# Low-level programming

## Lecture 1

Introduction

Marcin Radom, Ph.D.

Institute of Computing Science
Faculty of Computing and Telecommunication

# Organization

Lecture – rules of evaluation

Literature

Lectures plan

# Evaluation rules

- **Laboratories**
  - Exercises take place in laboratory,
  - Evaluation is based on projects and tests (or whatever rules the person responsible for the laboratories establish).
- Lectures are available on page: http://www.cs.put.poznan.pl/mradom
- **Lectures and laboratories are not connected in terms of the end-evaluation.**
- **Evaluation of the lectures – in form of a test probably in the last week.**

# Lectures plan

- **Lecture I**
  - Introduction to C: history, identifiers, C alphabet
  - Types, constants
- **Lecture II**
  - Instruction **if-else**
  - Instruction **switch**
  - Instruction **for**, **while**, **do while**
  - Instructions **break**, **continue**
  - Preprocessor: **#define** and macro definitions, conditional compilation
- **Lecture III**
  - Basic operators, bit operators, conditional operators
  - Priorities of operators
  - Data types transformation
  - Input and output operations
- **Lecture IV**
  - One and many-dimensional tables
  - String transformation (**string.h**)

# Lectures plan

- **Lecture V**
  - Dynamic memory allocation, pointer arrays
  - Structures and unions
  - Dynamical data structures – lists
  - Structure tables, pointers to structures
  - Connecting multiple files in a single project, compilation issues
  - Static variables
- **Lecture VI**
  - Functions – definitions, arguments
  - Structures and tables as function arguments
  - Recurrence
- **Lecture VII**
  - Pointers to functions
  - Tables of pointers to functions
  - I/O operations, files

# History of C

History, features, popularity

# C language history

- Developed in 1969-1973 (UNIX kernel written in 1973).
- Predecessor was the called B language and before that – BCPL (1966, Martin Richards).
- 1978: *The C Programming Language*, Brian Kernighan & Dennis Ritchie.
- Object extension: C++ language (non-object part of the language has been extended as well).
- In 1983 American National Standardization Institute (ANSI) created a committee with a task of formulation of C language specification.
- In 1989 – ANSI standard (i.e. ANSI C) – modern compilers realize most of the feature of such standard.

# C language history

- In 1990 C language standard has been written as ISO 9899. This document modified the ANSI standard. Such languages are informally called C89.

- Since that time many modifications have been made. The most important one is 9899:1999, and such language is called C99:
  - Changed comment system, comment: //
  - New standard functions and header files
  - Extended preprocessor
  - New keywords, e.g., const, enum, signed, void

- ISO 9899:1999 is not fully support by compilers. But, e.g., GNU C has most of the aforementioned changes.

# C language history

- Newest standard: ISO/IEC 9899:2011, known informally as C11. It introduces:
  - Multithreading support
  - New header files, e.g., <threads.h>, <srdatomic.h>, <uchar.h>
  - Anonymous structures and unions.
  - New keywords, e.g., _Generic, _Thread_local, _Alignas
  - Unicode signs support, i.e. new types of data independent from platform: char16_t, char32_t
  - Function gets() removed and replaced by gets_s()
  - More secure version of fopen: fopen_s
- GCC compiler supports C11 in a limited range. In order to compile in this standard, an option **–std=c11** or **–std=iso9899:2011** must be chosen.

# C language history

- **C (ANSI C)** is a structural language, predecessor of currently used object - oriented languages.
- It is a procedural programming language, with the following steering orders:
  - group of instructions, decisions: if-else
  - choosing from a set of cases: switch
  - repeating with checking the condition at the beginning (while, for) or at the end (do) of a loop
  - loop interrupting: break
- It is called a low-level programming language, because it deals with signs, number and addresses, instead of objects.
- It is a general-usage language (i.e., it has not specific task).
- It is not connected strictly with a precise operating system or CPU

# C language

- **Why it is worth knowing:**
  - The most commonly used programming language.
  - It is not that hard to learn.
  - Low-level programming allows writing fast programs.
  - Most of the currently used programming language has C syntax.

- **Cons of C:**
  - Some of its features go against intuition or (human) logic.
  - It is not so good in supporting the programmer in a task of errors finding and correcting.

# Programming languages popularity (2018)

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

| Feb 2018 | Feb 2017 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 1 | | Java | 14.988% | -1.69% |
| 2 | 2 | | C | 11.857% | +3.41% |
| 3 | 3 | | C++ | 5.726% | +0.30% |
| 4 | 5 | | Python | 5.168% | +1.12% |
| 5 | 4 | | C# | 4.453% | -0.45% |
| 6 | 8 | | Visual Basic .NET | 4.072% | +1.25% |
| 7 | 6 | | PHP | 3.420% | +0.35% |
| 8 | 7 | | JavaScript | 3.165% | +0.29% |
| 9 | 9 | | Delphi/Object Pascal | 2.589% | +0.11% |
| 10 | 11 | | Ruby | 2.534% | +0.38% |
| 11 | - | | SQL | 2.356% | +2.36% |
| 12 | 16 | | Visual Basic | 2.177% | +0.30% |
| 13 | 15 | | R | 2.086% | +0.16% |
| 14 | 18 | | PL/SQL | 1.877% | +0.33% |
| 15 | 13 | | Assembly language | 1.833% | -0.27% |
| 16 | 12 | | Swift | 1.794% | -0.33% |
| 17 | 10 | | Perl | 1.759% | -0.41% |
| 18 | 14 | | Go | 1.417% | -0.69% |
| 19 | 17 | | MATLAB | 1.228% | -0.49% |
| 20 | 19 | | Objective-C | 1.130% | -0.41% |

# C Programming

Identifiers, types, variables

# A simple example

```c
#include <stdio.h>
int main( ) {
   printf ( "Hello World!\n" );
   return 0;
}
```

- **#include <stdio.h>** - preprocessor instruction: add standard input / output library: *stdio.h* – **ST**an**D**ard **I**nput **O**utput.
- Function **main()** – the program always starts at the beginning of main(). In the example the function has no arguments (i.e., empty brackets, however they are necessary according to the C syntax) – the function not expects any arguments when called.
- Inside **main()** we call function **printf**("Hello World!\n"), which puts the text string on the screen. **\n** represents new line.
- Sign\ introduces a special character, impossible to write directly. For example: **\t** means tabulation, **\b** reverse the cursor by one position, **** introduces sign ****, **\"** introduces **"** (in cannot be directly written in the **printf**() braces).

# Simple example

- Function **printf()** will not automatically move cursor to the next line. So: without \n, these three function will print the text in the same line:

```
printf("Hello ");
printf("World!");
printf("\n");
```

- Curly brackets surrounds instructions within a function.
- **Reminder:**
  - Variables and constant variables are basic types of object used in a program.
  - Declarations introduces necessary variables and specify their type (and the starting values).
  - Operators decide what to do with them.
  - Type of variables determine the set of their values and operations which can be performed on them.

# C language alphabet

- C alphabet – set of signs which are used to write the programs in C language.
- **The language consists of:**
  - All signs of 8-bit ASCII code:
    - Large letters: A – Z
    - Small letters: a – z
    - Digits: 0 – 9
    - Special signs: ! * + \ " < # ( = | { > % ) ~ ; } / ^ - [ : , ? & _ ] ' and the space
- C language (precisely: C99) supports writing sign with **Unicode** norms (universal standard of signs coding)
  - Code will not be compatible with the older versions of ANSI C, which can reduce code reusability.
  - Most commonly used coding standard is UTF-8.

# Identifiers, orders, types

- The name of the variable can contain letters and digits, however it must start with either a letter or: _
- In the C language large and small letters are distinguished. So **variable** and **Variable** are two different variable names. Other examples:

```
alfa      Alfa      AlfA      ALFA
Milk_price    TransportCOst
```

- It is common to write variables with small letters and names of symbolic constants with large ones.
- **Keywords** of C are reserved and cannot be used as names of variables:
  - **auto** – old, unused local variable keyword
  - **double** – floating point value, double precision
  - **int** – integer type with sign
  - **struct** – structure declaration
  - **break** – exit from a loop or **switch**
  - **else** – optional part of **if**
  - **long** – modifier or type of integer
  - **switch** – instruction for choosing from a set of options

# Identifiers, orders, types

- **case** – alternative for switch order
- **enum** – enumerative type
- **register** – register variables, old, unused (mostly)
- **typedef** – for type creation
- **char** – single byte character
- **extern** – for global variable declared in different file
- **return** – instruction for returning from a function
- **union** – union declaration
- **const** – constant order
- **float** – floating point variable, single precision
- **short** – type modifier or data type
- **unsigned** – modifier, variable without sign

# Identifiers, orders, types

- **continue** – instruction for returning to the beginning of a loop
- **for** – loop instruction
- **signed** – modifier: variable with sign
- **void** – null data type
- **default** – default alternative in switch – case
- **goto** – jump instruction
- **sizeof** – size operator
- **volatile** – variable always read from memory
- **do** – part of do – while loop
- **if** – if – else instruction
- **static** – value of a variable is saved between consecutive returns to a function; static variable, local symbol
- **while** – loop instruction

# Comments

- Signs after **//** are considered one-line comments (like in C++)
  - **E.g.:**

```
instruction;  // comment, may be anything
// .............................
```

- **/*** and ***/** opens and closes multi-line comment.

```
/*
      ......................
      something wise
      ........................
*/
```

- Comments cannot be embedded (ANSI C standard ).
- They cannot be part of a string or constant statement.

# Types of integer values

- There are multiple subtypes of integer number type:

| type | signed | unsigned | bytes |
|---|---|---|---|
| `char` | – 128 , + 127 | 0 , 255 | 1 |
| `short` | – 32 768 , + 32767 | 0 , 65535 | 2 |
| `int,` `long` | – 2 147 483 648 , + 2 147 483 647 | 0 , 4 294 967 295 | 4 |
| `long` `long` | – 9 223 372 036 854 775 808 , + 9 223 372 036 854 775 807 | 0 , 18 446 744 073 709 551 615 | 8 |

- Additionally there are some qualifiers which can be used with integer types:
  - **short** and **long** keywords can also be used for modifying the range of variable values.

# Types of integer values

- The **int** type <u>in general represents an integer size dependent on a current computer architecture.</u>
- Type **short** often has **2 bytes**,  **long** type – **4/8 bytes.**
- **The compiler can freely choose the real sizes of such variable types, but with some restrictions::**
  - Types **short** and **int** must be at least 2 bytes in size.
  - Type **long** must have at least 4 bytes.
  - Type **short** cannot be larger than **int**, and **int** cannot be longer than **long**.
- Qualifiers like **signed** and **unsigned** can be used with a **char** type or any other integer type.
- Variables preceded with **unsigned** are always positive or equal to 0, they follow the modulo $2^n$ arithmetic ( n – number of bits in a given type), e.g. variable with a type **unsigned char** will have a range from 0 to 255.
- `char` `short` `int` `long` `long long` `signed` `unsigned`

# Floating-point variables

| type | range | bytes |
|---|---|---|
| `float` | $\pm 3.4 * 10^{\pm 38}$ | 4 |
| `double`, `long double` | $\pm 1.7 * 10^{\pm 308}$ | 8 |

- To store both integer part and fraction part.
- Type **long double** introduces a floating-point variable with a single precision.
- Sizes of variables of such type depend on implementation.
- Types **float**, **double**, **long double** can represent one, two or even three **different** sizes of variables.

# Floating-point variables

- Standard for such variables: **IEEE 754.** It defines two base classes of binary floating-point numbers:
  - 32-bits (single precision)
  - 64-bits (double precision)

| Format | Sign byte | Exponent bit | Significand bit |
|---|---|---|---|
| 32 bity | 1 | 8 | 23 |
| 64 bity | 1 | 11 | 52 |

- Where the pattern which code a number is as follows:

$$D_{FP} = \text{significand} * 2^{\text{exponent}}$$

# Types float / double

- Floating-point numbers are written with a decimal dot (e.g. 120.4) or with the exponent, e.g., 1e-2 or both.
- Example values:

```
1.25          0.343      .5       2.
35.56E-12      0.34e2     5e3     17.18E+28
```

- **Type of the number is assumed as follows:**
  - On the basis of the value (by default: **double**)
  - Given by the number itself
    - Letter ***f*** or ***F*** at the end of the number means **float**, letter ***l*** or ***L*** means **long double***.* E.g.:

```
12.545f                    // float
0.2345676543F              // float
0.5e-3lf                   // long double
0.9999998899E456LF         // long double
```

# Integer values (type int)

- **Such type is assumed as follows:**
  - On the basis of a value (default: **int**)
    - If variable without L on the end is not small enough to fit int **int** type, it is assumed to be **long**. For example:

```
12    25467      // signed int
34760548093// signed long long
```

  - Pointed in the number:
    - In variable **long** on the end of number there is *l* (small L) or *L*.
    - In variable with keyword **unsigned** on the end of a number value there is either *u* or *U.* Ending like *ul* or *UL* means **unsigned long**.
    - For example:

```
15L    077777l   0xFF4FFFL// signed long
254ll     -457LL   0xAB56LL     // signed long long
45211u    0xffau      // unsigned int
3000000000ul   0xC56AFB44UL     // unsigned long
-120ULL  78ull        // unsigned long long
```

# Character variables

- Integer value can be given also in octal or hexadecimal form:
  - **0** (zero) before a number signifies octal numeral system, e.g., 077 is 63 decimal
  - Hexadecimal number is proceeded by **0x** or **0X**, e.g., 0xFF = 0XFF = 255 in decimal system
  - Examples:

```
12    154555              // decimal
012   03777453            // octal
0xAB  0x5c5d   0xfff45a   // hexadecimal
```

- **Character variables (also integer type): `char`**
  - Single variable of such a type is an integer which can store values between 0 and 255. Also, a single random character fits in such a type, because it is always stored in one byte.
  - Such a character must be given in apostrophes.
  - Char type also stores special signs which are preceded by **** e.g., **\n**, **\t**

# Character variables

- <u>Special signs</u> (e.g., \n, \t and so on) can also be given using octal or hexadecimal system:

```
#define VTAB '\013'//ASCII: vertical tabulator
#define BELL '\007'//ASCII: alarm 'sign’
#define VTAB '\xb' //ASCII: vertical tabulator
#define BELL '\x7' //ASCII: alarm 'sign'
```

- **List of special character in C:**

```
\a          alarm                   \\          sign\
\b          back sign               \?          question sign
\f          new sign                \'          apostrophe
\n          new line                \"          quotation mark
\r          carriage return (CR)*   \ooo        octal number
\t          horizontal tabulation   \xff        hex number
\v          vertical tabulation
```

  - ***CR – *carriage return* – part of the „enter” \n\r - first sign means new line, the second reverse the cursor at left side of the page.**

# Strings, character constants

- <u>Example special signs</u>**:**

```
'a'  '5' '+' '.'
'A'  '\071'  '\x41'  '\x5F'
'\n'     '\t'     '\r'     '\\'     '\"'
```

- **'\o'** represents an empty sign and is commonly used to mark the end of the string / sequence of characters in a quotation marks

- **Constants**:

```
#define LEAP 1
int tab[31+30+LEAP+28]
```

- **Constant string** is a series of characters within a quotation mark, e.g., "I am a string" or "" (empty string).
  - Quotation marks are not part of the string, if we want them to be, we need to use \" special characters

# Strings

- Examples:

```
"Programming in C"
"Result: "
"\tName\tSurname\tAddress\n"
"\x16\x16\x02"          // SYN SYN STX
"He looked and said: \"I do not know\"."
```

- **String constant is a table, elements of which are characters**.
- Internal representation includes the **\o** special character, therefore the byte size of a string is always one byte longer in memory. E.g. :

**"ABC"** :

| | | | |
|---|---|---|---|
| **0x41** | **0x42** | **0x43** | **0x00** |

- On the other hand: there is no upper limit for a string size (other than the size of available computer memory).

# Strings

- Programs in C must read the whole string in order to determine its length.
- Useful function: **strlen(string)** – returns the size of the string (in integer value), but **without counting the \o special sign.**
- Function **strlen()** and many others are declared in **<string.h>** ; to use it:

  #include <string.h>.

- Difference between ' ' and " ":
  - 'c' is a single character, one from the ASCII table.
  - "c" is an array of characters consisting of c letter and hidden special sign \o denoting end of string (table) in memory.

# Enumeration variables

- **Enumeration** is a set of (integer!) constants assigned to unique strings:

```
enum boolean { NO, YES }
```

i.e., after **enum** a name is provided, and in braces: unique strings (without quotation marks!!!)

  - If not defined by the programmed, the assigned numbers starts from 0 and are being incremented: *NO* – 0, *YES* – 1.
  - Another example:

```
enum { JAN = 1, FEB, MAR, APR, MAI, JUN, JUL, AUG, SEP, OCT, NOV,
DEC }
```

*JAN* has numerical value of 1, then *FEB* - 2, *MAR* - 3, …., *DEC* – 12. If for example *JAN* had assigned value of 2, then *DEC* would have 13.

- The string in the enumeration **cannot** be repeated. Their assigned values on the other hand – **can**.

# Enumeration variables

- **Definition:**

```
enum  id_type   {   list_of_constants   } variable_id;
```

- **Examples:**

```
enum days {ni, po, wt, sr, cz, pi, so};          /* ni == 0, po == 1, ... , so == 6 */
enum days {ni=1, po, wt, sr, cz, pi, so};        /* ni == 1, po == 2, ... , so == 7 */
days Exam, Good = cz;
Exam = Good ;
 //the numerical values can be the same:
enum TW1 {t1, t2, t3 = 0, t4, t5, t6 = 1, t7};    /* t2 == 1 t4 == 1, t5 == 2, t7 == 2 */

enum {A = 0x41, B, C, X = 0x58} sign;
      // enumeration variable:
sign = C;          // correct

//such variable can be assigned with values which do not have corresponding
//constant strings – and the compiler not necessarily will warn about it!

sign = 0x49;      // error
sign = 0x41;      // error – cannot convert int to enum; znak = A; - ok
```

# Enumeration variables - example

```
#include <stdlib.h>     #include <time.h> #include <conio.h>
enum Science { ASTRONOMY, MATH, PHYSICS, CHEMISTRY, BIOLOGY};
int main(int argc, char* argv[]){
    enum Science question;
    srand((int)time(0));
    question = (enum Science)(rand() % 5);    /* random science */
    switch (question){
        case ASTRONOMY:
           printf("Where is the moon during the day? ");
           break;
        case MATH:
           printf("Which are more: integers or floats?");
           break;
        case PHYSICS:
           printf("What generates more gravity: Moon or the Sun? ");
           break;
        case CHEMISTRY:
           printf("Why use moles instead of grams? ");
           break;
        case BIOLOGY:
           printf("Difference between a dove and a dolphin? ");
           break;
    }

    printf("\n\n");
    getch();
    return 0;
}
```

# Variables

- Before using the variables must be declared, i.e., their type must be specified:

```
int a, b, c;  // declaration example
int a;      // as above
int b;      // but
int c;      // we can comment every single variable
            //in single line
```

- <u>Starting values</u> may be assigned just after the declaration:

```
int a = 10;
int b = 60;
int c = a;    //10, no surprise
float eps = 1.0e-5;
```

- **External variables (globals) and static ones** are by default set to 0.
- **Local variables (old name: automatic ones)** without the starting value given in a code by the programmer may have random starting values which may lead to bugs in code execution.

# Variables

- **Data types available in C and their relations between each other:**

```
char          signed char
int           signed           signed int
short         short int        signed short int
long          signed long      long int         signed long int
long  signed long
unsigned char
unsigned int    unsigned
unsigned short  unsigned short int
unsigned long   unsigned long int
unsigned long
float
double
long double
```

# Variables

- **Example of declarations and definitions of variables:**
  - The declaration of a variable informs the compiler about the new variable, but does no reserve the memory – yet. Therefore multiple declarations are possible, e.g., by using **extern int** syntax.
  - Definition of a variable additionally reserves the memory. So each definition is also a declaration.

```
int i;
char a, b, c;
unsigned long long_road;
float dollarPrice;
double mass, density;
int counter = 125,   sum = 0;
float accuracy = 0.0005, error = 0.001;
double power = 15e6,   loses = 1500;
double alfa = 3.34, beta, jota = 15.15, kappa;
```

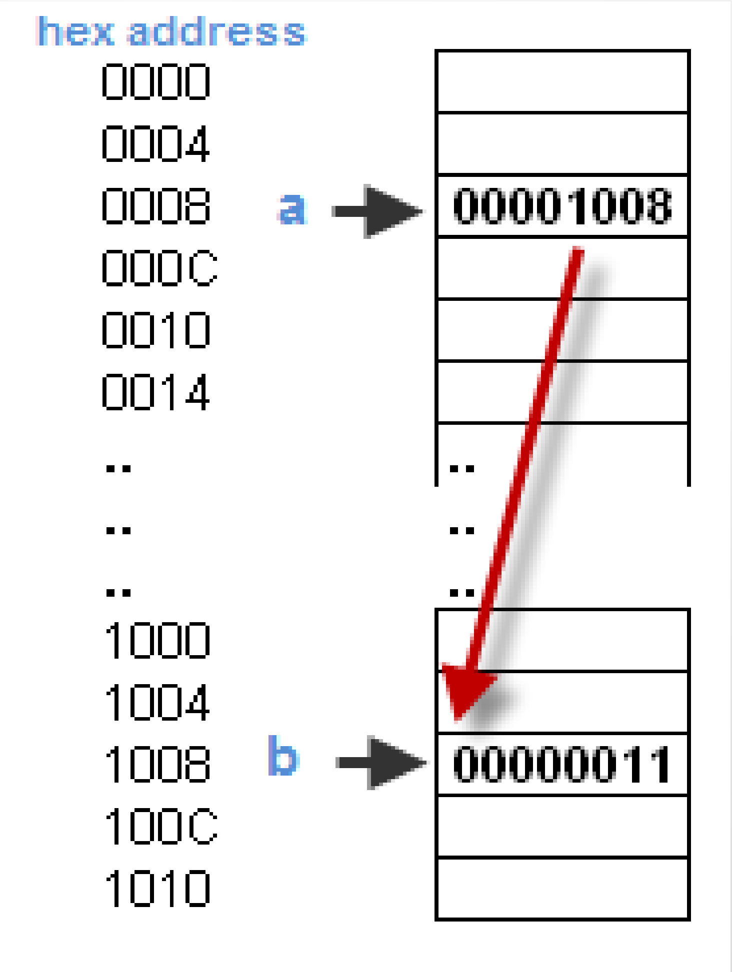
# Hungarian notation

- **To clarify: there is no single notation telling how to name our variables, so these are all good and bad at the same time:**
  - type before name WITHIN THE NAME: int_array_size
  - "pascal" notation: IntArraySize
  - "camel" notation : intArraySize
- **Hungarian notation** is a way of naming variables, where the first letter of a (random) name specifically tells us about the type.

| Prefix | Data type | Example |
|---|---|---|
| b | bool | `bOnceAgain` |
| c | char | `cOptionCode` |
| l | long | `lLongCaliber` |
| n | int | `nCounterFirst` |
| p | pointer | `pAddressOfPriceV` |
| a | table | `anDataTable` |
| s | string | `sSomeString` |

# Pointers

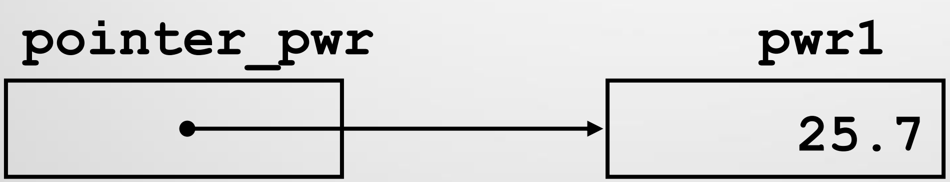
- Pointer, as the name suggest, **points to something**. This „thing" is a specific memory address where a value of some other variable is being stored.
- Memory can be treated as a table of consecutive addresses.
- <u>Pointer</u> itself consists of a group of bytes which represents some address in computer memory.
- In the picture ***a*** is a pointer, pointing at the starting address of some other variable ***b***.
- In 32-bit system pointer has 4 bytes.



# Pointer variables: declarations

- **Examples:**

```
int    *pt_i,  *pt_j;
double   *wsk1,  *wsk2;
float   pwr1 = 25.7,  pwr2,  *pointer_pwr = &pwr1;
void   *any,  *every;
```



# Pointer variables: declarations

- **Examples:**

```
int price = 25, *p_price, **p_p_price;

p_price = &price;

p_p_price = &p_price;
```


```
int   i = 5,   j = 7;

int   *pt = & i,  *pk = & j;

double way, time= 100, *pointer1,
       *pointer1 = & way;
```

# Pointer addressing operator &

- Unary operator: **&** gives us an address of a variable.
- For example:

```
p = &c; // int *p, c;
```

assigns to pointer p an address of memory, where the variable c is being stored. Now both the pointer and the c variable can modify/read the value stored there.

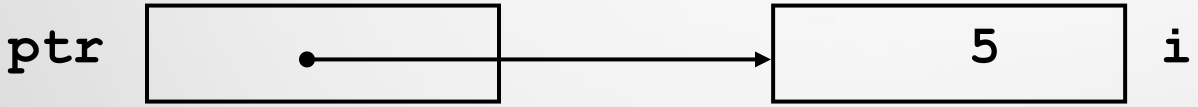
- Examples:

```
int lamps, forks;
int *p_goods;
p_goods= & lamps;
...............
p_goods= & forks;

float   Profit = 2.54, *p_float;
long    *p_long;
void    *p_void;

p_float = & Profit;    // OK
p_long = & Profit;     // ERROR, WRONG TYPE
p_void = & Profit;     // OK (type assigned automatically)
                       // but to read the value:
                       // float new_f = *((float*)p_void);
```

# Indirection operator: * (asterisk)
# AKA: indirect access operator

- Operator: ***** (asterisk, ***indirect access operator*** (*because DIRECTLY we would have used the variable itself,* ******* *is for pointers which INDIRECTLY but still point to some value*)).
- Used with the pointer gives us the value of the variable to which the pointer… points.
- **Examples:**

```
int i = 5, j;
int *ptr = & i;
j = *ptr;               // same as: j = i;
```




```
int x = 1;
int y = 2;
int *ip;        // declaration of pointer for variable type int
ip = &x;        // now ip points to the x variable, so indirectly to value 1
y = *ip;        // now is equal to 1, same as y = x;
*ip = 0;        // now x has value of 0, same as x = 0;
```

# Pointer variables

- When we use a variable, we directly do something with the value stored in such a variable:

```
int x = 0; // assign 0 as value of variable x
```

- When we use the pointer (but without * operator) we are dealing with some address of a memory. Therefore assigning a value directly to the pointer (without using *) is nonsense and it is forbidden:

```
int *p;    // p – pointer to int
p = 10;    // ERROR – p is a pointer, a memory address, which we
           // don’t know AND WILL NOT KNOW WHEN we are writing
           // the code
```

- So we have to use both * and & when using pointers:
  - & - gives us memory address (safely and in a proper way)
  - * used with a pointer changes the address into the value stored there

# Pointer variables

```
int *p;       // pointer for int type
int x = 15;   // integer variable x with starting value = 15
p = &x;       // assign the address of x to the pointer variable p

p = 5546;     // ERROR, the address cannot be assigned directly
p = 0xFA744EA4;   // looks legit, still wrong – WE DO NOT AND WILL NOT KNOW
              // the address while writing the code

              // how to assign new value:

*p = 30;      // indirect access (operator *) allows us to assign 30 to x
              // using x's pointer p (because in third line: p = &x; )

x = 30;       // directly assign new integer value to x
int y = 0;    // new variable y, type int


              // again, three lines doing the same thing:

y = *p;       // assign the value of x by using its pointer p, to variable y
y = x;        // write value of x to variable y directly
y = 30;       // or just write the damn number in the code without this
              // whole pointer mess :)
```

# Pointer variables – again, one last time…

- Summary:

```
int x = 10;
int *ip;
ip = &x;  // now ip points to x, so it has access
          // to its (x) value
```

- So from this moment if we want to do something with the value of x, we can use pointer **ip** and the operator *:

```
// two lines, same thing:
*ip = *ip + 10;
x = x + 10;
```

# References

- Reference in C is like a nickname for a variable. We can have access to the same value with two or more different names – but without this troublesome pointers syntax.
- Every operation done with a reference of a variable is equal in terms of final effects as the operations done with the variable itself.
- **Examples:**

```
int price;
int &ref_k = price;     // this can be done only once (the
connection)
ref_k = 1254;           // same as: price = 1254;

long a, b, &ref_a = a;
ref_a = 12;             // same as: a = 12;
b = ref_a;              // same as: b = a;

float moc_x, &ref_x = moc_x, *wsk_x;
wsk_x = & ref_x;    // same as: wsk_x = & moc_x
wsk_x = ref_x;      // ERROR, similar as e.g.: wsk_x = moc_x;
```

# Constant „variables” – *const*

- Qualifier **const** can be added to the declaration of almost any variable.
- Variable declared with this keyword informs the compiler, that it is now forbidden to assign any value to this variable again (even the same as the current one stored).

```
const float pi = 3.14;
const double e = 2.71828182845905;
```

- Keyword **const** can be used in tables declaration, so no cell of such a table can be modified after assigning the starting values:

```
const char msg[] = "Attention:  ";
int function(const int[]);
```

- Constant variables can be declared with comma:

```
const int days = 7, weeks = 52;
const float pi = 3.14159, e = 2.71828;
const double Avogadro = 6.022E23;
```

Questions?