Input/Output and Standard C Library. Preprocessor and Building Programs

Jan Faigl

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 06

B3B36PRG – Programming in C

Overview of the Lecture

Part I

Input and Output

Text vs. Binary Files

- In terms of machine processing, there is no difference between text and binary files.
- **Text files** are supposed to be human readable. *Without additional specific software tools.*
  - Bytes represent characters and the content is (usually) organized into lines.
  - Different markers for the *end-of-line* are used (1 or 2 bytes).
  - There can be a special marker for the *end-of-file* (Ctrl-Z).

    It is from CP/M and later used in DOS. It is not widely used in Unix like systems.
- Processing text files can be **character**, **formatted**, or **line** oriented with the functions from the standard library `stdio.h`.
  - Character oriented – `putc()`, `getc()`. *Or for* `stdout/stdin` – `putchar()`, `getchar()`.

    ```
    int putc(int c, FILE *stream);
    int getc(FILE *stream);
    ```
  - Formatted i/o – `fprintf()` and `fscanf()`. *Or for* `stdout/stdin` – `printf()`, `scanf()`.
  - Line oriented – `fputs()`, `fgets()`. *Or for* `stdout/stdin` – `puts()`, `gets()`.
- In general, text files are sequences of bytes, but numeric values as text need to be parsed and formatted in writing.
- Numbers in binary files may deal with byte ordering. *Endianness – ARM vs. x86.*

File open

- Functions for input/output are defined in the standard library `<stdio.h>`.
- The file access is through using a pointer to a file (stream) `FILE*`.
- File can be opened using `fopen()`.

```
FILE* fopen(const char * restrict path, const char * restrict mode);
```

Notice, the `restrict` keyword

- File operations are **stream oriented** – sequential reading/writing.
  - The current position in the file is like a **cursor**.
  - At the file opening, the cursor is set to the beginning of the file (if not specified otherwise).
- The mode of the file operations is specified in the `mode` parameter.
  - `"r"` – reading from the file – cursor is set to the beginning of the file.

    The program (user) needs to have sufficient rights for reading from the file.
  - `"w"` – writing to the file – cursor is set to the beginning of the file.

    A new file is created if it does not exists; otherwise the content of the file is cleared.
  - `"a"` – append to the file – the cursor is set to the end of the file.
  - The modes can be combined, such as `"r+"` open the file for reading and writing.

    See `man fopen`.

`fopen()`, `fclose()`, and `feof()`

- Test if the file has been opened.

```
1 char *fname = "file.txt";
2
3 if ((f = fopen(fname, "r")) == NULL) {
4   fprintf(stderr, "Error: open file '%s'\n", fname);
5 }
```

- Close file – `int fclose(FILE *stream);`

```
1 if (fclose(f) == EOF) {
2   fprintf(stderr, "Error: close file '%s'\n", fname);
3 }
```

- Test of reaching the end-of-file (EOF) – `int feof(FILE *stream);`

File Positioning

- Every stream has a cursor that associated to a position in the file.
- The position can be set using `offset` relatively to `whence`.

```
int fseek(FILE *stream, long offset, int whence);
```

  where `whence`
  - `SEEK_SET` – set the position from the beginning of file;
  - `SEEK_CUR` – relatively to the current file position;
  - `SEEK_END` – relatively to the end of file.

  If the position is successfully set, `fseek()` returns 0.
- `void rewind(FILE *stream);` sets the position to the beginning of file.
- The position can be stored and set by the functions using structure `fpos_t`.

```
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

See `man fseek`, `man rewind`.

File Stream Modes

- Modes in the `fopen()` can be combined.

```
FILE* fopen(const char * restrict path, const char * restrict mode);
```

  - `"r"` open for reading.
  - `"w"` Open for writing (file is created if it does not exist).
  - `"a"` open for appending (set cursor to the end of file or create a new file if it does not exists).
  - `"r+"` open for reading and writing (starts at beginning).
  - `"w+"` open for reading and writing (truncate if file exists).
  - `"a+"` open for reading and writing (append if file exists).
- There are restrictions for the combined modes with `"+"`.
  - We cannot switch from reading to writing without calling a file-positioning function or reaching the end of file.
  - We cannot switch from writing to reading without calling `fflush()` or calling a file-positioning function.

Temporary Files

- `FILE* tmpfile(void);` – creates a temporary file that exists until it is closed or the program exists.
- `char* tmpnam(char *str);` – generates a name for a temporary file in `P_tmpdir` directory that is defined in `stdio.h`.
  - If `str` is `NULL`, the function creates a name and store it in a static variable and return a pointer to it; otherwise the name is copied into the buffer `str`.

    The buffer `str` is expected to be at least `L_tmpnam` bytes in length (defined in `stdio.h`).

```
const char *fname1 = tmpnam(NULL);
printf("Temp fname1: \"%s\".\n",
    fname1);
const char *fname2 = tmpnam(NULL);
printf("Temp fname2: \"%s\".\n",
    fname2);
...
printf("Temp fname1: \"%s\".\n",
    fname1);
```

```
!clang demo-tmpnam.c -o demo && ./demo
Temp fname1: "/tmp/tmp.0.0dWD5H".
Temp fname2: "/tmp/tmp.1.R9OLiP".
The name is stored in the static variable.
The pointer fname1 points to the static
    variable.
Thus, its content is changed by the tmpnam
    () call.
Temp fname1: "/tmp/tmp.1.R9OLiP".
```

lec06/demo-tmpnam.c

File Buffering

- `int fflush(FILE *stream);` – flushes buffer for the given `stream`.
  - `fflush(NULL);` – flushes all buffers (all output streams).
- Change the buffering mode, size, and location of the buffer.
  `int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);`
  The `mode` can be one of the following macros.
  `_IOFBF` – full buffering. Data are read from the stream when buffer is empty and written to the stream when it is full.
  `_IOLBF` – line buffering. Data are read or written from/to the stream one line at a time.
  `_IONBF` – no buffer. Direct reading and writing without buffer.

```
#define BUFFER_SIZE 512
char buffer[BUFFER_SIZE];

setvbuf(stream, buffer, _IOFBF, BUFFER_SIZE);
```

See `man setvbuf`.

- `void setbuf(FILE * restrict stream, char * restrict buf);`
  is equivalent to `setvbuf(stream, buf, buf ? _IOFBF : _IONBF, BUFSIZ);`

Detecting End-of-File and Error Conditions

- Three possible "errors" can occur during reading data, such as using `fscanf`.
  - End-of-file – we reach the end of file.
    Or, the `stdin` *stream is closed.*
  - Read error – the read function is unable to read data from the stream.
  - Matching failure – the read data does not match the requested format.
- Each stream `FILE*` has two indicators.
  - **Error indicator** – indicates that a read or write error occurs.
  - **End-of-file (EOF) indicator** – is set when the end of file is reached.
    The EOF is set when the attempt to read beyond the end-of-file, not when the last byte is read.
- The indicators can be read (tested if the indicator is set or not) and cleared.
  - `int ferror(FILE *stream);` – tests the stream has set the error indicator.
  - `int feof(FILE *stream);` – tests if the stream has set the end-of-file indicator.
  - `void clearerr(FILE *stream);` – clear the error and end-of-file indicators.

Reading and Writing Single Character (Byte)

- Functions for reading from `stdin` and `stdout`.
  - `int getchar(void)` and `int putchar(int c)`.
  - Both function return `int` value, to indicate an error (`EOF`).
  - The written and read values converted to `unsigned char`.
- The variants of the functions for the specific stream.
  - `int getc(FILE *stream);` and
    `int putc(int c, FILE *stream);`
  - `getchar()` is equivalent to `getc(stdin)`.
  - `putchar()` is equivalent to `putc()` with the `stdout` stream.
- Reading byte-by-byte (`unsigned char`) can be also used to read binary data, e.g., to construct 4 bytes length `int` from the four byte (char) values.

Example – Naive Copy using getc() and putc() 1/2

- Simple copy program based on reading bytes from `stdin` and writing them to `stdout`.

```
1  int c;
2  int bytes = 0;
3  while ((c = getc(stdin)) != EOF) {
4     if (putc(c, stdout) == EOF) {
5        fprintf(stderr, "Error in putc");
6        break;
7     }
8     bytes += 1;
9  }
```

lec06/copy-getc_putc.c

Example – Naive Copy using getc() and putc() 2/2

- We can count the number of bytes, and thus the time needed to copy the file.

```
1  #include <sys/time.h>
2  ...
3
4  struct timeval t1, t2;
5  gettimeofday(&t1, NULL);
6
7  ... // copy the stdin -> stdout
8
9  gettimeofday(&t2, NULL);
10 double dt = t2.tv_sec - t1.tv_sec + ((t2.tv_usec - t1.tv_usec) / 1000000.0);
11 double mb = bytes / (1024 * 1024);
12 fprintf(stderr, "%.2lf MB/sec\n", mb / dt);
```

lec06/copy-getc_putc.c

- Example of creating random file and using the program.

```
clang -O2 copy-getc_putc.c
dd bs=512m count=1 if=/dev/random of=/tmp/rand1.dat
1+0 records in
1+0 records out
536870912 bytes transferred in 2.437674 secs (220239034 bytes/sec)
./a.out < /tmp/rand1.dat >/tmp/rand2.dat
398.45 MB/sec
```


Line Oriented I/O

- A whole line (text) can be read by `gets()` and `fgets()` functions.

```
char* gets(char *str);
char* fgets(char * restrict str, int size, FILE * restrict stream);
```

- `gets()` cannot be used securely due to lack of bounds checking.
- A line can be written by `fputs()` an `puts()`.
- `puts()` write the given string and a newline character to the `stdout` stream.
- `puts()` and `fputs()` return a non-negative integer on success and `EOF` on an error.
  See `man fgets`, `man fputs`.
- Alternatively, the line can be read by `getline()`.

```
ssize_t getline(char ** restrict linep, site_t * restrict linecapp,
FILE * restrict stream);
```

Expand the buffer via `realloc()`, *see* `man fgetline`.

Capacity of the buffer, or if `*linep==NULL` *(if* `linep` *points to* `NULL`*) a new buffer is allocated.*

Formatted I/O – fscanf()

- `int fscanf(FILE *file, const char *format, ...);`
- It returns a number of read items. For example, for the input

```
record 1 13.4
```

the statement

```
int r = fscanf(f, "%s %d %lf\n", str, &i, &d);
```

sets (in the case of success) the variable `r` to the value 3.

- For strings reading, it is necessary to respect the size of the allocated memory, by using the limited length of the read string.

```
char str[10];
int r = fscanf(f, "%9s %d %lf\n", str, &i, &d);
```

lec06/file_scanf.c

Formatted I/O – fprintf()

- `int fprintf(FILE *file, const *format, ...);`

```
int main(int argc, char *argv[])
{
   char *fname = argc > 1 ? argv[1] : "out.txt";
   FILE *f;
   if ((f = fopen(fname, "w")) == NULL) {
      fprintf(stderr, "Error: Open file '%s'\n", fname);
      return -1;
   }
   fprintf(f, "Program arguments argc: %d\n", argc);
   for (int i = 0; i < argc; ++i) {
      fprintf(f, "argv[%d]='%s'\n", i, argv[i]);
   }
   if (fclose(f) == EOF) {
      fprintf(stderr, "Error: Close file '%s'\n", fname);
      return -1;
   }
   return 0;
}
```

lec06/file_printf.c

Block Read/Write

- We can use `fread()` and `fwrite()` to read/write a block of data.

```
size_t fread(void * restrict ptr,
      size_t size, size_t nmemb,
      FILE * restrict stream);

size_t fwrite(const void * restrict ptr,
      size_t size, size_t nmemb,
      FILE * restrict stream);
```

Use `const` *to indicate (*`ptr`*) is used only for reading.*

Block Read/Write – Example 1/5

- Program to read/write a given (as `#define NUMB`) number of `int` values using `#define BUFSIZE` length buffer.
- Writing is enabled by the optional program argument `-w`.
- File for reading/writing is a mandatory program argument.

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>
#include <stdlib.h>

#include <sys/time.h>

#include "my_assert.h"

#ifndef BUFSIZE
#define BUFSIZE 32768
#endif

#ifndef NUMB
#define NUMB 4098
#endif

int main(int argc, char *argv[])
{
   int c = 0;
   _Bool read = true;
   const char *fname = NULL;
   FILE *file;
   const char *mode = "r";
   while (argc-- > 1) {
      fprintf(stderr, "DEBUG: argc: %d '%s'\n", argc, argv[argc]);
      if (strcmp(argv[argc], "-w") == 0) {
         fprintf(stderr, "DEBUG: enable writting\n");
         read = false; // enable writting
         mode = "w";
      } else {
         fname = argv[argc];
      }
   } // end while
```

lec06/demo-block_io.c

Block Read/Write – Example 2/5

```c
   file = fopen(fname, mode);
   if (!file) {
      fprintf(stderr, "ERROR: Cannot open file '%s', error %d - %s\n", fname, errno,
       strerror(errno));
      return -1;
   }
   int *data = (int*)malloc(NUMB * sizeof(int));
   my_assert(data __LINE__, __FILE__);
   struct timeval t1, t2;
   gettimeofday(&t1, NULL);
   if (read) {                                                    /* READ FILE */
      fprintf(stderr, "INFO: Read from the file '%s'\n", fname);
      c = fread(data, sizeof(int), NUMB, file);
      if (c != NUMB) {
         fprintf(stderr, "WARN: Read only %i objects (int)\n", c);
      } else {
         fprintf(stderr, "DEBUG: Read %i objects (int)\n", c);
      }
   } else {                                                       /* WRITE FILE */
      char buffer[BUFSIZE];
      if (setvbuf(file, buffer, _IOFBF, BUFSIZE)) { /* SET BUFFER */
         fprintf(stderr, "WARN: Cannot set buffer");
      }
```


Block Read/Write – Example 3/5

```c
      fprintf(stderr, "INFO: Write to the file '%s'\n", fname);
      c = fwrite(data, sizeof(int), NUMB, file);
      if (c != NUMB) {
         fprintf(stderr, "WARN: Write only %i objects (int)\n", c);
      } else {
         fprintf(stderr, "DEBUG: Write %i objects (int)\n", c);
      }
      fflush(file);
   }

   gettimeofday(&t2, NULL);
   double dt = t2.tv_sec - t1.tv_sec + ((t2.tv_usec - t1.tv_usec) / 1000000.0);
   double mb = (sizeof(int) * c)/ (1024 * 1024);
   fprintf(stderr, "DEBUG: feof: %i ferror: %i\n", feof(file), ferror(file));
   fprintf(stderr, "INFO: %s %lu MB\n", (read ? "read" : "write"), sizeof(int)*NUMB
    /(1024 * 1024));
   fprintf(stderr, "INFO: %.2lf MB/sec\n", mb / dt);
   free(data);
   return EXIT_SUCCESS;
}
```

lec06/demo-block_io.c

Block Read/Write – Example 3/5

- Default `BUFSIZE` (32 kB) to write/read 10^8 integer values (~480 MB).

```
clang -DNUMB=100000000 demo-block_io.c && ./a.out -w a 2>&1 | grep INFO
INFO: Write to the file 'a'
INFO: write 381 MB
INFO: 10.78 MB/sec

./a.out a 2>&1 | grep INFO
INFO: Read from the file 'a'
INFO: read 381 MB
INFO: 2214.03 MB/sec
```

- Try to read more elements results in `feof()`, but not in `ferror()`.

```
clang -DNUMB=200000000 demo-block_io.c && ./a.out  a
DEBUG: argc: 1 'a'
INFO: Read from the file 'a'
WARN: Read only 100000000 objects (int)

DEBUG: feof: 1 ferror: 0

INFO: read 762 MB
INFO: 1623.18 MB/sec
```

lec06/demo-block_io.c

Block Read/Write – Example 5/5

- Increased write buffer `BUFSIZE` (128 MB) improves writing performance.

```
clang -DNUMB=100000000 -DBUFSIZE=134217728 demo-block_io.c && ./
    a.out -w aa 2>&1 | grep INFO
INFO: Write to the file 'aa'
INFO: write 381 MB
INFO: 325.51 MB/sec
```

- But does not improve reading performance, which relies on the standard size of the buffer.

```
clang -DNUMB=100000000 -DBUFSIZE=134217728 demo-block_io.c &&
    ./a.out  aa 2>&1 | grep INFO
INFO: Read from the file 'aa'
INFO: read 381 MB
INFO: 1693.39 MB/sec
```

lec06/demo-block_io.c

Blocking and Non-Blocking I/O Operations

- Usually, I/O operations are considered as **blocking requested**.
  - System call does not return control to the program until the requested I/O is completed.
    It is motivated that we need all the requested data and I/O operations are usually slower than the other parts of the program. We have to wait for the data anyway.
  - It is also called **synchronous** programming.
- **Non-Blocking** system calls do not wait, and thus do not block the application.
  - It is suitable for network programming, multiple clients, graphical user interface, or when we need to avoid "deadlock" or too long waiting due to slow or not reliable communication.
  - Call for reading requested data read (and "return") only data that are actually available in the input buffer.
- **Asynchronous** programming with **non-blocking** calls.
  - Return control to the application immediately .
  - Data are transfered to/from buffer "on the background."

Callback function, triggering a signal, etc.

Non-Blocking I/O Operations – Example

- Setting the file stream (**file descriptor** – `fd`) to the `O_NONBLOCK` mode.

Usable also for socket descriptor.

- Note that using non-blocking operations does not make too much sense for regular files.
- It is more suitable for reading from block devices such as serial port `/dev/ttyACM0`.
  - We can set `O_NONBLOCK` flag for a file descriptor using `fcntl()`.

```c
#include <fcntl.h> // POSIX

// open file by the open() system call that return a file descriptor
int fd = open("/dev/ttyUSB0",  O_RDWR, S_IRUSR | S_IWUSR);

// read the current settings first
int flags = fcntl(fd, F_GETFL, 0);

// then, set the O_NONBLOCK flag
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

  - Then, calling `read()` might not provide the requested number of bytes if fewer bytes are currently available in the input buffer.

Key Press without Enter

- Reading from the standard (terminal) input is usually line oriented, which allows editing the program input before its confirmation by end-of-line using `Enter`.
- Reading character from `stdin` can be made by the `getchar()` function.
- However, the input is buffered to read line, and it is necessary to press the Enter key by default.
- We can avoid that by setting the terminal to a *raw* mode.

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

int c;
while ((c = getchar()) != 'q') {
  if (isalpha(c)) {
    printf("Key '%c' is alphabetic;", c);
  } else if (isspace(c)) {
    printf("Key '%c' is space character;", c);
  } else if (isdigit(c)) {
    printf("Key '%c' is decimal digit;", c);
  } else if (isblank(c)) {
    printf("Key is blank;");
  } else {
    printf("Key is something else;");
  }
  printf(" ascii: %s\n",
        isascii(c) ? "true" : "false");
}
return 0;
```

lec06/demo-getchar.c

Key Press without Enter – Example

- We can switch the `stdin` to the `raw` mode using `termios` or using `stty` tool.

```c
void call_termios(int reset)
{
   static struct termios tio, tioOld;
   tcgetattr(STDIN_FILENO, &tio);
   if (reset) {
      tcsetattr(STDIN_FILENO, TCSANOW, &tioOld);
   } else {
      tioOld = tio; //backup
      cfmakeraw(&tio);
      // assure echo is disabled
      tio.c_lflag &= ~ECHO;
      // enable output postprocessing
      tio.c_oflag |= OPOST;
      tcsetattr(STDIN_FILENO, TCSANOW, &tio);
   }
}
```

```c
void call_stty(int reset)
{
   if (reset) {
      system("stty -raw opost echo");
   } else {
      system("stty raw opost -echo");
   }
}
```

- `int system(const char *string);`

  hands `string` to the command interperter.
- Returns the program (shell) exit status.
- Returns 127 is the shell execution failed.

- Usage `clang demo-getchar.c -o demo-getchar`
  - Standard "Enter" mode: `./demo-getchar`
  - Raw mode - termios: `./demo-getchar termios`
  - Raw mode - stty: `./demo-getchar stty`

lec06/demo-getchar.c

Part II

Selected Standard Libraries

Standard Library

- The C programming language itself does not provide operations for input/output, more complex mathematical operations, nor
  - string operations;
  - dynamic allocation;
  - run-time error handling.
- These and further functions are included in the standard library.
  - Library – the compiled code is linked to the program, such as `libc.so`.

    E.g., see `ldd a.out`.
  - Header files contain function prototypes, types , macros, etc.

<assert.h>	**<inttypes.h>**	<signal.h>	**<stdlib.h>**
<complex.h>	<iso646.h>	<stdarg.h>	**<string.h>**
<ctype.h>	**<limits.h>**	<stdbool.h>	<tgmath.h>
<errno.h>	<locale.h>	<stddef.h>	<time.h>
<fenv.h>	**<math.h>**	**<stdint.h>**	<wchar.h>
<float.h>	**<setjmp.h>**	**<stdio.h>**	<wctype.h>

Standard library – Overview

- `<stdio.h>` – Input and output (including formatted).
- `<stdlib.h>` – Math function, dynamic memory allocation, conversion of strings to number.
  - Sorting – `qsort()`.
  - Searching – `bsearch()`.
  - Random numbers – `rand()`.
- `<limits.h>` – Ranges of numeric types.
- `<math.h>` – Math functions.
- `<errno.h>` – Definition of the error values.
- `<assert.h>` – Handling runtime erros.

- `<ctype.h>` – character classification, e.g., see `lec06/demo-getchar.c`.
- `<string.h>` – Strings and memory transfers, i.e., `memcpy()`.
- `<locale.h>` – Internationalization.
- `<time.h>` – Date and time.

Standard Library (POSIX)

Relation to the operating system (OS).

Single UNIX Specification (SUS).

POSIX – Portable Operating System Interface.

- `<stdlib.h>` – Function calls and OS resources.
- `<signal.h>` – Asynchronous events.
- `<unistd.h>` – Processes , read/write files, ...
- `<pthread.h>` – Threads (POSIX Threads).
- `<threads.h>` – Standard thread library in C11.

Advanced Programming in the UNIX Environment, 3rd edition, *W. Richard Stevens, Stephen A. Rago* Addison-Wesley, 2013, ISBN 978-0-321-63773-4

Mathematical Functions

- `<math.h>` – basic function for computing with "real" numbers.
  - Root and power of floating point number `x`.
    `double sqrt(double x);`, `float sqrtf(float x);`
  - `double pow(double x, double y);` – power.
  - `double atan2(double y, double x);` – *arctan* y/x with quadrand determination.
  - Symbolic constants – `M_PI`, `M_PI_2`, `M_PI_4`, etc.
    - `#define M_PI 3.14159265358979323846`
    - `#define M_PI_2 1.57079632679489661923`
    - `#define M_PI_4 0.78539816339744830962`
  - `isfinite()`, `isnan()`, `isless()`, `...` – comparision of "real" numbers.
  - `round()`, `ceil()`, `floor()` – rounding and assignment to integer.
- `<complex.h>` – function for complex numbers. *ISO C99*
- `<fenv.h>` – function for control rounding and representation according to IEEE 754.

  `man math`

Variable Arguments <stdarg.h>

- It allows writing a function with a variable number of arguments.

  Similarly as in the functions `printf()` *and* `scanf()`.
- The header file `<stdarg.h>` defines.
  - Type `va_list` and macros.
  - `void va_start(va_list ap, parmN);` – initiate `va_list`.
  - *type* `va_arg(va_list ap, type);` – fetch next variable.
  - `void va_end(va_list ap);` – cleanup before function return.
  - `void va_copy(va_list dest, va_list src);` – copy a variable argument list.
- We have to pass the number of arguments to the functions with variable number of arguments to known how many values we can retrieved from the stack.

  Arguments are passed with stack; thus, we need size of the particular arguments to access them in the memory and interpret the memory blocks, e.g., as int or double values.

Example – Variable Arguments <stdarg.h>

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  int even_numbers(int n, ...);
5  int main(void)
6  {
7     printf("Number of even numbers: %i\n", even_numbers(2, 1, 2));        // returns 1
8     printf("Number of even numbers: %i\n", even_numbers(4, 1, 3, 4, 5));  // returns 1
9     printf("Number of even numbers: %i\n", even_numbers(3, 2, 4, 6));     // returns 3
10    return 0;
11 }
12
13 int even_numbers(int n, ...)
14 {
15    int c = 0;
16    va_list ap;
17    va_start(ap, n);
18    for (int i = 0; i < n; ++i) {
19       int v = va_arg(ap, int);
20       (v % 2 == 0) ? c += 1 : 0;
21    }
22    va_end(ap);
23    return c;
24 }
```

`lec06/demo-va_args.c`

Error Handling – errno

- Basic error codes are defined in `<errno.h>`.
- These codes are used in standard library as indicators that are set in the global variable `errno` in a case of an error during the function call.
  - If `fopen()` fails, it returns `NULL`, which does not provide the cause of the failure.
  - The cause of failure can be stored in the `errno` variable.
- Text description of the numeric error codes are defined in `<string.h>`.
  - String can be obtain by the function.

    `char* strerror(int errnum);`

Example – errno in Fie Open fopen()

```
1  #include <stdio.h>
2  #include <errno.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6     FILE *f = fopen("soubor.txt", "r");
7     if (f == NULL) {
8        int r = errno;
9        printf("Open file failed errno value %d\n", errno);
10       printf("String error '%s'\n", strerror(r));
11    }
12    return 0;
13 }
```

`lec06/errno.c`

- Program output if the file does not exist.

```
Open file failed errno value 2
String error 'No such file or directory'
```

- Program output for an attempt to open a file without having sufficient access rights.

```
Open file failed errno value 13
String error 'Permission denied'
```

Testing Macro assert()

- We can add tests for a particular value of the variables, for debugging.

Test and indications of possible errors, e.g., due to a wrong function argument.

- Such test can be made by the macro `assert(expr)` from `<assert.h>`.
- If `expr` is not logical 1 (`true`) the program is terminated and the particular line and the name of the source file is printed.
- We can disable the macro by definition of the macro `NDEBUG`. `man assert.`
  - It is not for run-time errors detection.

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

int main(int argc, char *argv[])
{
   assert(argc > 1);
   printf("program argc: %d\n", argc);
   return 0;
}
```

lec06/assert.c

Example of assert() Usage

- Compile the program with the `assert()` macro and executing the program with/without program argument. lec06/assert.c

```
clang assert.c -o assert
./assert
Assertion failed: (argc > 1), function main, file assert.c, line 5.
zsh: abort      ./assert

./assert 2
start argc: 2
```

- Compile the program without the macro and executing it with/without program argument. lec06/assert.c

```
clang -DNDEBUG assert.c -o assert
./assert
program start argc: 1
./assert 2
program start argc: 2
```

The `assert()` macro is not for run-time errors detection!

Long Jumps

- `<setjmp.h>` defines function `setjmp()` and `longjmp()` for jumps across functions.

Note that the `goto` statement can be used only within a function.

- `setjmp()` stores the actual state of the registers and if the function returns non-zero value, the function `longjmp()` has been called.
- During `longjmp()` call, the values of the registers are restored and the program continues the execution from the location of the `setjmp()` call.

We can use `setjmp()` and `longjmp()` to implement handling exceptional states similarly as `try-catch`.

```c
#include <setjmp.h>
jmp_buf jb;
int compute(int x, int y);
void error_handler(void);
if (setjmp(jb) == 0) {
   r = compute(x, y);
   return 0;
} else {
   error_handler();
   return -1;
}
```

```c
int compute(int x, int y) {
   if (y == 0) {
      longjmp(jb, 1);
   } else {
      x = (x + y * 2);
      return (x / y);
   }
}
void error_handler(void)  {
   printf("Error\n");
}
```


Communication with the Environment – <stdlib.h>

- The header file `<stdlib.h>` defines standard program return values `EXIT_FAILURE` and `EXIT_SUCCESS`.
- A value of the environment variable can be retrieved by the `getenv()` function.

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

int main(void)
{
   printf("USER: %s\n", getenv("USER"));
   printf("HOME: %s\n", getenv("HOME"));
   return EXIT_SUCCESS;
}
```

lec06/demo-getenv.c

- `void exit(int status);` – the program is terminated as it will be by calling `return(status)` in the `main()` function.
- We can register a function that will be called at the program exit.

```
int atexit(void (*func)(void));
```

- The program can be aborted by calling `void abort(void)`.

The registered functions by the `atexit()` are not called.

Example – atexit(), abort(), and exit()

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

void cleanup(void);
void last_word(void);

int main(void)
{
   atexit(cleanup); // register function
   atexit(last_word); // register function
   const char *howToExit = getenv("HOW_TO_EXIT");
   if (howToExit && strcmp(howToExit, "EXIT") == 0) {
      printf("Force exit\n");
      exit(EXIT_FAILURE);
   } else if (howToExit && strcmp(howToExit, "ABORT") == 0) {
      printf("Force abort\n");
      abort();
   }
   printf("Normal exit\n");
   return EXIT_SUCCESS;
}

void cleanup(void)
{
   printf("Perform cleanup at the program exit!\n");
}

void last_word(void)
{
   printf("Bye, bye!\n");
}
```

- Example of usage.

```
clang demo-atexit.c -o atexit

% ./atexit; echo $?
Normal exit
Bye, bye!
Perform cleanup at the program exit!
0

% HOW_TO_EXIT=EXIT ./atexit; echo $?
Force exit
Bye, bye!
Perform cleanup at the program exit!
1

% HOW_TO_EXIT=ABORT ./atexit; echo $?
Force abort
zsh: abort HOW_TO_EXIT=ABORT ./atexit
134
```

lec06/demo-atexit.c

Part III

Preprocessor and Building Programs

Variables – Scope and Visibility

- **Local variables**
  - A variable declared in the body of a function is the **local variable**.
  - Using the keyword `static` we can declared **static local variables**.
  - Local variables are visible (and accessible) only within the function.
- **External variables** (global variables)
  - Variables declared outside the body of any function.
  - They have **static storage duration**; the value is stored as the program is running.

Like a local static variable.

  - External variable has **file scope**, i.e., it is visible from its point of the declaration to the end of the enclosing file.
    - We can refer to the external variable from other files by using the `extern` keyword.
    - In a one file, we define the variable, e.g., as `int var;`.
    - In other files, we declare the external variable as `extern int var;`.
  - We can restrict the visibility of the **global variable** to be within the single file only by the `static` keyword.

Organizing C Program

- Particular source files can be organized in many ways.
- A possible ordering of particular parts can be as follows:
  1. `#include` directives;
  2. `#define` directives;
  3. Type definitions;
  4. Declarations of external variables;
  5. Prototypes for functions other than `main()` (if any);
  6. Definition of the `main()` function (if so);
  7. Definition of other functions.

Header Files

- Header files provide the way how to share defined macros, variables, and use functions defined in other modules (source files) and libraries.
- `#include` directive has two forms.
  - `#include <filename>` – to include header files that are searched from system directives.
  - `#include "filename"` – to include header files that are searched from the current directory.
- The places to be searched for the header files can be altered, e.g., using the command line options such as `-Ipath`.
- It is not recommended to use brackets < and > for including own header files.
- It is also not recommended to use absolute paths.

Neither windows nor unix like absolute paths.

If you needed them, it is an indication you most likely do not understand the process of compilation and building the program/project.

Sharing Macros and Types, Function Prototypes and External Variables

- Let have three files `graph.h`, `graph.c`, and `main.c` for which we like to share macros and types, and also functions and external variables defined in `graph.c` in `main.c`.

`graph.h`:

```
#define GRAPH_SIZE 1000
typedef struct {
   ...
} edget_s;
typedef struct {
   edges_s *edges;
   int size;
} graph_s;
// make the graph_global extern
extern graph_s graph_global;
// declare function prototype
graph_s* load_graph(const char *filename);
```

`graph.c`:

```
#include "graph.h"
graph_s graph_global = { NULL, GRAPH_SIZE };
graph_s* load_graph(const char *filename)
{
   ...
}
```

`main.c`:

```
#include "graph.h"
int main(int argc, char *argv[])
{
   // we can use function from graph.c
   graph_s *graph = load_graph(...
   // we can also use the global variable
   // declared as extern in the graph.h
   if (global_graph.size != GRAPH_SIZE) {
    ...
```


Protecting Header Files

- Header files can be included from other header files.
- Due to sequence of header files includes, the same type can be defined multiple times.
- We can protect header files from multiple includes by using the preprocessor macros.

```
#ifndef GRAPH_H
#define GRAPH_H
...
// header file body here
// it is processed only if GRAPH_H is not defined
// therefore, after the first include,
// the macro GRAPH_H is defined
// and the body is not processed during therepeated includes
...
#endif
```

- Or using `#pragma once`, which is, however, non-standard preprocessor directive.

```
#pragma once
...
// header file body here
```


Macros

- Macro definitions are by the `#define` directive.
  - The macros can be parametrized to define function-like macros.
  - Already defined macros can be undefined by the `#undef` command.
- File inclusion is by the `#include` directive.
- Conditional compilation – `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`.
- Miscellaneous directives.
  - `#error` – produces error message, which can be combined with `#if`, e.g., to test sufficient size of `MAX_INT`.
  - `#line` – alter the way how lines are numbered (`__LINE__` and `__FILE__` macros).
  - `#pragma` – provides a way to request a special behaviour from the compiler.

C99 introduces `_Pragma` operator used for "destringing" the string literals and pass them to `#pragma` operator.

Predefined Macros

- There are several predefined macros that provide information about the compilation and compiler as integer constant or string literal.
  - `__LINE__` – Line number of the file being compiled (processed).
  - `__FILE__` – Name of the file being compiled.
  - `__DATE__` – Date of the compilation (in the form "Mmm dd yyyy").
  - `__TIME__` – Time of the compilation (in the form "hh:mm:ss").
  - `__STDC__` – 1 if the compiler conforms to the C standard (C89 or C99).
- C99 introduces further macros, such as the following versions.
  - `__STDC_VERSION__` – Version of C standard supported.
    - For C89 it is `199409L`.
    - For C99 it is `199901L`.
- It also introduces identifier `__func__` that provides the name of the actual function.

It is actually not a macro, but behaves similarly.

Defining Macros Outside a Program

- We can control the compilation using the preprocessor macros.
- The macros can be defined outside a program source code during the compilation, and passed to the compiler as particular arguments.
- For `gcc` and `clang` it is the `-D` argument.
  - `gcc -DDEBUG=1 main.c` – define macro `DEBUG` and set it to 1.
  - `gcc -DNDEBUG main.c` – define `NDEBUG` to disable `assert()` macro.

See `man assert`.

- The macros can be also undefined, e.g., by the `-U` argument.
- Having the option to define the macros by the compiler options, we can control the compilation process according to the particular environment and desired target platform.

Compiling and Linking

- Programs composed of several modules (source files) can be build by an individual compilation of particular files, e.g., using `-c` option of the compiler.
- Then, all object files can be linked to a single binary executable file.
- Using the `-l`*lib*, we can add a particular *lib* library.
- E.g., let have source files `moduleA.c`, `moduleB.c`, and `main.c` that also depends on the *math* library (`-lm`). The program can be build as follows.

```
clang -c moduleA.c -o moduleA.o
clang -c moduleB.c -o moduleB.o
clang -c main.c -o main.o

clang main.o moduleB.o moduleA.o -lm -o main
```

Be aware that the order of the files is important for resolving dependencies! It is incremental, and only the function(s) needed in first modules are linked from the other modules. For example functions called in `main.o` with implementation in `mainA.o` and `mainB.o`; and functions called in `mainB.o` that have implementation in `mainA.o`.

Makefile

- Some building system may be suitable for project with several files.
- One of the most common tools is the `GNU make` or the `make`.

Notice, there are many building systems that may provide different features, e.g., designed for the fast evaluation of the dependencies like `ninja`.

- For `make`, the building rules are written in the `Makefile` files.

`http://www.gnu.org/software/make/make.html`

- The rules define targets, dependencies, and action to build the targets based on the dependencies.

target : **dependencies** *colon*
action *tabulator*

- Target (dependencies) can be symbolic name or file name(s).

main.o : **main.c**
clang -c main.c -o main.o

- The building receipt can be a simple usege of file names and compiler options.

The main advantage of Makefiles is flexibility arising from unified variables, internal make variables, and templates, as most of the sources can be compiled similarly.

Example Makefile

- Pattern rule for compiling source files **.c** to object files **.o**.
- Wildcards are used to compile all source files in the directory.

Can be suitable for small project. In general, explicit listings of the files is more appropriate.

```
CC:=ccache $(CC)
CFLAGS+=-O2

OBJS=$(patsubst %.c,%.o,$(wildcard *.c))

TARGET=program

bin: $(TARGET)

$(OBJS): %.o: %.c
  $(CC) -c $< $(CFLAGS) $(CPPFLAGS) -o $@

$(TARGET): $(OBJS)
  $(CC) $(OBJS) $(LDFLAGS) -o $@

clean:
  $(RM) $(OBJS) $(TARGET)
```

ccache

CC=clang make vs CC=gcc make

- **The order of the files is important during the linking!**

Part IV

Part 3 – Assignment HW 04 and HW 06

HW 04 – Assignment

Topic: **Text processing – Grep**

Mandatory: **2 points**; Optional: **3 points**; Bonus : none

- **Motivation**: Memory allocation and string processing.
- **Goal**: Familiar yourself with string processing.
- **Assignment**: `https://cw.fel.cvut.cz/wiki/courses/b3b36prg/hw/hw04`
  - Read input file and search for a pattern.
  - Optional assignment – redirect of `stdint`; regular expressions; color output.
- Deadline: 13.04.2024, 23:59 AoE.

HW 06 – Assignment

Topic: **Circular buffer**

Mandatory: **2 points**; Optional: **2 points**; Bonus : none

- **Motivation**: Implement library according to defined header file with function prototypes. Compile and link shared library.
- **Goal**: Familiar yourself with circular buffer, building and usage of shared library.
- **Assignment**: `https://cw.fel.cvut.cz/wiki/courses/b3b36prg/hw/hw06`
  - Fixed size circular buffer.
  - Optional assignment – dynamically resized circular buffer.
- Deadline: 27.04.2024, 23:59 AoE.

Summary of the Lecture

Topics Discussed

- I/O operations
  - File operations
  - Character oriented input/output
  - Text files
  - Block oriented input/output
  - Non-blocking input/output
  - Terminal input/output
- Selected functions of standard library
  - Overview of functions in standard C and POSIX libraries
  - Variable number of arguments
  - Error handling
- Building Programs
  - Variables and their scope and visibility
  - Organizing source codes and using header files
  - Preprocessor macros
  - Makefiles
- Next: Parallel programming