Tutorial 8: Character Counter

Topics and References

- Arrays. See the lecture presentation deck, the lecture handout, and/or this page for an introduction on arrays and implementing functions with array parameters. Chapter 8 and Section 12.3 of the text contain information on arrays. Chapter 9 of the text contains information on functions.
- Characters and character strings. See the presentation deck on characters and character strings. Section 7.3 of the text contains information on characters while Chapter 13 discusses character strings.
- Pointers. See the presentation deck on pointers and Chapter 11 of the text, and/or this page for an introduction to pointers.
- Assertions. See the brief tutorial at the end of this document for assertions in C.

Learning Outcomes

- Using arrays to store homogeneous data
- Looping over arrays
- Implementing functions with array parameters
- Pointer dereferencing and implementing functions with pointer parameters
- Standard library character I/O
- Assertions and `assert` macro

Task: Computing Frequencies

The objective of this tutorial is to build a program, organized by functions, to count frequencies of Latin characters, control characters (such as tabs, newlines), and other characters (such as digits and symbols such as $+$). This tutorial is partially inspired by the Unix `wc` program that prints newline, word, and byte counts for a file. You'll have to implement three functions that are explained below. Read this carefully: for this tutorial, you cannot use any of the C standard library functions declared in `<ctype.h>` and `<string.h>` header files. Other than functions `fgetc`, `printf` or `fprintf` (which are declared in `<stdio.h>`), you must not use any other C standard library function in your submission. Read this again:

> ***Other than functions `fgetc`, `printf` or `fprintf` (which are declared in `<stdio.h>`), you must not use any other C standard library function in your submission. You can include `<stdbool.h>` to use the `bool` type and `true` and `false` constants.***

Initialization

Declare and define a function to zero out variables that accumulate frequencies of Latin characters, control characters, and non-Latin characters. The prototype of the function will look like this:

```c
void initialize(int latin_freq[], int size, int *ctrl_cnt,
                int *non_latin_cnt);
```

The function must zero out the `size` number of elements of array parameter `latin_freq` and the `int` variables pointed to by pointer parameters `ctrl_cnt` and `non_latin_cnt`. This zeroing out will allow other functions to correctly compute frequencies.

Computing frequencies

Declare and define a function that take an input file stream (to a text file) and computes the frequencies of Latin characters, control characters, and non-Latin characters. The prototype of the function will look like this:

```c
void wc(FILE *ifs, int latin_freq[], int *ctrl_cnt, int *non_latin_cnt);
```

Parameter `ifs` is initialized with an input file stream to a text file that was opened by driver function `main`. Function `wc` will use C standard library function `fgetc` to read individual characters in the file stream. See the brief tutorial on I/O with files in this document to understand the usage of function `fgetc`.

If a character is a Latin character (the program is not case sensitive), the corresponding element in array `latin_freq` must be incremented. That is, if a character is either `'a'` or `'A'`, subscripted variable `latin_freq[0]` is incremented. If a character is either `'c'` or `'C'`, subscripted variable `latin_freq[2]` must be incremented. Likewise, if the character is either `'z'` or `'Z'`, subscripted variable `latin_freq[25]` must be incremented. Notice that the size of array `latin_freq` is not passed as a parameter to function `wc`. Assume the size of array `latin_freq` is at least 26 (which is the number of either uppercase or lowercase Latin characters).

If the character is a control character (composed of character constants `'\a'` - alarm, `'\b'` - backspace, `'\f'` - formfeed page break, `'\n'` - newline, `'\r'` - carriage return, `'\t'` - horizontal tab, `'\v'` - vertical tab), then the `int` variable pointed to by `ctrl_cnt` must be incremented.

The `int` variable pointed to by `non_latin_cnt` will be incremented for all other (meaning non-Latin and non-control) characters.

It is good practice to break this function into smaller functions. Since you cannot include `<ctype.h>`, you must devise functions to identify uppercase and lowercase Latin characters and to identify whether a character is a control character.

Printing frequencies

Function `print_freqs` prints the frequencies of Latin letters followed by the total count of Latin characters, non-Latin characters, and control characters. The function declaration will look like this:

```c
void print_freqs(int const latin_freq[], int size, int const *ctrl_cnt,
                 int const *non_latin_cnt);
```

The format used to write the data can be determined from the sample output files. To get the same format as the output file, you will have to use 5 as the width parameter of the `printf` or `fprintf` functions. If you've forgotten about the width parameter, review here.

Assertions

Notice that at the bottom of function `main`, I've inserted a postcondition assertion that checks if you've implemented the task correctly. The assertion is straightforward: if the input text file's file size is not equivalent to the sum of the frequencies of Latin characters, non-Latin characters, and control characters, then the program will be terminated. This means that you've computed one of these values incorrectly and you must verify the code involved in these computations.

You can find a quick introduction to assertions and the `assert` macro at the bottom of this document.

Header and source files

As usual, you'll declare the required functions in `q.h` and define these functions in `q.c`. Make sure that `q.h` only includes `<stdio.h>` (`<stdbool.h>` is also permitted) and source file `q.c` only includes header file `q.h`. Download `qdriver.c` to obtain an idea of the structure of driver function `main`. As explained in previous tutorials, begin by implementing *stub functions* for the functions declared in `q.h`. Complete the definition and testing of a stub function before moving on to the next stub function. By now, you are expected to be familiar with the process of compiling source files, linking object files and C standard library functions into an executable file, and then testing the output from your program for certain inputs with the correct output.

The executable requires the name of a text file that will be parsed by the program. Therefore, you'll need to run your program `q.out` like this:

```
$ ./q.out input1.txt
```

Since the frequencies are printed horizontally, it would be better to redirect the output to a text file:

```
$ ./q.out input.txt > output.txt
```

If you're confident about characters and character strings and can devise algorithms and implement the required functions, you can proceed with the tutorial. Otherwise, you may wish to read the review of characters, file I/O, and assertions located at the bottom of this document.

Submission and automatic evaluation

1. In the course web page submit `q.c`and `q.h`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your `q.c` doesn't compile with the full suite of `gcc` options.
   - F grade if your `q.c` doesn't link to create an executable.
   - F grade if output of function doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests).
   - F grade if `q.h` or `q.c` include a C standard library header file other than `<stdio.h>` and `<stdbool.h>`.
   - The auto grader will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if output of function matches correct output of auto grader.
   - A deduction of one letter grade for each missing documentation block in `q.c` and `q.h`.

Your submission `q.c` must have one file-level documentation block and **three** function-level documentation blocks. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation blocks are missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to E.

Likewise, your submission `q.h` must have one file-level documentation block and **three** function-level documentation blocks. Remember, your clients will not have access to your definitions. Therefore, these function-level documentation blocks are important for your clients to understand the behavior of your statistics library.

Playing with `char`s

A `char` represents a single letter or character. In a computer, these values are represented with one byte (one byte is defined as eight bits). For example, the character `'A'` is stored on disk as `01000001`. The number 65 is also stored this way (`01000001==65`). Because both 65 and `'A'` are stored as `01000001` we have the bizarre result that:

```
'A' == 65
```

Because `'B'` comes just after `'A'` in the ASCII representation we find that `'B'==66`, `'C'==67`, etc. In fact, C is very happy to treat characters as integers and integers as characters: it doesn't care about the difference between them. This design means a programmer can perform arithmetic on characters as if they were numbers. If you try the following:

```
char c = '!' + '!';
printf("%c", c);
```

you will discover that in character arithmetic, `'!' + '!' == 'B'`, because `'!'` is 33 and `'B'` is 66. Similarly `'%'*2` is `'J'`, because of the numerical values of `'%'` and `'J'` are 37 and 74, respectively.

Here are some additional expressions that represent some useful properties of characters:

```
'A' + 3     == 'D'
'f' - 'a'   == 5
'0'         == 48
'\0'        == 0
'7'         != 7
'7' - '0'   == 7
('G' < 'M') == true
true        == 1
false       == 0
```

Notice that natively, C does not have a built-in boolean type, and uses the value 0 to mean false and non-zero values to mean true instead. However, by adding `#include <stdbool.h>`, we get access to the type `bool` and the constants `true` and `false`, which are just synonyms for 1 (or another non-zero value) and 0.

Even though we are suggesting that types are merely suggestions, there are still some differences between types. For instance, a `char` reserves a single byte in memory, while an `int` reserves 4 bytes. The size of a type can be determined using the `sizeof` operator.

ASCII

The way C encodes characters as integers is standard for many programming languages and programs dating back to the 1960s. C uses the ASCII character set, where ASCII stands for "American Standard Code for Information Interchange." The basic ASCII character set is a set of 128 characters, thus requiring only 7 bits, leaving 1 bit unused per byte (but requiring it to be 0). You can view a chart of the basic ASCII character set by typing the following into a terminal:

```
man ascii
```

Press `q` to return to the terminal. In order not to waste the unused bit, there is an Extended ASCII, that defines an additional 128 characters (Complete Extended ASCII Table). There are many other extensions to ASCII, which use the extended 128 characters in different ways to accommodate various, mainly European, languages.

There are plenty of more glyphs in this world, however. The most successful attempt of putting virtually every character set under the sun (Chinese, Arabic, Cyrillic, Devanagari, etc.) in one and the same list, obviously much longer than ASCII, is called *Unicode*. The most popular way to encode Unicode is called UTF-8 which extends ASCII by using the 8th unused bit to indicate that the following byte is a continuation of the first one (this can be done several times if needed, enabling full support of the long Unicode list). UTF-8 is widely adopted and used today on more than half of all web pages on the Internet. In C, we will stick with the basic ASCII.

Take a moment to inspect the ASCII character set. You will notice that the alphabet appears twice, in two big contiguous chunks, once in lowercase form, and once in uppercase.

The lesson is this: characters can be treated as numbers in many useful ways. You can add and subtract from characters to move around the ASCII chart. You can compare characters using greater-than and less-than relations, and because of the way the ASCII character set is laid out, letters will often have the quantitative properties you expect: that is, `'b'` is greater than `'a'` but less than `'c'`. In fact, `'b' + 1 == 'c'`.

Practice task

Print out the basic ASCII table. That is, for each integer from 1 to 127, print both the integer representation (like 65) and the character representation (like `'A'`). You can print the table by representing the same value/variable as both an `int` and a `char` in `printf` (use both the `%d` and `%c` directives in the string but use the same value for each).

Small functions to identify characters

We'll want to identify Latin characters, non-Latin characters, and control characters. In addition you may need to distinguish uppercase from lowercase letters. Write two functions:

```
int is_upper(int ch)
int is_lower(int ch)
```

It should return `1` if the character is an upper case letter (between `'A'` and `'Z'`) or is a lower case letter (between `'a'` and `'z'`); otherwise the function should return `0`. Because statements like `'B'>'A'` are themselves boolean values these functions can be written as simple one-liners. As an example function, `is_latin`, could be written using the above functions as:

```c
int is_latin(int ch) {
  return is_upper(ch) || is_lower(ch);
}
```

Writing these functions is usually a simple and straightforward task. As an example, the function `is_digit` can be defined like this:

```c
int is_digit(int ch) {
  return ('0' <= ch && ch <= '9') ? 1 : 0;
}
```

Making these sorts of functions manually is good practice. However, for future reference, there are plenty of useful functions for working with `char` s that are presented in standard library header file `ctypes.h`.

I/O with files

This tutorial will deal with functions `fopen`, `fclose`, and `fgetc` to handle file I/O. These functions allow you to do operations on specific files. Sometimes you want to open a file in your program to read some data from it, or you want to write some results to a specific file. The system permits a minimum of `FOPEN_MAX` (this macro is defined in `<stdio.h>`) files to be open at once. In Linux/gcc implementations, `FOPEN_MAX` is defined as 16. Three of these are opened by default: standard input stream `stdin`, standard output stream `stdout`, and standard error stream `stderr`. Therefore, on our Linux/gcc systems at least 13additional files can be opened by your program.

Working with files always proceeds the same way: you first open the file, do your operations on the file, and then close the file when you're done with it.

The `FILE` type

As noted, the first step in working with a file is opening it. Since many files can be open at once, the C standard library needs an unique way of identifying open files. Rather than using the file name, a special pointer of type `FILE` is used. In fact, the file name comes into play only once - when the file is opened. Thereafter, the `FILE` pointer is used to identify the particular file that is the target of an I/O operation.

`fopen`

As noted, in order to perform any I/O operation on a file, it first must be opened. `fopen` is most often used to open a file. It takes two arguments: the name of the file to be opened and the mode. Both arguments are of type `char []` or `char *`. The mode specifies the type of operation you want to perform on the file: read from it, write to it, add data to the end of it, or update it (do both reading and writing). The various modes recognized by `fopen` are summarized in the following table:

Access Mode	Allows you to
`r`	Read from the file
`w`	Write to the file; if the file already exists, its previous contents are lost; if file doesn't exist, it is created
`a`	Write to the end of the file; if file doesn't exist, it is created
`r+`	Read and write to the file (like `r`, but data can also be written to the file)
`w+`	Read and write to the file (works like `w`, but data can also be read from the file)
`a+`	Read and write to the file (works like `a` - writes can go only to the end of the file - but reading also permitted anywhere in the file)

All of these modes can optionally take a `b` at the end (as in `"rb"` or `"w+b"`) to indicate that a binary file is to be opened.

If you want to simply read some data from an existing text file, then you open it in `r` mode:

```
fopen("datafile", "r");
```

or

```
fopen("datafile", "rb");
```

if the file contains binary data.

If you want to create a new file to write some data to, you open it in`w` mode, being careful to remember that if the file already exists, you'll lose its contents forever:

```
fopen("results", "w");
```

or

```
fopen("results", "wb");
```

When a file is opened in append mode (`a` or `a+`), it's guaranteed that you won't be able to overwrite existing data in the file; all write operations will simply automatically append data to the end of the file:

```
fopen("logfile", "a");
```

or

```
fopen("logfile", "ab");
```

The three "update" modes - `"r+"` , `"w+"` , `"a"` - should be understood by you before you use them. They all allow both reading from and writing to the same file. With read update, it's assumed that you've an existing data file that you want to read and write. `w+` behaves like `w` except you can also read from the file. It's important to note that if the file already exists, its contents will be erased. So if you've a database that you want to make changes to, the file should be opened `r+` and not `w+`:

```
fopen("database", "r+b");
```

As noted, `a+` guarantees that writes will go to the end of the file; reads can be performed anywhere in the file.

After `fopen` opens the indicated file with the specified mode, it returns a `FILE` pointer that you must use to subsequently identify the file. If the open fails for some reason (e.g., you try open a nonexistent file in `r` or `r+` mode, or you don't have the proper access permissions on the file), then `fopen` returns a `NULL` value. You should always check the return value from `fopen` to make sure it succeeds. Using a `NULL` pointer for a subsequent I/O operation will frequently cause your program to terminate abnormally.

`fclose`

In order to close an open file, you call `fclose`. It takes a `FILE` pointer as the argument, writes any data that may be sitting in the buffer to the file, and then closes the file.

In Linux, all files are automatically closed whenever your program terminates normally (i.e., not due to a memory violation, floating point exception, or program interrupt). So in many cases, it's not necessary for you to close your files yourself. However, if you want your program to be portable and work in Windows or if for some reason you need to work with more than `FOPEN_MAX` files in a program, then you'll have to close files when you're done with them in order to work within the maximum.

`fprintf` and `fscanf`

These two functions are equivalent to their standard input and standard output counterparts, `printf` and `scanf`, except that they take an additional parameter that specifies the file the data is to be written to or read from. For example, if `infile` is a `FILE` pointer for a file that has been opened for reading, then the call

```
fscanf(infile, "%d %d", &month, &year);
```

will read two integers from the file. And if `outfile` points to a file opened for writing, then the call

```
fprintf(outfile, "The answer is %d\n", result);
```

will write the specified line to the file.

The following program creates a file called `names` by opening it in write mode and then writes some data to it:

```
// creating a file
#include <stdio.h>

int main(void) {
  FILE *outfile;
  if ( (outfile = fopen("names", "w")) == NULL ) {
    printf("Can't create file names\n");
    return 0;
```

```
 9    }
10    fprintf(outfile, "Tom\n");
11    fprintf(outfile, "Dick\n");
12    fprintf(outfile, "Harry\n");
13
14    fclose(outfile);
15    return 0;
16  }
```

The `if` on line 7 calls `fopen` to open the file `names` for writing. The resulting `FILE` pointer that is returned is assigned to variable `outfile` and then is tested against `NULL` to see if `fopen` has succeeded. If it failed the following `printf` is executed and the program exits.

The following program shows how to use append mode to add data to the end of the `names` file:

```c
// appending data
#include  <stdio.h>

int main(void) {
  FILE *outfile;
  if ( (outfile = fopen("names", "a")) == NULL ) {
    printf("Can't append to file names\n");
    return 0;
  }
  fprintf(outfile, "Clint\n");
  fprintf(outfile, "Paul\n");
  fclose(outfile);
  return 0;
}
```

Recall that the I/O routines use three predefined streams = `stdin`, `stdout` and `stderr` - that refer to the standard input, standard output, and standard error streams. These predefined streams can be given as arguments to any I/O routine in the C standard library that takes a `FILE` pointer as a parameter. So, for example, the call

```c
fprintf(stderr, "Cannot open the file for reading\n");
```

writes the indicated message to standard error stream. The call

```c
fprintf(stdout, "hello\n");
```

is equivalent to

```c
printf("hello\n");
```

just as the call

```c
fscanf(stdin, "%d", &i);
```

is equivalent to

```c
scanf("%d", &i);
```

`getchar` and `putchar`

`getchar` reads a single character from standard input, while `putchar` writes a single character to standard output. The following program simply copies standard input to standard output a single character at a time.

```c
// copy standard input to standard output
#include <stdio.h>

int main(void) {
  int ch;
  while ((ch = getchar()) != EOF) {
    putchar(ch);
  }
  return 0;
}
```

The characters read by `getchar` are written to standard output with `putchar`. Note that `putchar` can be given an `int` to write, since characters are converted to `int`s anyway when they're passed to functions. Eventually, `getchar` will return `EOF` after the last character has been read, causing the loop to terminate.

```
$ ./a.out
today is a good day
tomorrow will be a better day
EOF
$ ./a.out < test.txt
Here are some
sample lines of text to
see how the various I/O
routines work
$ ./a.out < test.txt > text2
Here are some
sample lines of text to
see how the various I/O
routines work
```

Even though this is such a simple program, it is actually quite powerful. The first time the program is executed, two lines are typed at the terminal. The program reads the lines one character at a time and then prints them back out. Remember that because I/O is buffered, the line that is typed is not made available to the program until the *ENTER* key is pressed, even though you're reading in single characters. Typing `CTRL-z` under Linux or `CTRL-d` under Windows sends an end-of-file condition to the program, causing it to terminate.

The second time the program is run, standard input is redirected from `test.txt`. When `getchar` is called to read a character from standard input, it will actually be reading characters from the file `test.txt`. `putchar` still writes to standard output, so the net result is that this form of execution of the program allows you to view the contents of a file as in the `cat` command in Linux or `type` command in Windows.

The last time the program is run, standard input is redirected from `test.txt` and standard output is redirected to `text2`. This form of execution allows you to copy one file to another as in the `cp` command in Linux or `copy` command in Windows.

Why does `getchar` return an `int`?

Notice that `ch` is declared as an `int` and not a `char`, even though you're using `getchar` to read characters. The reason for this is as follows: `getchar` is defined to return all possible character values, not just those in the normal character set. On most systems, this means that `getchar` can read and return any possible 8-bit value. In order to signal to the programmer that no more characters are left to be read from standard input, `getchar` returns the special defined value `EOF` (usually defined as -1 in `<stdio.h>`). Since this return value has to be distinguishable from any valid character that `getchar` can otherwise return, `getchar` is therefore defined to return an `int`. If `ch` is wrongly defined to be a `char`, then on some systems, the program shown will work and on others it won't. It all depends on whether or not the system defines a `char` as a `signed char` or `unsigned char`. Note that the standard leaves it to a particular system to specify whether a `char` is `signed char` or `unsigned char`. Suppose the system declares `char` type as `signed char`. Then the `int` value of -1 that is returned on end of file by `gechar` will be truncated from 32-bits to 8-bits with a resultant value of -1. When this value in `ch` is then compared to the defined value `EOF`, the 8-bit signed integer value of -1 will be sign extended to 32-bit `int` value of -1. This value will be compared against `EOF` value of -1 and the `while` loop will succeed. On the other hand, if the system declares `char` type as `unsigned char`, the `int` value of -1 that is returned on end of file by `getchar` will be truncated from 32-bits to $8-$bits with a resultant unsigned value of 255. When this unsigned value in `ch` is then compared to the defined value `EOF` (which is `int` value of -1), the8 -bit unsigned integer value of 255 will be extended to $32-$bit `int` value of 255. 255 will be compared against `EOF` value of -1 and the `while` loop will theoretically execute forever. Although declaring `ch` as `signed char` will prevent the infinite `while` loop, it still wouldn't allow you to use `getchar` to read in all possible 8-bit values since the value 255 would still be indistinguishable from the value `EOF`.

`fgetc` and `fputc`

These functions perform character I/O on specified files. `fgetc` works like `getchar` and reads single characters, returning the integer `EOF` on end of file. `fputc` works like `putchar` and writes single characters.

Suppose you've the following stored inside a file called `inspire.txt`:

```
today is a good day
tomorrow will be a better day

```

and you want to write the contents of this file to standard output. The following program does just that.

```c
#include <stdio.h>
int main(void) {
  FILE *infile;
  if ((infile = fopen("inspire.txt", "r")) == NULL) {
    fprintf(stderr, "Can't open inspire.txt!\n");
  return 0;
  }
  int ch;
  while ((ch = fgetc(infile)) != EOF) { fputc(ch, stdout); }
  return 0;
}
```

Executing the program

```
$ ./a.out
```

writes the following text to standard output:

```
today is a good day
tomorrow will be a better day

```

The program calls `fopen` to open the file `inspire.txt` for reading. The returned `FILE` pointer is assigned to `infile` and then is tested against `NULL` to see if the `fopen` succeeded. If it fails, then `fprintf` is called to write an error message to standard error and the program exits. If the `fopen` succeeds, then a `while` loop is entered to read the characters from the file. `fgetc` reads a character from the file specified by its argument. The character that is read is stored into the integer variable `ch`, and then tested against `EOF`. If a character was read, then `fputc` is called to write the character to standard output. After the last character has been read from `inspire.txt`, `fgetc` returns `EOF` and the `while` loop terminates.

The following program copies the contents of the file `names` to `names2`.

```c
// file copy ...
#include <stdio.h>
int main(void) {
  FILE *infile, *outfile;
  if ((infile = fopen("names", "r")) == NULL) {
    fprintf(stderr, "Can't read names\n");
  return 0;
  }
  if ((outfile = fopen("names2", "w")) == NULL) {
    fprintf(stderr, "Can't write names2\n");
  return 0;
  }
  int ch;
  while ((ch = fgetc(infile)) != EOF) { fputc(ch, outfile); }
  return 0;
}
```

The input file `names` is opened for reading and the `FILE` pointer returned by `fopen` is assigned to the `FILE` pointer `infile`. If the `fopen` fails, a message is logged and the program exits. The output file `names2` is then opened for writing and the `FILE` pointer returned by `fopen` is assigned to the pointer variable `outfile`. As before, if the `fopen` fails, a message is displayed and the program exits. Remember that if `names2` already exists and has some data in it, then that data will be lost when the file is opened in write mode. If both files are successfully opened, then the input file is copied to the output file one character at a time by corresponding calls to `fgetc` and `fputc`.

Rather than hard coding the two files `names` and `names2` into the program, a more flexible approach would be to allow the file names to be typed on the command line. The following program does just that.

```c
// copy files specified on command line
#include <stdio.h>
```

```c
int main(int argc, char *argv[]) {
  FILE *infile, *outfile;
  if (argc != 3) {
    fprintf(stderr, "Bad argument count\n");
    return 0;
  }
  if ((infile = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "Can't open file %s\n", argv[1]);
    return 0;
  }
  if ((outfile = fopen(argv[2], "w")) == NULL) {
    fprintf(stderr, "Can't create file %s\n", argv[2]);
    return 0;
  }
  int ch;
  while ((ch = fgetc(infile)) != EOF) { fputc(ch, outfile); }
  return 0;
}
```

The name of the file to be copied is passed to the program through `argv[1]` and the name of the output file through `argv[1]`. The two files are then opened as before and the contents copied.

Assertions in C

An *assertion* specifies that a program satisfies certain conditions at particular points in its execution. There are three types of assertion:

- *Preconditions* that specify conditions at the start of a function.
- *Postconditions* that specify conditions at the end of a function.
- *Invariants* that specify conditions over a defined region of a program.

An assertion violation indicates a bug in the program. Thus, assertions are an effective means of improving the reliability of programs. In other words, they are a systematic debugging tool.

In C, assertions are implemented with the standard C library `assert` macro that is defined in header file `<assert.h>`. The argument to `assert` must be true when the macro is executed, otherwise the program aborts and prints an error message. For example, the assertion

```c
assert(size <= LIMIT);
```

will abort the program and print an error message like this

```
Assertion violation: file test.c, line 20: size <= LIMIT
```

if the value stored in variable `size` is greater than macro `LIMIT`.

Preconditions

Preconditions specify the input conditions to a function. They can be best explained using the following analogy. Suppose you (the function) are employed as an apple-packer. One of the conditions of your contract is that the temperature in the warehouse will be no greater than 30°C. If the temperature exceeds 30°C, you are not obliged to do anything: you can keep packing apples if you want to, or you can choose to go to the beach. Your employer (the caller), however, knows that you are not required to pack apples if the temperature exceeds 30°C, so he or she makes sure that the air-conditioning in the warehouse is operating correctly.

Here is an example of a function with preconditions:

```c
int magic(int size, double *data) {
  assert(size <= LIMIT);
  assert(data != NULL);
  // other code here ...
}
```

These pre-conditions have two consequences:

1. Function `magic` is only required to perform its task if the pre- conditions are satisfied. Thus, as the author of this function, you are not required to make `magic` do anything sensible if `size` or `data` are not as stated in the assertions.
2. The caller is certain of the conditions under which function `magic` will perform its task correctly. Thus, if your code is calling function `magic`, you must ensure that the `size` or `data` arguments to the call are as specified by the assertions.

Postconditions

Postconditions specify the output conditions of a function. They are used much less frequently than preconditions, partly because implementing them in C can be a little awkward. Here is an example of a postcondition in function `magic`:

```c
int magic(int size, double *data) {
  // other code here ...
  assert( result <= LIMIT );
  return result;
}
```

The postcondition also has two consequences:

1. Function `magic` guarantees that the stated condition will hold when it completes execution. As the author of function `magic`, you must make certain that your code never produces a value of `result` that is greater than `LIMIT`.
2. The caller is certain of the task that function `magic` will perform (provided its preconditions are satisfied). If your program is calling function `magic`, then you know that the result returned by `magic` can be no greater than `LIMIT`.

Compare these consequences with the apple-picker analogy. Another part of your contract states that you will not bruise the apples. It is therefore your responsibility to ensure that you do not (and if you do, you have failed.) Your employer is thus relieved of the need to check that the apples are not bruised before shipping them.

Recommended practices

Writing preconditions

The simplest and most effective use of assertions is as preconditions - that is, to specify and check input conditions to functions. Two very common uses are to assert that:

1. Pointers are not `NULL`.
2. Indexes and size values are non-negative and less than a known limit.

Each assertion must be listed in the *Asserts* section of the function description comment in the corresponding header file. For example, the comment describing function `magic` will include:

```
/*
*  Asserts:
*      'size' is no greater then LIMIT.
*      'data' is not NULL.
*      The function result is no greater than LIMIT.
*/
```

If there are no assertions, say that by writing *Nothing*:

```
/*
*  Asserts:
*      Nothing
*/
```

Satisfying preconditions

When your code calls a function with preconditions, you must ensure that the function's preconditions are satisfied. This does not mean that you have to include code to check the argument to very function that you call! For example, in the following code, function `resize` does not need to check that the argument to function `measure` is `NULL`, since its own assertion ensures this:

```c
void resize(int *value) {
  assert(value != NULL);
  // some code here ...
  measure(value, 0);
  // other code here ...
}
```

In other words, you need to decide for yourself when and where values must explicitly be checked to avoid violating preconditions.

Assertion violations

If a precondition is violated during program testing and debugging, then there is a bug in the code that called the function containing the precondition. The bug must be found and fixed.

If a postcondition is violated during program testing and debugging, then there is a bug in the function containing the precondition. The bug must be found and fixed.

Assertions and error-checking

It is important to distinguish between program errors and run- time errors:

1. A program error is a bug, and should never occur.
2. A run-time error can validly occur at any time during program execution.

Assertions are not a mechanism for handling run-time errors. For example, an assertion violation caused by the user inadvertently entering a negative number when a positive number is expected is poor program design. Cases like this must be handled by appropriate error-checking and recovery code (such as requesting another input), not by assertions.

Realistically, of course, programs of any reasonable size do have bugs, which appear at run-time. Exactly what conditions are to be checked by assertions and what by run-time error- checking code is a design issue. Assertions are very effective in reusable libraries, for example, since

- the library is small enough for it to be possible to guarantee bug-free operation, and
- the library routines cannot perform error-handling because they do not know in what environment they will be used.

At higher levels of a program, where operation is more complex, run-time error-checking must be designed into the code.

Turning assertions off

By default, C compilers generate code to check assertions at run-time. Assertion-checking can be turned off by defining the `NDEBUG` flag to your compiler, either by adding the following directive to a source file

```
#define NDEBUG
```

or by calling your compiler with the `-D` option:

```
gcc -D NDEBUG ...
```

This should be done only you are confident that your program is operating correctly, and only if program run-time is a pressing concern.