

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

PRG – Programming in C

Overview of the Lecture

- Part 1 – Input and Output
File Operations
Character Oriented I/O
Text Files
Block Oriented I/O
Non-Blocking I/O
Terminal I/O

K. N. King: chapters 22

- Part 2 – Selected Standard Libraries
Standard library – Selected Functions
Error Handling

K. N. King: chapters 21, 23, 24, 26, and 27

- Part 3 – Preprocessor and Building Programs
Organization of Source Files
Preprocessor
Building Programs
- Part 4 – Assignment HW 04 and HW 06.

K. N. King: chapters 10, 14, and 15

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 exist; 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 – `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 exist).
 - "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.0.0d4D5H".
Temp fname2: "/tmp/tmp.1.R90LiP".
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.R90LiP".
```

File Operations Character Oriented I/O Text Files Block Oriented I/O Non-Blocking I/O Terminal I/O

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);

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 11 / 69

File Operations Character Oriented I/O Text Files Block Oriented I/O Non-Blocking I/O Terminal I/O

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.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 12 / 69

File Operations Character Oriented I/O Text Files Block Oriented I/O Non-Blocking I/O Terminal I/O

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);
 - getc() is equivalent to getc(stdin).
 - putc() 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.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 14 / 69

File Operations Character Oriented I/O Text Files Block Oriented I/O Non-Blocking I/O Terminal I/O

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

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 15 / 69

File Operations Character Oriented I/O Text Files Block Oriented I/O Non-Blocking I/O Terminal I/O

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 struct timeval t1, t2;
4 gettimeofday(&t1, NULL);
5 ... // copy the stdin -> stdout
6
7 gettimeofday(&t2, NULL);
8 double dt = t2.tv_sec + (t2.tv_usec - t1.tv_usec) / 1000000.0;
9 double mb = bytes / (1024 * 1024);
10 fprintf(stderr, "%.21f 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
```

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 16 / 69

File Operations Character Oriented I/O Text Files Block Oriented I/O Non-Blocking I/O Terminal I/O

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() and 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, size_t * restrict linecap,
FILE * restrict stream);
```

Expand the buffer via realloc(), see man getline.
Capacity of the buffer, or if *linep==NULL (if linep points to NULL) a new buffer is allocated.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 18 / 69

File Operations Character Oriented I/O Text Files Block Oriented I/O Non-Blocking I/O Terminal I/O

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

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 19 / 69

File Operations Character Oriented I/O Text Files Block Oriented I/O Non-Blocking I/O Terminal I/O

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

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 20 / 69

File Operations Character Oriented I/O Text Files Block Oriented I/O Non-Blocking I/O Terminal I/O

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.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 22 / 69

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.

```
1 #include <stdio.h> 19 int main(int argc, char *argv[])
2 #include <string.h> 20 {
3 #include <errno.h> 21     int c = 0;
4 #include <stdbool.h> 22     _Bool read = true;
5 #include <stdlib.h> 23     FILE *file;
6 24     const char *fname = NULL;
7 #include <sys/time.h> 25     const char *mode = "r";
8 26     while (argc-- > 1) {
9         fprintf(stderr, "DEBUG: argc: %d '%s'\n", argc, argv[argc]);
10        if (strcmp(argv[argc], "-w") == 0) {
11            fprintf(stderr, "DEBUG: enable writing\n");
12            #define BUFSIZE 32768
13            #define BUFSIZE 32768
14            #endif
15            #if BUFSIZE
16            #define NUMB 4098
17            #endif
18            #define NUMB 4098
19            #endif
20            mode = "w";
21            fname = argv[argc];
22        } else {
23            #define NUMB 4098
24            #endif
25            mode = "r";
26            fname = argv[argc];
27        }
28    }
29    // end while
30    // end while
31    // end while
32    // end while
33    // end while
34    // end while
35    // end while
36    // end while
37    // end while
38    // end while
39    // end while
40    // end while
41    // end while
42    // end while
43    // end while
44    // end while
45    // end while
46    // end while
47    // end while
48    // end while
49    // end while
50    // end while
51    // end while
52    // end while
53    // end while
54    // end while
55    // end while
56    // end while
57    // end while
58    // end while
59    // end while
60    // end while
61    // end while
62    // end while
63    // end while
64    // end while
65    // end while
66    // end while
67    // end while
68    // end while
69    // end while
70    // end while
71    // end while
72    // end while
73    // end while
74    // end while
75    // end while
76    // end while
77    // end while
78    // end while
79    // end while
80    // end while
81    // end while
82    // end while
83    // end while
84    // end while
85    // end while
86    // end while
87    // end while
88    // end while
89    // end while
90    // end while
91    // end while
92    // end while
93    // end while
94    // end while
95    // end while
96    // end while
97    // end while
98    // end while
99    // end while
100   // end while
101   // end while
102   // end while
103   // end while
104   // end while
105   // end while
106   // end while
107   // end while
108   // end while
109   // end while
110   // end while
111   // end while
112   // end while
113   // end while
114   // end while
115   // end while
116   // end while
117   // end while
118   // end while
119   // end while
120   // end while
121   // end while
122   // end while
123   // end while
124   // end while
125   // end while
126   // end while
127   // end while
128   // end while
129   // end while
130   // end while
131   // end while
132   // end while
133   // end while
134   // end while
135   // end while
136   // end while
137   // end while
138   // end while
139   // end while
140   // end while
141   // end while
142   // end while
143   // end while
144   // end while
145   // end while
146   // end while
147   // end while
148   // end while
149   // end while
150   // end while
151   // end while
152   // end while
153   // end while
154   // end while
155   // end while
156   // end while
157   // end while
158   // end while
159   // end while
160   // end while
161   // end while
162   // end while
163   // end while
164   // end while
165   // end while
166   // end while
167   // end while
168   // end while
169   // end while
170   // end while
171   // end while
172   // end while
173   // end while
174   // end while
175   // end while
176   // end while
177   // end while
178   // end while
179   // end while
180   // end while
181   // end while
182   // end while
183   // end while
184   // end while
185   // end while
186   // end while
187   // end while
188   // end while
189   // end while
190   // end while
191   // end while
192   // end while
193   // end while
194   // end while
195   // end while
196   // end while
197   // end while
198   // end while
199   // end while
200   // end while
201   // end while
202   // end while
203   // end while
204   // end while
205   // end while
206   // end while
207   // end while
208   // end while
209   // end while
210   // end while
211   // end while
212   // end while
213   // end while
214   // end while
215   // end while
216   // end while
217   // end while
218   // end while
219   // end while
220   // end while
221   // end while
222   // end while
223   // end while
224   // end while
225   // end while
226   // end while
227   // end while
228   // end while
229   // end while
230   // end while
231   // end while
232   // end while
233   // end while
234   // end while
235   // end while
236   // end while
237   // end while
238   // end while
239   // end while
240   // end while
241   // end while
242   // end while
243   // end while
244   // end while
245   // end while
246   // end while
247   // end while
248   // end while
249   // end while
250   // end while
251   // end while
252   // end while
253   // end while
254   // end while
255   // end while
256   // end while
257   // end while
258   // end while
259   // end while
260   // end while
261   // end while
262   // end while
263   // end while
264   // end while
265   // end while
266   // end while
267   // end while
268   // end while
269   // end while
270   // end while
271   // end while
272   // end while
273   // end while
274   // end while
275   // end while
276   // end while
277   // end while
278   // end while
279   // end while
280   // end while
281   // end while
282   // end while
283   // end while
284   // end while
285   // end while
286   // end while
287   // end while
288   // end while
289   // end while
290   // end while
291   // end while
292   // end while
293   // end while
294   // end while
295   // end while
296   // end while
297   // end while
298   // end while
299   // end while
300   // end while
301   // end while
302   // end while
303   // end while
304   // end while
305   // end while
306   // end while
307   // end while
308   // end while
309   // end while
310   // end while
311   // end while
312   // end while
313   // end while
314   // end while
315   // end while
316   // end while
317   // end while
318   // end while
319   // end while
320   // end while
321   // end while
322   // end while
323   // end while
324   // end while
325   // end while
326   // end while
327   // end while
328   // end while
329   // end while
330   // end while
331   // end while
332   // end while
333   // end while
334   // end while
335   // end while
336   // end while
337   // end while
338   // end while
339   // end while
340   // end while
341   // end while
342   // end while
343   // end while
344   // end while
345   // end while
346   // end while
347   // end while
348   // end while
349   // end while
350   // end while
351   // end while
352   // end while
353   // end while
354   // end while
355   // end while
356   // end while
357   // end while
358   // end while
359   // end while
360   // end while
361   // end while
362   // end while
363   // end while
364   // end while
365   // end while
366   // end while
367   // end while
368   // end while
369   // end while
370   // end while
371   // end while
372   // end while
373   // end while
374   // end while
375   // end while
376   // end while
377   // end while
378   // end while
379   // end while
380   // end while
381   // end while
382   // end while
383   // end while
384   // end while
385   // end while
386   // end while
387   // end while
388   // end while
389   // end while
390   // end while
391   // end while
392   // end while
393   // end while
394   // end while
395   // end while
396   // end while
397   // end while
398   // end while
399   // end while
400   // end while
401   // end while
402   // end while
403   // end while
404   // end while
405   // end while
406   // end while
407   // end while
408   // end while
409   // end while
410   // end while
411   // end while
412   // end while
413   // end while
414   // end while
415   // end while
416   // end while
417   // end while
418   // end while
419   // end while
420   // end while
421   // end while
422   // end while
423   // end while
424   // end while
425   // end while
426   // end while
427   // end while
428   // end while
429   // end while
430   // end while
431   // end while
432   // end while
433   // end while
434   // end while
435   // end while
436   // end while
437   // end while
438   // end while
439   // end while
440   // end while
441   // end while
442   // end while
443   // end while
444   // end while
445   // end while
446   // end while
447   // end while
448   // end while
449   // end while
450   // end while
451   // end while
452   // end while
453   // end while
454   // end while
455   // end while
456   // end while
457   // end while
458   // end while
459   // end while
460   // end while
461   // end while
462   // end while
463   // end while
464   // end while
465   // end while
466   // end while
467   // end while
468   // end while
469   // end while
470   // end while
471   // end while
472   // end while
473   // end while
474   // end while
475   // end while
476   // end while
477   // end while
478   // end while
479   // end while
480   // end while
481   // end while
482   // end while
483   // end while
484   // end while
485   // end while
486   // end while
487   // end while
488   // end while
489   // end while
490   // end while
491   // end while
492   // end while
493   // end while
494   // end while
495   // end while
496   // end while
497   // end while
498   // end while
499   // end while
500   // end while
501   // end while
502   // end while
503   // end while
504   // end while
505   // end while
506   // end while
507   // end while
508   // end while
509   // end while
510   // end while
511   // end while
512   // end while
513   // end while
514   // end while
515   // end while
516   // end while
517   // end while
518   // end while
519   // end while
520   // end while
521   // end while
522   // end while
523   // end while
524   // end while
525   // end while
526   // end while
527   // end while
528   // end while
529   // end while
530   // end while
531   // end while
532   // end while
533   // end while
534   // end while
535   // end while
536   // end while
537   // end while
538   // end while
539   // end while
540   // end while
541   // end while
542   // end while
543   // end while
544   // end while
545   // end while
546   // end while
547   // end while
548   // end while
549   // end while
550   // end while
551   // end while
552   // end while
553   // end while
554   // end while
555   // end while
556   // end while
557   // end while
558   // end while
559   // end while
560   // end while
561   // end while
562   // end while
563   // end while
564   // end while
565   // end while
566   // end while
567   // end while
568   // end while
569   // end while
570   // end while
571   // end while
572   // end while
573   // end while
574   // end while
575   // end while
576   // end while
577   // end while
578   // end while
579   // end while
580   // end while
581   // end while
582   // end while
583   // end while
584   // end while
585   // end while
586   // end while
587   // end while
588   // end while
589   // end while
590   // end while
591   // end while
592   // end while
593   // end while
594   // end while
595   // end while
596   // end while
597   // end while
598   // end while
599   // end while
600   // end while
601   // end while
602   // end while
603   // end while
604   // end while
605   // end while
606   // end while
607   // end while
608   // end while
609   // end while
610   // end while
611   // end while
612   // end while
613   // end while
614   // end while
615   // end while
616   // end while
617   // end while
618   // end while
619   // end while
620   // end while
621   // end while
622   // end while
623   // end while
624   // end while
625   // end while
626   // end while
627   // end while
628   // end while
629   // end while
630   // end while
631   // end while
632   // end while
633   // end while
634   // end while
635   // end while
636   // end while
637   // end while
638   // end while
639   // end while
640   // end while
641   // end while
642   // end while
643   // end while
644   // end while
645   // end while
646   // end while
647   // end while
648   // end while
649   // end while
650   // end while
651   // end while
652   // end while
653   // end while
654   // end while
655   // end while
656   // end while
657   // end while
658   // end while
659   // end while
660   // end while
661   // end while
662   // end while
663   // end while
664   // end while
665   // end while
666   // end while
667   // end while
668   // end while
669   // end while
670   // end while
671   // end while
672   // end while
673   // end while
674   // end while
675   // end while
676   // end while
677   // end while
678   // end while
679   // end while
680   // end while
681   // end while
682   // end while
683   // end while
684   // end while
685   // end while
686   // end while
687   // end while
688   // end while
689   // end while
690   // end while
691   // end while
692   // end while
693   // end while
694   // end while
695   // end while
696   // end while
697   // end while
698   // end while
699   // end while
700   // end while
701   // end while
702   // end while
703   // end while
704   // end while
705   // end while
706   // end while
707   // end while
708   // end while
709   // end while
710   // end while
711   // end while
712   // end while
713   // end while
714   // end while
715   // end while
716   // end while
717   // end while
718   // end while
719   // end while
720   // end while
721   // end while
722   // end while
723   // end while
724   // end while
725   // end while
726   // end while
727   // end while
728   // end while
729   // end while
730   // end while
731   // end while
732   // end while
733   // end while
734   // end while
735   // end while
736   // end while
737   // end while
738   // end while
739   // end while
740   // end while
741   // end while
742   // end while
743   // end while
744   // end while
745   // end while
746   // end while
747   // end while
748   // end while
749   // end while
750   // end while
751   // end while
752   // end while
753   // end while
754   // end while
755   // end while
756   // end while
757   // end while
758   // end while
759   // end while
760   // end while
761   // end while
762   // end while
763   // end while
764   // end while
765   // end while
766   // end while
767   // end while
768   // end while
769   // end while
770   // end while
771   // end while
772   // end while
773   // end while
774   // end while
775   // end while
776   // end while
777   // end while
778   // end while
779   // end while
780   // end while
781   // end while
782   // end while
783   // end while
784   // end while
785   // end while
786   // end while
787   // end while
788   // end while
789   // end while
790   // end while
791   // end while
792   // end while
793   // end while
794   // end while
795   // end while
796   // end while
797   // end while
798   // end while
799   // end while
800   // end while
801   // end while
802   // end while
803   // end while
804   // end while
805   // end while
806   // end while
807   // end while
808   // end while
809   // end while
810   // end while
811   // end while
812   // end while
813   // end while
814   // end while
815   // end while
816   // end while
817   // end while
818   // end while
819   // end while
820   // end while
821   // end while
822   // end while
823   // end while
824   // end while
825   // end while
826   // end while
827   // end while
828   // end while
829   // end while
830   // end while
831   // end while
832   // end while
833   // end while
834   // end while
835   // end while
836   // end while
837   // end while
838   // end while
839   // end while
840   // end while
841   // end while
842   // end while
843   // end while
844   // end while
845   // end while
846   // end while
847   // end while
848   // end while
849   // end while
850   // end while
851   // end while
852   // end while
853   // end while
854   // end while
855   // end while
856   // end while
857   // end while
858   // end while
859   // end while
860   // end while
861   // end while
862   // end while
863   // end while
864   // end while
865   // end while
866   // end while
867   // end while
868   // end while
869   // end while
870   // end while
871   // end while
872   // end while
873   // end while
874   // end while
875   // end while
876   // end while
877   // end while
878   // end while
879   // end while
880   // end while
881   // end while
882   // end while
883   // end while
884   // end while
885   // end while
886   // end while
887   // end while
888   // end while
889   // end while
890   // end while
891   // end while
892   // end while
893   // end while
894   // end while
895   // end while
896   // end while
897   // end while
898   // end while
899   // end while
900   // end while
901   // end while
902   // end while
903   // end while
904   // end while
905   // end while
906   // end while
907   // end while
908   // end while
909   // end while
910   // end while
911   // end while
912   // end while
913   // end while
914   // end while
915   // end while
916   // end while
917   // end while
918   // end while
919   // end while
920   // end while
921   // end while
922   // end while
923   // end while
924   // end while
925   // end while
926   // end while
927   // end while
928   // end while
929   // end while
930   // end while
931   // end while
932   // end while
933   // end while
934   // end while
935   // end while
936   // end while
937   // end while
938   // end while
939   // end while
940   // end while
941   // end while
942   // end while
943   // end while
944   // end while
945   // end while
946   // end while
947   // end while
948   // end while
949   // end while
950   // end while
951   // end while
952   // end while
953   // end while
954   // end while
955   // end while
956   // end while
957   // end while
958   // end while
959   // end while
960   // end while
961   // end while
962   // end while
963   // end while
964   // end while
965   // end while
966   // end while
967   // end while
968   // end while
969   // end while
970   // end while
971   // end while
972   // end while
973   // end while
974   // end while
975   // end while
976   // end while
977   // end while
978   // end while
979   // end while
980   // end while
981   // end while
982   // end while
983   // end while
984   // end while
985   // end while
986   // end while
987   // end while
988   // end while
989   // end while
990   // end while
991   // end while
992   // end while
993   // end while
994   // end while
995   // end while
996   // end while
997   // end while
998   // end while
999   // end while
1000  // end while
```

Block Read/Write – Example 2/5

```
36 file = fopen(fname, mode);
37 if (!file) {
38     fprintf(stderr, "ERROR: Cannot open file '%s', error %d - %s\n", fname, errno,
39             strerror(errno));
40     return -1;
41 }
42 int *data = (int*)malloc(NUMB * sizeof(int));
43 my_assert(data __LINE__, __FILE__);
44 struct timeval t1, t2;
45 gettimeofday(&t1, NULL);
46 if (read) {
47     fprintf(stderr, "INFO: Read from the file '%s'\n", fname);
48     c = fread(data, sizeof(int), NUMB, file);
49     if (c != NUMB) {
50         fprintf(stderr, "WARN: Read only %i objects (int)\n", c);
51     } else {
52         fprintf(stderr, "DEBUG: Read %i objects (int)\n", c);
53     }
54 } else {
55     char buffer[BUFSIZE];
56     if (setvbuf(file, buffer, _IOFBF, BUFSIZE)) { /* SET BUFFER */
57         fprintf(stderr, "WARN: Cannot set buffer");
58     }
59 }
60 }
```

Block Read/Write – Example 3/5

```
58     fprintf(stderr, "INFO: Write to the file '%s'\n", fname);
59     c = fwrite(data, sizeof(int), NUMB, file);
60     if (c != NUMB) {
61         fprintf(stderr, "WARN: Write only %i objects (int)\n", c);
62     } else {
63         fprintf(stderr, "DEBUG: Write %i objects (int)\n", c);
64     }
65     fflush(file);
66 }
67
68 gettimeofday(&t2, NULL);
69 double dt = t2.tv_sec - t1.tv_sec + ((t2.tv_usec - t1.tv_usec) / 1000000.0);
70 double mb = (sizeof(int) * c) / (1024 * 1024);
71 fprintf(stderr, "DEBUG: feof: %i\n", feof(file), ferror(file));
72 fprintf(stderr, "INFO: %s %lu MB\n", (read ? "read" : "write"), sizeof(int)*NUMB
73         / (1024 * 1024));
74 fprintf(stderr, "INFO: %.2lf MB/sec\n", mb / dt);
75 free(data);
76 return EXIT_SUCCESS;
```

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
```

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
```

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 transferred 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()`.

```
#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.

```
#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;
```

Key Press without Enter – Example

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

```
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);
    }
}
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 interpreter.
Returns the program (shell) exit status.
Returns 127 is the shell execution failed.
```

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 errors.
- <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 quadrant 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()`, ... – comparison 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 know how many values we can retrieve 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 File 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'
```

Standard library – Selected Functions Error Handling

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.

```
#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

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 45 / 69

Standard library – Selected Functions Error Handling

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!

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 46 / 69

Standard library – Selected Functions Error Handling

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`.

```
1 #include <setjmp.h>
2 jmp_buf jb;
3 int compute(int x, int y);
4 void error_handler(void);
5 if (setjmp(jb) == 0) {
6     r = compute(x, y);
7     return 0;
8 } else {
9     error_handler();
10    return -1;
11 }
12 int compute(int x, int y) {
13     if (y == 0) {
14         longjmp(jb, 1);
15     } else {
16         x = (x + y * 2);
17         return (x / y);
18     }
19 }
20 void error_handler(void) {
21     printf("Error\n");
22 }
```

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 47 / 69

Standard library – Selected Functions Error Handling

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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     printf("USER: %s\n", getenv("USER"));
7     printf("HOME: %s\n", getenv("HOME"));
8     return EXIT_SUCCESS;
9 }
```

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.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 48 / 69

Standard library – Selected Functions Error Handling

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 void cleanup(void);
5 void last_word(void);
6 int main(void)
7 {
8     atexit(cleanup); // register function
9     atexit(last_word); // register function
10    const char *howToExit = getenv("HOW_TO_EXIT");
11    if (howToExit && strcmp(howToExit, "EXIT") == 0) {
12        printf("Force exit!\n");
13        exit(EXIT_FAILURE);
14    } else if (howToExit && strcmp(howToExit, "ABORT") == 0) {
15        printf("Force abort!\n");
16        abort();
17    }
18    printf("Normal exit!\n");
19    return EXIT_SUCCESS;
20 }
21 void cleanup(void)
22 {
23     printf("Perform cleanup at the program exit!\n");
24 }
25 void last_word(void)
26 {
27     printf("Bye, bye!\n");
28 }
```

- 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

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 49 / 69

Organization of Source Files Preprocessor Building Programs

Part III

Preprocessor and Building Programs

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 50 / 69

Organization of Source Files Preprocessor 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 declare **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.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 52 / 69

Organization of Source Files Preprocessor Building Programs

Organizing C Program

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

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 53 / 69

Organization of Source Files Preprocessor Building Programs

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.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 54 / 69

Organization of Source Files Preprocessor Building Programs

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 {
    edget_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) {
        ...
    }
}
```

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 55 / 69

Organization of Source Files Preprocessor Building Programs

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
```

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 56 / 69

Organization of Source Files Preprocessor Building Programs

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 "dstringing" the string literals and pass them to `#pragma` operator.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 58 / 69

Organization of Source Files Preprocessor Building Programs

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.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 59 / 69

Organization of Source Files Preprocessor Building Programs

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 -DDEBUG main.c` – define `DEBUG` 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.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 60 / 69

Organization of Source Files Preprocessor Building Programs

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 `-llib`, 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`.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 62 / 69

Organization of Source Files Preprocessor Building Programs

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.

<pre>target : dependencies action</pre>	<pre>colon tabulator</pre>
---	----------------------------
- 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 usage 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.

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 63 / 69

Organization of Source Files Preprocessor Building Programs

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 $(CC)
CFLAGS+=-O2
OBSJS=$(patsubst %.c,%.o,$(wildcard *.c))
TARGET=program
bin: $(TARGET)
$(OBSJS): %.o: %.c
$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@
$(TARGET): $(OBSJS)
$(CC) $(OBSJS) $(LDFLAGS) -o $@
clean:
$(RM) $(OBSJS) $(TARGET)
CC=clang make vs CC=gcc make
```
- The order of the files is important during the linking!

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 64 / 69

Part IV

Part 3 – Assignment HW 04 and HW 06

Jan Faigl, 2024 PRG – Lecture 06: I/O and Standard Library 65 / 69

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 `stdout`; 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**