先頭アドレスを返します. もし, 一致する部分が無ければ NULL を返します.

【注意】 この関数は標準関数の中には含まれていません.

★ exercise-12-6 次の仕様をみたす, 文字列をトークン分解する関数を作りなさい.

【形式】

```
char *_strtok(char *t, const char *s, const char *d)
```

【機能説明】

文字列 s を「区切り文字の集合」 d によってトークン分解した, 最初のトークンを t に返し, 戻り値として, s の中でトークン t の「次の文字」へのポインタを返します. もし, トークンが一つも見つからなければ NULL を返します. そのとき t の内容は保証されません.

さらに, この関数を利用して, 文字列をトークン分解し, 各トークンを順に出力するプログラムを書きなさい.

【注意１】 この関数は標準関数の中には含まれていません. また, 標準関数にある strtok とは仕様が異ります.

【注意２】 「**トークン**」 (token) とは, 指定された「区切り文字」によって区切った文字列のことをいいます. また, 与えられた文字列を「**トークン分解**」 (**token decomposition**) するとは, その文字列をトークンに分解することを言います.

【例】

区切り文字が「英数字以外」であるとき, 文字列 “This␣is␣a␣test.” をトークン分解すると, “This”, “is”, “a”, “test” の４つのトークンに分解されます.

【ヒント】

ここまでに作成した文字列に関する関数をうまく組み合わせればよい.

★ exercise-12-7 与えられた文字列中に含まれる「単語」の長さに関する頻度表を出力するプログラムを書きなさい. すなわち n 文字長の単語がいくつ含まれるかを表示するプログラムを書きなさいということ. ただし, 文字列中に含まれる単語の最大長が 10 以上となるものは「10 文字以上」で一括して扱います. また, 単語とは「英文字以外の文字」を区切り文字とした「トークン」のことと定義します.

★ exercise-12-8 最大 1024 文字の文字列中に「１０進整数」をあらわす文字列が, 最大 100 個入っているとする. この時, その文字列を受け取って, 文字列中に含まれる「１０進整数」を int 型の配列に格納するプログラムを書きなさい.

ただし, 文字列中の区切り文字は「１つ以上の空白文字」し, 「１０進整数」をあらわす文字列とは, 「先頭に + または - の「符号文字」が０個以上１個以下あり, その後に標準関数 isalpha が非零の値を返す文字（「１０進数字」）が１個以上続く」ものとします. また, 文字列中には「空白文字」, 「符号文字」, 「１０進数字」以外の文字は含まれないものとします.

ただし, 「１０進整数をあらわす文字列」からそれがあらわす整数への変換は, 標準関数 atoi を利用してください. また, この問題では「桁あふれ」は考慮しないこととします.

★ exercise-12-9 （**難**）「非負の１０進整数」をあらわす文字列を受け取って, ex-10-10 で定義した 64 ビット整数として格納するプログラムを書きなさい.

逆に, 64 ビット整数を文字列として「１０進整数」に変換するプログラムを書きなさい.

★ exercise-12-10 （**難**）“strstr” のような「文字列検索」を行う際に, 通常（標準関数 strstr で実装されている方法）は以下のような方法である.

【例１】 検索対象文字列を “aabaabaabcabcd”, 検索文字列を “abc” としたとき, 及び, 検索文字列を “aabc” としたとき, 以下のようにマッチングを行う.

	12345678901234 aabaabaabcabcd
1	abc
2	␣abc
3	␣␣abc
4	␣␣␣abc
5	␣␣␣␣abc
6	␣␣␣␣␣abc
7	␣␣␣␣␣␣abc
8	␣␣␣␣␣␣␣abc

	12345678901234 aabaabaabcabcd
1	aabc
2	␣aabc
3	␣␣aabc
4	␣␣␣aabc
5	␣␣␣␣aabc
6	␣␣␣␣␣aabc
7	␣␣␣␣␣␣aabc

この方法は, いわゆる「ムリヤリな」方法であり, 検索対象文字列長を N, 検索文字列長を M としたとき, 最悪 $(N-M-1)M$ 回の文字照合が必要となる. 一般には $M \ll N$ と仮定してよいので, 「ムリヤリな」方法の文字照合回数は $O(NM)$ となる.

☆ exercise-12-10-1 $(N-M-1)M$ 回の文字照合が必要な検索対象文字列と検索文字列の組の例を挙げなさい.

以下では, 文字列検索アルゴリズムの改良を考えよう.

一つの改良として, 検索文字列の「**後ろから**」文字照合を行うと, より文字照合回数の少ない文字列検索アルゴリズムを得ることができる. その方法を**不一致文字法**と呼ぶ.

【例２】 123ababc の中で abc を検索しようとするとき, 以下のような手順をとる.

- 123 と abc のそれぞれの末尾文字を比較する. それらは一致しないばかりか, 3 は検索文字列の中にあらわれないため, この３文字 123 の中で「部分一致」することもあり得ないことがわかる. そのため, 次は検索対象文字列の４文字目から一致を調べればよい.
- aba と abc のそれぞれの末尾文字を比較する. それらは一致しないが, 今度は一致しなかった検索対象文字列の末尾 a は, 検索文字列の先頭にあらわれている. そのため, 次は末尾の a を検索文字列の先頭にあわせて一致を調べればよい.

- abc と abc のそれぞれの末尾文字を比較する．今回はそれらが一致するので, 検索対象文字列, 検索文字列を順に前に戻りながら全体の一致をしらべる．

他の例では, 次のようになる．

照合位置	1234567890123456 123aabaabaabcbcd		
3	abc	abc の中に 3 はあらわれない．	3 文字スキップ
6	␣␣␣abc	abc の中で b は 2 文字目に一致	1 文字スキップ
7	␣␣␣␣abc	abc の中で a は 1 文字目に一致	2 文字スキップ
9	␣␣␣␣␣␣abc	abc の中で b は 2 文字目に一致	1 文字スキップ
10	␣␣␣␣␣␣␣abc	abc の中で a は 1 文字目に一致	2 文字スキップ
12	␣␣␣␣␣␣␣␣␣abc	abc の中で b は 2 文字目に一致	1 文字スキップ
13	␣␣␣␣␣␣␣␣␣␣abc	abc の末尾が一致！	

すなわち, アルゴリズムは以下のようになる．

検索文字列長が M の時, 検索文字列の末尾の文字で照合を行い, その文字が検索文字列の中の k 文字目にあらわれたとき,（検索文字列中にあらわれないときには $k = 0$）検索位置を $M - k$ 文字スキップすることができる．（検索文字列中に同一文字が複数回出てくるときには, 最も右のものを利用する.）

不一致文字法をプログラムするためには, 最初に検索文字列を走査して, 不一致が発生した文字に対して次に何文字スキップができるかを示す skip 表を構成する．すなわち, skip[c] は, 検索文字列の末尾位置が検索対象文字列中の文字 c であるときに, 何文字スキップするかをあらわし, c が検索文字列中の右から j 文字目にあるときには skip[c] = M-j とすればよい．（c が検索文字列中にあらわれないときには skip[c] = M とする.）検索文字列を移動する文字数をあらわす．

☆ exercise-12-10-2 不一致文字法のアルゴリズムを書きなさい．

☆ exercise-12-10-3 不一致文字法を用いて strstr 関数を改良しなさい．

☆ exercise-12-10-4 不一致文字法では文字照合回数は $O(N + M)$ であり, アルファベットの種類が M に比べて十分に多ければ $O(N/M)$ 回程度の照合回数となる．この事実を証明しなさい．

【ヒント】 最悪の状況は, 最後にしか末尾文字の一致が発生せず, 末尾文字の一致が発生しないときには常にスキップ長が 1 になるときである．

文字列検索は, この他にも改良方法が存在する．

例１からわかるように, ステップ 2 で「2 文字」の一致を見たとき, 対象文字列の「3 文字目」b は検索文字列の 1 文字目 a とは一致しないため, ステップ 3 の検索は「ムダ」であることがわかる．検索文字列がこの場合には,「2 文字のみ」の一致を見たとき, 検索文字列の一致判定を行う場所を「2 文字戻す」のではなく,「1 文字のみ戻せばよい」ことがわかる．

	12345678901234 aabaabaabcabcd
1	abc
2	␣abc
4	␣␣␣abc
5	␣␣␣␣abc
7	␣␣␣␣␣␣abc
8	␣␣␣␣␣␣␣abc

	12345678901234 aabaabaabcabcd
1	aabc
3	␣␣aabc
4	␣␣␣aabc
7	␣␣␣␣␣␣aabc

この方法を **Knuth-Morris-Pratt 法** (**KMP 法**) と呼ぶ．KMP 法を実装するには, 次のように行う．

1. 検索文字列を走査して,「前方部分一致した時に何文字戻すか」(別の言い方では,「前方部分一致したときに, 次にどこから一致判定を行うか」）の表を作成する．
   具体的には, 以下のように行う．
   (a) 「検索文字列の前方 j 文字の部分文字列」とそのコピーを作成し, コピーの先頭文字と元の 2 文字目が対応するように重ねる．（検索文字列が abc, aabc, abab の時には次のようになる）

j	abc
1	a ␣a
2	ab ␣ab

j	aabc
1	a ␣a
2	aa ␣aa
3	aab ␣aab

j	abab
1	a ␣a
2	ab ␣ab
3	aba ␣aba

   (b) 文字列の前方部分一致が起るところまでコピーを右にずらしていく．

j	abc
1	a ␣a
2	ab ␣␣ab

j	aabc
1	a ␣a
2	aa ␣aa
3	aab ␣aab

j	abab
1	a ␣a
2	ab ␣␣ab
3	aba ␣␣aba

   (c) ここで得られた元とコピーの文字列が前方一致している長さを表にする．

j	abc
1	0
2	0

j	aabc
1	0
2	1
3	1

j	abab
1	0
2	0
3	1

   ここで得られた表を「検索文字列に対する next 表」と呼ぼう．next 表は, 前方部分一致が発生した際, どれだけ検索開始位置を動かせばよいかをあらわしている．
2. 検索対象文字列と検索文字列の照合を行い, 前方部分一致した際に, 検索対象文字列の検索位置を next 表にしたがって決定する．

☆ exercise-12-10-5 KMP 法のアルゴリズムを書きなさい．

☆ exercise-12-10-6 KMP 法を用いて strstr 関数を改良しなさい．

☆ exercise-12-10-7 KMP 法では文字列照合回数は $2(N + M)$ 回以下となる．この事実を証明しなさい．

【ヒント】 next 表の構成には $2M$ 回の文字照合が必要である．next 表を構成した後の文字照合回数は $2N$ 回である．

ここまでで見つけた strstr の改良は, 次のように組み合わせることが可能である．

例えば, 検索文字列が abc で検索対象文字列の部分列が xyc の時など, 不一致文字法では末尾文字が一致して, 検索文字列と検索対象文字列の部分列が一致しないとき, スキップ長は 1 と設定される．しかし, KMP 法で用いた next 表に対応する「逆向き next 表」があれば, スキップ

長を 1 よりも多くとることが可能になる. 実際, この例（検索文字列が abc, 検索対象の部分列が xyc の時）には, 不一致は後ろから 2 文字目で発生しているため, スキップ長を 2 ととることが可能となる.

このように不一致文字法に逆向きの next 表を組み合わせた検索方法を **Boyer-Moore 法** (**BM 法**) と呼ぶ.

☆ exercise-12-10-8 BM 法のアルゴリズムを書きなさい.

☆ exercise-12-10-9 BM 法を用いて strstr 関数を改良しなさい.

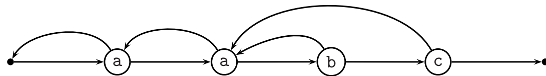
☆ exercise-12-10-10 BM 法では文字列照合回数は $O(N+M)$ 回以下となり, アルファベットの種類が M に比べて十分に多ければ $O(N/M)$ 回程度の照合回数となる. この事実を証明しなさい.

☆ exercise-12-10-11 具体的な文字列検索に対して,「ムリヤリな」方法, 不一致文字法, KMP 法, BM 法のそれぞれの長所・短所を比較しなさい. 具体的には, next 表や skip 表の構成が必要なことや, 検索文字列, 検索対象文字列の長さ, そこに含まれるアルファベットの種類などに目をつけて比較しなさい.

KMP 法の next 表は次のような意味を持つ. 例えば検索文字列が aabc の時の next 表に next[0] = -1 を追加して表を作る.

j	next
0	-1
1	0
2	1
3	1

この表にしたがって矢印を書けば, 次のような図ができる.



この図は, 検索文字列と対象文字列の照合の開始時には左のノードに位置し, 1 文字目の a との照合を行うときに 2 番目の a というノードに移動, そこで, 一致していれば右に進み, 次のノードに移動, 一致していなければ, 左の矢印に沿って次のノードに移動, 最も右のノードに達したとき, 検索文字列と対象文字列が一致したことをあらわす. このような図を「**状態遷移図**」と呼ぶ.

この図を使うと次のようなアルゴリズムを得ることができる.

- 検索対象文字列を s, 検索文字列を p とおき, s の中から p に最初に一致する位置を求める. i は s の中の文字照合のインデックスをあらわし, state は状態遷移図の状態位置をあらわす.
  1. state を -1, i を 0 とする.
  2. i が s の長さに到達するまで以下を繰り返す.
     (a) s の i 番目の文字が state の文字と一致したとき, 右向き矢印に沿って state を移動し, i を次に移動する.
     一致しない場合, state を不一致の場合の state に移動する. （左向き矢印に沿って移動する）
     (b) state が開始状態 -1 の場合, 右向き矢印に沿って state を移動し, i を次に移動する.
     (c) state が終了状態の時, 一致位置を i から p の長さ分だけ戻した位置としてアルゴリズムを終了する.
  3. s の中に p と一致する場所は無いとしてアルゴリズムを終了する.

☆ exercise-12-10-12 検索文字列を固定して, KMP 法の状態遷移図を利用してプログラムを書きなさい.

☆ exercise-12-10-13 検索文字列 10110101 に対して, KMP 法の状態遷移図を書きなさい.

【その他】 電子メールで「今日の講義の感想や意見」を送ってください.

★ ex12-1.c の内容

```
/* $Id: ex12-1.c,v 1.3 2004-06-30 08:01:11+09 naito Exp $ */
#include <stdio.h>

int main(int argc, char **argv)
{
  int   n = 1, m = 3 ;
  int   *pn, *pm ;

  pn = &n ;
/*    printf("*pn = %d, *pm = %d, n = %d, m = %d\n", *pn, *pm, n, m) ; */
  printf("*pn = %d, n = %d, m = %d\n", *pn, n, m) ;
  *pn = 2 ;  /* ここで n の値はどうなっているか？    */
  printf("*pn = %d, n = %d, m = %d\n", *pn, n, m) ;
/*    printf("*pn = %d, *pm = %d, n = %d, m = %d\n", *pn, *pm, n, m) ; */
  pm = pn ;  /* この時点で pm の指し示すものは何か？ */
  printf("*pn = %d, *pm = %d, n = %d, m = %d\n", *pn, *pm, n, m) ;
  pn = &m ;  /* この時点で pn, pm の指し示すものは何か？ */
  printf("*pn = %d, *pm = %d, n = %d, m = %d\n", *pn, *pm, n, m) ;
  *pm = 5 ;  /* これは n, m のいずれの（それとも両方の？）値を変更したことになるか？ */
  printf("*pn = %d, *pm = %d, n = %d, m = %d\n", *pn, *pm, n, m) ;


  printf("pn           = %p\n", pn) ;
  printf("pm           = %p\n", pm) ;
  printf("address of n = %p\n", &n) ;
  printf("address of m = %p\n", &m) ;
  return 0 ;
}
```

★ ex12-2.c の内容

```
/* $Id: ex12-2.c,v 1.4 2004-06-25 11:29:32+09 naito Exp $ */
/* 配列とポインタ */
#include <stdio.h>
int main(int argc, char **argv)
{
  int   a[10] = {0,1,2,3,4,5,6,7,8,9} ;
  char  s[11] = "ABCDEFGHIJ" ;
  int   *pa ;
  char  *ps ;
  int   i ;
  ps = s ;
  printf("%c %c %p %p\n", *ps, s[0], ps, &s[0]) ;
  printf("%c %c %p %p\n", *(ps+1), s[1], ps+1, &s[1]) ;
  ps = s+1 ;
  printf("%c %c %p %p\n", *ps, s[1], ps, &s[1]) ;
  printf("%c %c %p %p\n", *(ps+1), s[2], ps+1, &s[2]) ;
  ps = &s[2] ;
  printf("%c %c %p %p\n", *ps, s[2], ps, &s[2]) ;
  printf("%c %c %p %p\n", *(ps+1), s[3], ps+1, &s[3]) ;
  printf("%c %c %p %p\n", s[0], *s, &s[0], s) ;
  printf("%c %c %p %p\n", s[2], *(s+2), &s[2], s+2) ;
  ps = s+1 ;
  printf("%s\n", ps) ;
  ps = s ;
  for(i=0;i<10;i++)
    printf("%d %c %c %c %p %p %p\n", i, s[i], *(ps+i), ps[i], &s[i], s+i, ps+i) ;
  pa = a ;
  printf("%d %d %p %p\n", *pa, a[0], pa, &a[0]) ;
  printf("%d %d %p %p\n", *(pa+1), a[1], pa+1, &a[1]) ;
  pa = a+1 ;
  printf("%d %d %p %p\n", *pa, a[1], pa, &a[1]) ;
  printf("%d %d %p %p\n", *(pa+1), a[2], pa+1, &a[2]) ;
  pa = &a[2] ;
  printf("%d %d %p %p\n", *pa, a[2], pa, &a[2]) ;
  printf("%d %d %p %p\n", *(pa+1), a[3], pa+1, &a[3]) ;
  printf("%d %d %p %p\n", a[0], *a, &a[0], a) ;
  printf("%d %d %p %p\n", a[2], *(a+2), &a[2], a+2) ;
  pa = a ;
  for(i=0;i<10;i++)
    printf("%d %d %d %d %p %p %p\n", i, a[i], *(pa+i), pa[i], &a[i], a+i, pa+i) ;
  return 0 ;
}
```


★ ex12-3.c の内容

```
/* $Id: ex12-3.c,v 1.2 2004-06-25 10:17:10+09 naito Exp $ */
/* ポインタを引数にする関数 */
#include <stdio.h>

int foo_0(int) ;
int foo_1(int *) ;

int     a=1 ;

int main(int argc, char **argv)
{
  int   b=2 ;

  printf("(main)\taddress of a = %p\n", &a) ;
  foo_0(a) ; printf("%d\n", a) ;
  foo_1(&a) ; printf("%d\n", a) ;

  printf("(main)\taddress of b = %p\n", &b) ;
  foo_0(b) ; printf("%d\n", b) ;
  foo_1(&b) ; printf("%d\n", b) ;

  return 0 ;
}

int foo_0(int a)
{
  printf("(foo_0)\taddress of a = %p\n", &a) ;
  a += 1 ;
  return 0 ;
}

int foo_1(int *a)
{
  printf("(foo_1)\taddress of a = %p\n", a) ;
  *a += 1 ;
  return 0 ;
}
```

★ ex12-4.c の内容

```
/* $Id: ex12-4.c,v 1.6 2004-06-30 08:00:35+09 naito Exp $ */
/* 配列要素のコピー */
#include <stdio.h>
#include <strings.h>
#define N       11
char *_strcpy(char *, char *) ;
char *strcpy(char *, const char *) ;

int main(int argc, char **argv)
{
  char  a[N] = "0123456789" ;
  char  b[N], c[N], d[N], e[N] ;
  int   i, len ;

  len = strlen(a) ;
  for(i=0;i<=len;i++) *(b+i) = *(a+i) ;
  i = 0 ;
  while((*(c+i) = *(a+i))) i += 1 ;
  _strcpy(d,a) ;
  strcpy(e,a) ;
  printf("a = %s\n", a) ;
  printf("b = %s\n", b) ;
  printf("c = %s\n", c) ;
  printf("d = %s\n", d) ;
  printf("e = %s\n", e) ;
  /* ところが，以下はエラーになる */
  /* while((*e++ = *a++)) ; */
  return 0 ;
}

char *_strcpy(char *t, char *s)
{
  int   i = 0 ;
  while((*(t+i) = *(s+i))) i += 1 ;
  return t ;
}

char *strcpy(char *t, const char *s)
{
  char *save=t ;
  while((*t++ = *s++)) ;
  return save ;
}
```


★ ex12-5.c の内容

```
/* $Id: ex12-5.c,v 1.1 2004-06-24 17:44:15+09 naito Exp naito $ */
/* ポインタと配列の違い */
#include <stdio.h>

int main(int argc, char **argv)
{
  char  a[] = "message" ;
  char  *p  = "message" ;
  int   i ;

  printf("p = %s\n", p) ;
  for(i=0;*(p+i);i++) printf("%s\n", p+i) ;
  printf("p = %s\n", p) ;
  for(;*p;p++) printf("%s\n", p) ;

  printf("a = %s\n", a) ;
  for(i=0;*(a+i);i++) printf("%s\n", a+i) ;
  /* ところが，次はエラーになる */
  /* for(;*a;a++) printf("%s\n", a) ; */
  return 0 ;
}
```

★ ex12-6.c の内容

```
/* $Id: ex12-6.c,v 1.2 2004-06-25 11:28:34+09 naito Exp $ */
/* ポインタを引数にする関数 */
#include <stdio.h>
void swap_char(char *, char *) ;
void not_swap_char(char, char) ;
void swap_int(int *, int *) ;
int main(int argc, char **argv)
{
  char  a = 'a', b = 'b' ;
  int   n = 0x01020304, m = 0x05060708 ;
  not_swap_char(a,b) ;
  printf("a = %c, b = %c\n", a, b) ;
  swap_char(&a,&b) ;
  printf("a = %c, b = %c\n", a, b) ;
  swap_int(&n, &m) ;
  printf("n = %08x, m = %08x\n", n, m) ;
  /* 次は BUG が生じる */
  swap_char((char *)&n, (char *)&m) ;
  printf("n = %08x, m = %08x\n", n, m) ;
  /* 次は実行時エラーが生じる可能性がある */
  /* swap_int((int *)&a, (int *)&b) ; */
  return 0 ;
}
void swap_char(char *a, char *b)
{
  char  c ;
  c = *a ; *a = *b ; *b = c ;
  return ;
}
void not_swap_char(char a, char b)
{
  char  c ;
  c = a ; a = b ; b = c ;
  return ;
}
void swap_int(int *a, int *b)
{
  int   c ;
  c = *a ; *a = *b ; *b = c ;
  return ;
}
```


★ ex12-7.c の内容

```
/* $Id: ex12-7.c,v 1.2 2004-06-25 13:39:57+09 naito Exp $ */
/* 一般の swap 関数（汎用データポインタ） */
#include <stdio.h>

void swap(void *, void *, size_t size) ;

int main(int argc, char **argv)
{
  char  a = 'a', b = 'b' ;
  int   n = 0x01020304, m = 0x05060708 ;

  swap(&a,&b,sizeof(char)) ;
  printf("a = %c, b = %c\n", a, b) ;
  swap(&n,&m,sizeof(int)) ;
  printf("n = %08x, m = %08x\n", n, m) ;
  return 0 ;
}

void swap(void *a, void *b, size_t size)
{
  char  c ;
  int   i=0 ;

  while(i<size) {
    c = *(char *)a ;
    *(char *)a++ = *(char *)b ;
    *(char *)b++ = c ;
    i += 1 ;
  }
  return ;
}
```

★ ex12-8.c の内容

```
/* $Id: ex12-8.c,v 1.2 2004-06-25 20:15:21+09 naito Exp $ */
/* sizeof の効果 */
#include <stdio.h>

void foo_0(int *) ;
void foo_1(int []) ;
void foo_2(int [20]) ;
void foo_3(int [10]) ;

int main(int argc, char **argv)
{
  char  a[10] ;
  int   b[20] ;

  printf("%lu\n", sizeof(a)/sizeof(a[0])) ;
  printf("%lu\n", sizeof(b)/sizeof(b[0])) ;
  foo_0(b) ; foo_1(b) ; foo_2(b) ; foo_3(b) ;
  return 0 ;
}

void foo_0(int *a)
{
  printf("%lu\n", sizeof(a)/sizeof(a[0])) ;
  return ;
}

void foo_1(int a[])
{
  printf("%lu\n", sizeof(a)/sizeof(a[0])) ;
  return ;
}

void foo_2(int a[20])
{
  printf("%lu\n", sizeof(a)/sizeof(a[0])) ;
  return ;
}

void foo_3(int a[10])
{
  printf("%lu\n", sizeof(a)/sizeof(a[0])) ;
  return ;
}
```


★ ex12-9.c の内容

```
/* $Id: ex12-9.c,v 1.3 2004-06-26 17:51:03+09 naito Exp $ */
/* ポインタの配列とポインタへのポインタ */
#include <stdio.h>

int main(int argc, char **argv)
{
  char  **pstr ;
  char  *str[3] ;
  char  s0[]="string 0", s1[]="string 1", s2[]="string 2" ;
  char  *str_array[] = {"abc", "defg", "hijkl"} ;
  int   i ;

  str[0] = s0 ; str[1] = s1 ; str[2] = s2 ;
  pstr = str ;
  printf("%s\n", str[0]) ;
  printf("%s\n", str[1]) ;
  printf("%s\n", str[2]) ;
  for(i=0;i<3;i++) printf("%s\n", *(pstr+i)) ;

  for(i=0;i<3;i++) printf("%s\n", str_array[i]) ;

  for(i=0;i<argc;i++) printf("%s\n", argv[i]) ;
  return 0 ;
}
```

★ ex12-10.c の内容

```
/* $Id: ex12-10.c,v 1.6 2004-06-25 20:14:31+09 naito Exp $ */
/* 文字列に含まれる文字の最初の位置を見つける */
#include <stdio.h>

char *strchr(const char *, int) ;

int main(int argc, char **argv)
{
  char  s[] = "This is a test." ;
  char  *p ;

  if ((p = strchr(s,'i')) != NULL) {
    printf("%s\n", p) ;
    printf("%lu\n", p-s) ;
  }
  else
    printf("Not Found\n") ;
  if ((p = strchr(s,'x')) != NULL) {
    printf("%s\n", p) ;
    printf("%lu\n", p-s) ;
  }
  else
    printf("Not Found\n") ;
  return 0 ;
}

char *strchr(const char *s, int c)
{
  while((*s)&&(*s++ != c)) ;
  if (!*s) return NULL ;
  return (char *)(s-1) ;
}
```

● 前回・前々回の課題の解説

★ exercise-10-1 ex10-1.c を閏年の場合にも正しい値を返し, 存在しない日付の場合には -1 を返すように改良しなさい. なお year は１９７０年以降であるとします. すなわち, year として 1969 以下の値が与えられたときは「存在しない日付」と考えます. （これは, 以下同じ主旨の問題に対して同様とします.）


```
/* $Id: exercise10-1-2.c,v 1.7 2004-06-29 09:43:58+09 naito Exp $ */
/* ex10-1.c を閏年の場合にも正しい値を返し,
 * 存在しない日付の場合には -1 を返すように改良しなさい. */
#include <stdio.h>
int     days_of_year(int, int, int) ;
int     is_leapyear(int year) ;
int main(int argc, char **argv)
{
  int   i ;
  for(i=0;i<12;i++)
    printf("%d\n", days_of_year(2000,i+1, 1)) ;
  printf("%d\n", days_of_year(2000,12,31)) ;
  return 0 ;
}
/* year 年 month 月 day 日が何日目かを返す. */
int days_of_year(int year, int month, int day)
{
  static int    n_days_of_month[]={31,28,31,30,31,30,31,31,30,31,30,31} ;
  static int    l_days_of_month[]={31,29,31,30,31,30,31,31,30,31,30,31} ;
  int           *days_of_month ;
  int           days = 0 ;
  int           i ;
  if (year < 1970) return -1 ;
  if ((month < 1)||(month > 12)) return -1 ;
  if (is_leapyear(year)) days_of_month = l_days_of_month ;
  else                   days_of_month = n_days_of_month ;
  if (day > days_of_month[month-1]) return -1 ;
  for(i=0;i<month-1;i++) days += days_of_month[i] ;
  days += day ;
  return days ;
}
int is_leapyear(int year)
{
  return ((year%4 == 0)&&(year%100 != 0))||(year%400 == 0) ;
}
```

- 今回のプログラムの改良では, ポインタを用いて, 閏年と平年の場合によって利用する配列を切り替えている.

前回の資料のこのプログラム exercise10-1-1.c に BUG がありました. 以下のように修正します. （１９６９年以前を「存在しない日付」として扱っていませんでした.）

```
/* これより上，省略 */
int days_of_year(int year, int month, int day)
{
  /* 3行略 */
  int          i ;
  if (year < 1970) return -1 ; /* この行を挿入 */
  if ((month < 1)||(month > 12)) return -1 ;
/* これより下，省略 */
```

★ exercise-11-2 b を 2 以上 36 以下の整数としたとき, unsigned int 型の値を b 進表示した結果を表示するプログラムを以下のような関数を構成して書きなさい.

【形式】

void radix(char s[], unsigned int a, unsigned int b)

【機能説明】

unsigned int 型の値 a の b 進表示した文字列を s に返す.


```
/* unsigned int の b 進表示 */
#include <stdio.h>
#include <strings.h>
#define MAX_DIGIT      (36)
#define MAX_CHAR       (32+1)
char    digit[] = "0123456789abcdefghijklmnopqrstuvwxyz" ;
void    radix(char *, unsigned int, unsigned int) ;
void    _strrev(char *) ;

int main(int argc, char **argv)
{
  char  s[MAX_CHAR] ;
  int   i ;

  for(i=2;i<=MAX_DIGIT;i++) {
    radix(s, 60000, i) ;
    printf("%2d: %s\n", i, s) ;
  }
  return 0 ;
}

void radix(char *s, unsigned int a, unsigned int b)
{
  int   i = 0 ;
  unsigned int  r ;

  while(a) {
    r = a%b ;
    s[i++] = digit[r] ;
    a /= b ;
  }
  s[i] = '\000' ;
  _strrev(s) ;
  return ;
}
```

ただし _strrev は前のものをそのまま流用する. また, 関数の仕様中で char s[] を char *s に書き換えている.