Bit Operations

- Bit or flag
  - Smallest unit of information
  - Can take on the value 1 (`true`) or 0 (`false`)
  - Used to manipulate the bits of integral operands, `char`, `short`, `int`, and `long`, both `signed` and `unsigned`
  - Used to control the machine at the lowest level, specially in pixel-level graphics
- Byte
  - Collection of 8 bits
  - Can be represented as two hexadecimal numbers by using `0xHH` where `H` is a hexadecimal number
- Bit operators
  - Allow a programmer to manipulate individual bits in integer or character data types

Operator	Semantics
`&`	Bitwise `and`
`|`	Bitwise `or`
`^`	Bitwise `xor` (exclusive or)
`~`	Complement
`<<`	Shift left
`>>`	Shift right

  - The `and` operator
    * Compares corresponding bits in the two operands
      · Compare two bits and set the output to 1 if both the input bits are 1; otherwise, set the result to 0
    * Consider the following program

```
int main()
{
    char c1 = 0x45,
         c2 = 0x71;
    printf ( "Result of %x & %x = %x\n", c1, c2, c1 & c2 );
}
```

    * The operators `&` and `&&` are different
    * Using bitwise operator to check if a number is even

```
#define even(x) (((x) & 1) == 0)
```

  - The bitwise inclusive `or` operator
    * Compare two operands and set resultant bit to 1 if either of the corresponding bits is a 1
  - The bitwise exclusive `or` operator
    * Compare two operands and set resultant bit to 1 if either of the corresponding bits is a 1 but not both
  - The `not` operator
    * Also called one's complement, invert, or bit flip
    * Unary operator
    * Changes the corresponding bits to 0 if they are 1, or 1 if they are 0
    * *Does not change the value of a variable if used by itself*, for example `~x` does not change the value in `x`
- Setting and clearing bits
  - Use `&` and `|` operators

`x |= mask;`	Set bits set in `mask`
`x &= ~mask;`	Clear bits set in `mask`
`x &= mask;`	Clear all bits not set in `mask`
`x ^= x;`	Clear all bits in `x`

Bit-shift operators

- The left and right shift operators
  - Used to move the data a specified number of bits
  - Bits shifted out of the left side disappear
  - New bits coming in from the right side are zeros or ones depending on whether the number is positive or negative
  - Example – bitp.c
- Sign extension
  - Shifting left by one bit will fill the right side bits with zeroes
    * Equivalent to multiplication by 2
  - Right shifting positive numbers fills the left bits with zeroes
    * Right shifting by 1 bit is equivalent to divide by 2
  - Right shifting negative numbers is *not* always equivalent to divide by 2
    * Right shifting negative numbers will leave the sign bit unchanged and shift it as well
    * Shifting with sign extension is equivalent to division but sign extension is not guaranteed
  - Example: bits.c
- Operator precedence
  - Precedence of shift operators is lower than addition/subtraction
  - This may cause issues if you replace multiply/divide by shift because precedence of multiply/divide is higher than addition/subtraction
  - Example
    * Original: `x = a + b * 2;`, equivalent to $x = a + 2b$
    * Using shift operator: `x = a + b << 1;` equivalent to $x = 2(a + b)$
    * Should be: `x = a + ( b << 1 );`

Bit fields

- Declaration of bit fields or packed structures
  - Consider a structure with the following information:

`name`	$\leq$ 28 characters
`male`	1 or 0
`married`	1 or 0
`elderly`	1 or 0

  - The structure can be declared as

```
struct person
{
    char name[29];      /* 28 characters + end of string */
    unsigned male : 1,
             married : 1,
             elderly :1;
} people[1000];
```

- `male`, `married`, and `elderly` are bit-fields
  - The 1 following the colon indicates that each of these fields contains only one bit
  - These structure members can only have the values 0 or 1
  - Instead of 1, a greater number of bits may be chosen limited to the number of bits in a single machine word
- An assignment to bit fields can be made as

  `people[i].married = 0;`
- Bit fields occupy little space in the memory
  - In the above example, `male`, `married`, and `elderly` may be bits of a single machine word
  - Bit fields have no address of their own, and so, we cannot use pointers to them
  - `&(person[i].married)` is not a valid expression
- Another example of a bit field or packed structure

```
struct item
{
    unsigned int list:1;        /* item is in the list      */
    unsigned int seen:1;        /* item has been seen       */
    unsigned int number:14;     /* item number              */
                                /* at most 16383 items      */
};
```

- Code to extract data from bit fields is relatively large and slow
- Better for human consumption than bit operators which are complex and error prone

Enumerated types

- User-defined data types that make the code more readable
- Designed for variables that contain only a limited set of values
- Set of integer constants represented by identifiers or tags, known as *enumeration constants*
- Declared using the keyword `enum`
- Values in an enumeration start with 0, unless specified otherwise, and are incremented by 1
- Creating a new type `months`

```
enum months { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

- To number the months from 1 to 12, the enumeration is specified as

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

- The name of the `enum` type (`months` above) is optional and can be omitted

- Identifiers in an enumeration must be unique, and are generally written as upper case letters; they can be any valid C identifiers
- Value of each enumeration constant of an enumeration can be set explicitly in the definition by assigning a value to the identifier
- Multiple members of an enumeration can have the same integer value
  - The following is legal

    ```
    enum months { Jan = 1, January = 1, Feb = 2, February = 2 };
    ```
- The values assigned to `enum` constants must be integral, and can be in any order

  ```
  enum state { stopped = 0, waiting = 1, trace = 5, run = 10 };
  ```
- Using enumeration: enum.c
- *Cautions*
  - *Assigning a value to an enumeration constant after it has been defined is a syntax error*
- All `enum` constants must be unique within a scope; the following is illegal:

  ```
  enum prog { compiled, failed };
  enum result { passed, failed };

  int main()
  {
      return ( 0 );
  }
  ```
- I found the following post on stack overflow very instructive:
  `https://stackoverflow.com/questions/1102542/how-to-define-an-enumerated-type-enum-in-c`