Semantics (VIII)

Formatted File I/O

- For formatted input or output of files, the functions `fscanf` and `fprintf` may be used.
  - Identical to `scanf` and `printf`, except that the first argument is a file pointer

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

  - Show formatIO.c and input.txt, run ./formatIO input.txt output.txt, and show the output.txt.

```c
/**
 * formatted file I/O
 * formatIO.c
 */

#include <stdio.h>

int main (int argc, char* argv[]) {
     FILE *fp1, *fp2;
     int value;

     fp1 = fopen(argv[1], "r");
     fp2 = fopen(argv[2], "w");
     while (fscanf(fp1, "%d", &value) == 1) {
       fprintf(fp2, "%d", value);
     }

     fclose(fp1);
     fclose(fp2);

     return 0;
}
```

input.txt

```
1
10
100
1000
10000
```

```
$ ./formatIO input.txt output.txt
$ cat output.txt
110100100010000
```

  - Show formatIO2.c, run ./formatIO2 with the same input.txt, and show the output.txt. Ask why the outputs are different. [%d vs %c]

```c
/**
 * formatted file I/O
 * formatIO2.c
 */

#include <stdio.h>

int main (int argc, char* argv[]) {
      FILE *fp1, *fp2;
      unsigned char value;

      fp1 = fopen(argv[1], "r");
      fp2 = fopen(argv[2], "w");
      while (fscanf(fp1, "%c", &value) == 1) {
            fprintf(fp2, "%c", value);
      }

      fclose(fp1);
      fclose(fp2);

      return 0;
}
```

input.txt

```
1
10
100
1000
10000
```

```
$ ./formatIO input.txt output.txt
$ cat output.txt
1
10
100
1000
10000
```

Unformatted File I/O

- `int getc(FILE *fp)`
  - return the character from a file referred to by `fp`
  - it returns EOF for end of file or error
- `int putc(int c, FILE *fp)`
  - writes the character `c` to the file `fp` and return the character written, or EOF if an error occurs

- Fun facts
  - When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`, and are declared in <`stdio.h`>.
  - `getchar` and `putchar` can be defined in terms of `getc`, `putc`, `stdin`, and `stdout` as follow:
    - `#define getchar() getc(stdin)`
    - `#define putchar(c) putc(c, stdout)`
  - The file pointers `stdin` and `stdout` are objects of type `FILE *`. They are **constants**, however, not variables, so it is **not possible to assign to them**.

- Example:
  - Implement a C program to realize the command "`cat`"
  - "cat" can be used in two ways:
    - `$ cat file1 file2`: print the contents of file1 and file2 to the terminal
    - `$ cat`: print the standard input contents to the standard output

```
mbp-190250:code yingli$ cat test1.txt
A
B
C
D
mbp-190250:code yingli$ cat test1.txt test2.txt
A
B
C
D
1
2
3
4
mbp-190250:code yingli$ cat
HELLO
HELLO
WORLD
WORLD
mbp-190250:code yingli$
```

  - To implement it, we can either use a conditional statement to choose between `getChar` and `getC` to read the contents from files or standard input, or only use `getC` and choose

between using a regular file pointers for input files and using `stdin` for reading from standard input)

- The following program gives an example to use file pointers coupled with `getC` and `putC` to implement "`cat`"

```c
/**
 * cat: concatenate file
 */

#include <stdio.h>

/*filecopy: copy file ifp to file ofp */
void filecopy (FILE *ifp, FILE *ofp) {
     int c;

     while ((c = getc(ifp)) != EOF)
       putc(c, ofp);
}

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

     if (argc == 1) // no args; copy standard input
       filecopy (stdin, stdout);
     else
       while (--argc > 0)
              if ((fp = fopen(*++argv, "r")) == NULL) {
                     printf("cat: can't open %s\n", *argv);
                     return 1;
              }
              else {
                     filecopy(fp, stdout);
                     fclose(fp);
              }

     return 0;
}
```

Exception Semantics

- Error Handling
  - Function return value
    - C has the convention that a function that executes properly should return a 0 if it is not using the function's return value for something else.
  - Assertions. Check the correctness of assumptions during run-time
  - Signals. A software interrupt delivered to a process. OS uses it to report exceptional situations to an executing program (e.g., references to invalid memory addresses).
  - Exceptions are thrown/caught. This disrupts the normal flow of execution.

- Assertions
  - Statements used to check the correctness of assumptions made in a program during run-time.
  - Debug tool

- C example:

```
void assert (int expression);
```

  - If the expression meaning is 0 (false), the expression, source code filename, and line number are sent to standard error, and then the abort function is invoked.

```c
// GPACalc.c
#include <stdio.h>
#include <assert.h>

float GPACalc (int size, float *ary) {
      float sum = 0;
      for (int i = 0; i < size; i++) {
            // check invalid gpa
            assert (ary[i] >= 0.0 && ary[i] <= 4.3);
            sum += ary[i];
      }
      return sum/size;
}

int main () {
      float a[] = {3.5, 3.8, 4.0, 4.1, 4.5};

      printf("%f\n", GPACalc(5, a));

     return 0;
}
```

