Applied mechatronics

Programming refresher and intro to C programming

Sven Gestegård Robertz

sven.robertz@cs.lth.se

Department of Computer Science, Lund University

2020

Outline

Programming environment

- Compared to the introductory Java courses
  - We recommend (and support) writing the host (PC) programs in C rather than Java
    - the embedded (AVR) programs will be written in C
    - using the same language may simplify things
    - console and serial port I/O is more straight-forward in C
  - We provide no programming environment (for host), instead
    1. Edit source code in any text editor
    2. compile the source file(s) from the command line
    3. run the generated executable
  - You are free to use whatever you're familiar with

Compared to Java, C

- is not object-oriented: no classes
- has declarations in header (.h) files and code in .c files
  - separate compilation
  - C files may `include` .h files (similar to Java `import` clauses)
  - Declarations in header files roughly corresponds to the public interface of a class in Java.
- is not a safe language (no runtime safety net)
  - pointers instead of references
  - be careful with array indices and type casts
- has no automatic memory management (GC)
  - Be careful with (avoid) dynamic memory allocation

Hello world

Java:

```
class Hello {
    public static void main(String args[]){
        System.out.println("Hello, world!");
    }
}
```

C:

```
#include <stdio.h>

int main()
{
        fprintf(stdout, "Hello, world!\n");
        return 0;
}
```

To print on stdout. there is a separate function, printf:

```
    printf("Hello, world!");
```

Compiling and running
from the command line

- Edit the source file `hello.c`
- From the (cygwin) command line
  1. go to the directory where the source file is (e.g. src)

     ```
     > cd src
     ```

  2. compile the source code

     ```
     > gcc -Wall -o hello hello.c
     ```

  3. run the generated executable

     ```
     > ./hello
     ```

     or

     ```
     > ./hello.exe
     ```

Set output file name to `hello` *(defaults to* `a.out` *or* `a.exe`*)*
turn on compiler warnings

Paths in Cygwin and Windows

- The cygwin `/home` directory is in the windows directory `C:\WP\cygwin\home`
- The windows drives are mounted under `/cygdrive`.
  E.g., `C:` is accessible in cygwin at `/cygdrive/c`
- NB! Cygwin and Windows use different path separators

Programming recap, C examples

- C syntax similar to Java
- Control flow
  - alternatives: **`if`** statments
  - repetition: **`for`** and **`while`** statements
- Boolean expressions
- Array operations
- Terminal I/O (console, serial port)

if statements

```
if(condition) {
  then_statements...
} else {
  else_statements...
}
```

- **else** branch is optional
- condition is any integer (more on that)

while statements

```
while(condition) {
  statements...
}
```

```
do {
  statements...
} while(condition);
```

Like in Java

for statements

```
for(init; condition; increment) {
  statements...
}
```

Example:

```
int i; // NB! declaration outside of for statement
for(i=0; i < 10; ++i) {
  printf("%d ", i);
}
```

will print `0 1 2 3 4 5 6 7 8 9`

printf???

- the C standard I/O library has functions for formatted output (and input)
- to use: `#include <stdio.h>`
- output: `printf(format, arguments...)` where `format` is a string that may contain *placeholders* for the formatted values of the following arguments (in order of appearance).
- Example:

```
int x = 10;
float y = 12.8;
printf("x = %d, y = %f\n", x, y);
```

More on this later

`printf()` format conversions

character	argument type; convert to
d, i	signed decimal notation
x	unsigned hexadecimal notation (w/o leading 0x)
u	unsigned decimal notation
c	single character (converted to char)
f	double in decimal notation `[-]mmm.dddd`
s	string (`char *`)

- the conversion characters may be preceeded with arguments for minimum width and/or precision. E.g.,
  - `\%4.2f` : min 4 chars wide, 2 decimal positions
  - `\%.8x` : print (at least) 8 digits, padding with leading zeroes
- . . . and much more, consult a language reference

Boolean and bitwise operators

- Boolean operators (evaluate to `true` or `false`)
  - `!` – not
  - `&&` – and
  - `||` – or
- Bitwise operators (evaluate to a number)
  - `~` – not (bitwise complement)
  - `&` – and
  - `|` – or
  - `^` – xor (exclusive or)

Boolean expressions

- Boolean expressions
  - evaluate to true or false
  - same syntax as in Java but
    - there is no boolean type in C
    - instead, any integer (and hence, any type) can be used
    - zero is interpreted as false
    - non-zero is interpreted as true
  - Example: infinite loop

```
while(1) { // corresponds to while(true) in Java
    printf(".");
}
```

Primitive data types

- Similar to Java, but
  - char: typically 8 bits (= 1 byte, by definition)
  - unsigned integer types available
  - The common ones
    - **`char`** **`unsigned char`**
    - **`short`** **`unsigned short`**
    - **`int`** **`unsigned int`**
    - **`long`** **`unsigned long`**
    - **`float`**
    - **`double`**
- Type definitions for special purposes
  - `size_t`, `ssize_t`, `uint8`, `uint16`, `int8`, `int16`

unsigned integer types

E.g, unsigned char, unsigned short, unsigned int

- Avoids sign extension

```
signed   char x = -1;   // bit pattern 0xff
unsigned char y = 255;  // bit pattern 0xff
int ix = x; // ix = -1  ,  bit pattern 0xffffffff
int iy = y; // iy = 255 ,  bit pattern 0x000000ff
```

- Always use unsigned types when you care about a bit pattern, e.g., bit operations on HW registers etc.
- NB! On signed types, >> is *implementation defined*. Typically preserves the sign by shifting in ones (instead of zeroes) if the highest bit is 1 (i.e., a negative value). i
- For **char**, **signed** or **unsigned** must be specified (it is implementation defined if **char** is signed or unsigned), for the other types the default is signed (i.e., **int** is **signed int**).

arrays and strings

- arrays in C are similar to arrays of primitive types in Java, but
  - can be allocated on the stack (or heap, but avoid that)
  - have no length attribute or bounds checking
- a string in C is really a pointer to a the first character of a char array.

Arrays

- A sequence of values of the same type (homogeneous sequence)
- Similar to Java for primitive types
  - but *no safety net* – difference from Java
  - an array does not know its size – the programmer's responsibility
- *Can contain elements of any type*
  - Java arrays *can only contain references* (or primitive types)
- Can be a local variable (Difference from Java)
- Is declared `T a[size];` (Difference from Java)
  - The size must be *a (compile-time) constant*. (C99 has VLAs)

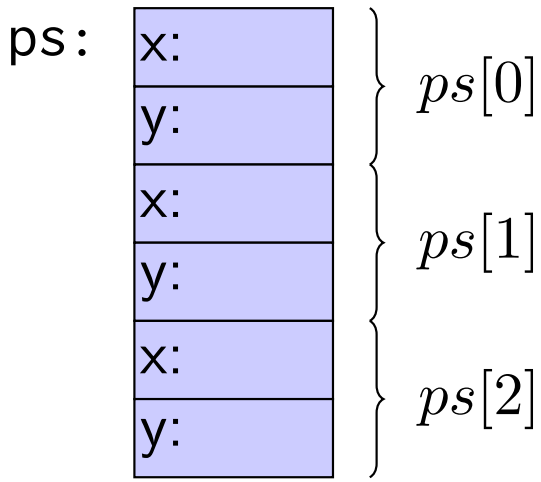
Arrays
Representation in memory

The elements of an array can be of any type

- Java: only primitive types or a reference to an object
- C: an object or a pointer

Example: array of `Point`

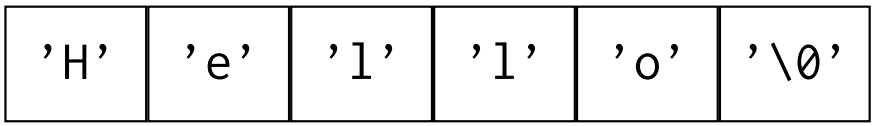
```
struct Point{
    int x;
    int y;
};

struct Point ps[3];
```



Important difference from Java: no fundamental difference between built-in and user defined types.

Data types
C strings

- C strings are `char[]` that are *null terminated*.
  Example: `char s[6] = "Hello";`



NB! A *string literal* is a C-style string (not a `std::string`)
The type of `"Hello"` is `const char[6]`.

arrays and strings, examples

Integer array:

```
int a[6];
int i;

for(i=0; i<6; i++) {
    a[i] = 2 * (i+1);
}

for(i=0; i<6; i++) {
    printf("%d ", a[i]);
}
```

string (**char** array):

```
char s[] = "hej";
int i;

for(i=0; i<3; i++) {
    printf("%c\n", s[i]);
}
s[1] = 'o';
for(i=0; i<3; i++) {
    printf("%c\n", s[i]);
}
```

NB! **char** s[], not **char** *s to make it mutable

C strings are not like in Java

- *null-terminated*: the end of the string is marked by the character null (i.e., the integer value zero)
- no length information (i.e., no safety net)
- copy, concatenate etc using library functions
  - get length by counting the number of chars from the start to the first null character.
- snprintf is a useful string builder
  - like printf, but writes to a char[]
  - make sure destination buffer is big enough

Array operations

- Syntax like in Java
- Example:

```
int sum(int numbers[], int len) {
  int result=0;
  int i;
  for(i=0; i<len; i++) {
          result = result + numbers[i];
  }
  return result;
}
```

- Unlike in Java, C does not do range checking, so *the programmer must keep track of the length* of arrays and avoid accessing past the end

Suggested array exercise

- read a string (**char**[] = **char***) from the terminal
  Example of raw input using `fgets`:

```
char buf[BUFSIZE]; // BUFSIZE-1 chars + terminating null
char* res = fgets(buf, BUFSIZE, stdin);
if(!res) { // an error occured...}
```

- print it
- print it backwards
- encode it in "rövarspråket"
  - for each consonant C, replace C with CoC
  - mekatronik → momekokatotroronononikok

A word on coding style

- Factor your code into small functions
  - One function should do only one thing
  - Rule of thumb: a function should
    - be at most 24 lines long,
    - have at most three block levels, and
    - have at most five local variables.
    - If not, split into smaller functions
- Only use stack allocation
  - Allocate buffers etc. in the main function
- I.e., (rule of thumb):
  - The main function should only consist of variable declarations and (a few) function calls

A typical main function

- Allocate storage
- Do initialization
- Call functions that do the main work
- Cleanup (deallocate/release resources)

```
#include "myproj.h"
#define BUFSIZE 100

int main()
{
    unsigned char buffer[BUFSIZE];
    FILE *f;
    f = init();
    do_work(f, buffer, BUFSIZE);
    cleanup(f);

    return 0;
}
```

Next lecture

- Pointers and structs
- Number representation
- Bitwise operators