

Visibility and lifetime (1)

Block is a section of code enclosed into braces. The body of function is a block. The compound statement used in looping (*while*) and branching (*if, if else*) is also a block, i.e. a block may contain other blocks. Let us have a function:

```
void fun(int x)
{
    int a = 0, b;
    ..... // some actions
}
```

Question: are the identifiers *x* (formal parameter), *a* and *b* (declared in function) now reserved or is it possible to use the same identifiers in other functions (for example in *main*) of the same program?

Answer: **variables declared in a block are visible only inside the same block** or in other words, they have **block scope**. This is valid also for formal parameters. So, identifiers *x*, *a* and *b* are reserved only inside function *fun*. You may freely use the same names in other functions. Example:

```
int main()
{
    int x = 10;
    fun(x); // one x is declared in main and the other in fun. Here the value of x from main
           // is copied into x declared as formal parameter of fun
```

Visibility and lifetime (2)

```
void fun(int x)
{
    int a = 0, b;
    ..... // some actions
}
```

Question: when the identifiers x (formal parameter), a and b (declared in function) get their memory field (4-byte each) and when they lose it?

Answer: the memory for them is **allocated when the program enters into their scope** and **freed when the program exits their scope**. So, when the function fun is called and starts to run, the operating system allocates memory for x , a and b . When function fun returns, x , a and b cease to exist and the operating system may use their memory for storing other information. Example:

```
int main()
{
    int i = 0, array[100];
    .....
    while (i < 100)
        fun(array[i]); // x, a and b get memory 100 times and lose 100 times.
                        // On each call a gets initial value 0 and b has each time
                        // some garbage initial value that may or may not be equal with the
                        // value it had at the end of previous run.
```

Visibility and lifetime (3)

```
void fun(int x)
{
    static int n;
    ..... // some actions
    n = n + 1;
}
```

Keyword *static* in declaration sets **global lifetime** (or storage duration). Memory for *static* variables is **allocated when the program starts to run** and is **freed when the program stops to run**. By default the *static* variables are **initialized to 0** but the programmer may set his / her own initial value. Example:

```
int main()
{
    int i = 0, array[100];
    .....
    while (i < 100)
        fun(array[i]); // n does not lose its memory. After first call its value is 1, after the
                       // second call its value is 2, etc.
```

On the same time, the **visibility of a *static* variable is local**, i.e. identifier *n* is reserved only inside function *fun*.

Visibility and lifetime (4)

A variable may be declared inside a block following the *while*, *if* or *else* keyword. In that case **their scope is not the function but the block inside the function**. Example:

```
void fun(int x)
{
    int a, b, k; // a gets memory
    ..... // some actions, a is used
    while (k < 10)
    {
        int a; // a gets memory
        ..... // some actions, a is used, a exists but is hidden, b may be used
    } // a is destroyed
    ..... // some actions , a is used
} // a is destroyed
```

Variables declared in a function or in a compound statement inside a function belong to the **auto (automatic) storage class** or **static storage class**. They can be accessed only within their scope, i.e. in the block they have been declared. Variables of static class with block scope preserve their value even when the program has exited their scope because the memory allocation for them is occurred only once when the program starts to run. Variables of auto class have memory and keep their current value only when the program is running in their scope.

Visibility and lifetime (5)

A variable may be declared out of functions: at the beginning of source code file between the preprocessor directives and the definition of the first function or between two definitions of functions. For example:

```
#include "stdio.h"
int fun1(int, int); // prototypes
void fun2(double, double);
double dm[50];
int i; // dm and i may be accessed in main, fun1 and fun2
int main()
{ // may access, read and modify i and dm
  .....
}
int fun1(int x, int y)
{ // may access, read and modify i and dm
  .....
}
void fun2(double d1, double d2)
{ // may access, read and modify i and dm
  .....
}
```

Visibility and lifetime (6)

Scope of variables declared out of functions includes all the functions defined in the current source code file and following to its declaration. Memory for them is allocated when the program starts to run and is freed when the program stops to run. By default they are initialized to 0 but the programmer may set his / her own initial value. So, they have file scope and global lifetime.

To extend their scope to functions defined in other source code files redeclare them with keyword *extern*, for example:

```
#include "stdio.h"
#include "stdlib.h"
int fun3(int, char); // prototypes
void fun4(double, double, int, int);
extern double dm[50];
extern int i; // dm and i may be accessed also in fun3 and fun4
int fun3(int x, char c)
{ // may access, read and modify i and dm
    .....
}
void fun4(double d1, double d2, int a, int b)
{ // may access, read and modify i and dm
    .....
}
```

Visibility and lifetime (7)

The names of variables with file scope and variables with block scope may match. In that case the variable with block scope shadows the variable with file scope:

```
#include "stdio.h"
int fun1(int, int); // prototypes
void fun2(double, double);
double dm[50];
int i;
int main()
{
    int i; // i is hidden and not accessible, dm is accessible
    .....
}
int fun1(int x, int y)
{
    .....
```

Remark: in C++ the hidden variables are accessible, but we have to use the `::` scope resolution operator. So in our *main*:

```
:: i = 10; // i is used
i = 10; // i is used
```

Visibility and lifetime (8)

If the names of the two or more variables with file scope match, linking fails. Such a situation may happen when there are several programmers working on the same large project.

However, **if a variable with file scope is declared as static, the linker does not handle it.** Static variables with file scope may be used only in the file in which they are declared. To extend them to other source code file is not possible. But two source code files may declare static variable with file scope having the same name:

```
#include "stdio.h"
int fun1(int, int); // prototypes
static double dm[50];
int i; // dm and i may be accessed in main and fun1
int main()
{ // may access, read and modify i and dm
    .....
}
int fun1(int x, int y)
{ // may access, read and modify i and dm
    .....
}
```


Visibility and lifetime (9)

Summary:

1. Variables may have block scope (called also as local variables) or file scope (called also as global variables).
2. Block scope variables may be of auto storage class or static storage class.
3. File scope variables if they are not of static storage class may be extended to other source code files. Thus we may get variables accessible in each function of our project.
4. File scope variables if they are of static storage class can be accessed only in the file where they are declared.
5. All the file scope variables as well as the block scope variables of static storage class have global lifetime: they get their memory when the program starts to run and lose when the program quits.
6. Block scope variables of auto storage class get memory when the program enters into their block and lose memory when the program leaves their block. It is said also that they have local lifetime.

Remark: in some books you may read that local variables of auto storage class may be optionally declared with keyword *auto*, for example:

```
auto int i = 10;
```

It is not true for C++ compilers: in C++ keyword *auto* means that the compiler must guess the variable type itself.

Informing about errors (1)

A function must not crash or hang. Therefore it must always check the values of arguments. If they are not correct, the function must inform the calling function and return. In our previous exercises, if the function was not able to perform its task, it returned *NAN*. But we can use *NAN* only if the return value type is *float* or *double*. In addition, return value *NAN* does not tell what exactly was the reason of failure.

A more universal solution is to use global variables, for example like this:

```
#define SUCCESS 0 // Do not use numbers without explanation: the code must be readable
#define PARAMETER_TOO_LARGE 1
#define PARAMETER_TOO_SMALL 2
#define NO_SOLUTION 3
int error_number = SUCCESS;
void fun(double);
.....
void fun(double x)
{
    if (x <= 1e3)
    {
        error_number = PARAMETER_TOO_SMALL;
        return;
    }
    .....
}
```

Informing about errors (2)

```
int main()
{
    double z;
    .....
    fun(z);
    if (error_number != SUCCESS)
    {
        if (error_number == PARAMETER_TOO_LARGE)
            printf("Parameter value %lg is too large\n", z);
        else if (error_number == PARAMETER_TOO_SMALL)
            printf("Parameter value %lg is too small\n", z);
        else
            printf("Calculation with parameter value %lg failed\n", z);
        printf("Press a key\n");
        _getch();
        return 1;
    }
    .....
```

Now the calling function as well as the human operator know what has happened.

Increment and decrement (1)

Let

```
int x = 1;
```

Then after statement

```
x++;
```

x is already 2. The same result we could achieve with statement

```
x = x + 1;
```

Increment `++` is an unary operator that increases the value of its operand by 1.

Decrement `--` decreases the value by, for example after

```
x--;
```

x is set back to 1, the alternative statement is

```
x = x - 1;
```

The operand must be an lvalue: an identifier or expression that specifies a certain memory field:

```
(x+10)++; // error
```

Increment and decrement come in two varieties:

```
<lvalue>++ // postfix mode
```

```
++<lvalue> // prefix mode
```

```
<lvalue>--
```

```
--<lvalue>
```

Increment and decrement (2)

```
int x = 10, y, z;  
x++; // x is now 11  
++x; // x is now 12, in those expressions there is no difference between the postfix and  
      // prefix modes  
y = x++; // After this expression x gets new value 13 and y gets new value 12.  
z = ++x; // After this expression x gets new value 14 and z gets new value 14.
```

In prefix incrementing (decrementing) the value is incremented (decremented) first and the new value is used for the following operations.

In postfix incrementing (decrementing) the other operations of expression are performed using the initial value. The operand is incremented (decremented) after being used.

Example:

```
#define ESC 27  
char line[40];  
int i = 0;  
while (i < 40)  
{  
    if ((line[i++] = _getche()) == ESC) // not line[++i]  
        break;  
}
```

Increment and decrement (3)

The same example written in the simplest mode:

```
char line[40];
int i = 0;
while (i < 40)
{
    char c = _getche();
    if (c == ESC)
        break;
    line[i] = c;
    i = i + 1;
}
```

One more version of this example:

```
char line[40];
int i = 0;
while (i < 40 && (line[i] = _getche()) != ESC)
    i++;
```

The shortest version:

```
char line[40];
int i = 0;
while (i < 40 && (line[i++] = _getche()) != ESC);
```

Looping (1)

```
double array[7] = { 1.0001, 1.256, 111.4, 44.5, 6.789, 12. 47, 54.98}, average;  
double sum = 0.0;  
int i = 0; // action 1: initialize the loop  
while (i < 7) // action 2: test the condition  
{  
    sum = sum + array[i];  
    i++; // action 3: update the condition  
}  
average = sum / 7;
```

In *for loop* those three actions are gathered into one place:

```
for (<initializing_statement(s)>; <condition>; <update_statement(s)>)  
    <loop_body_statement>
```

Example:

```
for (i = 0; i < 7; i++)  
    sum = sum + array[i];  
average = sum / 7;
```

There is no semicolon after update statement(s):

```
for (i = 0; i < 7; i++;) // error
```

Looping (2)

Exercise:

Write a function that fills an array with pseudo-random numbers and returns the maximum of them. To test write the *main* function.

Requirements:

- Use the *for* loop.
- The array to fill must be defined as global variable (i.e. it has file scope):
`int RandVal[100];`
- The prototype of function to write is
`int GetRandArray(int);`
where the parameter is the number of pseudo-random values to find.
- The *main* must read from keyboard the number of pseudo-random values to find and print the results.
- The *GetRandArray* function must check its input value. If it is negative, 0 or exceeds 100, the function may return any value.
- There must be a global variable to inform the *main* about success or errors in input data. The *GetRandArray* function must set the value of global variable. The *main* must check this value and in case of errors print the message.
- **Read the specification carefully!**

Looping (3)

```
for (<initializing_statement>; <condition>; <update_statement(s)>)  
    <loop_body_statement>
```

The initializing statement may be a variable declaration, for example:

```
int array[5][10]; // matrix (i.e.tabel) of 5 rows and 10 columns  
for (int i = 0; i < 5; i++) // declare local variable i  
{  
    for (int j = 0; j < 10; j++) // declare local variable j  
    {  
        array[i][j] = 0;  
    } // scope of j ends  
} // scope of i ends  
..... // after loop variables i and j are unrecognized
```

Looping (4)

Exercise:

Write a function that fills a global array

```
char Table[5][5];
```

in the following way:

```
* + + + +  
- * + + +  
- - * + +  
- - - * +  
- - - - *
```

The prototype of the function is your own choice. To test the result write *main* that prints the result.

Tips:

- You do not need the ASCII table. Use the character constants.
- For printing characters see the specification of *printf*.

Looping (5)

```
for (<initializing_statement>; <condition>; <update_statement(s)>)  
    <loop_body_statement>
```

Sometimes the initialization is not needed. In that case **write the empty statement** (i.e. only the semicolon). Example:

```
char buf[100]; // this array contains some text, we need to know the index of the last 'x'  
               // for example, in text "kxmlnxdiu" the result is 5  
int n = 0;  
for (; n < 100 && buf[n]; n++); // at the end of loop n is the index of terminating 0  
                               // semicolon right after for closing parentheses means that the  
                               // loop body is empty  
for (; n >= 0; n--) // moving backwards to the beginning of string  
{  
    if (buf[n] == 'x')  
        break;  
}  
if (n < 0)  
    printf("\'x\' not found\n");  
else  
    printf("%d is the index of last \'x\'\n", n);
```

Looping (6)

```
for (<initializing_statement(s)>; <condition>; <update_statement(s)>)  
    <loop_body_statement>
```

Sometimes we need several initialization statements. If they are not declaration statements, we may **separate them by comma**. Example:

```
int table[5][10]; // matrix (i.e. tabel) of 5 rows and 10 columns  
int sum[5]; // array of sums of table rows  
int i, j;  
for (i = 0; i < 5; i++)  
{  
    for (j = 0, sum[i] = 0; j < 10; j++) // but not int j =0, sum[i] = 0;  
    {  
        sum[i] = sum[i] + array[i][j];  
    }  
    printf("Row %d sum is %d\n", i, sum[i]);  
}
```


Looping (7)

```
for (<initializing_statement(s)>; <condition>; <update_statement(s)>
    <loop_body_statement>
```

Sometimes we do not need any update statement. In that case simply **omit it** (no semicolon to mark the empty statement). Example:

```
#define ESCAPE 27
char line[41];
int i;
for (i = 0; i < 40 && (line[i++] != _getche()) != ESCAPE; ); // empty body
```

Empty place



Some alternatives:

```
for (i = 0; i < 40 && (line[i] != _getche()) != ESCAPE; i++ );
```

or

```
for (i = 0; i < 40; i++)
{
    line[i] = _getche();
    if (line[i] == ESCAPE)
        break;
}
```

The resulting line is not a regular string: there is no terminating 0. To set it, write

```
line[i - 1] = '\0'; // the first version
```

```
line[i] = '\0'; // the second and third version
```

Looping (8)

```
for (<initializing_statement(s)>; <condition>; <update_statement(s)>
    <loop_body_statement>
```

Sometimes we may need several update statements. In that case **separate them by comma**.

Example:

```
int array1[40], array2[40];
for (int i = 0, j = 39; i < 40; i++, j--)
    array1[i] = array2[j]; // array1 is the array2 in reversed order
```

Sometimes we are not able to formulate the condition. In that case set it to 1 (i.e. TRUE) and use the *break* or *return* statements to exit the loop. Example:

```
#define ESCAPE 27
#define ENTER '\r' // CR or "carriage return"
printf("Press ESC to stop or ENTER to continue\n");
for (; 1; ) // no initializing, no updating
{
    char c = _getch();
    if (c == ESCAPE)
        return; // ESC was pressed, exit the function
    else if (c == ENTER)
        break; // ENTER was pressed, continue the function
} // if neither ESC nor ENTER was pressed, repeat the loop
```

Looping (9)

Exercise:

There are two global arrays:

```
char Buf1[81], Buf2[81];
```

Buf1 contains a text, consisting of words. The words are separated by sequences of spaces (character constant ' '). Those sequences may contain 1, 2 or more spaces. Example: "*I am a student*". The end of text is marked with byte containing zero. *Buf2* is empty.

Write a function that puts into *Buf 2* the same sentence but so that there is only one space between words. For example "*I am a student*".

The prototype of the function is your own choice. To test the result write *main* that reads the input text into *Buf1* and prints *Buf2*.

Tips:

- Do not forget that there must be 0 at the end of *Buf2*.
- To print *Buf2* write

```
printf("Text after compressing: %s\n", Buf2);
```

Looping (10)

Exercise (the specification continues on the next slide):

Write a program that is driven by the following menu:

v – count vowels

c - count consonants

x – exit

At the beginning the *main* prints this menu and reads the character from keyboard. If the user has pressed *v* (or *V*) or *c* (or *C*), the *main* prints "*Type input sentence*", reads the user's text into a global buffer, calls the function counting the vowels or consonants, prints the result and then prints the menu again. If the user has pressed *x* (or *X*), the *main* exits. Other input characters are ignored.

Tips:

The input text may contain any characters. To simplify the analysis, first convert all the letters to lowercase:

```
#include "ctype.h"
```

```
char c1 = 'A';
```

```
char c2 = tolower(c1); // c2 gets value 'a'
```

```
c1 = '1';
```

```
c2 = tolower(c1); // c2 is '1', only the letters are converted, the others are simply returned
```

```
c1 = 'a';
```

```
c2 = tolower(c1); // c2 is 'a', nothing to convert
```


Looping (11)

Exercise continues:

Then check is the character a letter:

```
#include "ctype.h"
char c1 = 'A';
if (isalpha(c1)) // returns TRUE
{.....}
c1 = '1';
if (isalpha(c1)) // returns FALSE
{.....}
```

If this is a letter check is it possible to find it in array

```
char vowels[] = { 'a', 'e', 'i', 'o', 'u', 'y' };
```

If not then it is a consonant.

It is useful to **split the program into 4 functions**: *main*, converter to lowercase, counter of vowels, counter of consonants.

In addition to *isalpha* and *tolower*, *ctype.h* presents more useful functions for text analyzing. See https://www.tutorialspoint.com/c_standard_library/ctype_h.htm

Looping (12)

The *continue* command causes the program to skip the remaining statements in the loop and start the next loop cycle (i.e. perform the update and test the condition). Example:

```
#define ESCAPE 27
#define ENTER '\r'
char line[41];
int i;
for (i = 0; i < 40; )
{
    char c = _getch(); // reads the character but not echoes on the screen
    if (isblank(c)) // see https://www.tutorialspoint.com/c\_standard\_library/ctype\_h.htm
        continue; // the user has typed a white character, ignore it
    if (c == ESCAPE || c == ENTER)
        break; // stop reading
    _putch(c); // echoes the input character
    line[i++] = c; // stores in the buffer
}
line[i] = 0; // put terminating zero
```

Looping (13)

In addition to *while* and *for* loops there is *do while* loop:

do

```
    <loop_body_statement>
```

```
while (<condition>);
```

In *while* as well as in *for* loops if already at the beginning the testing of condition gives us FALSE, the loop is not executed at all.

The *do while* loop is *always executed at least once*.

Example:

```
double array[7] = { 1.0001, 1.256, 111.4, 44.5, 6.789, 12. 47, 54.98}, average;
```

```
double sum = 0.0;
```

```
int i = 0;
```

```
do
```

```
{
```

```
    sum = sum + array[i];
```

```
    i++;
```

```
}
```

```
while (i < 7);
```

```
average = sum / 7;
```

Labels and jumping (1)

Label is an identifier marking a single statement:

```
<label>: <statement>
```


The *goto* statement:

```
goto <label>;
```

causes the program to jump to the statement bearing the indicated label.

Example:

```
double table1[5][5], table2[5][5];
int i, j;
for (i = 0; i < 5; i++) {
    for (j = 0; j < 5; j++) {
        if (table1[i][j] == 0)
            goto end; // cannot continue, break off the both loops
        else
            table2[i][j] = 1 / table1[i][j];
    }
}
end: if (!(i == 5 && j == 5))
    printf("Error: Table1 has zero elements\n");
```



You cannot use *goto* to jump to a label defined in another function.

Labels and jumping (2)

Usage of *goto* should be avoided because it makes the code hard to understand and debug. It is always possible to write a code without *goto* replacing it with *if else* and *break*:

```
double table1[5][5], table2[5][5];
int j;
for (int i = 0; i < 5; i++)
{
    for (j = 0; j < 5; j++)
    {
        if (table1[i][j] == 0)
            break; // jumps out of the inner loop
        else
            table2[i][j] = 1 / table1[i][j];
    }
    if (j != 5) // i.e. the inner loop ended abnormally
    {
        printf("Error: Table1 has zero elements\n");
        break; // jumps out of the outer loop
    }
}
```

Sometimes, however, it is considerable to use *goto* for jumping out from very long and complicated loop or *if else* bodies.

Multiple choice (1)

In some cases the *if else* statement can be replaced by *switch* statement:

```
switch (<expression_producing_integer>)  
{  
    case <integer_constant>: <statement>  
    .....  
    case <integer_constant>: <statement>  
    default: <statement>  
}
```

Example:

```
while (1)  
{  
    switch (_getche()) // produces one-byte integer  
    {  
        case 'v': CountVowels(); // if this integer is equal with 'v', call function CountVowels  
                break; // jump out of switch, continue the loop  
        case 'c': CountConsonants();  
                break;  
        case 'x': return;  
        default: break;  
    }  
}
```



// other than 'v', 'c' or 'x'

Multiple choice (2)

1. The expression in the parentheses following the keyword *switch* is evaluated. The result must be an integer (*char*, *int*, *long int*, etc.).
2. The list of labels formatted as *case* $\langle integer_constant \rangle$ is scanned.
3. If there is a label containing constant that matches the result of expression, the program jumps to this label and starts to execute the statements following it.
4. If there is no such a label, the program jumps to label *default*.
5. If the *default* label is not present, the program jumps to the statement following the *switch*.

```
switch (_getche())
{
    case 'v': CountVowels(); // as here is no break, the program calls CountVowels and
                          // after that CountConsonants.
                          // If the label was found, the program ignores all the other
                          // labels and runs until the end of switch statement. The break
                          // statement, however, forces the program to skip the
                          // following statements and jump out of switch.

    case 'c': CountConsonants();
              break;
    case 'x': return;
    default: break; // although not needed here, it is a good programming practice to
}                // include default into any switch
```

Multiple choice (3)

The place to jump may be marked with **multiple labels**. Example:

```
switch (_getche())
{
    case 'V':
    case 'v': CountVowels();
              break;
    case 'C':
    case 'c': CountConsonants();
              break;
    case 'X':
    case 'x': return;
    default: break;
}
```

The *switch* statement makes the code more readable. Therefore, the experienced software developers prefer, if possible, the *switch* statement to *if else*.

Type conversions (1)

Problem:

```
int i = 1;  
double d1, d2 = 1.5;  
d1 = d2 + i;
```

We have mixed types: $d1$ and $d2$ are double on 8 bytes, i is integer on 4 bytes. The addition is possible only between operands belonging to the same type. So, before the addition $d2$ must be converted into 4-byte integer or i into 8-byte floating point number. In the first case the fractional part of $d2$ is thrown away and the result is 2. **Conversion of floating point number to integer means that the fractional part is ignored.** In the second case the result is 2.5. Which of them?

Problem:

```
int k1 = 5, k2 = 2;  
d1 = d2 + k1 / k2;
```

The result may be 3.5 (if $k1$ and $k2$ are divided as integers) or 4 (if $k1$ and $k2$ are before division converted into *double*). Which of them?

Be very careful with expressions containing operands of different types. The result may be an unexpected value.

Type conversions (2)

If the expression includes operands of different types, **the compiler organizes automatic type conversions** according to the following rules:

If the expression contains operands of type *signed char* and *signed short int*, they are converted into type *signed int*, i.e. **widened from 1 or 2 bytes to 4 bytes**. Similarly, operands of type *unsigned char* and *unsigned short int* are converted into type *unsigned int*:

```
unsigned short s1 = 1000, s2 = 2000;
```

```
unsigned int u1, u2 = 100000;
```

```
u1 = s2 + s1 + u2; // as the first step, s1 and s2 are widened to 4-byte variable
```

After the first step, for each binary operation involving two different types, **the value of lower ranking is converted into the value of higher ranking**. The ranking from high to low is: *long double*, *double*, *float*, *unsigned long long int*, *signed long long int*, *unsigned long int*, *signed long int*, *signed int*. As seen, the types of higher ranking are with wider range – thus we avoid the losing of information.

In assignment, the final result of expression is converted to the type of lvalue.

Type conversions (3)

An expression may include constants. In that case:

Floating point constants (like $1.5e5$ or 0.125) are of type *double*. In C++: would you need to handle them as *float* values, complete them with **suffix F (or f)**, for example $0.125F$.

Integer constants in decimal numeral system (like 125) are of type *signed int*. If they are out of range of *signed int*, they are handled as *signed long int*, *signed long long int* or *unsigned long long int*.

Integer constants in hexadecimal or octal numeral systems (like $0xFE$ or 0125) are also of type *signed int*. If they are out of range of *signed int*, they are handled as *unsigned int*, *signed long int*, *signed long long int* or *unsigned long long int*.

If an integer constant due to its value belongs to type *signed int* but the programmer needs *signed long int*, the integer must be completed with **suffix L or l** (like 125L). In Visual Studio it is useless because *int* and *long int* are identical. In C++: **suffix U or u (LU or lu)** means that the constant must be handled as unsigned int (unsigned long int).

Character constants (like 'A') are of type *signed char*.

Remark the a **constant cannot be negative**. Statement

```
int x = -5;
```

means that we apply unary sign converting operation to constant 5.

Type conversions (4)

Examples:

```
double a, b = 10; // assignment, 10 as integer is converted into double 10.0
```

```
int c = 5, d = 2;
```

```
a = c / d * b; // c and d are divided as integers and the result is 2. This result is converted  
              // into double and multiplied by b. The final result is 20.0
```

```
a = c / d; // the final result is 2.0 because c and d are divided as integers
```

```
a = c / b; // the final result is 0.5 because before division c is converted to 5.0
```

```
a = c % b; // error because before division c is converted to 5.0, but modulus for real point  
          // numbers is not defined
```

```
double x = 3.56;
```

```
int n = 10, m = 7;
```

```
int k = x * n - (2 * n) / m; // A = x * n = 35.6, n is converted to double
```

```
                        // B = (2 * n) / m = 2, operations between integers
```

```
                        // A - B = 33.6, B is converted into double
```

```
                        // k = 33, during assignment the result is converted to int
```

Type conversions (5)

Assignment

```
int i;
```

```
double d = 5.5;
```

```
i = d; // depending on the compiler you may get an error message or warning
```

causes lose of information, because the fractional part of d is ignored and i gets value 5.

If we, despite of possible lose of information, need conversion from higher ranking to lower ranking, we have to apply the unary **cast operator**:

```
(<new_type>) <variable_or_expression>
```

Examples

```
int i;
```

```
unsigned u;
```

```
double d;
```

```
i = (int)d; // now it is correct
```

```
i = (int)u; // the range of unsigned int is wider, therefore lose of information is possible
```

Casting may be also used inside expressions to force the computer to ignore the automatic type conversion rules, for example:

```
double d;
```

```
int i = 5, j = 2;
```

```
d = i / (double)j; // the final result is 2.5 because i and j are now divided as doubles
```

Type conversions (6)

Type conversion (automatic or **implicit** as well as with casting or **explicit**) between integers is performed as follows:

1. Widening or promotion:

- a. If the old type is unsigned, simply the needed number of zero bits are added.

For example:

```
unsigned char uc = 10;
```

```
unsigned int ui = uc; // now on 4 bytes, the result is still 10
```

```
int i = uc; // now on 4 bytes, the result is still 10
```

- b. If the old type is signed and the new type is also signed, the needed number of zero bits are added but the **sign is copied** from the original value. For example:

```
char c = -10;
```

```
int i = c; // now on 4 bytes, the result is still -10
```

- c. If the old type is signed and the new type is unsigned, the results may seem **very strange**. For example:

```
char c1 = 10, c2 = -10;
```

```
unsigned int ui1, ui2;
```

```
unsigned int ui1 = c1; // now on 4 bytes, the result is still 10 (c1 was positive)
```

```
unsigned int ui2 = c2; // now on 4 bytes but the result is 4294967286 (c2 was
```

```
// negative); converting a negative value to unsigned is of course stupidity,
```

```
// but you are allowed to do it without any warning! The software developer
```

```
// is responsible!
```

Type conversions (7)

2. Narrowing or demotion:

When converting from a wider type to a narrower type, the rule is very rough: bits of the wider type that do not fit into the new narrower type are simply dropped. **We may lose information and the results may seem very strange.** Examples:

```
int i1 = 10, i2 = 300, i3 = -10;
```

```
char c1 = (char)i1; // still 10 because 10 is in the range of char
```

```
char c2 = (char)i2; // we get 44; as 300 is not in the range of char, we have lost 3 bytes
```

```
unsigned char uc = (unsigned char)i3; // we get 246; this was a senseless converting  
// but C is not responsible – you may do what  
// you want!
```

3. From signed to unsigned and vice versa on the same number of bytes:

There is no taking the absolute value. The rule is very rough: nothing is changed in the bit pattern but the interpretation of bits corresponds to the new type. **The results may seem very strange.** Examples:

```
char c1 = 10, c2 = -20;
```

```
unsigned char uc1 = (unsigned char) c1; // still 10
```

```
unsigned char uc2 = (unsigned char) c2; // we get 236
```

```
unsigned char uc3 = 250;
```

```
char c3 = (char)uc3; // we get -6
```

Type conversions (8)

What to remember about converting of integers:

1. When you convert from a narrower signed type to a wider unsigned type, the compiler widens the original memory with bits filled with zeroes and the value of the object might change.
2. When you convert from a narrower signed type to a wider signed type, the compiler widens the original memory with bits filled with zeroes without changing the sign and the value of the object is preserved.
3. When you convert from a narrower unsigned type to a wider type, the compiler widens the original memory with bits filled with zeroes and the value of the object is preserved.
4. When you convert from a wider type to a narrower type, the compiler truncates the original memory field and the value of the object might change.
5. When you convert between signed and unsigned types of the same width, the compiler effectively does nothing, the bit pattern stays the same and the value of the object might change.

Converting from **integer to double**: the value is reformatted but kept.

Converting from **double to integer**: the fraction part is deleted, value of the object may changed.

Type conversions (9)

Exercises: what are the results?

1. `int i = 100 / 3;`
2. `double d = 100 / 3;`
3. `int i = 100 / 3.0;`
4. `double d = 100 / 3.0;`
5. `int j = 100, k = 3;`
`int i = (double)j / k;`
`double d = j / (double)k;`
6. `double d = 3.2, x;`
`int i = 2, y;`
`x = d * (y = ((int)2.9 + 1.1) / d) + 1;`
7. `double x = 3.5, y = 2.5, z = 4.5, w;`
`int i = 2, j = 10;`
`w = 2 * x * y * z * i / j; // by Windows calculator = 15.75`
`w = 2 * i / j * x * y * z;`
`w = (double)(2 * i / j * x * y * z);`
`w = (double)2 * i / j * x * y * z;`

Overflow and underflow

If an operation with floating point numbers produces a result that does not fit into the range of the current type, the program crashes. This is the **overflow**.

The floating point value may be zero but cannot be too close to zero:



The reason is that the field of bits reserved for exponent (11 bits in case of *double*) can store numbers from a limited range. Thus, a double value may be in interval $-1.7 * 10^{308} \dots -1.7 * 10^{-308}$ and $+1.7 * 10^{-308} \dots +1.7 * 10^{308}$. If an operation with floating point numbers produces a result falling into **underflow gap**, this result is in most cases replaced by zero. This may lead to unpredictable distortion in the final results of a calculation.

There is no overflow detection in operations with integers:

```
unsigned int i = 4294967278; // 0xFFFFFFFFEE
```

```
unsigned int j = i * 2; // the result is 4294967260, that is wrong. The actual result is  
// 8589934556 which is out of the range of unsigned int.
```

```
// Without regarding to the error, the program continues to run
```

```
unsigned long long int k = (unsigned long long int)i * 2; // produces the correct result
```

This is the **responsibility of software developer** to foresee the possible overflow and underflow.

Compound assignments

Expression

```
x = x + y;
```

may be also written as

```
x += y; // addition assignment operator
```

There are also:

```
x -= y; // or x = x - y, subtraction assignment operator
```

```
x *= y; // or x = x * y, multiplication assignment operator
```

```
x /= y; // or x = x / y, division assignment operator
```

```
x %= y; // or x = x % y, modulus assignment operator
```

Left of compound assignment operator must be a lvalue, right of it may be any expression.

Example:

```
int sequence[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
for (int i = 0; i < 9; i++)
```

```
    sequence[i] += rand(); // add a random value to the each member
```

Exercise: what does the following code snippet prints?

```
int i = 1, j = 1, k = 1;
```

```
i += j += k;
```

```
printf("%d %d %d\n", i, j, k);
```

Conditional expression

Statement

```
if (i > 10)
```

```
    x = 1;
```

```
else
```

```
    x = -1;
```

may be written as

```
x = i > 10 ? 1 : -1;
```

Generally, the **conditional expression** is

```
<condition_expression> ? <expression_1> : <expression_2>
```

If the *<condition_expression>* produces TRUE, the program executes *<expression_1>* ; if FALSE, then *<expression_2>*.

Conditional expression is ternary (neither unary nor binary) because it has 3 operands.

Conditional expression helps us to write short and effective code. For example:

```
int i, j, max;
```

```
max = i >= j ? i : j;
```

Exercise: what does the following code snippet prints?

```
int i = 1, j = 1, k;
```

```
k = i < j ? i++ : j++;
```

```
printf("%d %d %d\n", i, j, k);
```

Operator precedence (1)

Precedence	Operator	Description	Associativity
1	++ and -- () []	Increment and decrement, postfix Function call Reading element from array	Left -> Right
2	++ and -- - ! (type)	Increment and decrement, prefix Sign conversion Logical NOT Type cast	Right->Left
3	* / %	Multiplication Division Modulus	Left -> Right
4	+ -	Addition Subtraction	Left -> Right
5	<= < >= >	Less or equal Less Greater or equal Greater	Left -> Right
6	== !=	Equal Not equal	Left -> Right

Operator precedence (2)

Precedence	Operator	Description	Associativity
7	&&	Logical AND	Left -> Right
8		Logical OR	Left -> Right
9	?:	Conditional	Right->Left
10	= += -= *= /* %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right->Left
11	,	Comma	Left -> Right

This table includes only the operators we already know. The complete table is on https://en.cppreference.com/w/c/language/operator_precedence.

Operator precedence (3)

If an expression contains operators with different precedences, the operators with higher precedence are performed first. For example in expression:

```
x = sqrt(y) + z * w;
```

the call to function has precedence 1 and is performed first. Multiplication has precedence 3 and is performed after that. Then addition follows. Assignment has the lowest precedence and is performed last. To change the default precedence use parentheses. For example in expression:

```
x = (sqrt(y) + z) * w;
```

the order is call to function, addition, multiplication, assignment.

According to this table we may write

```
if (a == b && c < d) {....}
```

because here the logical AND has the lowest precedence. But we cannot write

```
while (text[i] = _getche() != '\r') {.....}
```

because the output of `_getche` is compared with `'\r'` before assignment, i.e. the array will be filled with TRUE and the last element will be FALSE. The correct expression is:

```
while ((text[i] = _getche()) != '\r') {.....}
```

Associativity is used when two operators of same precedence appear in an expression:

```
a = b = c + d;
```

the order here is: addition, assignment to *b*, assignment to *a*.

If you are not sure, do not think long: use parentheses.