Applied mechatronics

More on C programming

Sven Gestegård Robertz

sven.robertz@cs.lth.se

Department of Computer Science, Lund University

2020

Outline

Pointers

- A pointer is a variable containing the address of a variable
- Syntax
  - in declarations
    - prefix * means "is pointer to"

```
int *x; // x is a pointer-to-int
```

  - in expressions
    - prefix * means "contents of"

```
int i = *x; // i gets the value x points to
```

    - prefix & means "address of"

```
int *y = &i;  // y points to the variable i
```

Function calls

- *call-by-value*: the value of the argument is copied

```
void f(int x) {
  printf("x = %d\n",x);
  x += 10;
  printf("x = %d\n",x);
}
```

  - x is an **int** variable: a *copy of* the argument
  - The change to x does not affect the value in the caller

- *call by reference*: a pointer to a variable is passed

```
void f(int *x) {
  printf("x = %d\n",*x);
  *x += 10;
  printf("x = %d\n",*x);
}
```

  - x is a *pointer to* an **int** variable
  - The change to *x changes the value in the caller

Structures (structs)

- Structured data, like Java objects without methods
- Useful for storing a set of variables that belong together
- Example: 3D point

```
struct 3dpoint {
    int x;
    int y;
    int z;
};
```

- Suggested idiom:
  - Allocate structs on the stack, typically in main()
    - NB! difference from Java: structs can be local variables and passed by value
  - Pass a pointer to the struct to other functions (call by reference)

Struct example

```
struct 3dpoint {
    int x;
    int y;
    int z;
};

void myfunction(struct 3dpoint *p)
{
    int px = p->x;
    int py = p->y;
    int pz = p->z;

    printf("%u,%u, %u\n",px,py,pz);
}
```

```
 int main()
{
    struct 3dpoint mypoint;

    mypoint.x=10;
    mypoint.y=20;
    mypoint.z=30;

    myfunction(&mypoint);
}
```

Structs, summary

- declaration and stack allocation

```
struct 3dpoint mypoint;
```

- struct member access (when `mypoint` is a *variable*)

```
mypoint.x
```

- function with call-by-reference (pointer) parameter

```
void myfunction(struct 3dpoint *p);
```

- struct member access *through pointer*

```
p->x
```

- `&`: addressof – get a pointer to a variable

```
&mypoint;
```

Structures (structs)
suggested programming idiom

- Allocate on the stack, typically in main()
  - declare in a scope with longer lifetime than all uses
  - returning a pointer to a local variable gives *undefined behaviour*
- Pass a pointer to the struct (call by reference)
  - Use the address of-operator (&)
  - Note how to access fields

```
myStruct.x or myPointer->x
```

- Call by reference can also be used for primitive type parameters to allow a function to change its arguments (but don't overuse it)

Representation of numbers
Positional number systems

- The *decimal* system: multiples of 10. ("Base 10")
  Example : $1502_{10} = 1 \cdot 10^3 + 5 \cdot 10^2 + 0 \cdot 10^1 + 2 \cdot 10^0$

- base 2: *binary* digits: $\{0, 1\}$
  Example : $= 1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 = 13$

- base 16: *hexadecimal* digits: $\{0 \ldots 9, a \ldots f\}$
  Example : $\texttt{0x73} = 73_{16} = 7 * 16^1 + 3 * 16^0 = 112 + 3 = 115$

Convert to hexadecimal from binary

Convert $0101\,1100_2$ to hex:

$$\begin{aligned}
\text{bits}\,0\ldots3: \quad 1100_2 &= 0\cdot 2^0 + 0\cdot 2^1 + 1\cdot 2^2 + 1\cdot 2^3 = \\
&= 12_{10} = c_{16} \\
\text{bits}\,4\ldots7: \quad 0101_2 &= 1\cdot 2^4 + 0\cdot 2^5 + 1\cdot 2^6 + 0\cdot 2^7 = \\
&= 16 + 64 = 5\cdot 16 \;=\; 50_{16}
\end{aligned}$$

$$\text{which gives the result (in hex)}: \quad 50 + c \;=\; 5c \quad (=\; 92_{10})$$

Observation:

- Each hexadecimal digit corresponds to 4 bits
- One *byte* is two 4 bit "nibbles", i.e., hex digits

Hexadecimal to binary

Convert $5a$ (`0x5a`) to binary:

- the value of each *nibble* can be calculated independently
- i.e., you only need to use the values $\{1, 2, 4, 8\}$ for each hexadecimal digit regardless of how big the number is

high nibble: $5 = 4 + 1 = 0101_2$
low nibble: $a = 8 + 2 = 1010_2$

which gives the result: $5a = 0101\,1010$

Representation of numbers

- Binary (base 2) and hexadecimal (base 16) representations are more convenient than decimal for doing bit operations
- In binary, each bit is directly represented
- In hex, each digit corresponds to 4 bits (a “nibble”)
- Converting to/from decimal is tedious; there is no simple way to find the value(s) of a (set of) bit(s)

Constants in C and Java

- Binary
  - digits: `0 1`
  - constants *in gcc and Java 7* are prefixed by `0b`
    e.g., `0b0001101011110100`
- Hexadecimal
  - digits: `0 1 2 3 4 5 6 7 8 9 a b c d e f`
  - constants in C and Java are prefixed by `0x`
    e.g., `0x1af4`

Hexadecimal notation is much more readable

Bitwise operators

- Bitwise operators evaluate to a number
- Works on the binary representation of the operand(s)
- Example: bitwise not (complement)
  - ~x - in the binary representation of x, invert each bit

```
unsigned char x,y; // 8 bits long
x = 10;
y = ~x;
```

$$x = 10 = \texttt{0x0a} = 0000\,1010$$
$$y = 1111\,0101 = \texttt{0xf5} = 245$$

Bitwise operators

`~x` - not

- the result is x with each bit inverted

`x & y` - and

- the result has a 1 (one) in each bit position where the bit value of both x **and** y ==1

`x | y` - (inclusive) or

- the result has a 1 (one) in each bit position where the bit value of either x **or** y ==1

`x ^ y` - exclusive or

- the result has a 1 (one) in each bit position where the bit value of **exactly one** of x **or** y ==1

Bitwise operators, examples

```
  ~    1111 0101
-------------------
  =    0000 1010

       1110 0111
  &    0111 1100
-------------------
  =    0110 0100

       0010 1010
  |    1100 0001
-------------------
  =    1110 1011
```

- & is used for *bit masking* and for *clearing* bits
- | is used for *setting* bits
- ^ is used for *toggling* bits

Masking and setting bits

Example:

A register : `unsigned char ctrl_reg;`

A bitmask : `FLAGS` (e.g., `0111 0000 == 0x70`)

Named bits : `ENABLE_X` (e.g., `0001 0000 == 0x10`), `ENABLE_Y` (`0x20`), ...

Clear the bits specified by `FLAGS` and set individual bits.

```
// clear FLAGS using bitmask
ctrl_reg = ctrl_reg & ~FLAGS;
// set individual bits
ctrl_reg = ctrl_reg | ENABLE_X | ENABLE_Y;
```

Can also be written:

```
ctrl_reg &= ~FLAGS;
ctrl_reg |= ENABLE_X | ENABLE_Y;
```

Shift operators

- In addition to the bitwise logical operators, C and Java has operators for shifting the bits in a number
- x << n shifts x n steps to the left
- x >> n shifts x n steps to the right
- Example:
  - `2 << 2 == 8` `(0b0010 << 2 == 0b1000)`
- Shifting can be viewed as multiplication or division by powers of two.

Masking and shifting

- Low level drivers often access hardware *registers* where a single, or a few, bits control something
- Example: motor servo control word (16 bits)

- - - posref[8] velref[4] enable

- Reading and writing posref:

```
unsigned short cr;//temporary for register value
unsigned char pr; //temporary for posRef value

cr = read_ctrl_reg();

pr = (cr >> 5) & 0xff;  //shift and mask
pr = pr + 10;           //new value for posRef
cr = cr & ~(0xff << 5); //clear posRef bits
cr = cr | (pr << 5);    //update poRef bits in register value

write_ctrl_reg(cr);
```

Masking and shifting
Use named constants

- Avoid using integer literals (like `0xff`) in your code
- Use macros or constant variables
- Example: accessing `posref` in control word (16 bits)

- - - posref[8] velref[4] enable

```
#define POSREF_MASK 0xff
#define POSREF_OFFSET 5
#define POSREF_BITS (POSREF_MASK << POSREF_OFFSET)

unsigned short cr;//temporary for register value
unsigned char pr; //temporary for posRef value

cr = read_ctrl_reg();

pr = (cr >> POSREF_OFFSET) & POSREF_MASK; //shift and mask
pr = pr + 10;                             //new value for posRef
cr = cr & ~POSREF_BITS;                   //clear posRef bits
cr = cr | (pr << POSREF_OFFSET);          //new value for register

write_ctrl_reg(cr);
```

C preprocessor macros

- A macro is a fragment of text that has been given a name
  - The C preprocessor replaces every occurrence of a name with its contents
  - Using macros for constants makes it easier (and safer) to change them, compared to writing the same literal at many places
  - Macros can be used to select between different variants (`#if` `#ifdef` `#ifndef` `#else`)
  - Macros can be given a value on the compiler command line (gcc -DMYMACRO=value ...)
  - Macros can have parameters, but it is not always trivial what that means. Useful for simple things.

C preprocessor macros
Example: include guards

- Each declaration must appear exactly once
- A header file may be included several times (e.g., via other header files)
- Solution: *include guards*. Example: in the file `example.h`

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

struct example_data{
  char* buf;
  size_t size;
  int something;
};
int example_func1(struct example_data*);
int example_func2();

#endif
```

C preprocessor macros
Examples: named constant

Source code:

```
#define BUFSIZE 80

char buf[BUFSIZE];
   ...
read(fd, buf, BUFSIZE);
```

After preprocessor:

```
char buf[80];
  ...
read(fd, buf, 80);
```

C preprocessor macros
Examples: debugprint.h

```
#ifdef DEBUG

#include <stdio.h>
#define debug_s(s)   printf("DEBUG: %s\n", s)
#define debug_d(s,d) printf("DEBUG: %s%d\n", s, d)
#define debug_x(s,d) printf("DEBUG: %s%x\n", s, d)

#else

#define debug_s(s)
#define debug_d(s,d)
#define debug_x(s,d)

#endif
}
```

```c
#include "debugprint.h"
#include <stdio.h>
int main()
{
    debug_s("Testing macros");
    int sum = 0;
    int i;
    debug_s("summing multiples of 20");
    for(i=0; i<128; i+=20){
        debug_x("i=0x",i);
        sum += i;
        debug_d("sum=",sum);
        debug_x("sum=0x",sum);
    }
    printf("Sum is: %d\n", sum);
    return 0;
}
```

With DEBUG defined, the code becomes

```
int main()
{
  printf("DEBUG: %s\n", "Testing macros");

  int sum = 0;
  int i;
  printf("DEBUG: %s\n", "summing multiples of 20");
  for(i=0; i<128; i+=20){
    printf("DEBUG: %s%d\n", "i=", i);
    printf("DEBUG: %s%x\n", "i=0x", i);
    sum += i;
    printf("DEBUG: %s%d\n", "sum=", sum);
    printf("DEBUG: %s%x\n", "sum=0x", sum);
  }
  printf("Sum is: %d\n", sum);
  return 0;
}
```

If DEBUG is not defined, the code becomes

```
int main()
{
  ;

  int sum = 0;
  int i;
  ;
  for(i=0; i<128; i+=20){
    ;
    ;
    sum += i;
    ;
    ;
  }
  printf("Sum is: %d\n", sum);
  return 0;
}
```

```c
#ifndef BAR
#define BAR 2
#endif
int main()
{
#if BAR == 1
printf("BAR is one\n");
#elif BAR==3
printf("BAR is three\n");
#else
printf("I don't care about bar if it's %d\n", BAR);
#endif
return 0;
}
```

Compiled with `gcc -o macros macros.c` , running it prints

```
I don't care about bar if it's 2
```

Compiled with `gcc -o macros -DBAR=3 macros.c` , running it prints

```
BAR is three
```