18　補足事項 〜これまでに説明できなかったこと〜

18–1　制御構造 〜繰返しからの脱出〜

while 文・do〜while 文・for 文の繰返しのブロックの中から、終了条件に達していない段階で例外的に抜け出したい、というような場合には、break 文や continue 文を用いる。

break 文・continue 文は、if 文のブロックに対しては無効。

(1) break 文: 実行されると直ちに、最も内側のループから抜け出し、その直後に飛ぶ。

```
...
{   /* ループここから */
    ...
    break;
    ...
}   /* ループここまで */
...         /*** break でここ（ループの直後）に飛ぶ ***/
```

break 文は、多岐条件分岐の switch 文でも用いられる。詳細後述。

(2) continue 文: 実行されると直ちに、最も内側のループの先頭へ戻る。

```
...         /*** continue でここ（ループの先頭）に戻る ***/
{   /* ループここから */
    ...
    continue;
    ...
}   /* ループここまで */
...
```

continue 文の戻り先:

- while 文では、条件式の評価へ
- do〜while 文では、ループの最初の実行式へ
- for 文では、増分式の評価へ

例 . break 文・continue 文による例外処理。

```
/*   escape.c   2010-07-19   */
#include <stdio.h>

int main( int argc, char **argv )
{
    int i;

    for( i=0; i<10; i++ )
    {
        if( i == 5 )
        {
            break;
        }
        printf("%2d", i);
    }
    printf(" <--- break\n");

    for( i=0; i<10; i++ )
    {
        if( i == 5 )
        {
            continue;
        }
        printf("%2d", i);
    }
    printf(" <--- continue\n");

    return 0;
}
```

実行例 . break 文・continue 文の効果の違いに注目。

```
 0 1 2 3 4 <--- break
 0 1 2 3 4 6 7 8 9 <--- continue
```

使用例としては、例えば、配列の中で条件を満たす要素を端から探していって、見付からなければ最後の要素まで見てやめるが、途中で見付かったら直ちに抜け出したい、というように、終了条件が複数ある時に有効。

18–2　制御構造 〜多岐条件分岐〜

例えば、幾つか (3 つ以上) の選択肢の中から一つ選んで入力を促し、その入力値によって場合分けをするような場合、勿論 `if 〜 else` の繰返しで書いても良いのだが、構造を正確に反映している気がしない。このように或る式の値によって多岐に場合分けを行なう場合には、`switch` 文を用いて、より自然に記述することが出来る。例で示そう。

例．`n` は整数型とする。

```
switch ( n ) {
  case 1:
       文 A;
  case 2:
       文 B;
       文 C;
       break;
  case 3:
  case 4:
       文 D;
       break;
  case 5:
       文 E;
  default:
       文 F;
}
```

この例では、`n` の値に従って、
1: A,B,C.
2: B,C.
3: D.
4: D.
5: E,F.
その他: F.
を実行して、次へ進む。

`switch` 文は式 (例では `n`) の値を評価し、その値に等しい値のラベル (なければ `default`) に飛ぶ。式は整数型に限るが、`int` 型でなくても良い。文字型 `char` も整数型である。そこから `break` 文に出会うか又は最後の `}` まで実行して、`}` の後に抜ける。基本的にはラベルは飛ぶ目印であるだけなので、`break` 文がなければ、そのまま次のラベルの後の文も実行してしまう。逆にこれを利用して、`case` を連続して書くことにより、複数の値に対して、実質的に同じ場所に飛ばすことが出来る。

例 . switch 文による多岐条件分岐。

```
/*   arith3.c   2010-07-19   */

#include <stdio.h>

int main( int argc, char **argv )
{
    int x, y;
    char c;

    printf("Input two integers combined by an arithmetic operator: ");
    scanf("%d%c%d", &x, &c, &y);

    switch( c ) {
      case '+':
          printf("%d + %d = %d\n", x, y, x+y);
          break;
      case '-':
          printf("%d - %d = %d\n", x, y, x-y);
          break;
      case '*':
          printf("%d * %d = %d\n", x, y, x*y);
          break;
      case '/':
          printf("%d / %d = %d\n", x, y, x/y);
          break;
      default:
          printf("'%c' is not valid.\n", c);
    }
}
```

このように並列に多岐分岐する場合に適切。

break 文を忘れると、そのまま次の printf() に突っ込んでしまうので注意。

実行例 . break 文と default ラベルの働きを確認のこと。

```
=> ./arith3
Input two integers combined by an arithmetic operator: 7*11
7 * 11 = 77
=> ./arith3
Input two integers combined by an arithmetic operator: 7@11
'@' is not valid.
=>
```

18–3 ファイル入出力

これまでのプログラムでの入出力は全て、標準入出力を介して行なっていた。その既定値は、標準入力の場合は「キーボードからの入力」であり、標準出力の場合は「ディスプレイへの出力」である。しかし、大量のデータを処理したい、データを再利用したい、などといった場合には、「ファイルからの入力」「ファイルへの出力」を行なうことが望まれる。それには、シェルの入出力リダイレクション (<, >) を利用して、標準入出力をファイルに変更するのがてっとり早く、標準入出力を介して入出力を行なう既存のプログラムをそのまま利用できる利点もあるが、限界もある。

例えば、複数のファイルから交互にデータを読みたい場合などは、入力リダイレクションでは無理。

C 言語にもファイル入出力の為の機構が標準で用意されている。その大まかな流れは次の通り。

(1) FILE へのポインタ変数を宣言する。

同時に利用するファイルの数だけ FILE へのポインタ変数を宣言して用いる。

```
FILE *fp;        /* ファイルポインタの宣言 */
```

ファイルへのアクセスは、全てこのポインタを介して行なう。

FILE 型は、ファイル入出力に必要な一連のデータから成る構造体型で、stdio.h 内で定義されている。その構造体の中身はシステムによって異なり得るが、とにかくこの型を使えばファイル入出力が出来るように実装すべく定められている。

(2) ファイルを開く (関数 fopen() を用いる)。

関数 fopen() は、ファイル名と形式を引数に取り、その指定に応じてファイルを開き、そこへのポインタを返す。ファイル名は、実行形式のあるディレクトリからの相対パスや絶対パスによって文字列で与える。形式には、読取専用なら r を、書込専用なら w を指定する。

```
fp = fopen("data.txt", "r");   /* data.txt を読み込み専用で開く */
```

関数 fopen() は、指定されたファイルのオープンに失敗した場合には、その値として NULL を返す。そのチェックを含めた以下のようなスタイルが一種の「決まり文句」である。

「ファイルを開く」とは実際には、FILE 型変数領域を確保し、読み書きに必要なデータを設定した上で、ポインタを介してそれにアクセス出来るようにすることだが、ユーザは余り意識しなくても良い。

代入式の値を利用する例。乱発すると判り難いソースになるが、このような決まり文句で使うと有効。

```
if( (fp = fopen("data.txt", "r")) == NULL )
{
    ファイルのオープンに失敗した場合の処理 ;
}
```

(3) ファイルにアクセス (読み書き) する。

ファイルからの一行入力には関数 fgets() を用いるのが良い。

```
fgets(buf, 100, fp);  /* 最大 100 文字まで fp から buf に読み込む */
```

ファイルへの一行出力には関数 fprintf() や fputs() を用いるのが良い。

```
fprintf(fp, "i=%d\n", i);  /* fp に指定の書式で書き込む */
fputs(str, fp);            /* fp に文字列 str を書き込む */
```

gets() は危険なので使わないこと！

(4) ファイルを閉じる (関数 fclose() を用いる)。

関数 fclose() は、FILE へのポインタを引数に取り、そのファイルを閉じる。

```
fclose(fp);    /* ファイルポインタ fp が指すファイルを閉じる */
```

次の 3 つのファイルポインタが標準で用意されている。

- stdin：標準入力 (読込専用)
- stdout：標準出力 (書込専用)
- stderr：標準エラー出力 (書込専用)

それぞれキーボード入力・ディスプレイ出力 (リダイレクションなどで変更された時は指定されたファイルなど) が指定されている。これらを fgets(), fprintf() などのファイルポインタ指定に用いることも出来るし、FILE * 型変数に代入することも出来るので、条件によってファイル出力と画面出力とを切替えることも容易に可能。

エラーメッセージなど正常な出力データに混ぜたくないものは、標準エラー出力 stderr に出力するのが良い。出力リダイレクション > やパイプライン | では、標準出力 stdout への出力のみが指定されたファイルや後に続くプログラムに向かい、標準エラー出力 stderr への出力はディスプレイに表示される。

前々回に現れた

```
fgets(str, MAXLEN, stdin);
```

は、これを使ったもの。

C シェルでは
>&: 標準エラー出力込みの出力リダイレクション
|&: 標準エラー出力込みのパイプラインが利用できる。

例．指定したファイルの内容を表示するプログラム (つまり自家製 cat)。

```
/*   mycat.c   2010-07-19   */
#include <stdio.h>
#include <stdlib.h>

#define MAXBUF 255

int main( int argc, char **argv )
{
    char buf[MAXBUF+1];
    int i;
    FILE *fp;

    if( argc == 1 )
    {
        printf("Usage: %s filename(s)\n", argv[0]);
        exit(1);
    }

    for ( i=1; i<argc; i++)
    {
        if( ( fp = fopen(argv[i], "r") ) == NULL )
        {
            printf("cannot open file: %s\n", argv[i]);
            exit(2);
        }

        while( fgets(buf, MAXBUF, fp) != NULL )
        {
            printf("%s", buf);
        }

        fclose(fp);
    }

    return 0;
}
```

`argc == 1`
とは、コマンドそのもののみ、即ち引数を取っていないこと。それ以外は
`argc > 1`
となる。

`argv[1]` 以降が (コマンド名の次からの) 引数。

関数 `fgets()` は、ファイルの終了を表す文字を読み込むと `NULL` を返すので、これをファイル終了の判定に用いている。

変数 `i` を使わずに、`argc` 自身をカウンタに流用する手や、`argv` 自身をインクリメントする手もあるが、まぁこれで良かろう。

課題の最終締切は
8月6日 (金)
とする。

引き続き秋学期の「情報処理 IV」も是非受講を。