Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Outline

Integers: `signed`, `unsigned` representations

- Integer types (`char`, `short`, `int`, `long`) may be:
  1. `signed`: bytes are interpreted as number with sign: if negative in two-complement
     - ★ by default `short`, `int`, `long` types are signed
     - ★ the default `char` may be `signed`/`unsigned` depending on implementation
  2. **`unsigned`**: bytes representation interpreted as positive number
     - ★ `unsigned` variables must be declared explicitly as in

```
unsigned int a;
```

- Examples on 8 bits

binary	`signed` value	`unsigned` value
11111111	-1	255
00000010	2	2
10000000	-128	128
10000001	-127	129

Integers: limits

- List of limits

num. bytes	signed min	signed max	unsigned min	unsigned max
n	-2^{8n-1}	$2^{8n-1}-1$	0	$2^{8n}-1$
1	-128	127	0	255
2	-32768	32767	0	65535
4	-2147483647	2147483648	0	4294967295
8	$\approx -8 \times 10^{18}$	$\approx 8 \times 10^{18}$	0	$\approx 16 \times 10^{18}$

- by including the header file `#include <limits.h>` with

```
#include <limits.h>
```

you can use the macros `INT_MIN`, `INT_MAX`, `USHRT_MAX`, etc. for the maximum/minimum constants of all the types

Integers: constants

- In C code integer constants are
  1. sequences of digits without a decimal dot "."
     - ★ if they start with "`0x`", they are interpreted in hexadecimal
     - ★ if they start with "`0`", they are interpreted in octal
     - ★ otherwise they are interpreted as decimal
  2. single characters within `'` (as in `'a'`) to represent the ASCII code of that character
  3. Best expression to write the ASCII code of the digit `n` is `'0'+n`
- constants may also be explicitly declared as unsigned, long or both, otherwise they are `int`
  - "`345U`" for unsigned
  - "`234L`" for long
  - "`2367LU`" for unsigned long
- *`test-int-const.c`*

Integer promotion

- The usage of variables shorter than `int` such as `char` or `short`, may be good to save memory in memory constrained devices (embedded systems)
- Still, operations by the CPU are more conveniently performed over the "word", which is as long as an `int`
- `char` and `short` variables are **promoted** to `int` when they appear in expressions

  `test-promote-char.c`
- also, the effect of `int`-promotion and mixing signed and unsigned integers in the same expression may generate unexpected results

Outline

The type boolean does not exist

- Although conditions do exist
- When evaluated as condition, a numerical expression `<expr>` is

  false if `<expr>` is equal to zero

  true otherwise
- Example of a for loop

```
/* Compact way to run 10 iterations */
for (i=10; i; i--) {
    /* body of the for loop */
}
```

Outline

Floating point: representation

- Two types for floating-point representation: `float`, `double`
- A floating-point number n is represented by
  - one bit for *sign* s of the number;
  - "biased" *exponent* e
    (biased exponent introduced to give a special meaning to $e = 0$)
  - *fraction* f, that is the sequence of digits after the "1,";

  Standard IEEE 754-1985
- Floating-point constants are written in C with the decimal dot "." or with the letter **e** (or **E**)

```
double a;

a = 10.0;
a = .3;
a = 84753933.;
a = 918.7032E-4;
a = 4e+12;
a = 3.5920E12;
```

Floating point: imprecise arithmetic

- The finite number of bits to represent real numbers introduces an approximation error
- The approx error may even lead to violation of basic properties, such as the associativity of addition

```
double d1 = 1e30, d2 = -1e30, d3 = 1.0;
printf("%lf\n", (d1 + d2) + d3);
printf("%lf\n", d1 + (d2 + d3));
```

- Also, if a floating point number needs to be tested if it is equal to zero **never** use == 0 or != 0
- Always, test proximity to zero (not equality) by some code as

```
double a, b, tol;
...
tol = 1e-6;    /* relative tolerance */
if (fabs(a-b) < tol*a) { ... }
```

- Testing now many conditions
  test-constants.c

Outline

Automatic type conversion

- In expressions with operands of different types, each operand is converted in the most expressive format
- Order of expressiveness

  `char < short < int < long < float < double`

- Example of automatic conversion in expressions

```
if (3/2 == 3/2.0) {
        printf("VERO :-)\n");
} else {
        printf("FALSO :-(\n");
}
```

Automatic type conversion

- In expressions with operands of different types, each operand is converted in the most expressive format
- Order of expressiveness

  `char < short < int < long < float < double`
- Example of automatic conversion in expressions

```
if (3/2 == 3/2.0) {
        printf("VERO :-)\n");
} else {
        printf("FALSO :-(\n");
}
```

- It is printed `FALSO :-(`

Conversion by assignment

- An expression assigned to a variable is converted to the type of the assigned variable
- Assignments to same type of smaller size are truncated
- Example of conversion by assignment

```
double a=1025.12;
int i;
unsigned char c;

i = a;  // i gets 1025 (fractional part truncated)
c = i;  // c gets 1    (least significant byte of int)
```

Explicit conversion: cast

- The programmer may specify a type conversion explicitly: *cast* `(type) expression`
- Example of explicit conversion in expressions

```
if (3/2 == (int)(3/2.0)) {
        printf("VERO :-)\n");
} else {
        printf("FALSO :-(\n");
}
```

Explicit conversion: cast

- The programmer may specify a type conversion explicitly: *cast*

  `(type) expression`
- Example of explicit conversion in expressions

```
if (3/2 == (int)(3/2.0)) {
        printf("VERO :-)\n");
} else {
        printf("FALSO :-(\n");
}
```

- It is printed `VERO :-)`
- The content of variable may be **altered** after a (explicit/implicit) type conversion

Example: type conversion

- *test-celsius.c*