Programming and modelling

Part - Programming

Lecture 5

Recapitulate

- What is higher (lower) level programming language?
  - examples
    - *low level*: assembler
    - *high level:* Pascal, C, PHP, Java, JavaScript
  - compiler language
    - Cobol, Fortran, Pascal, C/C++, Ada
  - interpreted language
    - JavaScript, PHP, Python, Basic

Recapitulate

- What does it mean?
  - compile, compiler, linker
  - object files (.o, .obj)
  - libraries (.lib, .a)
  - syntax, semantics
  - lexical elements of the language
  - lexical and syntactic analysis

C Programming Language

- 70th - AT&T Bell Laboratories
- authors: Kernighan a Ritchie
  - the first specification: K&R 77 (you can see in some literature K&R 78)
- strongly tied to UNIX
- 1988 – ANSI C standard
  - ANSI 99
  - ANSI C1X (2011)

How is Compiling in C processed?

- header files .h (.hpp)
  - files contain prototypes (headers of functions)
- preprocessor
  - it makes text modifications of the source code (replaces symbols, includes header files, removes comments, …)
  - directives of preprocessor – starts with #, they are not terminated by ;
  - example:

```
#define MAX 10
#include <stdio.h>
```

How is Compiling in C processed?

Note:

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

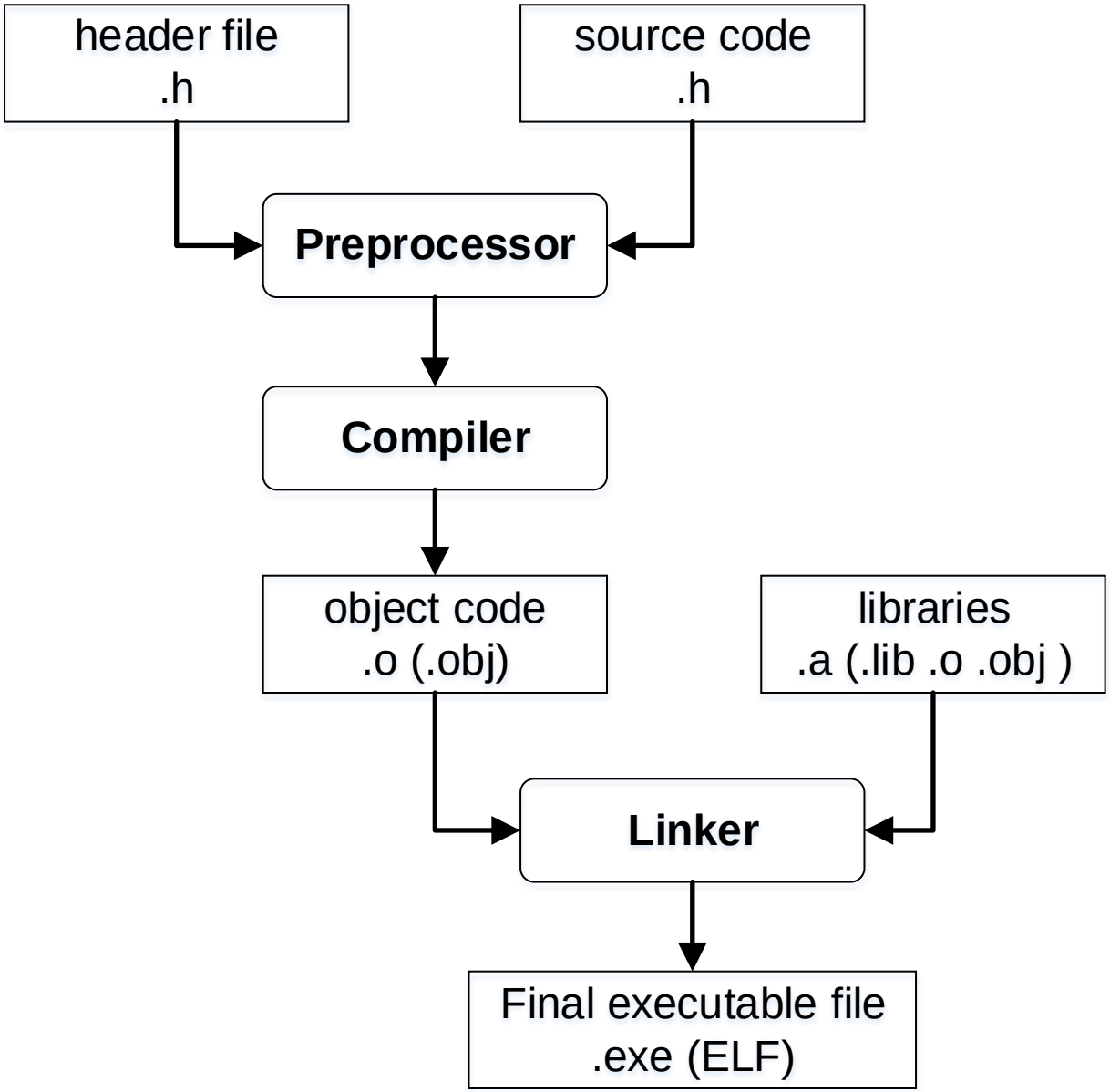
- the preprocessor searches `stdio.h` file in standard subdirectories of the installed compiler (typically subdirectory `include`)

```
#include "myfile.h"
```

- the preprocessor searches `myfile.h` file in the actual directory (typically where the project is stored)

- the path can be written:

```
#include "incl/soubor.h„
#include <ole/access.h>
```


header file
.h
source code
.h
Preprocessor
Compiler
object code
.o (.obj)
libraries
.a (.lib .o .obj)
Linker
Final executable file
.exe (ELF)

- we can realize *conditional assembly* using directive `#define:`

```
#define DEBUG

#ifdef DEBUG
  printf("Test print: ");
  printf("x = %X",x);
#endif
```

- we can realize *conditional assembly* using directive `#define:`

```
#define DEBUG


#ifdef DEBUG
  printf("Test print: ");
  printf("x = %X",x);
#else
 printf("x = %d",x);
#endif
```

C++ Programming Language

- 1983 - AT&T Bell Laboratories

  Bjarne Stroustrup - „C with objects“
- 1986 Bjarne Stroustrup

  The C++ Programming Language
- 1990 Bjarne Stroustrup:

  The Annotated C++ Reference Manual

- 1995 M.Ellis, B.Stroustrup
  The Annotated C++ Reference Manual
- 1997 - ANSI/ISO standard of C++
  - ISO/IEC 14882:1998
  - standard notated as C++98
- 2003 C++03
- 2011 C++11
- 2014 C++14
- 2017 C++17
- 2020 C++20
- 2023* C++23

*exactly 12/2022

- C++ language is extended C (object oriented programming is added), but this sentence *is not exact:*
  - C is not a subset of C++, some "non-object" features are different from original C standard (especially K&R)

- **nevertheless** most programs in C is possible to compile with C++ compiler, namely if ANSI C standard is complied

Recapitulation of C

- weak typing (in C++ stronger)
- case sensitive
- features of functional language
- block marked by { }
- one function in the code must have the the identifier `main`
  - `void main()`
  - `int main(int argc, char* argv[])`

Note: identifier = name of the function, variable

Literals (lexical elements):

- *constants*
  - decimal constants: 13, -5, 15L
  - octal constants: 056
  - hexadecimal constants: 0x20, 0X1F
  - floating point constants (double): 5.3, -4E-3
  - characters: `'a','\n','\t','0x0A'`
  - string constants: `"Hello, world!"`
- *variable identifiers*
  - they must start with letter, the accepted length depends on compiler (usually 31), C++ usually unlimited

Data types – "simple"

- *integers*
  - char (8 bits)
  - short, int, long (unsigned, signed), long long
  - sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)
- *floating point*
  - float, double, long double (ANSI)
  - some compilers: long long double

Typical sizes of types

Type	Size [bit]	Range
`char`	8	-128 to +127
`short`	16	-32 768 to +32 767
`int`	32	-2 147 483 648 to +2 147 48 3647
`long`	32 (64)	-2 147 483 648 to +2 147 48 3647
`long long`	64	-9 223 372 036 854 775 808 to +9 223 372 036 854 775 807
`float`	32	$-3{,}402823 \cdot 10^{+38}$ to $+3{,}402823 \cdot 10^{+38}$ the lowest positive number $1{,}175494 \cdot 10^{-38}$ valid digits 7 or 8
`double`	64	-1, $7976931348623157 \cdot 10^{+308}$ to +1, $7976931348623157 \cdot 103^{+308}$ the lowest positive number $2{,}225073858507202 \cdot 10^{-308}$ valid digits 15 to 16
`long double`	96 (80) 128 in 64-bit compiler	$-3{,}4 \cdot 10^{+4932}$ to $+3{,}4 \cdot 10^{+4932}$ the lowest positive number $1{,}1 \cdot 10^{-4932}$ valid digits 19

Typical sizes of types

- **standard of C does not define the sizes of types, it depends on compiler**
- **the following inequality must be satisfied:**

`sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)`

- ranges are defines as symbolic constants (macros) in `limits.h`, resp. `climits` and `float.h`, resp. `cfloat`
  - `INT_MAX, INT_MIN, LONG_MAX, ...`
  - `FLT_MAX, DBL_MAX, LDBL_MIN, ...`
- extended definitions of int data types are defined in `stdint.h`

- boolean
  - boolean type is not defined in C, it is replaced by **`int` type**
    - whatever non-zero value means **true**
  - type **`bool`** is defined in C++ with values **`true, false`**

```
bool is_open = true;
if (is_open) …
if (is_open == false) …
```

- string
  - string type is not defined in C, string variables are represented by arrays of char (char*)
  - `string` type is defined in C++

Comments

- multi-line comments in C

```
/* This is a multi – line
   comment
*/
```

- single line comments in C ++

```
// This is a single line comments
```

Variables:

- global
- local (in block)
- variable declaration:

```
type identifier_of_var;
int x=10; // initialization
char c;
float radius, area;
```

```
/* The variable "selection" is global,
   visible in both functions area and main*/

int selection;
int area()
{/* The variables x and y are local in this
  function and they are visible only here*/
  float x, y;
}
int main(int argc, char **argv)
{
/* The variable a is local in main and it is
  visible  only in main */
  int a;
}
```

Where can be variables declared?

- C
  - only at the beginning of blocks, before the first command
- C++
  - everywhere in block

Notes to declaration

```
int main()
{  int i;
   …
   for(i=0;i<10;i++)
   {  int x;
      …
   }
   /* only i exists in this place,
   not x */
}
```

Overlapping of declarations

```
int main()
{  int x;
   …
   for(i=0;i<10;i++)
   { int x; /* this declaration overlaps
   (shadows) previous declaration */
      …
   }
   /* "original" x is visible */
}
```

New user type definition

- with keyword **`typedef`**
- example: definition of enumerate data type
  - next week - structures

Problem:

- we want to store in our program information about colors, for example colors of car
- the source code must be readable, clear, modificable

Solution

1. using constants directly

```
int color;
color = 0;
if (color == 0)
```

   – I make a note on paper: 0 means black, 1 means red, …
     - the worst case
   – better: I put this information as comment at the beginning of the source code

```
/* 0 – black, 1 – red */
```

Solution

2. definition of symbolic constants (macros) in the source code, variable is **`int`**

```
#define BLACK 0
#define RED 1
int color;
color = RED;
if (color == RED)
```

- clear solution

3. definition of user data enum type

- the definition of user type in the beginning of the source code or in separate header file

```
typedef enum
{BLACK, RED, WHITE} Colors;
```

- the variable of the type `Colors` can be declared

```
Colors color;
color = WHITE;
if (color == RED)
```

- how is enumerate type implemented internally?

```
typedef enum
{BLACK, RED, WHITE} Colors;
```

  - using integers, it means `BLACK` by 0, `RED` by 1, `WHITE` by 2

  - internal values can defined by the programmer

```
typedef enum {BLACK, RED, WHITE
=6} Color;
```

- WARNING

```
typedef enum {BLACK, RED, WHITE,
  BLUE=1} Colors;
```

  - `BLACK` by the value 0, `RED` by the value 1,
    `WHITE` by the value 2, `BLUE` by the value 1
- thank to weak typing, the compile passes

```
color = 4;
```

- compileable in C
- compile error in C++ (invalid conversion from 'int' to ‘Colors')

Definitions of Constants

- in K&R: only symbolic ones (macros)

```
#define MAX 10
```

- in ANSI C a C++: keyword **`const`**
  - "variable" with allocated memory
  - it is not possible to use „everywhere“ (size of arrays in C)

```
const int MAX = 10;
```

Assignment:

- assignment is defined as expression in C:

```
y = 3*a + 12    i=j=1
```

  – consequences will be explained

- assignment command

  – expression terminated by ; is a command

```
y = 3*a + 12; a = 10; c = 'a';
  car_color = WHITE;
```

Operations:

- arithmetical operators: +, - ,* , /
  - % - modulo
  - warning: division – one operator for floor (integer) division and "normal" division
    - it depends on operands
    - `5/3, 5/3.0`
    - `int a,b,n; a/b, (double) a/b`
    - `1/n  1.0/n`
  - there is no operator for power in C/C++, only function `double pow(double b,double exp);`
- bitwise operators: &, |, ~, ^

Bitwise operators

Operator	Operation
&	Bitwise AND
|	Bitwise OR
^	eXclusive bitwise OR- XOR
~	Bitwise negation (complement)
<<	Bitwise left shift
>>	Bitwise right shift

```
unsigned char a = 0x85;
/* 133 dec, 10000101 bin */
unsigned char b = 0x46;
/* 70 dec, 01000110 bin */
unsigned char c,d,e,f,g,h;
c = a & b;
d = a | b;
e = a ^ b;
f = ~ a;
g = a << 2; // 2 bits left shift
h = b >> 3; // 3 bits right shift
```

Results of Bitwise Operations

10000101	10000101	10000101	10000101
&	|	^	~
01000110	01000110	01000110	
00000100	11000111	11000011	01111010

Results of Bitwise Operations

```
10000101   01000110

  << 2       >> 3
```

```
00010100   00001000
```

What Good is it ?

- example: some function returns 8-bits value (unsigned char) and each bit carries some piece of information, for example several type of error; return value is stored in the variable `error`

```
00010100
```

- we need to find out, if this bit is set
- we use constant called mask:

```
00000100
```

```
00000100 bin = 04 hex
```

- `if (error & 0x04 != 0)`

- unary increment, decrement (post- and pre-form (prefix, postfix forms): `++, --`

```
a++; ++a; a--; --a;
y = 3*a++;  z = 5/--a;
```

- assignment operator `op=`

`y *= 3;` is the same like `y = y*3;`

`y += 3;` is the same like `y = y+3;`

```
y >>= 2;
```

Commands

- each command terminated by `;`
- condition (branch):

```
if (a < 5) error();
if (x != 0) compute(); else err();
```

- relational operators:

  `<, <=, >, >=, !=, ==`

- logical operators:

  `&&, ||, !`

  – lower priority than arithmetical ones

Attention:

```
if (a == 5) ...
```
comparison

```
if (a = 5)  ...
```
assignment

```
int c;
c = getchar();
if (c=='\n') ...
if ((c=getchar())=='\n') ...
```

```
if (a) ...
```
comparison with 0

the same as `if (a != 0) ...`

- **loops:**

```
while(expression) command;

do command; while(expression);

do { command1; command2; }
while(expression);
```

- **loops:**

```
for(expr1;expr2;expr3) command;
```

equivalent with:

```
expr1;
while(expr2)
{ command;
  expr3;
}
```

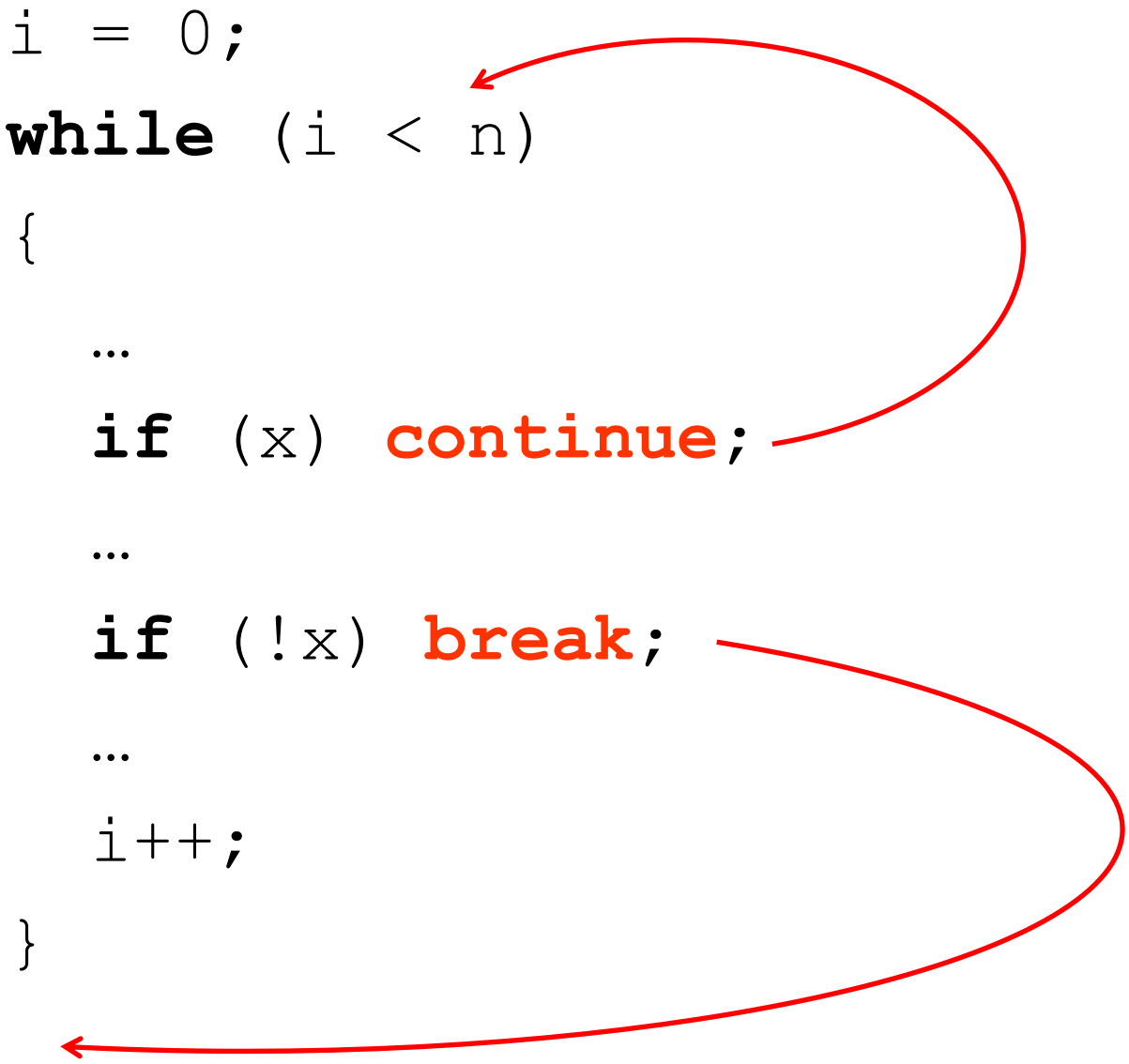
– most frequently used:

```
for(i=1;i<n;i++)
```

```
for(i=1;i<n;i=i+2)
```

- semicolon !!!
- what about "comma" operator ?

- **`continue`** and **`break`** in loops
  - `break` finish the loop immediately
  - `continue` finish actual execution of loop body and jumps to condition


i = 0;
while (i < n)
{
…
if (x) continue;
…
if (!x) break;
…
i++;
}

We create a program which reads from standard input (keyboard) the sequence of English characters terminated by Enter. It changes all upper characters to lower ones and it prints the text.

```
int main(int argc, char **argv)
{
  int c;
  c = getchar();
  while(c!='\n')
  {
    if (c >='A' && c <= 'Z') c+=('a'-'A');
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

- more effectively

```
int main(int argc, char **argv)
{
  int c;
  while((c = getchar())!='\n')
  {
    if (c >='A' && c <= 'Z') c+=('a'-'A');
    putchar(c);
  }
  return 0;
}
```

Conditional expression

- conditional expression: *ternary operator* (uses three expressions)
- syntax:

```
expr1 ? expr2 : expr3
```

  - the result of conditional expression is `expr2`, if `expr1` is non-zero (is true); otherwise, the result is `expr3`

-
```
(a==3) ? 1 : 0
// if a is equal to 3, the value of conditional expression is 1, otherwise 0
```

Conditional expression

- using in assignment:

- ```
  x = (a==3) ? 1 : 0
  // if a is equal to 3, the value of 1
  is assigned to x, otherwise 0
  ```

Notes:

- naturally, we can write this assignment using if:
  `if (a==3) x = 1; else x = 0;`
- it is not necessary to put the first expression in ( ), but programmers do it for clarity of the code
- arbitrary condition can not be replaced by conditional expression!
  ```

- version of the program with conditional expression

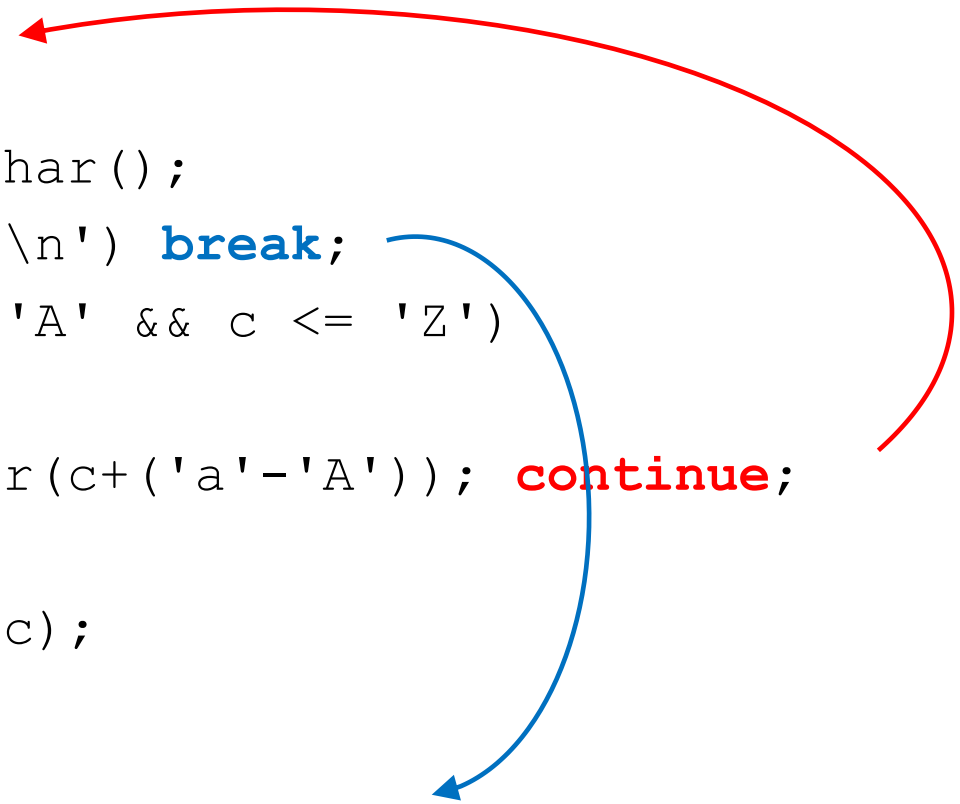
```
int main(int argc, char **argv)
{
  int c;

  while((c = getchar())!='\n')
  {
    putchar((c >='A' && c <= 'Z') ? c+('a'-'A'):c);
  }
  return 0;
}
```

- demonstration of **break** a **continue** (it is not transparent code)

```
int main(int argc, char **argv)
{
  int c;

  while(1)
  {
    c = getchar();
    if (c=='\n') break;
    if (c >='A' && c <= 'Z')
    {
      putchar(c+('a'-'A')); continue;
    }
    putchar(c);
  }
  return 0;
}
```



Operator "comma"

- "speciality" of C
- single expressions are separated in "multi-expression" (this allows to put several expressions where only one expression is allowed in C)

```
expr1, expr2, expr3
```

- these expressions are considered as one expression
- meaning:
  - expr1 is evaluated firstly, then expr2 and finally expr3
  - the final value of "multi-expression" is the value of the last expression

Operator "comma"

Example:

```
x = a=1, 3, a+b;
```

- firstly, assignment a=1 is evaluated, a is set to 1
- secondly, the expression 3 (constant is evaluated), the result is 3, but it is "forgotten"
- finally, `a` and `b` are added; because od a+b is the last expression in the "multi-expression", the result of the "multi-expression" is this addition and it assigned to variable x

Operator "comma"

Operator is practically used only in `for` loop:

```
s=0;
for(i=1;i<=n;i++) s += i;
```

```
for(s=0,i=1;i<=n;i++) s += i;
```

Example – operator “comma“

We create a program that reads one integer from the standard input and stores it to the x variable. If the value is negative the program prints a message and assign an absolute value of x to the y. We use conditional expression, operator comma, printf.

Solution:

- remember: printf is a function and it can be used in the expression. The solution can look like this:

Example – operator “comma”

```
int main(int argc, char **argv)
{
  int x,y;
  scanf("%d",&x);
  y = (x<0) ? printf("You entered value < 0."),-x : x;
  system("PAUSE");
  return 0;
}
```

- **branches:**

```
switch (x)
{
   case 1: command1; break;
   case 2:
   case 3: command3; break;
   default: command4;
}
```

- **break** !!!!
- **continue** is bind only with loops, not with **switch** !

Console input / output in C

- file `stdio.h`
- `printf(const char*format,...)`
- `scanf(const char*format, ...)`
- scanf: operator &

```
scanf("%d",&x)
```

Tools and environments

- Bloodshed Dev C++
  - http://www.bloodshed.net/devcpp.html
- CodeBlocks
- event. MS Visual C++