Chapter - 11
Bit operations

Binary Numbers

Almost all machines organize their memory into 8 bit bytes. This means that a single location can hold 8 binary digits (bits) such as: `01100100`

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Example: 0xAF = 1010 1111(binary).

Bit Operators

Operator	Meaning
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~	Complement
<<	Shift left
>>	Shift right

The and operator (&)

Bit 1	Bit 2	Bit1 & Bit2
0	0	0
0	1	0
1	0	0
1	1	1

```
int  c1, c2;
c1 = 0x45;
c2 = 0x71;
std::cout << "Result of " << hex << c1 << " & " << c2 <<
            " = " << (c1 & c2) << dec << '\n';
```

The output of this program is:

```
Result of 45 & 71 = 41
```

	c1 = 0x45	binary 01000101
&	c2 = 0x71	binary 01110001
=	0x41	binary 01000001

"and" example

```
inline int even(const int value)
{
    return ((value & 1) == 0);
}
```


The difference between "and" (&) and "and and" (&&)

```
#include <iostream>
main()
{
    int i1, i2; // two random integers

    i1 = 4; i2 = 2;     // set values

    // Nice way of writing the conditional
    if ((i1 != 0) && (i2 != 0))
        std::cout << "Both are not zero #1\n";

    // Shorthand way of doing the same thing
    // Correct C++ code, but rotten style
    if (i1 && i2)
        std::cout << "Both are not zero #2\n";

    // Incorrect use of bitwise and resulting in an error
    if (i1 & i2)
        std::cout << "Both are not zero #3\n";
    return (0);
}
```


Bitwise or (|)

<table>
<tr><th>Bit 1</th><th>Bit 2</th><th>Bit1 | Bit2</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

<table>
<tr><td></td><td>i1=0x4 7</td><td>0100011 1</td></tr>
<tr><td></td><td>i2=0x5 3</td><td>0101001 1</td></tr>
<tr><td>|</td><td>57</td><td>0101011 1</td></tr>
</table>

Bitwise exclusive Or (^)

Bit 1	Bit 2	Bit1 ^ Bit2
0	0	0
0	1	1
1	0	1
1	1	0

	0x47	01000111
	0x53	01010011
^	0x14	00010100

One's Complement / Not (~)

Bit 1	~Bit1
0	1
1	0

	0x47	01000111
~	0xB8	10111000

The Left and Right Shift Operators (<<, >>)

	c=0x1 C	0001110 0
c << 1	c=0x38	00111000
c >> 2	c=0x07	00000111

Right Shift Details.

	signed char	**signed char**	**unsigned char**
Expression	9 >> 2	-8 >> 2	248 >> 2
Binary Value>> 2	$0000\ 1010_2$ >> 2	$1111\ 1000_2$ >> 2	$1111\ 1000_2$ >> 2
Result	$??00\ 0010_2$	$??11\ 1110_2$ >> 2	$??11\ 1110_2$ >> 2
Fill	Sign Bit (0)	Sign Bit (1)	Zero
Final Result (Binary)	$0000\ 0010_2$	$1111\ 1110_2$	$0011\ 1110_2$
Final Result (short int)	2	-2	62

Defining Bits

A byte is divided into 8 bits.
Example: 0xAF or 10101111 binary.

7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	1

Suppose we want to define five flags and put them in a byte. We start by assigning each flag a bit.

Bit	Name
0	ERROR
1	FRAMING_ERRO R
2	PARITY_ERROR
3	CARRIER_LOST
4	CHANNEL_DOWN

Bit Values the hard way

Bit	Binary Value	Hex Constant
7	10000000	0x80
6	01000000	0x40
5	00100000	0x20
4	00010000	0x10
3	00001000	0x08
2	00000100	0x04
1	00000010	0x02
0	00000001	0x01

```
// True if any error is set
const int ERROR        = 0x01;
const int FRAMING_ERROR= 0x02;// Char frame error
const int PARITY_ERROR = 0x04;// Wrong parity
const int CARRIER_LOST = 0x08;// No carrier signal
const int CHANNEL_DOWN = 0x10;// Power lost, no contact
```


Bit Values the Easy Way

C++ Representation	Base 2 Equivalent	Result (Base 2)	Bit Number
1<<0	00000001_2 << 0	00000001_2	Bit 0
1<<1	00000001_2 << 1	00000010_2	Bit 1
1<<2	00000001_2 << 2	00000100_2	Bit 2
1<<3	00000001_2 << 3	00001000_2	Bit 3
1<<4	00000001_2 << 4	00010000_2	Bit 4
1<<5	00000001_2 << 5	00100000_2	Bit 5
1<<6	00000001_2 << 6	01000000_2	Bit 6
1<<7	00000001_2 << 7	10000000_2	Bit 7

```
const int ERROR =          (1<<0); // Set if any error
const int FRAMING_ERROR = (1<<1); // Frame error
const int PARITY_ERROR =  (1<<2); // Parity error
const int CARRIER_LOST =  (1<<3); // No carrier
const int CHANNEL_DOWN =  (1<<4); // Power lost
```

Setting, Testing and Clearing a bit

Setting

```
char    flags = 0;  // start all flags at 0

flags |= CHANNEL_DOWN; // Channel just died
```

Testing

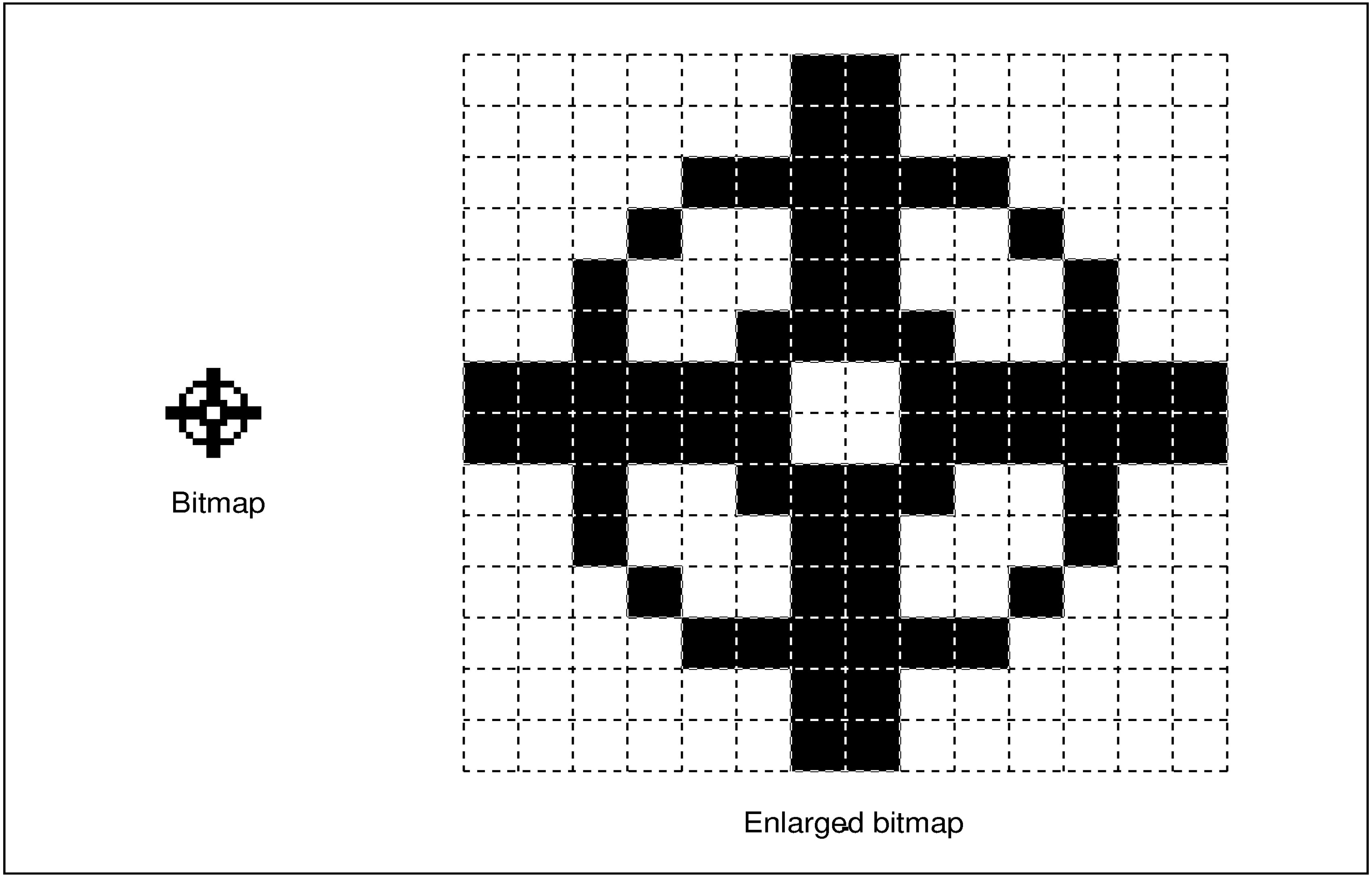
```
if ((flags & ERROR) != 0)
    std::cerr << "Error flag is set\n";
else
    std::cerr << "No error detected\n";
```

Clearing

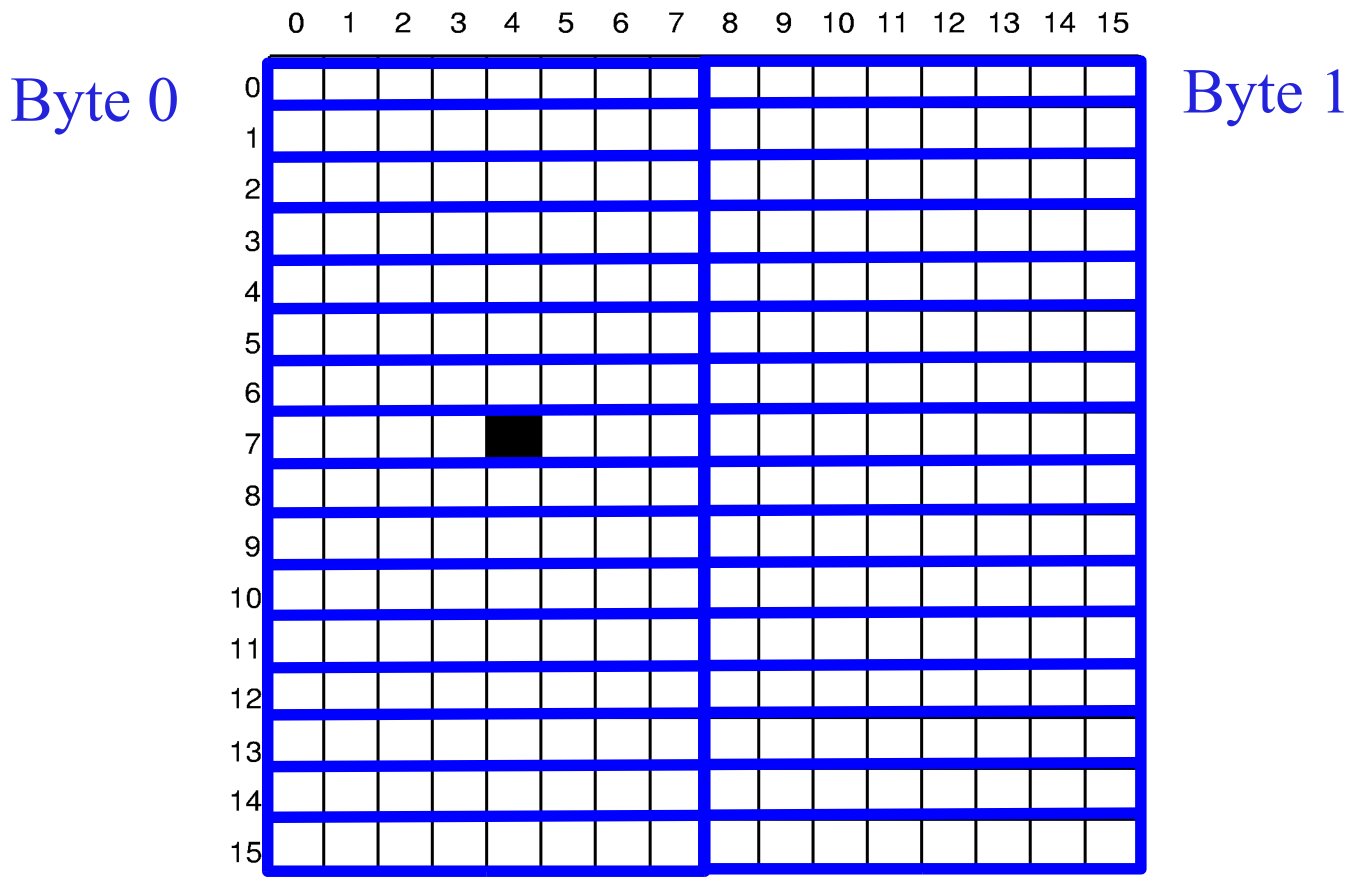
```
flags &= ~PARITY_ERROR; // Forget about parity
```

`PARITY_ERROR`	00000100
`~PARITY_ERROR`	11111011
`flags`	00000101
`flags & ~PARITY_ERROR`	00000001

Bitmapped Graphics



Setting a bit in a 16 by 16 bitmap



Pixel (4,7) set in a 16x16 array of bits

Array of Bytes

```
bit_array[0][7] |= (0x80 >> (4));
```


Bit addressing

Translating (X,Y) from a bit address to a byte address.
X needs to get turned into a byte, bit pair.

    X-Byte = X / 8 (8 bits per byte)
    X-Bit = 0x80 >> (X % 8);

(Start with left most bit and shift over one for each bit needed.)

Y translates directly (that was easy)

C++ Version:

```
void inline set_bit(const int x,const int y)
  {
      graphics[(x)/8][y] |= (0x80 >> ((x)%8))
  }
```



```
#include <iostream>


const int X_SIZE = 40; // size of array in the X direction
const int Y_SIZE = 60; // size of the array in Y direction
/*
 * We use X_SIZE/8 because we pack 8 bits per byte
 */
char graphics[X_SIZE / 8][Y_SIZE];   // the graphics data
/*******************************************************
 * set_bit -- set a bit in the graphics array.         *
 *                                                     *
 * Parameters                                          *
 *      x,y -- location of the bit.                    *
 *******************************************************/
inline void set_bit(const int x,const int y){
    graphics[(x)/8][y] |= (0x80 >>((x)%8));
}
main(){
    int   loc;        // current location we are setting
    void  print_graphics(void); // print the data


    for (loc = 0; loc < X_SIZE; ++loc)
        set_bit(loc, loc);


    print_graphics();
    return (0);
}
```



```
/********************************************************
 * print_graphics -- print the graphics bit array       *
 *              as a set of X and .'s.                   *
 ********************************************************/
void print_graphics(void)
{
    int x;     // current x BYTE
    int y;     // current y location
    int bit;   // bit we are testing in the current byte


    for (y = 0; y < Y_SIZE; ++y) {


        // Loop for each byte in the array
        for (x = 0; x < X_SIZE / 8; ++x) {


            // Handle each bit
            for (bit = 0x80; bit > 0; bit = (bit >> 1)) {
                if ((graphics[x][y] & bit) != 0)
                    std::cout << 'X';
                else
                    std::cout << '.';
            }
        }
        std::cout << '\n';
    }
}
```

One loop works, the other doesn't. Why?

```
#include <iostream>
main()
{
    short int i;

    // Works
    for (i = 0x80; i != 0; i = (i >> 1)) {
        std::cout << "i is " << hex << i << dec << '\n';
    }

    signed char ch;

    // Fails
    for (ch = 0x80; ch != 0; ch = (ch >> 1)) {
        std::cout << "ch is " << hex << int(ch) <<dec << '\n';
    }
    return (0);
}
```

