Input/Output and Standard C Library

Jan Faigl

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

Lecture 06

B3B36PRG – C Programming Language

Overview of the Lecture

Part I

Input and Output

Text vs Binary Files

- There is no significant difference between text and binary files from the machine processing perspective
- Text files are oriented to be a human readable
  - In text files, bytes represent characters
  - 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.
  - For parsing text files, we can use
    - Character oriented functions – `putchar()`, `getchar()`, `putc()`, `getc()`
    - Functions for formatted i/o – `printf()` and `scanf()` as shortcuts for the `fprintf()` and `fscanf()` with the `stdin` and `stdout` streams
    - Line oriented functions – `puts()`, `gets()` and variants `fputs()`, `fgets()`
- Text files can be considered as a sequence of bytes
  - Numeric values as text need to be parsed and formatted in writing
- Numbers in binary files may deal with byte ordering *E.g., 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

- Operations with the files are
  - Stream oriented – sequential reading/writing
  - The current position in the file is like a **cursor**
  - At the opening the file, the cursor is set to the beginning of the file
- The mode of the file operations is specified in the `mode` parameter
  - `"r"` – reading from the file
    The program (user) needs to have sufficient rights for reading from the file.
  - `"w"` – writing to 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, e.g., `"r+"` open the file for reading and writing

See `man fopen`

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

- Test if the file has been opened

```
char *fname = "file.txt";

if ((f = fopen(fname, "r")) == NULL) {
  fprintf(stderr, "Error: open file '%s'\n", fname);
}
```

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

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

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

File Positioning

- Every stream has a cursor, i.e., an associated file position
- 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

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

See `man fseek`, `man rewind`, etc

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 *s);` – generates a name for a temporary file
  - If `s` is `NULL`, it creates a name and store it in a static variable and return a pointer to it
  - Otherwise it copies the string into the provided character array (`s`) and returns the pointer to the first character of the array

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);
```

- `void setbuf(FILE * restrict stream, char * restrict buf);`
  – similar to `setvbuf()` but with default mode

Detecting End-of-File and Error Conditions

- Three possible "errors" can occur during reading data (e.g., `fscanf`)
  - End-of-file – we reach the end of file
    Or, the stream is closed, e.g., `stdin`
  - 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 indicator** – is set when the end of file is reached
- The indicators can be read (tested if the indicator is set or not) and clear the error and eof indicators
  - `int ferror(FILE *stream);`
  - `void clearerr(FILE *stream);`
  - `int feof(FILE *stream);`

Reading and Writing Single Byte

- Basic function for reading from `stdin` and `stdout` are
  - `getchar()` and `putchar()`
  - Both function return `int` value, to indicate an error (`EOF`)
  - The written and read values are converted to `unsigned char`
- The variants of the function for the specific stream are
  - `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 – Copy using getc() and putc() 1/2

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

```c
int c;
int bytes = 0;
while ((c = getc(stdin)) != EOF) {
   if (putc(c, stdout) == EOF) {
      fprintf(stderr, "Error in putc");
      break;
   }
   bytes += 1;
}
```

lec06/copy-getc_putc.c

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

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

```c
#include <sys/time.h>
...

struct timeval t1, t2;
gettimeofday(&t1, NULL);

... // copy the stdin -> stdout

gettimeofday(&t2, NULL);
double dt = t2.tv_sec - t1.tv_sec + ((t2.tv_usec - t1.tv_usec) / 1000000.0);
double mb = bytes / (1024 * 1024);
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 7.897227 secs (67982205 bytes/sec)
./a.out < /tmp/rand1.dat >/tmp/rand2.dat
326.10 MB/sec
```


Line Oriented I/O

- A whole line (text) can be read by

```c
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 error

See man fgets, man fputs

- Alternatively, the line can be read by getline()

```c
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, e.g., 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

  `r == 3`

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

```c
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, ...);`

```c
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

```c
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
```

```c
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");
   }
```

lec06/demo-block_io.c

Block Read/Write – Example 3/5

```
        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 application 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 for unrelated I/O to complete, 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 will 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" *Call back, triggering a signal, etc.*

Non-Blocking I/O Operations – Example

- Setting the file stream (file descriptor) to the `O_NONBLOCK` mode

  Also for socket descriptor
- For reading from regular files it does not make too much sense to use non-blocking operations
- Reading from block devices such as serial port, e.g., `/dev/ttyS10` may be more suitable
  - We can set `O_NONBLOCK` flag for a file descriptor using `fcntl()`

```
#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()` will provide the requested number of bytes are fewer bytes that are currently available in the buffer

Key Press without Enter

- Reading character from `stdin` can be made by the `getchar()` function
- However, the input is buffered to read line, i.e., it is necessary to press Enter key by default
- We can avoid that by setting the terminal to a *raw* mode

```
#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`

```
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);
      tio.c_lflag &= ~ECHO; // assure echo is disabled
      tio.c_oflag |= OPOST; // enable output postprocessing
      tcsetattr(STDIN_FILENO, TCSANOW, &tio);
   }
}
```

- Or we can use the `stty` tool

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

- 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 that is a part of the C compiler
  - Library – the compiled code is linked to the program, e.g., `libc.so`

    Viz e.g., `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

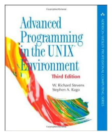
Standard Library (POSIX)

Relation to the operating system (OS)

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);

  va_copy() *has been introduced in* C99
- We have to pass the number of arguments to the functions with variable number of arguments

Example – Variable Arguments <stdarg.h>

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

int even_numbers(int n, ...);
int main(void)
{
   printf("Number of even numbers: %i\n", even_numbers(2, 1, 2));
   printf("Number of even numbers: %i\n", even_numbers(4, 1, 3, 4, 5));
   printf("Number of even numbers: %i\n", even_numbers(3, 2, 4, 6));
   return 0;
}

int even_numbers(int n, ...)
{
   int c = 0;
   va_list ap;
   va_start(ap, n);
   for (int i = 0; i < n; ++i) {
      int v = va_arg(ap, int);
      (v % 2 == 0) ? c += 1 : 0;
   }
   va_end(ap);
   return c;
}
```

lec06/demo-va_args.c

Error handling

- 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, e.g.,
  - If file open 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

- File open

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

int main(int argc, char *argv[]) {
   FILE *f = fopen("soubor.txt", "r");
   if (f == NULL) {
      int r = errno;
      printf("Open file failed errno value %d\n", errno);
      printf("String error '%s'\n", strerror(r));
   }
   return 0;
}
```

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 particular value of the variables, for debugging
- 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
- Macro includes particular code to the program

  It provides a relatively straightforward way to evaluate and indicate possible errors, e.g., due to a wrong function argument.
- We can disable the macro by definition of the macro `NDEBUG`

  `man assert`
- Example

```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 the `assert()` macro and executing the program with/without program argument

```
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

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

lec06/assert.c

Long Jumps

- The `goto` statement can be used only within a function
- `<setjmp.h>` defines function `setjmp()` and `longjmp()` for jumps across functions
- `setjmp()` stores the actual state of the registers and if the function return 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;
}
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 get be retrieved by the `getenv()`

```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 by the `int atexit(void (*func)(void));`
- The program can be aborted by calling `void abort(void)`, in this case, 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

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

- Next: Parallel programming