Low-level programming

Lecture 2

Core instructions set

Institute of Computing Science
Faculty of Computing and Telecommunication
Marcin Radom, Ph.D.

Instructions

- In broad sense an **instruction** is some task / order / equation / etc. doing something within code.
- In C language *x = 0;* or *float a = function(16, 20, 67.35);* becomes an **instruction** when ended with `;` sign.

```
float a = function(16, 20, 67.35);
```

- **The program** in a broad sense is a set of instructions performing some tasks.
- **Typical simple instructions:**

```
int   m,  n = 1;
;                   // empty instruction
m = n * n - 1;      // changing value of m
n++;                // increments of n
m + n;              // well... It is an instruction in theory,
                   // but the result will not be stored
```

Instructions

- **Block of instructions (or: a complex instruction)** is a set of instructions gathered in braces:

  { instr_1; instr_2; … }
- After ending } bracket there is no semicolon **;** !!!
- Blocks of instructions can be nested, so within {} there can be some other block with {}
- Variables can be declared within the block of instruction (it narrows their visibility).
- **Example:**

```
float p, q;
{
  p = 3.5;
  q = 7.1 + p++ ;
}
{p = q; q = 1;}   // semicolon after last instruction
                  // is necessary, but after last }  – not
```

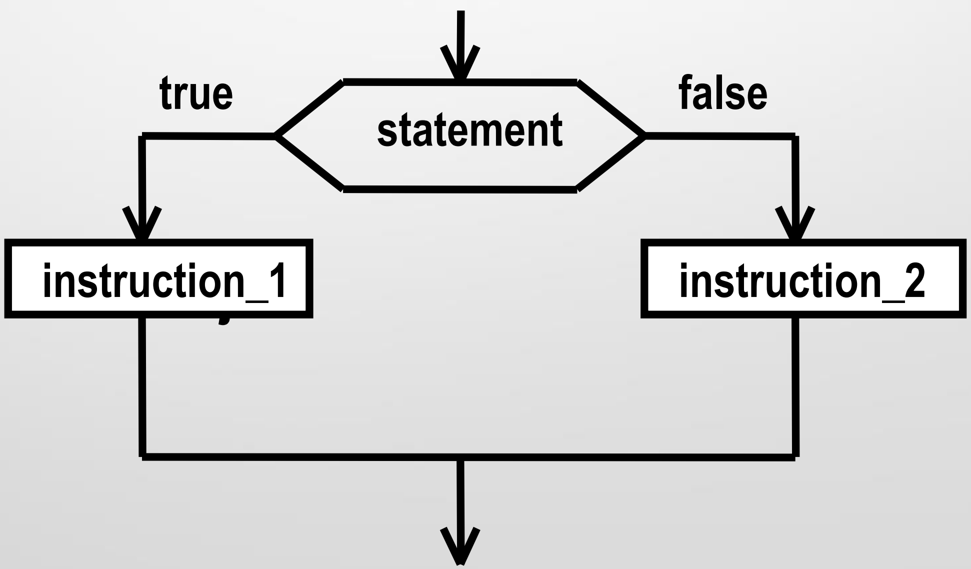
Instruction if-else

Theory and examples

Conditional instruction if-else

- When there is some decision to make in the code, we use this type conditional statement.
- **Formally:**

```
if (  statement )
     instruction_1
else
 instruction_2
```



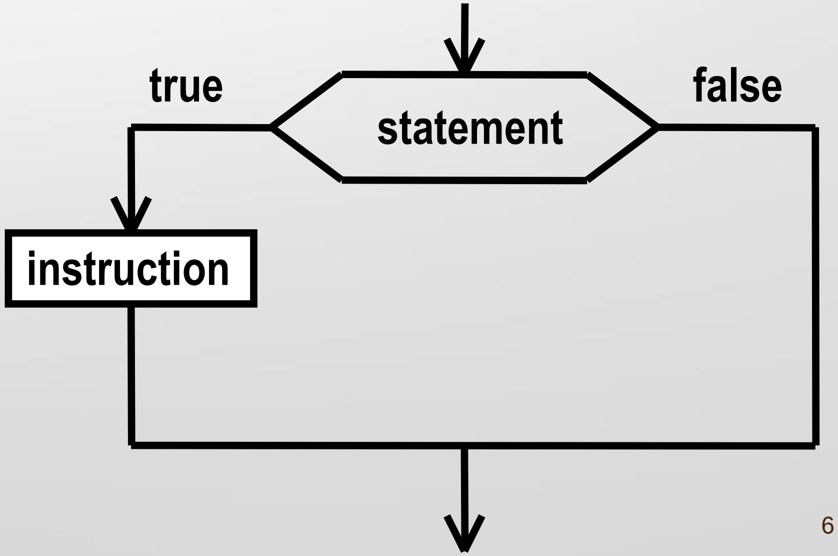
Conditional instruction if-else

- At the beginning the value of the statement must be determined. If this value is (arithmetically) different than 0, then (logically) it is considered **true** statement. Then the instructions will be computed / executed. If the statement is **false** (0 as value), then the instructions in else block will be performed **if this block exists (it is not obligatory).**
- According to this explanation how if-else works, there is no difference between:

```
if (statement) { /* … */ }
```
and
```
if (statement!= 0) { /* … */ }
```

- The else part can be completely omitted:

```
if (statement) instruction
```



Conditional instruction if-else

- Instructions **if-else** can be nested:
  - Because instruction **else** is not required for every **if**, there can be ambiguity to which **if** the **else** instruction belongs.
  - In such a case, it is assumed that **else** belongs always to the last used (written) **if** instruction (braces: {} matters!)
- **For example:** the **else** belongs to the second **if** (`if (a > b)`), no matter what the tabulation of the code suggests. Such an error can be difficult to detect!

```
if (n > 0)
    if (a > b)
        z = a;
else
    z = b;
```

Conditional instruction if-else

- **Example:**
  - Instruction **if** without **else**.

```
long  k, m;
char  flag;
if (k > m)   flag = 0;
if (k < m) {
    flag = 1;
    k = m - k;
}
if (m == 1)   {    // !!!
     if ( k )      // k != 0
               flag = 2;
     if ( !k )     // k == 0
         flag = 3;
}
```

Conditional instruction if-else

- Example for both **if-else**

```
int i,  f;
if (i > 5)
     f = 3;
else
     --f ;
/* semicolon ; before else is necessary! (syntax) */
double have,  pay,  account,  debt;
if (ma > pay) {
     account = have - pay;
     debt = -1;
} else {
     account = -1;
     debt = pay - have;
}
```

- **Another example:**

```
if   (a)   if   (b)   c;   else   d;
     /* is equal to */
if   (a)   {  if   (b)   c;
               else  d;  }
```

Conditional instruction if-else

```
if   (a)   if   (b)   c;   else  d;  else
if   (e)   f;   else   g;
   /* is equal to */
if (a) {
   if (b)
      c;
   else
      d;
}
else {
   if (e)
      f;
   else
      g;
}
```

- **Program examples:**

```
void main() { // requires: math.h
     // declarations
     double a, b,          // parameters
      G;          // result
     int good = 1;
     // reading data
     printf ("\nEnter value a:");
     scanf ("%lf", &a);
```

Conditional instruction if-else

```
printf ("\nEnter value b:");
scanf ("%lf", &b);
// calculate the result:
if (a >= b) {
     if ( -b > 0)
          G = a * a + log(-b);
     else
          good = 0;
} else {
     if (b >= 0)
          G = a - sqrt(b);
     else
          good = 0;
}
// show the result:
if (good)
     printf("\nG value is: %.4lf\n\n", G);
else
     printf("Cannot be calculated!\n\n");
}
```

Conditional instruction if-else

```
int main() { // requires math.h
     // declarations
     double x, y,         // parameters
               F;         // result
     // read data:
     printf ("\nEnter value x:");
     scanf ("%lf", &x);

     printf ("\nEnter value y:");
     scanf ("%lf",&y);

     // calculate the result
     if (x > y)
          F = x * x + y - 1;
     if (x == y)
          F = sin(y) + 2;
     if (x < y)
          F = cos(x) - y + 2;

     // show the result
     printf("\nF value is equal to: %.4lf\n\n", F);
     return 0;
}
```

Nested conditional instruction if-else

- **Construction:**

```
if (statement)    {
 instruction;
} else if (statement) {
 instruction;
} else if (statement) {
 instruction;
} else {
 instruction;
}
```

- **Example:**

```
if (x > y) z = 1;
else if (x < y) z = -1;
else z = 0; // x == y
```

- Such a form of **if** allows taking more complex decisions than binary one.
- In principle such a construction **if-else** is not a new type of instruction, but only an extension of **if**-**else**.
- In such a construction the first statement which match to a given condition, will be executed (i.e., its instructions).
- The last **else** instruction means „if all previous conditions were not meet” – then its instruction will be performed.

Instruction switch

And with it: case, default, break

Instruction switch

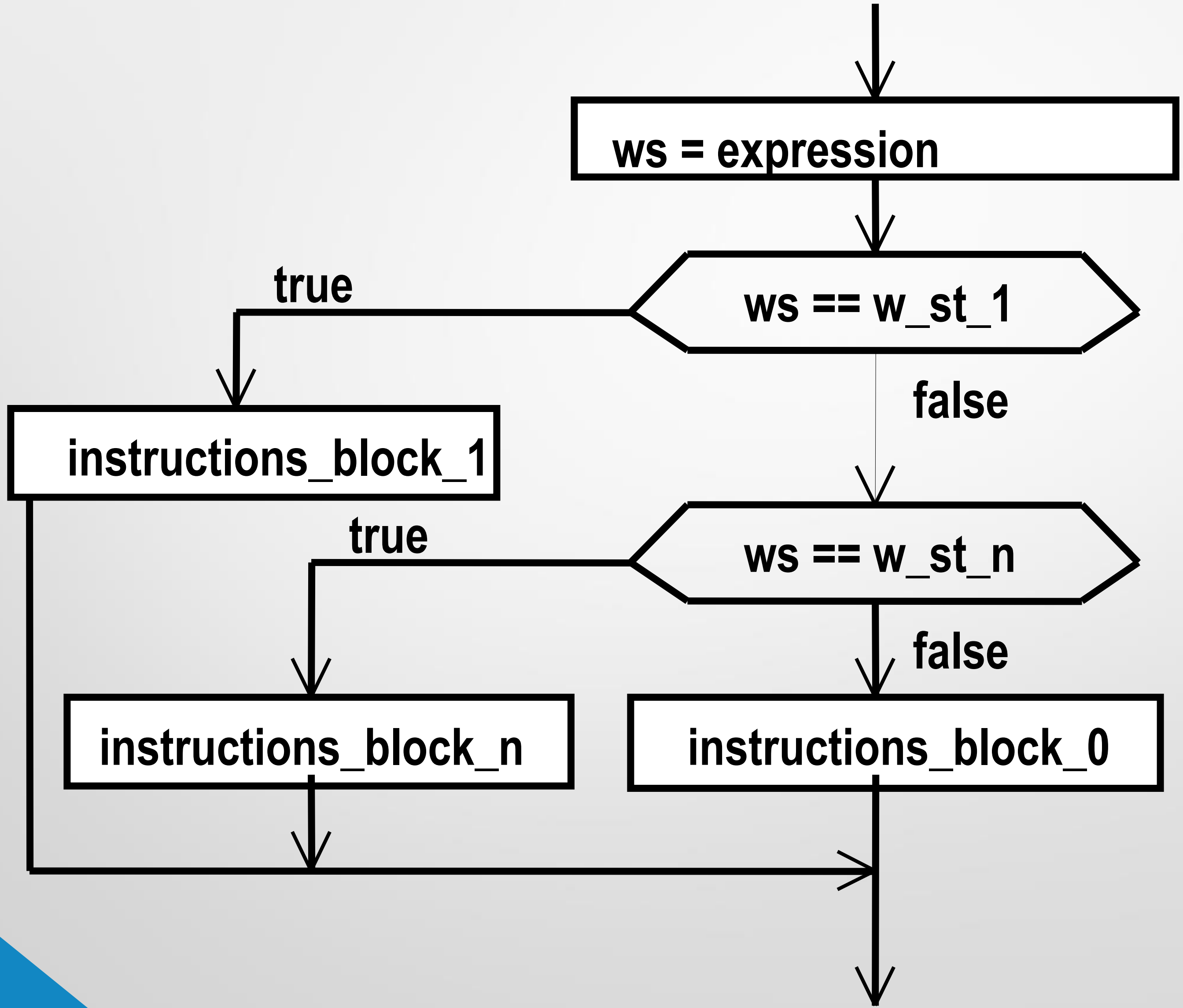
- The instruction **switch** allows the implementation of complex conditional and branching operations, much like nested if-else.
- **Construction:**

```
switch ( expression )
{
  case constant_expression_1:
    instruction_block;
    break;
  . . .
  case constant_expression_n:
    instruction_block;
    break;
  default:
    instruction_block;
}
```

Instruction switch

- With each **case** comes another condition in form of constant expression which is compared with the **switch** expression.
- If some **case** matches the expression, then its instructions will be executed. If no case matches, then the **default** instruction block (if it exists) will be performed.
- **default** is not obligatory. Without it, either if a **case** instruction block have been executed or not, the program starts doing instruction after **switch** braces.
- Instruction **break** tells the program to exit **switch** instruction block (the whole **switch** block).
- **Important: if there is no break, even if one case matches, the other will be at least checked, as well as the default block. The latter can also be additionally executed then (i.e. without break ending case instruction block).**
- It is wise to place **break** after every and each **case** block (however not obligatory in terms of C syntax).

Instruction switch



Instruction switch

- **Example:**

```
int howMany_a = 0,  howMany_b= 0,  howMany_xy= 0,  unknown= 0;
char zn;
switch  (zn) {
   case  'a' :  ++howMany_a;   break;
   case  'b' :  ++howMany_b;   break;
   case  'x' :
   case  'y' :  ++howMany_xy;  break;
   default   :  ++unknown;
}
// Example of different interpretation of a code by different compilers
// (VS2010 accept the code and will compile it properly, DevC++ do not want
// to compile it).  In short the example shows that expression constants can
// be computed using bit operations.
int state, next;
const int mask = 0x3A;    // const-qualified variable is not a constant expression
                          // it's a value you cannot modify
switch (state & mask ) {  //error:case label does not reduce to an integer constant
  case mask & 0x02 : next = 0x15; state = 0x21; break;
  case mask & 0x30 : next = 0x1F; state = 0x21; break;
  default : state = 0; next = 0;
}
```

Instruction switch

```
void main() {
     // declarations
     int x;          // parameters
     char option;
     // reading data
     printf ("\nEnter value x : ");
     scanf ("%d", &x);
     printf ("\nChoose option [D, H, X, F] :  ");
     fflush(stdin);             // clear buffer
     scanf ("%c", &option);
     // results
     switch (option & 0x5F) { // change small ASCII letters into large ones
          case 'D'  :    printf("\n%d\n\n", x); break; // decimal
          case 'H'  :    printf("\n%x\n\n", x); break; // hexadecimal, small letters
          case 'X'  :    printf("\n%X\n\n", x); break; // hexadecimal, large letters
          case 'F'  :    printf("\n%.2f\n\n", (float)x); break; // float
          default   :    printf("Wrong option.");
     }
}
```

Loop instruction: for

Theory and examples

Loop: for

- **Loop:**

```
for  ( initialization ; limit_statement/condition; counter)
                    repetitive_instructions
```

- Initialization part is performed once, before all other instructions
- Loop ends when the statement within () becomes **false** (while it is **true**, the loop is repeated).
- All the three parts (initialization, limit statement, counter) can be omitted, however two **;** signs must remain.
- If there is no limit statement within for loop, it is assumed it is always true, therefore we obtain an infinite loop

```
for ( ; ; )
{ /* … */ } //infinite loop
```

Loop for

- Example:

```
for( i = 0; i < n ; i++)      {
    // iterations with n repetitions
}
```

- Inside the body of loop, both the conditio (i < n) and the value of counter variable (variable ***i***) can be changed.
- After the last repetition (for whatever reasons) the counter variable (***i*** on the example) has its last assigned value (i.e., it is not cleared).
- Example of a **function changing series of signs into a number**:

```
#include <ctype.h>
int atoi(char s[]) {
      int i, n, sign;       // white signs: \n \t \v \f \r space
      for (i = 0; isspace(s[i]); i++)
           ;          // omit the white signs
      sign = (s[i] == '-') ? -1 : 1;
      if (s[i] == '+' || s[i] == '-') i++;  // omit the + or – sign for the value
           // isdigit return non zero value when the argument is a digit
      for (n = 0; isdigit(s[i]); i++)  // for all digits
           n = 10 * n + (s[i] - '0’); // e.g. 124 -> 10*0+1, 1*10+2, 12*10+4=124
      return sign * n;
}
```

Loop for

- **Example:**

```
int  s = 0;
for  ( int i = 0;  i <= 9;  ++i)  s += i;
/* initialization also defines the control variable here */
```

- **Example:**

```
int  i, k = 1525 ;
long m ;
// comma: , in the loop for, calculations
// performed from left to right
for  ( i = k, m = 0; i > 0; i -= 3 )
{
   if (i & 1)  ++m ; // counting the number of even indices
}
```

Loop for

```
bool go_on = true;
int where;
for (int i = 0; i < N && go_on; ++i)
{
     ........
     if ( .... )
         go_on = false;
        //finishing the loop without a break order
     else
    ........
}
// variable i is not being valid after the final  }
```

Loop for

```
bool go_on = true;
int i,  where = -1;
const int N = 12, searched = 333;
int Tab[N] = {0, 1, 333};
for (i = 0; i < N && go_on; ++i)
{
    if ( Tab[i] == searched)
       go_on = false;
}
// where value i == 3
```

Loop for

```c
#include <string.h>
/* reversing order of all elements of s[] */
void reverse(char s[]){
     int c, i, j;
     // comma in for, calculations from left to right
     for (i = 0, j = strlen(s)-1; i < j; i++, j--)
     {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
     }
}
```

```
instrukcja      i = 0, j = 9
anstrukcji      i = 1, j = 8
ajstrukcni      i = 2, j = 7
ajctruksni      i = 3, j = 6
ajckrutsni      i = 4, j = 5
ajckurtsni
```

- **It should be noted, that commas do not guarantee calculations from left to right (still, it is *often* calculated in that order).**
- *strlen(s)* – giving the number of elements of s (its size actually), we subtract 1 in the example because the enumeration of table elements starts from 0.

Loop instruction: while

Theory, examples and another loop: do – while

Loop while

- **Loop :**

```
while  (  condition )
           instructions
```

- At the beginning, the condition is being checked. If it is true (or: different than 0) then the instructions will be performed.
- Loop **while** *(condition)* will repeat its instructions to the moment when the limit statement will become **false**.
- In order to avoid the infinitive repetitions, somewhere within the body of loop { } there must be instructions which allow the condition to become false.

Loop while

- **Example:**

```
float sum = 1573.821, element= 3.51;
int counter= 0;

while (sum > 1E-10)       // 1E-10 = 1×10^-10
{
     sum -= element;
     element *= element;
     ++counter;
}
// counter = 4, sum = -21632.44
```

Loop while

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

int main() {
     int n = 0, K = 1;
     // read the starting values
     while (n < 1){
          printf("Enter integer value [n > 0] : ");
          scanf("%d", &n);
     }
     while (n > 1){
          K *= (n - 1) * (n - 1) + 1;
          --n; // modification of n (see: condition)
     }
     printf("K = %d\n", K);
     return 0;
}
```

Loop for vs while

- Using **for** or **while** usually depends on the programmer preferences.
- **Example:**

```
for (i = 0; i < 10; i++){ /* … */ }
```

is equal to:

```
i = 0;
while( i < 10 ){
    // …
    i++;
}
```

- In a situation where initial and counting values are easily available, it is better to use **for** because it groups them in one line.
- In **while** after opening bracket there is no initializing nor counter value parts (as they were withing the **for** loop) – these instructions must be put within the body of **while**.

Loop for vs while

```
#include <stdio.h>
#include <math.h>
void main() {
     int n, i;     // tmp variables
     double a, S;  // parameter
     n = -1;
         // reading values:
     while ( n < 1 )   {
         printf ("\nEnter limit value n (integer greater than 0) : ");
         scanf ("%d", &n);
     }
     printf ("\nEnter parameter a (real value): ");
     scanf ("%lf", &a);
         // counting sum
     S = 0.0; // neutral value
         // sum n times
     for ( i = 1 ; i <= n ; ++i )
         S += ( a * pow(i, 3.0) - 7 ) / ( i * i + 1 );
         // results:
     printf("\nSum is equal to: %.4lf\n\n", S);
}
```

Loop do-while

- **Loop:**

```
do   repetitive_instructions
while   (  condition ) ;
```

- Loops **`while`** and **`for`** check if they can start their iterations before starting. It is then possible that such loops won't perform even one iterations (if the condition is false/zero from the very beginning).
- In a loop **`do - while`** condition is checked after the instructions in the loop body are performed for the first time. <u>Therefore such a loop will have at least one iteration</u>.
- First the repetitive instructions are performed, then the limit statement is being calculated and checked if it is true. If it is false, the loop ends.

Loop do-while

```
long   ab = 3,  cd = 2;
// the loop will make at least one iteration
do  {
      ab *= ab;
      cd += cd;
}
while  (ab < cd);
// after the first iteration: ab == 9  cd == 4
```

Infinite loops

- Examples:

```
int s = 0, i; // i has not been initialized
for ( int n = 0; n < 10; ++i ) // infinitive loop, n does not change
     // value i is random, the result – also
    s += i;

float  A = 3.485e2,  eps = 1.38534e-2;
long  k;
while  (A != 0)
{
     A -= eps;
     ++k;
}    // ? – infinite loop, real value will probably never be zero ( !!! )

unsigned char  k = 5;
do
   k -= 2;
while (k != 0);    // infinite loop
```

Examples of using loops

Loops: for, while, do – while

Loops – Example 1

- **Example: converting text to decimal integer with a sign**

```
char* Text = "   -1574 "; // number in text form
int X = 0;
bool sign = false, // +, so its positive number
      flag = true;

// ignoring all sign which are NOT digit
while (*Text && flag)
     if (*Text == '+' || *Text == '-' || *Text >= '0' && *Text <= '9')
          flag = false;
     else // moving through elements using hidden pointer
          Text++;

if (flag) { // if true, then the end of text has been reached
     printf("\nNot a number.\n");
     return;
}
```

Loops – Example 1 continues

```
if (*Text < '0') {    // ASCII code for + or –
     if (*Text == '-’)
          sign = true;         // -, negative number
     Text++;
}
      // after the number there can be other signs,
     // spaces for example
while (*Text >= '0' && *Text <= '9’)
     X = X * 10 + *Text++ - 0x30; // X = X * 10 + *Text++ - ,0’
     // 0x30 hexadecimal for DIGIT 0;
     // *Text++ points to digits and move 1 digit forward
     // (*Text++ - 0x30) this subtraction gives us a digit
     // value in numerical form
     // Values X: 1; 1*10 + 5; 15*10 + 7; 157*10 + 4
     // Result: X = 1574

  if (sign)
     X = -X; // Result: X = -1574
  printf("\nX = %d\n\n", X);
```

```
X = 1
X = 15
X = 157
X = 1574

X = -1574
```

Loops – Example 2

- **Example: conversion of integer into a text**

```
int X = -31594;   // example value
int X = INT_MIN;  // minimal int value from limits.h

int Weight = 1000000000;   // 10E9 – initial divisor,
         // for int max. value approx. 2.4 billion

printf("\nX = ");      // starting
if (X == INT_MIN){     // for case INT_MIN
     printf("-2147483648\n\n"); // already known
     return;       // end of program
}

if (X < 0) {       // for negative value
     _putch('-'); // show – and count negative value
     X = - X; // would not work for INT_MIN, there is no opposite value
             // equal to 2147483648 (INT_MAX is equal to 2147483647)
}
```

Loops – Example 2 continues

```
if (X < Weight){                // while  X >= 1E9 divisor is not changed
                                // while result is still greater
     while (Weight / 10 > X)// e.g. for 31594 divisor is equal to 10000
          Weight /= 10;         // divide by 10
     Weight /= 10;              // ending division
}
if (Weight == 0)                // in order not to divide by zero
     _putch('0’);
else
     while (Weight >= 1) {      // e.g. 31594/10000=3, 1594/1000=1,
                                // 594/100=5, 94/10=9, 4/1=4 so: 3 1 5 9 4
          _putch(X / Weight + 0x30);
          X %= Weight;
          Weight /= 10;
 }
_putch('\n\n');
```

Loops – Example 3

- **Example: text into hexadecimal value**

```
char* Text = "   A1b2C3  "; // starting value
unsigned int X = 0;
bool flag = true;

// all signs which are not 0-9 nor A-F are ignored
while (*Text && flag)
     if (*Text >= '0' && *Text <= '9' ||
          (*Text & 0x5F) >= 'A' && (*Text & 0x5F) <= 'F')
     flag = false;
     else // moving through digits by pointer
          Text++;
        // if flag=true, then all text has been read
     if (flag) {
          printf("Not a number.");
     return;
 }
```

Loops – Example 3 continues

```
while (*Text >= '0' && *Text <= '9’
 || (*Text & 0x5F) >= 'A' && (*Text & 0x5F) <= 'F’) {
      // X = X * 16 + (digit or letter);
      // X <<= 4 corresponds to X = X * 16;
      X <<= 4;           // e.g. A * 10_16 (16_10) = A0; A1 * 10_16 (16_10) = A10
      if (*Text <= '9')            // digit 0-9
           X |= *Text++ - 0x30;    // A0+1=A1
      else               // letter A-F
           X |= (*Text++ & 0x5F) - 0x30 - 7;    // X=A10+B=A1B
                //* X = X | ((*Text++ & 0x5F) - 0x30 – 7)
                // e.g. A10:101000010000
                // B-0x30–7 (equal to 11_10):000000001011
                // result:101000011011 or A1B*/
 }
// X: A; A0 + 1; A10 + B;
// A1B0 + 2; A1B20 + C; A1B2C0 + 3
// Result: A1B2C3
printf("\nX = %X\n", X);
```

```
X = A
X = A1
X = A1B
X = A1B2
X = A1B2C
X = A1B2C3

X = A1B2C3
```

ASCII table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Loops – Example 4

- **Example: hexadecimal value to text**

```
int X = 0x9A0B7C;       // starting value
unsigned int Mask = 0xF;  //00000000000000000000000000001111
int L = sizeof (int) * 8; // # of bits – usually 32
unsigned char Char;     // ASCII code of next digit
any = false;
printf("\nX = ");       // starting
      //11110000000000000000000000000000
Mask = Mask << (L - 4); // for  L = 32 o 28
for (int i = 0; i < L >> 2; i++) {
      Char = (Mask & X) >> (L - (i + 1) * 4);
      if ( Char || any){
          any = true;      // if digit not zero
          if (Char > 9)
                _putch(Char + 0x37); // for A : 10_10 + 0x37 (55_10) = 65_10
          else
               _putch(Char + 0x30); // for 9 : 9_10 + 0x30 (48_10) = 57_10
      }
      // e.g. 00000000000011110000000000000000
      Mask >>= 4;
}
if (!any) putch('0’);
 _putch('\n');
```

Loops – Example 5

- **Example: decimal value to binary values and vice versa**

```
#include <conio.h>
int main() {
      unsigned int number = 0;
      //2^31 or 10000000000000000000000000000000
      unsigned int mask = 0x80000000;
      unsigned long long li;
      int go_on = 1, isZero= 0;
      char sign= 'X’;
      int count;
      printf("Enter binary value Bxxxx or decimal Dxxxxxx :\n");
      while(sign != 'B' && sign != 'D’)    {
           sign = getche();
           sign &= 0x5F;
  }
```

Loops – Example 5 continues

```
switch (sign) {
    case 'B’ :
        while(go_on) {
            znak = getche();
            if (sign != '0' && sign!= '1’) {
              printf("\nD %d\n", number);
              go_on= 0;
            } else {
              sign <<= 1;
              number |= sign- 0x30;
            }
        }
        break;
```

```
1        liczba = 1
0        liczba = 2
1        liczba = 5
1        liczba = 11
1        liczba = 23

bin =             liczba =  23
```

Loops – Example 5 continues

```
case 'D’ :
     scanf("%d", &number);
     count = 32;
     printf("\nB ");
     while(count != 0) {
          if ((number & mask) == 0) {
              if (isZero) putch('0’);
          } else {
              putch('1’);
              isZero= 1;
          }
          masa >>= 1;
          -- count;
     }
     putch('\n’);
     break;
 }
 printf("\n\n");
 return 0;
}
```

Loop interrupting: break, continue

Theory and examples

Instruction break

- Often the loops (**for**, **do**, **while**) must be ended (or we want to leave **switch** block) disregarding the current state of a condition.
- Instruction **break** forces loop to end immediately (the very inner loop where **break** resides)

```
for (int i = 0; i < n; i++) {// ...
     // ending loop never mind what i value really is
     if (other_variable== 10)
        break;
}
```

- Example: function *trim* deletes spaces, tabulations and \n sign at the end of text. Instruction **break** leaves the loop when first sign different than space or (\t, \n,) is found searching **from right to left** in text:

```
int trim(char s[]){
    int n;
    for (n = strlen(s) - 1; n >= 0; n--) {
         if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n’)
            break;
    }
    s[n+1] = '\0’;
    return n;
}
```

Instruction continue

- It forces the loops (**for**, **do**, **while**) to finish current iteration and begin the next one from the beginning **if possible** (depends on the condition).
- In a case of loops **do** and **while** it forces the loops to immediately check its condition (and if it is still valid, the next iteration starts from the beginning, i.e. its first instruction.
- In a case of loop **for** it forces the loop to change the counter, check the condition and if it is valid, to continue with the **next** iteration (from its first instruction).
- **Example:**

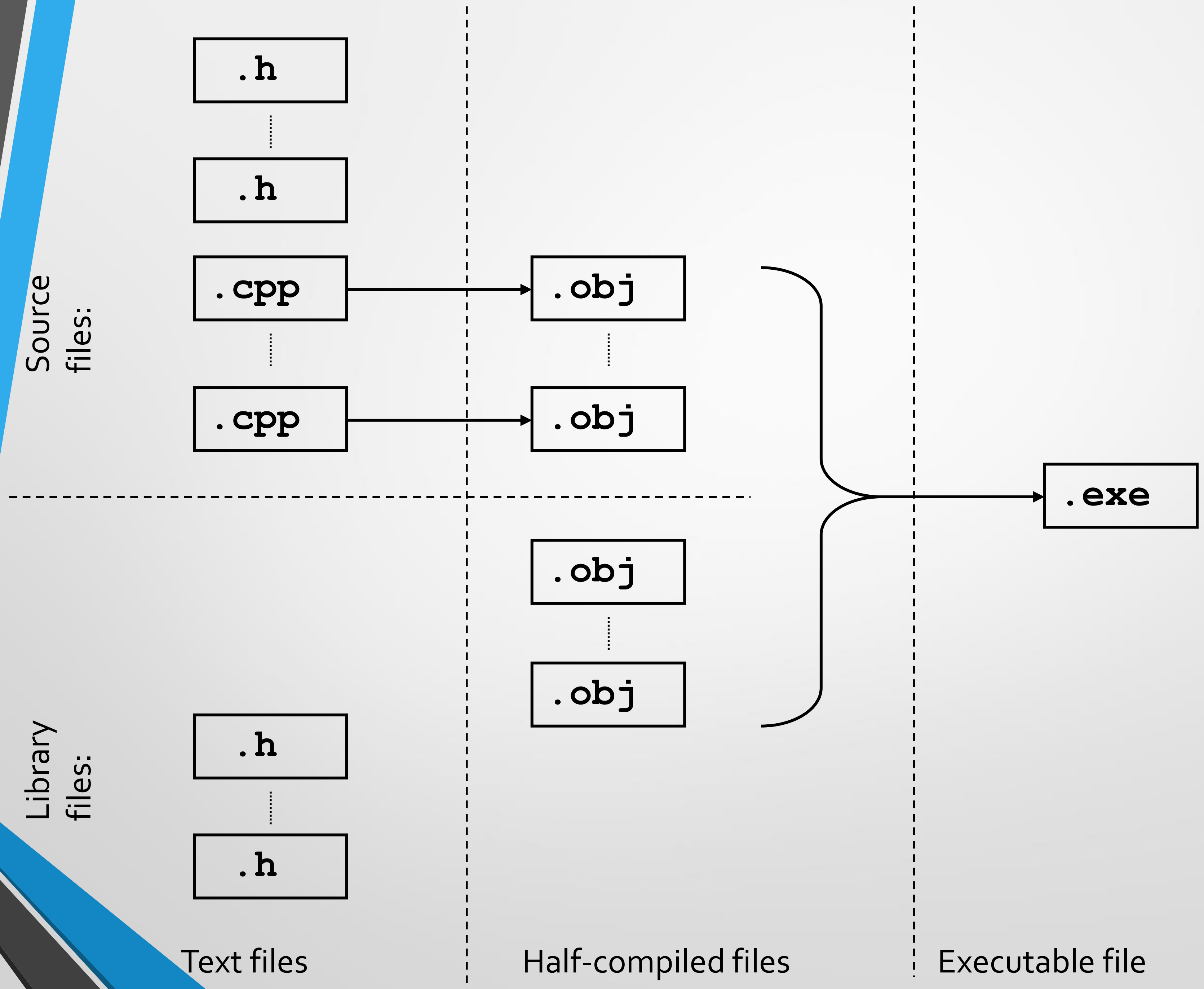
```
for( i = 0; i < 100; i++ ){
    int x = i * i;
    if(x % 2 == 0)
        continue;
    printf("%d", x); // only if x is odd value
}
```

Compilation, precompilation

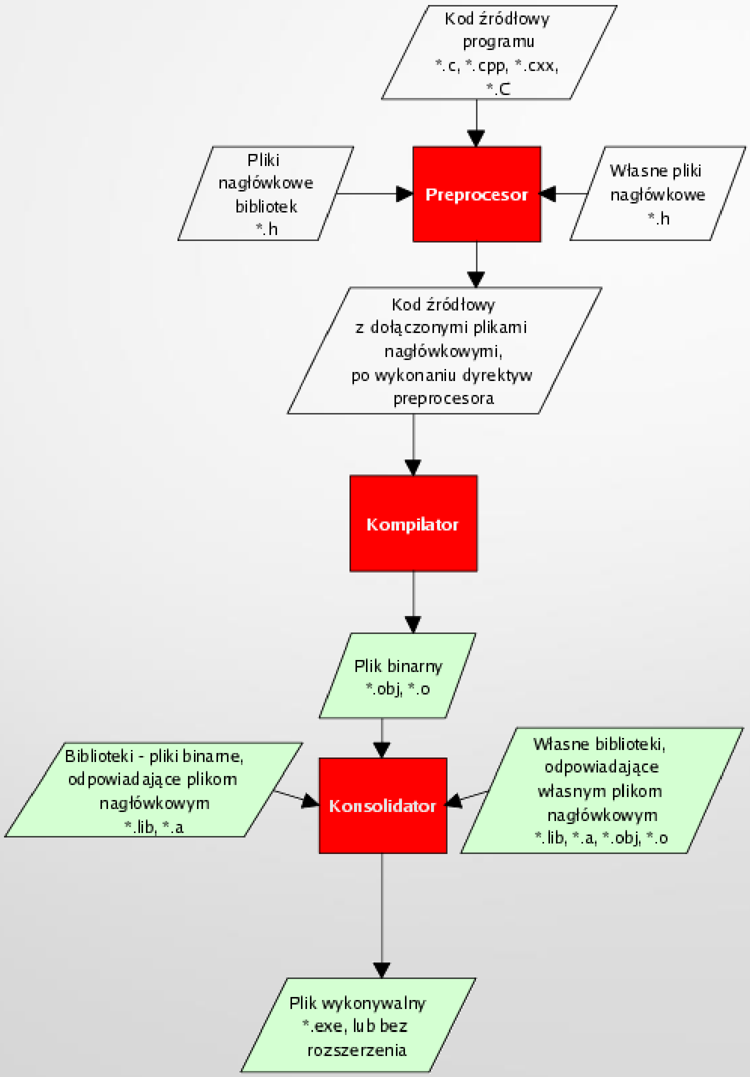
And some useful info about function **main (…)**

Preprocessor (precompiler)

- It is a compiler module which starts to change our code according to the language rules before the proper compiler will change it into a machine code.
- Using **directives** (the orders of the precompiler) it changes fragments of the code. Directives are not instructions, they don't end with **;**
- It can for example join files or define which sections of the code will be visible to the main compiler.
- Lines starting with **#** have order for pre-compiler. They can be anywhere in the code.
- Syntax: ***# directive arguments***
- Two already know directives are **#define** and **#include**.
- **Adding library files**: `#include`
  - `#include <stdio.h>` – library will be searched in the language directories.
  - `#include "functions.h"` – searching for such a file will start in the project main directory.



.h
.h
.cpp
.cpp
.obj
.obj
.obj
.obj
.exe
Source files:
Library files:
Text files
Half-compiled files
Executable file


Kod źródłowy programu *.c, *.cpp, *.cxx, *.C
Pliki nagłówkowe bibliotek *.h
Preprocesor
Własne pliki nagłówkowe *.h
Kod źródłowy z dołączonymi plikami nagłówkowymi, po wykonaniu dyrektyw preprocesora
Kompilator
Plik binarny *.obj, *.o
Biblioteki - pliki binarne, odpowiadające plikom nagłówkowym *.lib, *.a
Konsolidator
Własne biblioteki, odpowiadające własnym plikom nagłówkowym *.lib, *.a, *.obj, *.o
Plik wykonywalny *.exe, lub bez rozszerzenia

Preprocessor – changing texts: #define

- **Example:**

```
#define  name changed_into
```

will tell the preprocessor to change every ***name*** within our code into ***changed_into*** text.

- **Example:**

```
#define  SIZE    150
#define  mine    buying – selling
#define  EPS    3.5E-8
........
#undef   EPS     // deleting all EPS
#define  EPS    1.5E-8
```

- **Another example:**

```
#define forever for( ; ; ) // changes forever into infinite loop
```

Macrogenerations (macro)

- <u>More advanced text change tool – with arguments:</u>

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

  - First of all: it is not a C function (however look that way)/
  - Every *max* with arguments A and B will be changed into advanced form as defined in above macro.
  - **E.g.:**

```
x = max(p+q, r+s);
```

  - will be changed into:

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

- Can be dangerous is the syntax is not 100% correct!

Macrogenerations (macro)

- **Example:**

```
max(i++, j++)
```

Changes into:

```
((i++) > (j++) ? (i++) : (j++));
```

but the initially bigger value will be incremented twice – which is probably not what we wanted…

- **Examples:**

```
#define Makro1(x) x = sin(x) + 3 * x;
..........................
double akr = 2.544;
Makro1(akr)          // akr = sin(akr) + 3 * akr;
#define Makro2(x, y) x = x + y - 1;
..........................
double alfa = -12.74, beta = 0.21;
Makro2(alfa, beta)    // alfa = alfa + beta – 1
..........................
#define square(x) x * x;  // ERROR. square(z+1): z+1*z+1
square(z+1);
```

Conditional compilation

- Whole fragments of the code can be included or excluded from compilation using precompiler directives.
- Directives:
  - **#if**
  - **#endif** - *} in normal if body*
  - **#elif** - *else if*
  - **#else**
  - **#defined**(name) – gives true (1) if name is already defined (using **#define**), 0 otherwise.
  - **#ifndef**, **#ifdef** – checks if names is already defined

Conditional compilation

- **General form:**

```
#if    constant_statement_1
   source_text_1
#elif constant_statement_2
  source_text_1
...........................
#else
  source_text_n
#endif
```

- **Example**:

```
#if ! defined(HDR)
#define HDR
#endif
```

Examples

- **Example:**

```
#defined  identifier
// 1 : if is defined
// 0 : if not defined

#if   defined identifier
/* equal to */
#ifdef identifier

#if   ! defined identifier
/* equal to*/
#ifndef identifier
```

- **Example:**

```
#define trialVersion
//
#ifdef trialVersion
. . . . . . . . . . . . . . . . . . . . .
#else
. . . . . . . . . . . . . . . . . . . . .
#endif
```

Function *main(...)*

- In **C99** standard there can be 2 versions:

```
int main ( void )
    // version with no argument (up) and with arguments (below) :
int main ( int argc, char *argv[] )
```

- In **C89** using **`int main ( )`** is allowed, but it is advised to use **C99**
- Function *main()* should return a value.
- Wrong from the standard point of view, but it can work (however on some systems can be source of errors).

```
void main( void )
{ ... }
    // Bjarne Stroustrup:
    // " It is not and never has been C++, nor has it even been C."
void main ( )
{ ... }
```

Function *main*

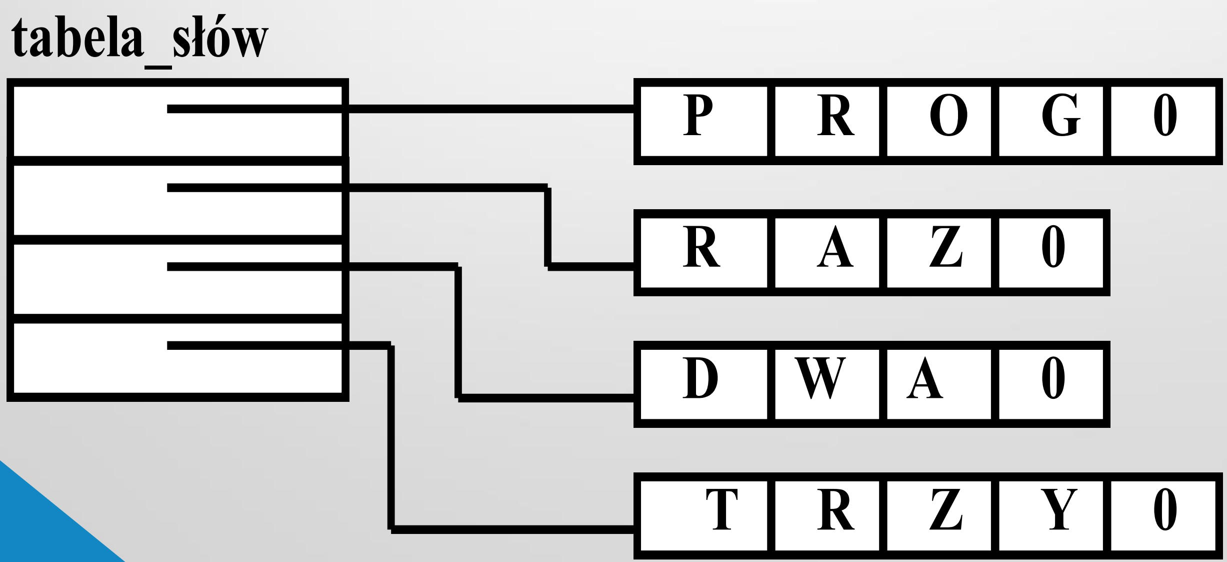
- <u>Example of *main*:</u>

```
int main ( int number_of_words, char *word_table[  ] )
{ ... }
```

  - **Example:**

    PROG RAZ DWA TRZY

    number_of_words: 4



Function *main*

- **Example:**

```c
int main (int LiPa, char* TaPa[]) {
	int index;
	if ( LiPa < 2 )
	{
		printf("\n No parameters.\n\n");
		return;
	}
	for ( index = 1; index < LiPa; index++)
		printf ("\n Parameter %d : %s", index, TaPa[index]);
	printf("\n\n");
}
```

Questions?