PROGRAMMAZIONE PROCEDURALE

A.A. 2022/2023

TYPE CONVERSIONS

CONVERSIONS

- Operands of different types can be combined in one operation

```
double dVar = 2.5;
dVar = dVar * 3;
if ( dVar < 10L ) { /* ... */ }
```

- When the operands have different types, **the compiler tries to convert them to a uniform type before performing the operation**.

IMPLICIT CONVERSIONS

- The compiler provides ***implicit*** *type conversions* when
  - ✓operands have mismatched types, or
  - ✓when you call a function using an argument whose type does not match the function’s corresponding parameter
  - ✓when initializing variables or otherwise assigning values to them
- If the necessary conversion is not possible, the compiler **issues an error message**
- Some other times you can get a **warning message**

CAST OPERATORS

- You can also convert values from one type to another ***explicitly*** using the *cast operator*:

(*type_name*) *expression*

3.3333

```
int sum = 10, count = 3;
double mean =  (double) sum / (double) count;
```

- The value of **sum** in this example is first converted to type **double**
- The compiler must then implicitly convert the divisor, the value of count, to the same type before performing the division.
- You should always use the cast operator whenever there is a possibility of losing information. Explicit casts avoid compiler warnings.

CONVERSION OF ARITHMETIC TYPES

HIERARCHY OF TYPES

- When arithmetic operands have different types, the implicit type conversion is governed by the types' **conversion rank.**
  - ✓Any two unsigned integer types have different conversion ranks. If one is wider than the other, then it has a higher rank.
  - ✓Each signed integer type has the same rank as the corresponding unsigned type.
  - ✓The standard integer types are ranked in the order:
    - _Bool < char < short < int < long < long long
  - ✓The floating-point types are ranked in the following order:
    - float < double < long double
  - ✓The lowest-ranked floating-point type, **float**, has a higher rank than any integer type.
  - ✓Enum have the same rank as int.

INTEGER PROMOTION

- In any expression, you can always use a value whose type ranks lower than **int** in place of an operand of type **int** or **unsigned int**.
- In these cases, the compiler applies *integer promotion*: any operand whose type ranks lower than **int** is automatically converted to the type **int**, provided **int** is capable of representing all values of the operand’s original type. If **int** is not sufficient, the operand is converted to **unsigned int**.
- Operations in the CPU are executed on 4 bytes at least

EXAMPLE

120

```
#include <stdio.h>
int main()
{
    char a = 30, b = 40, c = 10;
    char d = (a * b) / c;
    printf ("%d ", d);
    return 0;
}
```

At first look, the expression (a*b)/c seems to cause arithmetic overflow because signed characters can have values only from -128 to 127 (in most of the C compilers), and the value of subexpression ‘(a*b)’ is 1200 which is greater than 128. But integer promotion happens here in arithmetic done on char types and we get the appropriate result without any overflow.

USUAL ARITHMETIC CONVERSIONS

- The *usual arithmetic conversions* are the implicit conversions that are automatically applied to operands of different arithmetic types for most operators.
- The usual arithmetic conversions are performed implicitly for the following operators:
  - ✓Arithmetic operators with two operands: *, /, %, +, and –
  - ✓Relational and equality operators: <, <=, >, >=, ==, and !=

EXAMPLE

```
int x = 0;
int i = -1;
unsigned int limit = 200U;
long n = 30L;

if ( i < limit )
    x = limit * n;

printf("%d\n", x);                    0
```

WHAT DOES IT HAPPEN?

- The usual arithmetic conversions are applied as follows:
  - ✓If either operand has a floating-point type, then the operand with the lower conversion rank is converted to a type with the same rank as the other operand. Real types are converted only to real types.
  - ✓If both operands are integers, integer promotion is first performed on both operands. If after integer promotion the operands still have different types, conversion continues as follows:
    - **Rule1**: If one operand has an ***unsigned*** **type*T*** whose conversion rank is at least as high as that of the other operand’s type, then the other operand is converted to ***unsigned typeT***.
    - **Rule2**: Otherwise, one operand has a ***signed*** **type*T*** whose conversion rank is higher than that of the other operand’s type. The other operand is converted to ***signed*** **type*T*** only if ***signed typeT*** is capable of representing all values of its previous type. If not, then both operands are converted to the **unsigned type** that corresponds to the ***signed*** **type*T*** (***unsigned typeT***).

IN ITALIANO (REGOLA1)

- x op y
- La Regola1 dice che, se in un’espressione il tipo di x è **unsigned TipoT** (parliamo di tipi interi) il cui grado di conversion è per lo meno tanto alto tanto quello dell’altro operando (y), allora il tipo dell’altro operando (y) è convertito ad **unsigned TipoT**.
- Nell’esempio, l’operando di tipo int (i) viene convertito a unsigned int (il tipo di limit).

IN ITALIANO (REGOLA2)

- x op y
- Altrimenti se la prima regola non si applica, forse si applica la seconda. Se x ha tipo con segno **signed TipoT** (quindi NON unsigned) il cui grado di conversione è più elevato di quello dell’altro operando (y), si applica questa regola. L’altro operando (y) è convertito a **signed TipoT** solo se questo tipo è in grado di rappresentare tutti i valori di y. Altrimenti, tutti e due gli operandi (x e y) sono convertiti a **unsigned TipoT**.
- Nell’esempio, abbiamo n (long, 32 bit) e limit (unsigned int, 32 bit). Un long rappresenta fino a 2^{31}-1, mentre un unsigned int fino a 2^{32}-1. Per questo motivo, sia n che limit vengono convertiti a unsigned long.

EXAMPLES

```
int x = 0;
int i = -1;
unsigned int limit = 200U;
long n = 30L;

if ( i < limit )            <-- Rule1
    x = limit * n;          <-- Rule2

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

- In this example, to evaluate the comparison in the if condition, the value of i, –1, must first be converted to the type **unsigned int**. The result is a large positive number (next slide). Hence, **the if condition is false**.
- In the if, the value of **limit** is converted to **n**'s type, **long**, if the value range of **long** contains the whole value range of **unsigned int**. If not— for example, if both **int** and **long** are 32 bits wide—then both multiplicands are converted to **unsigned long**.

EXPLANATION OF THE EXAMPLE

- How is -1 represented in an **int**? (little endian, two's complement)
  - ✓11111111111111111111111111111111 (32 bits set to 1)
  - ✓An unsigned int is 32 bits (in my compiler)
  - ✓-1 is implicitly converted to unsigned int: its value is 4,294,967,295

OTHER IMPLICIT CONVERSIONS

- The compiler also automatically converts arithmetic values in the following cases:
  - ✓In assignments and initializations, the value of the right operand is always converted to the type of the left operand.
  - ✓In function calls, the arguments are converted to the types of the corresponding parameters.
  - ✓In return statements, the value of the return expression is converted to the function's return type.

MORE

- In a compound assignment, such as x += 2.5 (x = x + 2.5), the values of both operands are first subject to the usual arithmetic conversions, then the result of the arithmetic operation is converted, as for a simple assignment, to the type of the left operand
- If x has type int, x is converted to double and then the result x+ 2.5 (which has type double) is converted back to int

EXAMPLE

```
#include <math.h>          // Declares the function double sqrt( double ).
int i = 7;
float x = 0.5;             // The constant value is converted from double to float.

i = x;                     // The value of x is converted from float to int.

x += 2.5;                  // Before the addition, the value of x is converted to
                           // double. Afterward, the sum is converted to float for
                           // assignment to x.

x = sqrt( i );             // Calculate the square root of i:
                           // The argument is converted from int to double; the
                           // return value is converted from double to float for
                           // assignment to x.

long my_func() {           // The constant 0 is converted to long, the function's
/* ... */                  // return  type.
return 0;
}
```

CONVERSIONS TO UNSIGNED INTEGER TYPES

- Integer values are always preserved if they are within the range of the new unsigned type
  - ✓Between 0 and U*type*_MAX
- For values outside the new unsigned type's range, the value after conversion is the value obtained by adding/subtracting (U*type*_MAX + 1) as many times as necessary until the result is within the range of the new type.

```
unsigned short n = 1000;      // The value 1000 is within the range of
                              // unsigned short: ok

n = -1;                       // the value –1 must be converted.
```

- –1 + (USHRT_MAX + 1) = USHRT_MAX, the final statement in the previous example is equivalent to n = USHRT_MAX;

FLOATS AND INTEGERS

- To convert a real floating-point number to an unsigned or signed integer type, the compiler discards the fractional part.
- If the remaining integer portion is outside the range of the new type, the result of the conversion is undefined.

```
double x = 2.9;
unsigned long n = x;            // The fractional part of x is simply lost.
```

n = 2

CONVERSIONS TO SIGNED INTEGER TYPES

- The problem of exceeding the target type's value range can also occur when a value is converted from an integer type, whether signed or unsigned, to a signed integer type;
  - ✓for example, when a value is converted from the type **long** or **unsigned int** to the type **int**.
- The result of such an overflow on conversion to a signed integer type, unlike conversions to unsigned integer types, is left up to the implementation.
  - ✓Most compilers discard the highest (most significative) bits of the original value's binary representation and interpret the lowest bits according to the new type.

EXAMPLE

```
#include<stdio.h>
#include<limits.h>
int main() {
    long long int a= (LLONG_MAX-UINT_MAX)+1;
    int b= a;

    printf("b: %d\n", b);
    printf("a: %lld\n", a);
}
```

```
b: 1
a: 9223372032559808513
```

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110 -

11111111 11111111 11111111 11111111 00000000 00000000 00000000 00000000 =

00000000 00000000 00000000 00000000 11111111 11111111 11111111 11111110 +

10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 =

10000000 00000000 00000000 00000000 ~~11111111 11111111 11111111 11111110~~

CONVERSIONS TO REAL FLOATING-POINT TYPES

- Not all integer values can be exactly represented in floating-point types.
- For example, although the value range of the type **float** includes the range of the types **long** and **long long**, float is precise to only six decimal digits.
  - ✓Thus, some long values cannot be stored exactly in a float object.
  - ✓The result of such a conversion is the next lower or next higher representable value

```
float r_var = 16777217;
double l_var = 16777217;
printf("The rounding error (l_var - r_var) is %.2f\n", l_var - r_var);
printf("r_var is %.2f\n", r_var);
```

```
The rounding error (l_var - r_var;) is 1.00
r_var is 16777216.00
```

MORE ON

- Any value in a floating-point type can be represented exactly in another floating- point type of greater precision.
  - ✓Thus when a double value is converted to long double, or when a float value is converted to double or long double, the value is exactly preserved.
- In conversions from a more precise to a less precise type, however, the value being converted may be beyond the range of the new type.
  - ✓If the value exceeds the target type’s range, the result of the conversion is undefined.
- If the value is within the target type’s range, but not exactly representable in the target type’s precision, then the result is the next smaller or next greater representable value.