Computers Programming
Course 5

Iulian Năstac

Recap from previous course
Classification of the programming languages

- **High level** (Ada, Pascal, Fortran, etc.)
  - programming languages with strong abstraction from the details of particular computer
- **Medium level** (C, C++, FORTH, etc.)
- **Low level** (assembly languages)
  - programming languages that provide little or no abstraction from a computer's instruction set architecture

Recap

C programming language

- 1966 Martin Richards (University of Cambridge) developed **BCPL** (Basic Combined Programming Language)
- 1969 **Ken Thomson** with contributions from Dennis Ritchie – **B programming language**
- 1969-1973 **Dennis Ritchie** – **C programming language**
- 1978 **Dennis Ritchie** and **Brian Kernighan** had elaborated a famous book, "The C Programming Language".

Recap

The main properties of C programming language

1. Portability
2. Data types
3. Errors control
4. Work at assembler level
5. Few keywords
6. Structured language
7. Programmers' language

Recap

The structure of a C program

- global statements:
  - inclusions of header files
  - statements of constants and global variables
  - declarations of local functions
- function **main()**
- other functions

Recap

C preprocessor

- The preprocessor provides the ability for:
  - **inclusion of header files**
  - **macro expansions**
  - **conditional compilation**

Constants in C Language

- Constants can be very useful in C programming whenever you need a value that is repeated during the program

- Ex:

  #define PI 3.14159

- Declaring a constant allows you to quickly and easily change a value that is used throughout the program simply by changing the initial declaration.

Data types

- five basic arithmetic type specifiers:
  - char
  - int
  - float
  - double

  - void

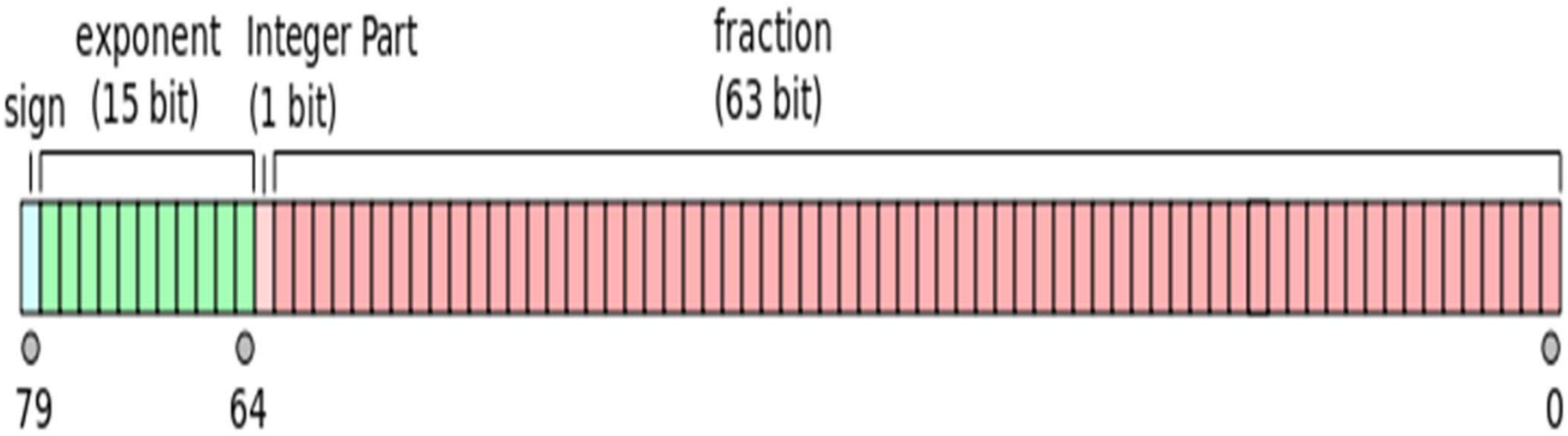
- optional specifiers:
  - signed,
  - unsigned
  - short
  - long

Type	Explanation
char	Smallest addressable unit (8 bits) that can contain basic character set. It is an integer type. Actual type can be either signed or unsigned depending on the implementation.
signed char	Same size as **char**, but guaranteed to be signed
unsigned char	Same size as **char**, but guaranteed to be unsigned
short **short int** **signed short** **signed short int**	***short*** signed integer type. At least 16 bits in size
unsigned short **unsigned short int**	Same as **short**, but unsigned
int **signed int**	Basic signed integer type. At least 16 bits in size
unsigned **unsigned int**	Same as **int**, but unsigned

Type	Explanation
long **long int** **signed long** **signed long int**	*long* signed integer type. At least 32 bits in size
unsigned long **unsigned long int**	Same as long, but unsigned
long long **long long int** **signed long long** **signed long long int**	*long long* signed integer type. At least 64 bits in size (specified since the C99 version of the standard).
unsigned long long **unsigned long long int**	Same as long long, but unsigned (specified since the C99 version of the standard).

Type	Explanation
float	Single-precision floating-point format is a computer number format that occupies 4 bytes (32 bits) in computer memory and represents a wide dynamic range of values by using a floating point.
double	Double-precision floating-point format is a computer number format that occupies 8 bytes (64 bits) in computer memory and represents a wide dynamic range of values by using floating point.
long double	Extended precision floating-point type. Unlike types float and double, it can be either 80-bit floating point format, or IEEE 754 quadruple-precision floating-point format if a higher precision format is provided.

long double



The 80-bit floating point format was widely available by 1984 after the development of C and similar computer languages, which initially offered only the common 32- and 64-bit floating point sizes.

Some C versions

Type	Option	Approximate size in bytes	Operating range
char	*char*	8	-127 ÷ 127
	unsigned char	8	0 ÷ 255
	signed char	8	-127 ÷ 127
int	*int* (signed int, short int, signed short int)	16	-32767 ÷ 32767
	unsigned int (unsigned short int)	16	0 ÷ 65535
	long int (signed long int)	32	-2.147.483.647 ÷ 2.147.483.647
	unsigned long int	32	0 ÷ 4.294.967.295
float	*float*	32	$\sim +/-3.4\times10^{-38} \div +/-3.4\times10^{38}$
double	*double*	64	$\sim +/-1.7\times10^{-308} \div +/-1.7\times10^{308}$
	long double	80 ÷ 128	$\sim +/-3.4\times10^{-4932} \div +/-3.4\times10^{4932}$

Notes:

- The actual size of integer types varies by implementation.
- The standard only requires size relations between the data types and minimum sizes for each data type.
- the **long long** is not smaller than **long**, which is not smaller than **int**, which is not smaller than **short**.

Notes:

- **char** size is always the minimum supported data type, all other data types can't be smaller.
- The minimum size for **char** is 8 bit, the minimum size for short and **int** is 16 bit, for **long** it is 32 bit and **long long** must contain at least 64 bit.
- Many conversions are possible in C.

Conversions in C

- Implicit conversion
  - If there are several types of data, then all of them will be converted to the larger one
- By assignation
  - when "equal" operator is involved, then the type of right side result is converted to the type of the left side.
- Logic conversion
  - when logic operators are involved

Explicit type conversion (cast conversion)

```
float a = 1.3;
double b = 2.5;
long c = 3.4;
int x;
…
x = (int)a + (int)b + (int)c;
```

Note:

- the result is x == 9
- if implicit conversion would be used (as with “x = a + b + c"), result would be equal to 7

Variables

- **Variables** are simply names used to refer to some location in memory.

- Types of variables:
  - Local variables
  - Global variables

A more comprehensive classification

- **Automatic variables**
  - variables which are allocated and deallocated automatically when program flow enters and leaves the variable's context
  - an automatic variable is a variable defined inside a function block
- **External variables**
  - variable defined outside any function block
- **Static local variables**
  - variables that have been allocated statically — whose lifetime extends across the entire run of the program
- **Register variables**
  - register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers

Automatic variable

- The term local variable is usually synonymous with automatic variable, since these are the same thing in many programming languages, but local is more general.
- Most local variables are automatic local variables, but static local variables also exist, notably in C.
- Automatic variables may be allocated in the **stack** frame of the procedure in which they are declared; this has the useful effect of allowing recursion and re-entrancy.
- The specific keyword is **auto** (not compulsory).

External variables

- As an alternative to automatic variables, it is possible to define variables that are external to all functions (variables that can be accessed by name by any function).

- An external variable may also be declared inside a function. In this case the **extern** keyword must be used, otherwise the compiler will consider it as a definition of a local variable, which has a different scope, lifetime and initial value. This declaration will only be visible inside the function instead of throughout the function's module.

- An external variable can be accessed by all the functions in all the modules of a program. It is a global variable.

Static local variables

- static variable is a variable that has been allocated statically — whose lifetime extends across the entire run of the program.

- This is in contrast to the more ephemeral automatic variables (local variables are generally automatic), whose storage is allocated and deallocated on the call stack.

- Static variables are declared as such with a special storage class keyword (**static**).

Register variables

- For efficiency, the optimizer will try to allocate some of local variables in processor registers.
- In most register allocators, each variable is either in a register or in memory.
- If a variable can not be assigned a register then all of the variable's usage, including its definition, is preceded by a load from memory.
- The specific keyword is **register** (not compulsory).
- The register must be big enough to host that variable.

Conversion format specifiers

- A **conversion format specifier** consist of both **%** (percent character) and **a terminating conversion character** that indicate the type of variable that is used.

Conversion format specifier	**Type of associate variable**
%c	single character
%s	string
%d	integer
%u	unsigned decimal integer
%p	pointer
%f	float
%lf	double
%Lf	long double

Note:

%5d – is associated with an integer that get maximum 5 character positions with right alignment

%-5d – is associated with an integer that get maximum 5 character positions with left alignment

Escape (backslash \) **character**

- C programming language specifies the doublequote character (") as a delimiter for a string literal.
- An escape character is a character which invokes an alternative interpretation on subsequent characters in a character sequence.
- The backslash (\) escape character typically provides special ways to treat the following part of a string or introduce a new character.

Note:

\" becomes " inside a string

Escape (backslash \) character	**Effect in a string**
\n	new line
\t	tab
\b	backspace
\v	vertical tabulation
****	backslash character
\/	slash

ASCII code

- The **American Standard Code for Information Interchange** (ASCII) is a character-encoding scheme originally based on the English alphabet that encodes **128 specified characters** (the numbers 0-9, the letters a-z and A-Z, basic punctuation symbols, control codes, and a blank space) into the **7-bit binary integers**.

ASCII includes definitions for 128 characters:

- 33 are non-printing control characters (many now obsolete)

- 95 printable characters, including the space (which is considered an invisible graphic)

USASCII code chart

Bits b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Row ↓ Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Example	Hex Code
ESC	1B
1 2 : 9	31 32 : 39
0	30
A B :	41 42 :
a b :	61 62 :

Standard I/O routines

- are substitute for missing of I/O instructions
- C programming language provides many standard library functions for input and output.
- These functions make up the bulk of the C standard library header <stdio.h> (also in <conio.h> and <stdlib.h>)

A. General output routines

1. printf function

- int printf (const char * format, ...);

- Writes the C string pointed by format to the standard output.
- If format includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.

- Ex.:

  printf("\n Result %d \n", x);

2. **puts** function

- int puts (const char * str);

- Writes the *C string* pointed by *str* and appends a newline character ('\n').

- Ex.:

printf(“Message \n”); ⇔ *puts(“Message”);*

3. **putchar** function

- int putchar (int character);

- Writes a *character* to the *output*

- Ex.:

```
#include <stdio.h>
int main()
{ char c;
for (c = 'A' ; c <= 'Z' ; c++) putchar(c);
return 0;
}
```

4. **putch** function

- int putch (int character);

- **putch** displays any alphanumeric characters to the standard output device. It displays only one character at a time.

B. General input routines

1. **scanf** function

- **int scanf (const char * format, ...);**
- Reads data and stores them according to the parameter *format* into the locations pointed by the additional arguments.

- Ex.:

  scanf("%d %d", &x, &y);

  scanf("%d, %d", &x, &y);

2. **gets** function

- **char * gets (char *str);**

- gets() accepts any line of string including spaces from the standard input device (keyboard).
- gets() stops reading character from keyboard only when the **enter** key is pressed.

```
#include<stdio.h>
#include<conio.h>


int main()
{
char a[20];
gets(a);
puts(a);

getch();
}
```

3. **getchar** function

- **int getchar (void);**

- getchar() accepts one character type data from the keyboard.

- Ex

variable_name = getchar();

```
#include <stdio.h>
int main ()
{       char c;
        printf("Enter character: ");
        c = getchar();

        printf("Character entered: ");
        putchar(c);
        return(0);
}
```

4. **getch** function

- **int getch (void);**

- ***getch()*** accepts only single character from keyboard.

- The character entered through ***getch()*** is not displayed in the screen (monitor).

5. **getche** function

- **int getche (void);**

- Like getch(), ***getche()*** also accepts only single character, but unlike getch(), ***getche()*** displays the entered character in the screen.

synthesis	
Output functions	**Input functions**
printf	scanf
puts	gets
putchar	getchar
putch	getch getche

Expressions

- An expression in a programming language is a combination of explicit values, constants, variables, operators, and functions that are interpreted according to the particular ***rules of precedence*** and of association for a particular programming language, which computes and then produces another value.
- This process, like for mathematical expressions, is called evaluation.
- The value can be of various types, such as numerical, string, and logical.

- primary expressions
- list of expressions

Order of operations

- In computer programming, the order of operations (operator precedence) is a rule used to clarify which procedures should be performed first in a given mathematical expression.

- The operators in C have a strict precedence level.

1	**() [] -> . ::**	Grouping, scope, array / member access
2	**! ~ - + * & sizeof *type cast* ++x - -x**	(most) unary operations, sizeof and type casts
3	*** / %**	Multiplication, division, modulo
4	**+ -**	Addition and subtraction
5	**<< >>**	Bitwise shift left and right
6	**< <= > >=**	Comparisons: less-than, ...
7	**== !=**	Comparisons: equal and not equal
8	**&**	Bitwise AND
9	**^**	Bitwise exclusive OR
10	**|**	Bitwise inclusive (normal) OR
11	**&&**	Logical AND
12	**||**	Logical OR
13	**?: = += -= *= /= %= &= |= ^= <<= >>=**	Conditional expression (ternary) and assignment operators
14	**,**	Comma operator

Automated conversion inside expressions

- Usually, the implicit conversion is involved (see a previous slide)

- ***Rule of implicit conversion in C***

The Rule of Implicit conversion

It works when a binary operator is applied to two operands.

The steps of the rule :

- First convert the operands of type ***char*** and ***enum*** to the type ***int***;
- If the current operator is applied to operands of the same type then the result will be the same type. If the result is a value outside the limits of the type, then the result is wrong (exceedances occur).
- If the binary operator is applied to operands of different types, then a conversion is necessary, as in the following cases:

- If one operand is **long double**, therefore the other one is converted to long double and **long double** is the result type.

- Otherwise, if one operand is **double**, therefore the other one is converted to **double** and **double** is the result type.

- Otherwise, if one operand is **float**, therefore the other one is converted to **float** and **float** is the result type.

- Otherwise, if one operand is **unsigned long**, therefore the other one is converted to **unsigned long** and **unsigned long** is the result type.

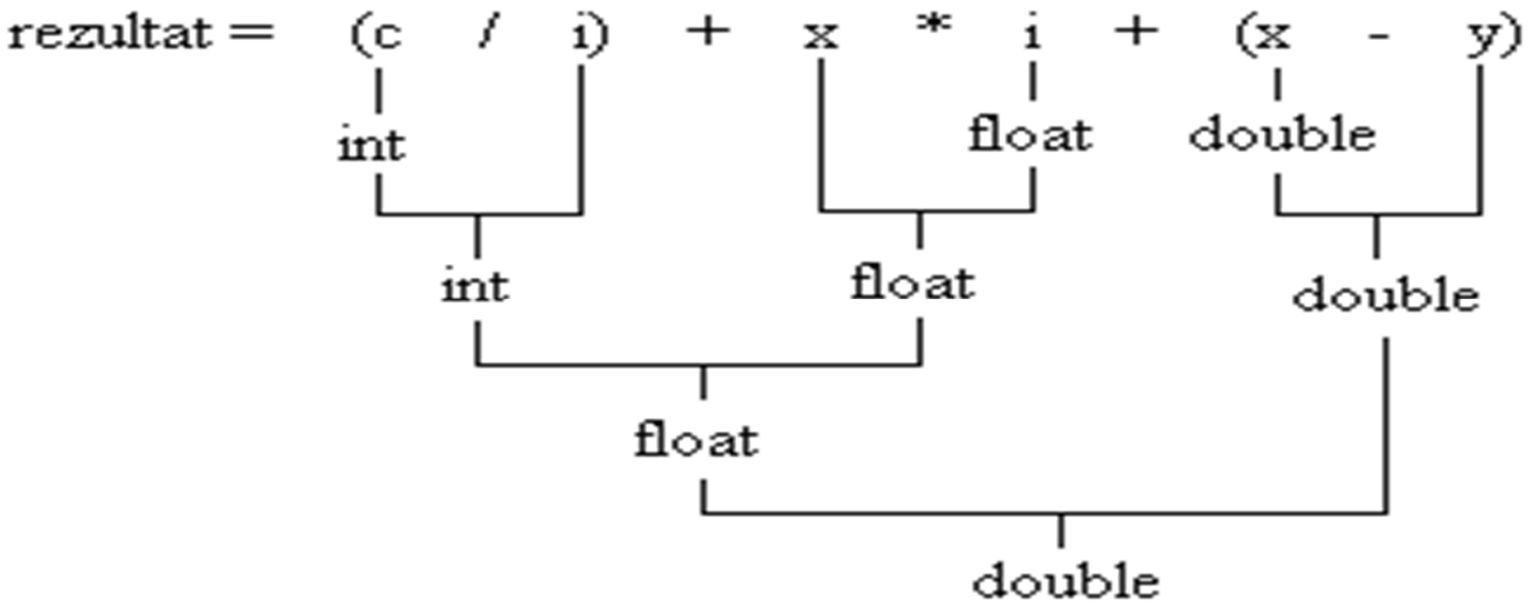
- Otherwise, if one operand is **long**, therefore the other one is converted to **long** and **long** is the result type.

- Otherwise, if one operand is **unsigned**, therefore the other one is converted to **unsigned** and **unsigned** is the result type.

Example:

- …

  float x;
  double y;
  char c;
  int i;
  . . .



Operators in C

- Programming languages typically support a set of operators, which differ in the calling of syntax and/or the argument passing mode from the language's functions.

- C programming language contains a fixed number of built-in operators.

1	**() [] -> . ::**	Grouping, scope, array / member access
2	**! ~ - + * & sizeof *type cast* ++x - -x**	(most) unary operations, sizeof and type casts
3	*** / %**	Multiplication, division, modulo
4	**+ -**	Addition and subtraction
5	**<< >>**	Bitwise shift left and right
6	**< <= > >=**	Comparisons: less-than, ...
7	**== !=**	Comparisons: equal and not equal
8	**&**	Bitwise AND
9	**^**	Bitwise exclusive OR
10	**|**	Bitwise inclusive (normal) OR
11	**&&**	Logical AND
12	**||**	Logical OR
13	**?: = += -= *= /= %= &= |= ^= <<= >>=**	Conditional expression (ternary) and assignment operators
14	**,**	Comma operator