Bit Manipulation in C

- C provides six bitwise operators for bit manipulation.
- These operators act on integral operands (char, short, int and long) represented as a string of binary digits.

**Bitwise Operators**

- Logical operators:
  - (unary) bitwise complement: ~
  - bitwise and: &
  - bitwise or: |
  - bitwise exclusive or: ^

- Shift operators:
  - left shift: <<
  - right shift: >>

**Examples of application of bitwise operators on 8 bit representations of integers x and y**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Representation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>00001111</td>
<td>15</td>
</tr>
<tr>
<td>y</td>
<td>11110110</td>
<td>-10</td>
</tr>
<tr>
<td>result = x &amp; y;</td>
<td>00000110</td>
<td>6</td>
</tr>
<tr>
<td>result = x</td>
<td>11111111</td>
<td>-1</td>
</tr>
<tr>
<td>result = x ^ y;</td>
<td>11110001</td>
<td>-7</td>
</tr>
<tr>
<td>result = ~x;</td>
<td>11110000</td>
<td>-16</td>
</tr>
<tr>
<td>result = x &lt;&lt; 2;</td>
<td>00111100</td>
<td>60</td>
</tr>
<tr>
<td>result = x &gt;&gt; 2;</td>
<td>00000011</td>
<td>3</td>
</tr>
</tbody>
</table>

- The bitwise AND operator & is often used to **mask off** some set of bits, e.g.,
  - \( x = x & 017; \) /* N.B. 017 is an octal value */

- The bitwise OR operator | is often used to **turn bits on**, e.g., \( x = x | y; \)

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>00000110</td>
</tr>
<tr>
<td>6</td>
<td>00000111</td>
</tr>
<tr>
<td>7</td>
<td>00000111</td>
</tr>
<tr>
<td>8</td>
<td>00000111</td>
</tr>
<tr>
<td>9</td>
<td>00000111</td>
</tr>
<tr>
<td>10</td>
<td>00000111</td>
</tr>
<tr>
<td>11</td>
<td>00000111</td>
</tr>
<tr>
<td>12</td>
<td>00000111</td>
</tr>
<tr>
<td>13</td>
<td>00000111</td>
</tr>
<tr>
<td>14</td>
<td>00000111</td>
</tr>
<tr>
<td>15</td>
<td>00000111</td>
</tr>
<tr>
<td>16</td>
<td>00000111</td>
</tr>
<tr>
<td>17</td>
<td>00000111</td>
</tr>
<tr>
<td>18</td>
<td>00000111</td>
</tr>
<tr>
<td>19</td>
<td>00000111</td>
</tr>
<tr>
<td>20</td>
<td>00000111</td>
</tr>
<tr>
<td>21</td>
<td>00000111</td>
</tr>
<tr>
<td>22</td>
<td>00000111</td>
</tr>
<tr>
<td>23</td>
<td>00000111</td>
</tr>
<tr>
<td>24</td>
<td>00000111</td>
</tr>
<tr>
<td>25</td>
<td>00000111</td>
</tr>
<tr>
<td>26</td>
<td>00000111</td>
</tr>
<tr>
<td>27</td>
<td>00000111</td>
</tr>
<tr>
<td>28</td>
<td>00000111</td>
</tr>
<tr>
<td>29</td>
<td>00000111</td>
</tr>
<tr>
<td>30</td>
<td>00000111</td>
</tr>
<tr>
<td>31</td>
<td>00000111</td>
</tr>
</tbody>
</table>

The bitwise AND operator & is often used to **mask off** some set of bits, e.g.,
- \( x = x & 017; \) /* N.B. 017 is an octal value */
- Setting bits to zero all but the low-order 4 bits of \( x \).

The bitwise OR operator | is often used to **turn bits on**, e.g., \( x = x | y; \)
- Setting bits to one in \( x \) the bits that are set to one in \( y \).
The bitwise exclusive OR operator ^ sets a one in each bit position where its operands have different bits, and zero where they are the same.
- The bitwise complement operator ~, also called the "one's complement operator", inverts the bit string it is applied to. The 0s become 1s and the 1s become 0s (Toggle)
- The shift operators << and >> perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be positive.

Structuring

- **Structs** are like records and can be used to group data elements of different types, i.e., a struct is (usually) a collection of mixed type items.
- Two ways to define a structure:
  - `struct struct_name {variables};`
  - `typedef struct {variables} Type_name;`

To Declare struct variables

- The **typedef** keyword allows to create name for a type, e.g.

  ```
  typedef unsigned char BYTE;
  BYTE one, bytes[5], *pr;
  ```

- This defines a new data type called 'Employee'. This definition is for the use of C compiler, it does not allocate any space for the component variables.
- This definition can be used to declare variables as follows:

  ```
  struct Employee manager;
  struct Employee worker1;
  EMPLOYEE manager;
  EMPLOYEE worker1;
  ```
Initialising a struct is similar to that used for arrays:

```c
struct employee worker2 = {"John", 'B', 32, 45800.15};
```

Memory allocation for a structure:

```
<table>
<thead>
<tr>
<th>surname</th>
<th>Age</th>
<th>Wage</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>32</td>
<td>45800.15</td>
</tr>
</tbody>
</table>
```

Accessing elements in a struct:

- The elements (members) of a struct are accessed using the **dot notation**:
  ```c
  worker1.firstInit = 'J';
  worker1.age = 22;
  worker1.wage = 30000.00;
  manager.wage = 2 * worker1.wage;
  ```

Bit Field:

- Another method of manipulating bits is to use a **bit field**:
- A bit field is constructed with a structure declaration that labels each field and determines its width;
- For example, to declare a Boolean data type:

```c
#define TRUE 1
#define FALSE 0

struct BOOL {
    unsigned int bit : 1;
};
```

This definition creates a new data type **BOOL** that contains one 1-bit field.

```c
struct BOOL flag;

flag.bit = TRUE;
flag.bit = FALSE;
```

Flag Register:

```c
struct FLAG {
    unsigned int overflow : 1;
    unsigned int sign : 1;
    unsigned int zero : 1;
    unsigned int aux_carry : 1;
    unsigned int parity : 1;
    unsigned int carry : 1;
    unsigned int direc : 1;
    unsigned int interrupt : 1;
};
```
Nested Structures

- A struct can contain or nest another struct;
- Nested structures are often used to construct complex data record;
- For example, we can create a structure called `names` to represent an employee’s name.

```c
struct names {
    char first[4];
    char surname[4];
};
struct Employee {
    struct names name; // nested struct
    int age;
    float wage;
};
```

Again, the elements of a struct can be accessed using the **dot notation**:

```c
struct Employee worker1;
strcpy((worker1.name).last, "Bob");
strcpy((worker1.name).surname, "John");
worker1.age = 43;
```

- The construction is interpreted from left to right.

Pointers to Structures

- Using pointers is easier to manipulate structures than the arrays;
- To overcome the problem of using structures as function arguments in some older C implementations;
- Enable to create dynamic size arrays

```c
/* employee record: demonstrating use of pointer to a structure */
#include <stdio.h>
#define LEN 11 // define a constant LEN
struct names { char last[LEN]; char surname[LEN];}
struct Employee {
    struct names name; // nested structure
    int age;
    float wage;
};
int main() {
    struct Employee workers[2] = {
        { "Ewen", "Villard" },
        23, 34000.00
    },
    { "Rodney", "Swillbelly" },
    41, 58000.00
    };
};
```
struct Employee *him; // a pointer to a structure
him = &workers[0]; // assign address to pointer

printf("workers[0].name.surname is %s\n", workers[0].name.surname);
printf("workers[0].age is %d", workers[0].age);
printf("(him).age is %d", (*him).age);
printf("him->age is %d", him->age);

him++; // point to the next structure

printf("(him->name.surname is %s\n", him->name.surname);

return 0;
}

workers[0].name.surname is Villard
workers[0].age is 23
(*him).age is 23
him->age is 23
him->name.surname is Swillbelly
him->wage is 58000.00

Linked List

- Linked lists contain a set of (mostly)
  similar data structures;
- An item can be added to or removed from
  a linked list and that the length of linked
  lists can be dynamically changed;
- The length of arrays is always fixed and it
  must be defined before use;

int main() {
    struct Employee workers[2] = {
        { NULL, // pointer to nowhere
          {"Ewen", "Villard"},
          23,
          34000.00
        },
        { NULL, // pointer to nowhere
          {"Rodney", "Swillbelly"},
          41,
          58000.00
        }
    };

    struct Employee *him; // a pointer to a structure
    him = &workers[0]; // assign address to pointer
    him->next = him+1; // link to the next record

    printf("(him->next).name.surname is %s\n",
           (*(him->next)).name.surname);

    him++; // point to the next structure
    printf("him->name.surname is %s\n",
           him->name.surname);
}
/* Book record: demonstrating linked list */
#include <stdio.h>
#include <stdlib.h>

struct Book {
    struct Book *next; // pointers to next struct in list
    int barcode;
    float price;
};

int main() {
    struct Book *head = NULL; // head of list
    struct Book *prev, *current;
    int i=1;

do { /* create memory block for Book structure */
    current = (struct Book *) malloc(sizeof(struct Book));
    if(head==NULL) {
        head = current; // first structure
    } else {
        prev->next = current; // subsequent structure
    }
    printf("Enter barcode for %d book: ", i);
    scanf("%d%*c", &current->barcode);
    printf("Enter price for %d book: ", i);
    scanf("%f%*c", &current->price);
    current->next = NULL; // set the next struct to NULL
    prev = current; // update prev pointer
    i++;
} while (i<5);

current = head; // pointer to head of list

while(current!=NULL) {
    printf("Book: %d   Price: %f
",
        current->barcode, current->price);
    /* pointer to subsequent struct */
    current = current->next;
}

return 0;
}