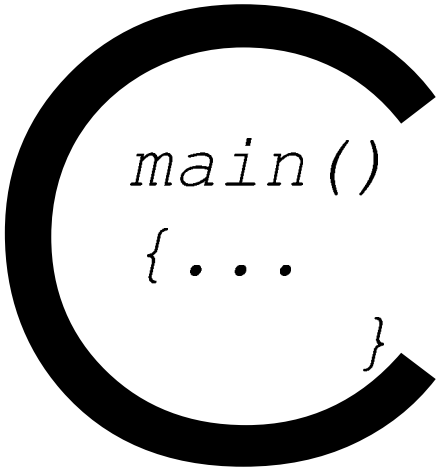
C Programming Practice (32H)

Yi Chen
leo.chen.yi@live.co.uk

- **No1** - Introduction (2H)
- **No2** - Types, Operators And Expressions (2H)
- **No3** - Control Flow (6H)
  - Statements and Blocks
  - Conditional Statements
    - If-Else
    - Switch
  - Loop
    - While
    - For
- **No4** - Functions And Program Structure (6H)
- **No5** - Pointers And Arrays (4H)
- **No6** - Case Studies And Practices (12H)

C Programming Practice
No[4-2]

Chen , Yi
leo.chen.yi@live.co.uk
30-Jul-2010



Input and Output

BETA 1.0.0.1

Contents

Contents

Standard Input and Output

```
#including <stdio.h>
#including <string.h>
#including <ctype.h>
```

The **ANSI** standard defines these library functions precisely, so that they can **exist in compatible** form on any system where C exists. Programs that confine their system interactions to facilities provided by the **standard library** can be moved from one system to another **without change**.

Each source file that refers to an input/output library function must contain the line *#include <stdio.h>*

Standard Input and Output

Two basic data file types:

Text: character-based; stores characters using ASCII or UNICODE character codes

Binary: stores characters in binary form; numbers are in binary, strings are in ASCII or UNICODE form; more compact storage

Standard Input and Output

consider the program converts its input to lower case:

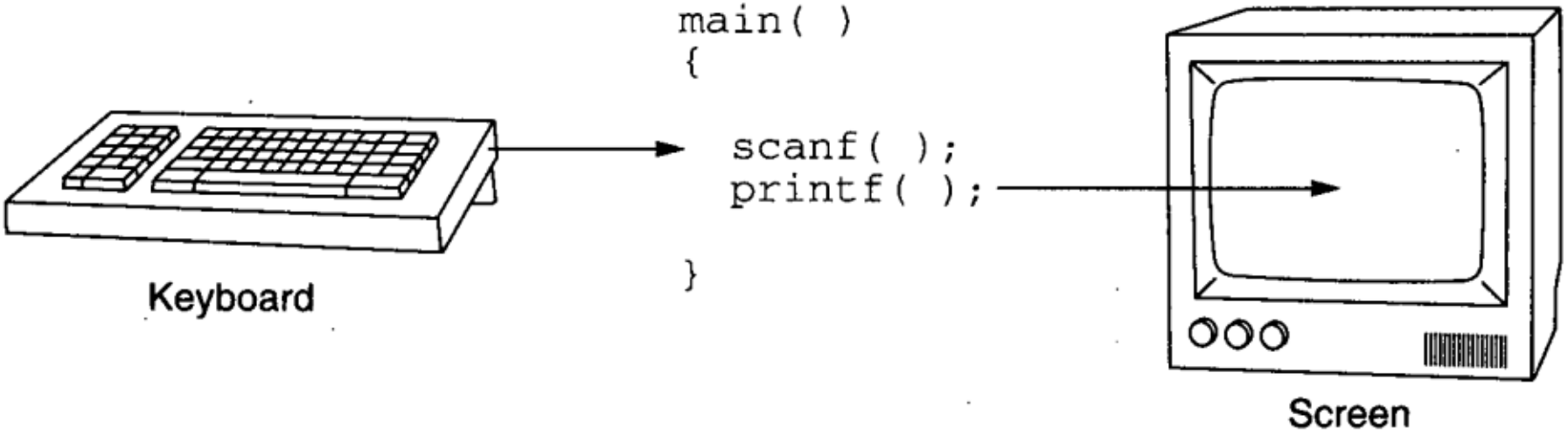
```
#include <stdio.h>
#include <ctype.h>

int main(void)
{ /* lower: convert input to lower case*/

    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));

    return 0;
}
```

Standard Input and Output

The function **tolower** is defined in **<ctype.h>;** it converts an upper case letter to lower case, and returns other characters untouched.

As we mentioned earlier, ``functions'' like **getchar** and **putchar** in **<stdio.h>** and **tolower** in **<ctype.h>** are often macros, thus avoiding the overhead of a function call per character

Standard Input and Output



Contents

Formatted Output

[1] `int printf(const char *format, ...)`

printf(...) *is equivalent to* ***fprintf***(*stdout, ...*).

[2] `int sprintf(char *s, const char *format, ...)`

sprintf *is the same as* ***printf*** *except that the output is written into the* ***string s****, terminated with* ***'\0'****.* ***s*** *must be* ***big enough*** *to hold the result. The return count does* ***not*** *include the '\0'.*

Formatted Output

[3]

```
int vprintf(const char *format, va_list arg)
int vfprintf(FILE *stream, const char *format, va_list arg)
int vsprintf(char *s, const char *format, va_list arg)
```

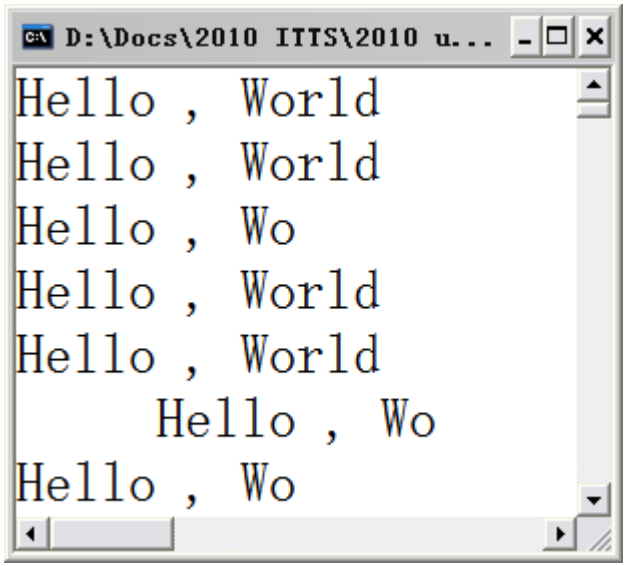
The functions *vprintf*, *vfprintf*, and *vsprintf* are equivalent to the corresponding **printf** functions, except that the variable **argument list** is replaced by **arg**, which has been initialised by the **va_start macro** and perhaps **va_arg** calls. See the discussion of <**stdarg.h**>

Character	fprintf() Argument type; Printed As
d,i	int; **decimal** number
o	int; unsigned **octal** number (without a leading zero)
x,X	int; unsigned **hexadecimal** number (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ...,15.
u	int; **unsigned decimal** number
c	int; **single character**
s	char *; print characters from the **string** until a **'\0'** or the number of characters given by the precision.
f	**double**; [-]***m.dddddd***, where the number of *d*'s is given by the precision (default 6).
e,E	**double**; [-]***m.dddddde+/-xx*** or [-]*m.dddddd*E+/-*xx*, where the number of *d*'s is given by the precision (default 6).
g,G	**double**; use **%e or %E** if the exponent is **less than -4** or greater than or equal to the precision; otherwise use **%f**. Trailing zeros and a trailing decimal point are not printed.
p	void *; **pointer** (implementation-dependent representation).
%	no argument is converted; **print a %**

Formatted Output

```
…

 int number = 10;

   printf("the factorial of %d is %d \n", number,
factorial(number) );

   getch();

   return 0;
…
```

Formatted Output




```
int printf_hello_world(void)
{
char str[] = "Hello , World";

printf("%s\n",str);
printf("%10s\n",str);
printf("%.10s\n",str);
printf("%.15s\n",str);
printf("%-15s\n",str);
printf("%15.10s\n",str);
printf("%-15.10s\n",str);

getch();

return 0;
}
```

Contents

Variable-length Argument Lists

The proper declaration for **printf** is

int printf(char *fmt, ...)

where the declaration **...** means that the number and types of these arguments may **vary**.

Contents

Formatted Input

The function **scanf** is the input analog of **printf**, providing many of the same conversion facilities in the opposite direction.

```
int scanf(char *format, ...)
int sscanf( const char *, const char *, ...)
```

scanf reads characters from the **standard input (keyboard)**, interprets them according to the specification in format, and stores the results through the remaining arguments.

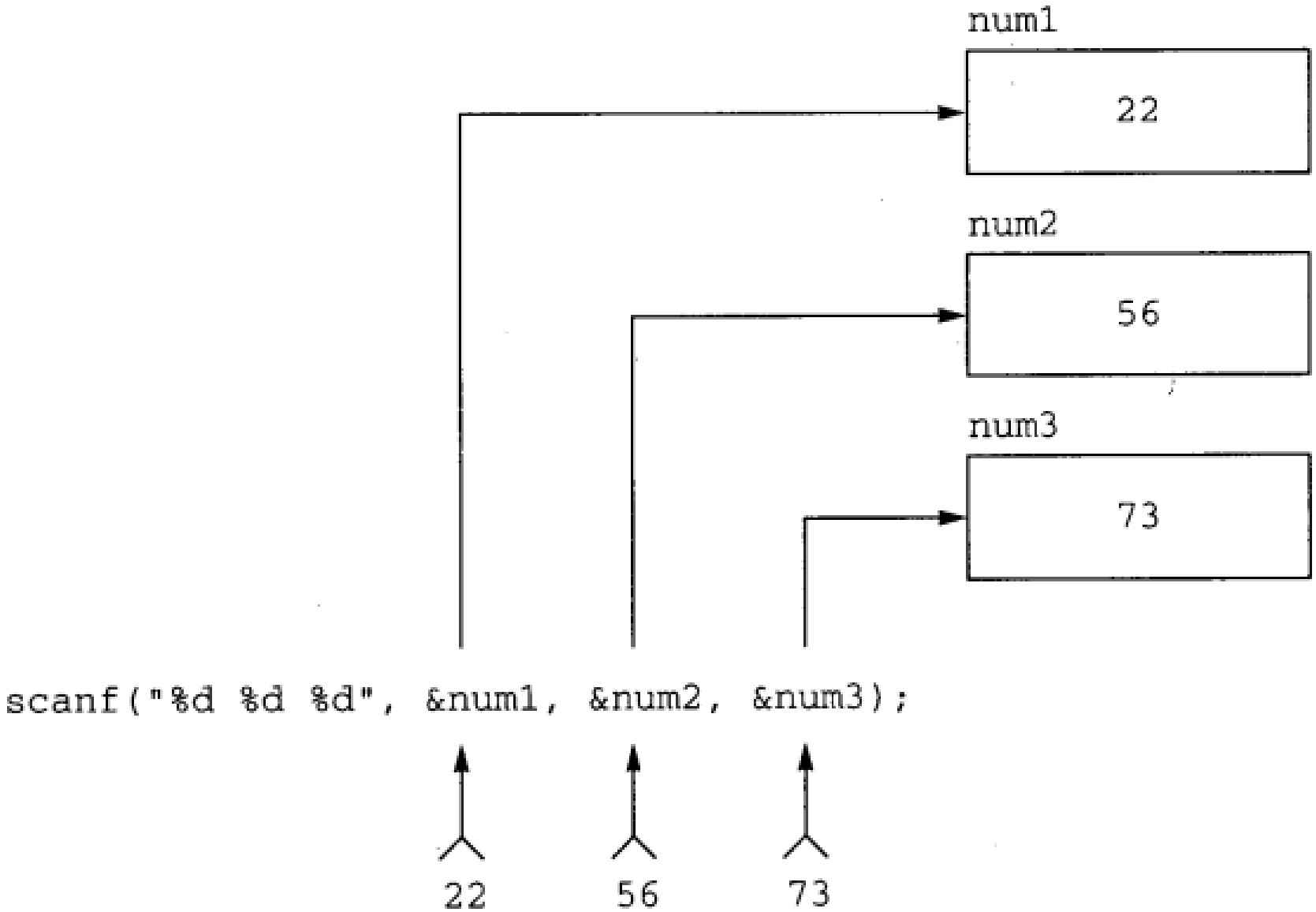
sscanf *reads characters from* ***const char ****

Formatted Input

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

scanf ignores blanks and **tabs** in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values.

Formatted Input



Character	scanf() Input Data; Argument type
d	**decimal integer**; int *
i	**integer**; int *. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	**octal** integer (with or without leading zero); int *
u	**unsigned decimal** integer; unsigned int *
x	**hexadecimal integer** (with or without leading 0x or 0X); int *
c	**characters**; char *. The next input characters (default 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to read the next non-white space character, use %1s
s	character **string** (not quoted); char *, pointing to an array of characters long enough for the string and a terminating '\0' that will be added.
e,f,g	**floating-point** number with optional sign, optional decimal point and optional exponent; float *
%	literal %; no assignment is made.

Formatted Input

A final warning: the **arguments** to scanf and sscanf **must be pointers**. By far the most common error is writing

```
scanf("%d", n);  /* wrong*/
```

instead of

```
scanf("%d", &n); /* ok */
```

This error is not generally detected at compile time.

Formatted Input

```
char buf[512] = {};
sscanf(“123456 ”, “%s”, buf);
printf(“%s\n”, buf);

output: 123456
```

```
"%[1-9a-z]",
..
output: 123456abcdedf
```

```
sscanf("123456 ", "%4s", buf);
printf("%s\n", buf);

output : 1234
```

```
sscanf("123456_abcdedf", "%[^_]",
buf);

printf("%s\n", buf);

output : 123456
```

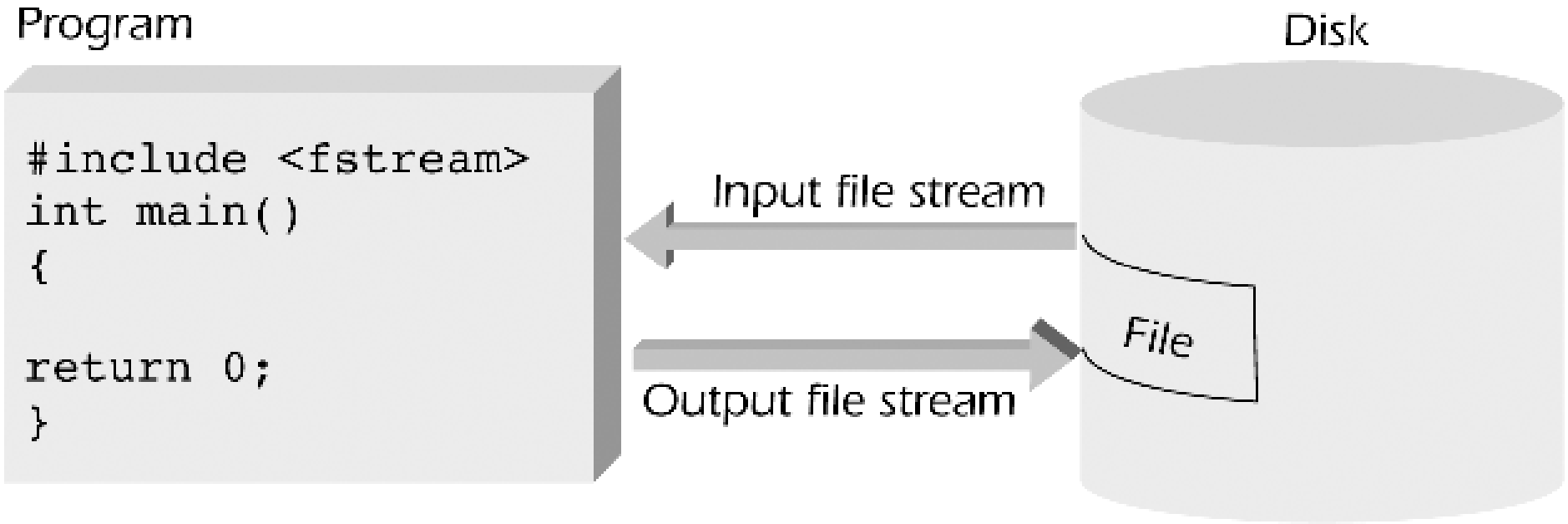
Contents

File Access

File stream modes:

Input: moves data from a physical file into a program

Output: sends or writes data to a file from a program



File Access

For **formatted input** or **output** of **files**, the functions **fscanf** and **fprintf** may be used.

These are identical to **scanf** and **printf**, except that the **first argument is a file pointer** that specifies the file to be read or written; the format string is the second argument.

```
int fscanf(FILE *fp, char *format, ...)

int fprintf(FILE *fp, char *format, ...)
```

File Access

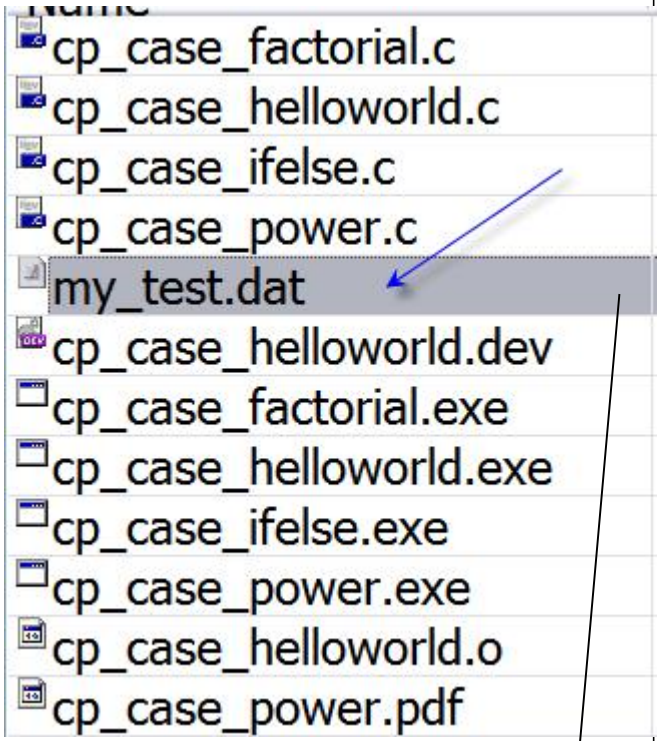
int fclose(FILE *fp)

is the inverse of fopen, it **breaks** the **connection** between the **file pointer** and the **external name** that was established by fopen, **freeing the file pointer for another file**.

Since most operating systems have some **limit** on **the number of files** that a program may have **open simultaneously**, it's a good idea to **free** the file **pointers** when they are **no longer needed.**

File Access

```
#include<stdio.h>
int main(void)
{
     char *str ="That's good news";
     int ind   = 617;
     FILE *fp  = NULL;

     fp=fopen("my_test.dat", "w");
     fputs("Your score of IELT", fp);
     fputc(':', fp);

     fprintf(fp, "%d\n", ind );
     fprintf(fp, "%s", str);

     fclose(fp);
}
```




```
Your score of IELT:617
That's good news
```

```
int read_IELT_file(void)
{

        char  *str     = NULL;
        char   mst[20] = {};
        int    ind     = 0;
        FILE  *fp      = NULL;

        fp=fopen("my_test.dat", "r");
        fgets(str, 24, fp);
        printf("%s", str);

        fscanf(fp, "%d", &ind);
        printf("%d", ind);
        putchar(fgetc(fp));

        fgets(mst, 17, fp);
        puts(mst);

        fclose(fp);
        getch();  return 0;
    }
```

Contents

Error Handling

The program signals errors in **two ways**.

First, the **diagnostic output** produced by **fprintf** goes to stderr, so it finds its way to the screen instead of disappearing down a pipeline or **into an output file**.

Second, the program uses the **standard library function exit**, which **terminates** program execution when it is **called**.

The argument of exit is available to whatever process called this one, so the **success** or **failure** of the program can **be tested** by another program that uses this one as a sub-process.

Error Handling

Conventionally, a **return value** of **0** signals that all is **well**; **non-zero** values usually signal **abnormal** situations.

exit calls fclose for each open output file, to flush out any buffered output.

```
static int SMAT_matlabcall_mxFree(
char *input_string /*I*/ )
{
      int status = SMAT_STATUS_INITIAL ;

    if ( input_string == NULL )
    {

      mexPrintf("SMAT_matlabcall_mxFree() NULL
input_string, Not need to free...\n");

    }
    else
    {
      mxFree( input_string );

      status = SMAT_STATUS_OK;
    }

      return status ;

}
```

Contents

Line Input and Output

The **standard library** provides an **input** and **output** routine **fgets**:

char *fgets(char *line, int maxline, FILE *fp)

fgets reads the **next** input line (including the newline) from file **fp** into the character array line at most **maxline-1** characters will be read.

The resulting line is terminated with **'\0'**. Normally fgets returns line on end of file or error it returns NULL.

Line Input and Output

For **output**, the function **fputs** writes a string (which need not contain a newline) to a file:

int fputs(char *line, FILE *fp)

It returns **EOF** if an error occurs, and non-negative otherwise.

The library functions gets and puts are similar to **fgets** and **fputs**, but operate on **stdin** and **stdout**. Confusingly, gets deletes the terminating **'\n'**, and puts adds it.

Contents

Miscellaneous Functions

The standard include files for ANSI C are:

assert.h	Assertions
ctype.h	Character identification
errno.h	Error handling
float.h	Max and Min values for floats
limits.h	limits for integral types
locale.h	Internationalisation info
math.h	Advanced math functions
setjmp.h	Non-local jump
signal.h	Exception handling
stdarg.h	Variable numbers of arguments
stddef.h	Standard definitions
stdio.h	Input/Output
stdlib.h	General Utilities
string.h	String Manipulation
time.h	Date and Time functions

Miscellaneous Functions

The **standard library** provides a wide variety of functions. This section is a brief synopsis of the most useful.

<string.h>
String Operations

strcat(s,t)	concatenate t to end of s
strncat(s,t,n)	concatenate n characters of t to end of s
strcmp(s,t)	return negative, zero, or positive for s < t, s == t, s > t
strncmp(s,t,n)	same as strcmp but only in first n characters
strcpy(s,t)	copy t to s
strncpy(s,t,n)	copy at most n characters of t to s
strlen(s)	return length of s
strchr(s,c)	return pointer to first c in s, or NULL if not present
strrchr(s,c)	return pointer to last c in s, or NULL if not present

Miscellaneous Functions

Character Class Testing and Conversion

<ctype.h>

int ungetc(int c, FILE *fp)

Storage Management

Mathematical Functions
<math.h>

Random Number generation
<stdlib.h>

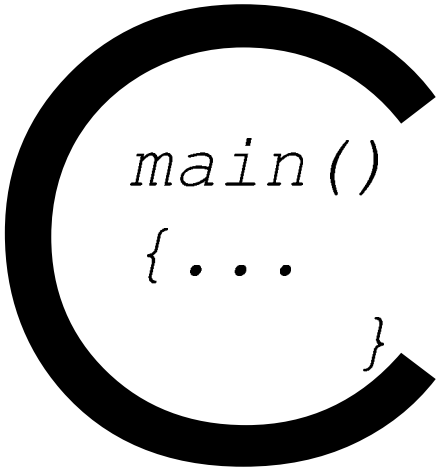
Miscellaneous Functions

One way to produce random floating-point numbers greater than or equal to zero but less than one is

(0,1)

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

Chen , Yi
leo.chen.yi@live.co.uk
30-Jul-2010



Input and Output

End

BETA 1.0.0.1

[七言.冬末春初.Glencoe一日游]

2015-02-21 Leo Chen

山远天高冬雪残，
孤舟长烟盈月寒，
飞花桥头初春暖，
桃红柳绿三月三。