This topic summaries the main ideas of Chapter 7 – Input and Output. The topic covers:

* Standard input and output
* Formatted Input and Output
* File Access
* Error Handling
* Line Input and output
* Miscellaneous functions
* The standard header <stdarg.h>
* Exercises

1. **Standard Input and Output**

A text stream consists of a sequence of lines; each line ends with a newline character. The simplest input mechanism is to read one character at a time from the standard input, normally the keyboard, with getchar:

*int getchar(void)*

getchar returns the next input character each time it is called, or EOF when it encounters end of file. The symbolic constant EOF is defined in **<stdio.h>.** The value is typically -1.

The function

*int putchar(int)*

Is used for output: putchar(c) puts the character c on the standard output, which is by default the screen. putchar returns the character written, or EOF is an error occurs.

Each source file that refers to an input/output library function must contain the line **#include <stdio.h>.** When the name is bracketed by < and > a search is made for the header in a standard set of places.

|  |
| --- |
| **#include <stdio.h>**  **int** **main**( ) {  **int** c;  **int** a;  printf( "Enter a value :");  c = getchar( );    printf( "\nYou entered: ");  a = putchar( c );  printf( "\na = %d\n", a);    **return** **0**;  } |

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows −

|  |
| --- |
| *Enter a value : Test putchar and getchar*  *You entered: T*  *a = 84* |

1. **Formatted Input and Output**

The output function printf translates internal values to characters.

int printf(char \*format, arg1, arg2, ...);

printf converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to printf. Each conversion specification begins with a % and ends with a conversion character.

Between the % and the conversion character there may be some character in order:

* A minus sign, which specifies left adjustment of the converted argument.
* A number that specifies the minimum field width. The converted argument will beprinted in a field at least this wide.
* A period, which separates the field width from the precision.
* A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.
* Character presented for variables.

**Note:** A width or precision may be specified as \*, in which case the value is computed by converting the next argument (which must be an int).

|  |
| --- |
| **#include <stdio.h>**  **int** **main**( ) {  **char** s[**12**] = "Hello, world";    printf(":s:\t\t:%s:\n",s);  printf(":10s:\t\t:%10s:\n",s);  printf(":15s:\t\t:%15s:\n",s);  printf(":.10s:\t\t:%.10s:\n",s);  printf(":-10s:\t\t:%-10s:\n",s);  printf(":.15s:\t\t:%.15s:\n",s);  printf(":-15s:\t\t:%-15s:\n",s);  printf(":15.10s:\t:%15.10s:\n",s);  printf(":-15.10s:\t:%-15.10s:\n",s);  printf(":\*.\*s:\t\t:%\*.\*s:\n", -**15**,**10**, s);  } |

The result is:

|  |
| --- |
| :s: :Hello, world:  :10s: :Hello, world:  :15s: : Hello, world:  :.10s: :Hello, wor:  :-10s: :Hello, world:  :.15s: :Hello, world:  :-15s: :Hello, world :  :15.10s: : Hello, wor:  :-15.10s: :Hello, wor :  :\*.\*s: :Hello, wor : |

A warning: printf uses its first argument to decide how many arguments follow and what their type is. It will get confused, and you will get wrong answers, if there are not enough arguments of if they are the wrong type.

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

int scanf(char \*format, ...)

scanf reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments. The format argument is described below; the other arguments, each of which must be a pointer, indicate where the corresponding converted input should be stored.

scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items

|  |
| --- |
| **#include <stdio.h>**  **int** **main**( ) {  **int** a,b,c,d;    d = scanf("%d %d %d",&a, &b, &c);    printf("Number of successful inputs: %d\n", d);  printf("a =%d b = %d c= %d\n", a, b, c);    **return** **0**;  } |

The result is:

|  |
| --- |
| 12 asdkfjakdj er  Number of successful inputs: 1  a =12 b = 0 c= 134513849 |

The function sprintf does the same conversions as printf does, but stores the output in a string:

int sprintf(char \*string, char \*format, arg1, arg2, ...);

sprintf formats the arguments in arg1, arg2, etc., according to format as before, but places the result in string instead of the standard output; string must be big enough to receive the result.

There is also a function sscanf that reads from a string instead of the standard input:

int sscanf(char \*string, char \*format, arg1, arg2, ...)

It scans the string according to the format in format and stores the resulting values through arg1, arg2, etc. These arguments must be pointers.

1. **File Access**

fopen negotiates with the operating system, and returns a pointer to be used in subsequent reads or writes of the file.

This pointer, called the file pointer, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred.

FILE \*fopen(char \*name, char \*mode);

FILE is a name type.

If a file that does not exist is opened for writing or appending, it is created if possible. Opening an existing file for writing causes the old contents to be discarded, while opening for appending preserves them. Trying to read a file that does not exist is an error, and there may be other causes of error as well, like trying to read a file when you don't have permission.

getc returns the next character from a file; it needs the file pointer to tell it which file.

int getc(FILE \*fp)

getc returns the next character from the stream referred to by fp; it returns EOF for end of file or error.

putc is an output function:

int putc(int c, FILE \*fp)

putc writes the character c to the file fp and returns the character written, or EOF if an error occurs.

When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called stdin, stdout, and stderr, and are declared in <stdio.h>. Normally stdin is connected to the keyboard and stdout and stderr are connected to the screen.

getchar and putchar can be defined in terms of getc, putc, stdin, and stdout as follows:

#define getchar() getc(stdin)

#define putchar(c) putc((c), stdout)

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

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

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

This is the example of program *cat* to concatenate files:

|  |
| --- |
| **#include <stdio.h>**  /\* cat: concatenate files, version 1 \*/  main(**int** argc, **char** \*argv[])  {  **FILE** \*fp;  **void** **filecopy**(**FILE** \*, **FILE** \*);    **if** (argc == **1**) /\* no args; copy standard input \*/  filecopy(stdin, stdout);  **else**  **while**(--argc > **0**)  **if** ((fp = fopen(\*++argv, "r")) == NULL) {  printf("cat: can't open %s\n, \*argv");  **return** **1**;  } **else** {  filecopy(fp, stdout);  fclose(fp);  }  **return** **0**;  }  /\* filecopy: copy file ifp to file ofp \*/  **void** filecopy(**FILE** \*ifp, **FILE** \*ofp)  {  **int** c;    **while** ((c = getc(ifp)) != EOF)  putc(c, ofp);  } |

The function

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. fclose is called automatically for each open file when a program terminates normally

1. **Error handling – Stderr and Exit**

stderr, is assigned to a program in the same way that stdin and stdout are. Output written on stderr normally appears on the screen even if the standard output is redirected.

This is the program cat using stderr:

|  |
| --- |
| **#include <stdio.h>**  /\* cat: concatenate files, version 1 \*/  main(**int** argc, **char** \*argv[])  {  **FILE** \*fp;  **void** **filecopy**(**FILE** \*, **FILE** \*);  **char** \*prog = argv[**0**]; /\* program name for errors \*/    **if** (argc == **1**) /\* no args; copy standard input \*/  filecopy(stdin, stdout);  **else**  **while**(--argc > **0**)  **if** ((fp = fopen(\*++argv, "r")) == NULL) {  fprintf(stderr, "%s: can't open %s\n",prog, \*argv);  exit(**1**);  } **else** {  filecopy(fp, stdout);  fclose(fp);  }  **if** (ferror(stdout)) {  fprintf(stderr, "%s: error writing stdout\n", prog);  exit(**2**);  }  exit(**0**);  }  /\* filecopy: copy file ifp to file ofp \*/  **void** filecopy(**FILE** \*ifp, **FILE** \*ofp)  {  **int** c;    **while** ((c = getc(ifp)) != EOF)  putc(c, ofp);  } |

the standard library function *exit* 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. 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.

The function ferror returns non-zero if an error occurred on the stream fp.

int ferror(FILE \*fp)

Although output errors are rare, they do occur (for example, if a disk fills up), so a production program should check this as well.

The function feof(FILE \*) is analogous to ferror; it returns non-zero if end of file has occurred on the specified file.

int feof(FILE \*fp)

1. **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.

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.

There are the code for fgets and fputs:

|  |
| --- |
| /\* fgets: get at most n chars from iop \*/  **char** \***fgets**(**char** \*s, **int** n, **FILE** \*iop)  {  **register** **int** c;  **register** **char** \*cs;  cs = s;  **while** (--n > **0** && (c = getc(iop)) != EOF)**147**  **if** ((\*cs++ = c) == '\n')  **break**;  \*cs = '\0';  **return** (c == EOF && cs == s) ? NULL : s;  }  /\* fputs: put string s on file iop \*/  **int** **fputs**(**char** \*s, **FILE** \*iop)  {  **int** c;  **while** (c = \*s++)  putc(c, iop);  **return** ferror(iop) ? EOF : **0**;  } |

1. **Miscellaneous functions**

* String Operations

The string functions found in <string.h>. In the following, s and t are char \*'s, and c and n are ints.

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

* Character Class Testing and Convention

Several functions from <ctype.h> perform character tests and conversions. In the following, c is an int that can be represented as an unsigned char or EOF. The function returns int.

isalpha(c) non-zero if c is alphabetic, 0 if not

isupper(c) non-zero if c is upper case, 0 if not

islower(c) non-zero if c is lower case, 0 if not

isdigit(c) non-zero if c is digit, 0 if not

isalnum(c) non-zero if isalpha(c) or isdigit(c), 0 if not

isspace(c) non-zero if c is blank, tab, newline, return, formfeed, vertical tab

toupper(c) return c converted to upper case

tolower(c) return c converted to lower case

* Command Execution

The function system(char \*s) executes the command contained in the character string s, then resumes execution of the current program.

* Storage Management

The functions malloc and calloc obtain blocks of memory dynamically.

void \*malloc(size\_t n)

returns a pointer to n bytes of uninitialized storage, or NULL if the request cannot be satisfied.

void \*calloc(size\_t n, size\_t size)

returns a pointer to enough free space for an array of n objects of the specified size, or NULL if the request cannot be satisfied. The storage is initialized to zero.

free(p) frees the space pointed to by p, where p was originally obtained by a call to malloc or calloc. There are no restrictions on the order in which space is freed, but it is a ghastly error to free something not obtained by calling malloc or calloc.

1. **The standard header <stdarg.h>**

The standard header <stdarg.h> contains a set of macro definitions that define how to step through an argument list.

The type va\_list is used to declare a variable (pointer list) that will refer to each argument in turn. The macro va\_start initializes this variable to point to the first unnamed argument. It must be called once before variable is used. There must be at least one named argument; the final named argument is used by va\_start to get

started. Each call of va\_arg returns one arg va\_arg uses a type name to determine what type to return and how big a step to take. Finally, va\_end does whatever cleanup is necessary. It must be called before the program returns.ument and steps variable to the next.

This example below will explain the function of each macro in <stdarg.h>:

|  |
| --- |
| **#include <stdio.h>**  **#include <stdarg.h>**  // Source: https://bobobobo.wordpress.com/2008/01/28/how-to-use-variable-argument-lists-va\_list/  **int** **addThemAll**( **int** numargs, ... )  {  // So this function can accept a variable number  // of arguments. No (practically speaking) limits.  // RULES you must know in order to be able to use "..." in one of your  // own functions:  //  // 1) The ... MUST appear exactly as ...  // It cannot be "..." (with the quotes),  // '...', or anything else weird.  //  // 2) The ... \_\_MUST GO LAST\_\_ IN THE ARGUMENT LIST  //  // 3) THERE MUST BE AT LEAST ONE MANDATORY, NON-OPTIONAL ARGUMENT,  // THAT COMES BEFORE THE ...  // We'll be using these macros here:  /\*  va\_list va\_start va\_end va\_arg  \*/  // All of the above va\_\* things are actually special MACROS,  // exclusively defined for us to use when working with  // \_V\_ariable \_A\_rgument lists.    // FIRST, we create a POINTER that will be used  // to point to the first element of the VARIABLE  // ARGUMENT LIST.  **va\_list** listPointer;  // Currently, listPointer is UNINITIALIZED, however,  // SO, now we make listPointer point to  // the first argument in the list  va\_start( listPointer, numargs );  // Notice that numargs is the LAST MANDATORY ARGUMENT  // that the addThemAll() function takes.  // By "LAST MANDATORY ARGUMENT", I mean 'numargs'  // is the last argument to the addThemAll() function  // JUST BEFORE the "..."  // NEXT, we're going to start to actually retrieve  // the values from the va\_list itself.  // THERE IS A CATCH HERE. YOU MUST KNOW THE  // DATA TYPE OF THE DATA YOU ARE RETRIEVING  // FROM THE va\_list. In this example, I'm assuming  // they're all ints, but you could always pass a format  // string that lets you know the types.  **int** sum = **0**;  **for**( **int** i = **0** ; i < numargs; i++ )  {  // GET an arg. YOU MUST KNOW  // THE TYPE OF THE ARG TO RETRIEVE  // IT FROM THE va\_list.  **int** arg = va\_arg( listPointer, **int** );  printf( " The %dth arg is %d\n", i, arg );  sum += arg;  }    printf("--");  printf("END OF ARGUMENT LIST\n\n");    // FINALLY, we clean up by saying  // va\_end(). Don't forget to do this  // BEFORE the addThemAll() function returns!  va\_end( listPointer );  printf("The total sum was %d\n\n", sum);  **return** sum;  }  **int** **main**()  {  // Try it out.  printf("Calling 'addThemAll( 3, 104, 29, 46 );' . . .\n");  addThemAll( **3**, **104**, **29**, **46** );  printf("Calling 'addThemAll( 8, 1, 2, 3, 4, 5, 6, 7, 8 );' . . .\n");  addThemAll( **8**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8** );  **return** **0**;  } |

1. **Exercises:**

These are solutions for chapter 7 exercises, which are added more comments to code for reader to following easily. These codes are get from <https://github.com/anotherlin/tcpl>.

Exercise 1:

|  |
| --- |
| **#include <ctype.h>**  **#include <stdio.h>**  **#include <string.h>**  /\*Exercise 7-1. Write a program that converts upper case to lower or lower case to upper,  depending on the name it is invoked with, as found in argv[0].\*/  **int** **main** (**int** argc, **char** \*argv[])  {  **int** c;  **while** ((c = getchar()) != EOF)  /\* Depending of the system, argv[0] may contain the full  \* calling path of the program. The simplest solution is to  \* use strstr() to search for the name. This is not perfect  \* however.  \*/  /\*check the name of execute file, whether it contains lower or not.  If have, convert all characters to lower case,  If not, convert all characters to upper case.\*/  putchar(strstr(argv[**0**], "lower") != NULL  ? tolower(c)  : toupper(c));    **return** **0**;  } |

Exercise 2:

|  |
| --- |
| **#include <ctype.h>**  **#include <stdio.h>**  **#define TABSIZE 8**  **#define LINESIZE 80**  /\*Exercise 7-2. Write a program that will print arbitrary input in a sensible way. As a  minimum, it should print non-graphic characters in octal or hexadecimal according to local custom, and break long text lines.\*/  **int** **main** ()  {  **int** column, c;  column = **0**;  **while** ((c = getchar()) != EOF) {  **if** (c == '\t')  column += TABSIZE;  **else** **if** (c == '\n')  column = **0**;  **else**  column++;  /\* Long lines are broken and non-graphic characters are printed  \* in hexadecimal.  \*/  /\*Show on the screen:  - If c can be showed:  check column versus LINESIZE, if column is bigger,  add new line, show c on the new line  - If c can not be showed:  show c in format of hexadecimal.  check column+3 versus LINESIZE, if column+3 is bigger,  add new line, show hexadecimal of c on the new line\*/  **if** (isprint(c) || c == '\t' || c == '\n') {  **if** (column > LINESIZE) {  putchar('\n');  column = c == '\t' ? TABSIZE : **1**;  }  putchar(c);  } **else** {  **if** ((column += **3**) > LINESIZE) {  putchar('\n');  column = **4**;  }  printf("\\x%.2x", c);  }  }    **return** **0**;  } |

Exercise 3:

|  |
| --- |
| **#include <ctype.h>**  **#include <stdarg.h>**  **#include <stdio.h>**  /\* All features mentionned in section 7.2, except computed width and precision,  \* are supported.  \*/  /\*Exercise 7-3. Revise minprintf to handle more of the other facilities of printf.\*/  /\*The ctype.h header file of the C Standard Library declares several functions that are useful for testing and mapping characters\*/  /\*The standard header <stdarg.h> contains a set of macro definitions that define how  to step through an argument list.\*/  **void** **minprintf** (**char** \*fmt, ...)  {  **va\_list** ap;  /\*The type va\_list is used to declare a variable that will refer to each  argument in turn. ap means argument pointer\*/  **char** conv[**100**], \*p, \*q, \*sval;  **long** lval;  **int** ival;  **double** dval;  **void** \*pval;  va\_start(ap, fmt);  /\*\*/  conv[**0**] = '%';  **for** (p = fmt; \*p != '\0'; p++)  **if** (\*p != '%')  putchar(\*p);  **else** {  p++;  q = &conv[**1**];  **if** (\*p == '-')  \*q++ = \*p++;  **while** (isdigit(\*p))  \*q++ = \*p++;  **if** (\*p == '.')  \*q++ = \*p++;  **while** (isdigit(\*p))  \*q++ = \*p++;  **if** (\*p == 'h' || \*p == 'l')  \*q++ = \*p++;    \*q++ = \*p;  \*q = '\0';  **switch** (\*p) {  **case** 'd': **case** 'i':  **case** 'o':  **case** 'x': **case** 'X':  **case** 'u':  **case** 'c':  **if** (\*(q - **2**) == 'l') {  lval = va\_arg(ap, **long**);  printf(conv, lval);  } **else** {  ival = va\_arg(ap, **int**);  printf(conv, ival);  }  **break**;  **case** 's':  sval = va\_arg(ap, **char** \*);  printf(conv, sval);  **break**;  **case** 'f':  **case** 'e': **case** 'E':  **case** 'g': **case** 'G':  dval = va\_arg(ap, **double**);  printf(conv, dval);  **break**;  **case** 'p':  pval = va\_arg(ap, **void** \*);  printf(conv, pval);  **break**;    **case** '%':  putchar('%');  **break**;  *default:*  printf("%s", conv);  **break**;  }  }    va\_end(ap); //clean up ap if necessary  }  **int** **main** ()  {  minprintf(":%s:\n", "hello, world");  minprintf(":%10s:\n", "hello, world");  minprintf(":%.10s:\n", "hello, world");  minprintf(":%-10s:\n", "hello, world");  minprintf(":%.15s:\n", "hello, world");  minprintf(":%-15s:\n", "hello, world");  minprintf(":%15.10s:\n", "hello, world");  minprintf(":%-15.10s:\n", "hello, world");  minprintf("%e\n", **3.1415926535**);  minprintf("%f\n", **3.1415926535**);  minprintf("%g\n", **3.1415926535**);  **return** **0**;  } |

Exercise 4:

|  |
| --- |
| **#include <ctype.h>**  **#include <stdarg.h>**  **#include <stdio.h>**  /\*Exercise 7-4. Write a private version of scanf analogous to minprintf from the previous  section.\*/  **int** **minscanf** (**char** \*fmt, ...)  {  **va\_list** ap;  **char** \*p, \*q, conv[**256**];  **int** n, no\_store, r;  va\_start(ap, fmt);  p = fmt;  n = **0**;  **while** (\*p != '\0') {  no\_store = **0**;  q = conv;  **for** ( ; ; ) {  **while** (\*p != '%' && \*p != '\0')    \*q++ = \*p++;  **if** (\*p == '%') {    \*q++ = \*p++;  **if** (\*p == '\*') {  \*q++ = \*p++;  no\_store = !**0**;  }  **while** (isdigit(\*p))  \*q++ = \*p++;  **if** (\*p == 'h' || \*p == 'l' || \*p == 'L')  \*q++ = \*p++;    **if** ((\*q++ = \*p) == '%')  p++;  **else**  **break**;  } **else**  **break**;  }  \*q = '\0';  **if** (no\_store)  r = scanf(conv);  **else**  **switch** (\*p) {  **case** 'd':  **case** 'i':  **case** 'o':  **case** 'u':  **case** 'x':  **if** (\*(q - **2**) == 'h') {    **short** \*sp;  sp = va\_arg(ap, **short** \*);  r = scanf(conv, sp);  } **else** **if** (\*(q - **2**) == 'l') {  **long** \*lp;    lp = va\_arg(ap, **long** \*);  r = scanf(conv, lp);  } **else** {  **int** \*ip;  ip = va\_arg(ap, **int** \*);  r = scanf(conv, ip);  }  **break**;  **case** 'c':  **case** 's': {  **char** \*cp;  cp = va\_arg(ap, **char** \*);  r = scanf(conv, cp);  **break**;  }  **case** 'e':  **case** 'f':  **case** 'g':  **if** (\*(q - **2**) == 'l') {  **double** \*dp;  dp = va\_arg(ap, **double** \*);  r = scanf(conv, dp);  } **else** **if** (\*(q - **2**) == 'L') {  **long** **double** \*ldp;    ldp = va\_arg(ap,  **long** **double** \*);  r = scanf(conv, ldp);  } **else** {  **float** \*fp;  fp = va\_arg(ap, **float** \*);  r = scanf(conv, fp);  }  **break**;  **case** '\0':  **default**:  r = scanf(conv);  **break**;  }  **if** (r == EOF) {  n = EOF;  **break**;  } **else** **if** (r == **0**)  **break**;  **else**  n += r;  **if** (\*p != '\0')  p++;  }  va\_end(ap);  **return** n;  }  **int** **main** ()  {  **int** c;  /\* To test, type "aaa%bbb 123" for example. This will make minscanf()  \* read 123.  \*/  printf("minscanf() returned %d ", minscanf("a aa%%b bb%d", &c));  printf("and read %d.\n", c);  **return** **0**;  } |

Exercise 5:

|  |
| --- |
| **#include <ctype.h>**  **#include <stdio.h>**  **#include <stdlib.h>**  **#define MAXOP 100**  **#define NUMBER '0'**  /\*Exercise 5-5. Rewrite the postfix calculator of Chapter 4 to use scanf and/or sscanf to do  the input and number conversion.\*/  **void** **push** (**double** f);  **double** **pop** (**void**);  **int** **getop** (**double** \*n);  **int** **main** ()  {  **int** type;  **double** n, op2;  **while** ((type = getop(&n)) != EOF)  **switch** (type) {  **case** NUMBER:  push(n);  **break**;    **case** '+':  **case** '\*':  **if** (type == '+')  push(pop() + pop());    **else**  push(pop() \* pop());  **break**;  **case** '-':  op2 = pop();  push(pop() - op2);  **break**;  **case** '/':  **if** ((op2 = pop()) != **0.0**)  push(pop() / op2);  **else**  printf("Error: Zero divisor!\n");  **break**;  **case** '\n':  printf("\t%.8g\n", pop());  **break**;  *default:*  printf("Error: Syntax Error!\n");  **return** -**1**;  }  **return** **0**;  }  **#define MAXVAL 100**  **int** sp = **0**;  **double** val[MAXVAL];  **void** **push** (**double** f)  {  **if** (sp < MAXVAL)  val[sp++] = f;  **else**  printf("Error: Stack is full, cannot push %g!\n", f);  }  **double** **pop** (**void**)  {  **if** (sp > **0**)  **return** val[--sp];  **else** {  printf("Error: Stack is empty, cannot pop!\n");  **return** **0.0**;  }  }  /\* Negative numbers are not supported as scanf() is unable to distinguish  \* "- 2" (substract then push 2) from "-2" (push -2).  \*/  **int** **getop** (**double** \*n)  {  **int** c;  **while** ((c = getchar()) == ' ' || c == '\t')  ;  **if** (c == EOF || c == '\n')  **return** c;  **else**  **switch** (c) {  **case** '+':  **case** '\*':  **case** '-':  **case** '/':  **return** c;  *default:*  ungetc(c, stdin);  **if** (scanf("%lf", n) == **1**)  **return** NUMBER;  **else**  **return** **0**;  }  } |

Exercise 6:

|  |
| --- |
| **#include <stdio.h>**  **#include <string.h>**  **#define BUFSIZE 80**  /\*Exercise 7-6. Write a program to compare two files, printing the first line where they differ\*/  **int** **main** (**int** argc, **char** \*argv[])  {  **FILE** \*f, \*g;    **if** (argc != **3**)  printf("Usage: diff file1 file2\n");  **else** **if** ((f = fopen(argv[**1**], "rb")) == NULL) {  fprintf(stderr, "Error: Cannot open %s!\n", argv[**1**]);  **return** -**1**;  } **else** **if** ((g = fopen(argv[**2**], "rb")) == NULL) {  fprintf(stderr, "Error: Cannot open %s!\n", argv[**2**]);  **return** -**1**;  } **else** {  **int** line;  **char** s[BUFSIZE], t[BUFSIZE], \*p, \*q;  /\* Lines are read in chunks of BUFSIZE characters. If the  \* files differ, the "current" chunks are printed, along with  \* the line number.  \*/  line = **1**;  **for** ( ; ; ) {  p = fgets(s, BUFSIZE, f);  q = fgets(t, BUFSIZE, g);  **if** (p == NULL || q == NULL || strcmp(s, t)) {  **if** (p != NULL) {  **if** (s[strlen(s) - **1**] == '\n')  s[strlen(s) - **1**] = '\0';  printf("%s: %d: %s\n",  argv[**1**], line, s);  }  **if** (q != NULL) {  **if** (t[strlen(t) - **1**] == '\n')  t[strlen(t) - **1**] = '\0';  printf("%s: %d: %s\n",  argv[**2**], line, t);  }  **break**;  } **else** **if** (s[strlen(s) - **1**] == '\n')  line++;  }  fclose(f);  fclose(g);  }  **return** **0**;  } |

Exercise 7:

|  |
| --- |
| **#include <stdio.h>**  **#include <stdlib.h>**  **#include <string.h>**  **#define MAXLINE 1000**  /\*Exercise 7-7. Modify the pattern finding program of Chapter 5 to take its input from a set of  named files or, if no files are named as arguments, from the standard input. Should the file  name be printed when a matching line is found?\*/    **int** **getline** (**FILE** \*fp, **char** \*line, **int** max);  **int** **main** (**int** argc, **char** \*argv[])  {  **int** c, except = **0**, number = **0**, found = **0**;  **while** (--argc > **0** && (\*++argv)[**0**] == '-')  **while** ((c = \*++argv[**0**]) != '\0')  **switch** (c) {  **case** 'x':  except = **1**;  **break**;  **case** 'n':  number = **1**;  **break**;  *default:*  fprintf(stderr,  "Error: Illegal option %c!\n",  c);  **return** -**1**;  }  **if** (!argc)  printf("Usage: find -x -n pattern [files]\n");  **else** {  **char** \*p, line[MAXLINE];  **FILE** \*fp;  p = \*argv++;  **if** (--argc)  fp = fopen(\*argv, "rb");  **else**  fp = stdin;  **for** ( ; ; ) {  **int** lineno = **0**;  **if** (fp == NULL) {  fprintf(stderr,  "Error: Cannot open %s!\n",  \*argv);  **return** -**1**;  }  **while** (getline(fp, line, MAXLINE) > **0**) {  lineno++;  **if** ((strstr(line, p) != NULL) != except) {  /\* Except for stdin, when a matching  \* line is found, name of the file is  \* printed.  \*/  **if** (fp != stdin)  printf("%s: ", \*argv);  **if** (number)  printf("%d: ", lineno);  printf("%s", line);  found++;  }  }  **if** (--argc)  fp = fopen(\*++argv, "rb");  **else**  **break**;  }  }  **return** found;  }  **int** **getline** (**FILE** \*fp, **char** \*line, **int** max)  {  **if** (fgets(line, max, fp) == NULL)  **return** **0**;  **else**  **return** strlen(line);  } |

Exercise 8:

|  |
| --- |
| **#include <ctype.h>**  **#include <stdio.h>**  **#define TABSIZE 8**  **#define LINESIZE 80**  /\*Exercise 7-8. Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file\*/  /\* Header and footer are 2 lines long each, so each page is a total of  \* PAGESIZE + 4 lines.  \*/  **#define PAGESIZE 40**  **char** \*filename;  **int** page, line;  **void** **printheader** (**void**)  {  printf("%.\*s\n\n", LINESIZE, filename);  page++;  line = **0**;  }  **void** **printfooter** (**void**)  {  **while** (line++ % PAGESIZE)  putchar('\n'); //insert \n until a page match number of defined lines  printf("\nPage %d\n", page);  }  **void** **printfile** (**FILE** \*fp)  {  **int** column, c;  column = **0**;  **while** ((c = fgetc(fp)) != EOF) {  **if** (c == '\t')  column += TABSIZE;  **else** **if** (c == '\n') {  putchar('\n');  **if** (++line % PAGESIZE == **0**) {  printfooter();  printheader();  }  column = **0**;  **continue**;  } **else**  column++;  /\*This part is the same as in exercise 7-2\*/  **if** (isprint(c) || c == '\t') {  **if** (column > LINESIZE) {  putchar('\n');  **if** (++line % PAGESIZE == **0**) {  printfooter();  printheader();  }  column = c == '\t' ? TABSIZE : **1**;  }  putchar(c);  } **else** {  **if** ((column += **3**) > LINESIZE) {  putchar('\n');  **if** (++line % PAGESIZE == **0**) {  printfooter();  printheader();  }  column = **4**;  }  printf("\\x%.2x", c);  }  }  }  **int** **main** (**int** argc, **char** \*argv[])  {  **if** (argc == **1**)  printf("Usage: print files\n");  **else**  **while** (--argc) {  **FILE** \*fp;  **if** ((fp = fopen(\*++argv, "rb")) == NULL) {  fprintf(stderr,  "Error: Cannot open %s!\n",  \*argv);  **return** -**1**;  }  filename = \*argv;  page = **0**;  printheader();  printfile(fp);  printfooter();  fclose(fp);  }  **return** **0**;  } |

Exercise 9:

|  |
| --- |
| **#include <stdio.h>**  /\*Exercise 7-9. Functions like isupper can be implemented to save space or to save time.  Explore both possibilities\*/  /\* The slow way is just to make comparisons. \*/  **int** **isupper\_slow** (**int** c)  {  **return** c >= 'A' && c <= 'Z';  }  /\* The fast way is to use a table look-up. c is supposed to have the value of  \* an unsigned char (0..255) or to be EOF (which we suppose is -1).  \*/  **int** **isupper\_fast** (**int** c)  {  **static** **const** **char** isupper[**257**] = {  **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**,  **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **1**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**, **0**,  };    **return** (isupper + **1**)[c];  }  **int** **main** ()  {  printf("isupper\_slow(EOF) = %d.\n", isupper\_slow(EOF));  printf("isupper\_fast(EOF) = %d.\n", isupper\_fast(EOF));  printf("isupper\_slow('A') = %d.\n", isupper\_slow('A'));  printf("isupper\_fast('A') = %d.\n", isupper\_fast('A'));  printf("isupper\_slow('Z') = %d.\n", isupper\_slow('Z'));  printf("isupper\_fast('Z') = %d.\n", isupper\_fast('Z'));  printf("isupper\_slow('z') = %d.\n", isupper\_slow('z'));  printf("isupper\_fast('z') = %d.\n", isupper\_fast('z'));  printf("isupper\_slow('0') = %d.\n", isupper\_slow('0'));  printf("isupper\_fast('0') = %d.\n", isupper\_fast('0'));  **return** **0**;  } |