Some observations of the exercise solutions of the students I

- You can make no assumptions about the default initial values of variables. Thus when defining pointers, for example, and you expect them to be NULL after the definition, you must explicitly make them NULL.
- The space for all the local variables is freed when the execution leaves the block where the definitons have been done (for example in functions). Only space reserved with malloc continues to exists and to be available.
- If you want to return several values from a function, you have the following options:
  - Make the return type struct and return the values in the structure.
  - Make the return type struct pointer and return a pointer to the structure containing the return values.

Some observations of the exercise solutions of the students II

- Return a pointer to the memory block where the values are. Then you must know in the calling program how many units you will check from the byte the pointer points to.
- Put the values to global variables.

- If the algorithm is not trivial, think of it before starting to write the code and make a sketch on paper. If you start at once to program, the program will probably be unnecessary complicated or erroneous.
- When you program basic operations for a data structure, think typical uses of that data structure. For example, the list data structure is meant for applications, where you store items one by one into the list and sometimes go through all the items. The list data structure is not meant for applications, where you must search single elements from the list. Thus the operations insert and delete should be as fast as possible, as a matter of fact the time requirement should be constant

Some observations of the exercise solutions of the students III

$\mathcal{O}(1)$. It follows that with lists you should avoid solutions or checkings where you go through all the list elements when executing the basic operations insert and delete.

Bit operations

Timo Karvi

September, 2012

Hexadecimal notation I

- On some occasions base-10 representation is not convenient for input or is confusing for output. For example, if a programmer needs to write a mask value or see the value of a pointer to debug a program, the value should be written as an unsigned integer and often is easiest to work with when written in hexadecimal form.
- The following table shows the specifiers needed with different representations when using scanf and printf.

	int	**long**	**short**	**char**
Decimal input	%u	%lu	%hu	
Decimal output	%u	%lu	%hu	%u
Hex input	%i or %x	%li or %1x	%hi or %hx	
Hex output	%04x or %08x	%081x	%04hx	%02x

Example of various numerical types I

The following example shows different integer types which are use in bit manipulations.

```
#include <stdio.h>
void main(void)
{
  unsigned ui, xi;
  unsigned short sui;
  unsigned long lui;

  printf("\n Please enter an unsigned int: ");
  scanf("%u", &ui);
  printf("      = %u in decimal and = %x in hex. \n",
                ui,ui);
  printf("\n Please enter an unsigned int in hex: ");
```

Example of various numerical types II

```
  scanf("%x",&xi);
  printf("       = %u in decimal and = %x in hex. \n",
                  xi,xi);

  printf(" short and long unsigned ints: ");
  scanf("%hu%lu", &sui, &lui);
  printf("       short in hu = %hu in hx = %hx\n",
                     sui, sui);
  printf("       long in lu = %lu in lx = %lx\n",
                    lui, lui);

  printf("  Error: short unsigned in hi format = %hi\n",sui);
  printf("  Error: long unsigned in li format = %li\n",lui);
}
```

Example of various numerical types III

- If first is given unsigned int 123, it is printed in decimal 123 and 7b in hex.
  If then an unsigned int is given in hex as 0xa1, it is printed in decimal 161 and in hex a1.
- If we gave instead a negative -132 in the first phase, it is accepted and the output in decimal would be 4294967164 and in hex ffffff7c. The output in base 10 looks like a garbage but from the hexadecimal output we see that the value has a 1 in the high-order position and, if interpreted as a negative signed integer, it would be a relatively small number.
- In the second group of print commands, if we give short and long unsigned ints 4096 and 65548, the output in hu and hx is 4096 and 1000c, and in lu and in lx 65548 and 1000c.

Example of various numerical types IV

- If in second phase we give 65530 and 4294967200, then we would get in hu and in hx 65530 and fffa, and in lu and in lx 4294967200 and ffffffa0.
- If we enter faulty data in the second phase, 65548 and -65548, it is accepted and the output is in hu 12 and in hx c, and in lu 4294901748, in lx fffefff4. The result seems to be garbage. The output shown for the unsigned short integer is too small by 2^16 because the high-order bit of the value does not fit into the variable and, therefore, is dropped: $12 + 65536 = 65548$.
- The negative number -65548 is entered for the long unsigned variable and converted with a `%lu` format. Even though the input format code was inappropriate for the input value, the hex output shows that the number was converted and stored correctly for a signed number. The garbage answer happened because of a mismatch between the stored value and the output format code. If we printed the same value with a `%i` format, the result would be the number we entered.

Example of various numerical types V

- Note also the library stdint, where there are types of the form `intN_t` or `uintN_t`. In other words, you can explicitly show the number of bits you are going to use. Usually, N is 8, 16, 32 or 64.

Bitwise Logical Operations I

- Complement operator $\sim$:

$$\sim (00000000) \quad == \quad (11111111).$$

- AND operator &:

$$(01010101)\&(10101011) \quad == \quad (00000001).$$

- Inclusive OR $|$:

$$(01010101)|(10101011) \quad == \quad (11111111).$$

- Exclusive OR ˆ :

$$(01010101)\hat{}(10101011) \quad == \quad (11111110).$$

Bitwise Logical Operations II

- Left Shift $<<$:

$$(00000010) << 2 \quad == \quad (00001000).$$

- Right Shift $>>$:

$$(00001000) >> 2 \quad == \quad (00000010).$$

Masks I

- When a program manipulates codes or uses hardware bit switches, it often must isolate one or more bits from the other bits that are stored in the same byte. The process used to do this job is called masking.
- In a masking operation, we use a bit operator and a constant called a mask that contains a bit pattern with a 1 bit corresponding to each position of a bit that must be isolated and a 0 bit in every other position.
- Example: We split a short integer into four portions, A (3 bits), B (5 bits), C (4 bits) and D (4 bits), and put them back together in the scrambled order C, A, D, B. We start by writing out the bit patterns, in binary and hex, for isolating the four portions. The hex version then goes into a `#define` statement. The masks are shown in the table below:

Masks II

Part	Bit pattern	#define, with Hex constant
A	11100000 00000000	`#define A 0xE000`
B	00111111 00000000	`#define B 0x1F00`
C	00000000 11110000	`#define C 0x00F0`
D	00000000 00001111	`#define D 0x000F`

- If the original bit string is in the variable b, we can now form the portions with the and operation:

  ```
  P1 = b && A; P2 = b && B; P3 = b && C; P4 = b && D;
  ```

  Then we can move the portions into the right positions:

  ```
  P1 = P1 >> 3; P2 = P2 >> 5; P3 = P3 << 8; P4 = P4 << 4;
  ```

  and combine:

  ```
  b = P1 | P2 | P3 | P4;
  ```

  Now b has been "encrypted".

Bit Fields in Structures

```
struct {
        unsigned Yorn   : 1;
        unsigned status : 2;
} bitFields;
```

declares a structure variable that contains 2 bit field members: Yorn is 1 bit, status is 2 bits.

bitFields.Yorn = 1; (only 0 and 1 can be used for assignment value)
bitFields.status = 2; (0,1,2,3 can be used for assignment value)

Decoding an Internet address I

- In the next example the program reads a 32-bit Internet address in the form used to store addresses internally and print it in the four-part dotted form that we customarily see.
- Thus input could be fa1254b9 in hexadecimal form and the program prints 250.18.84.185.
- We use a common technique in bit manipulations: mask. Using bitwise operations, mainly &, and a mask we can take part of a bit string. For example, if we have a bit string

$$b = 10101010101010101010101010101010,$$

and a mask

$$m = 11111111000000000000000000000000,$$

Decoding an Internet address II

then

$$m \ \& \ b$$

forms a bit string

$$10101010000000000000000000000000.$$

Thus we have the first eight bits as in b and the rest are zero.

- The mask in the example is needed, if a machine is a 64-bit machine. If all the machines were 32-bit machines, the mask would not be needed.

Decoding an Internet address III

```
#define BYTEMASK 0xffL /* L to make a long integer */
#include <stdio.h>

void main( void )
{
  unsigned long ip_address;
  unsigned f1, f2, f3, f4;

  printf("Please enter an IP address as 8 hex digits: ");

  scanf("%lx", &ip_address);
  printf("You have entered %08lx\n", ip_address);

  f1 = ip_address >> 24 & BYTEMASK;
  f2 = ip_address >> 16 & BYTEMASK;
```

Decoding an Internet address IV

```
  f3 = ip_address >>  8 & BYTEMASK;
  f4 = ip_address       & BYTEMASK;

  printf("The IP address in standard form is: ");
  printf("%i.%i.%i.%i \n\n", f1, f2, f3, f4 );
}
```

Encrypting and Decrypting I

- In modern cryptography, before the actual encryption a plain text is shuffled in many ways in order to reduce regularities. The following program permutes the bytes of a plain text. This is done using a fixed permutation.
- Consider a 16-bit block of a plain text. Divide that block into four unequal-length fields of 3, 5, 4, and 4, respectively, and permute those fields such that
  - the first field becomes the second,
  - the second the fourth,
  - the third the first, and
  - the fourth the the third.

Encrypting and Decrypting II

```
#include <stdio.h>

#define AE 0xE000
#define BE 0x1F00
#define CE 0x00F0
#define DE 0x000F

unsigned short encrypt (unsigned short n);
```

Encrypting and Decrypting III

```
void main ( void )
{
  short in;
  unsigned short crypt;
  printf("\n Enter a short integer to encrypt: ");
  scanf("%hi", &in);
  /* Cast the int to unsigned before calling encrypt. */
  crypt = encrypt( (unsigned short) in);

  printf("\n The input number in base 10 is:   %hi \n"
           " The input number in hexadecimal is: %hx \n\n"
           " The encrypted number in base 10 is:  %hu \n"
   " The encrypted number in base 16 is:  %hx \n\n",
 in, in, crypt, crypt );
}
```

Encrypting and Decrypting IV

```
unsigned short encrypt (unsigned short n )
{
  unsigned short a, b, c, d;

  a = (n & AE) >> 4; /* Isolate bits 0:2, shift to 4:6 */
  b = (n & BE) >> 8; /* Isolate bits 3:7, shift to 11:15 */
  c = (n & CE) << 8; /* Isolate bits 8:11, shift to 0:3 */
  d = (n & DE) << 5; /* Isolate bits 12:15, shift to 7:10 */

  return c | a | d | b;
}
```

A Device Controller I

From a book "Fischer, Eggert, Ross: Applied C: An Introduction and More."

- An artist has a studio with a high sloping ceiling containing skylights. Outside, each skylight is covered with louvers that can be opened fully under normal operation to let the light in or closed to protect the glass or keep heat inside the room at night.
- The louvers are opened and closed by a small computer-controlled motor with two limit switches that sense when the skylight is fully open or fully closed. To open the skylight, one runs the motor in the forward direction until the fully open limit switch is activated. To close the skylight, one runs the motor similarly in the reverse direction. To know the current location of the skylight, one simply examines the state of the limit switches.

A Device Controller II

- The motor is controlled by a box with relays and other circuitry for selecting its direction, turning it on and off, and sensing the state of the limit switches. The controller box has an interface to the computer through a multifunction chip using a technique known as *memory-mapped I/O*. This means that when certain main memory addresses are referenced, bits are written or read from the multifunction chip, rather than real, physical memory.
- In this program, we assume that the multifunction chip interfaces with the computer through two memory addresses: **0xffff7100** refers to an 8-bit data register (DR) and **0xfff7101** refers to an 8-bit data direction register (DDR). Each bit of the data register can be used to send data either from the chip to the program or vice versa. Data flows from chip to program through a bit if the corresponding bit of the DDR is 0 and from program to chip if the corresponding bit is 1.

A Device Controller III

- Certain bits of the DR then are wired directly to the skylight controller box as shown in the specification below:

Bit	In/Out	Purpose	Setting
0	out	Motor direction	0=forward
			1=reverse
1	out	Motor power	0=off
			1=on
2	in	Fully closed louver sensor	0=not fully closed
			1=fully closed
3	in	Fully open louver sensor	0=not fully open
			1=fully open

We start to write declarations for the skylight controller:

We use four bits to communicate with the multifunction chip; two are used by the prigram to receive status information from the chip and two are used to send control instructions to the chip. The leftmost four bits in

A Device Controller IV

the chip registers will not be used in this application. Therefore, the bitfield type declaration used to model a register begins with unnamed field for the four padding bits, followed by named fields for two status bits and two control bits.

```
typedef struct REG_BYTE {
unsigned int                       :4;
unsigned int fully_open            :1;
unsigned int fully_closed          :1;
unsigned int motor_power           :1;
unsigned int motor_direction       :1;
} reg_byte;

typedef volatile reg_byte * device_pointer;
```

A Device Controller V

The key word **volatile** means that something outside the program (in this case, the controller box) may change the value of a register byte at unpredictable times. We supply this information to the C compiler so that its code optimizer does not eliminate any assignments or reads from the location that, otherwise, would appear redundant.

Next follows the definitions of the codes for the multifunction chip:

```
enum power_values     {motor_off = 0, motor_on = 1};
enum direction values {motor_forward = 0, motor_reverse = 1 };

char * position_labels[] = {"fully closed", "partially open",
                            "fully open"};
typedef enum {fully_closed, part_open, fully_open} position;
```

A Device Controller VI

We have defined three enumerated types to give symbolic names to the various switch settings and status codes. An array of strings is defined to allow easy output of the device status.
Two of the enumerations are not within a **typedef**. They are used simply to give names to codes. A series of **#define** commands could be used for this purpose, but **enum** is better because it is shorter and it lets us group the codes into sets of related values.

Next we want a pointer variable DR to point at the address of the data register byte on the multifunction chip. We write its memory-mapped address as a hex literal, cast it to the appropriate pointer type, and store it in DR. The keyword **const** after the type name means that DR always points at this location and can never be changed.
A bitmask is used when the program is started to initialize the data direction register. The rightmost two bits are used to send control

A Device Controller VII

information to the chip from the the program, while the other two will be read by the program to check the chip's status.

```
device_pointer const DR = (device_pointer)0xffff7100;
device_pointer const DDR = (device_pointer)0xffff7101;
const reg_byte DDR_mask = {0,0,1,1};
const reg_byte DR_init = {0,0,0,0};
```

The main program uses the following functions:

```
position skylight_status ( void );
void open_skylight ( void );
void close_skylight ( void );
```

And the main program is:

A Device Controller VIII

```
void main (void )
{
char choice;
char * menu[] = {"O: Open skylight", "C: Close skylight",
                 "R: Report position", "Q: Quit"};

*DDR = DDR_mask;
*DR = DR_init;

for (;;) {
choice = toupper( menu_c( " Select operation:", 4, menu) );
if (choice = 'Q') break;
switch (choice) {
case 'O': open_skylight();        break;
case 'C': close_skylight();       break;
case 'R': /* Report on position */
```

A Device Controller IX

```
        printf( "Louver is %s\n",
         position_labels[skylight_status()] );
            break;
          default: puts( "Incorrect choice, try again.");
      }
    }
    puts( " Skylight controller terminated.\n");
}
```

We use functions

`menu_c()` and `toupper()`

in the main program. The former displays a menu and reads a selection. The latter is used to recognize both lower-case and upper-case letters. We do not show code for these interface functions.

A Device Controller X

```
position skylight_status( void )
{
if (DR->fully_closed) return fully_closed;
else if (DR->fully_open) return fully_open;
     else return part_open;
}
```

A Device Controller XI

```
void open_skylight( void )
{
reg_byte dr = {0, 0, motor_on, motor_forward };

if (DR->fully_open) return;

*DR = dr;

  while (!(DR->fully_open)); /* delay until open */

    dr.motor_power = motor_off;

    *DR = dr;
 }
```

A Device Controller XII

```
void close_skylight( void )
{
 reg_byte dr = { 0, 0, motor_on, motor_reverse };

 if (DR->fully_closed) return;

 *DR = dr;

 while (!(DR->fully_closed)); /* delay until closed */

 dr.motor_power = motor_off;
 *DR = dr;
}
```