CSE 230 Intermediate Programming in C and C++

Input/Output and Operating System

Fall 2017

Stony Brook University

Instructor: Shebuti Rayana

shebuti.rayana@stonybrook.edu

Outline

- Use of some input/output functions in the standard library, e.g. printf() and scanf() (although we have already used them, many details still need to be explained)
- Effect of various formats
- Input/output functions dealing with files and strings
- How to open file for processing and how to use a pointer to a file
- Operating system utilities
- Some more important tools, including compiler, make, touch, grep, beautifiers, and debuggers

Output Function `printf()`

- Two nice properties
  - *A list of arguments of arbitrary length can be printed*
  - *Printing is controlled by simple conversion specifications or formats*
- printf() delivers its character stream to the standard output file stdout, which is normally connected to the screen
- The argument list of printf() has two parts
  - *control_string and other_arguments*

Example printf()

- printf(“she sells %d %s for %f”, 99, “sea shells”, 3.77);
  - *control_String: “she sells %d %s for %f”*
  - *Other_arguments: 99, “sea shells”, 3.77*
- The expression in the other_arguments are evaluated and converted according to the formats in the control_string and then placed in the output stream.
- Characters in the control_string are not part of the format and placed directly in the output stream.
- The % symbol introduces a conversion specification
- A single-conversion specification is a string that begins with a % and ends with a conversion character.

printf() conversion characters

Conversion characters	how the corresponding argument is printed
c	character
d, i	decimal integer
u	unsigned decimal integer
o	unsigned octal integer
x, X	unsigned hexadecimal integer
e	floating-point number, 7.123000e+00
E	floating-point number, 7.123000E+00
f	floating-point number, 7.123000
g, G	floating-point number in e/E or f format which ever is shorter
s	string
p	pointer to void, printed as hexadecimal
n	pointer to integer into which the number of characters written so far is printed, argument not converted
%	with %% a single % is written to the output stream

More about printf()

- printf() returns as an integer the number of characters printed, in the following example
- *printf("she sells %d %s for %f", 99, "sea shells", 3.77);*

format	argument
%d	99
%s	"sea shells"
%f	3.77

- *Explicit formatting information can be included*
- *For %f 3.77 will be printed as 3.770000, with 6 digits after decimal point by default*

More about printf()

- Between % and conversion character there may be other conversion specifiers
  - *Zero or more flag characters*
  - *An optional positive integer that specifies the minimum field width of the converted argument*
  - *An optional precision, which is specified by a period followed by a nonnegative integer.*
- For integer d, i, o, u, x, X values minimum digits to be printed
- For float e, E, f number of digits after decimal point to be printed
- For g, G it specifies maximum number of significant digits
- For s, it specifies maximum number of characters to be printed

More about printf()

Declarations and initializations			
`char c = 'A', s[] = "Blue moon!";`			
Format	Corresponding argument	How it is printed in its field	Remarks
`%c`	`c`	`"A"`	field width 1 by default
`%2c`	`c`	`" A"`	field width 2, right adjusted
`%-3c`	`c`	`"A "`	field width 3, left adjusted
`%s`	`s`	`"Blue moon!"`	field width 10 by default
`%3s`	`s`	`"Blue moon!"`	more space needed
`%.6s`	`s`	`"Blue m"`	precision 6
`%-11.8s`	`s`	`"Blue moo "`	precision 8, left adjusted

More about printf()

Declarations and initializations			
`int i = 123;` `double x = 0.123456789;`			
Format	Corresponding argument	How it is printed in its field	Remarks
`%d`	`i`	`"123"`	field width 3 by default
`%05d`	`i`	`"00123"`	padded with zeros
`%7o`	`i`	`"    173"`	right adjusted, octal
`%-9x`	`i`	`"7b       "`	left adjusted, hexadecimal
`%-#9x`	`i`	`"0x7b     "`	left adjusted, hexadecimal
`%10.5f`	`x`	`"   0.12346"`	field width 10, precision 5
`%-12.5e`	`x`	`"1.23457e-01 "`	left adjusted, e-format

Input function `scanf()`

- Two nice properties
  - *A list of arguments of arbitrary length can be scanned*
  - *Input is controlled by simple conversion specifications or formats*
- scanf() reads characters from the standard input file stdin
- The argument list of scanf() has two parts
  - *control_string and other_arguments*

Example of scanf()

```
char a, b, c, s[100];
int n;
double x;
scanf("%c%c%c%d%s%lf", &a, &b, &c, &n, s, &x);
```

- control_String: "`%c%c%c%d%s%lf`"
- other_arguments: `&a, &b, &c, &n, s, &x`
- The other_argument following the control string consist of comma-separated list of pointer expressions or addresses
  - *Note in the above, writing &s would be wrong as s itself is an address*

Conversion characters

scanf() conversion characters		
Conversion character	Characters in the input stream that are matched	Corresponding argument is a pointer
c	any character, including white space	char
d, i	an optionally signed decimal integer	integer
u	an optionally signed decimal integer	unsigned integer
o	an optionally signed octal integer	unsigned integer
x, X	an optionally signed hexadecimal integer	unsigned integer
e, E, f, g, G	an optionally signed floating-point number	a floating type
s	a sequence of nonwhite space characters	char
p	what is produced by %p in printf(), usually an unsigned hexadecimal integer	void *
n, %, [...]	see the next table	

Conversion characters

scanf() conversion characters	
Conversion character	Remarks
n	No characters in the input stream are matched. The corresponding argument is a pointer to an integer, into which gets stored the number of characters read so far.
%	A single % character in the input stream is matched. There is no corresponding argument.
[...]	The set of characters inside the brackets [ ] is called the scan set. It determines what gets matched. (See the following explanation.) The corresponding argument is a pointer to the base of an array of characters that is large enough to hold the characters that are matched, including a terminating null character that is appended automatically.

More about scanf()

- Control string may contain
  - *white space, which matches optional white space in the input stream*
  - *Ordinary non-white space characters, other than %. Each must match to the next character in the input stream*
  - *Conversion specifications that begin with a % and end with a character. Between % and character, there may be optional * that indicates assignment suppression, followed by an integer that defines the maximum scan width, followed by optional h, l or L that modifies the specification.*
- Modifer h is used for integers which means the value stored in a short int or unsigend short int

More about scanf()

- Modifier l, which can precede integer or float conversion formats. For integers, it indicates long int or unsigned long int. For floats, it indicates double.
- Modifier L, precede float, indicates long double
- The characters in the input stream are converted to values according to the conversion specifications in the control string and placed at the address given by the corresponding pointer expression in the argument list.
- Except for character input, a scan field consists of contiguous nonwhite characters that are appropriate to the specified conversion. The scan field ends when a nonappropriate character is reached, or the scan width, if specified, is exhausted, whichever comes first.

More about scanf()

- When a string is read in, it is presumed that enough space has been allocated in memory to hold the string and an end-of-string sentinel \0, which will be appended. The format %1s can be used to read in the next nonwhite character. It should be stored in a character array of size at least 2.
- The format %nc can be used to read in the next n characters, including white space characters. When one or more characters are read in, white space is not skipped. In this case a null character is not appended.
- A format such as %1f can be used to read in a double. Floating numbers in the input stream are formatted as an optional sign followed by a digit string with an optional decimal point, followed by an optional exponent part. The exponential part consists of e or E, followed by an optional sign followed by a digit string.

More about scanf()

- A conversion specification of the form % [*string*] indicates that a special string is to be read in.
- If the first character in string is not a circumflex character ^ , then the string is to be made up only of the characters in *string*.
- If the first character in string is a circumflex, then the string is to be made up of all characters other than those in `string`.
- Example:
  - *the format %[abc] will input a string containing only the letters a, b, c, and will stop if any other character appears in the input stream, including a blank*
  - *The format %[^abc] will input a string terminated by any of a, b, or c, but not by space.*
  - scanf("%[^\n]s", sentence); // read input till you get a newline

More about scanf()

```
scanf("%[A B \n\t]", s);
```

- Read into a character array s a string containing A's, B's, spaces, tabs , newlines
- The specification %s skips white space and then reads in non-white space characters until a white space character is encountered or the end-of-file mark is encountered, whichever comes first.
- In contrast , the specification %5s skips white space and then reads in nonwhite characters, stopping when a white space character is encountered or an end-of-file mark is encountered or five characters been read in, whichever comes first.

More about scanf()

- The function scanf() returns the number of successful conversions performed.
- The value EOF is returned when the end-of-file mark is reached.
  - *Typically, this value is -1.*
  - *The value 0 is returned when no successful conversions are performed, and this is always different from EOF.*
- An inappropriate character in the input stream can frustrate expected conversions, causing the value 0 to be returned. As long as the stream can be matched to the control string, the input stream is scanned and values converted and assigned.
- The process stops if the input is inappropriate for the conversion specification.
- The value returned by scanf() can be used to test that occurred as expected, or to test that the end of the file was reached.

More about scanf()

- Example:

```
int i;

char c;

char string[15];

sccanf(“%d , %*s %% %c %5s %s”, &i, &c,
string, &string[5]);
```

- With the following characters in the input stream

```
45 , ignore_this % C read_in_this**
```

More about scanf()

- The value 45 is placed in i
- the comma is matched
- the string "ignore_this" is ignored
- the % is matched
- the character C is placed in the variable c
- the string" read_” is placed in string [0] through string [5] with the terminating \0 in string [5]
- Finally, the string “in_ this **" is placed in string [5] through stri ng [14], with string [14] containing \0.
- Because four conversions were successfully made, the value 4 is returned by scanf().

Functions fprintf(), fscanf()

- The functions fprintf() and fscanf() are file versions of printf() and scanf()
- They are used to take input from file
- Before knowing their use, we need to know how C deals with files.

FILE in C

- The identifier FILE is defined in stdio.h as a particular structure, with members that describe the current state of a file.
  - *To use files, a programmer need not know any details concerning this structure.*
  - *Also defined in stdio.h are the three file pointers stdin, stdout, and stderr. Even though they are pointers, we sometimes refer to them as files.*

Written in C	Name	Remark
stdin	Standard input file	Connected to keyboard
stdout	Standard output file	Connected to screen
stderr	Standard error file	Connected to screen

Functions fprintf(), fscanf()

- The function prototypes for file handling functions are given in stdio.h.
- prototypes for fprintf() and fscanf():

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

- A statement of the form

```
fprintf(file_ptr, control_string, other_arguments);
```

- writes to the file pointed to by file_ptr. The conventions for control_string and other_arguments conform to those of printf()

`fprintf(stdout, ... );` is equivalent to `printf( ... );`

- A statement of the form

```
fscanf (file_ptr, control_string , other_arguments);
```

- reads from the file pointed to by file_ptr.

`fscanf(stdi n, ... );` is equivalent to `scanf( ... );`

Functions sprintf(), sscanf()

- The functions sprintf() and sscanf() are string versions of the printf() and scanf(), respectively.
  - *Their function prototypes, found in stdio.h,*

```
int sprintf(char *s, const char *format, ...);

int sscanf(const char *s, const char *format, ...);
```

- The function sprintf() writes to its first argument, a pointer to char (string), instead of to the screen.
  - *Its remaining arguments conform to those for printf().*
- The function sscanf() reads from its first argument instead of from the keyboard.
  - *Its remaining arguments conform to those for scanf()*

Example: sprintf(), sscanf()

```
char str1[] = "1 2 3 go", str2[100] , tmp[100];
int a,  b, c;
sscanf(str1, "%d%d%d%s", &a, &b, &c, tmp);
sprintf(str2, "%s %s %d %d %d\n", tmp, tmp, a, b, c);
printf("%s", str2);
```

- The function sscanf() takes its input from str1. It reads three decimal integers and a string, putting them into a, b, c, and tmp, respectively.
- The function sprintf() writes to str2. More precisely, it writes characters in memory, beginning at the address str2. Its output is two strings and three decimal integers.
- To see what is in str2, we printf(). It prints the following on the screen:

go go 1 2 3

Functions sprintf(), sscanf()

- It is the programmer's responsibility to provide adequate space in memory for the output of sprintf().
- Reading from a string is unlike reading from a file in the following sense: If we sscanf() to read from str1 again, then the input starts at the beginning of the string, not where we left off before.

Functions fopen(), fclose()

- A file can be thought of as a stream of characters.
- After a file has been open, the stream can be accessed with file handling functions in the standard library.
- File has several important properties:
  - *Name*
  - *Must be opened and closed*
  - *Written to or read from or appended to*
  - *When it is open, we have access to it at its beginning or end*
  - *Until a file is opened nothing can be done with it*
  - *We have to tell the system which activity we want to perform on a file when we open it*
  - *When we finish using the file we close it*

Example

```
#include <stdio.h>
int main(void){
     int a, sum = 13;
     FILE *ifp, *ofp;
     ifp = fopen("my_file", ”r"); /*open for reading*/
     ofp = fopen("outfile", "w");/* open for wri ng */
     ……
```

- This opens two files in the current directory: my_file for reading and outfile for writing. (The identifier ifp is mnemonic for "infile pointer," and the identifier ofp is nmemonic for "outfile pointer.”)
- After a file has been opened, the file pointer is used exclusively in all references to the file.

Example

- Suppose that my_file contains integers. If we want to sum them and put the result in outfile, we can write

```
while (fscanf(ifp, “%d”, &a) == 1)
    sum += a;
fprintf(ofp, "The sum is %d.\n", sum);
```

- Note that fscanf(), like scanf(), returns the number of successful conversions.
- After we have finished using a file, we can write

```
fclose(ifp);
```

- This closes the file pointed to by ifp.

fopen()

- A function call of the form fopen (filename, mode) opens the named file in a particular mode and returns a file pointer.

mode	Meaning
“r”	Open text file for reading
“w”	Open text file for writing
“a”	Open text file for appending
“rb”	Open binary file for reading
“wb”	Open binary file for writing
“ab”	Open binary file for appending

- Each of these modes can end with a + character. This means that the file is to be opened for both reading and writing

fopen()

- Opening for reading a file that cannot be read, or does not exist, will fail. In this case fopen() returns a NULL pointer.
- Opening a file for writing causes the file to be created if it does not exist and causes it to be overwritten if it does.
- Opening a file in append mode causes the file to be created if it does not exist and causes writing to occur at the end of the file if it does.
- A file is opened for updating (both reading and writing) by using a + in the mode.
  - *between a read and a write or a write and a read there must be an intervening call to fflush() to flush the buffer, or a call to one of the file positioning function calls fseek(), fsetpos(), or rewind().*

fopen()

- In some operating systems, including UNIX, there is no distinction between binary and text files, except in their contents.
  - *The file mechanism is the same for both types of files.*
- In MS-DOS and other operating systems, there are different file mechanisms for each of the two types of files.

Example

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

void   double_space(FILE *, FILE *);
void   prn_info(char *);

int main(int argc, char **argv)
{
   FILE   *ifp, *ofp;

   if (argc != 3) {
      prn_info(argv[0]);
      exit(1);
   }
   ifp = fopen(argv[1], "r");      /* open for reading */
   ofp = fopen(argv[2], "w");      /* open for writing */
   double_space(ifp, ofp);
   fclose(ifp);
   fclose(ofp);
   return 0;
}

void double_space(FILE *ifp, FILE *ofp)
{
   int  c;

   while ((c = getc(ifp)) != EOF) {
      putc(c, ofp);
      if (c == '\n')
       putc('\n', ofp);   /* found a newline - duplicate it */
   }
}
```

Example

- Suppose we have compiled this program and put the executable code in the file dbl_space. When we give the command

```
dbl_space file1 file2
```

- the program will read from file1 and write to file2. The contents of file2 will be the same as file1, except that every newline character will have been duplicated.

```
void prn_info(char *pgm_name)
{
    printf("\n%s%s%s\n\n%s%s\n\n",
        "Usage:  ", pgm_name, "  infile  outfile",
        "The contents of infile will be double-spaced ",
        "and written to outfile.");
}
```

Using temporary files

- Library function `tmpfile()` to create a temporary binary file that will be removed when it is closed or on program exit.
- The file is opened for updating with “wb+”
  - *In MS-DOS binary files can be used as text files*
  - *In UNIX binary and text files are the same*
- Example: We read the contents of a file into a temporary file, capitalizing any letters as it does so. Then the program adds the contents of the temporary file to the bottom of the first file.

Example: *dbl_with_caps.c*

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

FILE *gfopen(char *filename, char *mode);
int main(int argc, char **argv)
{
        int c;
        FILE *fp, *tmp_fp;
        (if argc != 2) {
                fprintf(stderr, "\n%s%s%s\n\n%s\n\n" ,
                "Usage: ", argv [0], "  file name" ,
                "The file will be doubled and some letters capitalized.");
                exit(1);
        }
        fp = gfopen(argv[1] , "r+");
        tmp_fp = tmpfile();
        while ((c = getc(fp)) != EOF)
                putc(toupper(c), tmp_fp);
        rewind(tmp_fp);
        fprintf(fp, "---\n");
        while ((c = getc(tmp_fp)) != EOF)
                putc(c, fp);
        return 0;
}
```

Example: *Graceful Function*

```
FILE *gfopen(char *filename, char *mode)
{
        FILE *fp;
        if ((fp = fopen(filename, mode)) == NULL) {
                fprintf(stderr, "Cannot open %s - bye!\n", filename);
                exit(1);
        }
        return fp;
}
```

- If the file apple contains the following line

A is for apple and alphabet pie.

- After we give the command: *dbl_with_caps apple*
- the contents of the file will be

A is for apple and alphabet pie.

A IS FOR APPLE AND ALPHABET PIE.

Accessing a File Randomly

- The library functions `fseek()` and `ftell()` are used to access a file randomly.

```
ftell(file_ptr);
```

- Returns the current value of the file position indicator.
- The value represents the number of bytes from the beginning of the file, counting from zeros.
- Whenever a character is read from the file, the system increments the position indicator by 1.
- The file position indicator is a member of the structure pointed to by file_ptr.
- **Caution:** The file pointer itself does not point to individual characters in the stream. This is a conceptual mistake that many beginning programmers make.

Accessing a File Randomly

```
fseek(file_ptr, offset, place);
```

- It takes three parameters:
  - *a file pointer,*
  - *an integer offset, and*
  - *an integer that indicates the place in the file from which the offset should be computed.*
- fseek() sets the file position indicator to a value that represents offset bytes from place.
- The value for place can be 0, 1, or 2, meaning the beginning of the file, the current position, or the end of the file, respectively.
- Caution: The functions fseek() and ftell() are guaranteed to work properly only on binary files. In MS-DOS, if we want to use these functions, the file should be opened with a binary mode. In UNIX, any file mode will work.

Example: Writing a file backwards

```c
/* Write a file backwards. */

#include <stdio.h>
#define MAXSTRING 100

int main(void)
{
        char fname[MAXSTRING];
        int c;
        FILE *ifp;
        fprintf(stderr, "\nInput a filename: ");
        scanf("%s" , fname);
        ifp = fopen(fname, "rb"); /* binary mode for MS_DOS */
        fseek(ifp, 0, SEEK_END); /* move to end of the file */
        fseek(ifp, -1, SEEK_CUR); /* back up one character */
        while (ftell(ifp) > 0) {
                c = getc(ifp); /* move ahead one character */
                putchar(c);
                fseek(ifp, -2, SEEK_CUR); /* back up two characters */
        }
        return 0;
}
```

File Descriptor Input/Output

- A file descriptor is a nonnegative integer associated with a file.

File Name	Associated file descriptor
Standard input	0
Standard output	1
Standard error	2

- Functions in the standard library that use a pointer to FILE are usually buffered.
- Functions that use file descriptors may require programmer-specified buffers

Example: *change_case.c*

```c
/* Change the case of letters in a file. */

#include <ctype.h>
#include <fcntl.h>
#include <unistd.h> /* use io.h in MS_DOS */

#define BUFSIZE 1024
int main(int argc, char **argv)
{
        char mybuf[BUFSIZE], *p;
        int in_fd, out_fd, n;
        in_fd = open(argv[1], O_RDONLY);
        out_fd = open(argv[2] , O_WRONLY | O_EXCL | O_CREAT, 0600);
        while ((n = read(in_fd, mybuf, BUFSIZE)) > 0){
                for (p = mybuf; p - mybuf < n; ++p)
                        if (islower(*p))
                                *p = toupper(*p);
                        else if (isupper(*p))
                                *p = tolower(*p);
                write(out_fd, mybuf, n);
        }
        close(in_fd);
        close(ouLfd);
        return 0;
}
```

File Access Permissions

- In UNIX, a file is created with associated access permissions
- The permissions determine access to the file for the owner, the group, and for others.
- The access can be read, write, execute, or any combination of these, including none.
- When a file is created by invoking open(), a three-digit octal integer can be used as the third argument to set the permissions.
- Each octal digit controls read, write, and execute permissions.
  - *The first octal digit controls permissions for the user,*
  - *the second octal digit controls permissions for the group, and*
  - *the third octal digit controls permissions for others.*

File Access Permissions

Meaning of each octal digit in the file permissions		
Mnemonic	Bit representation	Octal representation
r--	100	04
-w-	010	02
--x	001	01
rw-	110	06
r-x	101	05
-wx	011	03
rwx	111	07

Now, if we pack three octal digits together into one number, we get the file access permissions.
The first, second, and third group of three letters refers to the user, the group, and others, respectively.

Example of File Access Permissions

Mnemonic	Octal representation
rw-------	0600
rw----r--	0604
rwxr-xr-x	0755
rwxrwxrwx	0777

The permissions rwxr-xr-x mean that

- the owner can read, write, and execute the file
- the group can read and execute the file
- that others can read and execute the file.
- In UNIX, the mnemonic file access permissions are displayed with the ls -l command.
- In MS-DOS, file permissions exist, but only for everybody.

Executing commands from within a C program

- The library function system() provides access to operating system commands.
  - *In both MS-DOS and UNIX, the command date causes the current date to be printed on the screen. If we want this information printed on the screen from within a program, we can write*

```
system("date");
```

- The string passed to system() is treated as an operating system command.
- When the statement is executed, control is passed to the operating system, the command is executed, and then control is passed back to the program.

Executing commands from within a C program

- In UNIX, vim is a commonly used text editor. Suppose that from inside a program we want to use vim to edit a file that has been given as a command line argument. We can write

```
char command[MAXSTRING];

sprintf(command, "vi %s", argv[1])'

printf("vi on the file %s is coming up ... \n", argv[1]);

system(command);
```

Using Pipes from within a C program

- UNIX system has popen() amd pclose() to communicate with the operating system.
  - *Not available in MS-DOS*

```
#include <ctype.h>
#include <stdio.h>
int main(void){
     int c;
     FILE *ifp;
     ifp = popen(”ls", "r");
     while ((c = getc(ifp)) != EOF)
          putchar(toupper(c));
     pclose(ifp);
     return 0;
}
```

Using Pipes from within a C program

- The first argument to popen() is a string that is interpreted as a command to the operating system; the second argument is a file mode, either “r” or "w".
- When the function is invoked, it creates a pipe between the calling environment and the system command that is executed. Here, we get access to whatever is produced by the ls command. Because access to the stream pointed to by ifp is via a pipe, we cannot use file positioning functions.
- For example, rewind (ifp) will not work.
- We can only access the characters sequentially. A stream opened by popen() should be closed by pclose().
  - *If the stream is not closed explicitly, then it will be closed by the system*

Environment Variables

- Environment variables are available in both UNIX and MS-DOS
- An environment variable is a dynamic-named value that can affect the way running processes will behave on a computer. They are part of the environment in which a process runs.
  - *For example, a running process can query the value of the TEMP environment variable to discover a suitable location to store temporary files, or the HOME or USERPROFILE variable to find the directory structure owned by the user running the process.*
- The following program prints the environment variables

```
#include <stdio.h>
int main(int argc, char *argv[], char *env[]){
     int i;
     for (i = 0; env[i] != NULL; ++i)
     printf("%s\n" , env[i]);
     return  0;
}
```

Environment Variables

- The third argument to main() is an array of strings.
- The system provides the strings, including the space for them.
- The last element in the array env is a NULL pointer.
- In UNIX system, this program prints something like

```
HOME=/c/c/blufox/center_manifold
SHELL=/bin/csh
TERM=vt102
USER=blufox
.....
```

- To the left of the equal sign is the environment variable; to the right of the equal sign is its value, which should be thought of as a string.
- You can also use getenv() to get the environment variables

How to time a C code

- Access to the machines internal clock is possible through some library functions in time.h
- This header file also contains a number of other constructs, including the type definitions for clock_t and time_ t, which are useful dealing with time.

```
typedef long clock_t
typedef long time_t
```

- Function prototypes:

```
clock_t clock(void)
time_t time(time_t *p)
double difftime(time_t time1, time_t time0)
```

How to time a C code

- When a program is executed, the operating system keeps track of the processor time that is being used. When clock() is invoked, the value returned is the system's best approximation to the time used by the program up to that point in seconds.
- The function time() returns the number of seconds that have elapsed since 1 January 1970 (UNIX timestamp - Epoch).
- If two values produced by time() are passed to difftime(), the difference expressed in seconds is returned as a double.

How to time a C code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAXSTRING 100

typedef struct {
        clock_t begin_clock, save_clock;
        time_t begin_time, save_time;
} time_keeper;

static time_keeper tk; /* known only to this file */

void start_time(void){
        tk.begin_clock = tk.save_clock = clock();
        tk.begin_time = tk.save_time = time(NULL);
}
  double prn_time(void){
          char s1[MAXSTRING], s2[MAXSTRING];
          int field_width, n1, n2;
          double clocks_per_second = (double) CLOCKS_PER_SEC, user_time, real_time;
          user_time = (clock() - tk.save_clock) / clocks_per_second;
          real_time = difftime(time(NULL) , tk.save_time);
          tk.save_clock = clock();
          tk.save_time = time(NULL);
          n1 = sprintf(s1, "%.1f" , user_time);
          n2 = sprintf(s2, "%.1f" , real_time);
          field width = (n1> n2)  ? n1 : n2;
          printf("%s%*.1f%s\n%s%8.1f%s\n\n"
          "User time: ", field_width, user_time, " seconds",
          "Real time: ", field_width, real_time, " seconds");
          return user_time;
  }
```

How to time a C code

```
/* Compare float and double multiplication times. */
#include <stdio.h>
#include "u_lib.h"
#define N 100000000 /* one hundred million */
int main (void){
      long i;
      float a, b = 3.333, c = 5.555;
      double x, Y = 3.333, z = 5.555;
      printf("Number of multiplies: %d\n\n", N);
      printf("Type float:\n\n");
      start_time();
      for (i = 0; i < N; ++i)
            a = b * c;
      prn_time();
      printf("Type double:\n\n");
      for (i = 0; i < N; ++i)
            x =  y * z;
      prn_time();
      return 0;
}
```