Computer Programming

"Learn science first and then continue with the practice born from that science"

Leonardo da Vinci

Outline

- Who and What
- Problem solving process
  - Stages
- Algorithm
  - Definition, features, ways to describe
- Programming Languages
- C Introduction
  - Basic program structure
  - Data Types
  - Constants and Variables
  - Simple I/O
    - scanf
    - printf

Who and What

- Who:
  - Marius Joldoş: lectures + laboratory supervision
    - Marius.Joldos@cs.utcluj.ro
    - http://users.utcluj.ro/~jim/CP
  - Laboratory supervision
    - Raluca.Brehar@cs.utcluj.ro
    - Ciprian.Pocol@cs.utcluj.ro
    - Ion.Giosan@cs.utcluj.ro
    - Alex Cosma <alexx@gmail.com>

Who and What

- What: Computer Programming
  - Lectures: 2 hours/week, 14 weeks – every Monday, 12:00 hours, G. Bariţiu 26-28, room P03
  - Laboratory work: 2 hours/week, 14 weeks, as scheduled for each half-group, Observatorului 2, second floor
  - Self-study: 84 hours
  - Worth 5 credits (1/12 of a years total)

What: Computer Programming

- What you will acquire
  - As theory
    - To describe algorithms in pseudo-code
    - To modularize an algorithm
    - C(C++) foundations (w/o objects...)
    - Some algorithms
  - As abilities
    - How to design and implement algorithms in C(C++)
    - An adequate programming style
    - Master some algorithms (numeric, set)

What: Computer Programming

- Lectures
  - 1. Problem solving process. C Introduction
  - 2. Variables. Expressions
  - 3. Statements. Programming Style.
  - 4. Functions
  - 5. Modular programming
  - 6. Pointers and pointer operations. Memory allocation/de-allocation

What: Computer Programming

- Lectures
  - 7. Pointers and functions
  - 8. Character strings
  - 9. Data types: structure, union, enumeration
  - 10. Files
  - 11. Recursion
  - 12. C standard library
  - 13. Program samples
  - 14. Wrap-up and review

What: Computer Programming

- Laboratory work
  - 0. Using Codeblocks IDE.
  - 1. Standard I/O
  - 2. Expressions
  - 3. Statements I
  - 4. Statements II
  - 5. Functions
  - 6. Modular programming

What: Computer Programming

- Laboratory work
  - 7. Pointers
  - 8. Pointers and memory management
  - 9. Character strings
  - 10. Data types: structure, union, enumeration
  - 11. Files – high level
  - 12. Recursion
  - 13. Wrap up
  - 14. Laboratory test

What: Computer Programming

- Course info resources
  - Handouts (pdf, slides, on course web page)
  - Laboratory guide (pdf, on course web page)
  - http://users.utcluj.ro/~jim/CP
  - Moodle CMS site: https://labacal.utcluj.ro
  - Paul Deitel, Harvey Deitel, **C How to Program**, 6/E, Pearson Education, 2010
  - K.N. King, **C Programming. A Modern Approach**, W.W. Norton, 2008
  - Brian Kernighan, Dennis Ritchie The **C Programming** Language. Prentice Hall, 2 edition,. 1988
  - Stephen Prata, **C Primer Plus,** 5/E, Sams, 2004

What: Computer Programming

- To get your credits
  - Attend classes
  - Study, learn
  - Do the assignments
- Grading
  - Laboratory evaluation (LE) + written exams (WE)
    - In class tests (IT, 4*15min tests)
    - LE ≥ 5 and WE ≥ 5
    - Formula: 0.1*IT + 0.30*LE + 0.60*WE

Problem solving process

- Process from problem *specification* (i.e. what) to *concrete program*
- Steps:
  - Definition (what)
  - Analysis (what)
  - Algorithm development (how)
  - Coding and debugging
  - Testing
  - Documentation (in every step)

Rigorous approach to solving problems

- First *construct an exact model* in terms of which we can express allowed solutions.
  - Finding such a model is already half the solution. Any branch of mathematics or science can be called into service to help model the problem domain.
- Once we have a suitable mathematical model, we can *specify a solution in terms of that model*.

Defining a problem

- Basic requirements for a well-posed problem:
  - The known information is clearly specified.
  - We can determine when the problem has been solved.
  - The problem does not change during its attempted solution.

Algorithm

- ***Algorithm:*** Given both the problem and the device: precise characterization of a method of solving the problem, presented in a language comprehensible to the device.
- ***Properties***:
  - Application of the algorithm to a particular input set or problem description results in a ***finite*** sequence of actions.
  - The sequence of actions has a ***unique initial action***.
  - Each action in the sequence has a ***unique successor***.
  - The sequence ***terminates*** with either a solution to the problem, or a statement that the problem is unsolvable for that set of data.

Algorithm features

- An algorithm has
  - A *domain* (set of *input* values)
  - A *range* (set of *output* values)
- Input, output and intermediate data are denoted by *symbolic names* or *identifiers*
  - Using identifiers, and algorithm shows its applicability for any values with its domain

Algorithm example

- Convert a number from base 10 to base B, where $B \geq 2$
  1. Read the number to convert, n, and the target base, B.
  2. Assign to counter i a value of 1.
  3. Assign to C the quotient resulted from the division of n by B, and assign to R_i the remainder of that division.
  4. If C=0, goto step 8, otherwise continue with next step.
  5. Increment the counter i by 1.
  6. Assign the value C to n.
  7. Goto step 3.
  8. Write the remainders in reverted order, i.e. R_i, R_{i-1},... ,R_1.
  9. Stop.

Algorithm execution example

Step	*n*	*B*	*i*	*C*	R_1	R_2	R_3
–	×	×	×	×	×	×	×
1	1987	16	×	×	×	×	×
2	1987	16	1	×	×	×	×
3	1987	16	1	124	3	×	×
4	1987	16	1	124	3	×	×
5	1987	16	2	124	3	×	×
6	124	16	2	124	3	×	×
7	124	16	2	124	3	×	×
3	124	16	2	7	3	12	×
4	124	16	2	7	3	12	×
5	124	16	3	7	3	12	×
6	7	16	3	7	3	12	×
7	7	16	3	7	3	12	×
3	7	16	3	0	3	12	7
4	7	16	3	0	3	12	7
8	7	16	3	0	3	12	7

Algorithm features

- Repeated execution for different input data results in different sequences of states.
- A sequence of states an algorithm passes through is called a *computation*.
- Domain is infinite →infinite computation.
- An algorithm operates with variable and constant values.
- *n, B, C, i, R* are variables (their values changes)
- *n, B, C, i* occupy a single memory location
- *R* needs more locations (one for each of R_i, R_{i-1},... ,R_1)
- Algorithms execute sequentially as long as there are no jumps as results of decisions

Algorithm general requirements

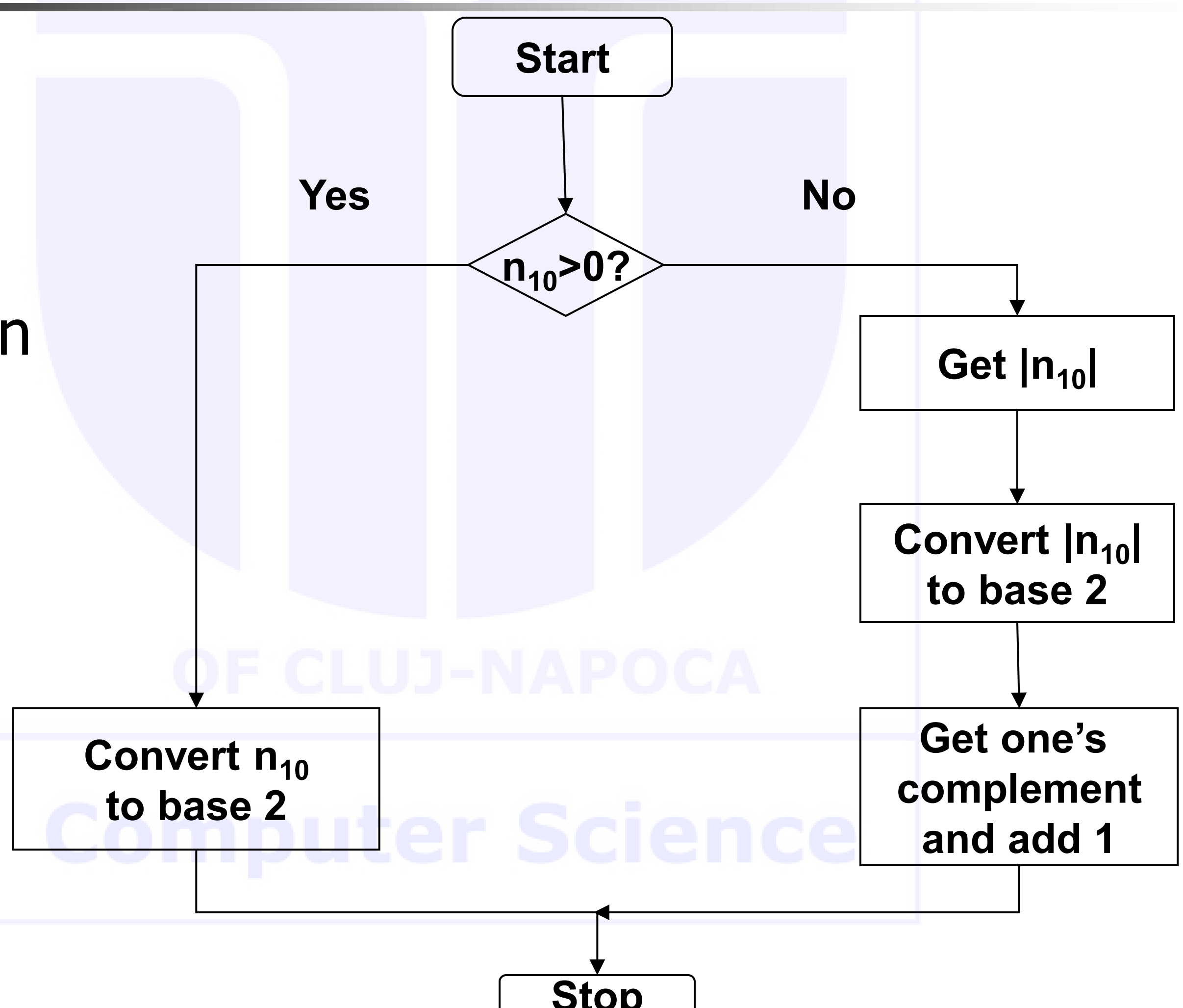
- *Finiteness*, i.e. The algorithm terminates after a number of steps. This property is also called *potential realizability*. In the example, *C* becomes 0, then step 8 is executed and computation stops.
- *Well-defined*, i.e. Each step is expressed non-ambiguously
- *Effectiveness*, i.e. The running time should be as short as possible, and the memory requirements as low as possible
- *Universality*, i.e. To allow for a class of problems to be solved using it.

Algorithm description. Flowchart

- **Flowcharts** - graphically depict the logical steps to carry out a task and show how the steps relate to each other. (More on this: https://en.wikipedia.org/wiki/Flowchart)
  - Use geometric symbols connected by arrows (flow lines).
  - Within each symbol is a phrase presenting the activity at that step.
  - The shape of a symbol indicates the type of operation that is to take place.
  - The flow is from top to the bottom of each page.
  - Advantage: provides for good presentation of tasks, easy to follow.
  - Disadvantage: time-consuming to write and update.

Algorithm description. Flowchart example

- Obtain two's complement representation for a decimal number, n_{10}



Algorithm description. Pseudocode

- **Pseudocode** – uses English-like statements which outline a particular task or process.
  - Short version of the actual computer code.
  - Advantages:
    - Can be translated into a programming language easily.
    - Compact
    - Looks like the final code.

Algorithm description.
Pseudocode example

```
ALGORITHMDESIGN(informal problem)
    formalize problem (mathematically) [Step 0]
    repeat
            devise algorithm [Step 1]
            analyze correctness [Step 2]
            analyze efficiency [Step 3]
            refine
        until algorithm good enough
    return algorithm
```

Algorithm description. Hierarchy chart

- **Hierarchy charts** similar to a company's organization chart.
  - Displays the overall program structure, describes what each part (module) does, and the relations between modules.
  - Read from top to bottom and from left to right.
  - Mainly used for initial planning of a program by creating independent parts.

What Is Programming?

- **Programming** =
  - Process of taking an algorithm and encoding it into a notation, a programming language, so that it can be executed by a computer.
  - The important first step = the need to have the solution
  - No algorithm => no program.

Programming Languages

- Programming language: notation for writing programs
  - Specific syntax
  - Use keywords with well-defined semantics
- Program: data description + processing statements
- Programming: program development = algorithm development + coding in an appropriate programming language

Programming Languages

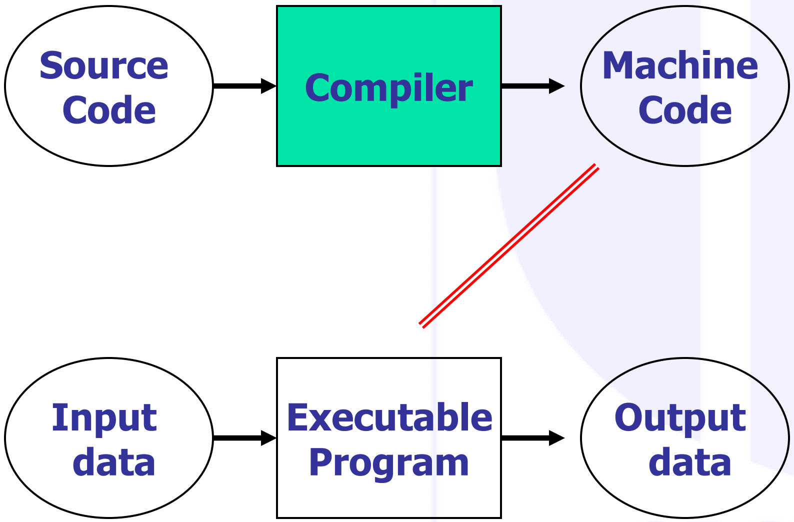
- Short history:
  http://www.princeton.edu/~ferguson/adw/programming_languages.shtml
  - 1955, John W. Backus (IBM) – FORTRAN (1957)
  - John McCarthy (MIT) - LISP
  - 1960, Peter Naur – Algol 60
  - 1959, a committee COBOL
  - 1964, BASIC
  - 1970 Alain Colmerauer and Philippe Roussel - Prolog
  - 1971, Niklaus Wirth (ETHZ) – Pascal
  - 1972, Dennis M. Ritchie and Brian W. Kernighan – C
  - 1980, Bjarne Stroustrup (AT&T) – C++
  - 1995, Sun Microsystems – Java
  - 2000+, Microsoft C#

  http://www.levenez.com/lang/history.html

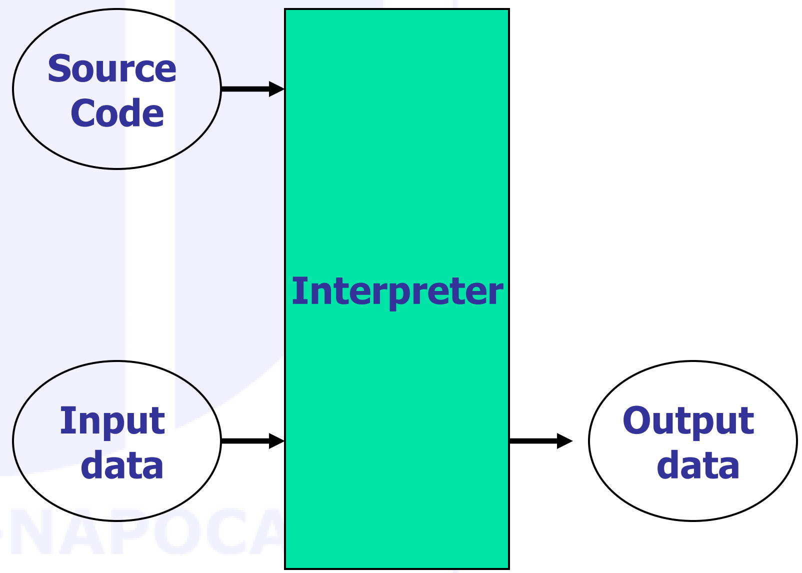
Higher level languages

- Programming Paradigms:
  - Imperative Programming: describes the exact *sequences of commands* to be executed
    - Structured programming, procedural programming
      - FORTRAN, C, PASCAL, …
    - Object oriented programming
      - C++, Java, C#, …
  - Declarative programming: program describes *what* it should do, *not how*
    - Functional programming
      - Lisp, ML, …
    - Logic Programming
      - Prolog

Compilers/Interpreters



Compiler: analyzes program and translates it into machine language
Executable program: can be run independently from compiler as many times => fast execution



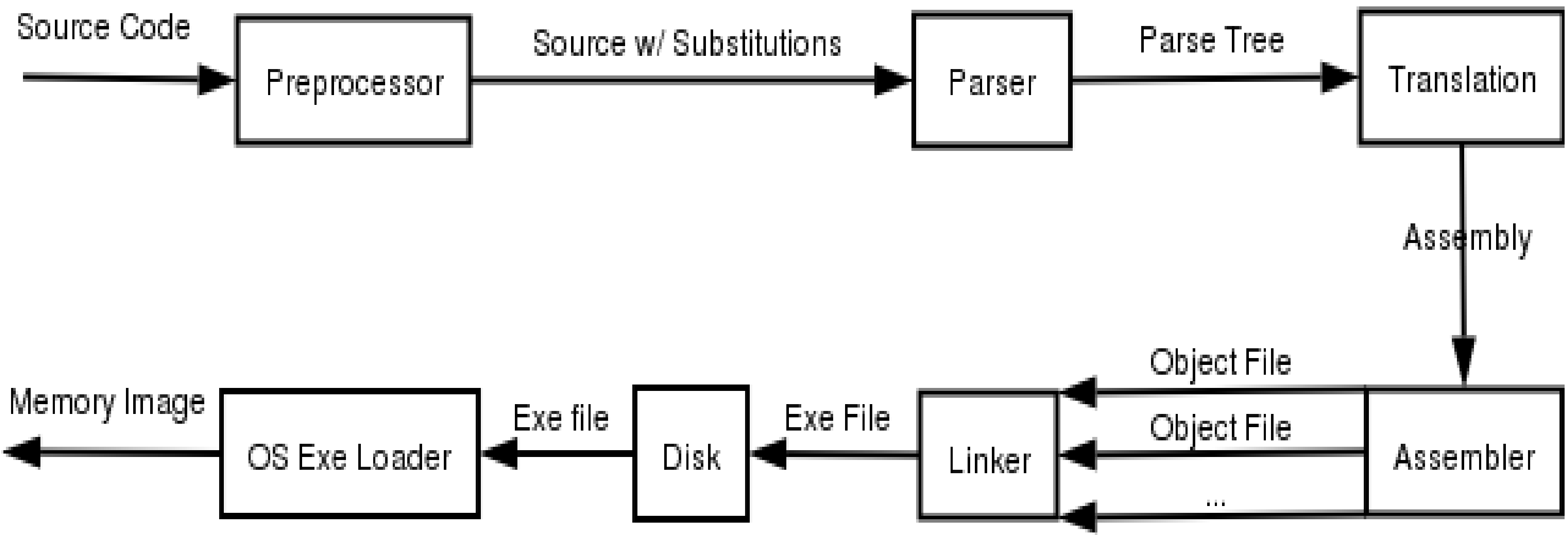
Interpreter: analyzes and executes program statements at the same time
Execution is slower
Easier to debug program

From Source to Executable

- **Compilation**: source code ==> relocatable object code (binaries)
- **Linking**: many relocatable binaries (modules plus libraries) ==> one relocatable binary (with all *external references satisfied*)
- **Loading**: relocatable ==> absolute binary (with all code and data references bound to the addresses occupied in memory)
- **Execution**: control is transferred to the first instruction of the program
- At compile time (CT), absolute addresses of variables and statement labels are not known.
- In *static* languages (such as Fortran), absolute addresses are bound at load time (LT).
- In block-structured languages, bindings can change at run time (RT).

From source code to executable

- Using a compiler



The C Programming Language

- Developed by Dennis Ritchie at AT&T Bell Laboratories in the early 1970s
- Growth of C tightly coupled with growth of Unix: Unix was written mostly in C
- Success of PCs: need of porting C on MS-DOS
- Many providers of C compilers for many different platforms => need for standardization of the C language
- 1990: ANSI C (American National Standards Institute)
- International Standard Organization: ISO/IEC 9899:1990
- 1999: standard updated: C99, or ISO/IEC 9899:1999
- 2011: C11 (formerly C1X) is an informal name for ISO/IEC 9899:2011

C Features

- C = low-level language
  - suitable language for systems programming
- C = small language
  - relies on a “library” of standard functions
- C = permissive language
  - it assumes that you know what you’re doing, so it allows you a wider degree of latitude than many languages. It doesn’t mandate the detailed error-checking found in other language

C Strengths

- Efficiency: intended for applications where assembly language had traditionally been used.
- Portability: hasn't splintered into incompatible dialects; small and easily written
- Power: large collection of data types and operators
- Flexibility: not only for system but also for embedded system commercial data processing
- Standard library
- Integration with UNIX

Weaknesses

- Error-prone
  - Freedom has a price
- Difficult to understand
  - Using good style helps
- Difficult to modify
  - Good (in-code) documentation helps

C fundamentals

- Keywords

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

Program structure

- A **program** is composed of one or more functions, of which one is mandatory: function **main**
  - The other defined functions are user-defined
- A C program basically has the following form:
  - Preprocessor Commands
  - Type definitions
  - Function prototypes -- declare function types and variables passed to a function.
  - Variables
  - Functions

The format in C

- Statements are terminated with semicolons
- Indentation is nice to be used for increased readability.
- Free format: white spaces and indentation is ignored by compiler
- **C is case sensitive** – pay attention to lower and upper case letters when typing !
  - All C keywords and standard functions are lower case
  - Typing INT, Int, etc instead of int is a compiler error
- Strings are placed in double quotes
- New line is represented by \n (Escape sequence)

The first C program

uses standard library input and output functions (printf)

the program

begin of program

statements

end of program

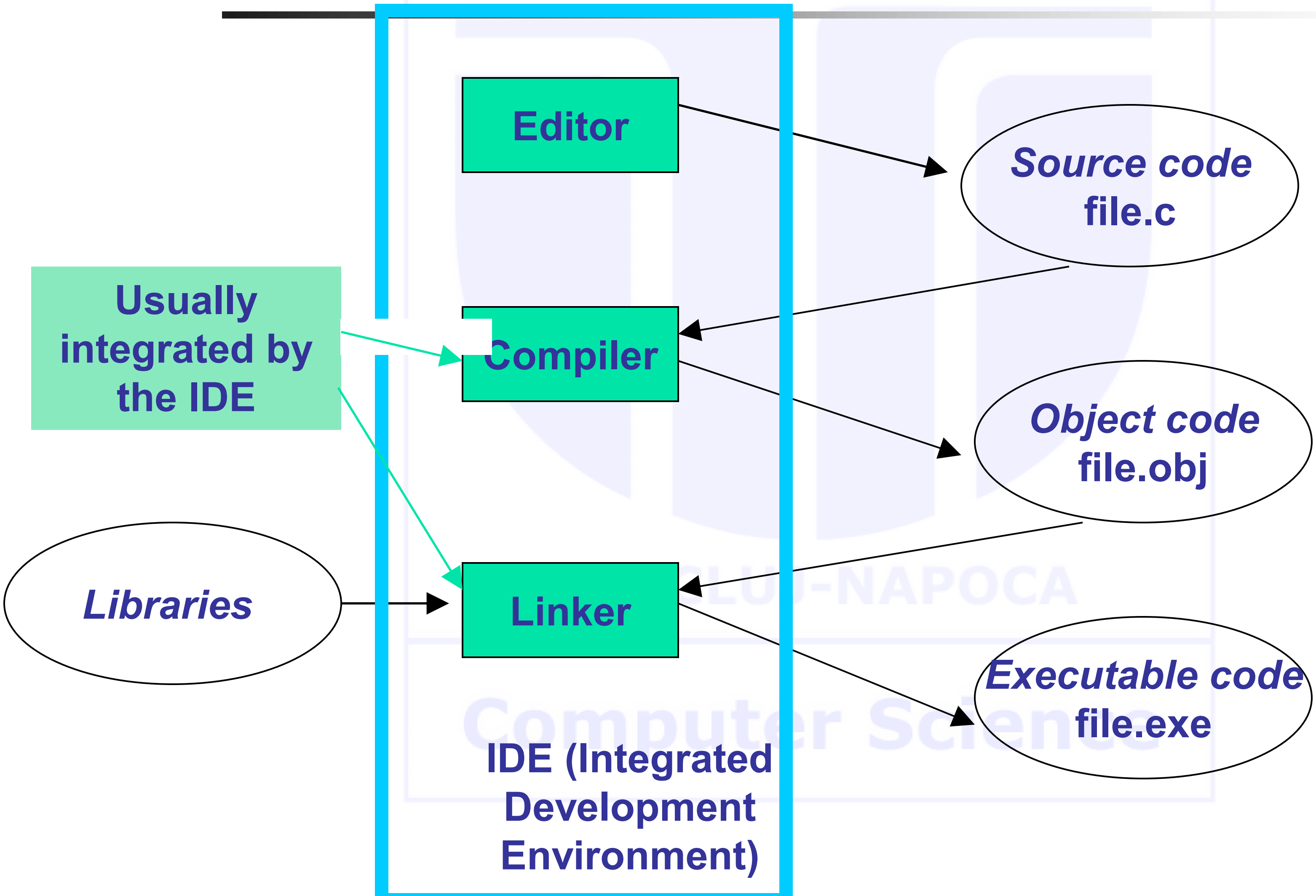
```
#include <stdio.h>

int main (void)
{
      printf ("Programming is fun.\n");
      return 0;
}
```

main: a special name that indicates where the program must begin execution. It is a special *function*.

first statement: calls a routine named printf, with argument the string of characters “Programming is fun \n”

last statement: finishes execution of main and returns to the system a status value of 0 (conventional value for OK)

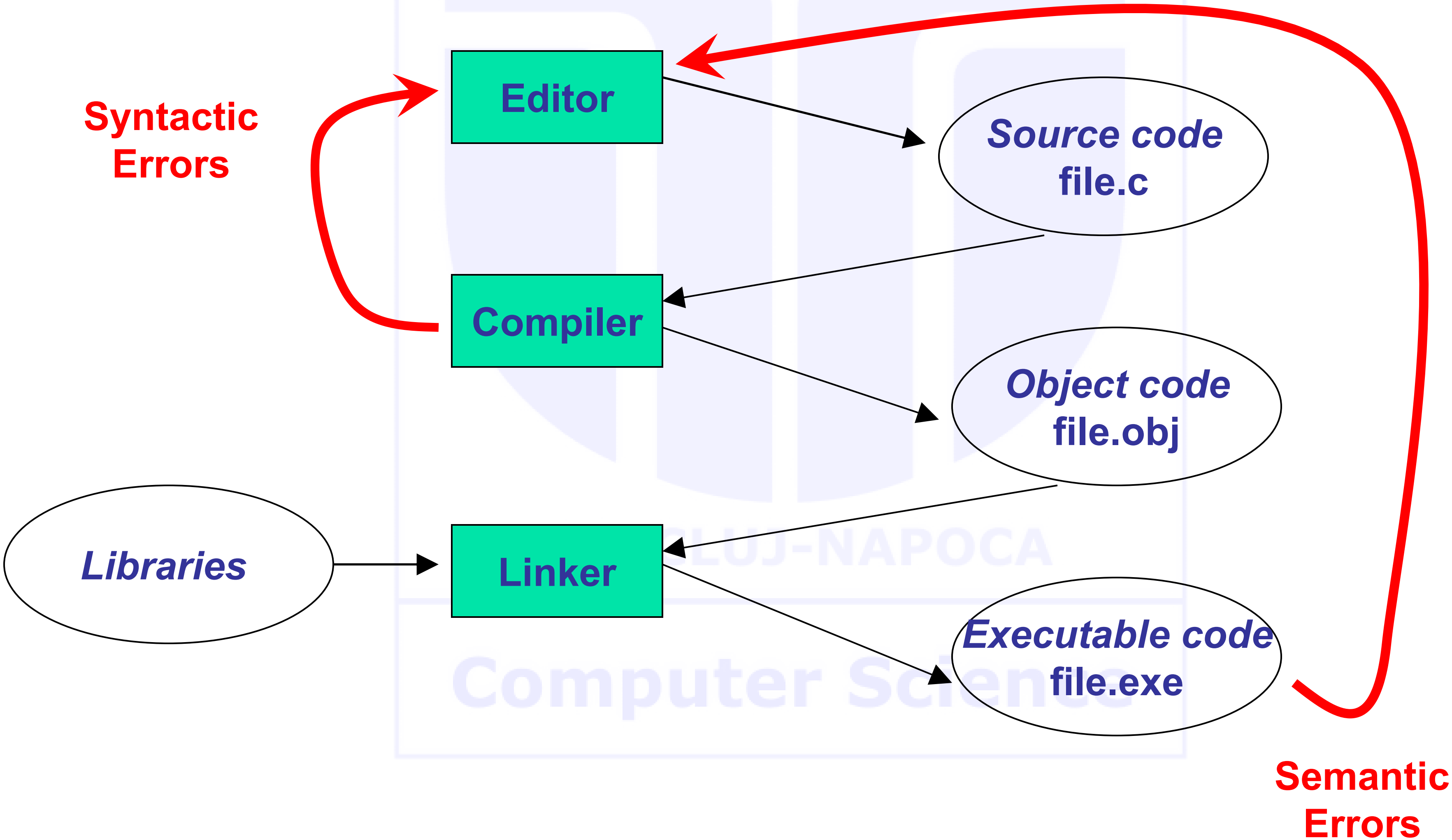
Compiling and running C programs



C Compilers and IDE's

- One can:
  - use a text editor to edit source code, and then use independent command-line compilers and linkers
  - use an IDE: everything together + facilities to debug, develop and organize large projects
- There are several C compilers and IDE's that support various C compilers
- Lab: CodeBlocks, Free Software (under the GNU General Public License)
  - Works with gcc (GNU C Compiler)
    - supports the C99 standard
    - available on Windows and Unix
  - The GNU Project (http://www.gnu.org/): launched in 1984 in order to develop a complete Unix-like operating system which is free software - the GNU system.

Debugging program errors



Syntax and Semantics

- Syntax errors: violation of programming language rules (grammar)
  - "*Me speak English good*."
  - Use valid C symbols in wrong places
  - Detected by the compiler
- Semantics errors: concern meaning:
  - "*This sentence is excellent French*."
  - Programs are syntactically correct but don't produce the expected output
  - User observes output of running program

Identifiers and symbols

- A source program is composed of tokens separated by *whitespace* (space `' '`, tab `'\t'`, newline `'\n'`)
- Tokens=identifiers and symbols
  - ***Identifiers***=names of constants, types, variables, functions
  - Name=sequence of letters and digits and underscore, first character is letter or underscore
    - Name length limited to 31 characters (ANSI C)
    - Example identifiers:
      - Correct: `A a alpha a1 a_1 AnIdentifier`
      - Bad: ~~`A! 2alpha a*`~~

Identifiers and symbols

- ***Symbols*** are groups of characters which are not identifiers. E.g.
  - Operators: `+ ++ && < <= != >` etc.
  - Numeric constants: `20.5 30 0x2d`;
  - Characters: `'A' 'z' '7'`
  - Character strings: `"C/C++ Programming Language"`

Comments

- Comments are used to facilitate understanding of programs, and ease program maintenance and teamwork
  - C comments are specified as

    `/* comment */`
  - Example (excerpt form NASA Style Guide, p. 79)

```
/******************************************************************
*
* FUNCTION NAME: GetReference
*
* ARGUMENT LIST:
*
* Argument Type IO Description
* ------------- -------- -- ---------------------------------
* ref_type int I Type of reference data requested
* = 1, S/C position vector
* = 2, S/C velocity vector
* ...
*
* RETURN VALUE: void
*
*******************************************************************/
void GetReference(int ref_type, double t_request, double t_wait,
double ref_vector[3])
```

Comments

- C++ also allows single line comments:

  ```
  // comment
  ```

- Example from NASA C++ style guide (http://aaaprod.gsfc.nasa.gov/WebSite/Files/Cplus/C++)

```
//
//  Main sequence: get and process all user requests
//

while (!finish())
{
    inquire();
    if (requestcode !=0)
    {
        //If the request code is non-zero, then perform
        //intermediate processing to generate request information
        generateRequestInfo(requestCode);
    }
    process();
}
```

C Data Types

- char, int, float, double
- long int (long), short int (short), long double
- signed char, signed int
- unsigned char, unsigned int
- 1234L is long integer
- 1234 is integer
- 12.34 is float
- 12.34L is long float

C Basic Integer Types

Type (32 bit)	Smallest Value	Largest Value
short int	-32,768(-2^{15})	32,767($2^{15}-1$)
unsigned short int	0	65,535($2^{16}-1$)
int	-2,147,483,648(-2^{31})	2,147,483,647($2^{31}-1$)
unsigned int	0	4,294,967,295
long int	-2,147,483,648(-2^{31})	2,147,483,647($2^{31}-1$)
unsigned long int	0	4,294,967,295

C Floating Types

float	**single-precision floating-point**
double	**double-precision floating-point**
long double	**extended-precision floating-point**

Type	Smallest Positive Value	Largest Value	Precision
float	$1.17*10^{-38}$	$3.40*10^{38}$	6 digits
double	$2.22*10^{-308}$	$1.79*10^{308}$	15 digits

```
double x;
scanf("%lf", &x);
printf("%lf", x);
```

```
long double x;
scanf("%Lf", &x);
printf("%Lf", x);
```

Constants

- A constant has a type and a value which cannot be changed during run time
- Integer constants:
  - Decimal: string of decimal digits optionally preceded by a sign
    - To indicate length and signedness:
      - L, l: long
      - U, u: unsigned
      - UL, ul, LU, lu: unsigned long
    - Examples: `100, 100L, 100U, 100ul`

Constants. Integers

- Octal constants
  - Begin with a zero and contain only octal digits
  - Are unsigned or unsigned long
  - Examples: 0144 (=100 decimal), 0176 (=126)
- Hexadecimal
  - Begin with 0x or 0X and contain only hex digits
  - Are unsigned or unsigned long
  - Examples: 0xab1 (=2737 decimal)

Constants. Character

- Character constants have their ASCII code as a value and **int** as a type
  - Printable characters: 'printable_char', e.g. `'Z'`, `'s'`
  - Escape sequences

Sequence	Meaning
`\a`	Alert (ANSI C).
`\b`	Backspace.
`\f`	Form feed.
`\n`	Newline.
`\r`	Carriage return.
`\t`	Horizontal tab.
`\v`	Vertical tab.
`\\`	Backslash (\).
`\'`	Single quote (').
`\"`	Double quote (").
`\?`	Question mark (?).
`\0oo`	Octal value. (o represents an octal digit.)
`\xhh`	Hexadecimal value. (h represents a hexadecimal digit.)

Constants. Character strings

- Character string: a sequence of characters included between double quotes
  - Escape sequence may appear in the sequence
  - Examples:

```
”character string”
”an apostrophe, ’, is represented as usual”
”GNU \”C\” Compiler”
```

  - If continued on next line, at the end of the continued line \ followed by the Enter key must be typed, e.g. `“this is a continued\`
`line”`
  - Memory area pattern:

0	1	2	…	n-1	n
ASCII code	ASCII code	ASCII code	…	ASCII code	‘\0’

Variable declaration

- For simple variables:

  ```
  type identifier {, identifier };
  { type identifier {, identifier };}
  ```

  - Examples:

    ```
    int i, j, k;
    char c;
    double x, y;
    ```

- For array variables:

  ```
  base_type identifier[lim] {[lim] } {,
    identifier[lim] {[lim] }};
  ```

  - **Indices run from 0 to lim-1.**
  - **Limits are constant expressions, evaluated at compile time**
  - Examples:

    ```
    int alpha[100];
    double matrix[10][15];
    ```

Standard I/O functions

- If interactive processing, then:
  - The standard terminal is the terminal used to run the program. Three files are attached to it:
    - Input file (**stdin**)
    - Output file (**stdout**)
    - Error file (**stderr**)
- C/C++ do not have *statements* for reading and writing. I/O operations are achieved by functions with prototypes in **stdio.h**
  - Note: **conio.h** is an extension, it is NOT part of the standard

Function `gets` . Do not use it!

- **gets**:
  - Reads characters from the stream **stdin** up to the next newline character, and stores them in the string given as argument
  - The newline character is discarded.
  - If gets encounters a read error or end-of-file, it returns a null pointer; otherwise it returns its argument.
  - **Warning**: The gets function is very dangerous because it provides no protection against overflowing the string *s*. The GNU library includes it for compatibility only. *You should always use **fgets** or **getline** instead*.
  - Prototype:

```
char *gets(char *s);
```

Functions `puts`

- **puts**:
  - Writes the string given as an argument to the stream `stdout` followed by a newline.
  - The terminating null character of the string is not written.
  - puts is the most convenient function for printing simple messages.
  - Returns the code of the last output character
  - Prototype:

```
int puts(const char *s);
```

Functions **gets** and **puts**. Example

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   char s[200];
   printf("\nInput a character string and press
   Enter\n");
   gets(s); /* the character string is stored at the
   address where s points */
   printf("\nThe string you input is:\n");
   puts(s);
   system("PAUSE");
   return 0;
}
```

Function `scanf`

- **scanf**
  - reads *formatted input* from the stream stdin under the control of the template string given as its first argument.
  - The next arguments are optional arguments, and are *pointers* to the places which receive the resulting values.
  - The return value is normally the number of successful assignments.
  - If an end-of-file condition is detected before any matches are performed, including matches against whitespace and literal characters in the template, then EOF is returned.
  - Prototype:

```
int scanf(const char *template, {pointer});
```

Function **`scanf`**. Template string I

- A *scanf template string* contains *format specifiers*
- A *scanf template string* general form:

  `% flags width type conversion`

- Input conversion specification: an initial '`%`' character followed in sequence by:
  - An optional **flag character, ***, which says to ignore the text read for this specification.

Function `scanf`. Template string

- Input conversion specification (cont'd) :
  - An optional **decimal integer** that specifies the **maximum field width**.
    - Reading of characters from the input stream stops either when this maximum is reached or when a non-matching character is found, whichever happens first.
    - Most conversions discard initial whitespace characters (those that don't are explicitly documented), and these discarded characters don't count towards the maximum field width.
    - String input conversions store a null character to mark the end of the input; the maximum field width does not include this terminator.
  - An optional *type modifier character*.
  - A character that specifies the conversion to be applied.

Function `scanf`. Table of input conversions I

- Table of input conversions (1)

`%d`	**An optionally signed integer written in decimal.**
`%i`	**An optionally signed integer in any of the formats that the C language defines for specifying an integer constant.**
`%o`	**An unsigned integer written in octal radix.**
`%u`	**An unsigned integer written in decimal radix**
`%x, %X`	**An unsigned integer written in hexadecimal radix.**
`%e, %f, %g, %E, %G`	**An optionally signed floating-point number.**

Function `scanf`. Table of input conversions II

- Table of input conversions (2)

`%s`	**A string containing only non-whitespace characters**
`%[`	**A string of characters that belong to a specified set.**
`%c`	**A string of one or more characters; the number of characters read is controlled by the maximum field width given for the conversion.**
`%p`	**A pointer value in the same implementation-defined format used by the %p output conversion for printf.**
`%n`	**Doesn't read any characters; it records the number of characters read so far by this call.**
`%%`	**A literal % given in the input stream**

Function `scanf.` Scan sets

- Scan sets
  - Set of characters enclosed in square brackets `[]`
    - Preceded by `%` sign
  - Scans input stream, looking only for characters in scan set
    - Whenever a match occurs, stores character in specified array
    - Stops scanning once a character not in the scan set is found
  - Inverted scan sets
    - Use a caret **^**: `"%[^aeiou]"`
    - Causes characters not in the scan set to be stored
- Skipping characters
  - Include character to skip in format control
  - Or, use `*` (assignment suppression character)
    - Skips any type of character without storing it

Function `scanf`. Examples

- Usage examples for **scanf**
  - Read a character
    ```
    char ch;
    scanf("%c", &ch);
    ```
  - Read a character string
    ```
    char s[40];
    scanf("%s", s);
    ```
  - Read three integers with values in decimal, octal, and hexadecimal respectively
    ```
    int a, b, c;
    scanf("%d %o %x", &a, &b, &c);
    ```
  - Read reals of type float, double, long double
    ```
    float x;
    double y;
    long double z;
    scanf("%f %lf %Lf", &x, &y, &z);
    ```

Function `printf`

- **printf**
  - prints the optional arguments under the control of the template string template to the stream stdout.
  - returns the number of characters printed, or a negative value if there was an output error.
  - Prototype:

```
int printf (const char *template, ...){expression_list});
```

Function `printf`. Template string I

- A *printf template string* contains *format specifiers*
- A *printf template string* has the general form:
  `% { param-no $} flags width { . precision } type conversion`
- An *output conversion* specification: an initial '%' character followed in sequence by:
  - An optional specification of the parameter used for this format. Normally the parameters to the printf function are assigned to the formats in the order of appearance in the format string.
    - The **param-no** part of the format must be an integer in the range of 1 to the maximum number of arguments present to the function call.

Function `printf`. Template string II

- *printf template string format specifiers (cont'd)*:
  - Zero or more **flag characters** that modify the normal behavior of the conversion specification.
  - An optional **decimal integer** that specifies the **minimum field width**.
    - This is a minimum value; if the normal conversion produces more characters than this, the field is not truncated. Normally, the output is right-justified within the field.
    - A field width of '*' => the next argument in the argument list (before the actual value to be printed) , a positive int, is used as the field width.

Function `printf`. Template string III

- *printf template string format specifiers (cont'd)*
  - An optional **precision** : number of digits to be written for the numeric conversions.
    - Precision: specified as a period (.) followed optionally by a decimal integer (which defaults to zero if omitted).
    - Precision specified as *. => the next argument in the argument list, a positive int (before the actual value to be printed) is used as the precision.

Function `printf`. Template string IV

- *printf template string format specifiers (cont'd)*
  - An optional **type modifier character**, which is used to specify the data type of the corresponding argument if it differs from the default type. (For example, the integer conversions assume a type of int, but you can specify h, l, or L for other integer types.)
  - A character that specifies the conversion to be applied.
    - Conversion characters are the same as for scanf, except for %[which is not used here

Function `printf`. Table of output conversions I

- Table of output conversions (1)

`%d %i`	**Print an integer as a signed decimal number.**
`%u`	**Print an integer as an unsigned decimal number.**
`%o`	**Print an integer as an unsigned octal number.**
`%x %X`	**Print an integer as an unsigned hexadecimal number. %x uses lower-case letters and %X uses upper-case.**
`%f`	**Print a floating-point number in normal (fixed-point) notation**
`%e, %E,`	**Print a floating-point number in exponential notation. %e uses lower-case letters and %E uses upper-case.**
`%g, %G`	**Print a floating-point number in either normal or exponential notation, whichever is more appropriate for its magnitude. %g uses lower-case letters and %G uses upper-case.**

Function `printf`. Table of output conversions II

- Table of output conversions (2)

`%a %A`	**Print a floating-point number in a hexadecimal fractional notation which the exponent to base 2 represented in decimal digits. %a uses lower-case letters and %A uses upper-case.**
`%c`	**Print a single character.**
`%s`	**Print a string.**
`%p`	**Print the value of a pointer.**
`%n`	**Get the number of characters printed so far.**

Function `printf`. Flags for integer conversions

- Modifier **flags** for *integer* conversions:
  - –: Left-justify the result in the field (instead of the normal right-justification).
  - +: For the **signed %d** and **%i** conversions, print a plus sign if the value is positive.
  - **#**:
    - For **%o** conversion, forces the leading digit to be 0, as if by increasing the precision.
    - For **%x** or **%X**, prefixes a leading **0x** or **0X** to the result.
    - Doesn't do anything useful for the %d, %i, or %u conversions.
  - 0: Pad the field with zeros instead of spaces. The zeros are placed after any indication of sign or base.
    - This flag is ignored if the – flag is also specified, or if a precision is specified.

Function `printf`. Example: integer

- Example of integer numbers printing (from libc help)
  - Values printed are, in order: 0, 1, -1, 100000

Specifier:

```
"|%5d|%-5d|%+5d|%+-5d|% 5d|%05d|%5.0d|%5.2d|%d|\n"
```

Output:

```
|    0|0    |   +0|+0   |    0|00000|     |   00|0|
|    1|1    |   +1|+1   |    1|00001|    1|   01|1|
|   -1|-1   |   -1|-1   |   -1|-0001|   -1|  -01|-1|
|100000|100000|+100000|+100000| 100000|100000|100000|100000|100000|
```

Specifier:

```
"|%5u|%5o|%5x|%5X|%#5o|%#5x|%#5X|%#10.8x|\n"
```

Output:

```
|    0|    0|    0|    0|    0|    0|    0|  00000000|
|    1|    1|    1|    1|   01|  0x1|  0X1|0x00000001|
|100000|303240|186a0|186A0|0303240|0x186a0|0X186A0|0x000186a0|
```

Function `printf`. Flags for integer conversions

- Modifier **flags** for *floating point number* conversions
  - –: Left-justify the result in the field. Normally the result is right-justified.
  - +: Always include a plus or minus sign in the result.
    - If the result doesn't start with a plus or minus sign, prefix it with a space instead. Since the + flag ensures that the result includes a sign, this flag is ignored if you supply both of them.
  - #: Specifies that the result should always include a decimal point, even if no digits follow it. For the %g and %G conversions, this also forces trailing zeros after the decimal point to be left in place where they would otherwise be removed.
  - 0: Pad the field with zeros instead of spaces; the zeros are placed after any sign. This flag is ignored if the '–' flag is also specified.
  - A **type modifier** is supported: **L**
    - An uppercase L specifies that the argument is a long double.

Function `printf`. Example: floating point

- Example of floating number printing (from libc help)
  - Values printed are, in order:0, 0.5, 1, -1, 100, 1000, 10000, 12345, 100000, 123456

Specifier: `"|%13.4a|%13.4f|%13.4e|%13.4g|\n"`

Output:

```
|  0x0.0000p+0|       0.0000|   0.0000e+00|            0|
|  0x1.0000p-1|       0.5000|   5.0000e-01|          0.5|
|  0x1.0000p+0|       1.0000|   1.0000e+00|            1|
| -0x1.0000p+0|      -1.0000|  -1.0000e+00|           -1|
|  0x1.9000p+6|     100.0000|   1.0000e+02|          100|
|  0x1.f400p+9|    1000.0000|   1.0000e+03|         1000|
| 0x1.3880p+13|   10000.0000|   1.0000e+04|        1e+04|
| 0x1.81c8p+13|   12345.0000|   1.2345e+04|    1.234e+04|
| 0x1.86a0p+16|  100000.0000|   1.0000e+05|        1e+05|
| 0x1.e240p+16|  123456.0000|   1.2346e+05|    1.235e+05|
```

Reading

- Deitel, chapter 2, 3.1, 3.2, 3.3, chapter 9
- King, chapters 1, 2, 3
- Prata, chapters 1, 2, 3
- (see slide 14 for complete book names)

Summary

- Who and What
- Problem solving process
  - Stages
- Algorithm
  - Definition, features, ways to describe
- Programming Languages
- C Introduction
  - Basic program structure
  - Data Types
  - Constants and Variables
  - Simple I/O
    - scanf
    - printf