Contents

1 Generals **2**

1.1 Some Facts 2

1.2 headers 4

1.3 `printf()` formatting 4

1.4 Symbolic Constants 4

1.5 Character Input and Output 5

1.6 Arrays 6

1.7 Enumeration constant 6

1.8 type-cast an expression 7

1.9 Bitwise operators 7

1.10 Operators can be used with assignment operators 8

1.11 External Variables 8

1.12 Command-line Arguments 8

1.12.1 Example: `echo` 9

1.12.2 Example: `pattern_finding` 10

1.12.3 Optional arguments example: `pattern_finding` extended 12

1.13 Pointers to Functions 16

1.13.1 Example: qsort() which takes a comp() function pointer 16

2 Input and Output **17**

2.1 Standard Input and Output 17

2.1.1 Input redirection 17

2.1.2 Output redirection 18

2.1.3 Pipe between two programs 18

2.1.4 Include header file 19

2.1.5 Macros in standard library 19

2.1.6 Formatted output: printf 19

2.1.7 Function `sprintf()` 19

2.2 Variable-length Argument Lists 20

2.2.1 Declare a function that takes varying amounts of arguments 20
2.2.2 Traverse the argument list and final cleanup 20
2.2.2.1 Type `va_list` 20
2.2.2.2 Macro `va_start` 20
2.2.2.3 Macro `va_arg` 21
2.2.2.4 Macro `va_end` 21
2.2.3 `miniPrintf()` example 22
2.3 Formated Input: `scanf()` 24
2.3.1 A simple example 24
2.3.2 Declaration and arguments 24
2.4 File Access 26
2.4.1 Opening a file 26
2.4.2 Accessing the file 27
2.4.3 `stdin`, `stdout` and `stderr` 27
2.4.4 Formatted input and output of files 28
2.4.5 Example: replicate program `cat` 28
2.4.6 Line input and output 28
2.5 MISC Functions 29
2.5.1 Storage Management 29

3 The UNIX System Interface 30
3.1 File Descriptors 30
3.2 Low Level I/O: `read` and `write` 31

4 Place Holder 31

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:

  ```c
  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 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:

```c
#include <stdio.h>

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

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.

1.5 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`.

1.6 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;
}
```

1.7 Enumeration constant

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

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

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

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

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

```
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:

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

1.8 type-cast an expression

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

```
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.

1.9 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 `!=`.

1.10 Operators can be used with assignment operators

p64

`+, -, *, /, %, «, », &, ^, |`

1.11 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:

```
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.

1.12 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]
  ```

1.12.1 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;
}
```

1.12.2 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:

```
#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");
```

```c
  else
    while (getlines(line, MAXLINE) > 0)
      if (strstr(line, argv[1]) != NULL) {
        printf("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;
}
```

1.12.3 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:

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

```c
#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':
        except = 1;
        break;
      case 'n':
```

```
        number = 1;
        break;
      default:
        printf("find: illegal option %c\n", c);
        argc = 0;  // this will terminate the program
        found = -1;
        break;
      }
    }

  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:
        Print the line based on value of variable except and the found
↪   result.
        To 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;
        When except = 0, we print lines that are found, so found == 1;
      */
      if ((strstr(line, *argv) != NULL) != except) {
        if (number)  // if the number flag is true, we print the line
         ↪  number
```

```c
          printf("%d", line_num);
        printf("%s", line);
        found++;
      }

    }

  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;
}
```

1.13 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.

1.13.1 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:

```c
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);
}
```

2 Input and Output

2.1 Standard Input and Output

2.1.1 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.

2.1.2 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.

2.1.3 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`.

2.1.4 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`).

2.1.5 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.

2.1.6 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 `*`.

2.1.7 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.

2.2 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.

2.2.1 Declare a function that takes varying amounts of arguments

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

```
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).

2.2.2 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>`.

2.2.2.1 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:

```
#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
  //...
}
```

2.2.2.2 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`:

```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
  //...
}
```

`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.

2.2.2.3 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`:

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

2.2.2.4 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);
```

2.2.3 `miniPrintf()` example

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.

2.3 Formated Input: scanf()

p171.

2.3.1 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).

2.3.2 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.

2.4 File Access

2.4.1 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:

- location of a buffer
- current character position in the buffer
- 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`.

2.4.2 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.

2.4.3 `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)
```

2.4.4 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 firs 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");
```

2.4.5 Example: replicate program `cat`

p176: normal error handling

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

2.4.6 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.

2.5 MISC Functions

2.5.1 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)`.

3 The UNIX System Interface

3.1 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.

3.2 Low Level I/O: `read` and `write`

Input and output uses the `read` and `write` system calls. This two system calls are accessed from C programs through two functions called `read()` and `write()`.

4 Place Holder