Fundamentals of Programming 1

Basics of the C Language

Arkadiusz Chrobot

Department of Information Systems

October 25, 2024

Outline

Initialization of a Variable

Before a variable can be used (more precisely, it can be read) it has to be initialized. That means it must be assigned an initial value. Many program errors have their roots in lack of the initialization. Fortunately variables of the global scope have a default value of zero. However, such an initial value is not always suitable for the program.
In this lecture most of the names of variables will be a single lowercase letter.

Assignment Instruction

Using an assignment instruction is probably the easiest way of initializing a variable. In the C language this instruction has a = symbol. Generally, the assignment instruction is used for changing a value of the variable, not only initializing it. Formally, such an instruction evaluates an expression standing on its right side and converts the type of the result to the type of the variable placed on its left side. As a side-effect of such a conversion the variable gets the value of the expression. The programmers are more interested in the side-effect than in the conversion. In the C language the assignment instruction is also an operator, which means it returns a value.

Initialization Methods

Initialization of Integer Variables

A variable can be assigned an initial value in the place where it is declared. The listing below shows how it can be done for integer variables.

```
int a = 3, b = 075, c = 0xab, d = 1u;

int main(void)
{
    return 0;
}
```

If the number is prefixed by zero, then it is an octal number. If the prefix is `0x`, then it is a hexadecimal number. The integer numbers can also have a suffix, like `u` or `U`, which means the numbers are unsigned or `l` or `L`, which means they are of the `long` type. Both of the suffixes can be combined into `lu` or `LU`.

Initialization Methods

Initialization of Integer Variables

Because in the C language the assignment instruction is also an operator, it is possible to assign a single value to many variables, just like that:

```
int a, b, c;

int main(void)
{
    a=b=c=3;
    return 0;
}
```

Initialization Methods

Character Variables Initialization

A character variable (of the `char` type) may be assigned a number that can mean an ASCII code or be assigned a character in apostrophes.

```
char a = 65, b = 'a';

int main(void)
{
    return 0;
}
```

Please remember that the variables of such a type may store both characters and numbers.

Initialization Methods

Floating-Point Variables Initialization

The floating-point variables (of the `float`, `double` and `long double` types) can be initialized with the use of a regular decimal number with fraction part or by a number expressed in the scientific notation. For example the `0.01` number may be expressed as `1e-2` which means $1 \cdot 10^{-2}$. If the decimal number should be of the `float` type, then it ought to be ended with an `f` suffix.

```
double a = 0.001f, b = 0.02, c = 1e-2;

int main(void)
{
    return 0;
}
```

Constants and the const Keyword

The const keyword can be used, instead of the preprocessor #define directive, to define a constant, like this:

```
const int SEVEN = 7;

int main(void)
{
    return 0;
}
```

The const keyword means that the value of SEVEN will not change during the program run[1]. Please note, that the identifier of the constant is, according to the convention, written in uppercases.

[1] There are ways to change the value of such an constant, but they won't be discussed here.

Operators

Operators are for building expressions. The value of an expression may be assigned to a variable with the help on the assignment instruction which is also an operator. There are several categories of operators in the C language, but only a few of them will be discussed during this lecture. Similarly as in mathematics operators are evaluated in the expressions in a specific order. In other words there is defined a precedence order for the operators. Other attribute of the operators is associativity, which can be right-to-left or left-to-right.

Relational Operators

Relational operators are binary operators. In this context it means that they require two arguments. The operators test the relation between values of those arguments and return 1 if it is true or 0 otherwise.

Operators	**Description**
==, !=	Equality and inequality operators. The first one returns one if its arguments are equal, the second one does the same if they are not equal.
<, >, <=, >=	“less than”, “greater than”, “less than or equal“, ”greater than or equal“

Arithmetic Operators

Operators	Description
++, --	Unary increment and decrement operators. They may be left-to-right associative or right-to-left associative, for example `++a;` (pre-increment), `a++;` (post-increment), `--a;` (pre-decrement), `a--;` (post-decrement). They increment or decrement the value of variable by one and return the result of this operation.
+, -	Those operators may be binary or unary. In the former case they simply mean addition and subtraction. In the later case the - operator changes the sign of a number and the + does nothing.
*, /	Multiplication and division operators. **Warning**: If the arguments of the division operator are integers, then the result is also an integer, irrespectively of the type of the variable in which it is stored.
%	The modulus operator returns the remainder after division of two integer numbers.

Arithmetic Operators

Integer Overflow

The integer overflow happens when the result of an arithmetic operation (i.e. an operation that involves arithmetic operators) is outside of the range of the variable where it is stored. The final value of the result is thus incorrect.

```
unsigned char a = 255;
char b = -128;

int main(void)
{
    a = a + 1; //The value of "a" is 0;
    b = b - 1; //The value of "b" is 127.

    return 0;
}
```

Arithmetic Operators

Integer Overflow — Explanation

$$\begin{array}{rr} & 1\,1\,1\,1\,1\,1\,1\,1 \\ + & 0\,0\,0\,0\,0\,0\,0\,1 \\ \hline & 1\,0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

$$\overset{+1}{\leftarrow}\overset{+1}{\leftarrow}\overset{+1}{\leftarrow}\overset{+1}{\leftarrow}\overset{+1}{\leftarrow}\overset{+1}{\leftarrow}\overset{+1}{\leftarrow}\overset{+1}{\leftarrow}$$

The integer overflow in a variable of the `unsigned char` type.

Arithmetic Operators

Integer Overflow — Explanation

$$
\begin{array}{rr}
 & 1\,0\,0\,0\,0\,0\,0\,0\,0 \\
+ & 1\,1\,1\,1\,1\,1\,1\,1 \\
\hline
 & 1\,0\,1\,1\,1\,1\,1\,1\,1
\end{array}
$$

The integer overflow in a variable of the `char` type.

Arithmetic Operators

Modulus Operator – Description

The modulus operator in the C language is applied only to integers. Its result always fulfills the following equation: `(x/y)*y+(x%y)==x`. The examples show the result of the modulus operation when at least one of its arguments is negative.

```
5%-2  // The value of the expression is 1.
-5%2  // The value of the expression is -1.
-5%-2 // The value of the expression is -1.
```

Like in the case of a regular division operation, the divisor must not be zero.

Arithmetic Operators

Division Operator – Description

The expressions in the example yield a different results, although they look very similar. In the first expression both arguments are integers. In the second one, the first argument is a floating-point number. The C language compilers recognize all numbers without the decimal point as an integer number.

```
double a;

int main(void)
{
    a = 4/5;   // The result is 0.
    a = 4.0/5; // The result is 0.8.

    return 0;
}
```

Arithmetic Operators

Shorthands

In some cases, when the value of an expression is assigned to a variable which itself is used in the expression it is possible to use a shorthand form of the assignment.

```
int a=2, b=2;
int main(void)
{
    b+=a; // Instead of b=b+a;
    b-=a; // Instead of b=b-a;
    b*=a; // Instead of b=b*a;
    b/=a; // Instead of b=b/a;
    b%=a; // Instead of b=b%a;

    return 0;
}
```

Arithmetic Operators

Description of the Increment and Decrement Operators

```
int a = 4, b;
int main(void)
{
    b=++a; //After: "b" is 5, "a" is 5.
    a=4;
    b=a++; //After: "b" is 4, "a" is 5.
    a=4;
    b=--a; //After: "b" is 3, "a" is 3.
    a=4;
    b=a--; //After: "b" is 4, "a" is 3.
    a=4;
    a++; //After: "a" is 5.
    a=4;
    ++a; //After: "a" is 5.
    a=4;
    a--; //After: "a" is 3.
    a=4;
    --a; //After: "a" is 3.

    return 0;
}
```

Boolean Operators

Boolean (logical) operators are used in expressions which evaluate to true or false. In the C language the following boolean operators are available:

Operator	Description
`||`, `&&`	Binary operators of logical sum (or) and logical product (and).
`!`	The unary logical negation operator.

Although the operators yield `1` (true) and `0` (false), it is worth to remember, that in the C language **every expression that evaluates to value different than zero is simultaneously true and every expression that evaluates to zero is simultaneously false**. The `&&` operator returns true (`1`) if both of its argument are true and false (`0`) otherwise. The `||` operator yields false (`0`) if both its arguments are false and true otherwise.

Boolean Operators

Short-Circuit Evaluation

If the first argument of the && operator is false than the second one is not evaluated – there is no need to do so, the whole expression is false. Similarly, if the first argument of the || operator is true, then the second one is ignored – the whole expression is true. This is called a short-circuit evaluation and in some cases can have a side-effect.

```
int a,b;
int main(void)
{
    (a=0)&&(b=4); //Both variables will be zero.
    (a=4)&&(b=3); //The "a" variable will be 4 and "b" will be 3.
    (a=0)||(b=0); //Both variables will be zero.
    (a=3)||(b=4); //The "a" variable will be 3 and "b" will be 0.

    return 0;
}
```

Bitwise Operators

The bitwise operators are similar to boolean operators but operate on pairs of corresponding bits of their arguments instead of the whole values. The arguments have to be integers. The bitwise operators may be used in shorthand assignments like the arithmetic operators.

Operator	Description
`|`, `&`, `^`	The bitwise or, bitwise and, and bitwise exclusive or (xor) operators.
`~`	Unary bitwise complement operator.
`>>`, `<<`	The bitwise left and right shift operators. **Warning**: In C language both of those operators may be applied to negative numbers. Particularly, shifting right a negative number results in negative number – the sign bit (Most Significant Bit in two's compliment) is copied on the left.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 =$

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 0$

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 00$

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 000$

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 0000$

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 00000$

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 000000$**0**

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 0000000$

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 00000001$

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The & Operator

Animation

The Bitwise & Operator – an Example

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 00000001 = 1$

The bitwise & operator evaluates pairs of bits in its arguments. If both bits in a specific pair are set (equal one), then a bit on the same position in the result will be also set. Otherwise the bit will be cleared (its value will be zero). This operator is often used to test the value of a specific bit or group of bits in the first argument. The second argument in that case is of a known value and is called a *mask*. The whole operation is called *masking*.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 =$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 = 0$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 = 00$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 = 000$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 = 0000$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 = 00000$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 = 000001$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 = 0000011$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 = 00000111$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The | Operator

Animation

The Bitwise | Operator – an Example

$5 \mid 3 = 00000101 \mid 00000011 = 00000111 = 7$

Similarly to & operator the | operator evaluates every pair of bits in both of its arguments. However if both bits in a specific pair are cleared the bit on the same position in the result is also cleared. Otherwise it is set.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 =$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 0$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 00$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 000$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 0000$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 00000$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 000001$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 0000011$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 00000110$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ^ Operator

Animation

The Bitwise ^ Operator – an Example

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 00000110 = 6$

The bitwise exclusive or (^) operator is similar to the previously described bitwise operators, but in its case, if both bits in a specific pair have different value the resulting bit is set, otherwise it is cleared.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 =

This bitwise operator is unary. It flips all bits in its argument.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 = 1

This bitwise operator is unary. It flips all bits in its argument.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 = 11

This bitwise operator is unary. It flips all bits in its argument.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 = 111

This bitwise operator is unary. It flips all bits in its argument.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 = 1111

This bitwise operator is unary. It flips all bits in its argument.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 = 11111

This bitwise operator is unary. It flips all bits in its argument.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 = 111110

This bitwise operator is unary. It flips all bits in its argument.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 = 11111101

This bitwise operator is unary. It flips all bits in its argument.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 = 11111010

This bitwise operator is unary. It flips all bits in its argument.

The ~ Operator

Animation

The Bitwise ~ Operator – an Example

~ 5 = ~ 00000101 = 11111010 = 250

This bitwise operator is unary. It flips all bits in its argument.

The << Operator

Illustration

The Bitwise « Operator – an Example

$5 << 2 = 00000101 << 2 = 00010100 = 20$

The << operator shifts values of all bits of its first argument to the left. The values are shifted as many positions as it is indicated by the second argument. In the result the values of the first argument's most significant bits are dropped and the least significant are padded with zeros. The shifting to the left is equivalent to the multiplication by a power of two. In the example it is the 2^2, so the 5 is multiplied by 4. The operator can only be applied to integers.

The >> Operator

Illustration

The » Operator – an Example

$5 >> 2 = 00000101 >> 2 = 00000001 = 1$

The >> operator shifts values of all bits in its first argument to the right. The values are shifted as many positions as it is indicated by the second argument. The least significant bits of the first argument are dropped. The most significant bits are padded according to the sign of the argument. If it is negative number, than the bits are padded with ones, otherwise with zeros. Such an operation is equivalent to the integer division by a power of two, with the exception of negative numbers. For example, the result of shifting -1 will always be -1, no matter how many positions the number is shifted. In the example the 5 is divided by 4. The fraction part of the result is truncated. The result is always an integer and the operator can only be applied to integers.

Bitwise Operators

Summary

The symbols of bitwise and logical operators look very similar and thus may be confusing. Burce Eckel, the autor of "Thinking in Java" book devised a rule, that can help to apply the right operator: *"Bits are small, so only one character is needed for a bitwise operator."*

The Ternary Operator

The ternary operator (conditional operator) is similar to the conditional statement that will be discussed in the next lecture. Unlike the statement, the operator yields a result. The pattern shows how to apply the operator:

```
variable=condition?first_expression:second_expression;
```

If the condition evaluates to truth, the operator yields the result of the evaluation of the first expression, otherwise it will return the result of the evaluation of the second expression. The result will be stored in the variable. It is possible to skip the assignment, if only the side-effect of evaluation of the second or the first expression is of interest.

```
int a=5, b=3, max;
int main(void)
{
    max=(a>b)?a:b;
    return 0;
}
```

Type Casting

Type casting is used for changing the type of a value. Type conversion may be implicit or explicit. The former one is made by compiler without participation of the programmer. The later is forced by the programmer with the use of the *casting operator*.

```
int a;
int main(void)
{
   a=12.3;
   return 0;
}
```

In the example the original type of the `12.3` number, i.e. `double` is implicitly converted to `int`. It means that the fraction part of the number will be lost. Such a side-effect always happens when a "larger" type is converted to a "smaller" one.

Type Casting

Explicit Conversion

In some cases it is necessary to force the type conversion. It can be accomplished with the use of the cast operator, i.e. by placing the name of the desired type in the parentheses before a variable or an expression.

```
double a;
int x = 4, y = 3;
int main(void)
{
        a=(double)x/y;
        return 0;
}
```

In the example the type of the value of the `x` variable is converted to the `double`. Thanks to that the result of division is a floating-point number, not an integer with truncated fraction part.

The Address Operator

The address operator returns the memory address of the variable that it is applied to. The symbol of an address operator is &, just the same as the bitwise and. The compiler recognizes those operators by the context in which they are used. The address operator is unary and the bitwise and operator is binary. The memory address is needed by the `scanf()` function to store in a variable the data acquired from a keyboard. But this is only one of many applications of the address operator.

The sizeof Operator

The sizeof operator returns the size of variable that it is applied to. The size is in bytes. It is preferred to put the name of variable in parentheses, but it is not necessary. Also instead of variable's name, the identifier of its type may be used.

```
long unsigned int a,b,c;

int main(void)
{
    a=sizeof b;
    a=sizeof(c);
    return 0;
}
```

Operators Precedence

The precedence of operators in the C language is defined by their priorities. However, it may be changed with the use of parentheses. As for the discussed operators the post-increment and the post-decrement operators are of the highest priority. Then, the pre-increment and pre-decrement operators. Next, all the unary operators (including the `sizeof` and cast operators). Later, all the arithmetic operators, just like in the mathematics. Subsequently, the bitwise shift operators and next the relational operators (the equality and inequality operators have lower priority than the rest of them). Later the bitwise operators (in order: the and, the exclusive or and the or) and after them the boolean operators, in the same order, except for exclusive or, which is nonexistent. Next goes the ternary operator. The lowest priority have the assignment operator and assignment shorthands.

Basic Input/Output Operations

In this part of the lecture the `scanf()` and `printf()` functions will be discussed. Both of them are defined in the `stdio.h` header file. The `scanf()` function allows the program to store in a variable a value typed by the user on the keyboard. The `printf()` function displays values on a computer screen. Other input/output functions will be discussed in later lectures.

The scanf() Function

The scanf() function allows the user to enter a value to a variable with the use of a keyboard.

```
#include<stdio.h>
int a;
int main(void)
{
    scanf("%d",&a);
    return 0;
}
```

In the simplest case the function takes two arguments that are separated by a comma. The first one is a *format string* which defines the type of the entered value. It consists of a *conversion specifier* placed in quotation marks. The conversion specifier is a single character or sequence of characters prefixed by a percent sign (%). The second argument is the address of the variable, where the value should be stored.

The scanf() Function

The Most Important Conversion Specifiers

The table contains examples of conversion specifiers that will be used during the lectures.

Conversion Specifiers	Description
%d	A decimal integer of the int type.
%ld	A decimal integer of the long int type.
%hd	A decimal integer of the short int type.
%hhd	A decimal integer of the char type.
%u	A decimal natural number of the unsigned int type.
%lu	A decimal natural number of the unsigned long int type.
%hu	A decimal natural number of the unsigned short int type.
%hhu	The decimal natural number of the unsigned char type.
%c	A character.
%f	A floating-point number of the float type.
%lf	A floating-point number of the double type.

The scanf() Function

Entering Single Characters

When applying a scanf() function for entering several single characters the programmer has to remember that each of them is confirmed by the Enter key, which also leaves a single character that can be read by subsequent calls of the scanf() function. To avoid this unwanted effect a solution presented in the listing may be applied.

```
#include<stdio.h>
char a,b;
int main(void)
{
        scanf("%c",&a);
        scanf(" %c",&b);
        return 0;
}
```

The solution consist in adding a space between the first quotation mark (") and the percent sign (%) in the second conversion specifier.

The `printf()` Function

The `printf()` function is similar to the `puts()` function. Both of them display a string (a series of characters in quotation marks) on screen. However, the `printf()` function does not move the cursor to the beginning of the next line. Moreover, it can display values of variables and expressions. To this end the programmer has to place in the format string as many conversion specifiers as values she or he wants to display. Then, after the string she or he has to place all the values, variables and expressions and separate them by commas.

The printf() Function

An Example

```c
#include <stdio.h>

int a,b;

int main(void)
{
    puts("Please, enter an integer:");
    scanf("%d",&a);
    puts("Please, enter a second integer");
    scanf("%d",&b);
    printf("%d & %d = %d\n",a,b,a&b);
    return 0;
}
```

The `printf()` Function

Formatting Strings

Formatting String	Description
`%d`	A decimal integer of the `int` type.
`%ld`	A decimal integer of the `long int` type.
`%u`	A natural number of the `unsigned int` type.
`%c`	A character.
`%f`	A floating-point number of the `double` type. The string may contain additional information about formatting, for example `%.3f` means that only the first three digits of the number's fraction part will be displayed.
`%lf`	A floating-point number of the `long double` type. It also may contain additional information about formatting.
`%e`, `%E`	A floating-point number of the `double` type expressed in the scientific notation, for example `3e-9` (for the `%e` formatting) or `3E-9` (for the `%E` formatting).
`%le`, `%lE`	A floating-point number of the `long double` type expressed in the scientific notation (please, note the examples above).
`%x`, `%X`	A natural number in hexadecimal, for example `a5` (for the `%x` formatting) or `A5` (for the `%X` formatting).
`%o`	A natural number in octal.

The `printf()` Function

Escape Sequences

For moving the cursor to beginning of the next line on the screen or displaying a tabulator, *escape sequences* have to be used. These usually consist of two characters, like a backslash and an another character. The table contains descriptions of some of them.

Escape Sequence	**Description**
`\n`	A newline character.
`\r`	A beginning of the line character.
`\\`	The \ character (A single \ sign will be displayed on the screen.
`\"`	The quotation mark.
`\t`	The tabulator.
`%%`	The percent.

Questions

?

Thanks

Many thanks to Grzegorz Łukawski, PhD and Leszek Ciopiński, PhD for helping me to complete the Polish version of this slides.

The End

Thank You for Your attention!