

# **Programming Language 'C'**

**EUGENE AGEENKO**

**2000**

## **TABLE OF CONTENTS**

<b>INTRODUCTION.....</b>	<b>1</b>
<b>EXPRESSIONS.....</b>	<b>4</b>
<b>STATEMENTS.....</b>	<b>8</b>
<b>DATA TYPES.....</b>	<b>15</b>
<b>CONSOLE INPUT / OUTPUT.....</b>	<b>20</b>
<b>FORMATTED CONSOLE INPUT / OUTPUT.....</b>	<b>21</b>
<b>ARRAYS AND STRINGS.....</b>	<b>26</b>
<b>POINTERS AND ARRAYS.....</b>	<b>28</b>
<b>DYNAMIC MEMORY ALLOCATION.....</b>	<b>30</b>
<b>FUNCTIONS.....</b>	<b>32</b>
<b>SCOPES AND DECLARATION.....</b>	<b>37</b>
<b>FILE I/O.....</b>	<b>40</b>
<b>PREPROCESSOR.....</b>	<b>44</b>
<b>ADVANCED TOPICS.....</b>	<b>48</b>
<b>POINTERS TO FUNCTIONS.....</b>	<b>49</b>
<b>VARIABLE-LENGTH ARGUMENT LISTS.....</b>	<b>50</b>
<b>ABSTRACT DATA TYPES.....</b>	<b>52</b>
<b>DYNAMIC DATA STRUCTURES.....</b>	<b>56</b>

# INTRODUCTION

## Course goal

- Learn basics of C-language
- Write efficient small C-programs w/o manual
- Ability to understand programming literature about C
- Understand advanced programming concepts, such a dynamic memory allocation, and data structures
- Be ready to participate in larger programming assignment

## Course does not cover

- Basics of programming
- Graphic programming
- OOP and C++

## History of C

- 1970: **B**, Ken Thompson, B-language, first UNIX
- 1978: **K&R C**, first C-language by Brian Kernighan, Dennis Ritchie, Ken Thompson
  - fundamental data types, expressions, statements: assignments, function calls, control flow constructions for structured programming, pointers (machine independent address arithmetic), preprocessing
  - C is a middle-level language, providing for efficient programming (near to low-level control over the code + structured programming and derived data types)
- 1989: **ANSI-C**, ANSI standard, (1983-88)
- 1999: **C99**, latest ANSI standard.

## Why C

- Small language (easy to learn)
- Wide range of functions in **standard library**
- Powerful language: data types and control structures match capability of most computers
- Machine independence
- Retain philosophy: 'know what you are doing'
- Application independent
- A range of compilers
- Proven to be extremely effective and expressive.

## First program

```
#include <stdio.h>
/* This program computes maximal integer */

main()
{
    int value, max;
    int k=0;
    int array[10];

    max = 0;
    scanf("%d",&value);

    while (value != 0)
    {
        array[k] = value;

        if (value > max)
            max = value;

        scanf("%d",&value);
        k++;
    }

    printf ("Maximal value is %d \n",max);
}
```

## Program structure

```
Include library header files
Global definition and variables
main()
{
    Variable definitions

    Statements
}
```

## Include library header

- #include <stdio.h>

## Comment

- /\* ... \*/
- Nested comments are not allowed
- // single line comment (C++, C99, most GNU C)

## Constants

- Numerical:  
0            0  
-1          -1  
0xFF        hexadecimal 255

- Character:
  - 'a'            symbol 'a'
  - ' '            space
  - '\n'          end-of-line symbol

## Variables:

- type variable;
- int value;
- declaration with initialization  
int k = 0;
- array declaration  
int array[10];

## Simple types:

- char            single byte holding one character
- int            an integer (of natural size for machine)
- float          single-precision floating point number
- double        double-precision floating point number

## Statement

- statement;
  - max = 0;
  - scanf("%d",&value);  
  &value
  - max = 0;
  - k++;
  - while (value != 0) {...}  
  (value != 0)
  - {  
  statement;  
  statement;  
  /\* ... \*/  
  statement;  
  }
  - if (value > max)  
  max = value;
- ';' – terminator  
assignment  
function call  
– address of value  
assignment  
increment  
while loop statement  
condition: value ≠ 0  
compound  
statement  
(*block*)  
works as single  
statement  
conditional  
statement

## Formatted input / output

- scanf("%d",&value);  
  "%d"  
  &value
  - printf("... %d \n",max);  
  "... %d \n"  
  \n
- input an integer  
format specification  
address of variable  
output an integer  
line-break after  
end-of-line symbol

## Difference from Pascal

Pascal	C
:=	=

=	==
<>	!=
a^	*a
'	"
(* *)	/* */
a: ineger	int a
THEN	
DO	
OR	
AND	&&
NOT	!
NIL	NULL
TRUE	!=0
FALSE	0
A[x,y]	A[x][y]
A[0..9]	A[10]
<u>condition</u>	(condition)
BEGIN	{
END	}
PROGRAM	main
; is separator	; is terminator

## EXPRESSIONS

### Expression

- Variable
- Constant
- Function call
- Operators (assignment, arithmetic, logic, relational)
- And their combination
- Evaluates to a value

### Identifier (variable / function) name

- Letters, digits (not first), '\_', small/capital different
- | <u>Correct</u> | <u>Incorrect</u> |
|----------------|------------------|
| count          | 1count           |
| tmp1           | hi!there         |
| min_value      | min.value        |
- Identifier length is 31 (6 for external) symbols

### Variable

- Named location of memory holding a value

### Pointer

- Address of the variable

## Variable definition

- `type variable = constant;` for a variable
- `type *variable = constant;` for a pointer

## Simple types

- `char` single byte holding one character
- `int` an integer (of natural size for machine)
- `float` single-precision floating point number
- `double` double-precision floating point number

## Variable type qualifiers

- `const int a=10;`  
variable may not be changed by your program
- `volatile char *port;`  
do not optimize, variable may be changed outside of a program (i.e. port)

## Numeric constants

<u>data type</u>	<u>example</u>
<code>int</code>	123 -123
<code>float</code>	12.34F 1.234e-2f
<code>double</code>	1.0 12.34 1.234e-2

## Character constants

- `'A'` character A

## String constants

- `"Hello world!"` - a string
- `""` - an empty string
- `const char s[] = "Hello world!";`  
string constant

## Special characters

- defined using ESCape sequences:

<code>\0</code>	null character (terminates a string)
<code>\xhh</code>	symbol with hexadecimal code <i>hh</i>
<code>\b</code>	backspace (remove last symbol)
<code>\n</code>	new-line symbol
<code>\f</code>	form-feed symbol
<code>\t</code>	horizontal tab
<code>\"</code>	double quote
<code>\'</code>	single quote
<code>\\</code>	Backslash
<code>\?</code>	question mark
<code>\a</code>	alert (beep)

## Operators (grouped by precedence)

++	increment
--	decrement
-	unary minus
*	multiply
/	divide
%	modulus (remainder of division)
+	add s
-	subtract

- integer division truncates any remainder part:
- 5/2 is 2, when 5.0 / 2.0 is 2.5

## Assignment operator (=)

- rvalue = lvalue;
- rvalue - expression referring to an object
- lvalue - the value of the expression
- a = 2 + 5;
- type conversions in assignment:  

```
float f;
int a=5, b;
f = a;           // a converted to float type
f = a/2;        // result of operation is still 2
f = a/2.0;      // result of operation is 2.5
```

## Increment / decrement

- x = x+1;                      x++;                      x--;
- x++;                      increment x after obtaining value of x;
- ++x;                      increment x before obtaining value of x;
- Example:  

```
x = 10;           // sets x to 10
y = x++;         // sets y to 10, and x to 11
```

## Multiple assignments

- x = y = z = 0;
- x = (y = (z = 0));

## Compound assignments

- exp1 = exp1 operator exp2;                      x = x+5;
- exp1 operator= exp2;                              x += 5;

## Parentheses

- [ ]                      index of an array:                      a[5]
- ( )                      increase precedence:                      a\*(b+c)
- { }                      compound statement:                      {a=2; b=3;}

## Logical expressions

- Consists of logical and relational operators
- Returns 1 if true, or 0 if false.
- Any non-0 expression can be evaluated as true.

## Relational operators

>	Greater than
<	Less than
>=	Greater or equal than
<=	Less or equal than
==	Equal
!=	Not equal

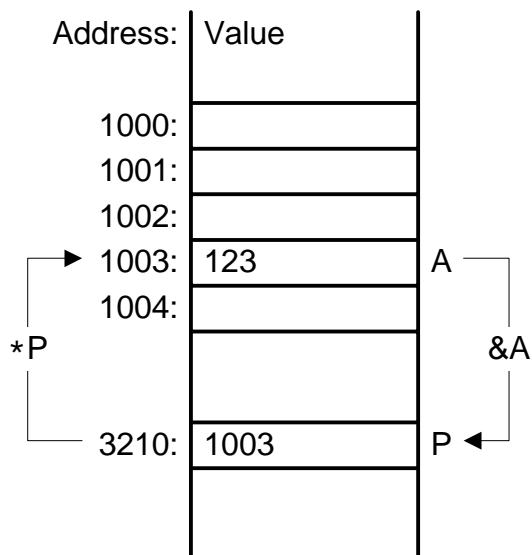
## Logical operators

&&	AND
	OR
!	NOT

## Pointer operators

&	Memory address of operand
*	The value at given address

## Memory



```
int A;
int *P;

P = &A;
// P is an address of A

*P = 100;
A = 100;
// *P is the value by the address P
// therefore *P equals to A
```

## Operator sizeof

- `sizeof(variable)`
- returns the length in bytes of a given variable
- compile-time operator

- used for generation of portable code depending upon the size of the built-in data types

## Precedence (a full list)

- `()` *(brackets)*
- `(arg.list) [] -> . ++ --` *(postfix expressions)*
- `! ~ ++ -- - * & sizeof` *(unary operators)*
- `(type)` *(type cast)*
- `* / %` *(multiplicative operators)*
- `+ -` *(additive operators)*
- `<< >>` *(bit-shifts operators)*
- `< <= > >=` *(relational operators)*
- `== !=` *(equality operators)*
- `&` *(bitwise AND operator)*
- `^` *(bitwise XOR operator)*
- `|` *(bitwise OR operator)*
- `&&` *(logical AND operator)*
- `||` *(logical OR operator)*
- `?:` *(conditional operator)*
- `= += -= *= %= /= <<= >>= &= ^= |=` *(assignment)*
- `,` *(comma operator)*

## Order of evaluation

- Order of evaluation is not specified in standard
- `printf("%d %d", ++n, n+2) /* WRONG */`
- `a[i] = i++; /* ALSO WRONG */`

## Statements

### Statement

- Expression;
- Braces `{ }` group statements into one
- `;` – an empty statement

### Decision statement

- `if (expression)`  
`statement1`  
`else`  
`statement2`
- the **else** clause is optional.
- executes statement1 if expression has non-0 value, otherwise executes statement2 if present.
- `if (value)` is similar to `if (value != 0)`

## Examples (nested if):

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

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

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

## If-else-if ladder (staircase):

```
if (x < 10)
    n = 1;
else if (n < 100)
    n = 2;
else if (n < 1000)
    n = 3;
else
    n = 0;
```

## Multi-way decision statement (switch)

- `switch (expression) {`
  - `case const1: statements1 break;`
  - `case const2: statements2 break;`
  - `default statements3 break;``}`
- the expression must evaluate to an integer type
- the execution is continued from the statements that are associated with the constant corresponding to the value of expression
- the execution is continued until the **break** statement or the end of the **switch** statement is reached
- when no match is found, the **default** case is executed (optional)
- no two **case** constants can have identical value
- 257 **case** statements (1023 in C99)
- last **break**; is not necessary at all but advised

## Example:

```
int c;
int ndigit[10]={0,0,0,0,0,0,0,0,0,0};
int nblank = 0;
int nother = 0;

while ((c = getchar()) != EOF)
{
    switch (c) {
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
        case '0': ndigit[c-'0']++; break;
        case ' ':
        case '\t':
        case '\n': nblank++; break;
        default: nother++; break;
    }
}
```

## while loop

- while (expression)  
    statement

## Example (Copying from input to output)

```
#include <stdio.h>
main()
{
    int c;

    c = getchar(); // read first
    while (c!=EOF)
    {
        putchar(c);
        c = getchar(); // read next
    }
}
```

```
#include <stdio.h>
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

## Skipping symbols with `while`

- This loop will skip all the space symbols from the input:

```
while (c==' ')
    c = getchar();
```

## Using variable (flag) to control execution

- The loop will execute until **found** flag is raised:

```
/* look for a digit symbol */
int c;
int found = 0;

c = getchar();
while ((!found) && (c!=EOF))
{
    if ((c >= '0') && (c <= '9'))
        found = c;
    c = getchar();
}
```

## do-while loop

- do  
    statement  
while (expression);
- Works similar to while loop, but evaluates the expression after executing statement
- Executes statement at least **once**
- Unlike in Pascal execution continued **while** expression is **true** (until it is **false**)
- Use **do-while** when statement **must be** executed at least once. Otherwise use **while** loop:

## Example

```
/* look for a digit symbol */
int c;
int found = 0;

do
{
    c = getchar();
    if ((c >= '0') && (c <= '9'))
        found = c;
}
while ((!found) && (c!=EOF))
```

## Another example (expecting valid response)

- The good choice for **do-while** loop is for waiting for a valid response. Invalid response causes a re-prompt.

```

int num;

do {
    printf ("Enter value:\n");
    scanf ("%d", &num);

    if (num < 1)
        printf ("too small, re-enter\n");
    else if (num > 100)
        printf ("too big, re-enter\n");
}
while ((num < 1) && (num > 100));

```

## for loop

- for (initialization; test; increment;)
  - statement
- initialization;
  - while (test) {
  - statement
  - increment;
  - }

```

/* Replace tabs by spaces */
int i, tab;
char line[];

for ( i = tab = 0; i < MAX; i++ ) {
    if ( line[i] == '\t' ) {
        tab++;
        line[i] = ' ';
    }
}

```

- Simplest loop, increment a counter:
  - for (i = 0; i < n; i++)
- Infinite loop – run until terminated:
  - for (i = 0; ; i++)
  - printf ("%d \n", i);

## Example

```

/* GETC.C: This program uses getchar to read a single line of input
from standard input, places this input in buffer, then terminates
the string before printing it to the screen. */

#include <stdio.h>

void main( void )
{
    char buffer[81];
    int i, c;

```

```
printf( "Enter a line: " );

/* Read in single line from "stdin": */
for (i = 0; (i < 80) &&
      ((c = getchar()) != EOF) && (c != '\n');
      i++)

    buffer[i] = c;

/* Terminate string with null character: */
buffer[i] = '\0';
printf( "%s\n", buffer );
}
```

### Comma operator (,)

- expression1 , expression2
- A pair of expressions evaluated left to right
- Type and value of result is a type and value of expression2

### Example 1

```
int k = 0;
char a, b = 'A';

a = k++, b++;
// a equals to 'B'
```

### Example 2

```
/* reverse string s in place */

char s[] = "Hello world!";
int c, i, j;

for (i = 0, j = strlen(s)-1;
      i < j; i++, j--)
{
    c = s[i];
    s[i] = s[j];
    s[j] = c;
}
```

### Unconditional branch

- continue                    forces the next iteration of the loop
- break                        terminates a loop or switch
- return                        return a value from a function
- goto                         proceed to marked statement

### continue statement

- Terminates execution of the successful statements in the loop and forces the next loop iteration
- Pass the control to conditional tests of the loop
- In a **for** loop, passes control to iteration part

- Forces a conditional test to perform sooner

```

/* look for a letter */
/* convert to upper case */
int c;
int letter = 0;

do {
    c = getchar();
    if ((c >= 'A') && (c <= 'Z')) {
        letter = c;
        continue;
    }
    if ((c >= 'a') && (c <= 'z')) {
        letter = toupper(c)
        continue;
    }
} while ((c!= EOF) && (!letter))

```

## break

- Terminates a loop or **switch** statement – causes the innermost enclosing loop or **switch** to be exited

```

#include <ctype.h>
/* look for a letter */
/* convert to upper case */

int c;
int letter = '\0';

while ((c = getchar()) != EOF)
{
    if ((c >= 'A') && (c <= 'Z')) {
        letter = c;
        break;
    }
    if ((c >= 'a') && (c <= 'z')) {
        letter = toupper(c)
        break;
    }
}

```

Often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

## goto statement

- Pass the control to the specified point – label
- identifier: - label definition
- Formally never necessary.
- Common use – to abandon processing in deeply nested structures

```

/* search for a common element */

// a is an array with n elements
// b is an array with m elements

    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if (a[i] == b[j])
                goto found;

found:
    printf("Common element %d at %d,%d",
           a[i],i,j);

```

## return statement

- return value;
- Terminates the function
- Return value to calling point

## Data Types

### Standard data types:

- char                    one byte (to hold one character)
- int                     an integer (of natural size for machine)
- float                  single-precision floating point number
- double                 double-precision floating point number

### Type size modifiers (for int)

- short                  short length integer
- long                   long length integer
- both can be used with or without int

### Type sign modifiers

- signed                 integer (i.e.  $-2^{length-1} .. 2^{length-1} - 1$ )
- unsigned              positive integer ( $0..2^{length} - 1$ )
- both can be applied to any integral type
- int is **signed** by default!
- char is signed on **some** machines

### Examples

- unsigned char                    – one byte, 0..255
- long                                – same as long int
- unsigned long int

### Typical type ranges (PC, Windows)

- actual ranges see <limits.h>

Type	Size (bytes)	Range
char	1	-128 .. 127
unsigned char	1	0 .. 255
int	2 or 4	at least -32,768 .. 32,767
unsigned int	2 or 4	at least 0 .. 65,535
short int	2	-32,768 .. 32,767
unsigned short int	2	0 .. 65,535
long int	4	-2,147,483,648 .. 2,147,483,647
unsigned long int	4	0 .. 4,294,967,295
float	4	$3.4 \cdot 10^{+/-38}$ (7 digits)
double	8	$1.7 \cdot 10^{+/-308}$ (15 digits)

## Signed integers

- highest bit used as a flag

## Negative numbers (2-complement system)

- to obtain  $-value$ :
  1. take *value*
  2. reverse all bits
  3. add 1
  4. set sign flag

## Examples:

1, obtaining a negative value:

(char) 1:

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

(char) -1:

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

2, value interpretation depends on type:

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

1	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---

127	127	0	1	1	1	1	1
-----	-----	---	---	---	---	---	---

-128	128	1	0	0	0	0	0
------	-----	---	---	---	---	---	---

-127	129	1	0	0	0	0	1
------	-----	---	---	---	---	---	---

-1	255	1	1	1	1	1	1
----	-----	---	---	---	---	---	---

## Type conversion is performed:

1. When a value of one type is assigned to a variable of a different type or an operator converts the type of its operand before performing an operation
2. When a value of one type is explicitly cast to a different type
3. When a value is passed as an argument to a function or when a type is returned from a function

## Integral promotion:

- A **char**, a **short int**, bit field, (all either signed or not), an object of **enum** when used in expression are converted to **int** (if it can represent all the values of original type), otherwise to **unsigned int**.

## Arithmetic conversion rules

- Bring operands of the operator into common type:
  1. If one operand is **double**, convert other to **double**
  2. Else, if one operand is **float**, convert other to **float**
  3. Else, perform an **integral promotion**, and:

4. If one operand is **unsigned long**, convert other to **unsigned long**.
5. Else, if one operand is **long** and other is **unsigned int**, convert both to **unsigned long**.
6. Else, if one operand is **long**, convert other to **long**.
7. Else, if one operand is **unsigned int**, convert other to **unsigned int**.
8. Otherwise, both operands have type **int**.

### Examples:

```
char ch;
int i, result;
long l;
float f;
double d;

result = (ch/i) + (f*d) - (f/l);
```

```
float    f = 1;           // = 1.0
double   d;
int      i = -2;
unsigned long ul = 5;    // = 5UL

d = i * ul;
// i converted to unsigned long
// result converted to double
// -2 * 5UL => 4294967286, not -10

d = ul + f;
// ul converted to float
// result converted to double
// d is 6.0
```

### Constant type modifiers:

Suffix	Example	Type
<b>L</b>	-2L	long int
<b>U</b>	2U	unsigned int
<b>UL or LU</b>	5UL	unsigned long int
. (decimal point used)	1.0	double
. and <b>F</b>	1.0F	float

### Type cast operator ( type )

- (type) expression
- explicitly convert the **value** of the expression to the given type

### Examples:

```
int a = 5, b = 2, n, m;
double d;

d = (double) a / b; // d is 2.5

n = (int) d / a;
m = (int) (d / a);
```

## Integral conversion:

### Conversion of values to unsigned type

- If type is broader, the bit-pattern of unsigned value is zero-filled, and signed value is sign-extended.
- If type is narrower, the high-order bits are truncated.
- The result is interpreted as an unsigned value.

### Conversion to signed type

- Bit-pattern is converted (i.e. same technique) so that value is unchanged
- If cannot be represented in the target type, then implementation-specific.

### Microsoft C specific demotion:

- ANSI 3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented, is implementation specific.
- When a long integer is cast to a short, or a short is cast to a char, the least-significant bytes are retained.
- Examples:
  - `char c = (char) 0x1234; // c = 0x34`
  - casting `-2 (0xFE)` to an unsigned value yields 254 (also `0xFE`).

### Floating to Integral conversion

- The fractional part is truncated. No rounding takes place in the conversion process.
- Truncation means that a number like 1.3 is converted to 1, and `-1.3` is converted to `-1`.
- Undefined if result is out of range

### Integral to Floating conversion

- Converts exactly, but precision may be lost.
- Undefined if result is out of range

# Console Input / Output

- Input and output is performed using library functions
- `<stdio.h>` - the header for the functions

## Reading characters

- `int getchar(void);`
- Input a character from console.
- Return character's ASCII code or EOF (-1).
- The function uses type **int** to handle end-of-file situation.
- However, the result (unless it is EOF) may be assigned to type **char**.

## Note:

- Many systems employ **line-buffered** input: the characters are accumulated in the buffer and passed to the program when ENTER is pressed

## Writing characters

- `int putchar(int c);`
- Writes a character (given character's ASCII code) to console at the current cursor position
- Returns the character written or EOF
- May be called using a character argument

## Input redirection, substitute file for keyboard

- `myprog < infile`
- - causes **myprog** to read characters from **infile** rather than from keyboard
- Redirection is on a system level, the program is unaware of the change

## Output redirection, substitute file for screen

- `myprog > outfile`
- - causes **myprog** to write characters to **outfile** rather than to screen
- Input and output redirections can be combined

## Pipelining

- `myprog1 | myprog2`
- - puts the standard output of a program **myprog1** to the standard input of program **myprog2**
- Pipelines may not be supported by the system

## Notes:

- The `<stdio.h>` library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline ('\n') character.
- If the system doesn't operate in this way, the library does whatever it necessary to make it appear as if it does. For instance the library may convert CR and LF to '\n' on input and back again on output.

## Formatted Console Input / Output

### Formatted Output:

- `int printf(const char *format [, argument]... );`
- The `printf` function formats and prints a series of characters and values to the standard output stream (**stdout**).
- The format argument consists of ordinary characters, escape sequences, and (if arguments follow format) format specifications. The ordinary characters and escape sequences are copied to **stdout** in order of their appearance.
- The format string must contain specifications that determine the output format for every argument that follow the format string.
- Functions return the number of characters printed or a negative value if an error occurs.

### Example:

- `printf("Number is %d, string is %s.\n", 10, "\Hello\");`
  - Number is 10, string is "Hello".

### Format specifications

- % [flags] [width] [.precision] [format modifier] type // no spaces in between
- Other symbols printed as is. To print %, use %%
- type
  - determines the way the associated argument is interpreted
- flags
  - control justification of output and printing of signs, blanks, decimal points... More than one flag can appear in a format specification.
- width
  - optional number that specifies the minimum number of characters output.
- precision
  - optional number that specifies the maximum number of characters printed for all or part of the output field, or the minimum number of digits printed for integer values.
- format modifier
  - optional prefixes that specify the size of argument.

## Types (printf)

%c	Character
%d or %i	Signed decimal integer
%u	Unsigned decimal integer
%o	Unsigned octal integer
%x (%X)	Unsigned hexadecimal integer using abcdef (ABCDEF)
%f	Floating point (double)
%e (%E)	Floating point in scientific notation: <i>d.dddde ddd (E)</i>
%g (%G)	%e (%E) or %f whichever compact, trailing zeros truncated
%p	Pointer
%s	String (pointer to character array, until '\0' or precision)

## Format modifier (printf)

h	Short (for decimals)
l	Long (for decimals)
L	Long (for floating point)

## Special type (printf)

%n	Stores in argument ( <i>i</i> th*) number of characters printed so far
----	------------------------------------------------------------------------

## Flags (printf)

Flag	Meaning	Default
-	Left align the result	Right align
+	Prefix the signed value with a sign	Sign only for negative value
0	Pad with zeros (not for integer)	Pad with spaces
space	Prefix the signed positive with space	-
#	Prefix octal/hexadecimal with 0/Ox	-
#	Force floating point value contain decimal point	Decimal point appears only if digits follow

## Width specification (printf)

- Minimal number of characters printed for the argument (Never causes truncation!).
- If the width is \*, an **int** argument from the list supplies the value.

## Precision specification (printf)

Type	Meaning	Default
<i>integer</i>	Minimum number of digits to be printed (if less, result is padded with zeros)	1
<i>floating point</i>	The precision specifies the number of digits to be printed after the decimal point. No decimal point is printed if precision is 0.	6
%g %G	The maximum number of significant digits printed.	6
<i>string</i>	The maximum number of characters. Characters in excess of given <i>precision</i> are not printed.	Until '\0'

## Formatted Input:

- **int scanf(const char \*format [, argument] ... );**
- The **scanf** function reads data from the standard input stream **stdin** and writes the data into the location given by argument
- An input field is defined as all characters up to the first white-space character or up to the first character that cannot be converted according to the format specification, or until the field width (if specified) is reached.

- The format argument specifies the template and interpretation for the input
- Each argument must be a pointer to a variable of a type that corresponds to a type specifier in format.
- Return the number of fields successfully **assigned**
- The return value is **EOF** for an error or if the **end-of-file** character or the **end-of-line** character is encountered in the first attempt to read a character

### Example:

- `scanf("%d %d", &a, &b);`
- Reads two integers (**int a** and **int b**) separated by any blank symbol

### Format specifications (scanf)

- `% [*] [width] [format modifier] type` // no extra spaces in between
- The format specifies the interpretation of the input and can contain:
  - **White-space characters:** blank (' '); tab ('\t'); or newline ('\n'). Cause to skip the sequence of white-space characters in the input (if any) until next non-white-space character.
  - **Non-white-space characters,** except for the percent sign (%). Cause to read, but not store, matching characters. If the next character does not match, **scanf** terminates.
  - **Format specifications,** introduced by the percent sign (%). Cause to read appropriate data and store into the argument.
- The format is read from left to right. Characters outside format specifications are expected to match the sequence of characters in **stdin**; the matching characters in **stdin** are scanned but not stored. If a character in **stdin** conflicts with the format specification, **scanf** terminates, and the character is left in **stdin** as if it had not been read.

### Types (scanf)

Character	Type of input	Type of argument
c	Character. No skip over white space. To read non-white character use %1s	char*
d	Decimal integer	int*
i	Decimal, hexadecimal, or octal	int*

u	Unsigned decimal integer	unsigned int*
o	Octal integer (with or without leading 0)	int*
x	Hexadecimal integer (with or w/o leading 0x)	int*
f, e, E, g, G	Floating point (double)	float*
s	String, up to first white-space character	char* or char []
[ ]	String using scanf	char* or char []

### Format modifier (scanf)

h	Short (for decimals)	l	Double (for floating point)
l	Long (for decimals)	L	Long double (for floating point)

### Width specification (scanf)

- Maximum number of characters to be read for argument
- If applied with %c (%nc), reads n characters into a character array

### Scanset [ ]

- Substitutes for %s, read strings not delimited by space characters.
- Defines a set of characters accepted for the string.
- **Example:** %[a-z]: read small letters
- Input is read up to the first character that does not appear in the scanset.
- If first character is caret (^), defines characters not belonging to the scanset
- **Example:** %[\^\\n]: read all symbols to the end of line.

### An asterisk (\*) following %

- Suppresses assignment of the next input field.
- The field is scanned but not stored.



- `char str[81];` - a string of 80 characters
- A string constant "Hello" is zero-terminated automatically by a compiler

### String functions:

- `#include <string.h>`

Function	Meaning
<code>strcpy(s1,s2)</code>	Copies <i>s2</i> into <i>s1</i>
<code>strcat(s1,s2)</code>	Add <i>s2</i> to the end of <i>s1</i>
<code>strlen(s)</code>	Returns the length of <i>s</i>
<code>strcmp(s1,s2)</code>	Performs lexicographic case-sensitive comparison, returns 0 if strings are same, less than 0 if <i>s1</i> < <i>s2</i> , greater than 0 if <i>s1</i> > <i>s2</i> (for locale specific comparison use <code>strcoll</code> )
<code>strchr(s,ch)</code>	Returns a pointer to the first occurrence of ( <i>int</i> ) <i>ch</i> in <i>s</i>
<code>strstr(s1,s2)</code>	Returns a pointer to the first occurrence of <i>s2</i> in <i>s1</i>

### Two-dimensional array

- an array of one-dimensional arrays
- `type variable [size1] [size2];`
- `int d[2][5];` declares the following array:

		⊗		

- `d[1][2]` is referencing the element ⊗ above
- rightmost index changes faster!

### Example:

```
int d[2][5];
int row, col;

for (row = 0; row < 2; row++)
    for (col = 0; col < 5; col++)
        d[row][col] = row*5 + col;
```

- Result:

0	1	2	3	4
5	6	7	8	9

- In memory:

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

- Amount of memory required to hold an array is:  
`total bytes = sizeof(type) * size1 * size2;`

## Array of strings (uncommon)

- `char str_array[10][81];`
- - declares the array of 10 80-character strings
- Access individual string as: `str_array[number]`
- Ex: `scanf ("%80s", str_array[i]);`

## Array initialization

- `type variable [size1] ... [sizeN] = {value list};`
- `int num[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};`
- Character array allows initialization using string constant:
  - `char variable [size] = "string";`
  - `char str[4] = "ABC";`
  - `char str[4] = {'A', 'B', 'C', '\0'};`
- Multi-dimensional array:
  - `int d[2][5] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};`
  - `int d[2][5] = {  
    {0, 1, 2, 3, 4},  
    {5, 6, 7, 8, 9}  
};`
- Unsized array initialization:
  - `char str[] = "Read error";`

## POINTERS AND Arrays

### Pointer assignments

```
int x=99;
int *p1, *p2;

p1 = &x;           // address of x
p2 = p1;           // address of x
(*p1)++;           // x++;
printf ("%i", *p2); // prints: 100
```

### Pointer conversions

- `void *` **generic** pointer
- refers to raw memory (for memory initialization)
- can be assigned to/from any pointer type
- other *explicit* pointer conversions are undefined!

### Pointer arithmetic

- Allowed: **addition** and **subtraction** of constants, and **subtraction** of one pointer from another.
- Considering previous example:

- `p1++` makes pointer `p1` to point to the next integer in memory located after the one pointed by `p1`;
- `p2+=2` makes pointer `p2` to point to the second-next integer in memory after `*p2`;
- pointer subtraction results in a number of objects of base type that separate, i.e. `p2-p1` results in 1.

## Pointer Comparisons

- pointers can be compared to each other if belong to the same array

## Arrays and Pointers

- arrays and pointers are closely related
1. Array name refers to a pointer to the first element of array
    - `char str[81] ⇒ str ≅ &str[0]`
  2. Array element may be referenced two ways:
    - using array indexing or pointer arithmetic
    - `char str[81] ⇒ *(str+5) ≅ str[5]`
    - `int d[2][5] ⇒ *((int*)d+1*5+2) ≅ d[1][2]`

## Example:

```
void putstr(char *s)
{
    int k, *p;

    for (k = 0; s[k] ; k++)
        putchar (s[k]);

    p = s; // p = &s[0];
    while (*p)
        putchar (*p++);
}
```

## Initializing pointers

- Pointer must be initialized prior to use
- **Convention:** a pointer currently not pointing to a valid memory location is given the value 0
- NULL a special constant for pointer initialization

```
int *p = NULL; // = 0;
```

- However you can not assign a value by this pointer

```
int *p = NULL;
*p = 10; // WRONG!
```

## Initializing pointers with string constants

- `char str[] = "Read error";`
- creates an array `str[11]` and initializes it

```
str:  ['R'  'e'  'a'  'd'  ' '  'e'  'r'  'r'  'o'  'r'  0]
```

- you can later store another data in that array



- allocates memory, returns a pointer to the 1<sup>st</sup> byte of allocated memory or 0 (NULL) if allocation fails
- `void free (void* ptr);`
- returns previously allocated memory to the system
- `ptr` is a pointer to memory previously allocated with `malloc`. Never call `free` with an invalid argument!

## Dynamically allocated arrays

- You can operate on memory returned by `malloc` as if it were an array:

```
int *p;

p = (int*) malloc (sizeof(int)*100);
if (!p) // if (p!=NULL)
{
    printf ("Out of memory");
    terminate function or program
}
```

## Multidimensional arrays case:

1. Declare a pointer that specifies all but the leftmost array dimension
2. Allocate sufficient memory

## Example

- Recalling:
- `int d[2][5];` statically declares an array:


- Dynamic allocation:

```
int (*d)[5];

p = (int(*)[5]) malloc (2*5*sizeof(int));
```

## Typical errors using pointers

1. Type mismatch

```
int x, *p;

x = 10;
p = x; // ERROR, use either &x or *p
```

2. Not allocated pointer

```
int x, *p;

x = 10;
*p = x; // ERROR, missing malloc
```

3. Out of memory condition

```
int x, *p;

x = 10;
```

```
p = (int*) malloc (sizeof(int));
*p = x; // ERROR, if p is NULL
```

#### 4. Comparing pointers not pointing to common object

```
int x, y;
int *p, *q;

p = &x; q = &y;
if (p < q) // ERROR, result undefined
```

#### 5. Indexing adjacent arrays as one

```
int a[10], b[10];
int k;
int *p;

p = a;
for (k = 0; k < 20; k++) // ERROR
    *p++ = k;
```

#### 6. Exceeding the allocated memory

```
int k;
int *p;

p = (int*) malloc (sizeof(int)*10);
for (k = 0; k <= 10; k++)
    p[k] = k; // ERROR on last iteration

free(p);
```

#### 7. Leaving memory allocated or calling free with wrong argument

```
int k;
int *p;

p = (int*) malloc (sizeof(int)*10);
for (k = 1; k < 10; k++)
    *p++ = k;

free(p); // ERROR, p is changed
```

## FUNCTIONS

### General Form of a Function

- ret type function name (parameter list)  
{  
    statements;  
    ...  
    return expression;  
}

- ret type specifies the type of data that function returns, or void if returns nothing;
- **implicit int rule:** if ref type is not specified assumed int (not valid in C99 and C++);

- parameter list is a coma separated list of variables passed to the functions and their types (type specified FOR EACH variable), or void if function does not accept any parameters;
- Ex: double product (int x, int y)
- non-void function returns a value using return

## Function's scope

- Function defines a **block scope**:  
function's code is private to that function and cannot be accessed by any statement outside of it and cannot interact with the code or data (unless global variables) defined outside of it.
- No goto inside or outside of function is allowed

## Global variables

- variables that may be used by any code;
- placed outside of any function (usually at top);
- storage allocated on compilation time;
- avoid using global variables unless absolutely necessary! (I.e.: tracking off memory allocation)

## Local variables

- variables that declared inside of a function (or block of code enclosed in { } );
- exist only while block is executing;
- declared in the beginning of a block after '{' (C99 and C++ - variables declared anywhere);
- if declared with the same name as variable in enclosing block, then **hides** this outer variable;
- storage allocated on stack.

## Formal Parameters

- variables that accept the function arguments;
- declared after the function name inside of ( ) .

```
int a = 10;                // global

void func (int x)         // formal
{
    int b = 15;          // local
}
```

## Call by Value (the only one used in C)

- Copies the value of argument into the formal parameter of the function;
- When parameter is changed, the argument is not!

```
int sqr (int k)
{
    k = k*k
    return k;
}
```

```

int main (void)
{
    int k = 10;
    int a;
    a = sqr (k); // a = 100, k unchanged
}

```

## Call by Reference

- Method which reflects change of formal parameter on the argument:
  1. Pass the address of the argument as a parameter;
  2. Use the reference to access the argument inside of the function

```

void swap (int *x, int *y)
{
    int temp;
    temp = *x; // save value of 1st
    *x = *y; // put 2nd into 1st
    *y = temp; // put 1st into 2nd
}

int main(void)
{
    int a = 10, b = 20;
    swap (&a, &b) // set a = 20, b = 10
}

```

## Passing array to a function

- when array is used as a function argument, its address is passed to a function
- function is able to alter the content of array

```

void inc (int *ar, int len)
{
    int k;

    for (k=0; k<len; k++)
        ar[k]++;
}

int main(void)
{
    int a[5] = {1,2,3,4,5};
    inc (a,5); // set a to {2,3,4,5,6}
}

```

```
void inc (int ar[], int len)
```

## Parameters as constants

- Const pointer passed to function – prevent from modification of an array

```
void func (const char *str)
```

## Program example

```
#include <stdio.h>
int printstr(unsigned char *str)
{
    register int k;

    for (k = 0; *str ; k++)
        putchar (*str++);

    return k;
}

int main(void)
{
    unsigned char buf[100] = "";
    unsigned char *ptr;
    int c, l;
    register int k;

    ptr = buf;
    for (k=0; ((c=getchar())!=EOF) && (c!='\n'); k++)
    {
        if (k<99)
            *ptr++ = (unsigned char) c;
    }

    l = ptr - buf;
    *ptr = '\0';
    printf("Length = %i\n",l);

    l = printstr(buf);
    printf("\nPrinted = %i\n",l);
    return 0;
}
```

## Returning values

- Function terminates execution after the last statement or after the statement return

## What function should return?

1. **Computational** function return its result
  - double sqrt (double x)
2. Function **manipulating** data returns the success/failure status
  - printf() returns number of character written
3. **Procedural** function does not return anything
  - void exit() terminates a program

## Returning pointers

- return specified pointer or generic pointer

```
char *match (char c, char *s) {
    while (c!=*s && *s) s++;
    return(s);
}
```

```

void *allocate (size_t bytes) {
    void *ptr;

    if ((ptr = malloc(bytes))!= NULL)
        allocated += bytes; // global
    return (ptr);
}

```

## return statement

- return expression;
- returns the value of expression, expression can be ignored if function is of type void.

## What does main() return?

- main() return integer to the calling process, therefore it must be declared as int main

## Function declaration (prototypes)

- Function should be declared prior to be used!
- Declaration is necessary when function is used in the program earlier than defined
- Function declaration is similar to definition, but parameter names are optional and declaration is terminated by ‘;’

```

char *match (char, char);
char *match (char c, char *s)
{
    while (c!=*s && *s) s++;
    return(s);
}

```

## Command line arguments

- the way to pass information into the program;
- myprog argument1 argument2 argument3 ...
- command line arguments are separated by spaces
- argc - holds the number of arguments;
- argv - array of character pointers (each element is pointing to com. line argument);
- argv[0] points to the full program name;
- if no arguments are specified argc is 1

```

#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    printf ("Program %s\n", argv[0]);
    if (argc >= 2)
        printf ("Argument %s\n", argv[1]);
    else
        printf ("No argument is given\n");
    return 0;
}

```

## Recursion

- when function call itself, a new set of parameters are allocated storage on the stack
- advantage: clearer and simpler programs
- disadvantage: may cause stack overrun

```
/* recursive */
int factr(int n)
{
    int answer;

    if (n==1) return (1);
    answer = factr(n-1) * n; // recursion
    return (answer);
}
```

```
/* non-recursive */
int factr(int n)
{
    int t, answer;

    answer = 1;
    for (t=1; t<=n; t++)
        answer = answer * t;

    return (answer);
}
```

## SCOPES AND DECLARATION

### Four scopes of C

- 1. File scope**
  - Starts at beginning of file (translation unit), ends with the end of it. Refers to identifiers declared outside of all functions. Identifiers are visible through the entire file; variables are global.
- 2. Block scope**
  - Begins with '{' , ends with '}'. Extends to formal parameters. Variables with block scope are local to their block.
- 3. Function prototype**
  - Identifiers declared in prototype are visible only within prototype.
- 4. Function scope**
  - Begins with opening '{' of a function, ends with '}'. Applies only to labels (used as target of goto statement)

### Definition

- causes storage to be allocated for the object.

### Declaration

- declares the name and type of an object.

## All files are compiled independently

### Linkage

#### 1. External

- identifier is available to all files constituting a program, by default function or global variable.

#### 2. Internal

- identifier is available in the current file only

#### 3. None

- local variables, available only in the block where defined

### Storage class modifiers

#### 1. extern

- **declares** the object with external linkage (the object should be actually **defined** in another file or later in the program)

#### 2. static

- specifies file scope object (global variable) to have an internal linkage (private for the compiled file, cannot be used outside of it);
- specifies local variable to have permanent storage, value is retained between function calls.

#### 3. register

- variable (local or formal parameter only) is held in CPU not in memory (optimized for speed);
- address (&) can not be obtained.

### Examples:

- Static global variable (as private object)

```
/* static global variable */
static size_t allocated = 0;

void *allocate(size_t bytes)
{
    ...
    allocated += bytes;
    ...
}

void deallocate(void *ptr, size_t bytes)
{
    ...
    allocated -= bytes;
    ...
}

size_t memory_use(void)
{
    return allocated;
}
```

## Examples:

- Static local variable (to retain value between calls)

```
#include <stdio.h>

int counter(void)
{
    static int k=1;
    return k++;
}

int main(void)
{
    printf("%i\n",counter()); // 1
    printf("%i\n",counter()); // 2
}
```

- Static local constant (to assign value only once)

```
#include <stdio.h>

int tempcheck(int temp)
{
    static const int t[4] = {40,70,95,120};
    int k;

    for (k = 0; k < 4; k++)
        if (temp <= t[k]) break;

    return k; // k in range 0 (cold) .. 4 (hot)
}
```

## Examples:

- Multi-file compilation, global variables, extern

```
/* file main.c */
extern size_t allocated;
void* allocate(size_t bytes);

int main()
{
    int *dyn_array;

    dyn_array = allocate (5*sizeof(int));
    printf ("Allocated %i bytes\n",
           (int)allocated);
    free (dyn_array);
    return 0;
}
```

```
/* file module.c */
size_t allocated;

void* allocate(size_t bytes)
{
```

```
} ...
```

Compilation:

```
>gcc -c -I. -Wall main.c
>gcc -c -I. -Wall module.c
>gcc -lm -o myprog main.o module.o
```

## Examples:

- Multi-file compilation using header files

```
/* file main.c */
#include "module.h"

int main()
{
    int *dyn_array;
    dyn_array = allocate (5*sizeof(int));
    ...
    return 0;
}
```

```
/* file module.c */
#include "module.h"

size_t allocated;
void* allocate(size_t bytes)
{
    ...
}
```

```
/* file module.h */
/* globals */
extern size_t allocated;
/* functions */
void* allocate(size_t bytes);
```

## FILE I/O

### Stream

- an abstraction level to work with files on a variety of physical devices from terminals to disk drives
- Even though devices are different, the buffered file system transforms each into a logical abstraction

### Text Stream

- Sequence of characters organized in lines terminated by new-line symbol ('\n'), which is optional on the last line (character translations may occur)

## Binary stream

- Sequence of bytes one-to-one corresponding to the bytes in the external device (no character translations occur)

## File I/O

- Functions, types and constants: `<stdio.h>`

## File pointer

- `FILE *`
- file is defines as a pointer to the structure `FILE`, which stores information about current state of stream; used in all stream I/O operations.

## Opening a file

- `FILE *fopen(const char *name, const char *mode)`
- opens a stream and links with a file (the number of files open at the moment is limited)
- initialize file position indicator to start of file
- name points to a string that make up a file name
- mode determines how the file is opened:

Mode	Meaning
r	Open file for reading
w	Create file for writing, overwrites if exists
a	Append a file
r+	Open file for read/write (must exist)
w+	Create file for read/write (overwrites)
a+	Open for read/append (create if necessary). Write only after end of file.

- Translation modifiers:

t	file opened in text mode (usually default)
b	file opened in binary mode

- `fopen` returns a pointer to file, or `NULL` (error)

## Example (open existing binary file for read/write):

```
FILE *fp;  
fp = fopen("test", "w+b");
```

## Closing a file

- `int fclose(FILE *stream);`
- closes a stream, writes any data remaining in the disk buffer to the file, and closes a file;
- returns 0 if success, EOF if error occurs;
- after file is closed in may be opened again;
- no file can be opened for writing twice!

## Writing/reading a character

- **int putc(int \*ch, FILE \*stream);**
  - writes character (byte 0..255) to file
  - returns the character written or EOF
- **int getc(FILE \*stream);**
  - reads a character (byte 0..255) from file
  - returns the character read or EOF (in case of either error or end-of-file)

## File error handling routines

- **int feof(FILE \*stream);**
  - tests for end-of-file on a stream;
  - Returns a nonzero value after the first read operation that attempts to read past the end of file
- **int ferror(FILE \*stream);**
  - tests for an error on a stream;
  - Returns 0 if no error has occurred on stream. Otherwise, it returns a nonzero value.

## Example

```
#include <stdio.h>
int main (int argc, char *argv[])
{
    FILE *fin, *fout;
    int ch;

    fin = fopen(argv[1], "rb");
    fout = fopen(argv[2], "wb");

    ch = getc(fin);
    while (!feof(fin))
    {
        putc(ch, fout);
        ch = getc(fin);
    }

    fclose(fin);
    fclose(fout);
    return 0;
}
```

## Rewinding file

- **void rewind(FILE \*stream);**
  - repositions file pointer to the beginning of a file;
  - clears the error indicators for the stream as well as the end-of-file indicator;
  - clears keyboard buffer, if applied to stdin,.

## Erasing files

- **int remove(const char \*filename);**
  - deletes a file from the device;

- Returns 0 if the file is successfully deleted. Otherwise, it returns -1.

## Flushing a stream

- **int fflush(FILE \*stream);**
- flushes a stream; stream remains open.
- If stream is open for output, it writes the contents of the buffer to the associated file.
- If stream is open for input, it clears the contents of the buffer.
- **fflush(NULL)** flushes all streams opened for output.
- **fflush** has no effect on an unbuffered stream
- Normally **fflush** returns 0, or EOF (if error).

## Reading/writing strings

- **int fputs(const char\* string, FILE \*stream);**
- writes string to the output stream at the current position (terminating null-character not written);
- returns nonnegative value if writing is successful.
- **char\* fgets(char\* str, int length, FILE \*stream);**
- reads at most length - 1 characters from the current stream position to string str stopping if a newline ('\n') or end of file is encountered;
- the newline character, if read, is included in the string!
- the str is terminate with null character,

## Reading/writing bytes

- **size\_t fread** (void\* buffer, size\_t size, size\_t count, FILE \*stream);
- reads up to count items of size bytes from the input stream and stores them in buffer (character translation occurs on textual files);
- returns the number of full items actually read.
- **size\_t fwrite** (const void\* buffer, size\_t size, size\_t count, FILE \*stream);
- writes up to count items, of size length each, from buffer to the output stream;
- returns the number of full items actually written.

## Formatted I/O on streams

- **int fprintf(FILE \*stream, const char \*format, ... );**
- **int fscanf(FILE \*stream, const char \*format, ... );**

## Random access I/O

- The file pointer can be repositioned (if supported) and random read/write operations are performed
- **int fseek(FILE \*stream, long offset, int origin);**
- moves the file pointer (if any) associated with stream to a new location that is offset bytes from origin. The next operation on the stream takes place at the new location
- The origin must be one of three values:
  - **SEEK\_SET** beginning of a file

- SEEK\_CUR      current position
  - SEEK\_END     end of file
- Returns 0 if successful and non-0 otherwise.
- long ftell(FILE \*stream);
- returns the current file position.
- NOTE: value returned by ftell may not reflect the physical byte offset for streams opened in text mode, because text mode causes carriage return–linefeed translation.
- On text files, use fseek only from beginning of file with the value returned by ftell!

## Standard streams

- When a program starts three streams are open automatically:
  - stdin             standard input (keyboard)
  - stdout            standard error (screen)
  - stderr            standard error (screen, for errors)

## Examples

```
int c;

/* similar constructions */
putchar(c);
putc(c, stdout);

/* similar constructions */
getchar();
getc(stdin);
```

```
fprintf(stdout, "Hello world!\n");
fprintf(stderr, "Error occurred!\n");
```

# PREPROCESSOR

## Preprocessing

- **Preprocessing** is the first phase in program translation (followed by the code generation and linking).
- The result of the preprocessing phase is a sequence of tokens that, taken together, define a “**translation unit**”.

## Preprocessor directives

- directives for the compiler that may affect the compilation process by expanding the scope of the programming environment
- Preprocessor directives always begin with #
- Only one preprocessor directive can appear on a given line. For example:
  - #include <stdio.h>
- Preprocessor directives are typically used to make source programs easy to change and easy to compile in different execution environments.

- Directives in the source file tell the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text.
- Preprocessor lines are recognized and carried out before macro expansion. Therefore, if a macro expands into something that looks like a preprocessor command, that command is not recognized by the preprocessor.

## #define

- **#define** identifier token string
- defines an identifier and a token string
- Preprocessor will substitute token string for all subsequent occurrences of an identifier in the source file; identifier however remains defined.
- The identifier is not replaced if it appears in a comment, within a string, or as part of a longer identifier.
- A **#define** without a token string removes occurrences of identifier from the source file.
- Can be used to give a meaningful name to a constant in your program:

```
#define DEBUG
#define ONE 1
#define TWO ONE+1
#define FILE_ERR "File error"
#define MAX_SIZE 100
#define EOL '\n'
```

```
printf("> " FILE_ERR);
    > File Error

printf("> " "FILE_ERR")
    > FILE_ERR
```

## #define (defining a macro)

- **#define** identifier (parameters) token string
- allows the creation of function-like macros
- formal parameters in parentheses will be replaced by the actual parameters (token-strings) passed to the macro-call
- formal parameters are separated by commas

```
#define ERR_V(n) printf("Error %i", (n))

#define putchar(c) putc((c), stdout)

#define random(min, max) \
    ((rand() % (int)((max) + 1) - (min)) \
    +(min) )
```

## #undef

- **#undef** identifier
- removes (undefines) an identifier previously created with **#define**.
- The **#undef** directive is typically paired with a **#define** directive to create a region in a source program in which an identifier has a special meaning.

```
#define ADD(X,Y)  (X) + (Y)
...
#undef ADD
```

## #if, #elif, #else, and #endif directives

- The **#if** (**#ifdef**, **#ifndef**) directive, with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file.
- If the expression you write (after the **#if**) has a nonzero value, the line group immediately following the **#if** directive is retained in the translation unit.

<b>#if</b> <u>constant expression</u> <u>statements</u>	
<b>#elif</b> <u>constant expression</u> <u>statements</u>	optional
<b>#else</b> <u>statements</u>	optional
<b>#endif</b>	

- constant expression must contain only:
  - integer constants
  - character constants
  - defined operator

```
    c = getchar();
    #if defined DEBUG
        printf("User typed ");
        putchar(c);
    #endif
```

## #ifdef and #ifndef directives

- **#ifdef** name                      **#if defined** name
- **#ifndef** name                      **#if !defined** name

## #error

- **#error** error message
- produces compiler-time error message

```
#if defined(__cplusplus)
#error C++ compiler required.
#endif
```

## #include

- tells the preprocessor to treat the contents of a specified file as if it had appeared in the source program at the point where the directive appears
- **#include** <file>
- searches for a file along the path specified by the /I compiler option, then along the path specified by the INCLUDE environment variable
- **#include** "file"

- searches for a file in the same directory as source program, then in the directories of all files that include that file, then as specified in previous case

```
#include <stdio.h>
#include "myheader.h"
```

## Preprocessor directives, notes:

- The number sign (#) must be the first nonwhite-space character on the line containing the directive;
- White-space characters can appear between the number sign and the first letter of the directive.
- Some directives include arguments or values.
- Lines containing preprocessor directives can be continued by immediately preceding the end-of-line marker with a backslash (\).
- Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file.

## Preprocessor Operators

- preprocessor-specific operators are used in the context of the **#define** directive

### Stringizing operator (#)

- converts macro parameters (after expansion) to string constants

```
#define ERR(s) printf("> " #s "\n")

ERR(Error Occurred);
ERR("Error Occurred");
```

- Output:

```
> Error Occurred
> "Error Occurred"
```

### Token-Pasting Operator (##)

- concatenates two tokens into one

```
#define S_NAME(s) s##_n
#define S_LENGTH(s) s##_len

char* S_NAME(mystr); // char* mystr_n;
int S_LENGTH(mystr); // int mystr_len;
```

## Preprocessor directives in header files

- You can use preprocessor directives to include header file only once in the translation unit

```
/* file prog.h */

#if ! defined(__PROG_H)
#define __PROG_H
```

```

/*
  Here follow function and variable
  declarations and type definitions
*/

#endif /* __PROG_H */

```

## Notes:

- Two string constants written one after another are concatenated automatically by the compiler, i.e.
  - `printf("AB" "CD");`
- translates into:
  - `printf("ABCD");`

## ADVANCED TOPICS

### Alternative operator ?

- test expression ? expression1 : expression2
- If test expression is evaluated as true (has non-zero value) then expression1 is evaluated and becomes the value of the entire expression, otherwise expression2 is evaluated and becomes the value of the entire expression.
- Useful in macro-definitions:

### Example

```

#define SIGN(x) ((x) > 0) ? 1 : -1

x = 10;
y = SIGN(x); // y = (x > 0) ? 1 : -1;

/* same as */

if (x > 0) y = 1;
else      y = -1;

```

## PROCESS CONTROL FUNCTIONS

### exit() function

<stdlib.h>

- **void exit(int return code);**
- terminates the program, forcing a return to the operating system, after performing cleanup operations.
- The value of the return code is returned to the calling process.
- **exit** calls, in last-in-first-out (LIFO) order, the functions registered by **atexit**, then flushes all file buffers before terminating the process.
- within **main**, **return expr;** is equivalent to **exit(expr);**

### abort() function

<stdlib.h>

- **void abort(void);**
- prints the message "Abnormal program termination", then terminates the current program and returns an exit code of 3 (prints the message "Abnormal program termination").

## perror() function

<stdio.h>

- **void perror(const char\* string);**
- prints an error message (string) to stderr.

## atexit() function

<stdlib.h>

- **void atexit( void (\*function) (void) );**
- Schedule void function (void) for execution at program termination
- **atexit()** (returns 0 if successful)

```
void done(void)
{
    printf("Terminating...\n");
}

int main(void)
{
    atexit(done);
    /* other statements here */
}
```

## assert() macro

<assert.h>

- **void assert(int expression);**
- Evaluates an expression and when the result is **false (0)**, prints a diagnostic message and calls abort to terminate the program.
- The diagnostic message includes the failed expression and the name of the source file and line number where the assertion failed.

```
int k;
char str[80], c;

/* some loop is here */
assert((k >= 0) && (k < 80));
str[k] = c;
```

## system() function

<stdlib.h>

- **int system(const char \*command);**
- Executes an operating system command.
- Returns the value returned by the interpreter or 0, if interpreter is not found.

## POINTERS TO FUNCTIONS

- In C, it is possible to obtain a pointer to function, which can be assigned, placed in arrays, passed to another function, returned by function, etc.

### Pointers to functions - definition

- **type (\*fptr) (parameter type list);**
- defines the variable **fptr** as the pointer to function that returns the value of a given **type** and accepts arguments of the types specified in the **parameter type list**:

## Example (using qsort)

- `void qsort(void *base, size_t num, size_t width, int (*comp)(const void *elem1, const void *elem2));`
- The `qsort` function implements a quick-sort algorithm to sort an array of `num` elements, `width` bytes each, in increasing order. The `base` is a pointer to the base of the array to be sorted.
- The `comp` is a pointer to a function that compares two array elements and returns a value specifying their relationship (return value is positive if `elem1 > elem2`, negative if `elem1 < elem2`, 0 if `elem1 == elem2`)
- `qsort` calls the `comp` routine one or more times during the sort, passing pointers to two array elements on each call

## Example:

```
/* QSORT.C: This program reads the command-line
 * parameters and uses qsort to sort them.
 * It then displays the sorted arguments.
 */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int scomp(const void *arg1, const void *arg2);

void main( int argc, char **argv )
{
    int i;
    /* Eliminate argv[0] from sort: */
    argv++;
    argc--;

    /* Sort remaining args using Quicksort */
    qsort((void*)argv, (size_t)argc,
          sizeof(char*), scomp);

    /* Output sorted list: */
    for(i = 0; i < argc; ++i)
        printf("%s ", argv[i]);
    printf("\n");
}

int scomp(const void *arg1, const void *arg2)
{
    /* Compare all of both strings: */
    return strcmp(*(char**)arg1, *(char**)arg2);
}
```

## VARIABLE-LENGTH ARGUMENT LISTS

### Declaring variable-length argument lists

- You can specify a function that has a variable number of parameters using ellipsis  
'...'

- A function that accepts variable list of arguments must have at **least one** actual parameter
- Requires <stdio.h> and <stdarg.h>
- `int printf (const char *s, ...);`

## Accessing optional arguments

- `va_list arg;`
- defines a variable `arg` pointing to list of arguments
- `va_start(va_list arg, prevparam);`
- macro to set `arg` to beginning of the list of optional arguments
- `type va_arg(va_list arg, type);`
- macro to retrieve current argument
- `void va_end(va_list arg);`
- macro to reset `arg`

## Example:

```

/* This functions computes an average among
 * given numbers. The list of numbers must
 * end with the value -1. Example:
 *
 * int ave;
 * ave = average (10,20,30,-1);
 */

int average( int first, ... )
{
    int count = 0, sum = 0, i = first;

    /* Define variable for the argument list */
    va_list marker;

    /* Initialize variable arguments. */
    va_start (marker, first);

    while( i != -1 )
    {
        sum += i;
        count++;
        /* Access variable arguments */
        i = va_arg (marker, int);
    }

    /* Reset variable arguments.      */
    va_end (marker);

    return( count ? (sum / count) : 0 );
}

```

## Interpreting complex declarations

- A simple way to interpret complex declarators is to read them “from the inside out,” using the following four steps:
  1. Start with the identifier and look directly to the right for brackets or parentheses (if any).

2. Interpret these brackets or parentheses, then look to the left for asterisks.
3. If you encounter a right parenthesis at any stage, go back and apply rules 1 and 2 to everything within the parentheses.
4. Apply the type specifier.

```
char *( *(*var)() ) [10];
  ^   ^  ^ ^ ^ ^   ^
  7   6  4 2 1 3   5
```

- In this example, the steps are numbered in order and can be interpreted as follows:
  1. The identifier var is declared as
  2. a pointer to
  3. a function returning
  4. a pointer to
  5. an array of 10 elements, which are
  6. pointers to
  7. **char** values.

## **ABSTRACT DATA TYPES**

### **Custom data types:**

- structure – grouping of variables under same name (aggregate type)
- union – define same memory location as two or more different types of variables
- enumeration – list of named integer constants
- typedef keyword – defines new type name

### **Structure**

- a collection of variables (called as *members* or *fields*) referenced under one name:
  - struct structure type

```
{
    type member;
    // ...
};
```
  - Structure declaration forms a template used to define structure variables:
    - struct structure type structure variables;
  - Declaration of structure type and definition of structure variables can be combined:
    - struct structure type

```
{
    type member;
    // ...
} structure variables;
```

### **Accessing structure members**

- Individual members of the structure variable are accessed using dot (.) operator:
  - structure variable . member;
- It references a member as an individual variable.

## Example

```
struct PERSON      /* Declare PERSON structure */
{
    int    age;          /* Declare members */
    char   sex;
    char   name[25];
    char   family[50];
};

/* Define objects of type struct PERSON */
struct PERSON brother, sister;

/* Assign values to members */
sister.age    = 13;
sister.sex    = 'f';
strcpy (sister.name,"Paula");
strcpy (sister.family,"Smith");

brother.sex   = 'm';
brother.age   = sister.age + 5;
strcpy (brother.name,"Matti");
strcpy (brother.family,sister.family);
```

## Structure assignments

- Content of one structure can be copied to another structure of the same type using assignment (=)

```
struct PERSON twinsister;
twinsister = sister;
strcpy (twinsister.name, "Marja");
```

## Passing an entire structure to function

```
void PrintPerson (struct PERSON person)
{
    printf ("PERSONAL INFORMATION:\n");
    printf ("Name: %s %s\n", person.family,
            person.name);
    printf ("Sex: %c\n",person.sex);
    printf ("Age: %i\n",person.age);
}
```

## Returning an entire structure from function

```
struct PERSON CopyPerson (struct PERSON x)
{
    struct PERSON y;
    y = x;          /* modify y if necessary */
    return y;
}

/* usage */
twinsister = CopyPerson (sister);
```

## Pointer to structure

- Structure address can be obtained and used in pointer operations
- Structure members can be accessed through the pointer using arrow (->) operator

```
struct PERSON *person;
person = &sister;
person->age++; // same as sister.age++
```

- A more efficient way to pass structure to function is by using pointer to structure:

```
void PrintPerson (struct PERSON *p)
{
    printf("PERSONAL INFORMATION:\n");
    printf("Name: %s %s\n",p->family, p->name);
    printf("Age: %i\n",p->age);
}
```

```
struct PERSON * CopyPerson (struct PERSON *x)
{
    struct PERSON *y;
    y = malloc (sizeof (struct PERSON));
    *y = *x;      /* modify *y if necessary */
    return y;
}
```

```
struct PERSON *p_twinsister;
p_twinsister = CopyPerson (&sister);
```

## Structures within structures

- Structure members can be structures as well:

```
struct FAMILY {
    struct PERSON husband;
    struct PERSON wife;
    struct PERSON *children;
} Smith;

Smith.wife.age = 30;
```

## Array of structures

- Structures can be arrayed

```
struct PERSON list[100];

struct PERSON *plist;
plist = malloc(100*sizeof(struct PERSON));
```

- To access specific structure, index array name:

```
plist[5].age = 25;
```

## Union

- a memory location shared by two or more different types of variables:

➤ union union type

```
{  
    type member;  
    // ...  
} union variables;
```

- Compiler allocated enough storage to hold largest member of the union

```
union VARIABLE {  
    int number;  
    char string[81];  
} x;  
  
if (scanf("%i",&x.number)==1)  
    printf("Number is %I \n",x.number);  
  
else if (scanf("%80s",&x.string)==1)  
    printf("String is %s \n",x.string);  
  
else  
    printf("No input\n");
```

## Enumeration

- a set of named integer constants (*enumerators*)
- enum enum type {enum list} variable list;
- By default, the first enumerator has a value of 0
- Each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator
- Enumerator's values can be similar
- Enumerator's names should be unique within the scope of declaration and are treated as constants

```
enum Days {  
    saturday,          // saturday = 0 by default  
    sunday = 0,       // sunday = 0 as well  
    monday,           // monday = 1  
    tuesday,          // tuesday = 2  
    wednesday,        // etc  
    thursday,  
    friday  
} today;  
  
today = monday;
```

- An enumerator can be promoted to an integer value NOT VICE VERSA!

```
if (today != 0)  
    printf ("Today is working day!\n");
```

## Defining new data types (typedef)

- A new name for the **existing data type** can be defined using typedef keyword:
- typedef existing type type name;

- where existing type is any valid data type and type name is the new name for this type

```
typedef unsigned long ULONG;
ULONG u1; // unsigned long u1;

typedef unsigned char BYTE;
typedef *int PINT;
typedef int BOOLEAN;
typedef struct tagPERSON
{
    int age;
    char name[25];
} PERSON;
PERSON student; // struct tagPERSON student;

typedef int (*FUNCPTR)();
// FUNCPTR - pointer to function returning int

FUNCPTR funcarr[10]; // int (*funcarr[10])();
```

- typedef declarations do not introduce new types – they introduce new names for existing types.
- Using type declaration is easier to port programs.

## DYNAMIC DATA STRUCTURES

### Linked LISTS

- data structure consisting of items linked together, so that each item in the list has a **link** (pointer) to the **next** item;
- items of **double linked list** have also link to a **previous** item;

#### Advantages (flexibility):

- The list items can be located anywhere in memory (not necessary continuously).
- The length of the list can be varied.
- The items can be added to the list and removed from the list in any order.

#### Disadvantages (no indexing + memory overhead):

- Items of the list cannot be accessed directly.  
They must be located by traversing the list from the head (or tail).
- Additional memory overhead to store link pointers

