# Programming basics
## (GKNB_INTA023)

Hatwagner F. Miklós, PhD.

Széchenyi István University, Győr, Hungary

https://github.com/sze-info/ProgrammingBasics

November 27, 2020

# Command line arguments

At the execution of a program in command line space separated *arguments* can be passed to the program to influence its behavior.

## Listing directory contents – succint format

```
wajzy@lenovo:~/Dokumentumok/gknb_inta023/ProgrammingBasics/lectures/lecture10$ ls
cities1.c      lecture10.log        lecture10.tex    operation1.c  roster1.pdf  sagrada.jpg
cities2.c      lecture10.nav        lecture10.toc    operation2.c  roster1.svg
cities3.c      lecture10.out        lecture10.vrb    operation3.c  roster2.c
```

## Listing directory contents – long format

```
wajzy@lenovo:~/Dokumentumok/gknb_inta023/ProgrammingBasics/lectures/lecture10$ ls -l
összesen 1484
-rw-rw-r-- 1 wajzy wajzy   1455 nov   21 18:09 cities1.c
-rw-rw-r-- 1 wajzy wajzy   1507 nov   21 18:09 cities2.c
-rw-rw-r-- 1 wajzy wajzy   1403 nov   21 18:09 cities3.c
```

# Command line arguments

Task: write a program that lists its arguments.

### Output 1

```
wajzy@lenovo:~/Dokumentumok/gknb_inta023/ProgrammingBasics/lectures/lecture13$
./args1 one two three "contains space"
Name of the started program: ./args1
Arg. #1: one
Arg. #2: two
Arg. #3: three
Arg. #4: contains space
```

### Output 2

```
wajzy@lenovo:~/Dokumentumok/gknb_inta023/ProgrammingBasics/lectures/lecture13$ ./args1
Name of the started program: ./args1
No command line arguments were given.
```

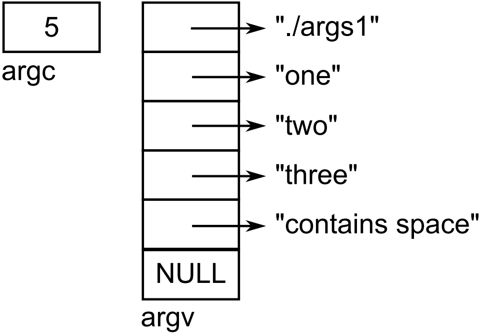
# Command line arguments

## Formal parameters of `main`

```
int main(int argc, char* argv[]) { /* ... */ }
int main(int argc, char** argv) { /* ... */ }
```

argc
: The number of command line arguments (argument count), including the name of the program

argv
: Array of pointers to strings (argument vector)



# Command line arguments

args1.c

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  printf("Name of the started program: %s\n", argv[0]);
  if(argc == 1) {
    printf("No command line arguments were given.\n");
  } else {
    for(int i=1; i<argc; i++) {
      printf("Arg. #%d: %s\n", i, argv[i]);
    }
  }
  return 0;
}
```

# Command line arguments

args2.c

```c
#include <stdio.h>

int main(int argc, char** argv) {
  printf("Name of the started program: %s\n", *argv);
  if(argc == 1) {
    printf("No command line arguments were given.\n");
  } else {
    for(argv++; *argv != NULL; argv++) {
      printf("%s\n", *argv);
    }
  }
  return 0;
}
```

# Command line arguments

Task

Generating a random number in an interval specified by its bounds

Problem

Type of command line arguments is not apropriate (strings instead of integers)

Solution

Using the `atoi` function (ASCII to int)

# Command line arguments

## args3.c

```c
#include <stdio.h>
#include <stdlib.h> // srand, rand, atoi
#include <time.h>   // time

int main(int argc, char* argv[]) {
  if(argc != 3) {
    printf("Usage: %s min max\n", argv[0]);
  } else {
    int min = atoi(argv[1]);
    int max = atoi(argv[2]);
    srand(time(NULL));
    printf("A random-generated number in the [%d, %d] interval: %d\n",
      min, max, min + rand()%(max-min+1));
  }
  return 0;
}
```

# High-level input and output

**Task**

Write a program that prints the content of a text file specified on the command line

**Solution**

Usage of high-level I/O (operating system independent, portable solution but only the most important services can be used). Different sources and destinations of data series (eg. files, keyboard, screen, printer) are handled in the same way, with *streams*.

Steps of file handling:

1. Opening the stream (of appropriate type)
2. Performing i/o operations
3. Closing the stream

The streams `stdin` (standard input), `stdout` (standard output) and `stderr` (standard error) are automatically opened and closed

# High-level input and output

## read1.c – Reading a file character by character

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  if(argc != 2) {
    printf("Usage: %s filename\n", argv[0]);
  } else {
    FILE* f = fopen(argv[1], "rt");
    if(f) {
      int c;
      while((c=fgetc(f)) != EOF) putchar(c);
      fclose(f);
    } else {
      fprintf(stderr, "File opening error.\n");
    }
  }
  return 0;
}
```

# High-level input and output

## Opening a stream

```
FILE *fopen(const char *pathname, const char *mode);
```

`FILE*`
: Address of a structure containing the data of a stream, or `NULL` in case of error

`pathname`
: Name of the file (the maximum length is `FILENAME_MAX` characters)

`mode`
: Opening mode (flags that can be combined)

| Mode flag | Effect |
|---|---|
| r | reading from the beginning of the stream |
| w | overwriting (deleting of the existing file) then writing from the beginning of the stream |
| a | appending (writing at the end of the stream). Creates the stream if it does not exist. |
| r+ | renew (update): reading and writing from the beginning of the stream |
| w+ | deleting an existing file then opening for renewing at the beginning |
| a+ | reading from the beginning and appending. Creates the stream if it does not exist. |

# High-level input and output

- Further possible characters of `mode`: `t` (text) and `b` (binary)
- Text mode: *translation* on Microsoft OS-es (CR-LF ↔ LF), file end character (0x1A)
- Eg.: `"rt+"`, `"r+t"` (`t` and `b` can be anywhere after the first character, default: `t`)
- Before changing the direction of data flow (input ↔ output) a positioning function, eg. `fseek()` must be called

# High-level input and output

## Reading one character

```
int fgetc(FILE *stream);
```

Return value
: The read character or `EOF` in case of reaching the end of file or error

`stream`
: Stream identifier

## Closing the stream

```
int fclose(FILE *stream);
```

Return value
: 0 or `EOF` in case of an error

`stream`
: Stream identifier

## Formatted printing in a file

```
int fprintf(FILE *stream, const char *format, ...);
```

`stream`
: Stream identifier (`printf` always prints to `stdout`)

# High-level input and output

## The output of our program

```
wajzy@lenovo:~/Dokumentumok/gknb_inta023/ProgrammingBasics/lectures/lecture13$
./read1 guns.txt
She's got a smile it seems to me
Reminds me of childhood memories
Where everything
Was as fresh as the bright blue sky
Now and then when I see her face
```

## The output of cat (concatenate files and print)

```
wajzy@lenovo:~/Dokumentumok/gknb_inta023/ProgrammingBasics/lectures/lecture13$
cat guns.txt
She's got a smile it seems to me
Reminds me of childhood memories
Where everything
Was as fresh as the bright blue sky
Now and then when I see her face
```

# High-level input and output

read2.c – Every word in separate strings

```c
#include <stdio.h>
#define MAX 256

int main(int argc, char* argv[]) {
  if(argc != 2) {
    printf("Usage: %s filename\n", argv[0]);
  } else {
    FILE* f = fopen(argv[1], "rt");
    if(f) {
      char buf[MAX];
      while(fscanf(f, "%s", buf) != EOF)
        printf("%s\n", buf);
      fclose(f);
    } else {
      fprintf(stderr, "File opening error.\n");
    }
  }
  return 0;
}
```

Output

```
She's
got
a
smile
it
seems
to
me
Reminds
me
of
childhood
memories
Where
everything
Was
as
fresh
as
```

# High-level input and output

Formatted reading (scanning)

```
int fscanf(FILE *stream, const char *format, ...);
```

Return value

The number of read, converted and stored elements or `EOF` in case of reaching the end of file or error

`stream`

Stream identifier

# High-level input and output

read3.c – Reading whole lines

```c
#include <stdio.h>
#define MAX 256

int main(int argc, char* argv[]) {
  if(argc != 2) {
    printf("Usage: %s filename\n", argv[0]);
  } else {
    FILE* f = fopen(argv[1], "rt");
    if(f) {
      char buf[MAX];
      while(fgets(buf, MAX, f)) {
        printf("%s", buf);
      }
      fclose(f);
    } else {
      fprintf(stderr, "File opening error.\n");
    }
  }
  return 0;
}
```

# High-level input and output

## Reading whole lines

```
char *fgets(char *s, int size, FILE *stream);
```

**Return value**
The address of the character buffer (`s`) or `NULL` in case of reaching the end of file or error

**`s`**
The address of the buffer. `fgets` stores at most `size-1` characters. The terminating null character is always included and even the new line character, if the line contained it

**`size`**
Size of the buffer

**`stream`**
Stream identifier

# High-level input and output

## write1.c – Writing lines

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  if(argc != 2) {
    printf("Usage: %s filename\n", argv[0]);
  } else {
    FILE* f = fopen(argv[1], "wt");
    if(f) {
      char* song[] = {
        "She's got a smile it seems to me",
        "Reminds me of childhood memories",
```

# High-level input and output

## write1.c – Writing lines

```c
        "And if I'd stare too long",
        "I'd probably break down and cry"
      };
      for(unsigned i=0; i<sizeof(song)/sizeof(song[0]); i++) {
        fprintf(f, "%s\n", song[i]);
      }
      fclose(f);
    } else {
      fprintf(stderr, "File opening error.\n");
    }
  }
  return 0;
}
```

# High-level input and output

### copy1.c – Copy file character by character

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  if(argc != 3) {
    printf("Usage: %s source destination\n", argv[0]);
  } else {
    FILE* in = fopen(argv[1], "rt");
    if(in) {
      FILE* out = fopen(argv[2], "wt");
      if(out) {
        int c;
        while((c=fgetc(in)) != EOF) {
          fputc(c, out);
        }
```

# High-level input and output

## copy1.c – Copy file character by character

```c
        fclose(out);
      } else {
        fprintf(stderr, "Opening error: %s\n", argv[2]);
      }
      fclose(in);
    } else {
      fprintf(stderr, "Opening error: %s\n", argv[1]);
    }
  }
  return 0;
}
```

# High-level input and output

## copy2.c – Copy by block

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 65536

int main(int argc, char* argv[]) {
  if(argc != 3) {
    printf("Usage: %s source destination\n", argv[0]);
  } else {
    FILE* in = fopen(argv[1], "rb");
    if(in) {
      FILE* out = fopen(argv[2], "wb");
      if(out) {
        char* buffer = (char*)malloc(SIZE);
        int amount;
```

# High-level input and output

copy2.c – Copy by block

```c
        do {
          amount = fread(buffer, 1, SIZE, in);
          fwrite(buffer, 1, amount, out);
        } while(amount == SIZE);
        free(buffer);
        fclose(out);
      } else {
        fprintf(stderr, "File opening error: %s\n", argv[2]);
      }
      fclose(in);
    } else {
      fprintf(stderr, "File opening error: %s\n", argv[1]);
    }
  }
  return 0;
}
```

# High-level input and output

Reading blocks

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

Writing blocks

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

Return value
: The number of read/written blocks

`ptr`
: Address of buffer (its size must be at least `size*nmemb`)

`size`
: Size of a block

`nmemb`
: Number of blocks to be processed at once

`stream`
: Stream identifier

# High-level input and output

## Copy by character

```
time ./copy1 bigfile.dat copy.dat
real 5m33.072s
user 4m54.301s
sys 0m19.405s
```

## Copy by block

```
time ./copy2 bigfile.dat copy.dat
real 0m53.501s
user 0m0.212s
sys 0m12.795s
```

## Copy with OS utility

```
time cp bigfile.dat copy.dat
real 0m50.821s
user 0m0.102s
sys 0m12.247s
```

# Bit-level operators

In order of priority:

| | |
|---|---|
| ~ | one's complement |
| << | shift to left |
| >> | shift to right |
| & | and |
| ^ | exclusive or |
| \| | (permissive) or |

They can be used only with integer data!

# One's complement

- performs integer promotion, if needed
- the type of the result is the converted type

not.c – Example (assuming 32 bit `int`s)

```c
#include<stdio.h>

int main(void) {
  unsigned char c = 0xA; /* 1010 */
  printf("%x\n",~c); /* output: fffffff5
    thus 1111 1111 1111 1111 1111 1111 1111 0101 */
  return 0;
}
```

# Shift

- *op1*<<*op2* and *op1*>>*op2*
- Shifting the bits of *op1* by *op2* positions (0 bits come in from the right, 0 (`unsigned`) or the value of the bit indicating the sign (`signed`) come from the left according to the sign handling of *op1*)
- Operands are integers
- Performs integer promotion if needed
- The type of the result is the type of the converted type of *op1*
- If *op2*<0 or *op2*≥the size of *op1* in bits → undefined result
- If it does not cause overrun then $op1 \ll op2 \equiv op1 * 2^{op2}$
- The integer part of $op1/2^{op2} \equiv op1 \gg op2$

# Shift

shift.c – Example (assuming 32 bit ints)

```c
#include<stdio.h>

int main(void) {
  signed char c = (signed char)0xAA; /* 1010 1010 */
  printf("%x\n",c>>4); /* output: fffffffa */
  return 0;
}
```

# Rotate

### rotate.c – Example (assuming 32 bit ints)

```c
#include <stdio.h>
#include <limits.h>
#define WORDLENGTH sizeof(unsigned)*8

unsigned rotl(unsigned num) {
  return(num<<1 | num>>(WORDLENGTH-1));
}

unsigned multirotl(unsigned num, unsigned n) {
  return(num<<n | num>>(WORDLENGTH-n));
}

void print(unsigned n){
  unsigned i, mask = INT_MIN;
  for(i=0; i<WORDLENGTH; i++, n<<=1) {
    if(n & mask) putchar('1');
    else putchar('0');
  }
}
```

# Rotate

rotate.c – Example (assuming 32 bit ints)

```c
int main(void) {
  unsigned num = -(INT_MAX);
  printf("Original number:\t\t");
    print(num);
  printf("\nRotated to the left by 1 bit:\t");
    print(rotl(num));
  printf("\nRotated to the left by 4 bits:\t");
    print(multirotl(num, 4));
  return 0;
}
```

Output

```
Original number:                10000000000000000000000000000001
Rotated to the left by 1 bit:   00000000000000000000000000000011
Rotated to the left by 4 bits:  00000000000000000000000000011000
```

# Bit-level and (&), exclusive or (^), or (|)

- Implicit type conversion is performed, if needed
- Result is in the converted type
- Watch out! If a==1 and b==2 then a&&b==1, but a&b==0

| *op1* | *op2* | *op1&op2* | *op1^op2* | *op1\|op2* |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |

### Clearing bits

```
 0111 1110
&1100 0011
 ---------
 0100 0010
```

### Zeroing

```
 0101 0101
^0101 0101
 ---------
 0000 0000
```

### Setting bits

```
 0011 1100
|1100 0101
 ---------
 1111 1101
```