

Contents

1	Some Facts	1
2	headers	3
3	printf() formatting	3
4	Symbolic Constants	3
5	Character Input and Output	4
6	Arrays	5
7	Enumeration constant	5
8	type-cast an expression	6
9	Bitwise operators	6
10	Operators can be used with assignment operators	7
11	External Variables	7
12	Command-line Arguments	7
12.1	Example: <code>echo</code>	8
12.2	Example: <code>pattern_finding</code>	9
12.3	Optional arguments example: <code>pattern_finding</code> extended	11
13	Pointers to Functions	15
13.1	Schematic picture	15
13.2	Implementation	15

1 Some Facts

- assignments associate from right to left.

- arithmetic 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 `*`, `++`, `--`)

place holder.

2 headers

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

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.Of", 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
```

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:

```

#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.

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.

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;
}
```

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
```

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.

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 `!=`.

10 Operators can be used with assignment operators

p64

`+`, `-`, `*`, `/`, `%`, `«`, `»`, `&`, `^`, `|`

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.

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 inputted 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]
```

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:

```
#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;
}

```

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");
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;
}

```

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 optional 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 any other strings occurred 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':
                    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
    ↪ examined 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

```

```

        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;
}

```

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. In this section, a sorting program will be used as an example to illustrate the idea. It will also combine the command line arguments.

13.1 Schematic picture

The sorting program has following feature:

1. the program can be invoked by `$ sort -n < text.txt` or `$ sort < text.txt`
2. if optional argument `-n` is used, the lines will be sorted numerically, otherwise, lines will be sorted lexicographically
3. read all lines of input and store them
4. sort the lines by sorting pointers pointing to them (based on whether `-n` is given or not)
5. print the lines in order

13.2 Implementation

For the base program (read lines and qsort them), please check the corresponding example code.

Code for this part (along with explanations):

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

#define MAXLINES 5000 // max number of lines to be sorted

char* lineptr[MAXLINES]; // an array of char pointers to text lines
```

```

int readlines(char* lineptr[], int maxlines);
void writelines(char* lineptr[], int nlines);
// the fourth parameter is a pointer to function, we may choose different
↪ comparing functions to be used in sorting
void qsorts(void* lineptr[], int left, int right, int (*comp)(void*,
↪ void*));
int numcmp(char*, char*);

// with optional arguments
int main(int argc, char* argv[]) {
    int nlines; // hold the total number of lines read
    int numeric = 0; // flag of -n option; 1 if numeric sort requested

    // check if optional argument is typed
    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;

    // read lines and sort according to numeric value
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        /* qsorts(lineptr, 0, nlines - 1); */
        qsorts((void **) lineptr, 0, nlines - 1, (int (*)(void*,
↪ void*)) (numeric ? numcmp : strcmp));

        writelines(lineptr, nlines);
        return 0;
    }

    else {
        printf("error: input too big to sort\n");
        return 1;
    }
}

```



```

}

}

#define MAXLEN 1000 // max length of any input line
int getlines(char*, int);
char* alloc(int); // memory allocator, declared here, compiled in a
    ↪ separate file (using example 5-4: a memory allocator toy)

/* readlines():
How does this work:
- use getlines() function to get each line from the input buffer and store
    ↪ in line[] variable; Also store the length of each line (not including )
- call alloc() to request a chunk of memory from a pre-allocated memory
    ↪ space (reservoir), a pointer to this chunk of memory will hold the
    ↪ address
- call strcpy() to copy content in line[] to the requested chunk of memory
- store the memory address hold in p to lineptr[]'s slots
store each read-line to the C-string array pointed by lineptr
The maximum input line number is the maxlines parameter
*/
int readlines(char* lineptr[], int maxlines) {
    int len; // hold the length of line being read
    int nlines; // hold the number of lines read, should be smaller than
        ↪ maxlines

    char* p; // a pointer used to hold the address of allocated
    char line[MAXLEN]; // temp char array to hold currently read line, pass
        ↪ in getline() function

```

```

nlines = 0;

while ((len = getlines(line, MAXLEN)) > 0) // if len > 0, a line has
    ↪ been read
    /*
        - if line num exceeded max, or memory allocation failed (not enough
    ↪ room)
        - the amount of memory is len, which is returned by getlines()
        - this length contains the tailing '\n' character. So len = # of char +
    ↪ 1
        - but we'll replace '\n' with '\0'
        - also, line[] filled by getlines() has following structure:
            ['a', 'b', 'c', '\n', '\0']
            after replacing '\n' with '\0':
            ['a', 'b', 'c', '\0', '\0']

            We'll only allocate room for four chars, but the fifth one will still
    ↪ be copied to the master reservoir (used by alloc() to dispense memory).
    ↪ It will be over-written by the first letter of next line, because
    ↪ alloc() thinks this slot is the next empty slot, and the next request:
    ↪ p = alloc(len) will return p pointing to this.
    */
    if (nlines >= maxlines || (p = alloc(len)) == NULL)
        return -1; // indicate readlines() function failed to read all the
        ↪ lines
    else {
        line[len - 1] = '\0'; // replace '\n' with '\0', so line[] is a
        ↪ C-string
        strcpy(p, line); // copy content in line to spaces pointed by p
        lineptr[nlines++] = p; // the address stored in p is now in
        ↪ lineptr[]
    }

```

```

    }

    return nlines; // return the number of lines read from the input buffer
}

/* writelines():
Print the lines stored in the master reservoir by dereferencing pointer
↪ stored in lineptr[]
*/
void writelines(char* lineptr[], int nlines) {
    for (int i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}

/* swap():
Swap two elements in an array of pointers. Used by qsorts()
Parameters:
    void* v[]: an array of pointers, potentially to any type
    int i: index of first element
    int j: index of second element

swap v[i] and v[j]
*/
void swap(void* v[], int i, int j) {
    void* temp;
    temp = v[j];
    v[j] = v[i];
    v[i] = temp;
}

```

/ qsorts():*

Sort v[left] ... v[right] into increasing order

Parameters:

void v[]: void* is a generic pointer type, here, it means pointer*

→ *accepts an array of pointers to any type (like template?)*

int left: left boundary of sorting

int right: right boundary of sorting

*int (*comp)(void*, void*): a pointer to a function which accepts two*

→ *void* arguments (two pointers to any type), the return type of this*

→ *function is int.*

Pay attention that comp is declared as a pointer to function. We can use it

→ *as a normal function by dereferencing it : (*comp)*

The parenthese is needed when writting the parameter or actual using

→ *function in the body of qsorts. If no parenthese, int *comp(void*,*

→ *void*) means a function named comp that returns a pointer to integer*

→ *(not a pointer to function!)*

**/*

```
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);
}

/* numcmp():
   compare s1 and s2 on a leading numeric value
*/
int numcmp(char* s1, char* s2) {
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);

    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}

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

```

```

// count < max - 1 for storing the last null terminator
for (count = 0; count < max - 1 && (ch = getchar()) != EOF && ch != '\n';
    ↪ ++count)
    *(line++) = ch;

if (ch == '\n') {
    *(line++) = ch;
    count++;
}

*line = '\0'; // add null terminator to the array

return count; // not containing the '\0' terminator
}

```