Computer Programming

# Review

Marius Minea

marius@cs.upt.ro

12 January 2016

# Bit operators

Every value is composed of bits.
Bit operators only apply to ints (`char`, `unsigned`, `uint32_t`, etc.)

Bit operators work on *all* bits of the integer.
There is no value of e.g., 5 bits. But we can make all others zero.

Logical OR | *puts together* parts (assuming other bits are zero)

```
int32_t date = sec | (min << 6) | (hr << 12) | ...;
```

To *extract* a part: `hr = (date >> 12) & 0x1F;`
right-shift to low-order bits, AND with mask of 11..1 (no. of bits):
or `hr = (date << 15) >> 27;`
shift left to high-order bits, right to low-order bits (makes rest 0)

Use *fixed-width* integers (`stdint.h`) if exact width matters

Integer encoding (big-endian/little-endian) depends on processor!
little-endian = least significant byte first

*Avoid* right-shifting a signed number
if negative, may insert bits of 1 at left (implementation-defined)
usually, we want to insert zeroes ⇒ cast to *unsigned*

# Type casts

(*forced-type*) *expression*

For *values*: if conversion makes sense

```
double exact_div = (double)1/3; //floating division
```

(int)3.14    (integer part)

For *pointers*

to add number of bytes, not number of *objects/elements*

```
int a[5], *p = a + 3; //p points 3 integers after a
char *s = (char *)a + 2; //s points two bytes after a
```

to view memory according to representation of another type:

```
float f = 5; uint32_t f_bits = *(uint32_t *)&f;
```

(put bit pattern of `f` into an int for further processing)

# Parameter passing

In C, parameters are passed *by value*.
Arguments are *expressions* that are *evaluated*.

Cannot pass a *variable* to a function: *value* of variable is passed.

Function does not know value came from a variable
⇒ *cannot change* variable. *NO EXCEPTIONS!*
(even if in function, formal parameter is assigned/changed).

Pointers are *no exception*: *value* of pointer is passed.

```
void upcase(char *s) { for (; *s = toupper(*s); ++s); }
int main(void) {
  char t[] = "hello";
  upcase(t);   // changes contents, not address t
  return 0;
}
```

# Array and pointer parameters

Arrays cannot be passed to functions – only *address* of array

Compiler converts `void f(int a[])` to `void f(int *a)`

Address carries *no size information* ⇒ *must pass array size*
as additional parameter (so function knows it).
Ordinary arrays have no terminator value (only strings have 0)

`sizeof` is NOT `strlen`
`sizeof` is a *compile-time operator* (size of type)
`strlen` traverses the string at run-time until 0

`sizeof` on array parameter *cannot give size of array*!

```
int a[10], n = sizeof(a); //n is 10 * sizeof(int)
```

```
void f(int a[]) { int n=sizeof(a); } // n is sizeof(int *)
```

because the above is actually `void f(int *a) ...`

# Size in the type: pointers to array

`v` and `&v` have distinct values (second is variable's address)

Exception: the *address* of an array
`int a[10];`    a and &a have *same value*
but type of a is `int *`,    type of &a is `int (*)[10]`
(address of an array of 10 ints)

If we *know* function alwas gets an array of fixed size, can state this in the type: function takes *address* of an array of that type

```
int int24(char (*b)[3])
  { return (*b)[0] | (*b)[1] << 8 | (*b)[2] << 16; }
int main(void) {
  char b3[] = { 0x3, 0x2, 0x1 }; // 256*256 + 2*256 + 3
  char t5[] = "test"; // compiler deduces: 5 bytes
  printf("%d\n", int24(&b3));
  printf("%d\n", int24(&t5));  // compiler warning
  return 0;
}
```

```
expected char (*)[3] but argument is of type char (*)[5]
```

# Void pointers

A `void *p` is a pointer to something unspecified
cannot dereference ~~`*p`~~ since we don't know result type
cannot do arithmetic ~~`p + 3`~~ (means: 3 objects further)
since we don't know `sizeof` for what it points to
thus cannot index ~~`p[i]`~~ since this means ~~`*(p + i)`~~

But `void *` is *compatible* with any pointer type
⇒ used for functions that directly manipulate memory (malloc, memcpy)
⇒ for function types that must accept anything (qsort comparison)
⇒ for pointers to abstract types