#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* do loop execution */
    LOOP:do
    {
        if( a == 15 )
        {
            /* skip the iteration */
            a = a + 1;
            goto LOOP;
        }
        printf("value of a: %d\n", a);
        a++;

    }while( a < 20 );

    return 0;
}
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* do loop execution */
    LOOP: do
    {
        if( a == 15 )
        {
            /* skip the iteration */
            a = a + 1;
            goto LOOP;
        }
        printf("value of a: %d\n", a);
        a++;
    } while( a < 20 );

    return 0;
}
```c
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* do loop execution */
    LOOP: do
    {
        if( a == 15 )
        {
            /* skip the iteration */
            a = a + 1;
            goto LOOP;
        }
        printf("value of a: %d\n", a);
        a++;
    }while( a < 20 );

    return 0;
}
```

Output

- value of a: 10
- value of a: 11
- value of a: 12
- value of a: 13
- value of a: 14
- value of a: 16
- value of a: 17
- value of a: 18
- value of a: 19
Extra-credit
Chapter 14:

Streams
A little bit of computer architecture ...

C runtime

My C program

Stream

OS device driver

KBD

Display

Printer

HDD

The (disk) files a computer user is familiar with are here!

Hardware I/O devices
<table>
<thead>
<tr>
<th>Name</th>
<th>Streams</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdin</td>
<td>Standard input</td>
<td>Keyboard</td>
</tr>
<tr>
<td>stdout</td>
<td>Standard output</td>
<td>Screen</td>
</tr>
<tr>
<td>stderr</td>
<td>Standard error</td>
<td>Screen</td>
</tr>
</tbody>
</table>

**NOTE**

In the days of DOS, there were two additional streams, `stdprn` (for the printer port) and `stdaux` (for the serial port). However, these were never ANSI Standard and were supported only by DOS and Windows. Seeing that a significant advantage of C programming is its portability, you should stick with the three streams listed in Table 14.1, and you won’t have any issues. However, in case you are reading an older book or run across some older code, it’s good to know these additional streams.
<table>
<thead>
<tr>
<th>Uses One of the Standard Streams</th>
<th>Requires a Stream Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf()</td>
<td>fprintf()</td>
<td>Formatted output</td>
</tr>
<tr>
<td>vprintf()</td>
<td>vfprintf()</td>
<td>Formatted output with a variable argument list</td>
</tr>
<tr>
<td>puts()</td>
<td>fputs()</td>
<td>String output</td>
</tr>
<tr>
<td>putchar()</td>
<td>putc(), fputc()</td>
<td>Character output</td>
</tr>
<tr>
<td>scanf()</td>
<td>fscanf()</td>
<td>Formatted input</td>
</tr>
<tr>
<td>vscanf()</td>
<td>vfscanf()</td>
<td>Formatted input with a variable argument list</td>
</tr>
<tr>
<td>gets()</td>
<td>fgets()</td>
<td>String input</td>
</tr>
<tr>
<td>getchar()</td>
<td>getc(), fgetc()</td>
<td>Character input</td>
</tr>
<tr>
<td>perror()</td>
<td></td>
<td>String output to stderr only</td>
</tr>
</tbody>
</table>

All these functions require that you include stdio.h. The function perror() may also require stdlib.h. The functions vprintf() and vfprintf() also require stdargs.h. On UNIX systems, vprintf() and vfprintf() may also require varargs.h. Your compiler’s Library Reference states whether any additional or alternative header files are needed.
What does `gets()` return?
/ stream.c--Demonstrates the equivalence of stream input and output. */
#include <stdio.h>

int main( void )
{
    char buffer[256];

    /* Input a line, then immediately output it. */
    puts(gets(buffer));

    return 0;
}

function
gets
char * gets ( char * str );

function
puts
int puts ( const char * str );

Output ▼
This is what's typed in.
This is what's typed in.
getchar() is **buffered**; we need to press ENTER after the character.

```c
#include <stdio.h>

#define MAX 80

int main( void )
{
    char ch, buffer[MAX+1];
    int x = 0;

    while ((ch = getchar()) != '\n' && x < MAX)
        buffer[x++] = ch;

    buffer[x] = '\0';

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

    return 0;
}
```
getch() is not a standard function; we must include conio.h for its prototype

getch() is not buffered; we don’t need to press ENTER after the character.
Practical tip

The character input functions (any of them!) can be used to make the program stop until the user presses a key, e.g.

```c
#include <stdio.h>

int main() {
    int c;
    c = getchar();
    c = getch();
    // or even just
    getchar();
    getch();
    return 0;
}
```

The C compiler allows to ignore the return value of a function!
Naming rule for I/O functions

If the function name starts with \textit{f}, the function can work with \textit{f}iles other than the standard ones (stdin, stdout, stderr)

\begin{itemize}
  \item puts()
  \item fputs()
  \item gets()
  \item fgets()
  \item getch()
  \item fgetch()
  \item Etc.
\end{itemize}
fgets()

#include <stdio.h>

#define MAXLEN 10

int main( void )
{
    char buffer[MAXLEN];

    puts("Enter text a line at a time; enter a blank to exit.");

    while (1)
    {
        fgets(buffer, \textcolor{green}{MAXLEN}, stdin);

        if (buffer[0] == \textbackslash n)
            break;

        puts(buffer);
    }
    return 0;
}

Unlike scanf() and gets(), fgets() does include potential newlines at end of strings!
fgets()
### TABLE 14.3 The Type Specifier Characters Used in `scanf()` Conversion Specifiers

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument</th>
<th>Meaning of Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d</code></td>
<td><code>int *</code></td>
<td>A decimal integer.</td>
</tr>
<tr>
<td><code>i</code></td>
<td><code>int *</code></td>
<td>An integer in decimal, octal (with leading 0), or hexadecimal (with leading 0x or 0X) notation.</td>
</tr>
<tr>
<td><code>o</code></td>
<td><code>int *</code></td>
<td>An integer in octal notation with or without the leading 0.</td>
</tr>
<tr>
<td><code>u</code></td>
<td><code>unsigned int *</code></td>
<td>An unsigned decimal integer.</td>
</tr>
<tr>
<td><code>x</code></td>
<td><code>int *</code></td>
<td>A hexadecimal integer with or without the leading 0x or 0X.</td>
</tr>
<tr>
<td><code>c</code></td>
<td><code>char *</code></td>
<td>One or more characters are read and assigned sequentially to the memory location indicated by the argument. No terminating \ 0 is added. If a field width argument isn’t given, one character is read. If a field width argument is given, that number of characters, including white space (if any), is read.</td>
</tr>
<tr>
<td><code>s</code></td>
<td><code>char *</code></td>
<td>A string of nonwhite-space characters is read into the specified memory location, and a terminating \ 0 is added.</td>
</tr>
<tr>
<td><code>a,e,f,g</code></td>
<td><code>float *</code></td>
<td>A floating-point number. Numbers can be input in decimal or scientific notation.</td>
</tr>
<tr>
<td><code>[...]</code></td>
<td><code>char *</code></td>
<td>A string. Only the characters listed between the brackets are accepted. Input ends as soon as a nonmatching character is encountered, the specified field width is reached, or Enter is pressed. To accept the ] character, list it first: []; [...]. A \ 0 is added at the end of the string.</td>
</tr>
<tr>
<td><code>[^...]</code></td>
<td><code>char *</code></td>
<td>The same as [...] , except that only characters not listed between the brackets are accepted.</td>
</tr>
<tr>
<td><code>%</code></td>
<td>None</td>
<td>Literal %: Reads the % character. No assignment is made.</td>
</tr>
</tbody>
</table>
### TABLE 14.4  The Precision Modifiers

<table>
<thead>
<tr>
<th>Precision Modifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hh</td>
<td>When placed before the type specifier d, i, o, u, x, x, or n, the modifier hh specifies that the argument is a pointer to a signed char or unsigned char.</td>
</tr>
<tr>
<td>h</td>
<td>When placed before the type specifier d, i, o, u, x, x, or n, the modifier h specifies that the argument is a pointer to type short int or unsigned short int.</td>
</tr>
<tr>
<td>l</td>
<td>When placed before the type specifier d, i, o, u, x, x, or n, the modifier l specifies that the argument is a pointer to type long or unsigned long.</td>
</tr>
<tr>
<td>ll</td>
<td>When placed before type specifier d, i, o, u, x, x, or n, the modifier ll specifies that the argument is a pointer to a long long or unsigned long long.</td>
</tr>
<tr>
<td>L</td>
<td>When placed before the type specifier a, A, e, E, f, F, g, or G, the modifier L specifies that the argument is a pointer to type long double.</td>
</tr>
</tbody>
</table>
How `scanf()` handles missing or extra chars.

```c
scanf("%d %d");
```

- The line the user inputs matches the format string. For example, suppose the user enters `12 14` followed by Enter. In this case, there are no problems. `scanf()` is satisfied, and no characters are left over in `stdin`.

- The line that the user inputs has too few elements to match the format string. For example, suppose the user enters `12` followed by Enter. In this case, `scanf()` continues to wait for the missing input. After the input is received, execution continues, and no characters are left over in `stdin`.

- The line that the user enters has more elements than required by the format string. For example, suppose the user enters `12 14 16` followed by Enter. In this case, `scanf()` reads the `12` and the `14` and then returns. The extra characters, the `1` and the `6`, are left waiting in `stdin`. 
Differences in how `scanf()` handles numbers, characters, and strings

Note what happens here!

The newline following 4 is ignored by the second `scanf`!

The newline following x is not ignored by the second `scanf`!

```c
int a, b;
char c, d;
char e[10], f[10];

puts("Enter two numbers on separate lines: ");
scanf("%d", &a);
scanf("%d", &b);
printf("%d %d\n", a, b);

puts("Enter two characters on separate lines: ");
scanf("%c", &c);
scanf("%c", &d);
printf("%c %c\n", c, d);

puts("Enter two strings on separate lines: ");
scanf("%s", e);
scanf("%s", f);
printf("%s %s\n", e, f);
```
How to tell `scanf()` to consume one character (without assigning it to anything)

* is called the assignment suppression flag

```c
int a, b;
char c, d;

puts("Enter two numbers on separate lines: ");
scanf("%d", &a);
scanf("%d%c", &b);
printf("%d %d\n", a, b);

puts("Enter two characters on separate lines: ");
scanf("%c%c", &c);
scanf("%c", &d);
printf("%c %c\n", c, d);
```
Another way to handle extra characters in `scanf()`

```c
// clearing.c -- Clearing stdin of extra characters.

#include <stdio.h>

void clear_kb(void);

int main( void )
{
    int age;
    char name[20];

    // Prompt for user's age.
    puts("Enter your age:");
    scanf("%d", &age);

    // Clear stdin of any extra characters.
    clear_kb();
}
```
// Now prompt for user's name:

puts("Enter your first name:");
scanf("%s", name);
// Display the data.

printf("Your age is %d.\n", age);
printf("Your name is %s.\n", name);

return 0;
}

void clear_kb(void)

// Clears stdin of any waiting characters.
{
    char junk[80];
    gets(junk);
}

Reads from stdin until the next newline. Most of the time, the next newline is the next character, left over from the previous scanf().
A third way to handle extra characters in `scanf()`

Portability problem: `fflush()` is only guaranteed by the standard to work with `stdout`. Although it works with `stdin` in MSVS under Windows, it does not work with gcc under Linux!
How to tell `scanf()` to match only characters in a given set

```c
char dna[20], junk[20];
scanf("%[GATCgatc]s", dna);
puts(dna);
scanf("%s", junk);
puts(junk);
```

The first `scanf` stops reading here.

The second `scanf` resumes reading here.
How to tell `scanf()` to match only characters not in a given set

```c
char non_numeric[20];

scanf("%[^0123456789]s", non_numeric);
puts(non_numeric);
```
Your turn: Write a `scanf()` to match only consonants

```c
char dna[20], junk[20];
scanf("%[GATCgatc]s", dna);
puts(dna);
scanf("%s", junk);
puts(junk);

char non_numeric[20];
scanf("%[^0123456789]s", non_numeric);
puts(non_numeric);
```
Your turn: Write a `scanf()` to match only consonants

What do you think about this solution?

```c
scanf("%[^aeiouAEIOU]s", input);
```
SKIP pp.347-58 in ch.14:

- Controlling output
- STDERR ()
To do for next time:

Read and understand `scanf()` examples in text: 14.7, 14.8, 14.9.
Chapter 17:

Using Disk files

Continued from Ch.14 ...
C has two types of streams/files

• Text:
  – The stream/file is a sequence of lines
  – Each line ends with *newline* `\n`
  – Lines can be empty
  – Lines are not strings! (No terminator `\0`)

• Binary
  – There are no lines
  – Data is read and written as a continuous stream
  – No `\n` (nor `\0`)
Common misconceptions

• Text:
  – The \textit{newline} character '\n' that C uses \textit{internally} is not to be confused with the \textit{external} one in the disk file.
  – Most C compilers use ASCII 10 (LF) for '\n'.
  – \textbf{Newline} in the Unix/Linux OS is also LF, but MS Windows uses CR+LF, while other OSs use LF+CR, CR, etc.

• Binary
  – No individual bits are transferred!
  – The smallest unit of information is a Byte.
C views each file simply as a **sequential stream of bytes**.

It ends *as if* there is an **end-of-file** marker.

Each OS has its own way to represent the **EOF**.

A file with **n** bytes.
Filenames and paths

• MS Windows uses backslash as path separator
  – The backslash is a special character in C, so it must be doubled:
    `char *filename = "c:\\my_dir\\my_file.txt"`

• Unix/Linux does not have this problem, as the separator is forward slash:
  `char *filename = "c:/my_dir/my_file.txt"`
Opening a file

```c
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

Structure, also declared in stdio.h

*const char* means that fopen() cannot modify these strings
FILE and EOF in stdio.h

```c
typedef struct {
    char *fpos; /* Current position of file pointer (absolute address) */
    void *base; /* Pointer to the base of the file */
    unsigned short handle; /* File handle */
    short flags; /* Flags (see FileFlags) */
    short unget; /* 1-byte buffer for ungetc (b15=1 if non-empty) */
    unsigned long alloc; /* Number of currently allocated bytes for the file */
    unsigned short buffincrement; /* Number of bytes allocated at once */
} FILE;

#define EOF (-1)
```
Opening a file

FILE *fopen(const char *filename, const char *mode);

If an error occurs, fopen() returns the NULL pointer (remember malloc())?

The argument mode specifies the mode of opening the file. The valid values for this argument are described in the following table:
<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Open a text file for reading.</td>
</tr>
<tr>
<td>w</td>
<td>Truncate to zero length or create a text file for writing.</td>
</tr>
<tr>
<td>a</td>
<td>Append; open or create a text file for writing at the end-of-file.</td>
</tr>
<tr>
<td>rb</td>
<td>Open a binary file for reading.</td>
</tr>
<tr>
<td>wb</td>
<td>Truncate to zero length or create a binary file for writing.</td>
</tr>
<tr>
<td>ab</td>
<td>Append; open or create a binary file for writing at the end-of-file.</td>
</tr>
<tr>
<td>r+</td>
<td>Open a text file for read/write.</td>
</tr>
<tr>
<td>w+</td>
<td>Truncate to zero length or create a text file for read/write.</td>
</tr>
<tr>
<td>a+</td>
<td>Append; open or create a text file for read/write. You can read data anywhere in the file, but you can write data only at the end-of-file.</td>
</tr>
<tr>
<td>r+b or rb+</td>
<td>Open a binary file for read/write.</td>
</tr>
<tr>
<td>w+b or wb+</td>
<td>Truncate to zero length or create a binary file for read/write.</td>
</tr>
<tr>
<td>a+b or ab+</td>
<td>Append; open or create a binary file for read/write. You can read data anywhere in the file, but you can write data only at the end-of-file.</td>
</tr>
</tbody>
</table>
All files which are opened and associated with streams should be closed before a program terminates.

Function `fclose()` shall be used to close a file opened by function `fopen()`. It is prototyped as

```
int fclose(FILE *stream);
```

`fclose()` causes the stream pointed to by `stream` to be flushed and the associated file to be closed. It returns 0 when successful. Otherwise, it returns non-zero value.
Very Short Example – just opening and closing two files:

```c
#include <stdio.h>
void main()
{
    FILE *fpt1, *fpt2;

    fpt1 = fopen("filename", "w");
    fpt2 = fopen("filename", "r");
    ..........
    fclose(fpt1);
    fclose(fpt2);
}
```
Short Example – opening and closing two files, testing for errors:

```c
#include <stdio.h>
void main()
{
    FILE *fpt1, *fpt2;

    if((fpt1 = fopen("filename", "w")) == NULL)
        printf("Cannot create or open the file
");
    if((fpt2 = fopen("filename", "r")) == NULL)
        printf("Cannot open the file
");

    ............
    fclose(fpt1);
    fclose(fpt2);
}
```
Oh, no, another character input function?! 😊

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

int main( void )
{
    FILE *fp;
    char ch, filename[40], mode[4];

    while (1){
        puts("\nEnter a filename: ");
        gets(filename);
        puts("\nEnter a mode (max 3 characters): ");
        gets(mode);

        // Try to open the file.
        if ( (fp = fopen( filename, mode )) != NULL )
        {
            printf("\nSuccessful opening %s in mode %s.\n", filename, mode);
            fclose(fp);
            puts("Enter x to exit, any other to continue.");
            if ( (ch = getc(stdin)) == 'x' )
                break;
        }
    }
}```
else
    continue;
} else
{
    fprintf(stderr, "Error opening file %s in mode %s.\n", filename, mode);
    puts("Enter x to exit, any other to try again.");
    if ( (ch = getc(stdin)) == 'x')
        break;
    else
        continue;
}
return;
Reading and Writing Characters

fgetc() reads one character from a file

```
int fgetc(FILE *stream);
```

fgetc(stdin) is equivalent to getchar()

fputc() writes one character to a file

```
int fputc(int c, FILE *stream);
```

fputc('a', stdout) is equivalent to putchar('a')
/ Function that copies a file char by char */

int copyfile(const char *inputfile, const char *outputfile) {
    FILE *fp1, *fp2;
    char c;
    if((fp1 = fopen(inputfile, "rb")) == NULL)
        return -1;
    if((fp2 = fopen(outputfile, "wb")) == NULL) {
        fclose(fp1);
        return -1;
    }
    c = fgetc(fp1);
    while(!feof(fp1)) {
        fputc(c, fp2);
        c = fgetc(fp1);
    }
    fclose(fp1);
    fclose(fp2);
    return 0;
}

Why binary?
To avoid complications due to newline representations in different OS

In a text file, we could use
while (c != EOF) instead.
fgets() reads a line from a file

    char *fgets(char *s, int n, FILE *stream);

fputts() writes a line to a file

    int fputs(const char *s, FILE *stream);
int copyfile2(const char *inputfile, const char *outputfile) {
    FILE *fp1, *fp2;
    char line[BUFSIZE];

    if((fp1 = fopen(inputfile, "rb")) == NULL) {
        return -1;
    }
    if((fp2 = fopen(outputfile, "wb")) == NULL) {
        fclose(fp1);
        return -1;
    }
    fgets(line, BUFSIZE, fp1);
    while(!feof(fp1)) {
        fputs(line, fp2);
        fgets(line, BUFSIZE, fp1);
    }
    fclose(fp1);
    fclose(fp2);
    return 0;
}
General Reading and Writing

`fscanf/fprintf` are the file processing equivalents of `scanf` and `printf`.

```c
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

The function calls

```c
printf(format, arglist);
scanf(format, arglist);
```

are equivalent to

```c
fprintf(stdout, format, arglist);
fscanf(stdin, format, arglist);
```
// fscanfnums.c -- Reading formatted file data with fscanf().

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

int main( void )
{
    float f1, f2, f3, f4, f5;
    FILE *fp;

    if ( (fp = fopen("numberinput.txt", "r")) == NULL )
    {
        fprintf(stderr, "Error opening file.\n");
        exit(1);
    }

    fscanf(fp, "%f %f %f %f %f", &f1, &f2, &f3, &f4, &f5);
    printf("The values are %f, %f, %f, %f, and %f\n."
            , f1, f2, f3, f4, f5);

    fclose(fp);
    return(0);
}
The values are 123.449997, 87.000999, 100.019997, 0.004560, and 1.000500
Function `feof()`

- Tests a stream for end-of-file (EOF).
- Prototype:

  ```c
  int feof(FILE *stream);
  ```

- Returns nonzero if and only if the end-of-file indicator is set for the `stream` argument.
int feof(FILE *stream);

Why do we need an extra function? Can’t we just test for -1, like this?

```c
while ((c = fgetc(fp)) != EOF)
```

```c
#define EOF (-1)
```
What does this program do?

Explain line-by-line!

```c
#include <stdio.h>

int main ()
{
    FILE *fp;
    int c;

    fp = fopen("file.txt","r");
    while(1)
    {
        c = fgetc(fp);
        if( feof(fp) )
        {
            break ;
        }
        printf("%c", c);
    }
    fclose(fp);
    return(0);
}
```

Source: http://www.tutorialspoint.com/c_standard_library/c_function_fwrite.htm
What does this program do?

Now give a short, high-level explanation!
3. **BUG BUSTER:** Is anything wrong with the following code?

```c
FILE *fp;
int c;

if ( ( fp = fopen( oldname, "rb" ) ) == NULL )
    return -1;

while ( ( c = fgetc( fp ) ) != EOF )
    fprintf( stdout, "%c", c );

fclose ( fp );
```
Direct File I/O
Remember the FILE struct (in stdio.h)?

```c
typedef struct {
    char *fpos; /* Current position of file pointer (absolute address) */
    void *base; /* Pointer to the base of the file */
    unsigned short handle; /* File handle */
    short flags; /* Flags (see FileFlags) */
    short unget; /* 1-byte buffer for ungetc (b15=1 if non-empty) */
    unsigned long alloc; /* Number of currently allocated bytes for the file */
    unsigned short buffinc; /* Number of bytes allocated at once */
} FILE;
```

When a file is opened in **append** mode, `fpos` points to the end of the file.
In any other mode, `fpos` initially points to the beginning of the file.
Sequential vs. random access to files

All I/O functions covered so far update **fpos** automatically, advancing it **sequentially** through the file.

The C library also has functions that allow to manipulate **fpos** directly; this is called **random** access to the file, i.e. we can move from point A to point B inside the file w/o reading all intervening elements.
fp = fopen( "file.txt" , "r" );
while(1)
{
    ch = fgetc(fp);
    if(feof(fp) )
    {
        break ;
    }
    printf("%c", ch);
}
rewind(fp);
printf("\n");
while(1)
{
    ch = fgetc(fp);
    if( feof(fp) )
    {
        break ;
    }
    printf("%c", ch);
}
fclose(fp);
ftell() and fseek()

```c
fp = fopen("file.txt", "r");
if( fp == NULL )
{
    perror ("Error opening file");
    return(-1);
}
seek(fp, 0, SEEK_END);
len = ftell(fp);
close(fp);
printf("Total size of file.txt = %d bytes\n", len);
```

Other possible origin values: 
SEEK_SET (beginning) and SEEK_CUR (current position)

Offset in Bytes

Return offset of current position, in Bytes, from beginning
QUIZ:

Explain in your own words the difference between **sequential** and **random** access to files.
Write a program that:

• Writes 10 doubles to a file `temp.dat`

• Reads only the second, fifth, and last numbers, in the variables `a`, `b`, and `c`, respectively

• Calculates and displays `ab+c`

• Displays the length of the file in Bytes

• Repositions `fpos` at the beginning of the file
  – This can be done 4 different ways!
File management

High-level operations on existing files:
• Delete
• Rename
• Copy
• Etc.
int main( void )
{
    char filename[80];

    printf("Enter the filename to delete: ");
    gets(filename);

    if ( remove(filename) == 0 )
        printf("The file %s has been deleted.\n", filename);
    else
        fprintf(stderr, "Error deleting the file %s.\n", filename);
    return(0);
}
int main( void )
{
    char oldname[80], newname[80];

    printf("Enter current filename: ");
    gets(oldname);
    printf("Enter new name for file: ");
    gets(newname);

    if ( rename(oldname, newname ) == 0 )
        printf("%s has been renamed %s.\n", oldname, newname);
    else
        fprintf(stderr, "An error has occurred renaming %s.\n", oldname);
    return(0);
}
User-defined copy function

```c
int file_copy( char *oldname, char *newname )
{
    FILE *fold, *fnew;
    int c;

    /* Open the source file for reading in binary mode. */
    if ( ( fold = fopen( oldname, "rb" ) ) == NULL )
        return -1;

    /* Open the destination file for writing in binary mode. */
    if ( ( fnew = fopen( newname, "wb" ) ) == NULL )
    {
        fclose( fold );
        return -1;
    }
```
User-defined copy function

/* Read one byte at a time from the source; if end of file */
/* has not been reached, write the byte to the */
/* destination. */

while (1)
{
  c = fgetc( fold );

  if ( !feof( fold ) )
    fputc( c, fnew );
  else
    break;
}

fclose ( fnew );
fclose ( fold );

return 0;
tmpnam()

char buffer[10], *c;

// Get a temporary name in the defined buffer.
tmpnam(buffer);

/* Get another name, this time in the function's internal buffer. */
c = tmpnam(NULL);

// Display the names.
printf("Temporary name 1: %s", buffer);
printf("\nTemporary name 2: %s\n", c);
char buffer[10], *c;

// Get a temporary name in the defined buffer.
tmpnam(buffer);

/* Get another name, this time in the function's internal buffer. */
c = tmpnam(NULL);

// Display the names.
printf("Temporary name 1: %s", buffer);
printf("\nTemporary name 2: %s\n", c);
#include <direct.h>

void main()
{
    _mkdir("C:\\hello");
}

Not in text
To do for next time:

• Read and take notes: pp.431-52 of our text.
• Read and understand Listings 17.4, 17.5, 17.6.
• All quizzes.
Exercise 2

Problem not from text:

- Open a file `primes.dat` and write in it the prime integers from 2 to 15. Close the file.
- Reopen in `append` mode and add the remaining primes up to 20. Close the file.
- Reopen and read all the numbers in an array `primes`. Print the array.