PROGRAMMAZIONE PROCEDURALE

A.A. 2021/2022
TYPES
**TYPES**

Programs have to store and process different kinds of data, such as integers and floating-point numbers, in different ways. To this end, the compiler needs to know what kind of data a given value represents.

In C, the term **object** refers to a location in memory whose contents can represent values. Objects that have names are also called **variables**.

An object’s **type** determines:
- How much space the object occupies in memory.
- The values that a variable can have.
- The operations that can be performed on that variable.
int main()
{
    int x = 1, y = 2, z = 3;
    printf(" x = %d, y = %d, z = %d \n", x, y, z);
{
    int x = 10;
    float y = 20;
    printf(" x = %d, y = %f, z = %d \n", x, y, z);
{
        int z = 100;
        printf(" x = %d, y = %f, z = %d \n", x, y, z);
    }
    printf(" x = %d, y = %d, z = %d \n", x, y, z);
}
    return 0;
}
TYPES IN C

- **Basic** type
  - Standard and extended integer types
  - Real and complex floating-point types
- **Enumerated types**
- The type `void`
- **Derived** types
  - Pointer types
  - Array types
  - Structure types
  - Union types
  - Function types
The basic types and the enumerated types together make up the **arithmetric types**.

The **arithmetric types and the pointer types** together are called the **scalar types**.

Finally, **array types** and **structure types** are referred to collectively as the **aggregate types**.

A **function type** describes the interface to a function; that is, it specifies the type of the function’s return value, and may also specify the types of all the parameters that are passed to the function when it is called.
INTEGER
INTEGERS

- There are five signed integer types. Most of these types can be designated by several synonyms:
  - signed char
  - int [signed, signed int]
  - short [short int, signed short, signed short int]
  - long [long int, signed long, signed long int]
  - long long (C99) [long long int, signed long long, signed long long int]

- C defines only the *minimum* storage sizes of the other standard types: the size of type short is at least two bytes, long at least four bytes, and long long at least eight bytes. Furthermore, although the integer types may be larger than their minimum sizes, the sizes implemented must be in the order:
  - sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)
In C there is no true or false
0 is false
Any value different from 0 is true
1
-25
123456

```c
if (3)
    printf("YES\n");
else
    printf("NO\n")
```

```c
if (0)
    printf("YES\n");
else
    printf("NO\n")
```
BOOLEANS

C99 introduced the unsigned integer type _Bool to represent Boolean truth values.

The Boolean value true is coded as 1, and false is coded as 0.

If you include the header file stdbool.h in a program, you can also use the identifiers bool, true, and false. The macro bool is a synonym for the type _Bool, and true and false are symbolic constants equal to 1 and 0.
The type char is also one of the standard integer types. However, the one-word type name char is synonymous either with signed char or with unsigned char, depending on the compiler.

- It occupies 1 byte
- Check the correspondence in the ASCII table
  - http://www.asciitable.com
You can do arithmetic with character variables. It’s up to you to decide whether your program interprets the number in a char variable as a character code or as something else.

```c
char ch = 'A';       // A variable with type char.
printf("The character \%c has the character code \%d.\n", ch, ch);
printf("\%c", ch + 1);
```

The character A has the character code 65.

B
BINARY/OCTAL/HEXADECIMAL
In mathematics and digital electronics, a binary number is a number expressed in the binary numeral system or base-2 numeral system which represents numeric values using two different symbols: typically 0 (zero) and 1 (one).

Because of its straightforward implementation in digital electronic circuitry using logic gates, the binary system is used internally by almost all modern computers and computer-based devices. Each digit is referred to as a bit.
HEXADECIMAL

Hexadecimal (also base 16, or hex) is a positional numeral system with a radix, or base, of 16.

It uses sixteen distinct symbols, most often the symbols 0–9 to represent values zero to nine, and A, B, C, D, E, F (or alternatively a, b, c, d, e, f) to represent values ten to fifteen.

Hexadecimal numerals are widely used by computer system designers and programmers. As each hexadecimal digit represents four binary digits (bits), it allows a more human-friendly representation of binary-coded values.

One hexadecimal digit represents a nibble (4 bits), which is half of an octet or byte (8 bits).
The octal numeral system, is the base-8 number system, and uses the digits 0 to 7.
FROM BINARY AND HEXADECIMAL TO DECIMAL

\[ d_n2^n + d_{n-1}2^{n-1} + \ldots + d_12^1 + d_02^0 = N_{10} \]

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hexadecimal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

1001 = 9

10\text{X}1 = ?

002B = 43
FROM DECIMAL TO BINARY

<table>
<thead>
<tr>
<th>÷2</th>
<th>remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>156</td>
<td>0</td>
</tr>
<tr>
<td>78</td>
<td>0</td>
</tr>
<tr>
<td>39</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

10011100 = 156
FROM DECIMAL TO HEXADECIMAL

\[
\begin{array}{c|c|c}
\text{÷16} & \text{remainder} \\
1565 & 13 = d \\
97 & 1 \\
6 & 6 \\
0 & \\
\end{array}
\]

Octal follows the same algorithm
HOW MANY NUMBERS CAN I REPRESENT?

- From 0 to \(2^N - 1\)
- With 8 bits (one byte)
  - From 0 to 255
    
    \[
    \begin{array}{c c c c c}
    0 & 0 & 0 & 0 & 0 \\
    1 & 1 & 1 & 1 & 1 \\
    \end{array}
    \]

- However, it is useful to also represent negative numbers
- Different representations
  - Sign and magnitude
  - Two’s complement
REPRESENTATION IN MEMORY
It uses one bit (usually the leftmost if big endian) to indicate the sign. "0" indicates a positive integer, and "1" indicates a negative integer. The rest of the bits are used for the magnitude of the number. E.g.:

- 1001 1000
- -24

if 1001 1000 is used to represent positive numbers only?

- 152
HOW MANY NUMBERS CAN I REPRESENT?

- With n bits
  - From \((-2^{N-1} + 1)\) to \((2^{N-1} - 1)\)
  - and \pm 0

- For instance, with 8 bits,
  - from -127 to +127

    1111 1111 0111 1111
A PROBLEM

- Two different representations of 0
  - 0000 0000  (+0)
  - 1000 0000  (-0)

- A solution is a different representation: two’s complement
# TWO'S COMPLEMENT

<table>
<thead>
<tr>
<th>Binary value</th>
<th>Two's complement</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td>01111110</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>01111111</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>10000000</td>
<td>−128</td>
<td>128</td>
</tr>
<tr>
<td>10000001</td>
<td>−127</td>
<td>129</td>
</tr>
<tr>
<td>10000010</td>
<td>−126</td>
<td>130</td>
</tr>
<tr>
<td>⋮</td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td>11111110</td>
<td>−2</td>
<td>254</td>
</tr>
<tr>
<td>11111111</td>
<td>−1</td>
<td>255</td>
</tr>
</tbody>
</table>

In two's-complement, there is only one zero, represented as 00000000. Negating a number (whether negative or positive) is done by inverting all the bits and then adding one to that result.
HOW TO GET THE COMPLEMENTARY

- From a positive number to its complement: from 5 to -5
- Flip all the bits and then + 1
- 0000 0101 (value 5)
  ✓ 1111 1010 (flip)
  ✓ 1111 1011 (+1)
- When an integer number starts with 1 it means that it is negative; if it is negative you have to do the inverse
  ✓ 1111 1011 value (-5)
  ✓ 1111 1010 (-1)
  ✓ 0000 0101 (flip)
HOW MANY NUMBERS CAN I REPRESENT?

- With n bits
  - From \((-2^{N-1})\) to \((2^{N-1} - 1)\)
  - There is no “-0”, so it is possible to represent one more negative number

- For instance, with 8 bits,
  - from -128 to +127
    
  \[
  \begin{array}{c|c}
  \text{1000 0000} & \text{0111 1111} \\
  \end{array}
  \]

- The rule in the previous slide to get the complimentary does not work because 128 is not representable with 8 bits in two’s complement
OPERATION EXAMPLES

0000 1111  (15)
+ 1111 1011  (-5)

0111  (7)
+ 0011  (3)

0000 1111  (15)
- 1111 1011  (-5)

-2^{(4-1)} --- 2^{(4-1)}-1

Ok!

01111 111  (carry)
0000 1111  (15)
+ 1111 1011  (-5)
=============
0000 1010  (10)

Arithmetic overflow!

011  (carry)
0111  (7)
+ 0011  (3)
=============
1010  (-6) invalid!

Ok!

11110 000  (borrow)
0000 1111  (15)
- 1111 1011  (-5)
=============
0001 0100  (20)
ENDIANNESS

Endianness refers to the sequential order used to numerically interpret a range of bytes in computer memory as a larger, composed word value.

It also describes the order of byte transmission over a digital link.

Words may be represented in big-endian or little-endian format, depending on whether bits or bytes or other components are numbered from the big end (most significant bit) or the little end (least significant bit).

As examples, the IBM z/Architecture mainframes and the Motorola 68000 series use big-endian while the Intel x86 processors use little-endian (in the 1970s).
MEMORY REPRESENTATION

- An array of bytes
- Every byte has its logical address (a positive number)
- A **logical address** is the **address** at which an item appears to reside from the perspective of an executing application program.

2 represented on 4 bytes (little end.):
0100 0000
0000 0000
0000 0000
0000 0000

For 64-bit architectures the upper limit $2^{64} - 1$
0010 0100
BIG ENDIAN, LITTLE ENDIAN

Big endian: right to left

Two’s complement from now on

Little endian: left to right

Big endian:
0010 = 2

Big endian:
1010 = -6

Little endian
0010 = 4

Little endian
1010 = 5

Big endian:
0000 0000
0000 0000
0000 0000
0000 0000
0000 0010

Little endian:
0100 0000
0000 0000
0000 0000
0000 0000
0000 0000
BACK TO C
### Representation

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage size</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>(same as either signed char or unsigned char)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned char</td>
<td>one byte</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>signed char</td>
<td>one byte</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>int</td>
<td>two bytes or four bytes</td>
<td>-32,768 or -2,147,483,648</td>
<td>32,767 or 2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>two bytes or four bytes</td>
<td>0</td>
<td>65,535 or 4,294,967,295</td>
</tr>
<tr>
<td>short</td>
<td>two bytes</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>two bytes</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>long</td>
<td>four bytes</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>four bytes</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long long (C99)</td>
<td>eight bytes</td>
<td>-9,223,372,036, 854,775,808</td>
<td>9,223,372,036, 854,775,808</td>
</tr>
<tr>
<td>unsigned long long (C99)</td>
<td>eight bytes</td>
<td>0</td>
<td>18,446,744,073, 709,551,615</td>
</tr>
</tbody>
</table>

So, how are integer represented in C? Sign magnitude or two’s complement?
To obtain the exact size of a type or a variable, use the `sizeof` operator. The expressions `sizeof(type)` and `sizeof(expression)` yield the storage size of the object or type in bytes. If the operand is an expression, the size is that of the expression’s type.

```c
int iIndex,
iIndex = 1000;
```

`sizeof(int)` and `sizeof(iIndex)` returns 4
LIMITS

You can find the value ranges of the integer types for your C compiler in the header file `limits.h`, which defines macros such as INT_MIN, INT_MAX, UINT_MAX, and so on.

```c
int main() {
    printf(" char %d %d %d\n", sizeof(char), CHAR_MIN, CHAR_MAX );
    printf(" int %d %d %d\n", sizeof(int), INT_MIN, INT_MAX );
    return 0;
}
```
FLOAT
C also includes special numeric types that can represent non-integers with a decimal point in any position. The standard floating-point types for calculations with real numbers are as follows:

- float: for variables with single precision
- double: for variables with double precision
- long double: for variables with extended precision
A floating-point value can be stored only with a limited precision, which is determined by the binary format used to represent it and the amount of memory used to store it.

The precision is expressed as a number of significant digits. So that its conversion back into a six-digit decimal number yields the original six digits.

The position of the decimal point does not matter, and leading and trailing zeros are not counted in the six digits.

The numbers 123,456,000 and 0.00123456 can both be stored in a type with six-digit precision.
AR. OPERATIONS IN DOUBLE PREC.

- In C, arithmetic operations with floating-point numbers are performed internally with double or greater precision.

```c
float height = 1.2345, width = 2.3456;  // Float variables have single precision.
double area = height * width;          // The actual calculation is performed with double (or greater) precision.
```

- If you assign the result to a float variable, the value is rounded as necessary.
FLOATS

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The macros FLT_MIN, FLT_MAX, and FLT_DIG indicate the value range and the precision of the float type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage size</th>
<th>Value range</th>
<th>Smallest positive value</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>±3.4E+38</td>
<td>1.2E−38</td>
<td>6 digits</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>±1.7E+308</td>
<td>2.3E−308</td>
<td>15 digits</td>
</tr>
<tr>
<td>long double</td>
<td>10 bytes</td>
<td>±1.1E+4932</td>
<td>3.4E−4932</td>
<td>19 digits</td>
</tr>
</tbody>
</table>
E NOTATION

It's know as E notation, which is plain text representation of scientific notation.

1.234e+56 means $1.234 \times 10^{56}$
IEEE 754 FORMAT

Each finite number is described by three integers: $s = \text{a sign}$ (zero or one), $c = \text{a significand}$ (or “mantissa”), $q = \text{an exponent}$. The numerical value of a finite number is
\[ (-1)^s \times c \times b^q \]
where $b$ is the base (e.g., 2 or 10), also called \textit{radix}.

For example, if the base is 10, the sign is 1 (indicating negative), the significand is 12345, and the exponent is $-3$, then the value of the number is $-12.345$

The 754 format for single precision is

- Sign 1 bit
- Exponent 8 bits
- Significand 23 bit
EXAMPLE

value = (-1)^{\text{sign}} \times \left( 1 + \sum_{i=1}^{23} b_{23-i} \times 2^{-i} \right) \times 2^{(e-127)}

1 + 1 \times 2^{-2} = 124 - 127

value = (+1) \times 1.25 \times 2^{-3} = +0.15625
DOUBLE AND EXTENDED PRECISION

Double precision: `double` in C

Extended precision: `long double` in C
Rounding error is inherent in floating-point computation

```c
#include <stdio.h>

int main()
{
    int a = 16777217;
    float b = a;
    printf("%f\n", b);
}
```

```
16777216.000000
```

Diagram:
- **REAL NUMBERS**
- **FLOATING-POINT NUMBERS**
MORE ERRORS

```cpp
float height = 1.2345, width = 2.3456; // Float variables have
  // single precision.

double area = height * width; // The actual calculation
  // is performed with
  // double precision.
```

```
float height = 1.2345, width = 2.3456;
float area = height * width;
```
HOW TO AVOID PROBLEMS

The easiest way to avoid accumulating error is to use high-precision floating-point numbers (this means using **double instead of float**). On modern CPUs there is little or no time penalty for doing so, although storing doubles instead of floats will take twice as much space in memory.
ENUM
ENUM TYPES

Enumerations are integer types that you define in a program.

The definition of an enumeration begins with the keyword `enum`, possibly followed by an identifier for the enumeration, and contains a list of the type’s possible values, with a name for each value:

```c
enum [identifier] { enumerator-list };  
```

An example is

-`enum color { black, red, green, yellow, blue, white=7, gray };`
- the constants listed have the values 0, 1, 2, 3, 4, 7, 8
EXAMPLES

```c
enum color fgColor = blue;  // Define two variables
bgColor = yellow;           // of type enum color.

void setFgColor( enum color fgc ); // Declare a function with a parameter
                                    // of type enum color.
```

- You may perform ordinary arithmetic operations with variables of enumerated types:
  - `red + red = 2`

- Different constants in an enumeration may have the same value:
  - `enum signals { OFF, ON, STOP = 0, GO = 1, CLOSED = 0, OPEN = 1 };`
enum boolean { false, true };

enum boolean check;

#include <stdio.h>

enum week { sunday, monady, tuesday, wednesday, thursday, friday, saturday };

int main(){
    enum week today;
    today = wednesday;
    printf("Day %d",today+1);
    return 0;
}

Day 4
WHEN TO USE ENUM

- You should always use enums when a variable (especially a method parameter) can only take one out of a small set of possible values.

- If you use enums instead of integers, you avoid errors from passing in invalid constants, and you document which values are legal to use. Moreover, it is more mnemonic to use them instead of integer values.
VOID
The type specifier void indicates that no value is available.

Consequently, you cannot declare variables or constants with this type, but you can use:

- void in function declarations
- Void expressions
- Pointers to void (back to this when we will study pointers)
VOID IN FUNCTIONS

A function with no return value has the type void.

✓ void error(int a) {}

void in the parameter list of a function prototype indicates that the function has no parameters:

✓ void printMenu(void) {}

The compiler issues an error message if you try to use a function call such as printMenu(3).