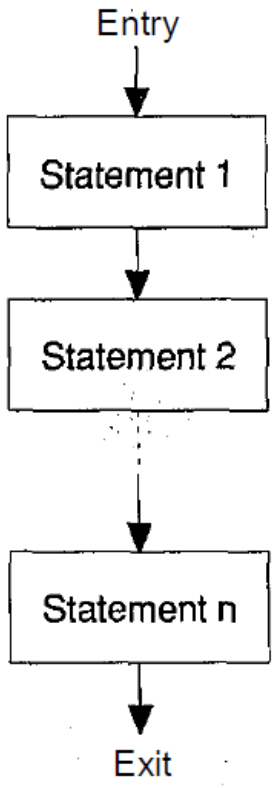
Lecture 1
Sequential Structure

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

The sequential structure



Example 1

printing Victor Hugo's famous quotation:

```
Knowledge is Power.
```

Here is the program:

```
#include <stdio.h>
int main(void)
   {
     printf("Knowledge is Power\n");
     return 0;
   }
```

Example 2

Consider a program that computes the total interest earned on an investment that pays a fixed rate of simple interest when the principal remains invested for a specified number of years. Here is an algorithm for solving this problem:

1. Read the amount of principal, interest rate (as a percentage), and the number of years for which the principal is to be invested.
2. Convert the percentage rate read into a decimal fraction.
3. Compute the total interest earned using the formula

   ```
   interest = principal X rate x years.
   ```

4. Print interest.

```
/*
     * This program computes the total interest accrued on an
 * investment that pays simple interest at a fixed rate when the
     * principal is invested for the specified number of years.
 */

#include <stdio.h>

int main(void)
  {
    /* variable declarations */
    float principal, rate, interest;
    int     years;

        /* prompt the user to provide input values */
    printf("principal, rate, and years of investment? ");

    /* read the input values */
     scanf("%f %f %d", &principal, &rate, &years);

    /* compute interest */
    rate = rate / 100;  /* convert percentage into fraction */
    interest = principal * rate * years;

    /* print interest */
    printf("interest = %f\n", interest);

    /* successful completion */
    return 0;
  }'
```

Character Set

The set of characters that may appear in a legal C program is called the *character set* for C. This set includes some *graphic* as well as *non-graphic* characters.

The graphic characters are those that may be printed.

The non-graphic characters are represented by escape sequences consisting of a backslash \ followed by a letter.

The character set also includes a *null* character, represented as \0.

C character set (graphic characters)

Character	Meaning	Character	Meaning
0,1,. • . ., 9	Decimal digits	:	Colon
A, B, . . . , , z	Uppercase letters	;	Semicolon
a, b, . . ., , z	Lowercase letters	<	Less than
j	Exclamation point	=	Equal to
"	Double quotation mark	>	Greater than
#	Number/pound sign	9	Question mark
$	Dollar sign	0	"At" sign
%	Percent sign	[	Left bracket
&	Ampersand sign	\	Backslash
'	Apostrophe/single quotation mark	]	Right bracket
		A	Caret/circumflex
(	Left parenthesis		Underscore
)	Right parenthesis	T	Accent grave/back quotation mark
*	Asterisk		
+	Plus	{	Left brace
,	Comma	1	Vertical bar
—	Minus/hyphen	)	Right brace
.	Period	~	Tilde '
/	Slash	¥	Blank/space

C character set (non-graphic characters)

Character	Meaning	Character	Meaning
\a	Audible alert (bell)	\b .	Back space
\f	Form feed	\n	New line
\r	Carriage return	\t	Horizontal tab
\v	Vertical tab		

The graphic characters, other than decimal digits, letters, and blank, are referred to as *special* characters.

Blank, horizontal and vertical tabs, newlines, and formfeeds are called *whitespace* characters.

The C compiler groups characters into *tokens*.

A token is a sequence of one or more characters that have a uniform meaning.

Some tokens, such as /, *, -, >, and =, are only one character long; others, such as ->, ==, >=, comments, and variable names, are several characters long.

When collecting characters into tokens, the compiler always forms the longest token possible.

C is a *free-format language,*

tokens can go anywhere on a page

and can be separated by any number of whitespaces.

```
int main(void)
  {
    printf("Knowledge is Power\n");
    return 0;
  }
```

can equivalently be written as

```
int main(void){printf("Knowledge is Power\n");return 0;}
```

Data Types

Data is differentiated into various *types*.

The type of a data element restricts the data element to assume a particular set of values.

`char`	a character in the C character set
`int`	an integer
`float`	a single-precision floating-point number
`double`	a double-precision floating-point number

The qualifiers `short` and `long` may be applied to some of the above data types. Thus we have the following additional data types:

`short int`	also abbreviated as `short`
`long int`	also abbreviated as `l o n g`
`long double`	an extended-precision floating-point number

The qualifiers `signed` and `unsigned` may be applied to

`char`, `short int`, `int`, or `long int`.

`unsigned` variables can only have nonnegative values,

`signed` variables can have both positive and negative values.

In the absence of explicit `unsigned` specification,

`int`, `short int`, and `long int` are considered to be `signed`.

Ones' complement

8 bit ones' complement

Binary value	Ones' complement interpretation	Unsigned interpretation
00000000	+0	0
00000001	1	1
⋮	⋮	⋮
01111101	125	125
01111110	126	126
01111111	127	127
10000000	−127	128
10000001	−126	129
10000010	−125	130
⋮	⋮	⋮
11111101	−2	253
11111110	−1	254
11111111	−0	255

Two's complement

8 bit two's complement

Binary value	Two's complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
⋮	⋮	⋮
01111110	126	126
01111111	127	127
10000000	−128	128
10000001	−127	129
10000010	−126	130
⋮	⋮	⋮
11111110	−2	254
11111111	−1	255

Many implementations represent

a `char` in 8 bits,
a `short` in 16 bits,
an `int` in 16 or 32 bits,
a `long` in 32 bits,

The width for `float` is often 32 bits, and that of `double` 64 bits.

These bit widths have not been specified by the C language,
but have become traditional.

All that C specifies is that

the range of `int` may not be smaller than `short`
the range of `long` may not be smaller than `int`
a `long double` is at least as precise as a `double`
a `double` at least as precise as a `float`

The particular bit widths for an implementation are specified in that implementation's standard header files <limits. h> and <float. h>.

All types of integers and characters are collectively referred to as of *integral* type,

the types `float`, `double`, and `long double` as *of floating-point* type

The term *arithmetic* type is used to refer collectively to the integral and floating-point types.

Classes of Data

A computer program manipulates two kinds of data

variables and constants

Variables

A *variable* is an entity used by a program to store values used in the computation.

Every variable in C has a type and it must be declared before it is used.

Declarations

A *declaration* consists of a type name followed by a list of one or more variables of that type, separated by commas. Thus, the declarations

```
int     mint;
char    cherry;
double  trouble;
float   swim, snorkel;
```

The variables `swim` and `snorkel` could have been equivalently declared in two separate declarations:

```
float swim;          /* time spent swimming in minutes • */
float snorkel;  /* time spent snorkeling in hours */
```

It is possible to assign an initial value to a variable in its declaration by following its name with an equal sign and the value.

```
float start = -1.0, final = 1.0, increment = Q.l;
```

Names

C places some restrictions on what can be a *variable name*. The following are the rules for naming a variable:

1. A variable name must begin with a letter or underscore, and may be made up of any combination of letters, underscores, or the digits 0-9. Whitespace characters are not permitted within a name. Thus,

   `foo` `r2d2` `Agent707` `ANSI_C` `_mvBytes`

   are valid names, whereas the following are not, for the reasons indicated:

   `4 eve r` The first character is not a letter or underscore.
   `x2 . 5` Period (.) is not allowed in a variable name.
   `. ANSI C` Embedded spaces are not permitted.

2. A variable name may use uppercase letters, lowercase letters, or both. Changing the case of even one character makes a different name. For example, the variable names

```
interest        Interest        INTEREST
```

are all recognized to be different. However, it is bad programming style to have distinct names that differ only in the case of their letters.

3. The number of characters that can be used in a variable name is compiler-dependent. The original description of C specified that only the first eight characters of a variable name were significant, and hence names like `average_weight` and `average_width` would be indistinguishable since they are identical up to the first eight characters. ANSI C permits at least 31 significant characters in variable names, and hence accidental name collision rarely becomes a problem.

4. C reserves certain names, called *keywords*, for specific meanings, and they cannot be used as variable names. The 32 standard keywords are:

```
auto      break    case     char     const     continue
default   do       double   else     enum ,    extern
float     for      goto     if       int       long
register  return   short    signed   sizeof    static
struct    switch   typedef  union    unsigned  void
volatile  while
```

Technically, a variable name is an identifier.

Identifiers are names given to various program entities.

Examples of identifiers other than variable names include function names, enumeration constants, symbolic constants, etc.

The rules discussed above for naming variables apply to all identifiers.

Constants

A *constant* is an entity whose value does not change during program execution.

Constants are of five different types:

integer, floating-point, character, string, and enumeration.

Integer Constants

An *integer constant* is a number that has an integer value.

Integer constants may be specified in

decimal, octal, or hexadecimal notation.

A *decimal* integer constant consists of a sequence of one or more decimal digits 0 through 9. The first digit of the sequence cannot be 0, unless the decimal integer constant is 0. Thus,

```
0     255     32767   32768   65535   2147483647
```

are valid decimal integer constants.

An *octal* integer constant consists of the digit `0`, followed by a sequence of one or more octal digits 0 through 7. Thus,

```
012     037     0177     01000     077777     0100000
```

are valid octal integer constants corresponding to the decimal integer constants

```
10      31      127      4096      32767      32768
```

A *hexadecimal* integer constant consists of the digit 0, followed by one of the letters `x` or `X`, followed by a sequence of one or more hexadecimal digits `0` through `9`, or letters `a` through `f`, or letters `A` through `F`. Thus,

```
0x1f    0XIF    0xff    0xABC    0x10000    0x7FFFFFF
```

are valid hexadecimal integer constants corresponding to the decimal integer constants

```
31      31      255     2748     65536      2147483647
```

The type of an integer constant is normally `int`.

However, if the value of a decimal integer constant exceeds the largest positive integer that can be represented in type `int`, its type instead is `long`.

An integer constant followed by the letter l or L,such as 123456789L, is an explicit `long` type.

It is also possible to explicitly express `unsigned` constants. by suffixing the constant with an u or U as in 123U

an `unsigned long` integer constant by suffixing the constant with an ul or UL, as in 12345678 9UL

Floating-Point Constants

A *floating-point* constant is a number that has a real value. The *standard decimal form* of a floating-point constant is a number written with a decimal point. Thus,

```
1.0   1.   .1   0.   .0
```

are valid floating-point constants, whereas the following are not for the reasons indicated:

1 Contains no decimal point.
1,000.0 Contains a comma.
1 000.0 Contains a space.

The *scientific notation* is often used to express numbers that are very small or very large.

Thus, 0.000000011 is written as 1.1×10^{-8}

20000000000 as 2×10^{10}.

C provides an *exponential form* for such numbers

(*coefficient*) e (*integer*)

The part appearing before e is called the *mantissa* the part following e is called the *exponent.*

Thus, 1.1×10^{-8} is written as 1.1e-8 and 2×10^{10} as 2e10.

The uppercase E can also be used instead of the lowercase e.

The type of a floating-point constant is double, unless suffixed.

The suffixes f or F indicate a float constant,
and 1 or L a long double constant.

Thus,

1.23F is a floating-point constant of type float,

1e-10L is a floating-point constant of type long double.

Character Constants

A *character constant* consists of a single character enclosed within apostrophes.
For example,

'O' 'a' 'Z' '?' '%'

Character constants are of type `int`.

The value of a character constant is the numeric value of the character in the machine's character set, and hence

depends on whether the machine uses the ASCII or EBCDIC character set.

Character Constant	ASCII	EBCDIC
'O'	48	240
'a'	97	129
'z'	90	233
'?'	63	111
'%'	37	108

If the apostrophe or the backslash is to be used in a character constant, it must be preceded by a backslash as shown below:

```
'\"'        '\\'
```

Non-graphic characters may also be used to form character constants. Thus, the following are valid character constants:

`'\a'` (Audiblealarm)	`'\b'` (Backspace)	`'\f'` (Formfeed)
`'\n'` (Newline)	`'\r'` (Carriagereturn)	`'\t'` (Horizontaltab)
`'\v'` (Vertical tab)		

In addition, an arbitrary byte-sized bit pattern can be used to specify the value of a desired character constant by writing

'\ooo'

where the escape \ ooo consists of a backslash followed by 1,2, or 3 octal digits. The bit pattern may also be expressed in hexadecimal by writing

'*xhh*'

where the escape *xhh* consists of a backslash, followed by an `x` and `1` or `2` hexadecimal digits. Here are some examples:

Character Constant		Decimal	ASCII	EBCDIC
Octal	Hexadecimal	Value	Character	Character
`'\100'`	`'\x40'`	64	`'0'`	`- -`
`'\135'`	`'\x5d'`	93	`']'`	`')'`
`'\176'`	`'\x7e'`	126	`/ ^ /`	`' ='`

A special case of the above construction is \0, which represents the character with the value zero, that is, the null character.

String Constants

A string constant, or simply a *string,* consists of zero or more characters enclosed within double quotation marks. Non-graphic characters may also be used as part of a character string. Here are some examples:

```
"Madam, I'm Adam"
"Programming in C is fun\n"
"\n\n\t\t***** inventory report *****\n\n"
```

The double quotation marks are not part of the string;

they only serve to delimit it.

```
I' m a "and I'm a \
```

is written as

```
"I'm a \" and I'm a \\"
```

The length of a string is the number of characters that form the string.

There is no limit on the length of a string.

A string can be continued by putting a backslash (\) at the end of the line as in:

```
"Great things are not done by impulse,\
 but by a series of small things brought together"
```

Adjacent string constants are concatenated at compile time. Thus, the last string can equivalently be written as

```
"Great things are not done by impulse,"
" but by a series of small things brought together"
```

or

```
"Great things are not done by impulse, "
" but by a series of small things"
" brought together"
```

The compiler automatically places the null character \0 at the end of each string so that a program scanning a string can find its end.

The physical storage required is thus one byte more than the number of characters enclosed within double quotation marks.

When the compiler concatenates two string constants,
it puts the null character at the end of the resultant concatenated string

Note that a character constant, say 'Z', is not the same as
the string that contains the single character "Z".

The former is a single character and
the latter a character string consisting of characters Z and \0.

ARITHMETIC OPERATORS

An *operator* is a symbol that causes specific mathematical or logical manipulations to be performed.

arithmetic operators

addition (+), *subtraction* (-), *multiplication* (*), *division* (/),
remainder (%), *unary plus* (+), *unary minus* (-)
increment (++) and *decrement* (—)

All these operators, except remainder (%), operate on operands of any arithmetic type. The remainder operator (%) takes only integral operands.

The first five arithmetic operators are *binary* operators require two operands.

The result of addition (+), subtraction (-), multiplication (*), and division (/) is the same as in mathematics,

except that when division is applied to integer data, the result is truncated to the integer value.

The operator % obtains the remainder when one integer is divided by another.

```
12 + 9 = 21        12. + 9. = 21.
12 - 9 = 3         12. - 9. = 3.
12 * 9 = 108       12. * 9. = 108.
12 / 9 = 1         12. / 9. = 1.33
12 % 9 = 3
```

The unary minus is a *unary* operator and requires only one operand.

The result of the unary minus applied to an operand is the negated value of the operand.

The unary plus is also a *unary* operator.

The result of the unary plus applied to an operand is the value of the operand.