Contents

1 Generals

1.1 Some Facts

Below are some general facts about C language.

- assignments associate from right to left.
- arithemetic operators associate left to right
- relational operators have lower precedence than arithmetic operators
- expressions connected by `&&` or `||` are evaluated left to right. `&&` has a higher precedence than `||`, both are lower than relational and equality operators. (higher than assignment operators?)
- printable characters are always positive
- The standard headers `<limits.h>` and `<float.h>` contain symbolic constants for all of the sizes of basic data types, along with other properties of the machine and compiler.
- a leading 0 (zero) on an integer constant means octal
- a leading `0x` or `0X` (zero x) means hexadecimal.
- you can use escape sequence to represent number. Check it at p51. The complete set of escape sequences are in p52.
- `strlen()` function and other string functions are declared in the standard header `<string.h>`.
- external and static variables are initialized to zero by default.
- for portability, specify `signed` or `unsigned` if non-character data is to be stored in `char` variables. (p58)
- to perform a type conversion:

  ```
  double a = 2.5;
  printf("%d", (int) a);
  ```

result will be `2`.

- unary operator associate right to left (like `*`, `++`, `--`)
- strcpy() function in C needs a pointer to a character array! A pointer to character will cause segmentation fault

Place holder.

1.2 headers

- `<stdio.h>`: contains input/output functions
- `<ctype.h>`: some functions regarding to characters

1.3 `printf()` formatting

Check p26-p27 of the textbook.

Use `%` with symbols to print the variables in different format. Example:

```
printf("%c", a)  //print a in format of character
printf("%s", a)  //print a in format of character string
printf("%nc", a)  //print a in format of character, using a character width
↪  of size n (at least)
printf("%f", a)  //print a in format of float
printf("%nf", a)  //print a in format of float, using a width of size n
printf("%n.0f", a)  //print a in format of float, using a character width of
↪  size n, with no decimal point and no fraction digits
printf("%n.mf", a)  //print a in format of float, using a character width of
↪  size n, with decimal point and m fraction digits
printf("%0.mf", a)  //print a in format of float, with decimal point and m
↪  fraction digits. The width is not constrained.
printf("%d", a)  //print a in format of integer
printf("%o", a)  //print a in format of octal integer
```

```
printf("%x", a)  //print a in format of hexadecimal integer
```

1.4 Character Input and Output

- `getchar()`: it reads the next input character from a text stream and returns that (from the buffer?).
- `putchar()`: it prints a character each time it is called and passed a char into.

Pay attention that a character written between single quotes represents an integer value equal to the numerical value of the character in the machine's character set. This is called a character constant. For example, `'a'` is actually `97`.

1.5 Arrays

The syntax is similar with C++. For example, to define an array of integers with a size of 100, you do:

```
int nums[100];
```

Remember to initialize each slot:

```
for (int i = 0; i < 100; i++)
  nums[i] = 0;
```

You can also to use assignment operator and `{ }` to initialize the array when defining. For example, the following C-string is initialized when being defined:

```
int main() {
  char s[] = {'a', 'b', 'c' };
  printf("%s", s);
  return 0;
}
```

1.6 Enumeration constant

An enumeration is a list of constant integer values. For example:

```c
enum boolean { NO, YES };
```

The first name in an `enum` has value 0, the next 1, and so on, unless explicit values are specified:

```c
enum boolean { YES = 1, NO = 0 };
```

If not all values are specified, unspecified values continue the progression from the last specified value:

```c
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
↪ };
// FEB is 2, MAR is 3, etc.
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration. Enumeration works like using `#define` to associate constant values with names:

```c
#define JAN 1
#define FEB 2
// etc
```

1.7 type-cast an expression

Explicit type conversions can be forced ("coerced") in any expression. For example:

```c
int main() {
  int n = 2;
  printf("%f", (float) n);
  return 0;
}
```

In the above example, when being printed, the type of `n` has been modified to `float`. Notice that `n` itself is not altered. This is called a *cast*, it is an unary operator, has the same high precedence as other unary operators.

1.8 Bitwise operators

p62

There are 6 bitwise operators for bit manipulation. They may be applied to integral operands only.

They are:

- `&` : bitwise AND
- `|` : bitwise inclusive OR
- `^` : bitwise exclusive OR
- `<<` : left shift
- `>>` : right shift
- `~` : one's complement (unary)

The precedence of the bitwise operators `&`, `^` and `|` is lower than `==` and `!=`.

1.9 Operators can be used with assignment operators

p64

`+, -, *, /, %, <<, >>, &, ^, |`

1.10 Symbolic Constants

A `#define` line defines a symbolic name or symbolic constants to be a particular string of characters. You use it like: `#define` *name replacement text*. You put this at the head of your code (outside scope of any function to make it globally). Example:

```
#include <stdio.h>

#define LOWER 0
#define UPPER 300
#define STEP 20
```

```c
int main() {

  for (int i = LOWER; i <= UPPER; i += STEP) {
    printf("%5d\t%20f", i, 5 * (i - 32) / 9.0);
    printf("\n");
  }

  return 0;
}
```

Pay attention that symbolic name or symbolic constants are not variables. They are conventionally written in upper case. No semicolon at the end of a `#define` line.

2 Functions and Program Structure

2.1 External Variables

If an external variables is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an `extern` declaration is mandatory. For example, a function using external variables in a different source file can declare these variables in following manner:

```c
int addNum(int a) {
  extern int ADDAMOUNT;  // variable ADDAMOUNT is in different source file

  return a + ADDAMOUNT;
}
```

Array sizes must be specified with the definition, but are optional with an `extern` declaration.

2.2 The C Preprocessor

2.2.1 File Inclusion

2.2.2 Macro Substitution

2.2.2.1 General A definition of a macro Substitution has the form:

```
#define name replacement_text
```

After this line, subsequent occurrences of the token `name` will be replaced by the `replacement_text`. `name` has the same form as a variable name (so no white space is allowed between characters), `replacement_text` is arbiturary.

`replacement_text` is the rest of the line. If you need long definition, you can place a `'\'` at the end of each line to be continued. For example:

```
#include <stdio.h>
#define say_hi_5_times  (for (int i = 0; i < 5; i++) \
^^I^^I^^I  printf("Hi\n");)

int main() {
  say_hi_5_times
  return 0;
}
```

This program will print `"Hi"` 5 times. Notice that there is no `;` after the macro, this is because `say_hi_5_times` calls for a macro substitution, every occurrence of `say_hi_5_times` will be replaced by:

```
for (int i = 0; i < 5; i++)
  printf("Hi\n");
```

Thus, after preprocessing, the code is actually:

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

```c
int main() {
  for (int i = 0; i < 5; i++)
    printf("Hi\n");
  for (int i = 0; i < 5; i++)
    printf("Hi\n");
  for (int i = 0; i < 5; i++)
    printf("Hi\n");
  for (int i = 0; i < 5; i++)
    printf("Hi\n");
  for (int i = 0; i < 5; i++)
    printf("Hi\n");

  return 0;
}
```

Notice: you can use `'\'` inside the parenthese.

2.2.2.2 Arguments It is possible to define macros with arguments, so the replacement text can be different for different calls of the macro. For example:

```c
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Do not treat it as function call, there is nothing relating to function happening here. It is still a macro substitution. Each occurrence of a formal parameter (here `A` or `B`) will be replaced by the corresponding actual argument. For example, if `max(p+q, r+s)` appeared, after preprocessing, this line will become:

```c
((p+q) > (r+s) ? (p+q) : (r+s))
```

You can even put function names in the parameter:

```c
#include <stdio.h>
#define plus(A, B) (A)() + (B)()
```

```c
int one() {
  return 1;
}

int two() {
  return 2;
}

int main() {
  printf("result: %d", plus(one, two));
  return 0;
}
```

The output is: `result: 3`

2.2.2.3 Pitfalls There are pitfalls hidden in the macro substitution. For example, in the `max()` macro:

```c
#define max(A, B) ((A) > (B) ? (A) : (B))
```

each expression is evaluated twice. Thus usages like `max(i++, j++)` will increment the larger value twice (first in the comparing part, next in the "returning" part). User of `max(i++, j++)` may expect single increment.

An other example of pitfall is

```c
#define square(x) x * x
```

Notice there is no parenthese, so if we have an expression like `square(a + 1)`, after macro substitution, the actual expression is:

```c
a + 1 * a + 1
```

which is not `(a + 1) * (a + 1)`. So, make sure to use parenthese to enclose your parameters to avoid such mistake. In the above example, `square(x)` should be:

```
#define square(x) ((x) * (x))
```

2.2.3 Conditional Inclusion

3 Pointers and Arrays

3.1 Command-line Arguments

p128 in CPL.

We can pass command-line arguments or parameters to a program when it begins executing. An example is the echo program. On the command prompt, you enter `ehco`, followed by a series of arguments:

```
$ echo hello world
```

then press enter. The command line window will repeat the inputed arguments:

```
$ echo hello world
$ hello world
```

The two strings `"hello"` and `"world"` are two arguments passed in echo program.

Basically, when `main()` is called, it is called with two arguments: `argc` and `argv`.

- `argc`: stands for argument count. It is the number of command-line arguments when the program was invoked (i.e. how many strings are there in the line that invoked the program). In the above echo example, `argc == 3`, the three strings are: "echo", "hello" and "world", respectively.
- `argv`: stands for argument vector. It is a pointer to an array of character strings that contain the actual arguments, one per string. You can imagine when you type in command line to invoke a program, what you typed in was stored somewhere in an array of character strings. Additionally, the standard requires that `argv[argc]` be a null pointer. In the echo example, you typed "echo hello world", and following array of characters was stored:

  ```
  ["echo", "hello", "world", 0]
  ```

3.1.1 Example: `echo`

Knowing this, we can write a program that mimic the `echo` function: re-print what we typed in when we invoke the program to terminal:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  while (*(++argv))
    printf("%s%s", *argv, *(argv + 1) ? " " : "");  // the second %s is for
     ↪  the space

  printf("\n");

  return 0;
}
```

3.1.2 Example: `pattern_finding`

This program will try to find any lines in the input buffer that contains the keyword passed in when invoking it. For example, in command line prompt:

```
$ pattern_finding love < text.txt
```

it will print all lines that contain `love` to the terminal.

The program uses `strstr()` to search the existence of a certain keyword in target string. We also write a `getline()` function to get one single line from input buffer (using `getchar()`). Pay attention that in the new C library (`stdio.h`), a `getline()` function has been added. So we rename our function to `getlines()`. The code is as follows:

```c
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000
```

```
int getlines(char* line, int max);

//find: print lines that match pattern from 1st arg
int main(int argc, char* argv[]) {
  char line[MAXLINE];  // used to hold a line of string
  int found = 0;

  if (argc != 2)
    printf("Usage: find pattern\n");
  else
    while (getlines(line, MAXLINE) > 0)
      if (strstr(line, argv[1]) != NULL) {
^^Iprintf("No.%d: %s", ++found, line);
      }

  return found;
}

int getlines(char* line, int max) {
  char ch;

  while (--max > 0 && (ch = getchar()) != EOF && ch != '\n') {
    *(line++) = ch;
  }

  if (ch == '\n')
    *(line++) = ch;  // no need to worry about not enough space, since if ch
     ↪  == '\n', it is not stored in line yet, because the loop was not
     ↪  executed
  *line = '\0';
```

```
  if (ch == EOF)
    return -1;

  return 1;
}
```

3.1.3 Optional arguments example: `pattern_finding` extended

Now we extend our `pattern_finding` program so it can accept optional arguments. A convention for C programs on UNIX systems is that an argument that begins with a minus sign introduces an optical flag or parameter. Optional arguments should be permitted in any order, they can also be combined (a minus sign with two or more optional arguments, without space between each other).

There is no magic about optional arguments. They are collected as strings in `argv[]` when the program is invoked, just like anyother strings occured when invoking the function. We extend the `pattern_finding` program to include support for two optional arguments:

1. -x: print lines that doesn't contain the target pattern;
2. -n: in addition to print lines, the program will also print the corresponding line number before the line.

So, the program can be invoked in following way:

```
$ pattern_finding -n -x keyword < text.txt
```

in this case, when `main()` is called, `argc == 4`, `*argv == {"pattern_finding", "-n", "-x", "keyword"}`. `< text.txt` is just redirect `stdin` to the text.

Or, we can combine the two optional arguments:

```
$ pattern_finding -xn keyword < text.txt
```

in this case, when `main()` is called, `argc == 3`, `*argv == {"pattern_finding", "-xn", "keyword"}`.

Thus, we have to write code to analyze argument strings that has `"-xxx"` form. Generally, we keep a list of flags inside the program. If we encountered any optional argument in the string, we can set the corresponding flag to true.

The code and explanation is as follows:

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getlines(char* line, int max);

//find: print lines that match pattern from 1st arg
// with optional arguments enabled
int main(int argc, char* argv[]) {
  char line[MAXLINE];  // temporary container to hold line read from buffer
  char c;  // to check optional arguments

  int line_num = 0;  // record the number of line
  int except = 0;  // flag of optional argument x, if this is true, print
  ↪  lines that doesn't have pattern
  int number = 0;  // flag for optional argument n , if this is true, print
  ↪  the corresponding line number
  int found = 0;


  // check inputted arguments and set flag accordingly
  // use prefix to skip the first argv (which is the name of the function)
  while (--argc > 0 && (*++argv)[0] == '-')  // outter while loop check each
  ↪  "-xxx" styled optional argument
    while (c = *++argv[0]) {  // inner while loop check each char in the
    ↪  "-xxx" styled argument
```

```
      switch (c) {
      case 'x':
^^Iexcept = 1;
^^Ibreak;
      case 'n':
^^Inumber = 1;
^^Ibreak;
      default:
^^Iprintf("find: illegal option %c\n", c);
^^Iargc = 0;  // this will terminate the program
^^Ifound = -1;
^^Ibreak;
      }
    }

  if (argc != 1)  //we should have only one argument at this point, which is
   ↪  the pattern we are going to find. All optional arguments have been
   ↪  examed by the previous while loop
    printf("Usage: find -x -n pattern\n");  // print a message showing how to
     ↪  use this program
  else
    while (getlines(line, MAXLINE) > 0) {
      line_num++;  // update the line number

      /*Notes:
^^IPrint the line based on value of variable except and the found result.
^^ITo print a line, the truth value of found and except should be different.
 ↪  When except = 1, we print lines that not found, so found == 0;
^^IWhen except = 0, we print lines that are found, so found == 1;
      */
```

```
      if ((strstr(line, *argv) != NULL) != except) {
^^Iif (number)  // if the number flag is true, we print the line number
^^I  printf("%d", line_num);
^^Iprintf("%s", line);
^^Ifound++;
      }

    }

  return found;

}

int getlines(char* line, int max) {
  char ch;

  while (--max > 0 && (ch = getchar()) != EOF && ch != '\n') {
    *(line++) = ch;
  }

  if (ch == '\n')
    *(line++) = ch;  // no need to worry about not enough space, since if ch
     ↪  == '\n', it is not stored in line yet, because the loop was not
     ↪  executed
  *line = '\0';

  if (ch == EOF)
    return -1;

  return 1;
```

```
}
```

3.2 Pointers to Functions

It is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on.

To declare a pointer to a function, you write:

```
return_type (*ptr_name)(parameter1_type, parameter2_type, ...)
```

Explanation:

- `return_type`: the return type of the function this pointer pointing to.
- `ptr_name`: the name of the pointer variable
- `parameter_type`: the type of the function this pointer referring to.

Example:

```c
#include <stdio.h>

int add(int a, int b) {
  return a + b;
}

int main() {
  int (*a)(int, int);
  a = &add;
  printf("%d\n", (*a)(2, 3));
}
```

When calling the function pointer, you have to use parenthese to enclose `*` and pointer name. Use `&` and function name to get the “address” of the function.

3.2.1 Example: qsort() which takes a comp() function pointer

(Example 5-11).

A quick sort function which takes a function pointer to be used in its body to sort is as follows:

```
void qsorts(void* v[], int left, int right, int (*comp)(void*, void*)) {
  int last;

  if (left >= right)
    return;

  swap(v, left, (left + right) / 2);
  last = left;

  for (int i = left + 1; i <= right; i++)
    if ((*comp)(v[i], v[left]) < 0)
      swap(v, i, ++last);

  swap(v, left, last);
  qsorts(v, left, last - 1, comp);
  qsorts(v, last + 1, right, comp);
}
```

4 Input and Output

4.1 Standard Input and Output

4.1.1 Input redirection

In many environments, a file may be substituted for the keyboard as the source of standard input by using the < convention for input redirection. For example, we have following code:

```c
#include <stdio.h>

int main() {
  char c;
  while ((c = getchar()) != EOF)
    printf("%c", c);

  return 0;
}
```

When we call the program, we use < to redirect standard input with a file:

```
$ ./a.out < out.txt
```

the effect of this program is to print all content in `out.txt` to standard output.

4.1.2 Output redirection

We can also redirect a program's standard output to a file. We use > convention to do it, the syntax is:

```
$./a.out > result.txt
```

in this way, all standard output of `a.out` will be redirected to file `result.txt`. The file will be created if not exist.

Output produced by `putchar()` and `printf()` are the same, they will both finds its way to the standard output.

4.1.3 Pipe between two programs

It is possible to use one program's standard output as another program's standard input:

```
$./prog1 | ./prog2
```

the above line puts the standard output of `prog1` into the standard input of `prog2`.

4.1.4 Include header file

When you include a file with brackets <>, the compiler will search the header in a standard set of places (typically: `/usr/include`).

4.1.5 Macros in standard library

"Functions" like `getchar` and `putchar` in `<stdio.h>`, and `tolower` in `<ctype.h>` are often macros, thus avoiding the overhead of a function call per character.

4.1.6 Formatted output: printf

p167 on textbook. A table of `printf()`'s conversion characters are shown in table 7-1 in the book (p168).

A width or precision may be specified as `.*`, the value is computed by converting the next argument (which must be an `int`). For example:

```
int main(int argc, char* argv[]) {
  char* s = "abcdefg";
  int length = 4;
  printf("%.*s\n", length, s);
  return 0;
}
```

the above program printed the first `length` characters in string `s`. Don't forget the dot before `*`.

4.1.7 Function `sprintf()`

This function does the same conversions as `printf()`. It accepts a `char* string` argument, and will place the result in `string` instead of to the standard output. `string` must big enough to receive the result.

4.2 Variable-length Argument Lists

This section will use an implementation of a minimal version of `printf()` to show how to write a function that processes a Variable-length argument list in a portable way.

4.2.1 Declare a function that takes varying amounts of arguments

To declare a function whose argument number is not fixed (which may vary), we do:

```c
void miniPrintf(char* format, ...)
```

the declaration `...` means that the number and types of these arguments may vary. It can only appear at the end of a list of named argument (there must be at least one named argument).

4.2.2 Traverse the argument list and final cleanup

The standard header `<stdarg.h>` contains a set of macro definitions that define how to step through an argument list. To build functions that takes varying amounts of arguments, you have to include `<stdarg.h>`.

4.2.2.1 Type `va_list` A data type named `va_list` is defined in `<stdarg.h>`. We declare a variable of this type, then use this variable to refer to each unnamed argument passed in the function. It works like a pointer. For example, we can have following declaration:

```c
#include <stdarg.h>
void miniPrintf(char* format, ...) {
  va_list ap;  // points to each unnamed argument in turn
  va_start(ap, format);  // make ap point to 1st unnamed argument
  //...
}
```

4.2.2.2 Macro `va_start` After the declaration `va_list ap;`, `ap` is an object of type `va_list`. How to use it to actually point to the unnamed arguments? We begin by using a macro named `va_start`. After declaring `ap`, we call this macro to "initiate" `ap`:

```
#include <stdarg.h>
void miniPrintf(char* format, ...) {
  va_list ap;  // points to each unnamed argument in turn
  va_start(ap, format);  // make ap point to 1st unnamed argument
  //...
}
```

va_start() "accepts" two tokens. The first one is the va_list type variable which will be used to refer to unnamed arguments in turn, here we use ap. The second one should be the **LAST** named argument from the function call. va_start will use this to locate the beginning of unnamed argument. After this line, ap will be referring to the first unnamed argument.

But how could we "retrieve" the unnamed argument being referred by ap and move to next argument? We call va_arg macro to do this job.

4.2.2.3 Macro va_arg va_arg is a macro defined in <stdarg.h>. It "accepts" two tokens, the first one is an object of va_list type (we used ap), the second one is the type name you wish to collect from current argument which ap is appointing to. When this macro is called, it returns one argument of the type you specified and steps ap to the next. The type name you provided will be used by va_arg to determine what type to return and how big a step to take. You have to use another variable of the same type to hold the returned argument, so you can use later.

For example, following call of va_arg will return an integer argument, and we hold it using an integer variable named ival:

```
int ival;
ival = va_arg(ap, int);
```

4.2.2.4 Macro va_end va_end is a macro defined in <stdarg.h>. It takes one token, which is the va_list object we used in the program. This macro will do whatever needs to cleanup. It must be called before the function returns:

```
va_end(ap);
```

4.2.3 Example: `miniPrintf()`

In this example, `miniPrintf()` takes two arguments, the first one is a pointer to char, which will be the format string or content it will be printing. Every character of `%` indicates there is an argument in the argument list waiting to be printed in a certain format. Here, we just use the next character after `%` to determine what type of argument we retrieve from the argument list. The function is declared as:

```c
#include <stdarg.h>
#include <stdio.h>
void miniPrintf(char* format, ...)
```

To retrieve arguments in the unamed argument list, we declare an object of type `va_list`:

```c
va_list ap;
char *p;  // to traverse format string
char* sval;  // to hold string argument
int ival;  // to hold integer argument
double dval;  // to hold double argument
```

Before processing, we need to initialize the `va_list` object:

```c
va_start(ap, format);
```

Then, we go over the `format` string. If no `%` encountered, we call `putchar()` to print it directly:

```c
for (p = format; *p; p++) {
  if (*p != '%') {
    putchar(*p);
    continue;
  }

  // do things when '%' is found
}
```

When `%` is found, we need to check the next character and determine what data type we need to retrieve from the unamed argument list:

```
for (p = format; *p; p++) {
  if (*p != '%') {
    putchar(*p);
    continue;
  }

  switch (*++p) {  // check next char
  case 'd':
    ival = va_arg(ap, int);
    printf("%d", ival);
    break;
  }
  case 'f':
    dval = va_arg(ap, double);
    printf("%f", dval);
    break;
  case 's':
    for (sval = va_arg(ap, char*); *sval; sval++)
      putchar(*sval);
    break;
  default:
    putchar(*p);
    break;
}
```

When the style token after `%` is `s`, it means we have to print a string. So the return type of `va_arg` is a pointer to `char`. We print the C-string one character by one character, until we reach the `'\0'` terminator.

4.3 Formated Input: `scanf()`

p171.

4.3.1 A simple example

An example of using `scanf()`:

```
#include <stdio.h>

int main() {
  int a;
  int b;
  int c;
  int d;
  int num;
  scanf("%d%d%d%d", &a, &b, &c, &d);
  printf("a = %d\nb = %d\nc = %d\nd = %d\n", a, b, c, d);

  return 0;
}
```

here, we read four inputs and store them to four variables. Notice we have to pass in the address of each variable to `scanf()`. In this way, `scanf()` can modify the variable directly (passed by value).

4.3.2 Declaration and arguments

`scanf()` is declared as:

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

It will use the `format` string to retrieve information via certain format, convert them and assign to variables in the followed list. `scanf()` stops when it exhausts its format string, or when some input fails to match the control specification. It returns the number of successfully matched and assigned input items (to variable in the unamed argument lists).

The `format` string may contain:

1. blanks or tabs. These will be automatically ignored
2. ordinary characters (not `%`). `scanf()` will try to match these characters with the corresponding non-whitespace character of the input stream. For example:

   ```
   scanf("%dabcde%d", &a, &b);
   printf("a = %d\nb = %d\n", a, b);
   ```

   input: `1abcde2`, output:

   ```
   a = 1
   b = 2
   ```

3. conversion specifications, which is explained below.

A conversion specification is some characters starting with `%`, which will be used by `scanf()` to convert the next **input field** and assign to corresponding variable. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width has been reached (the width of the field may be specified by conversion specification, see below).

In the conversion specification, we may find:

- %: indicating starting of a conversion specification
- *: assignment suppresion marker. If this is present, the input field is skipped, no assignment to variable is made
- number: a number that specifies the maximum width of the input field (of which this current conversion specification is taking care)
- `h, l or L`: indicating the width of the target. `%h`: a short integer; `%l`: a long integer.
- a conversion character: indicating what type to convert to, like `%d, %c, %s` etc. (i.e. the interpretation of the input field).

Some examples of using `scanf()` can be found on p172, 173.

4.4 File Access

4.4.1 Opening a file

The `<stdio.h>` library has a type `FILE` and a function `fopen()` that provides tools to work on files. The function `fopen()`'s declaration is as follows:

```
FILE *fopen(char* name, char* mode)
```

It accepts the name of the file and mode for opening this file. It will return a pointer to a `FILE` object. The type `FILE` is defined with a `typedef`, and is a structure that contains information about the file, such as:

- a pointer to a buffer
  - a buffer is used so file can be read in large chunks
- a count of the number of characters left in the buffer
- a pointer to the next character position in the buffer
- the file descriptor
- flags describing:
  - file openning mode: read or write
  - error states: if error has occurred
  - EOF states: whether end of file has occurred

To obtain a pointer to a file, we do:

```
FILE* fp;
fp = fopen(name, mode);
```

the allowable modes include:

- r: read mode
- w: write mode

- a: append mode
- b: append b to open in binary mode (for some systems)

When errors occurred during file opening, `fopen()` will return a `NULL`.

4.4.2 Accessing the file

Once the file is opened, we access it through the `FILE` pointer `fp`. We have following choices:

- `char getc(FILE *fp)`: (maybe) a macro that accepts a `FILE` pointer, returns the next character from the file (character position is recorded inside the `FILE` object). It returns `EOF` for end of file or error.
- `char putc(char c, FILE *fp)`: (maybe) a macro that accepts a character `c` and a `FILE` pointer. It will write `c` to the file and returns the character written, or returns `EOF` if an error occurs.

After using the file, we have to call `fclose()` to disconnect program from the file, freeing the file pointer for another file.

4.4.3 `stdin`, `stdout` and `stderr`

When a C program is started, the operating system environment is responsible for opening three files and providing file pointers for them to the program. These files are:

- standard input, file pointer: `stdin`
- standard output, file pointer: `stdout`
- standard error, file pointer: `stderr`

These file pointers are declared in `<stdio.h>`. Normally, `stdin` is connected to the keyboard, `stdout` and `stderr` are connected to the screen. `stdin` and `stdout` may be redirected to files or pipes as described earlier. Pay attention that `stderr` normally appears on the screen even if the standard output is redirected, this prevents error message disappearing down the pipeline.

Since C programs use these three file pointers to communicate with outside components, when we get char from input, or print char on output, we are actually getting or printing these characters via these file pointers to the final destination (standard input, standard output and standard error). Thus, `getchar()` and `putchar(c)` can be defined in terms of `getc, putc, stdin` and `stdout` as:

```
#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)
```

4.4.4 Formatted input and output of files

To format input or output of files, we can use `fscanf()` and `fprintf()`. These functions are similar with `scanf()` and `printf()`, except the first argument is a file pointer. The declaration of these two functions are:

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

An example of sending formated error message to `stderr` is:

```
fprintf(stderr, "Error occurred!\n");
```

4.4.5 Example: replicate program `cat`

p176: normal error handling

p177: advanced error handling (using `stderr` and `exit()`)

4.4.6 Line input and output

The standard library provides an input routine `fgets()`, which can reads the next input line (including `'\n'` character) from a `FILE` pointer to a char array. It will return a `char` pointer pointing to this char array. Its declaration is as follows:

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

At most `maxline - 1` characters will be read. The resulting line is automatically terminated with `'\0'`. When end of file reached or error occurred, it returns `NULL`.

The standard library provides an output routine `fputs()`, which can write a string (which need not contain a newline) to a file. The declaration is as follows:

```
int fputs(char* line, FILE *fp);
```

It returns `EOF` if an error occurs, and zero otherwise.

The library functions `gets` and `puts` are similar to `fgets` and `fputs`, but operate on `FILE` pointers `stdin` and `stdout`. `gets` deletes the terminal `'\n'`, and `puts` adds it.

4.5 MISC Functions

4.5.1 Storage Management

Two functions are used to obtain blocks of memory dynamically:

```
void* malloc(size_t n);
void* calloc(size_t n, size_t size);
```

`malloc()` will return a pointer to `n` bytes of uninitialized storage, or `NULL` if the request cannot be satisfied.

`calloc()` will return a pointer to enough space for an array of `n` objects of the specified size, or `NULL` if the request cannot be satisfied. The storage is initialized to zero.

The pointer returned by `malloc()` or `calloc()` has the proper alignment for the object requested (proper amount of memory), however, it must be case into the appropriate type before assigning to a pointer to hold. For example:

```
int* ip;
ip = (int*) calloc(n, sizeof(int));
```

To free the space pointed by a pointer `p`, of which initially obtained by a call to `malloc()` or `calloc()`, we can call `free(p)`.

5 The UNIX System Interface

5.1 File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files. All peripheral devices are abstracted as files in the file system. So, a single homogeneous interface handles all communication between a program and peripheral devices.

Consider an example of a C program that read content from, or write content to files on the system. Before you can do this, you must inform the system that you wish to **ACCESS** that particular file. The system will check your right to do so (does the file exist? do you have permission to access it?). If you have the access, the system will return a **small non-negative integer** called a *file descriptor*.

A file descriptor is a small non-negative integer, which is an abstract indicator (handle) used to access a file on the system (a file can be an actual file, a pipe, a network socket). All information about an open file is maintained by the system, the user program refers to the file only by the file descriptor.

As mentioned, the input/output are also abstracted as files on the system. If a program wants to access them, it must intend the system to check acceesbility and return the corresponding file descriptors to the program. However, since input/output are used so commonly, that when a program is called by the command interpreter (the "shell"), three files will be opened, their file descriptors `0`, `1` and `2`, will be returned to program so it can use it. By default, the three files are keyboard file (for input), monitor file and monitor file (for output and error display). In fact, the three file descriptors `0`, `1` and `2`, are used as ways for standard input, standard output and standard error of the program. The program don't have to worry about opening files to use them.

The user of a program can redirect I/O to and from files with `<` and `>` when typing the shell command. If these symbols are used, the default assignment of file descriptor `0` and `1` will be changed to the named files. For example:

```
$prog < text1.txt
```

In the above example, the `text1.txt` file will replace keyboard file as the standard input file, system will use file descriptor `0` to identify `text1.txt` and return file descriptor `0` to `prog`. `prog` will use file descriptor `0` to get input.

Similarly, for standard output redirect:

```
$prog > result.txt
```

the `result.txt` file will replace monitor file as the standard output file, system will use file descriptor `1` to identify `result.txt` and return file descriptor `1` to `prog`. `prog` will use file descriptor `1` to do output.

Pay attention that, the change of file assignments are done by the shell, not the program. For program, it always deal with file descriptor `0`, `1` and `2`. It does not know where is input coming from and where is output going to.

5.2 Low Level I/O: `read()` and `write()`

Input and output uses the `read` and `write` system calls. These two system calls are accessed from C programs through two functions called `read()` and `write()`. To use these two functions, you have to `#include <unistd.h>`.

5.2.1 `read()`

The function header for `read()` is:

```c
ssize_t read(int fd, char *buf, size_t count)
```

Parameters

- `fd`: file descriptor, referring the file you wish to read data from
- `buf`: a pointer to a chunk of memory where the program store the data read from the file. Should be a pointer to a char or an array of char, since each char type is one byte. The data is read byte-by-byte.
- `count`: amount of information you want to read from the file in one call of `read()`, in bytes

Behavior

`read(fd, buf, count)` attempts to read up to `count` bytes from the file referred to by the file descriptor `fd` into the buffer started at `buf`.

Return Value

The return value of `read()` can be:

1. the number of bytes read from `fd`. When `fd` doesn't have enough data, the returned value may be smaller than `count`.
2. 0. This indicates end of file has been reached, nothing is read from `fd`.
3. -1. This indicates error occurred.

Any number of bytes can be read in one call. The most common values are 1, which means one character at a time ("unbuffered"), or a number like 1024 or 4096 that corresponds to a physical block size on a peripheral device. Larger sizes will be more efficient because fewer system calls will be made.

5.2.2 `write()`

The function header for `write()` is:

```
ssize_t write(int fd, char *buf, size_t count)
```

Parameters

- `fd`: the file descriptor referring the file you want to write data to
- `buf`: a pointer to a chunk of memory where the program store the data ready to be written to the file. Should be a pointer to a char or an array of char, since each char type is one byte. The data is read byte-by-byte.
- `count`: amount of information you want to write to the file in one call of `write()`, in bytes

Behavior

`write(fd, buf, count)` writes up to `count` bytes from the buffer starting at `buf` to the file referred to by the file descriptor `fd`.

Return Value

The return value of `write()` can be:

1. the number of bytes written to `fd`. If this number is different from `count`, it indicates an error has occurred, for example: there is insufficient space on the underlying physical medium. This can be used to do error checking.

Any number of bytes can be written in one call, as mentioned in `read()`.

5.2.3 Example: copy input to output

This example will show the basic use of `read()` and `write()` function. In order to receive what we read from `read()`, we need a buffer to hold it. We read data from `fd` to the buffer, and call `write()` to put the content in buffer to `fd`. We repeat this process until the returned value of `read()` is not positive (0 or -1).

We'll use `0` as the file descriptor in `read()`, since this is the standard input file descriptor. We'll use `1` as the file descriptor in `write()`, since this is the standard output file descriptor. The code is as follows:

```c
#include <unistd.h>

#define BUFSIZ 5
/* copy input to output */
int main() {
  char buf[BUFSIZ];
  int n;  // hold the number of bytes read

  while ((n = read(0, buf, BUFSIZ)) > 0)
    write(1, buf, n);

  return 0;
}
```

The buffer size is defined as 5. This means each system call we'll process 5 characters. But the reading will not stop unless 0 or -1 is returned by `read()`. If file descriptor `0` is referring keyboard file (keyboard is the standard input), it will continue to read until the keyboard buffer is empty, then the program will wait until the user type other things (I guess these typed-in characters will first go to the keyboard buffer, then they will be read by program). For example, if we print the number of characters being read during each while loop, by adding `printf("\n%d characters has been read.\n", n);` into the while loop. Then we call the program (use default standard input), and input: `12345678`, the result in console would be:

```
12345678
12345
5 characters has been read.
678

4 characters has been read.
```

Notice that each call of `read()` only read 5 characters.

5.2.4 Example: `getchar()`

We can use `read()` to construct `getchar()`, which is higher-level routine. First, let's compare the function header of these two functions:

```
ssize_t read(int fd, char *buf, size_t count)
char getchar(void)
```

Function `getchar()` has no parameter, it returns one character that is read from input stream. It does not require the user of `getchar()` bother the idea of file descriptor, buffer or the number of characters read. It conceals these details in its implementation so user can use it directly in the expected way. This is an example of using lower-level bricks to build higher level structures in the software architecture.

The implementation is simple:

```
#include <unistd.h>
```

```c
#include <stdio.h>

char getcharacter(void) {
  char c;
  return ((read(0, &c, 1)) == 1) ? c : EOF;
}

int main() {
  char c;

  while ((c = getcharacter()) != EOF)
    printf("%c", c);
  return 0;
}
```

5.3 `open()`, `creat()`, `close()`, `unlink()`

When a C program is invoked, only the three default files will be automatically opened and linked to the program (they are standard input, standard output and standard error files, refered by file descriptors `0`, `1` and `2`).

If you want to work with other files, you have to explicitly open other files in order to read or write them. In section File Access, we mentioned a way of operating files by routines defined in `<stdio.h>`. Here, we introduce low-level system calls to do this.

There are two system calls for file operation: `open()` and `creat()`. To use them in your C program, you have to `#include <fcntl.h>`.

5.3.1 `open()`

5.3.1.1 Generals The documentation for `open()` can be found here.

The header of this system call is:

```c
int open(char *name, int flags, int perms);
```

Parameters

- `name`: the file name you want to open
- `flags`: an `int` that specifies how the file is to be opened (constants are defined in `<fcntl.h>`):
  - `O_RDONLY`: open for read-only
  - `O_WRONLY`: open for write-only
  - `O_RDWR`: open for reading and writing
- `perms`: (not mentioned in the book), it is always zero for the uses of `open()` that we will discuss)

Notice that when openning a file, the content of the file will not be deleted.

Return value

If successful, the return value of `open()` is a file descriptor, it will be the lowest-numbered file descriptor not currently open for the process. The file descriptor can be used by subsequent system calls, such as `read()`, `write()`.

If not successful, return -1.

5.3.1.2 Example: open a file for reading

The code is as follows:

```c
#include <fcntl.h>

int fd;  // to hold file descriptor
char name[] = "abc.txt";  // the file name
fd = open(name, O_RDONLY, 0)
```

5.3.2 `creat()`

5.3.2.1 Generals

The documentation for `creat()` can be found here.

The header of this system call is:

```
int creat(char *name, int perms);
```

Parameters

- `name`: the file name to be created
- `perms`: this integer specifies the permission of the file to be created. UNIX file system associates a small integer (with a length of 9-bits) with each file to specify the permission information of them. This integer controls different types of access by different users:
  - Access types:
    * read
    * write
    * execute
  - User types:
    * owner of the file
    * the owner's group
    * all others

  This is a 9-bit long integer, so we can use a 3-digit octal number for specifying the permissions. For example, `0755` (octal number for: `1 1110 1101`) specifies read, write and execute permission for the owner; and read, execute permission for all others.

Return value

`creat()` will return a file descriptor if it was able to creat the file. Otherwise, it will return `-1`.

If the file already exists, `creat()` will truncate it to zero length.

5.3.3 `close()`

The function `close(int fd)` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file.

5.3.4 `unlink()`

The function `unlink(char *name)` removes the file `name` from the file system.

5.3.5 Example: mini `cp` program

This is a simple version of `cp` program. It copies one file to another. The file names are given as command line arguments. It will not copy the permission flag (the 3-digit octal number that describes the permission of the file), but invent a default permission of the copied file.

The behavior of the program is:

```
$ cp file_1 file_2
```

After typing the above line, a file named `file_2` will be created, with the same content as `file_1`.

How to copy? First, we have the file name from the command line argument (should be `argv[1]` and `argv[2]`). We use system call `open()` to open the first file (the file being copied), and use system call `creat()` to create a file with name `argv[2]`. We use two integer variable to hold the file descriptor returned by this two system calls. If no error occurred, we use `read()` system call to read content in `file_1` to our buffer, then use `write()` system call to write content in our buffer to the file. If there is any error occurred during the writing process (i.e. the returned value of `write()` is different from the intended value), we display error.

In this first version,we call `printf()` to display error message through standard output, and call `exit(1)` to exit the program. The code is as follows:

```
#include <stdio.h>   // for displaying error message
#include <stdlib.h>  // for exit() function
#include <unistd.h>  // for read(), write()
#include <fcntl.h>   // for open(), creat()
```

```c
#define PERMS 0666   // default permission flag for copied file
^^I^^I     // RW for owner, group, others
#define BUFFER 100   // buffer size for a single call of read()

/* cp: copy f1 to f2 */
int main(int argc, char *argv[]) {
  int fd1, fd2;  // hold two file descriptors
  int n;  // hold number of bytes read from file1
  char buf[BUFFER];

  // check number of command line arguments
  if (argc != 3) {
    printf("Usage: cp f1 f2");
    exit(1);
  }

  // oepn first file and create second file, exit if error occurred
  if ((fd1 = open(argv[1], O_RDONLY, 0)) == -1) {  // open failed
    printf("Error: can't open %s\n", argv[1]);
    exit(1);
  }

  if ((fd2 = creat(argv[2], PERMS)) == -1) {  // create failed
    printf("Error: can't create %s\n", argv[2]);
    exit(1);
  }

  // use read() and write() system to copy file
  while ((n = read(fd1, buf, BUFFER)) > 0)
    if (n != write(fd2, buf, n)) {  // check if write succeeded
```

```
      printf("Error: couldn't write on file %s\n", argv[2]);
      exit(1);
    }

  return 0;
}
```

5.3.6 Example: mini `cp` program with self-implemented `error()` display

Here, we implement a function called `error()` to combine following function calls:

1. `printf("...", argv[...])`
2. `exit(1)`

In short, it will display error information to standard error file, then call `exit(1)` to stop the program.

This is similar with the simple `printf()` function we built earlier. It is a function with variable-length argument lists. To process this kind of function, we `#include <stdarg.h>`, and the header is:

```
void error(char *format, ...)
```

The three dots represents this function may have variable-length argument(s) after the named argument `format`. To navigate each un-named argument, we declare a `va_list` type object and initialize it:

```
#include <stdarg.h>

void error(char *format, ...) {
  va_list args;
  va_start(args, format);
}
```

Then, we use `fprintf()` to print error message to `stderr` (this is a file pointer):

```
fprintf(stderr, "error: ");
vfprintf(stderr, format, args);
fprintf(stderr, '\n');
```

Here, we use other version of `fprintf()` to print the argument related error message: `vfprintf()`. Its header is as follows:

```
int vfprintf(FILE *fp, const char *format, va_list arg);
```

Parameters

- `fp`: pointer to a `FILE` object that identifies an output stream
- `format`: C string that contains a format string that follows the same specifications as in `printf()`
- `arg` a variable of `va_list` type which has been initialized by calling the `va_start` macro.

Return Value

If succeed, the total number of characters written is returned.

Behavior

Writes the C string pointed by `format` to the file pointed by `fp`, replacing any format specifier in the same way as `printf()` does, using the elements in the variable argument list identified by `arg`, which is a `va_list` type object initialized by the `va_start` macro. So, we don't have to manually scan the `format` string and when a `'%'` is found, call `va_arg()` to retrieve the next argument in `arg` and print according to its type (check miniPrintf() for details of how this is done).

In another words, this function is like automatically extracting all arguments in `arg` and call:

```
fprintf(stderr, format, arg1, arg2, arg3);
```

After displaying error message, we call `va_end()` macro to end the argument retrieving. Then we call `exit(1)` to stop the program. The combined code is as follows:

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

```c
#include <stdlib.h>  // for exit() function

void error(char *format, ...) {
  va_list args;  // for retrieving arguments
  va_start(args, format);  // initialize args

  // print error messages
  fprintf(stderr, "error: ");
  vfprintf(stderr, format, args);
  fprintf(stderr, "\n");

  // call exit() to stop the program
  exit(1);
}
```

Now `error()` has been defined, we call `error()` directly when an error occurred, instead of `printf()` and `exit()`. The code is as follows:

```c
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define PERMS 0666  // permission code: RW for owner, group, others
#define BUFFSIZE 100  // buffer size for read(), write()

void error(char *format, ...);

int main(int argc, char *argv[]) {
  int f1, f2, n;
```

```c
  char buf[BUFFSIZE];

  // try to open files and display error message if failed
  if (argc != 3)  // check command line argument amount
    error("Usage: cp from to");
  if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
    error("cp: can't open %s", argv[1]);
  if ((f2 = creat(argv[2], PERMS)) == -1)
    error("cp: can't create %s, mode%03o", argv[2], PERMS);

  // copy
  while ((n = read(f1, buf, BUFFSIZE)) > 0)
    if (write(f2, buf, n) != n)
      error("cp: write error on file %s", argv[2]);

  return 0;
}
```

5.3.7 Exercise: mini `cat` program

This is exercise 8-1 in the textbook. The behavior of `cat` is:

- if no other file name provided as command line arguments, read from standard input and write to standard output file.
- if other file name provided as command line arguments, read from each file and write to standard output file.
- if error occured, use `error()` to print error message and stop the program

The code is as follows (with simple explanations)

```c
#include <stdio.h>  // for stdin, stderr, stdout file pointers
#include <unistd.h>  // for read(), write()
```

```
#include <stdlib.h>  // for exit()
#include <stdarg.h>  // for variant-length argument function error()
#include <fcntl.h>  // for open() mode flags

#define BUFFSIZE 100

void error(char *format, ...);

int main(int argc, char *argv[]) {

  char buf[BUFFSIZE];  // buffer for read(), write()
  int n;  // count byte number
  int fd;  // to hold input file descriptor

  // check number of arguments
  if (argc == 1) {  // copy stdin to stdout
    while ((n = read(0, buf, BUFFSIZE)) > 0)
      if (n != write(1, buf, n))
^^Ierror("cat: write error to standard output");
    if (n == -1)
      error("cat: read error from standard input");
  } else {
    while (--argc > 0) {
      // try to open file
      if ((fd = open(*++argv, O_RDONLY, 0)) == -1)
^^Ierror("cat: can't open %s", *(argv));
      // write to file
      while ((n = read(fd, buf, BUFFSIZE)) > 0)
^^Iif (n != write(1, buf, n))
^^I  error("cat: write error to standard output");
```

```
      if (n == -1)
^^Ierror("cat: read error from file %s", *(--argv));
      close(fd);  // free fd, so it is ready for next file
    }
  }


  return 0;
}

void error(char *format, ...) {
  va_list args;
  va_start(args, format);

  fprintf(stderr, "Error: ");
  vfprintf(stderr, format, args);
  fprintf(stderr, "\n");

  va_end(args);
  exit(1);
}
```

5.4 Random Access: `lseek()`

The system call `lseek()` provides a way to move around in a file without reading or writing any data.

5.4.1 `lseek()`

The header of `lseek()` is as follows:

```
long lseek(int fd, long offset, int origin);
```

Parameters

- `fd`: the file descriptor which referring the file that are being worked with
- `offset`: number of bytes that `lseek()` offsets the current position.
- `origin`: the code for the position used as relative starting point to measure `offset`. `origin` can have following values:
  - 0: `offset` is to be measured from the beginning
  - 1: `offset` is to be measured from the current position
  - 2: `offset` is to be measured from the end

Return Value

If no error occurred, returns a `long` that gives the new position in the file. Return -1 if an error occurs.

Behavior

After calling this function, the current working position of file referred by `fd` will be moved to the new position (affects system calls like `read()` and `write()`). Pay attention that if position is beyond the range of the file, `read()` will have error, `write()` will add `null` between the end of file to the beginning of newly written content.

5.4.2 Examples

If we want to append data to the end of a file, we open a file in a mode that supports write, then call `lseek(fd, 0L, 2)`, so the current working position will move to end of file.

To get back to beginning (`rewind()`), we just need to call `lseek(fd, 0L, 0)`.

5.5 Example: an implementation of `fopen()` and `getc()`

In this section we build a `FILE` type from scratch and implement `fopen()` and `gect()`. We can use what we build to access files through `FILE` type, not the file descriptor (file descriptors are buried in the back stage).

5.5.1 `FILE` type build-up and Macros

Files in the standard library are described by file pointers rather than file descriptors. A file pointer is a pointer to a structure which is defined as a `FILE` type with a `typedef`, it contains information about a file:

- a pointer to a buffer
  - a buffer is used so file can be read in large chunks
- a count of the number of characters left in the buffer
- a pointer to the next character position in the buffer
- the file descriptor
- flags describing:
  - file openning mode: read or write
  - error states: if error has occurred
  - EOF states: whether end of file has occurred

The code is as follows:

```
// named constants
#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024    //buffer size
#define OPEN_MAX  20      // max number of files open at once

typedef struct _iobuf {
  int cnt;              // characters left in the buffer
  char *ptr;            // next character position in the buffer
  char *base;           // location of buffer
  int flag;             // store info bits (mode of file access and other status)
  int fd;               // file descriptor
```

```
} FILE;

FILE _iob[OPEN_MAX];        // an array of FILE type structures, each element
 ↪  is a FILE

// named constants
#define stdin  (&_iob[0])  // pointer to _iob[0]
#define stdout (&_iob[1])  // pointer to _iob[1]
#define stderr (&_iob[2])  // pointer to _iob[2]

/*Notes:
In the FILE structure, we use only one integer flag to record the status bit,
 ↪  thus a single integer can record multiple status (each bit is a flag)
*/
enum _flags {     /* a leading zero on an integer constant means octal */
  _READ  = 01,    /* file open for reading */
  _WRITE = 02,    /* file open for writing */
  _UNBUF = 04,    /* file is unbuffered */
  _EOF   = 010,   /* EOF has occurred on this file */
  _ERR   = 020    /* error occurred on this file */
};

int _fillbuf(FILE *);          // function header
int _flushbuf(int, FILE *);    // function header

// macros with arguments, argument type should be pointer to FILE structure
#define feof(p) (((p)->flag & _EOF) != 0)       // when EOF has occurred,
 ↪  this is true
#define ferror(p) (((p)->flag & _ERR) != 0)     //when error occurred, this
 ↪  is true
```

```
#define fileno(p) ((p)->fd)                     // get the file descriptor

#define getc(p) (--(p)->cnt >= 0 ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p) (--(p)->cnt >= 0 ? *(p)->ptr++ = (x) : _flushbuf((x), p))

#define getchar() getc(stdin)
#define putchar(x)  putc((x), stdout)
```

5.5.1.1 Setting flags Notice that we have a member in `FILE` structure named `flag`. This single integer is used to record various status of the `FILE` object. Each bit of the inter represents true/false of a specific status. We defined the status in the `enum` type:

```
enum _flags {     /* a leading zero on an integer constant means octal */
  _READ  = 01,    /* file open for reading */
  _WRITE = 02,    /* file open for writing */
  _UNBUF = 04,    /* file is unbuffered */
  _EOF   = 010,   /* EOF has occurred on this file */
  _ERR   = 020    /* error occurred on this file */
};
```

To make it clearer, we translate these octal number to binary to see which bit corresponds which information:

```
00001: read enabled
00010: write enabled
00100: file is unbuffered
01000: EOF reached
10000: error encountered
```

To check if `flag` has any of the above bit set, we just simplily use `&`, as shown in macros `feof()`, `ferror()`, `fileno()`.

5.5.1.2 **Macro** `getc()`

In standard library, `getc(p)` accepts a pointer to a `FILE` object, returns the next character from the file.

In our implementation, to reduce the number of system calls, every required character was first looked in buffer. We first check if `cnt` is 0. This is an integer in `FILE` structure recording the number of remaining characters in the buffer. If this is zero, it means the buffer has gone empty. We call function `_fillbuf(p)` to fill the buffer (the buffer is filled by data retrieved from file via `read()` system call). If its not zero, we give the next character (pointed by `ptr` in `FILE` structure). The character returned is cast into `unsigned char` type to ensure that all characters will be positive.

Macro `getchar()` is also declared, by changing `getc(p)` to `getc(stdin)`.

5.5.1.3 **Macro** `putc()`

In standard library, `putc(x,p)` accepts a character `x` and a pointer `p` to a `FILE` object. It will write `x` to the file and returns the `x`.

In our implementation, to reduce the number of system calls, every character that is intended to be written into file goes to buffer first (we add it to buffer). In this case, `cnt` represents the number of free slots in the buffer. First we check if the buffer is full or not. If it is not full (it can still hold additional characters), we push add the character to the position pointed by `ptr` and update `ptr`'s position. If the buffer is already full, we call function `_flushbuf((x), p)` to flush the buffer (call `write()` to transfer characters in buffer to the file).

Macro `putchar()` is also declared, by changing `putc(x,p)` to `putc((x), stdout)`.

5.5.2 `fopen()`

The original `fopen()`'s behavior: accepting a file name (`char *name`) and the file openning mode (`char *mode`), it will return a pointer to a `FILE` object, which holds information about the status and information of the file. (`FILE` object is stored in an array by the system. The array holds all opened file, each file has a `FILE` object corresponding with). The basic structure of a `FILE` structure object is as follows:

```
typedef struct _iobf {
  int cnt;
```

```
    char *ptr;
    char *base;
    int flag;
    int fd;
} FILE;
```

The working steps of our implementation of `fopen()` function is as follows:

- check open mode passed in to make sure it is valid
- check the array holding `FILE` objects (`_iob`). If the array is full (all `FILE` objects in that array are associated with a file), we return a `NULL` (indicating file openning is not possible at this time). Otherwise, we declare a pointer to `FILE` and let it point to the next available `FILE` object in the `_iob` array. To determine wehther a `FILE` object is associated with a file or not, we check the `fp->flag`. Normally, if a `FILE` object is associated with a file, it must be opened either for writting or reading.
- work by calling system calls to create or open files. The system calls we need is `open()`, `creat()` and `lseek()`. The `lseek()` is used when the file is opened in appending mode (`'a'`).
- update the `FILE` object through the `FILE` pointer `fp` so it is associated with the file being processed. This includes store the file descriptor returned by system calls to `fp->fd`, initialize the opening mode in `FILE` structure.
- return the pointer to the `FILE` object

The code is as follows:

```
#include <fcntl.h>    // for open() mode constants
#include <file.h>     // for defined file structure

#define PERMS 0666    // RW for owner, group, others

FILE *fopen(char *name, char *mode) {
```

```
int fd;      // hold file descriptor
FILE *fp;    // a pointer to FILE stored in _iob[] (defined in file.c)

// check open mode passed in, if error, return NULL
if (*mode != 'r' && *mode != 'w' && *mode != 'a')
  return NULL;

// find an empty slot in _iob[] array, which holds opened FILE
for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
  if ((fp->flag & (_READ | _WRITE)) == 0)
    break;
if (fp >= _iob + OPEN_MAX)
  return NULL;

// we now have an empty slot of FILE, now deal with mode and create FILE if
↪  necessary
// create FILE by calling system call, according to different modes
if (*mode == 'w')
  fd = creat(*name, PERMS);
else if (*mode == 'a') {
  // first try to open the file, if file not exist, create one
  if ((fd = open(*name, O_WRONLY, 0)) == -1)  // file open failed
    fd = creat(*name, PERMS);
  lseek(fd, 0L, 2);
}
else
  fd = open(*name, O_RDONLY, 0);



// now, update the FILE object (empty FILE object in _iob[])
```

```c
  if (fd == -1)
    return NULL;

  fp->fd = fd;
  fp->cnt = 0;
  fp->base = NULL;
  fp->flag = (*mode == 'r') ? _READ : _WRITE;

  return fp;
}
```

Notice that, in `fopen()`, we don't allocate any buffer space (we initialize the buffer pointer `fp->base` as `NULL`, set buffer counter `fp->cnt` as 0). In `fopen()`, we mostly do the following two things:

1. getting the file opened and positioned at the right place
2. setting the flag bits to indicate the proper state.

5.5.3 `_fillbuf()`

In function `fopen()` we didn't engage in allocating new buffer spaces for the `FILE` object. This is done by the function `_fillbuf()`. When working with file pointer, we call the macro `getc()` to get a character from the opened file. In our implementation, `getc()` will first examine if the `fp->cnt` is zero, if so, it will call `_fillbuf()` to solve this issue. In a higher level, `_fillbuf()` will take care of buffer related operations and return the next character (possibly read the file first and fill the buffer, then get character from the buffer).

The header of `_fillbuf()` is:

```c
int _fillbuf(FILE *p);
```

The working step of `_fillbuf()` is as follows:

- first, we have to check the open mode of the file. `getc()` must work when `_READ` flag is set and no errors or end of file reached. If not, return `EOF`

- determine the buffer size. If _UNBUF flag is set for FILE, the buffer size should be 1. Otherwise, the buffer size is determined by predefined value (BUFSIZ).
- check if the FILE object holds buffer that was previously allocated, if not, we have to allocate it and update fp->base to let it store the address of this newly allocated buffer memory.

  ```
  if (fp->base == NULL)
    if (fp->base = (char*) malloc(bufsize) == NULL)
      return EOF;
  ```

  The pointer type returned by malloc() is void, we have to cast it into char*. We also checked if the memory allocation is successful. If error occured, return EOF.
- after we allocated the memory space, we have to update fp->ptr, so it points to the beginning of the allocated memory space.
- use system call read() to read data from file to the buffer (in FILE), update fp->cnt so it reflects
- check the return value from read(). The possible return values from read() are:
  - the number of bytes read from fd (the file descriptor argument passed in read()
  - 0: end of file has been reached, nothing is read from fd
  - -1: error occurred

  If end of file or error occurred, we need to update the flag.
- at the end of the _fillbuf(), we need to return the next character read (this is to finish getc()'s task)

5.5.4 Initialization of _iob[]

We are using an array of FILE structure objects (the structure name is _iobuf, FILE is a type name referring this structure, defined by typedef) to hold the opened file. We have to define the _iob[] array somewhere in the program and initialize for stdin, stdout and stderr. These three files use 0, 1 and 2 as their file descriptors. We use this to initialize the FILE object:

```
FILE _iob[OPEN_MAX] = {
  {0, (char *) 0, (char *) 0, _READ, 0},
  {0, (char *) 0, (char *) 0, _WRITE, 1},
  {0, (char *) 0, (char *) 0, _WRITE | _UNBUF, 2}
};
```

Notice that `stderr` is to be written unbuffered.

6 Place holder