C Bitwise Operators and Eratosthenes Sieve

The 'C' Programming Language was designed as a high-level language intended to replace the need to use assembly languages to design operating system kernels and compilers etc. Other uses of 'C' for programming small embedded systems and device drivers also require direct access to bits stored in hardware. The bitwise operators can be used only on integral values and variables, i.e of types char, short, int and long.

Portability of programs using these operators is dependant upon whether signed or unsigned data is used, and the number of bytes allocated to specific data types on particular platforms.

Eratosthenes Sieve is a prime-number generating algoritm, used to demonstrate some of the bitwise operators. Prime numbers are of interest to mathematicians and to cryptographers. Cryptography is the mathematical foundation of computing and network security.

Summary of Bitwise Operators

operator	coding	notes
left shift	<<	shifts LHS left by no. of bits in RHS
right shift	>>	shifts LHS right by no. of bits in RHS
bitwise AND	&	returns parrallel and of LHS and RHS input bits
bitwise OR	|	returns parrallel or of LHS and RHS input bits
exclusive OR	^	returns 1 if LHS and RHS bits the same, 0 if different
twos complement	~	Unary NOT operator. Returns 0 for RHS 1 and 1 for 0.

Note that in-place versions also exist which modify the LHS operand in place rather than returning the result for a seperate assignment, e.g. a >>= b performs a right shift of b bits directly on variable a . These work in the same manner as += , -= *= operators compared to + - and * .

Left and Right Shift Operators

The >> operator shifts a variable to the right and the << operator shifts a variable to the left. Zeros are shifted into vacated bits, but with signed data types, what happens with sign bits is platform dependant. The number of bit positions these operators shift the value on their left is specified on the right of the operator. Uses include fast multiplication or division of integers by integer powers of 2, e.g. 2,4,8,16 etc.

Example:

```
#include <stdio.h>
int main(void){
	unsigned int a=16;
	printf("%d\t",a>>3); /* prints 16 divided by 8  */
	printf("%d\n",a<<3); /* prints 16 multiplied by 8 */
	return 0;
}
```

output: 2	128

Bitwise AND and inclusive OR operators

Single & and | operators (bitwise AND and OR) work differently from logical AND and OR (&& and ||). You can think of the logical operators as returning a single 1 for true, and 0 for false. The purpose of the & and | bitwise operators is to return a resulting set of output 1s and 0s based on the boolean AND or OR operations between corresponding bits of the input. Example:

```
#include <stdio.h>

int main(void){
	unsigned char a='\x00',b='\xff',c;
	c='\x50' | '\x07'; /* 01010000 | 00000111 */
	printf("hex 50 | 07 is %x\n",c);
	c='\x73' & '\x37'; /* 01110011 & 00110111 */
	printf("hex 73 & 37 is %x\n",c);
	return 0;
}
```

Output:

```
hex 50 | 07 is 57
hex 73 & 37 is 33
```

Bitwise exclusive OR operator

Symbol: ^

For each bit of output, this output is a 1 if corresponding bits of input are different, and the output is a 0 if the input bits are the same.

One's complement operator

Symbol: ~

This is a unary operator in the sense that it works on a single input value. The bit pattern output is the opposite of the bit pattern input - with input 1s becoming output 0s and input 0s becoming output 1s.

Setting a particular bit within a byte

The following prototype was used: void setbitn(unsigned char *cp, int bitpos, int value);
/* setbitn sets bit position 0 - 7 of cp to value 0 or 1 */
The byte cp (assuming chars are 1 byte wide) is passed by reference.

```
void setbitn(unsigned char *cp,int bitpos,int value){
 /* setbitn sets bit position 0 - 7 of cp to value 0 or 1  */
      unsigned char template=(unsigned char)1;
      /* first make template containing just the bit to set */
      template<<=bitpos;
      if(value) /* true if value is 1 false for 0. Bitwise OR will set templated bit in cp to 1 whatever
                   its current value, and leave other bits unchanged */
        *cp=*cp | template;
      else
        /* Invert template 1s and 0s. Next use bitwise AND to force templated bit in cp to 0
         * and leave all other bits in cp unchanged */
        *cp=*cp & ( ~ template);
}
```

Getting a particular bit within a byte

To return the value of a particular bit within a byte without changing the original, the following prototype was used: int getbitn(unsigned char c, int bitpos);

/* getbitn gets bit position 0 - 7 of c, returns 0 or 1 */

Call by value is used for the byte concerned, so this can be changed within the function, but as this is a copy the original byte won't be changed.

```
int getbitn(unsigned char c, int bitpos){  /*getbitn gets bit position 0 - 7 of c, returns value 0 or 1.
  * This function clobbers the c parameter, but we are using pass by value, so this won't affect
   * the original calling copy. */
      unsigned char template=(unsigned char)1; /* make template containing just the bit to get */
      template<<=bitpos;
      c&=template; /* if relevant bit set then c is assigned non null,
                  otherwise c is assigned null (all zeros) */
      if(c)   return 1;
      else    return 0;
}
```

Simulating a Boolean Array

'C' doesn't have a Boolean type. We can store 0s and 1s in chars or ints one bit per variable, but this wastes most of the space. If we want to be able to access individual bits from a large memory-efficient set of bits we can use a char array to allocate the storage, and then, once we have identified the correct char (byte) we can use our getbitn and setbitn functions to set and get individual bits.

We can't use array notation for this, but we can design accessor and mutator functions which can be called in a similar manner to array notation. If we define a constant: MAXBYTES for the purpose of allocating memory to our string array, e.g. `#define MAXBYTES 1000000` , we can allocate the memory needed for the bit array: char bitarray[MAXBYTES] This gives storage of MAXBYTES*8 bits, and we can use the constant within the accessor and mutator functions to avoid a buffer overrun.

```
void setbit(unsigned char *sp, size_t bitpos, int value);
  /* setbit sets bit position 0 - ((MAXBYTES * 8) - 1) to value 0 or 1 */
int getbit(unsigned char *sp, size_t bitpos);
  /* getbit gets bit position 0 - ((MAXBYTES * 8) - 1), returns 0 or 1 */
```

Parameter sp is passed the address of the array storing the bits, e.g. bitarray.

The setbit function

```
void setbit(unsigned char *sp, size_t bitpos, int value){
  /* setbit sets bit position 0 - ((MAXBYTES * 8) - 1) of string sp
   * to value 0 or 1 */
      size_t byteno;
      int bit07; /* will store a value from 0 - 7 indicating bit in byte */
      byteno=bitpos/8; /* floor division to get byte number in string */
      if(byteno >= MAXBYTES){
            fprintf(stderr,"setbit: buffer overflow trapped\n");
            exit(1); /* needs #include <stdlib.h> */
      }
      bit07=(int)bitpos%8; /* remainder value 0 - 7 */
      setbitn(sp+byteno,bit07,value);
}
```

The getbit function

```
int getbit(unsigned char *sp, size_t bitpos){
  /* getbit gets bit position 0 - ((MAXBYTES * 8) - 1) of string sp,
   * returns 0 or 1 */
        size_t byteno;
        int bit07;
        byteno=bitpos/8; /* floor division to get byte number in string */
        if(byteno >= MAXBYTES){
                fprintf(stderr,"getbit: attempt to read beyond allocated buffer\n");
                fprintf(stderr,"bitpos: %d\n",bitpos);
                exit(1);
        }
        bit07=(int)bitpos%8; /* remainder value 0 - 7 */
        return getbitn(*(sp+byteno),bit07);
}
```

Prime Number Generating Algorithms

Prime numbers have exactly 2 natural number factors, themselves and 1. Natural numbers are the integers (counting numbers) in the set starting 1, 2, 3 . The first prime number is therefore 2, and other primes include 3, 5, 7, 11, 13 and 17. A factor is a number that divides exactly into another number, e.g. 3 is a factor of 9 but not of 10.

Prime numbers are interesting because all numbers larger than 2 are either primes, or can be made by multiplying prime factors together, e.g. 12 which is not prime is made by multiplying 2 x 2 x 3.

Being able to make very large prime numbers is important because computing security relies on encryption algorithms which depend upon the fact that it is very quick to multiply a pair of large (e.g. 100 digit) prime numbers together to make a 200 digit number, but it would take a very long time to factorise the larger 200 digit number. Algorithms which can generate large prime numbers quickly therefore have important implications for computing security.

To generate a set of prime numbers, we could print all prime numbers less than a given number H (highest number to try). We could print the first prime number, 2. We could then try each value V starting with 3 up to H, checking it for factors between 2 and V-1. Number without factors are prime.

In pseudocode:

```
input highest (H) number to try
print 2
for values V from 3 to H:
  for values i from 2 to V - 1:
    if V%i == 0: // true if i is a factor if V
        // V isn't prime
        break // skip to next item i
    print V // we didn't find a factor so value is prime
```

This algorithm doesn't need much memory, but it is very slow. We don't need to check values of i greater than the square root of V as these can't be factors. We also don't need to bother checking values of i which are not prime. E.G. if 4 is a factor of V then 2 is as well, and 2 would have been found first. However, checking values V for prime factors less than or equal to the square root of V requires these primes be stored in an array, which will use more memory. Algorithm 2 is faster.

In pseudocode:

```
input highest (H) number to try
print and store 2 in primes array
for values V from 3 to H:
  for all items i in primes array from 2 <= sqrt(V):
    if V%i == 0: // true if i is a factor if V
        break // skip to next item i
    print V and store V in primes array // no factor: V is prime
```

Eratosthenes Sieve

A faster algorithm, Eratosthenes Seive, involves having a large array of 1s and 0s . Positions in this array all start at 0 (meaning the number or index of the position could be prime) except for positions 0 and 1 (which are not prime numbers. Starting with 2 this number is multiplied by 2, 3, 4 etc. to strike out the even numbers starting 4, 6, 8 etc. up to the highest position in the boolean array. These bits are set to 1s, meaning they can't be prime. The next 0 in the array is then found, which is the next prime number 3, and this number is multiplied by all higher numbers, and all multiples are set to 1. This process continues until the first prime greater than the square root of the highest bit position is found. Bitwise operators are needed to implement Eratosthenes Sieve algorithm without wasting memory.

```
int main(void){  /* from eratosthenes.c , Richard Kay, Sept 2005 */
        int i,j,nextprime=2,k;
        unsigned char a[MAXBYTES]; /* allocate memory for bit array */
        for(i=0;i<MAXBYTES;i++) a[i]=0x00; /* initialise all bits to 0s */
        setbit(a,0,1); setbit(a,1,1); /* seed 0 and 1 as not prime numbers */
        printprime(2);
        while(nextprime+1<MAXBYTES*8){
                k=nextprime;
                /* mark multiples of nextprime as not being possibly prime */
                while(nextprime*k<MAXBYTES*8){
                        setbit(a,nextprime*k,1);
                        k++;
                }
                /* find nextprime by skipping non-prime bits marked 1 */
                while(nextprime+1<MAXBYTES*8 && getbit(a,++nextprime));
                printprime(nextprime);
        }
        return 0;
}
```

```
void printprime(int prime){ /* prints a prime number in next 8 columns */
	static int numfound=0; /* static so only initialises at compile time,
	  and previous values are remembered in subsequent calls */
	if(numfound%8==0)
		printf("\n"); /* start next row */
	if(prime+1<MAXBYTES*8)
		printf("%d\t",prime);
	numfound++;
}
```

More efficient prime number generating algorithms exist. The memory efficiency of the Eratosthenes algoritm can be doubled if the number represented by a position in the bit array is obtained by doubling the position and adding 1 to it. This works because the first prime: 2 is hardcoded, and no other even number is prime. A recent further optimisation known as Atkin's Sieve is faster, but this relies upon more advanced mathematics.

Further reading

http://en.wikipedia.org/wiki/Sieve_of_Atkin http://en.wikipedia.org/wiki/Primality_test