Imperative programming

Basetypes

Tamás Kozsik et al

Eötvös Loránd University

September 20, 2022

Table of contents

Role of types

- Protection against programming errors
- Expression of programmers' thoughts
- Help to form abstractions
- Help efficient code generation

- They express how to interpret a bit sequence
- They define what value a variable can take
- They determine what operations can be performed

Different type object in memory

...	C0	07	00	00	00	00	00	00	00	6B	00	00	...

int book = 1984;

`char k = 'k';`

...	07	B1	00	00	07	00	00	00	14	00	00	00	...

`date[0]` `date[1]` `date[2]`

int date[3] = {1969,7,20};

24102388 (0x16FC5F4)

24102396 (0x16FC5FC)

...	C0	07	00	00	00	00	00	00	F4	C5	6F	01	...

int book = 1984;

int *p = &book;

p == 24102388
*p == 1984

Type checking

- Are variables and function used according to their types
- Non-type-correct programs are pointless

Static and dynamic type system

- C compiler checks type correctness in *compile time*
- Some languages check type correctness in *runtime*

Strongly and weakly typed languages

- Weakly typed languages convert types of values automatically if needed
  - Seems comfortable for the first sight
  - Easy to make mistake
- In C the rules are relatively strict

Integers

- Decimal form: `42`
- Octal and hexadecimal form: `0123`, `0xCAFE`
- Unsigned representation: `34u`
- Long representation: `9999999999L`
- Combined: `0xFEEL`

Floating point numbers

- Trivial: `3.141593` `5.` `.3`
- With exponent: `31415.93E-4`
- Long representation: `3.14159265358979L`
- Combined: `31415.9265358979E-4L`

Character and text

- Characters: `'a'`, `'9'`, `'$'`
- Strings: `"a"`, `"appletree"`, `"1984"`
- Escape-sequences: `'\n'`, `'\t'`, `'\r'`, `"\n"`, `"\r\n"`
- Multi-part string: `"apple" "tree"`
- Multi-line string:

```
"apple\
tree"
```

Characters

- An integer in fact!
- One-byte character code, e.g. ASCII

```
char c = 'A';      /* ASCII: 65 */
```

- Escape sequences
- Special characters: `\n`, `\r`, `\f`, `\t`, `\v`, `\b`, `\a`, `\\`, `\,`, `\"`, `\?`
- Octal code: `\0` – `\377`
- Hexadecimal code, e.g. `\x41`

Logical type?

ANSI C: Doesn't exist

false: 0, true: everything else (but mainly 1)

```
int right = 3 < 5;
int wrong = 3 > 5;
printf("%d %d\n", right, wrong);
```

C99-től

```
_Bool v = 3 < 5;
int one = (_Bool) 0.5;
int zero = (int) 0.5;
```

```
#include <stdbool.h>
...
bool v = true;
```

Complex numbers

Real and imaginary part, e.g.: $3.14 + 2.72i$ (where $i^2 = -1$)

C99

```
float _Complex fc;
double _Complex dc;
long double _Complex ldc;
```

Complex numbers from C99

```
#include <complex.h>
...
double complex dc = 3.14 + 2 * I;
```

Operators

- Arithmetic
- Assignment
- Increment/decrement
- Relational
- Logical
- Conditional
- Bitwise
- `sizeof`
- Cast

Arithmetic operators

```
+ operand
- operand
left + right
left - right
left * right
left / right
left % right
```

“Real” division

```
5.0 / 2.0 == 2.5

(C, Python2)
5 / 2 == 2

(Python3)
5 / 2 == 2.5
5 // 2 == 2
```

Integer division and remainder

- Integer division: rounding to zero
- Sign of remainder: same as sign of `left`

```
(left / right) * right + (left % right) == left

10 / (-3) == -3      10 % (-3) == +1
(-10) / 3 == -3      (-10) % 3 == -1
```

Power

```
#include <math.h>

pow(5.1, 2.1);
```

Assignment operators

```
n = 3
n += 3        n = (n + 3)
n -= 3        n = (n - 3)
n *= 3        n = (n * 3)
n /= 3        n = (n / 3)
n %= 3        n = (n % 3)
```

Increment/decrement operators

Side effect

```
c++;     c += 1;    c = (c + 1);
++c;     c += 1;    c = (c + 1);

c--;     c -= 1;    c = (c - 1);
--c;     c -= 1;    c = (c - 1);
```

Value

```
c++      c
++c      c + 1

c--      c
--c      c - 1
```

Relational operators

```
left == right
left != right
left <= right
left >= right
left <  right
left >  right
```

What does this do?

```
if (x = 5)
{
  printf("Hello World!");
}
```

What does this do?

```c
if (x = 5)
{
  printf("Hello World!");
}
```

```c
3 < x < 7
```

What does this do?

```
if (x = 5)
{
  printf("Hello World!");
}
```

```
3 < x < 7
(3 < x) < 7
```

Bitwise operations

```
int two = 2;                  // 00000010
int sixteen = 2 << 3;         // 00010000
int one = 2 >> 1;             // 00000001
int zero = 2 >> 2;            // 00000000

int three = two | one;        // 00000011
int thirteen = 13;            // 00001101
int seven = 7;                // 00000111
int five = 13 & 7;            // 00000101
int nine = 9;                 // 00001001
int twelve = 9 ^ five;        // 00001100
int minusOne = ~zero;         // 11111111
```

Logical operations

```
left && right
left || right
! operand
```

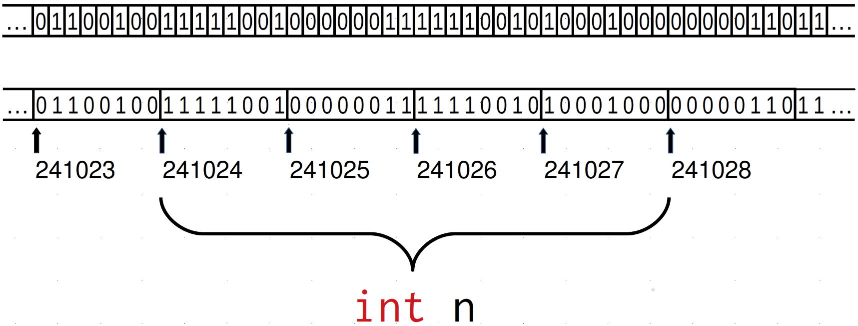
Conditional operator

```
condition ? left : right
```

For example:

```c
int x = 1 < 2 ? 10 : 20;
printf("%d\n", x);    // 10
int y = 2 < 1 ? 10 : 20;
printf("%d\n", y);    // 20
```

Size of objects



sizeof

- Size of type or data in memory
- Can be evaluated in compile-time

```
sizeof(char) == 1
sizeof(int)
sizeof(42)
sizeof(42L)

char str[7];
sizeof(str) == 7
```

- `size_t`
- `printf("\lu", sizeof(42L))`

Conversion between types

```
float five = 5;                               /* 5.0 (automatic) */
float how_much = 5 / 2;                                  /* 2.0 */
float two_and_half = 5. / 2                              /* 2.5 */
```

Conversion between types

```
float five = 5;                            /* 5.0 (automatic) */
float how_much = 5 / 2;                                /* 2.0 */
float two_and_half = 5. / 2                            /* 2.5 */

float pi = 3.141592;
int three = (int) pi;                                    /* 3 */
```

Conversion between types

```c
float five = 5;                            /* 5.0 (automatic) */
float how_much = 5 / 2;                                /* 2.0 */
float two_and_half = 5. / 2                            /* 2.5 */

float pi = 3.141592;
int three = (int) pi;                                    /* 3 */

float one = three / 2;                                 /* 1.0 */
float one_and_half = ((float) three) / 2;              /* 1.5 */
```

Representing numbers as a sequence of bits in memory

- Integer – an interval in $\mathbb{Z}$
  - Unsigned
  - Signed
- Floating point numbers (float) $\subsetneq \mathbb{Q}$

Size of integer types

- `short`: at least 16 bits
- `int`: at least 16 bits
- `long`: at least 32 bits
- `long long`: at least 64bits (C99)

```
sizeof(short) <= sizeof(int) <= sizeof(long)
```

Unsigned numbers

Four bits

$1011 = 2^3 + 2^1 + 2^0$

n bits

$$b_{n-1} \dots b_2 b_1 b_0 = \sum_{i=0}^{n-1} b_i 2^i$$

in C

```
unsigned int big = 0xFFFFFFFF;
if (big > 0) { printf("It's big!"); }
```

ELTE
EÖTVÖS LORÁND
TUDOMÁNYEGYETEM

Signed numbers ("Two's complement")

First bit: sign, other bits: local values

Four bits

0000	0			
0001	1	1111	-1	
0010	2	1110	-2	
0011	3	1101	-3	0011
0100	4	1100	-4	+1101
0101	5	1011	-5	-----
0110	6	1010	-6	10000
0111	7	1001	-7	
		1000	-8	

in C

```
signed int small = 0xFFFFFFFF;
if (small < 0) { printf("It's small"); }
```

Signed and unsigned char

```
  signed char a = '\xFF';      /* a < 0 */
unsigned char b = '\xFF';      /* b > 0 */
         char c = '\xFF';      /* platform-dependent */
```

Wide representation

```
wchar_t w = L'é';
```

- Implementation-defined!
  - Windows: UTF-16
  - Unix: usually UTF-32
- From C99 "Unicode", e.g. `\uC0A1` and `\U00ABCDEF`

Signed arithmetics

- Asymmetry: one more negative value
- Unnatural
  - Sum of two big integers can be negative
  - Negation of a negative can be negative
- Example: average of two numbers?

$$\frac{a+b}{2} \qquad vs \qquad \frac{a}{2}+\frac{b}{2}$$

Floating point numbers

$$1423.3 = 1.4233 \cdot 10^{3}$$
$$13.233 = 1.4233 \cdot 10^{1}$$
$$0.14233 = 1.4233 \cdot 10^{-1}$$

Size of floating point numbers

- `float`
- `double`
- `long double`

```
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

Binary representation

$(-1)^s \cdot m \cdot 2^e$
(s: sign, m: mantissa, e: exponent)

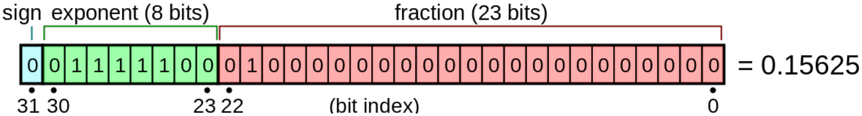
Represented on fixed bits

- Sign
- Exponent
- Valuable digits

IEEE 754

- Binary system
- In most computer systems
- Different size numbers
  - single (32 bits: 1 + 23 + 8)
  - double (64 bits: 1 + 52 + 11)
  - extended (80 bits: 1 + 64 + 15)
  - quadruple (128 bits: 1 + 112 + 15)
- Mantissa between 1 and 2 (e.g. 1.01101000000000000000000)
- Implicit first bit

32 bit example



- Sign: 0 (non-negative)
- Characteristics: 0111110, that is 124
- Exponent: Characteristics - 127 = -3
- Mantissa: 0.01000...0, that is 1.25

Meaning: $(-1)^0 \cdot 1.25 \cdot 2^{-3} = 1.25/8$

Properties of floating point numbers

- Wide range
- Very big and very small numbers
- Not even distribution
- Over and underflow
- Positive and negative zeros
- Infinities
- NaN
- Denormalized numbers

Floating point arithmetics

```
2.0 == 1.1 + 0.9
2.0 - 1.1 != 0.9
2.0 - 0.9 == 1.1
```

Money shouldn't be represented with floating point numbers!