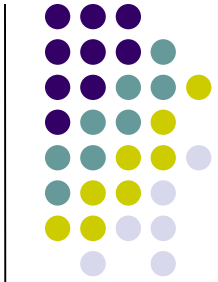
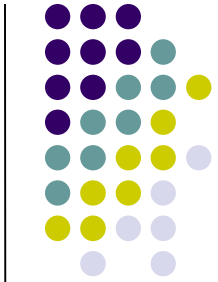
Type conversion



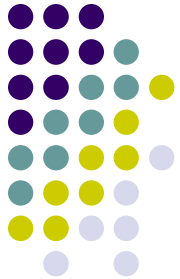
- When two different types are combined in some operations, there is a need to convert types.
- Conversion can be:
  - **Implicit**
    - **Assignment**
    - **Function call**
      - **Parameter passing**
      - **Return value passing**
    - **Arithmetic conversions** – Most often cause of confusion.
    - **Type promotion**
  - **Explicit**
    - **Also called "casting"** – `(int)x`

Type conversion



```
int foo(float x) {
    return x;
}

float r;

void main() {
    unsigned long y;

    y = 5;

    r = foo(y);
}
```

Type of literals



- **Every operand (expression) has a type!**
- For literals, type is determined by suffix and the existence of decimal point.
- Examples:

```
2U                  -37                         'c' // not the same as "c"
3L 3l 31            051         // 41
5UL                 0x2b        // 43
7LL                 0xFFFFFFD1  // -47
11ULL
13.0                3.14159     // 3.14159
17.                 6.02e23     // 6.02 x 10^23
19.0F               1.6e-19     // 1.6 x 10^-19
23.F
29.0L
31.L
```

```
int_least16_t niz[30000];
int_fast16_t i;
for (i=0; i<INT16_C(30000); i++)
```

16b

```
short niz[30000];
int i; // 32b
```

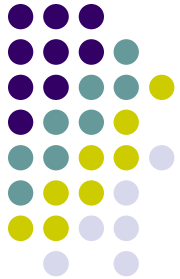
16b

```
short niz[30000];
short i; // 16b
```

64b

```
short niz[30000];
int i; // 128b
```

Type promotion



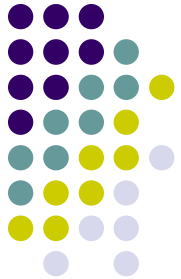
Type promotion is a special case of implicit conversion.

- Integer promotions
  - In operations on types smaller than int (char or short) operands are firstly promoted to corresponding int (signed or unsigned), then the operation is performed, and finally the result is converted back to the original type by truncating excess MSBs.

```
char a;
char b;
char c = a + b;
```

- Float promotion
  - float is promoted to double

Arithmetic conversions: example



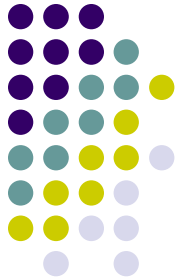
```
double polovina = 1/2;
```

What will be the value in **`polovina`**?
Why?

What are the types of first and the second operand?
What is the type of the result?

Every operand, i.e. expression, has type, and therefore expression “1/2” has type, and that type does not depend in any way of the type of the variable on the left side of assignment!

Arithmetic conversions: example



Literals 1 and 2 are of int type.

If both operands are **int**, so is the type of the result. Therefore, the resulting integer value of the expression is 0.

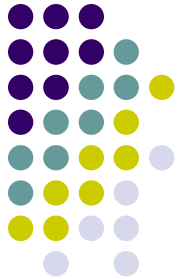
```
double polovina = 1.0/2.0;
```

but it is also OK to do this:

```
double polovina = 1.0/2;
```

```
In case operands are int variables:
double kolicnik = ((double)x)/y;
```

Arithmetic conversions

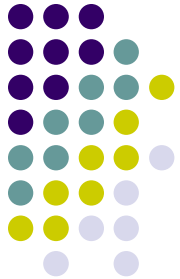


- Numerical types are ranked, from the lowest to the highest, like this:

  `char, short, int, long, long long, float, double, long double`
- If operands of two different types participate in a binary operation, then operand with lower ranking type is firstly converted to the higher ranking type of the other operand.
- Signedness is considered separately, in the next step. If the types of operands in binary operation are not of the same signedness, then singed operand is converted to unsigned. (Of course, this only works if both types are integer types)
- Lesson: be very careful with this!
- It is best to mostly use explicit conversions, if you need them.
- Compilers will usually generate warnings form many problematic cases. Take warning seriously.

- For details look at chapter 6.3 of C standard, in particular: chapter 6.3.1.8.

Potential problem related to conversion



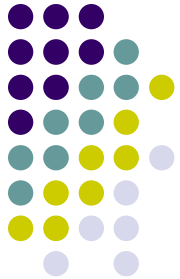
- Dangerous conversions
  - **Value loss**: Conversion of larger (wider) type into smaller (narrower) type
  - **Sign loss**: Conversion between signed and unsigned types
  - **Precision loss**: Conversion of higher precision type to lower precision type
- Safe conversions
  - Conversion of integer type into larger (wider) integer type
  - Conversion of floating point type into larger (wider and more precise) floating point type

Pointer conversions

- Conversion that you should be very careful about:
  - between pointer and integer type
    - This conversion is undefined by the standard, meaning that any code that uses this conversion is not portable, and maybe even unsafe.
    - Sometimes it is inevitable. For example, when we want to access memory mapped registers, or some other hardware specific thigs. Then we have to read the platform manual carefully.
  - between pointers to two different types
    - In this example problem arises due to incompatible alignment or endianness (alignment and endianness will be covered in later lectures)

```
uint8_t p1[4];
uint32_t* p2;
p2 = (uint32_t*)p1; /* incompatibile alignment */
```

Pointer conversions



- Valid conversions:
  - From void pointer (void*) into pointer to some other type, and vice versa.
  - Conversion of NULL value into pointer**.** NULL is actually literal 0 (in many implementations it is explicitly converted to void*) and that represents the only defined conversion from integer to pointer.
  - Conversion of a pointer that has value NULL into pointer to any other type. (Consequence of the previous point)
  - Advice: always use symbol NULL when working with pointers, never 0.

- Conversions that have to be explicit:
  - Object pointer into another object pointer..
  - Function pointer into another function pointer..
- Invalid conversions:
  - Function pointer into object pointer (including void pointer), and vice versa.