Part 5: Bits and Bytes

Contents [DOCUMENT FINALIZED]

C++ : feature available in C++ but not in C

Bit- and Byte-Level Computations

Manipulating bytes and individual bits in memory can be advantageous in terms of saving space and computation time.

Moreover, when hardware device registers are accessed by reading and writing to memory locations ("memory mapped input/output"), it is often necessary to read status bits or to change individual bits to trigger hardware functions.

Therefore, system programming languages need to support bit-level operations that allow us to set, clear, and read individual bits in memory.

Another aspect of bit-level access is performing many bit operations in parallel, which can speed up certain algorithms considerably.

In what follows, we will discuss bit-level operations in C, as well as bit fields, unions, and byte order issues in some detail beginning with a review of binary arithmetic and integer representations.

Number Systems

- Common base: 10 (decimal system)
- Other bases: 1 (unary), 2 (binary), 3 (ternary), 8 (octal), 16 (hexadecimal) ...
- Binary number system: digits 0 and 1
  (binary digit = "bit")
- Used in today's computers
  (e.g., low voltage = 0, high voltage = 1)
- Byte = sequence of 8 bits
  (contents of a memory cell)
- Integers are represented as sequence of bits
- One byte stores one of $2^8 = 256$ different values

  $00000000_2 = 0_{10}$ $\quad 00000001_2 = 1_{10}$
  $00000010_2 = 2_{10}$ $\quad 00000011_2 = 3_{10}$
  ...
  $11111110_2 = 254_{10}$ $\quad 11111111_2 = 255_{10}$

Binary Arithmetic

- Instead of digits 0..9, we now only have 0,1
- Arithmetic is done analogously. E.g.
  $1_2{+}1_2 = 10_2$, $10_2{+}1_2 = 11_2$, $11_2{+}1_2 = 100_2$, ...
-
```
 87      1010111
+ 9    +    1001
-1--   ----1111-- carry bits
 96      1100000
```
- Weight of bit k (starting with 0 from right to left) is 2^k, rather than 10^k
- Given k bits $x_{k-1}, \ldots, x_0$, the unsigned value this sequence represents is

$$2^{k-1}x_{k-1} + \cdots + 2^1 \cdot x_1 + 2^0 \cdot x_0$$

$$= \sum_{i=0}^{k-1} 2^i x_i$$

Integer Representation

k-bit unsigned number: range $0 \ldots 2^k - 1$

k-bit signed number: range $-2^{k-1} \ldots 2^{k-1} - 1$

Example: $k = 16$ $\rightsquigarrow$ 0..65535 -32768..32767

		unsigned	signed
$0000\ 0000\ 0000\ 0000_2$	=	0_{10}	0_{10}
$0000\ 0000\ 0000\ 0001_2$	=	1_{10}	1_{10}
$0000\ 0000\ 0000\ 0010_2$	=	2_{10}	2_{10}
...			
$0111\ 1111\ 1111\ 1111_2$	=	32767_{10}	32767_{10}
$1000\ 0000\ 0000\ 0000_2$	=	32768_{10}	-32768_{10}
$1000\ 0000\ 0000\ 0001_2$	=	32769_{10}	-32767_{10}
...			
$1111\ 1111\ 1111\ 1111_2$	=	65535_{10}	-1_{10}

Octal and Hexadecimal Numbers

Main purpose: short description of long bit sequences

Octal: base 8, digits 0...7

- $123_8 = 1 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0 = 83_{10}$
- converting to/from binary numbers easy: each octal digit represents group of 3 bits
- $100\ 110\ 001_2 = 461_8$
- in C, numbers with leading 0 are interpreted as octal numbers. E.g. $0377 = 11\ 111\ 111_2 = 255_{10}$

Hexadecimal: base 16, digits 0...9, a...f ("nibble")

- $3af_{16} = 3 \cdot 16^2 + 10 \cdot 16^1 + 15 \cdot 16^0 = 809_{10}$
- converting to/from binary numbers easy: each nibble represents group of 4 bits
- $1001\ 1101\ 0111_2 = 9d7_{16}$
- in C, prefix 0x indicates hexadecimal numbers. E.g. $0xff = 1111\ 1111_2 = 255_{10}$

Bitwise Operators

Useful for manipulating individual bits or groups of bits in integers

Is fast because up to 64 bits can be manipulated in one step (when using long long variables)

Can be used to change individual bits which allows us to create packed bit vectors (later)

~ : complement

& : bitwise AND

| : bitwise inclusive OR

^ : bitwise exclusive OR (XOR)

<< : left shift

>> : right shift

Complement

Think of `int x` as a 32-bit sequence: $x_{31} \ldots x_1 x_0$

x_0: least-significant bit, x_{31}: most-significant bit

E.g.,

```
int x = 127;  // 00000000 00000000 00000000 01111111
```

`z = ~x`

- inverts bits (0 → 1, 1 → 0)
- $z_i = \neg x_i$ $(i = 0..31)$
- $\neg 0 = 1,\ \neg 1 = 0$

```
 x = 0..[0]1010110
~x = 1..[1]0101001
```

AND

`z = x & y`

- applies operator $\wedge$ (Boolean AND) to pairs of corresponding bits
- $z_i = x_i \wedge y_i$ $(i = 0..31)$
- $a \wedge b$ true if and only if both a, b are true:

  $0 \wedge 0 = 0,\ 0 \wedge 1 = 0,\ 1 \wedge 0 = 0,\ 1 \wedge 1 = 1$

```
  x   = 0..01111100
  y   = 1..10101001
--------------------
x & y = 0..00101000
```


OR

`z = x | y`

- applies operator $\vee$ (Boolean OR) to pairs of corresponding bits
- $z_i = x_i \vee y_i$ $(i = 0..31)$
- $a \vee b$ true if and only if at least one of a, b is true:

  $0 \vee 0 = 0,\ 0 \vee 1 = 0,\ 1 \vee 0 = 0,\ 1 \vee 1 = 1$

```
  x   = 0..01111100
  y   = 1..10101001
--------------------
x | y = 1..11111101
```


XOR

`z = x ^ y`

- applies operator $\oplus$ (Boolean XOR) to pairs of corresponding bits
- $z_i = x_i \oplus y_i$ $(i = 0..31)$
- $a \oplus b$ true if and only if $a \neq b$:

  $0 \oplus 0 = 0,\ 0 \oplus 1 = 1,\ 1 \oplus 0 = 1,\ 1 \oplus 1 = 0$

```
  x   = 0..01111100
  y   = 1..10101001
--------------------
x ^ y = 1..11010101
```

Don't confuse `x^y` (XOR) with x^y (exponentiation)!

There is no exponentiation operator in C++
(use `pow(x,y)` to compute x^y)

Left Shift

`z = x << k`

- shifts all bits in `x` `k` places to the left and sets `k` least-significant bits to 0 $(0 \le \mathtt{k} < 32)$
- `k` most-significant bits are lost
- $z_{i+k} = x_i$ $(i = 0..31 - \mathtt{k})$, $z_0..z_{k-1} = 0$
- Left shifts are equivalent to multiplications with 2:
  `x << 1 == x * 2`

```
  x    = 0..01011111
--------------------
x << 1 = 0..10111110

  x    = 0..00001011111
-----------------------
x << 3 = 0..01011111000
```

Right Shift

`z = x >> k`

- shifts all bits in `x` `k` places to the right ($0 \leq$ `k` <32)
- `k` least-significant bits are lost
- `x` unsigned: clear `k` most-significant bits
- `x` signed: clone most-significant bit `k` times
- $z_{i-k} = x_i$ $(i = k..31)$, $z_{31}..z_{32-k}$ = 0 or 1 if $x < 0$.
- For unsigned numbers, right shifts are equivalent to division by 2: `x >> 1` $=$ `x / 2`
- For signed numbers, right shifts are only equivalent to division by 2 for positive numbers. If negative, the result is rounded down, rather than towards 0.

unsigned x:

```
  x    = 11..10111
------------------
x >> 3 = 00011..10
```

signed x:

```
  x    = 11..10111
------------------
x >> 3 = 11111..10
```


Example 1

```
int a, b, c, f, g;

a = a & 0xff;   // clears all but the lowest 8 bits
        // 0xff = 11111111 ( 0&0=0, 0&1=0, 1&0=0, 1&1=1 )

b = b | 5;    // sets the lowest and third lowest bit in b
              // 5 = 101_2 ( 0|0=0, 0|1=1, 1|0=1, 1|1=1 )

c = c ^ 0xffff0000;    // inverts the highest 16 bits in c
                       // ( 0^0=0, 0^1=1, 1^0=1, 1^1=0 )

f = ~f;          // negates all bits in f
                 // ~0=0xffffffff, ~0x55555555=0xaaaaaaaa

g = 16 >> 4;  // 1
g = 16 << 4;  // 256
g = -1 >> 4;  // -1
g = -1 << 4;  // -16
```


Example 2

```
// write number in octal format to standard output

unsigned char n = 65; // 8-bit unsigned integer

int d0 = n & 7;      // all bits but the lowest 3 set to 0
int d1 = (n >> 3) & 7; // middle 3-bit group of n
int d2 = (n >> 6) & 7; // left 2-bit group of n

printf("octal(%d) = %d%d%d\n", n, d2, d1, d0);

output: octal(65) = 101
```


Example 3

```
unsigned int x, k = 9;

// set k-th least-significant bit of x to 1
// without changing anything else
x = x | (1 << k);   // or x |= (1 << k);

// x          =  ???????? ???????? ???????? ????????
// 1 << k     =  00000000 00000000 00000010 00000000
// x | (1<<k) =  ???????? ???????? ??????1? ????????
// (? marks an arbitrary bit that is not altered)

// set k-th least-significant bit of x to 0
// without changing anything else
x = x & ~(1 << k);  // or x &= ~(1 << k);

// x           =  ???????? ???????? ???????? ????????
// 1 << k      =  00000000 00000000 00000010 00000000
// ~(1 << k)   =  11111111 11111111 11111101 11111111
// x & ~(1<<k) =  ???????? ???????? ??????0? ????????
```

Bit Vectors

How to implement a vector of 1024 bits, say, for which each bit can be addressed by its index?

```
BitVector bv(1024);   // 1024 bits
bv.set(533, 1);       // set bit number 533 to 1
bool a = bv.get(533); // get bit number 533
```

Using a char variable for each bit is wasteful because it would require 1024 bytes, whereas a packed representation requires only $1024/8 = 128$ bytes (8 bits per byte).

Idea: allocate array of 1024/8 char variables and identify bits by their bit index `i` ≥ 0:

Index of byte in which bit `i` resides : `i / 8` (= `i >> 3`) (round down)

Position of bit `i` in this byte: `i % 8 (= i & 7)`

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
byte	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2
bit	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0 ...

BitVector Implementation (1)

BitVector.h

```
class BitVector
{
public:
  // construct bit vector of size length
  // and set all bits to 0
  BitVector(int length);

  // destructor, releases memory
  ~BitVector();

  // return size
  int size();

  // clear all bits
  void clear();

  // returns bit i after index check in debug mode
  bool get(int i);

  // sets bit i to x after index check in debug mode
  void set(int i, bool x);

private:
  char *bytes; // bits stored in byte array
  int n;       // length
};
```


BitVector Implementation (2)

BitVector.c

```
#include "BitVector.h"
#include <assert.h>

// construct vector of length size
BitVector::BitVector(int size)
{
  assert(size > 0);
  n = size;
  bytes = new char[(n + 7)/8]; // can you explain this?
  clear();
}

// destructor
BitVector::~BitVector()
{
  delete [] bytes;
  bytes = 0; // safeguards
  n = 0;
}

// clear all bits
void BitVector::clear()
{
  for (int i=(n+7)/8-1; i >= 0; i--) {
    bytes[i] = 0;
  }
}
```


BitVector Implementation (3)

BitVector.c

```
// return size
int BitVector::size()
{
  return n;
}

// returns bit i after index check in debug mode
bool BitVector::get(int i)
{
  assert(i >= 0 && i < n);
  return bytes[i >> 3] & (1 << (i & 7));
}

// sets bit i to x after index check in debug mode
void BitVector::set(int i, bool x)
{
  assert(i >= 0 && i < n);
  // reference to byte where bit resides
  char &b = bytes[i >> 3];
  b &= ~(1 << (i & 7));   // clear bit
  b |= x << (i & 7);      // set bit value
}
```

BitVector Application

BitVectorMain.c

```
#include "BitVector.h"
#include <stdio.h>

int main()
{
  const int N = 32;
  for (int i=0; i < N; i++) {
    BitVector bv(N);
    bv.set(i, 1);

    for (int j=0; j < N; j++) {
      printf("%d", bv.get(j));
    }
    printf("\n");
    // bv destructor called here!
  }
}

output:
10000000000000000000000000000000
01000000000000000000000000000000
00100000000000000000000000000000
...
00000000000000000000000000001000
00000000000000000000000000000100
00000000000000000000000000000010
00000000000000000000000000000001
```


Bit Fields

```
// 64 bit struct storing file properties
struct FileData
{
  unsigned int sec    :  6;  // 0..59
  unsigned int min    :  6;  // 0..59
  unsigned int hour   :  5;  // 0..23
  unsigned int r      :  1;  // 0..1
  unsigned int w      :  1;  // 0..1
  unsigned int x      :  1;  // 0..1
  unsigned int d      :  1;  // 0..1
  unsigned int owner : 11;  // 0..2047
  unsigned int day;  : 22;  // 0..4194303
  signed   int level : 10;  // -512..511
};

FileData d;    // every entry undefined
d.r = 0;       // set read flag to false
d.sec = 59;    // set second to 59
d.sec += 10;   // 5
d.level = 511;
d.level++;     // -512
```

Integer variable followed by $: k$ declares a bit field that spans k bits. Can be signed or unsigned.

Bit Fields (Continued)

Useful for packing as much information as possible into a struct — minimizing waste.

Number ranges:

Unsigned: $0 \ldots 2^k - 1$

Signed: $-2^{k-1} \ldots 2^{k-1} - 1$

In each bit field component arithmetic is done modulo 2^k (wrap around) similar to regular C integer arithmetic.

Extracting and setting bit field values is more expensive than reading/writing aligned values of regular size because neighbouring components must not be changed. Here is what conceptually happens:

```
FileData d;
d.min = 7;  // = 000111 (k=6) at pos. 6

// if d is represented by 64-bit integer
// x above assignment is equivalent to
x &= ~(0x3f << 6); // clear 6 min bits
x |= (7 << 6);     // set 6 min bits to 7
```


Unions

```
enum SType { ST_INT, ST_FLOAT, ST_CHAR, ST_DOUBLE };

union Shared {
  int    i;  // all variables stored
  float  f;  // in the same location
  char   c;  // size of union is the maximum
  double d;  // size of each component (8 here)
};

struct __attribute__((packed)) SaveSpace {
  Shared u;  // can hold variable of one of 4 types
  char t;    // type of what is stored (1 byte)
};

SaveSpace s;                 // sizeof(s) = 9!
s.u.f = 3.5; s.t = ST_FLOAT;  // store float value
s.u.d = 4.7; s.t = ST_DOUBLE; // store double value
s.u.i = 5;   s.t = ST_INT;    // store int value
```

- Unions are space-saving structs (identical syntax)
- All data members are stored in the same location (only works for C data types)
- Therefore, a flag needs to be added that tells us what is currently stored for later when we access the data

Unions (Continued)

Application:

```
void printSafeSpace(const SaveSpace &s)
{
  switch (s.t) {
  case ST_FLOAT:
    printf("%f", s.u.f); // float converted to double
    break;
  case ST_DOUBLE:
    printf("%f", s.u.d);
    break;
  case ST_CHAR:
    printf("%c", s.u.c);
    break;
  case ST_INT:
    printf("%d", s.u.i);
    break;
  default:
    printf("unknown type");
    exit(1);
}
```


Byte Order

Sometimes we'd like to get access to individual bytes of variables, for instance to send data byte by byte through a network channel:

```
// Send n bytes buffer points to
void send(char *buffer, int n);
...
int a[100];
send(a, sizeof(a)); // error, not char*

// Solution: type cast from int* to char*
// This switches off the compiler's type check
send((char*)a, sizeof(a)); // OK? Maybe not ...
```

This works as long as the receiving computer uses the same byte ordering, i.e. how integers are stored:

- Low-order byte first ("little-endian"): int 1000 = $3E8_{16}$ is stored as E8 03 00 00 in memory
- High-order byte first ("big-endian"): 1000 = $3E8_{16}$ is stored as 00 00 03 E8 in memory

However, the client's endianness can't be enforced.

Sending Low-Order Bytes First

The solution to the problem is to use an endianness independent network protocol, e.g. sending low-order bytes first, and then decoding messages in clients depending on their endianness.

```
void send_int(int x)
{
#ifdef BIG_ENDIAN
  reverse_bytes(x);
#endif
  send_int_first_byte_first(x);
}

int receive_int()
{
  int x = receive_int_first_byte_first();
#ifdef BIG_ENDIAN
  reverse_bytes(x);
#endif
  return x;
}
```

This assumes that macro `BIG_ENDIAN` is set properly on the sending and receiving side. How to implement `reverse_bytes`?

Reverse Bytes

By using a pointer type cast we first get byte-level access to the int variable, and then use the standard char swap function to reverse the byte order:

```
#include <algorithm>

// reverse the byte order in x C++
void reverse_bytes(int &x)
{                            // e.g.:
  char *p = (char*) &x;      // x = 0xaa1122bb
  std::swap(p[0], p[3]);     // x = 0xbb1122aa
  std::swap(p[1], p[2]);     // x = 0xbb2211aa
}

// C version
void reverse_bytes_c(int *px)
{
  char *p = (char*) px, t;
  t = p[0]; p[0] = p[3]; p[3] = t;
  t = p[1]; p[1] = p[2]; p[2] = t;
}
```