What is programming?

Computer programming, also known as coding, is the process of creating software. In other words, computer programming means the writing of instructions for a computer to perform.

For example, if we have two sets of some data and we need to create from them a third set of data, we have to write the instructions for the following operations:
•Loading the first set of data into the memory of computer.
•Loading the second set of data into the memory of computer.
•Creating the third set of data (for example by selecting some units of data from the both initial sets and merging them into more complicated units for the resulting set).
•Saving the resulting set in a disk file.
•Reporting about the successful end (or failure) on the screen.

These instructions may be written even by a simple text editor. The set of instructions is called as source code. Of course, it cannot be not written in English or Estonian. We have to use some specific language with strictly specified syntax and semantics that the computer is able to undestand. There are several thousands of programming languages but only some of them are widely known and used. In this course we'll base on language called "C".

Integrated development environment

The source code can be written using simple text editors like Notepad. In C the filename extension of source code file is *.c or *.cpp. But this is just the first step in the software developing process. To get a program that can run (i.e. the executable program packed into an *.exe file) we need specific tools that are able to create executable from the initial *.c or *.cpp file(s). In other words, we need tools to build the program.

At most cases, the first versions of a program contain errors. The process of finding and fixing errors is called debugging. For that we need specific tools called debuggers.

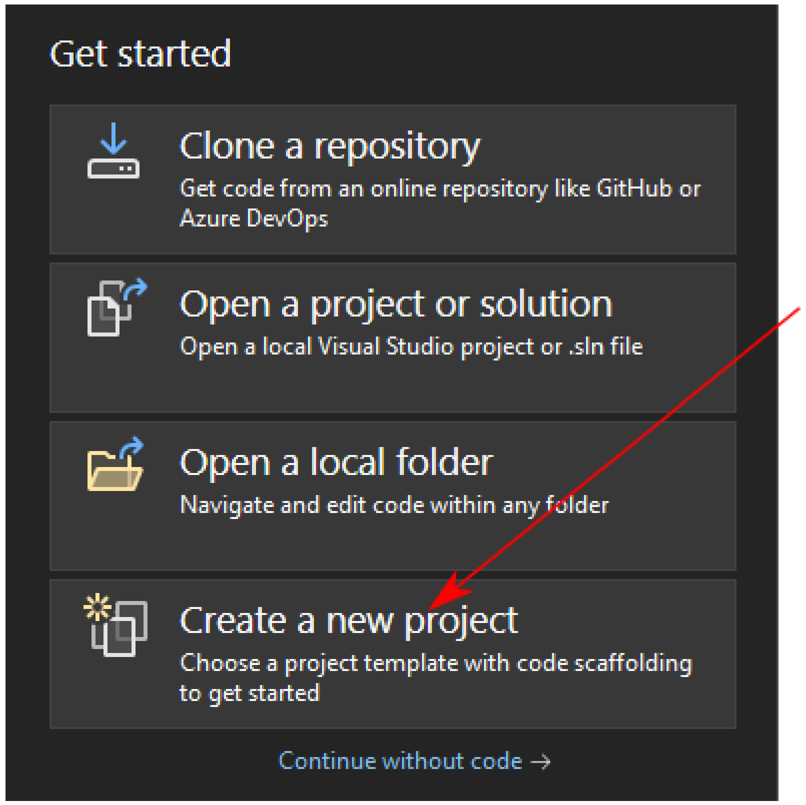
Integrated development environment (IDE) is a very large and complicated software application providing almost all the facilities a programmer needs for software development: text editor, building tools, debuggers, etc. In our course we use IDE called Microsoft Visual Studio 2022.

Advice: open https://visualstudio.microsoft.com/downloads/ and install into your laptop freeware Microsoft Visual Studio Community Edition 2022**.** It is the simplified version of Visual Studio. After installation create shortcut for "C:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\devenv.exe". Clicking on this shortcut launches the IDE.

Some other popular IDE-s for programming in C: Code::Blocks, Eclipse, Qt Creator, NetBeans, CLion.

My first program (1)

Launch Visual Studio



Get started

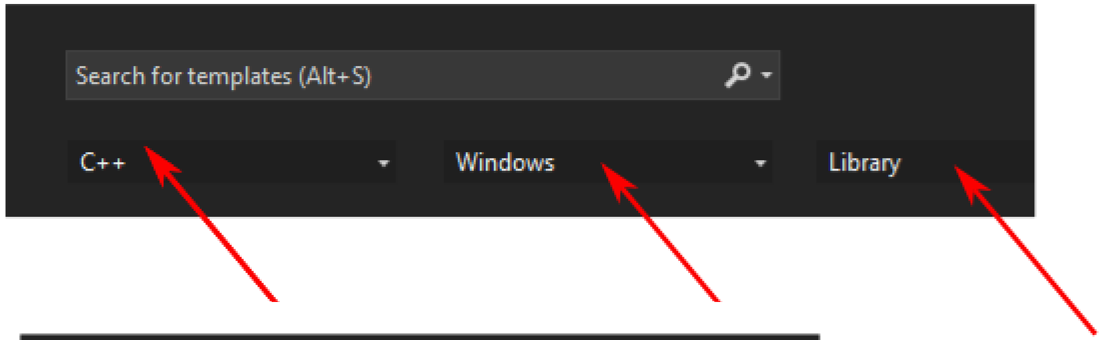
Clone a repository
Get code from an online repository like GitHub or Azure DevOps

Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

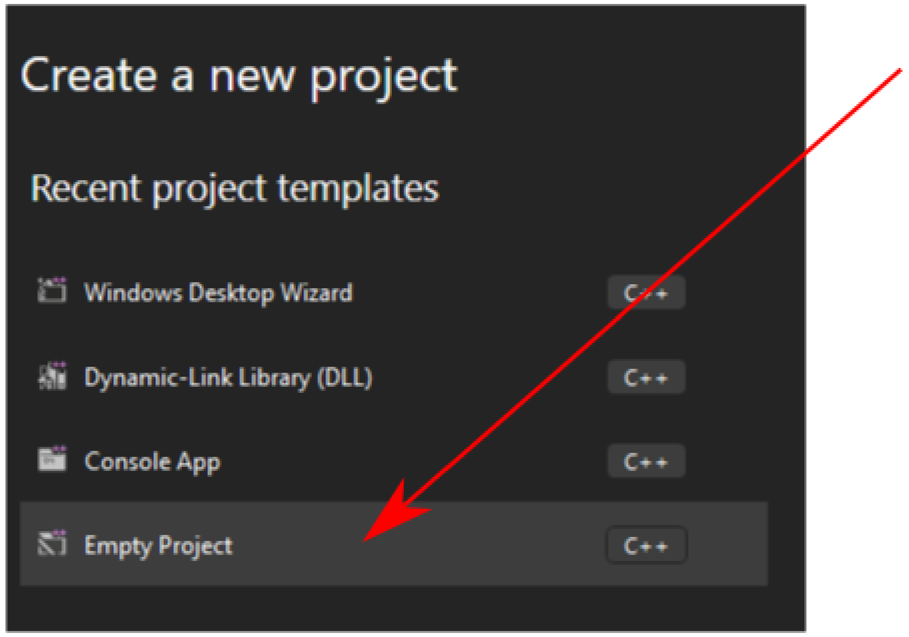
Create a new project
Choose a project template with code scaffolding to get started

Continue without code →



Search for templates (Alt+S)

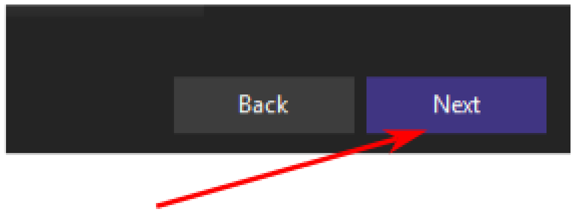
C++ Windows Library



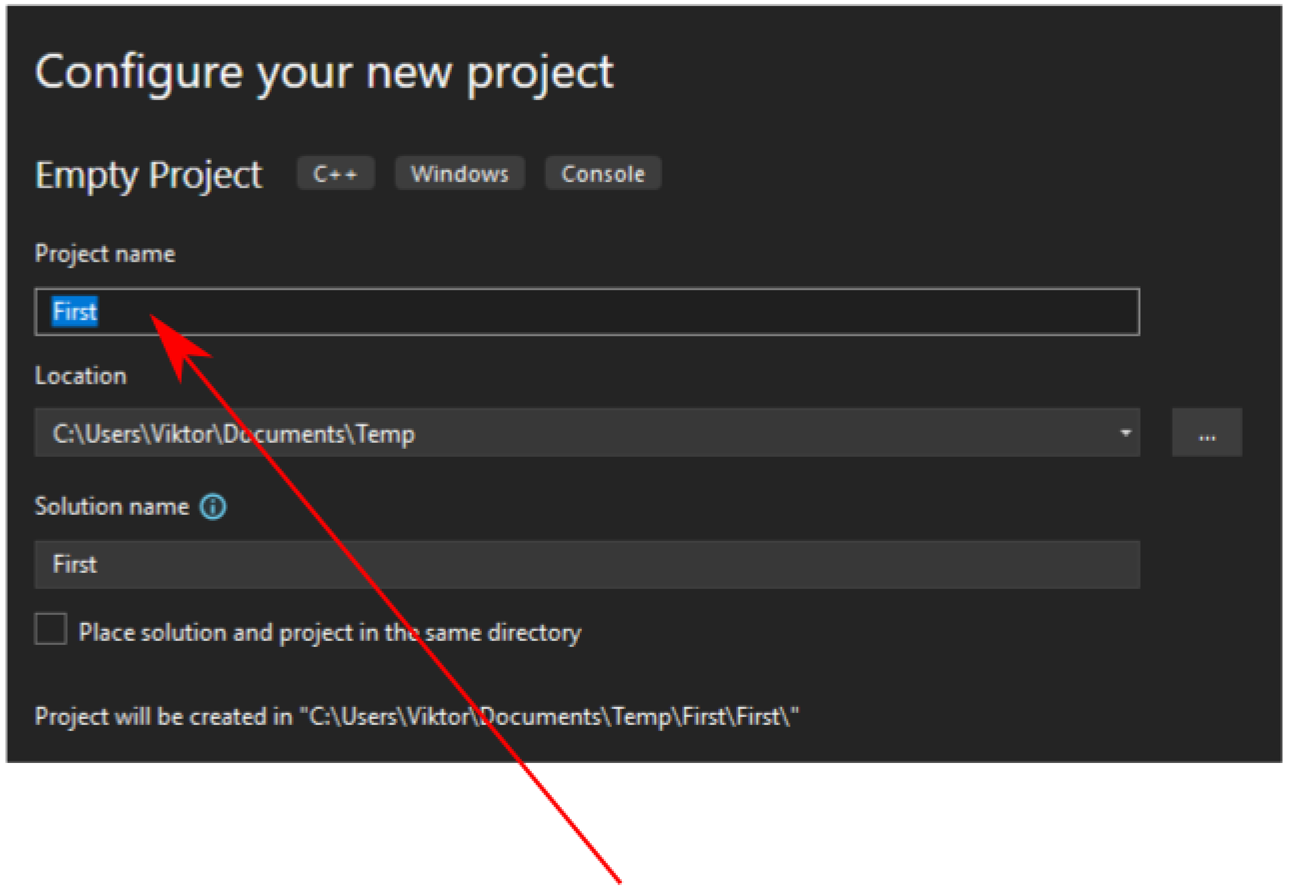
Create a new project

Recent project templates

Windows Desktop Wizard	C++
Dynamic-Link Library (DLL)	C++
Console App	C++
Empty Project	C++



My first program (2)


Configure your new project

Empty Project C++ Windows Console

Project name

First

Location

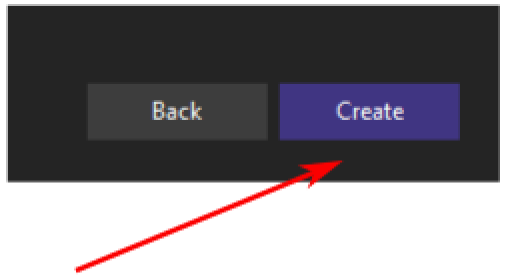
C:\Users\Viktor\Documents\Temp

Solution name

First

Place solution and project in the same directory

Project will be created in "C:\Users\Viktor\Documents\Temp\First\First\"

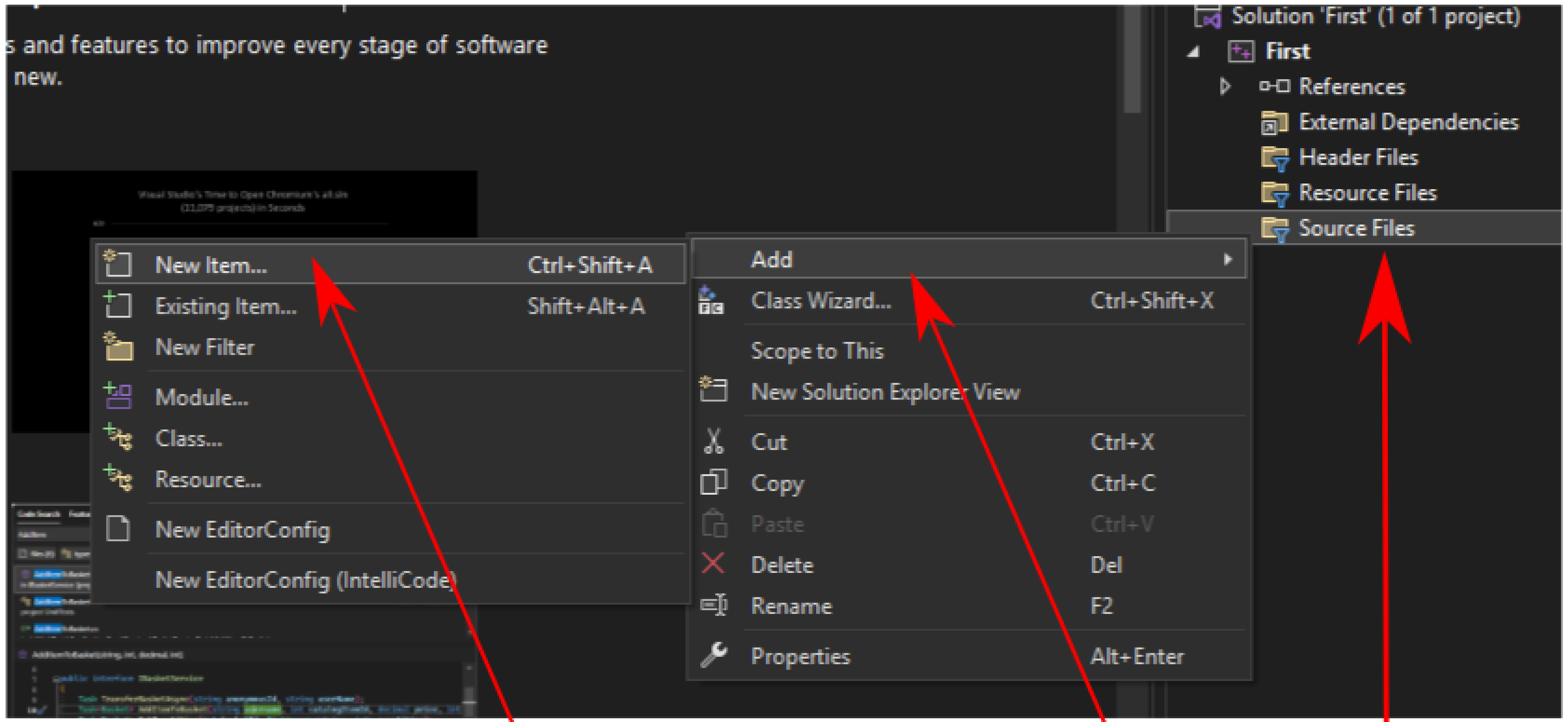


Type the project name (*First*). In our case the solution name is set automatically (also *First* is acceptable.

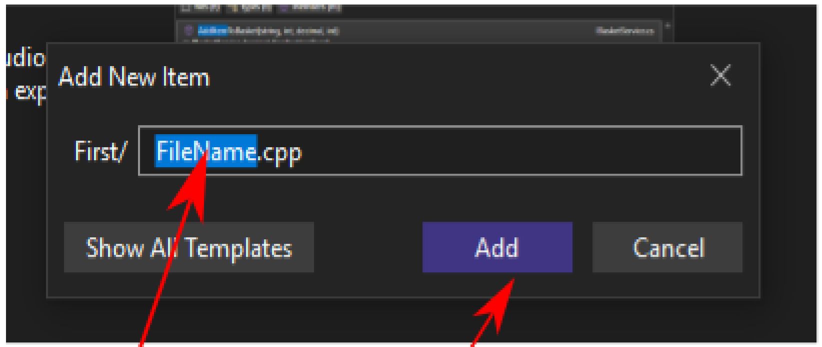
Use the location selected by Visual Studio wizard. But write it down on a piece of paper. Later you need it for opening the project folder and if you do not remember the location, you will have trouble to find it.

Do not get confused: C is the sybset of C++.

My first program (3)



right click

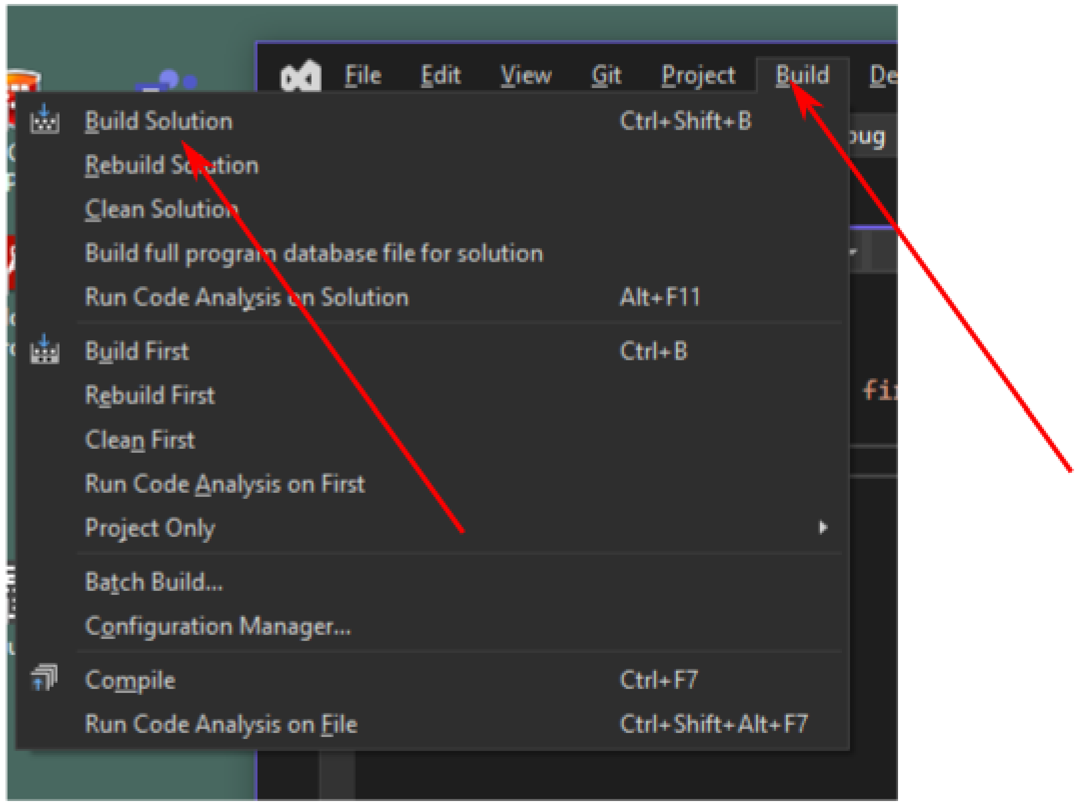


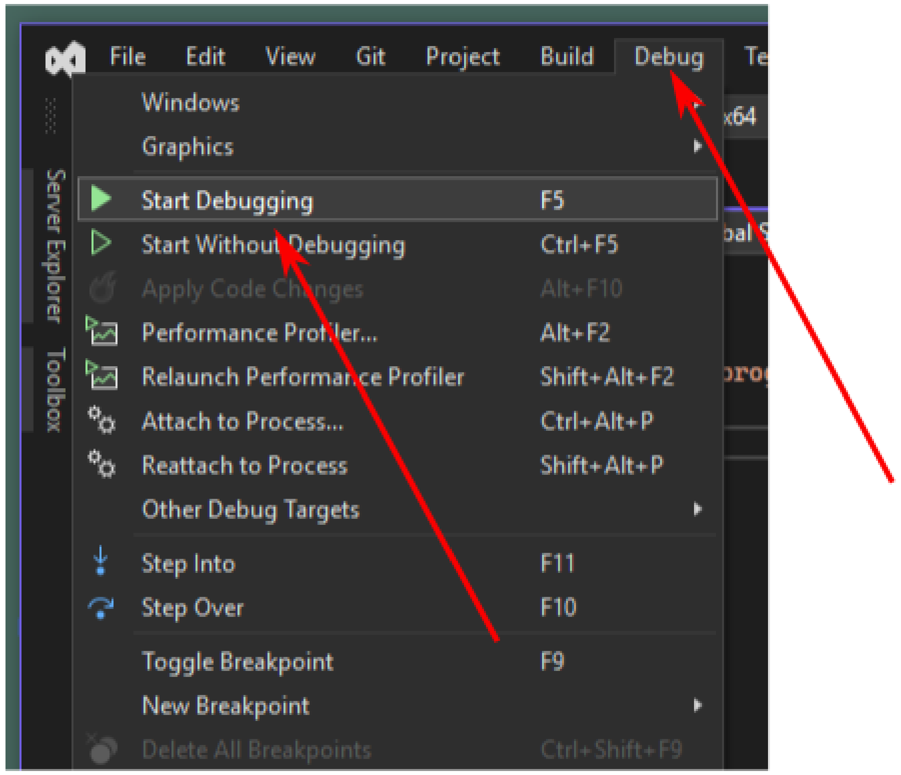
Set *Main.cpp* as the source file name

My first program (4)

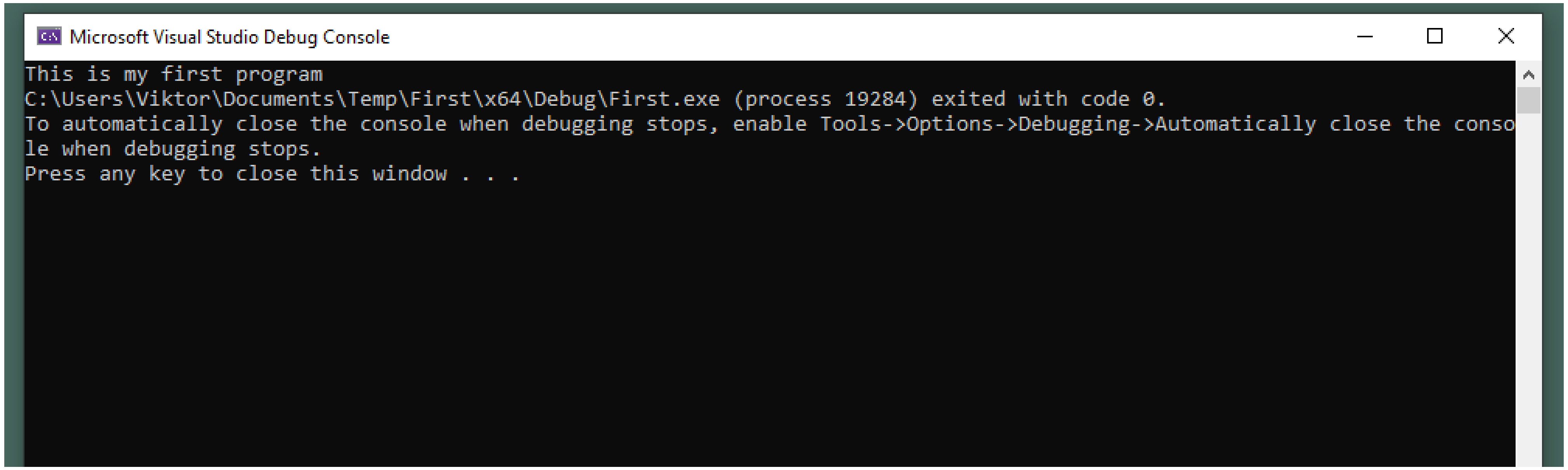
Type the code:

```
#include "stdio.h"
int main()
{
  printf("This is my first program");
  return 0;
}
```


File Edit View Git Project Build
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Build full program database file for solution
Run Code Analysis on Solution Alt+F11
Build First Ctrl+B
Rebuild First
Clean First
Run Code Analysis on First
Project Only
Batch Build...
Configuration Manager...
Compile Ctrl+F7
Run Code Analysis on File Ctrl+Shift+Alt+F7


File Edit View Git Project Build Debug
Windows
Graphics
Start Debugging F5
Start Without Debugging Ctrl+F5
Apply Code Changes Alt+F10
Performance Profiler... Alt+F2
Relaunch Performance Profiler Shift+Alt+F2
Attach to Process... Ctrl+Alt+P
Reattach to Process Shift+Alt+P
Other Debug Targets
Step Into F11
Step Over F10
Toggle Breakpoint F9
New Breakpoint
Delete All Breakpoints Ctrl+Shift+F9
Server Explorer
Toolbox

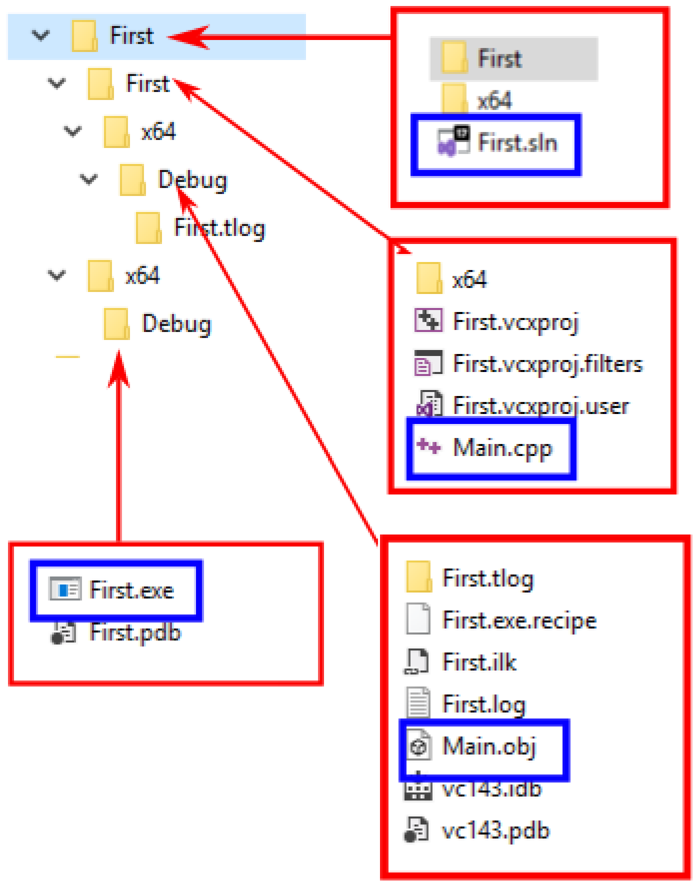
My first program (5)


Microsoft Visual Studio Debug Console

```
This is my first program
C:\Users\Viktor\Documents\Temp\First\x64\Debug\First.exe (process 19284) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

This is the result in command prompt (or console) window.

My first program (6)



We created only one file: *Main.cpp.* All the other files and folders are created automatically by Visual Studio wizard according to our initial settings (C++, empty project, etc). The location of *Main.cpp* is also determined by wizard. Do not try to change the location of files: Visual Studio will get confused and the results are unpredictable.

By terms of Visual Studio, we have now solution named *First*. The solution file *.sln contains the general description of solution.

By default, the solution is configured into debug mode. It means that the executable file contains a lot of additional information helping to find and fix errorrs (and is therefore very long).

When we suppose that there are no more errors, we may reconfigure the solution to the release mode. Then the wizard creates several new folders and the build tools generate a new and shorter *.exe without auxiliary information needed for debugging.

My first program (6)

```
#include "stdio.h" // header file handled by preprocessor
int main() // function header
{  // the function body begins
  printf("This is my first C-program");
  return 0;
} // the body function ends
```

A program may consist of thousands of instructions and may be written by several programmers. Therefore the source code is divided into sections. In C these sections are called functions.

A C function consists of header and the following to it body. The body is enclosed in braces { }.

The body includes statements (here we have 2 statements). Each statement must be terminated with semicolon ;

The function header specifies the function name. The name of our function is *main*.

An experienced programmer always completes his source code with comments (various remarks, explanations, etc.). The comments are for humans and the building tools ignore them. An one-line comment starts with //. Comments consisting of several lines must start with /* and end with */.

My first program (8)

```
#include "stdio.h"
int main()
{
  printf("This is my first C-program");
            // main is the calling function
            // printf is the called function
            // "This is my first C-program" is the parameter for printf specifying the task
  return 0;
}
```

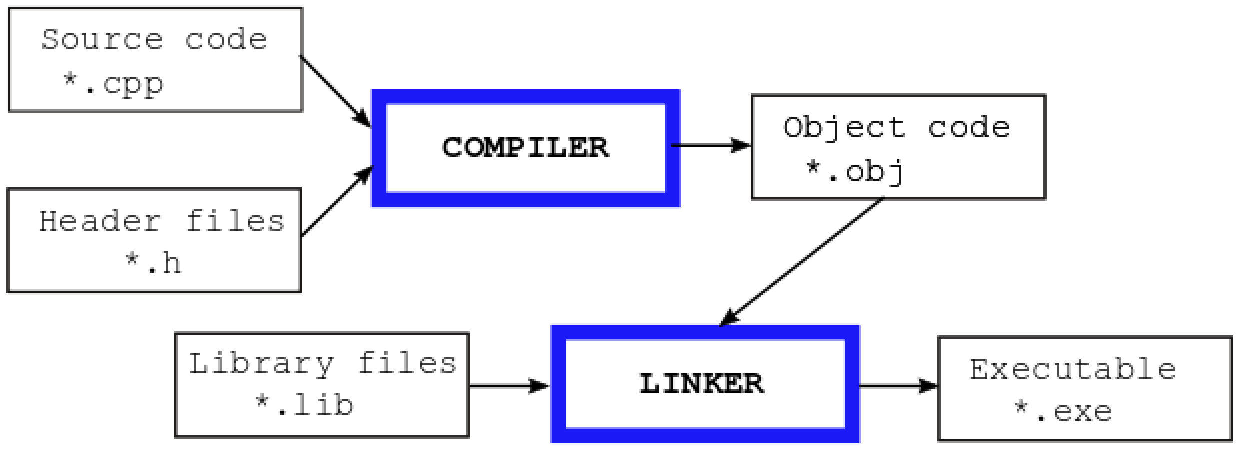
A function can call some other function. It means that a function asks another function to perform some operations or calculate some values and send back the results.

Our program actually consists of two functions: *main* and *printf*. The *main* is written by us. The *printf* is the standard part of Visual Studio. Here the *main* orders *printf* to print in the command promt window the text presented in parentheses.

When *main* reaches the line containing the call, the control is turned over to *printf*. When *printf* has finished its task, the control is returned to the calling function.

The calling function must exactly specify the task that the called function has to perform (currently, what to print). Therefore, the call starts with the name of called function followed by parameters within the parentheses.

My first program (9)



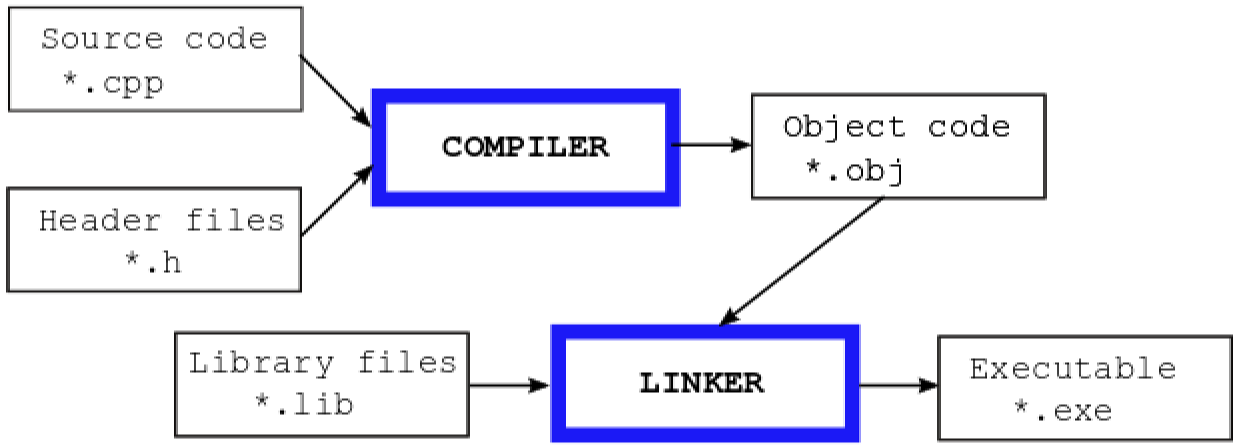
The building has two stages:

1. Compiling – the compiler checks the correctness of source code and creates the object code.
2. Linking – the linker connects the different parts of program into one executable.

```
#include "stdio.h"
int main()
{
  printf("This is my first C-program");
  return 0;
}
```

The computer runs programs that are in machine code. The compiler transforms the source code written in C into machine code. But to do this, it needs information about standard functions (currently, *printf*) – for example, what are the allowed parameters. This information is stored in the standard header files (for *printf* in file *stdio.h*). Therefore the compiler at first preprocesses the source code, including the *.h files specified by programmer into the *.cpp file.

My first program (10)



```
#include "stdio.h"
int main()
{
  printf("This is my first C-program");
  return 0;
}
```

The standard functions like *printf* are already compiled. Their object code files (*.obj) are stored in standard library files. The library files as well as the header files are the standard part of Visual Studio. The linker extracts the needed standard functions from the libraries and joins them with object code compiled from source code written by us.

Keywords

Integrated Development Environment (IDE)
Wizard
Solution *.sln
Source code file *.c or *.cpp
Object code file *.obj
Header file *.h
Library file *.lib
Executable file *.exe
Machine code
Build – preprocessor, compiler, linker
Debug – breakpoint
Debug mode and release mode
Function – header and body, braces {…}, function name
Statement, semicolon
Comments, //… and /*…*/
Call to function, function parameters, parentheses (…)

My second program (1)

```
#include "stdio.h" // to describe standard functions printf and gets_s
#include "stdlib.h" // to describe standard function atoi
int main()
{
  int x, y, z;
  char line[81];
  printf("Type the first integer: ");
  gets_s(line);
  x = atoi(line);
  printf("Type the second integer: ");
  gets_s(line);
  y = atoi(line);
  z = x + y;
  printf("The sum is %d", z);
  return 0;
}
```

#include followed by *.h file name in quotation marks is the preprocessor directive. The number sign # must be the first character in the directive row. This row may contain only one directive (which may be followed by comment). The directive decribing a standard function must be located before the first usage of this function.

Positional numeral systems (1)

Our everyday decimal numeral system uses only 10 digits: 0, 1, … , 9 (the radix is 10). But as the system is positional, we may write down any integers: in value "3" digit 3 presents "ones", in value "37" digit 3 presents "tens", in value "373" digit "3" presents "hundreds" and also "ones": 37 = 3*10 +7*1 and 373 = 3*100 + 7 *10 + 3*1.

But the radix may be greater or lesser than 10. In the hexadecimal system the radix is 16 and the digits are 0, 1, 2, …, 9, A, B, C, D, E, F. Consequently:

$0_{10} = 0_{16}$,…., $9_{10} = 9_{16}$, $10_{10} = A_{16}$, $11_{10} = B_{16}$, $12_{10} = C_{16}$, $13_{10} = D_{16}$, $14_{10} = E_{16}$, $15_{10} = F_{16}$, $16_{10} = 10_{16}$, $17_{10} = 11_{16}$,…., $255_{10} = FF_{16}$,…., $4096_{10} = 1000_{16}$,….

In the binary system the radix is 2 and the digits are 0 and 1. Consequently:

$0_{10} = 0_2$, $1_{10} = 1_2$, $2_{10} = 10_2$, $3_{10} = 11_2$, $4_{10} = 100_2$, $5_{10} = 101_2$, $6_{10} = 110_2$, $7_{10} = 111_2$, ,….

The easest way to convert is to use the Windows or mobile phone calculators set to programmer's mode.

Az $16 = 2^4$, converting from hexadecimal to binary is very simple: replace each hexadecimal digit with 4 binary digits. And vice versa: divide the binary digits into groups of 4 and replace the groups with the proper hexadecimal digit. For example $123_{10} = 7B_{16} = 0111 \mid 1011_2$

Each software engineer must know without any aid resource that:

Positional numeral systems (2)

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Bits, bytes, nibbles and words

The basic unit of memory is bit. It can hold only two values: 0 or 1 (i.e. a binary digit).

The byte consists of 8 bits. The smallest integer that can be stored in a byte is 0000 | 0000 or simply 0. The biggest integer that can be stored in a byte is 1111 | 1111 or FF_{16} or 255_{10}

Nibble consists of 4 bits. We may say that a byte contains two nibbles.

Word is the natural unit of memory for the given processor. For 32-bit processor the word consists of 32 bits. For 64-bit processor the word consists of 64 bits.

To handle data stored in the computer memory we need to know, exactly where it is located or in other words, what is the address of our data. The smallest addresable unit is byte.

Do not forget that a kilobyte (KB) is 1024 (and not 1000) bytes. Similarly, a megabyte (MB) is 1024*1024 bytes (1024 KB) and so on. However, some manufacturers do not follow those definitions and define a KB as 1000 bytes.

Integers (1)

If our program uses integers from 0 to 255, we may store each integer on a separate byte. But if we work with integers out of this range, we need fields consisting of two or even more bytes:

Integers from range 0… 65,535 need a field of 2 bytes.

Integers from range 0…4,294,967,295 need a field of 4 bytes.

Generally, if we have a field of n bits, the greatest integer we can store on this field is 2^{n-1}.

C statement

```
unsigned char c;
```

means that our program needs one byte memory for storing integers from range 0…255. In the following statements, *c* is the identifier of this byte. In terms of C, we define a variable of type *unsigned char*. In statement

```
unsigned char c1, c2, c3;
```

we declare 3 variables of type *unsigned char* with names *c1*, *c2* and *c3*.

C statements used for specifying variables are called declarations.

To store a value on byte *c*, we have to write an assignment statement, for example

```
c = 10;
```

Integers (2)

Do not forget that declaration

```
unsigned char c;
```

does not initialize variable *c*, i.e. the byte we have got contains some occasional value. But if we need, we may write the initial value right into the declaration:

```
unsigned char c = 10; // declaration + initialization
```

or

```
unsigned char c1 = 10, c2 = 20, c3;
c3 = c1 + c2;  // c3 value is now 30
```

You must at first declare a variable (i.e. allocate memory for it) and initialize it or write a statement to assign to it a value. Only after that you may use this variable.

```
unsigned char c1 = 10, c2, c3;
c3 = c1 + c2;  // error, c2 is neither initialized nor never been assigned a value
c3 = c1 + c4; // error, variable c4 is not declared
```

If we use integers out of range 0…255, we have to use types *unsigned short int*, *unsigned int*, *unsigned long int*, *unsigned long long int:*

```
unsigned short int si; // memory field of 2 bytes
unsigned int i; // memory field of 4 bytes
unsigned long int li; // memory field of 4 bytes
unsigned long long int lli; // memory field of 8 bytes
```

Integers (3)

Values of variables of unsigned types may be only positive integers. If we have also negative integers, we must use signed types:

signed char c; // one byte
signed short int si; // memory field of 2 bytes
signed int i; // memory field of 4 bytes
signed long int li; // memory field of 4 bytes
signed long long int lli; // memory field of 8 bytes

Keyword *signed* may be omitted.

As the memory consists of bytes and the bytes are sequences of bits and bits may be in state "0" or "1", we have only one possibility to store the sign: to agree that one of the bits presents the sign (for example, if this bit is 0, the integer is positive and if 1, the integer is negative. But in case of one-byte integers we have now only 7 bits for the value. Consequently the ranges of signed variables and unsigned variables are different:

signed char: – 128…+127
signed short int: -32768…+32767
signed int and *signed long int*: -2147483648…+2147483647

Arrays

Declaration

```
char c;
```

gives us one byte for storing integers from range -128...+127. Declaration

```
char mc[10];
```

gives us a sequence of 10 bytes. Each byte can be used for storing an integer from the mentioned range. This sequence of bytes is called as array. Here, *mc* is the name of array and *10* is the number of members or the dimension of array. Similarly

```
int mi[100];
```

gives us a sequence of 400 bytes. Each 4-byte field can be used for storing a variable of type *int*.

To declare an array, one has to specify the type of members, the name of array and the dimension. The dimension (enclosed in brackets) must be a constant (not variable).

To access a member of array, write the array name and the ordering number (index) of the member. The index is also closed in brackets, for example:

```
mc[0] = 10;
mi[10] = 1000;
```

The index of the first member is always 0. The index of the last member is always dimension -1. Members of *mi* are *mi[0]*, *mi[1]*, …., *mi[99]* but not *mi[100]*.

ASCII characters and strings (1)

As the memory consists of bytes and the bytes are sequences of bits and bits may be in state "0" or "1", we have no possibility to store characters (letters, numbers, etc.) directly. We can store only numerical values. Consequently, we need to have an encoding tabel, replace characters in text with their encoded values (integers) and store the text as an array of integers.

In the ASCII (American Standard Code for Information Interchange) character encoding standard characters are encoded with one-byte integers. For example, *I* is 73, *C* is 67, *T* is 84, *1* is 49, *2* is 50 and *hyphen* is 45. So, text *ICT-121* in memory is

```
char ict121[8];
ict121[0] = 73; ict121[1] = 67; ict121[2] = 84; ict121[3] = 45; ict121[4] = 49;
ict121[5] = 50; ict121[6] = 49; ict121[7] = 0;
```

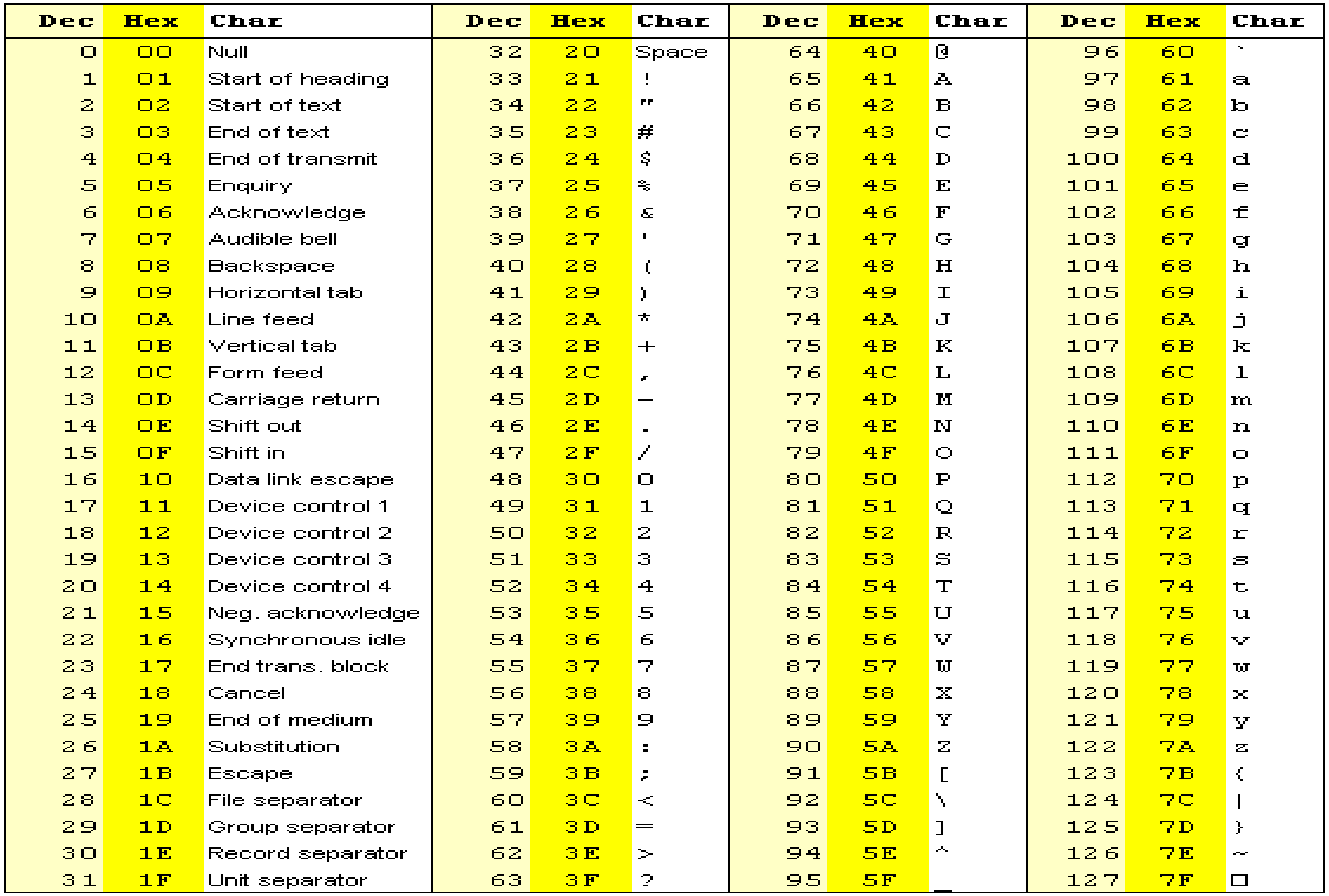
Terminating zero at position 7 marks the end of text. In terms of C *ict121* is a string – array of one-byte integers encoded by ASCII table and terminated with byte containing zero.

You may find the full ASCII table at http://www.asciitable.com/ .

It is very unconformtable to dig all time in ASCII table. There is a simpler way:

```
char ict121[8];
ict121[0] = 'I'; ict121[1] = 'C'; ict121[2] = 'T'; ict121[3] = '-'; ict121[4] = '1';
ict121[5] = '2'; ict121[6] = '1'; ict121[7] = 0;
```

ASCII characters and strings (2)



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

ASCII characters and strings (3)

char c1 = 1, c2 = 2, c3;
c3 = c1 + c2; // c3 is now 3
int i1 = 1, i2 = 2, i3;
i3 = i1 + i2; // i3 is also 3

The difference is only that *char* variables occupy one byte of memory, *int* variables four byte of memory.

But

char c4 = '1', c5 = '2', c6;
c6 = c4 + c5; // c6 is now 49 + 50 = 99

'1', *'2'*, *'A'*, *'a'*, *'+'*, etc. are character constants. The compiler replaces them with the corresponding integeres from ASCII table.

More examples:

char c7 = ' '; // space
char c8 = c7 + 50; // c8 is now 32 + 50 = 82
char c9 = '1' + '2'; // c9 is now 99
char c10 = 'b' - 'a'; // c10 is now 98 – 97 = 1
char c11 = 'q' + 1; // c11 is now 113 + 1 = 114

My second program (2)

```
#include "stdio.h" // to describe standard functions printf and gets_s
#include "stdlib.h" // to describe standard function atoi
int main()
{
 int x, y, z; // declare three variables (4-byte signed integers) without initialization
 char line[81]; // declare an array consisting of 81 one-byte signed integers
 printf("Type the first integer: "); // call function printf and tell what to print
 gets_s(line);
        // gets_s reads the text you have typed and converts it to C-string, i.e. puts
        // the integers corresponding to ASCII table into array line. If you, for example
        // type 12, then line[0] will get value 49, line[1] will get value 50 and line[2]
        // is used for the terminating zero. Dimension is 81 because the command
        // prompt window width is 80 positions
 x = atoi(line);
        // atoi converts the string into four-byte integer. The result is assigned to
        // variable x which becomes now initialized. Below we may use x for
        // arithmetic operations
 printf("Type the second integer: ");
 gets_s(line); // use the same array line
 y = atoi(line);
```

My second program (3)

```
  z = x + y; // calculates the sum of two variables and assigns it to variable z
  printf("The sum is %d", z);
            // prints the result. Here %d means that argument z is an four-byte signed
            // integer which printf has to convert into string
  return 0;
}
```

Some conclusions:

1. Any value read from keyboard is stored as a string – an array of integers of type *char*. The last member of string array is always the terminating zero (byte in which all the bits are 0). It is added always automatically, the user simply needs to press *ENTER*.
2. Arithmetical operations are possible only with values stored as integers (*signed* or *unsigned*, 1, 2 4 or 8 bytes). Operations with strings are not supported.
3. Output into command prompt window is also a string.
4. Converting from strings into integers and vice versa is an inevitable part of C program.

My third program(1)

```
#include "stdio.h"
#include "stdlib.h"
int main()
{
  int x, y;
  char line[81];
  printf("Type the first integer: ");
  gets_s(line);
  x = atoi(line);
  printf("Type the second integer: ");
  gets_s(line);
  y = atoi(line);
  if (x > y)
            printf("%d is greater than %d", x, y);
  else if (x < y)
            printf("%d is less than %d", x, y);
  else
            printf("%d is equal with %d", x, y);
  return 0;
}
```

Branching (1)

The *if* statement is called a branching statement because it provides a junction where the program has to select which of the two paths to follow:

```
if (<expression>)
        <statement_1>;
<statement_2>;
```

If the expression is true, *statement_1* and after that *statement_2* is performed. If the expression is false, *statement_1* is skipped.

In most cases the expression here is a relational expression like $x > y$, $x < y$, $x == y$ (true if x and y are equal, do not forget that $x = y$ means that value of y is assigned to x), $x >= y$ (equal or greater), $x <= y$ (equal or less), $x != y$ (true when x and y are not equal).

But generally, the expression may be any expression. In that case the result of this expression is considered to be true if it is a none-zero value. The zero value is considered as false. For example:

```
int i = 3, j = 3, k;
if (k = i – j) // probably, the programmer wanted to write k == ( i – j)
        statement_1; // here always skipped
if (k = i + j) // probably, the programmer wanted to write k == ( i + j)
        statement_2; // here always executed
```

Branching (2)

The statement to be skipped or executed may be simple (just one expression) or compound (several expressions). In the last case we have to use braces:

```
if (<expression>)
{
        <expression_1>;
        <expression_2>;
        ……………….
}
```

For better readability it is recommended to use the braces in all cases.

The *if else* statement enables to choose between two statements:

```
if (<expression>)
        <statement_1>; // executed if the expression is true
else
        <statement_2>; // executed if the expression is false
<statement_3>; // executed after statement_1 or after statement_2
```

else if simply means that *statement_2* itself is also an *if* statement:

My third program (2)

```
if (x>y)
{
        printf("%d is greater than %d", x, y);
}
else
{
        if (x < y)
        {
                printf("%d is less than %d", x, y);
        }
        else
        {
                printf("%d is equal with %d", x, y);
        }
}
```

Floating point variables (1)

A floating point number has sign (+ or -), integer part, fractional part and decimal separator (point or comma). To store it in computer memory, we have to find a way how to handle the separator (remember that we have nothing but bits in state 0 or 1). The solution is specified by IEEE (Institute of Electrical and Electronics Engineers) standard 754 and its main idea may be explained with the following example:

$+1.234 = +0.1234 * 10^{+1}$
$-12.34 = -0.1234 * 10^{+2}$
$+123.4 = +0.1234 * 10^{+3}$
$-0.1234 = -0.1234 * 10^{0}$
$+0.012.34 = +0.1234 * 10^{-1}$
$-0.0023.4 = -0.1234 * 10^{-2}$

So, we may store a floating point number as a set of four components:

- sign of the original value
- value of the mantissa
- sign of the exponent
- value of the exponent

C has 3 floating point types: float, double and long double. In Visual Studio, however, there is no difference between *double* and *long double*.

Floating point variables (2)

float f = 5.6; // single precision variable

declares 4-byte variable *f* and initializes it to 5.6. On this memory field:
sign of the value – 1 bit
mantissa – 23 bits
exponent with sign - 8 bits

The maximum value of numbers of type *float* is approximately $3.4 * 10^{38}$.
double d = 5.6; // double precision variable

declares 8-byte variable *d* and initializes it to 5.6. On this memory field:
sign of the value – 1 bit
mantissa – 52 bits
exponent with sign - 11 bits

The maximum value of numbers of type *double* is approximately $1.7 * 10^{308}$.

Type *float* is seldom used because the range and precision of *double* variables is much better. Remark that for example (https://www.rapidtables.com/convert/number/decimal-to-binary.html)

$0.4_{10} = 0.6666666666666666666\ldots_{16} = 0.01100110011001100 11\ldots_2$

a "nice" decimal number has endless fractional part in hexadecimal and binary systems. Therefore the number of bits used for storing mantissa affects the accuracy of calculations very significally.

My first unaided written program (1)

Task: write a program that calculates and prints the roots of quadratic equation
$ax^2 + bx + c = 0$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Coefficients *a*, *b* and *c* must be variables of type *double*. Read them from the keyboard as we did it in the previous examples. Instead of *atoi* use standard function *atof*.

Examples of arithmetic operations in C:

Addition `x = y + z;`

Subtraction `x = y – z;`

Multiplication `x = y * z;`

Division `x = y / z;`

Changing the algebraic sign `x = -y;`

Order of the arithmetical operations and the usage of parentheses to change it are as in everyday algebra.

In *printf* to print *double* values instead of *%d* use *%lg*.

My first unaided written program (2)

To compute the square root use standard function sqrt:

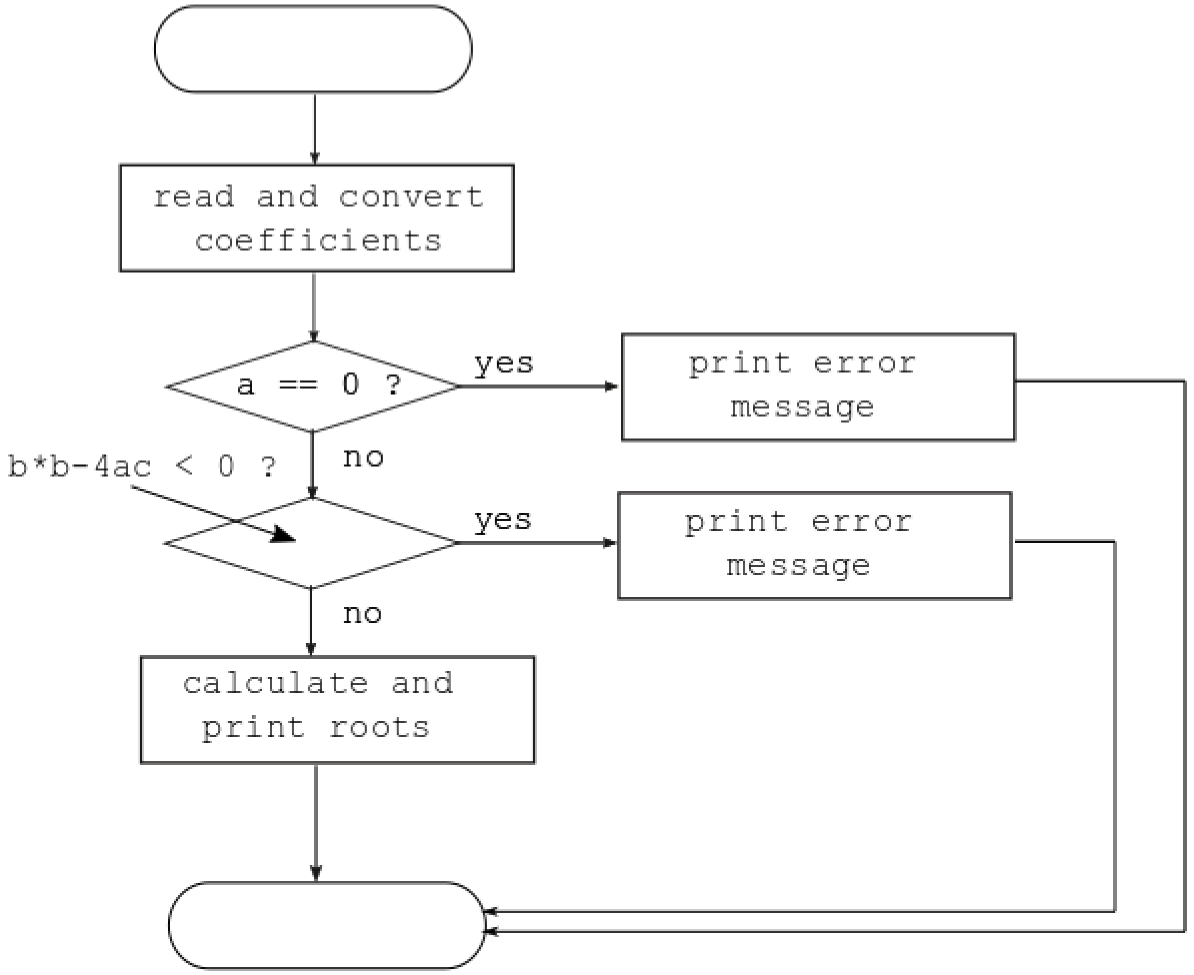
```
#include "math.h" // description of functions like sqrt, exp, sin, cos, etc.
double x, y;
x = sqrt(y);
```

However, before calling *sqrt* your program has to check is the argument positive or 0. If you try to call *sqrt* with negative argument, your program will crash, but this is unacceptable.

So, if $b^2 - 4ac$ is negative, your program has to print a message like "the equation has complex roots" and skip the calculation of x_1 and x_2.

Division `x = y / z;` crashes when *z* is zero. Consequently, if the program prints *"Type value of a"* and the user types zero, the equation has no solution. In that case your program has to print a message like "error in input data" and skip the following operations.

My first unaided written program (3)


read and convert coefficients
a == 0 ?
yes
print error message
no
b*b-4ac < 0 ?
yes
print error message
no
calculate and print roots