Lecture 03 - C language basics

(last updated: 2023-09-19)

The C programming language greatly influenced languages that came after it, including C++, Java, Go, Javascript, Python, and Rust. A lot of C language features may seem familiar or even obvious for those who have programmed in other languages.

This lecture note will assume you already know one of those aforementioned languages (primarily Java) and focus on:

- features where C may differ from those languages
- features not as frequently used in those languages
- other C quirks that you should be aware of

Integral data types in C

C types tell you precisely how much memory a variable of that type occupies. C has the following primitive data types, whose sizes are such that:

```
char <= short <= int <= long <= long long
```

The C standard does not specify the byte sizes of these types! On most systems:

```
  - char:       1 byte
  - short:      2 bytes
  - int:        4 bytes
  - long:       8 bytes on 64-bit UNIX systems (such as CLAC);
                4 bytes on most 32-bit systems and 64-bit Windows
  - long long:  8 bytes
```

When you define a variable of a certain type:

```
int x;
```

You are saying, "the variable x takes up 4 bytes."

When you declare a variable of a certain type:

```
extern int x;
```

You are saying, "the variable x is defined elsewhere with 4 bytes."

sizeof

Use the sizeof operator to obtain the size of a type:

```
printf("size of long: %lu\n", sizeof(long));
```

Or the size of the type of an expression:

```
long i;

printf("size of i: %lu\n", sizeof(i));

printf("size of some arithmetic: %lu\n", sizeof(i + 2));
```

Some quirks about sizeof:

- sizeof isn't a function---it just looks like one! sizeof is an operator.
- The sizeof of any type is known at compile time.
- sizeof does not evaluate its argument:

```
int foo(void) {
    printf("foo() was called\n"); // never printed
}

...

printf("size of foo's return type: %lu\n", sizeof(foo()));
```

- You can write sizeof without parentheses when the argument is an expression:

```
long i;

printf("size of i: %lu\n", sizeof i); // also ok, but bad style
```

Binary, octal, decimal, hexadecimal

We normally read, write, and think about numbers in base 10 (decimal), but machines compute and store numbers in base 2 (binary). So in C code, when we write "6" or "11", we are actually expressing the bit values 110 and 1011.

C also lets us write numbers in octal and hexadecimal notation, whose digits neatly map to binary digits. Octal notation begins with '0', whereas hexadecimal begins with '0x'.

when we write...			...the machine sees
octal	decimal	hex	binary
00	0	0x0	0000
01	1	0x1	0001
02	2	0x2	0010
03	3	0x3	0011
04	4	0x4	0100
05	5	0x5	0101
06	6	0x6	0110
07	7	0x7	0111
010	8	0x8	1000
011	9	0x9	1001
012	10	0xa	1010
013	11	0xb	1011
014	12	0xc	1100
015	13	0xd	1101
016	14	0xe	1110
017	15	0xf	1111

C writing tip: use decimal when expressing numerical values (e.g., for arithmetic); use hexadecimal for expressing bit values; octal is rarely used.

Q. How many hex digits do we need to represent all possible char values?

A. 2 hex digits. Reasoning:

A char is 1 byte = 8 bits, so there are 2^8 = 256 possible char values.
A single hex digit can represent 16 possible values (0-15 inclusive);
two hex digits can represent 16 * 16 = 256 possible values.

Q. How many hex digits do we need to represent all possible int values?

A. 8 hex digits. Reasoning:

An int is 4 bytes = 32 bits, so there are 2^32 = lots of possible values.
We previously saw that 2 digits were sufficient for 1 byte (i.e., a char),
so 4 * 2 = 8 digits for 4 bytes.

Signed vs. unsigned integers

Each integral type can be used in two flavors: signed or unsigned. For example, for the 4-byte 'int' type:

```
int x;          // possible values go from -2^31 to +2^31 - 1

unsigned int x; // possible values go from 0 to +2^32 - 1
```

That is, the unsigned flavor of integral types let you double the range of positive values at the expense of negative values.

The binary representation of positive values is straightforward. How, then, do we represent negative numbers in binary?

We use two's complement encoding to represent n-bit signed numbers as follows:

- Most significant bit (MSB) represents sign; 0 is positive, 1 is negative
- When the MSB is 1, subtract 2^(n-1) from the lower n-1 bits to obtain the encoded value; in other words, assign negative weight to the MSB
- There is an asymmetry: there is one more negative number
- To negate a number, flip all of its bits and binary-add 1; equivalent to binary-subtracting the number from 2^n

Some important numbers at the boundaries:

Hex	description	decimal
0x00....00	zero	0
0xFF....FF	negative one	-1
0x7F....FF	the largest positive number	2^(n-1) - 1
0x80....00	the lowest negative number	-2^(n-1)

In C, conversions between signed and unsigned values preserve bit patterns:

```
int main()
{
    char c = -1;
    unsigned char uc = c; // force a conversion from negative to unsigned
    int i = uc; // i will keep the value of uc because int is much bigger
    printf("%d\n", i); // prints 255
}
```

Integer literals

Here are some examples of integer variable declarations to show different ways of writing integer literals:

```
int i = 0xffffffff; // i is -1

unsigned int i = 0xffffffff; // i is 4294967295

int i = 'A'; // i is 65, the ASCII code for the character A

int i = '\n'; // i is 10, the ASCII code for the newline character

int i = '\11'; // i is 9 because backslash followed by up to 3 octal digits
               // in single quotes evaluates to an octal number

int i = '\0'; // so this is 0
int i =   0 ; // so is this (obviously)
int i =  '0'; // but this is 48, the ASCII code for the character 0
```

Other C types

If you need to ensure byte sizes, #include <stdint.h> to use intN_t (signed) and uintN_t (unsigned), defined in the C99 standard, e.g.,:

- int8_t: 8-bit (1-byte) signed integer
- uint8_t: 8-bit (1-byte) unsigned integer
- uint32_t: 32-bit (4-byte) unsigned integer
- int64_t: 64-bit (8-byte) signed integer

Floating point numbers: float is 4 bytes and double is 8 bytes.

```
float  x = 123.4f; // 'f' suffix for floating point literals of type float
double y = 123.4;
```

Arrays and pointers are central to C programming. For example, there is no String type in C. Strings are represented using arrays of char ending with 0. We will cover them soon!

There is no Boolean type in C, only non-zero and zero integer values representing true and false.

You can use typedef to create an alias for a type, e.g.,:

```
typedef int bool;
```

Expressions vs statements

Expressions are things in the C language that always evaluate to a value; statements do not. For example, the following are expressions:

```
42              // evaluates to 42
x               // evaluates to the contents of variable x
```

```
    1 + 6           // evaluates to 7
    foo()           // evaluates to the return value of calling foo()
    a == b          // evaluates to 1 if a and b are equal, 0 otherwise
    b ? 42 : 66     // evaluates to 42 if b is non-zero; 66 otherwise
```

Meanwhile, statements may contain expressions, but do not evalaute to a value themselves. For example, these are all statements:

```
    return 1 + 6;
    break;
    while (x) { f(); }
    if (b) return 42; else return 66;
```

One way to think about this distinction is the fact that you can only assign values; so this makes sense:

```
    x = 1 + 6;      // assign the value 5 to the variable
```

But this does not (and is a syntax error):

```
    x = return 1 + 6;   // ??
```

Bitwise vs logical operators

Operators are built-in operations we can perform on expressions; most are binary, i.e., take two arguments. For example, +, *, and - are examples of arithmetic operators, while >, ==, and >= are comparison operators. Some operators are unary, i.e., only take one argument: for example, sizeof (which we've seen before) and - (as in when you write -3).

C has two sets of operators that look similar and behave similarly: logical operators and bitwise operators. They are:

```
    &&    logical AND
    ||    logical OR
    !     logical NOT
```

and:

```
    &     bitwise AND
    |     bitwise OR
    ~     bitwise NOT
    ^     bitwise XOR
```

(Note that there is no logical XOR.)

The difference is that the logical operators only consider whether their operands are zero or non-zero; bitwise operators operate on a per-bit level.

C also includes two bit-shift operators, which shift the bits in their left operand by the amount specified in the right operand:

- <<: bitwise shift left
- >>: bitwise shift right (sign-extending)

For example, 5 << 2 shifts the bits in 5 to the left by 2 positions, producing the value 20:

```
    00000000 00000000 00000000 00000101
                                   / /  << 2
    00000000 00000000 00000000 00010100
```

Bits "shifted out" the end of a number are discarded:

```
    00000000 00000000 00000000 00000101
                                     \  >> 2
    00000000 00000000 00000000 00000001
```

When left-shifting, bits "shifted in" from the right are always 0; when right-shifting, bits shifted in from the left are 0 if the MSB is 0, and 1 if the MSB is 1 (preserving the sign-bit; this is why we call it sign-extension). For example:

```
    10000000 00000000 00000000 00000101
     \                               \  >> 2
    11100000 00000000 00000000 00000001
```

Short-circuit evaluation

An important characteristic of logical operators is that they "short-circuit" their evaluation process in the middle if the remaining part of the expression will not change the result. Consider the following example:

```
    // assume x == 0 at this point

    if (x > 0 && f(x)) {

        // code continues
```

Since x is equal to 0, and the value of whole expression will be false no matter what f(x) returns, the evaluation of the logical AND expression will terminate after it was determined that x > 0 is false. This means that the function f() will not even be called. This can matter if f() produces side-effects -- like printing something for example.

The same applies to the logical OR operator -- if the left-hand side is true, the right-hand side is not evaluated.

Assignments as expressions

One peculiar thing about C is that an assignment is an expression. This syntactic feature allows us to write this:

```
    x = y = 3;
```

Which should be read as:

```
    x = (y = 3);
```

The value of an assignment expression is the right hand side of the assignment, i.e., what is being assigned; so when we write x = y = 3, both x and y are assigned the value 3.

This quirk also allows us to write:

```
if (err = f())
    // do something...
```

instead of:

```
err = f();

if (err)
    // do something...
```

However, since = is often mistaken for ==, we conventionally add additional parentheses to make our intent clear:

```
if ((err = f()))
    // do something...
```

Compound assignment and increment/decrement expressions

C also provides shorthand for common assignment patterns:

```
When we write...        C treats that as...

    x += y                  x = x + y
    x *= y                  x = x * y
    x &= y                  x = x & y
    x <<= y                 x = x << y
```

C also provides prefix and postfix increment and decrement operators:

	Prefix	Postfix
Increment	++x	x++
Decrement	--x	x--

The value of a prefix increment/decrement is the value of x after incrementing/decrementing; the value of postfix increment/decrement is the value of x before incrementing/decrementing:

```
assert(x == 0);
y = x++;
assert(y == 0);
z = ++x;
assert(z == 2);
```

Undefined behavior

C allows you to write things that are nonsensical. For example, consider this:

```
x = x++;
```

What should the value of x be after the assignment?

And what about:

```
x = x++ + ++x;
```

Instead of painstakingly over-specifying what the behavior should be, the C standard simply leaves the behavior "undefined." Undefined behavior means that the behavior of such statements is beyond the scope of well-behaved C programs. For example, the compiled program may assign an arbitrary value to your variables, or even crash or run forever.

The ambiguity of undefined behavior often gives compilers the flexibility to generate more efficient compiled code. Some compilers can warn you about some forms of undefined behavior. For example, GCC will warn you about x = x++ if you use -Wall. However, other instances of undefined behavior cannot be known at compile-time. For example, signed integer overflow is undefined.