S&T2024

Computer Programming

(Part 2 – Advanced C Programming Language)

Chapter 3

Lecturer
A/Prof Tay Seng Chuan

E-mail: pvotaysc@nus.edu.sg

Office of the Provost
National University of Singapore

Ground Rules

- Switch off your handphone and pager
- No talking while lecture is going on
- No gossiping while the lecture is going on
- Raise your hand if you have question to ask
- Be on time for lecture
- Be on time to come back from the recess break to continue the lecture
- Bring your printed lecture notes to lecture or use a laptop for the e-copy. Do not use handphone to read the pdf file.
- Do not open any app, except this pdf file and an editing tool (to take notes).

Chapter 3
Bit Manipulation

1. Introduction

- C allows programs to manipulate data at bit level.
- In order to speed operations, bits are organized into groups such as a *byte,* which is normally eight bits, or a *word* containing several bytes.
- Word sizes vary from machine to machine; on commercially available computers they range from 16 bits at the lower end to 64 bits on some large, scientifically oriented machines.
- In C an unsigned int would normally corresponds to a machine word and is the most natural type to use if bits are being manipulated, although signed int is sometimes used.

Basic Operations

The operations available on words considered as bit values are the logical operations and various types of shift. The logical operations are:

& Bitwise AND

Each bit of the left-hand operand is logically ANDed with each bit of the right-hand operand. ANDing two bits together gives the result zero unless both bits are one. **(both must be one to get one)**

&	0	1
0	0	0
1	0	1

The & Table

E.g.,

```
unsigned i = 269, j = 187, k;
k = i & j;
printf("k = %u",k);
```

will give **k = 9**

```
   .......00100001101  (269)
& .......00010111011  (187)
   .......00000001001  (  9)
```

| Bitwise inclusive OR

Each bit of the left-hand operand is logically ORed with each bit of the right-hand operand. Inclusively ORing two bits together gives the result one if either of the two bits is one, or both bits are one.
(either one or both to get one)

|	0	1
0	0	1
1	1	1

The | Table

E.g.,

```
unsigned i = 269, j = 187, k;
k = i | j;
printf("k = %u",k);
```

will give **k = 447**

```
 .......00100001101  (269)
|.......00010111011  (187)
 .......00110111111  (447)
```


^ Bitwise exclusive OR
(^ : Pronounced as Caret)

Each bit of the left-hand operand is exclusively ORed with each bit of the right-hand operand. Exclusively ORing two bits together gives the result one if either, but not both, of the two bits is one.
(one and only one to get one)

^	0	1
0	0	1
1	1	0

The ^ Table

E.g.,

```
unsigned i = 269, j = 187, k;
k = i ^ j;
printf("k = %u",k);
```

will give **k = 438**

```
 .......00100001101  (269)
^.......00010111011  (187)
 .......00110110110  (438)
```

~ Bitwise complement (also called NOT)

This is a monadic operator (i.e., has no left-hand operand) and it reverses each bit of its operand.
E.g.,

```
unsigned i = 269;
i = ~ i;
printf("i = %u",i);
```

will give **i = 65266** on a 16-bit machine.

~	0000	0001	0000	1101 (269)
	1111	1110	1111	0010 (65266)

<< Left shift operator

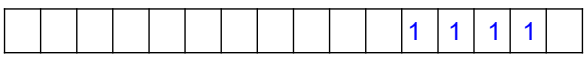
E.g., target << n, where target and n can be expressions. n must be a positive value. Bit contents will be dropped when shifted to the left end.
E.g.,

```
unsigned i = 19, j;

j = i << 2;
   /* meaning that shift the bit pattern of i to the
      left by 2 places and assign the result to j */

printf("%u",j);
```

will give **76**

...00010011 << 2 = ...01001100
= 76

Take note that shift left 2 places is equivalent to "multiplied by 4".
What is the result after shifting left 5 places ?



>> Right shift operator

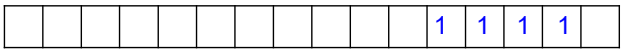
E.g.,

```
unsigned i = 110;
i = i >> 3;
/* means to shift the contents of bit pattern
   in i to the right by 3 places and assign
   the result back to i */
printf("%u",i);
```

will give **13**

...1101110 = ...1101 (some bits on the right hand side are dropped)

= 13



Take note that shift right 3 places is equivalent to "divided by 8".
What is the result after shifting right 5 places?

Short Form :

.a = a+b; can be written as a += b;
.Similarly a = a|b; can be written as a|=b;
.Other operators include &=, ^=, <<= and >>=.

Bitwise Operations :

E.g., Let a = 1101 1100 0101 1100 in binary. To extract the rightmost eight bits of a, we use the & operator as follows:

```
  1101 1100 0101 1100
& 0000 0000 1111 1111
-------------------
  0000 0000 0101 1100
```

Same technique can be used to test individual bits or groups of bits.
E.g., to test the right-most bit of V

if (v & 1) ...

```
  1101 1100 0101 110?
& 0000 0000 0000 0001
-------------------
  0000 0000 0000 000?
```

E.g., to test the right-most 3 bits of V

if (v & 7) ...

```
  1101 1100 0101 1???
& 0000 0000 0000 0111
-------------------
  0000 0000 0000 0???
```

Application of & Operator:

All odd numbers have a 1 in the right-most bit of its binary representation. So, to check if a number is odd we simply check its right-most bit. The following function will return 0 (false) if its parameter is an even number and 1 (true) if it is odd.

```
int odd(int n)
{
   return(n & 1);      /* Test if n is odd */
}
```

To make use of this function to check if num is odd :

```
int num;
      :
      :
if odd (num)    /* call the above function */
   printf ("num is odd");
else
   printf ("num is even");
```


Application of | Operator :

OR operations can be used to set bits in a word. For instance, to set the leftmost four bits of a 16-bit integer v to 1, leaving the others unchanged, we can use the expression v |= 0xf000.

E.g., v = 0x639c;

```
  0110 0011 1001 1100  (0x639c)
| 1111 0000 0000 0000  (0xf000)
-------------------------------
  1111 0011 1001 1100  (0xf39c)
```

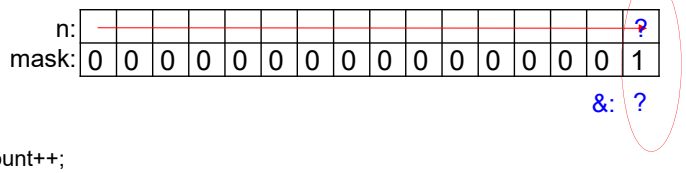
3.An Efficient Way to Perform Bit Counting

A simple practical example involves trying to count the number of bits in a word which are set to 1, a problem which can arise in data communication.

The obvious way is to check every bit. This is implemented by a static loop instruction.

Program

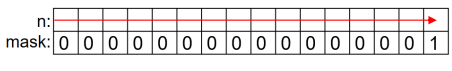
```
/* AC2-1.C */
#include<stdio.h>

// Assume 16 bits in integer
int Count_Bits_Obvious(unsigned n)
{
  int count=0,i;
  int mask=1;

  for (i=0;i<16;i++)
  {
    if (n & mask)  count++;
    n>>=1;
  }
  return count;
}
```




```
main()
{
  int number;

  printf("Enter an integer:");
  scanf("%d",&number);
  printf("There are %d 1-bits.\n", Count_Bits_Obvious(number));

  return 0;
}
```

Screen Output

Enter an integer:*57*
There are 4 1-bits.



Time Efficiency Consideration :

It requires 16 iterations for any bit patterns of n. This is very inefficient when n contains a lot of 0-bits. E.g., if n = 16384. It takes 16 iterations to discover that the number of 1-bits in 16384 (2^{14}) is only 1.

A better method to count the number of 1-bits:

In each iteration we hop to the nearest 1-bit and erase it. Repeat these operations until the number becomes 0. The number of iterations is the number of 1-bits.

Example :

Assume 16 bits

n = 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

It incurs 3 iterations before n becomes 0.
The number of 1-bits in the number is 3.

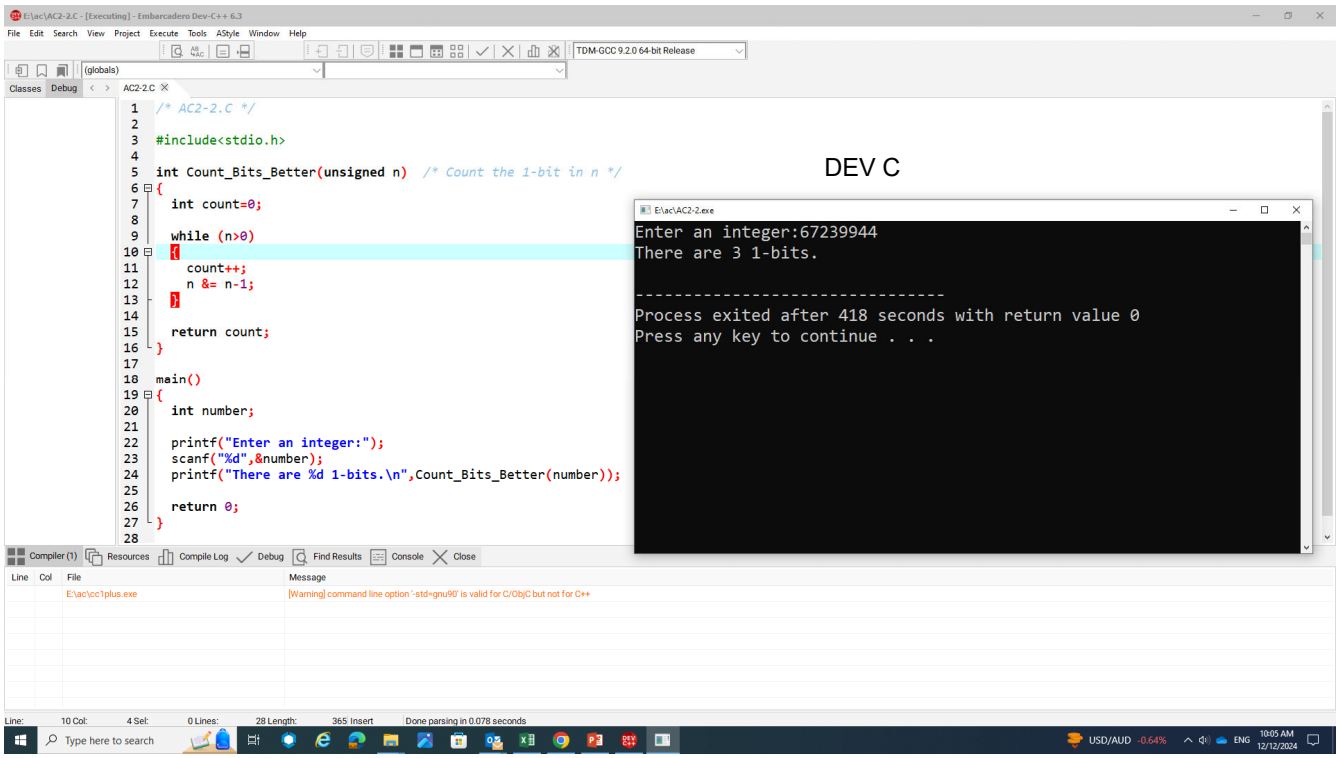
Program

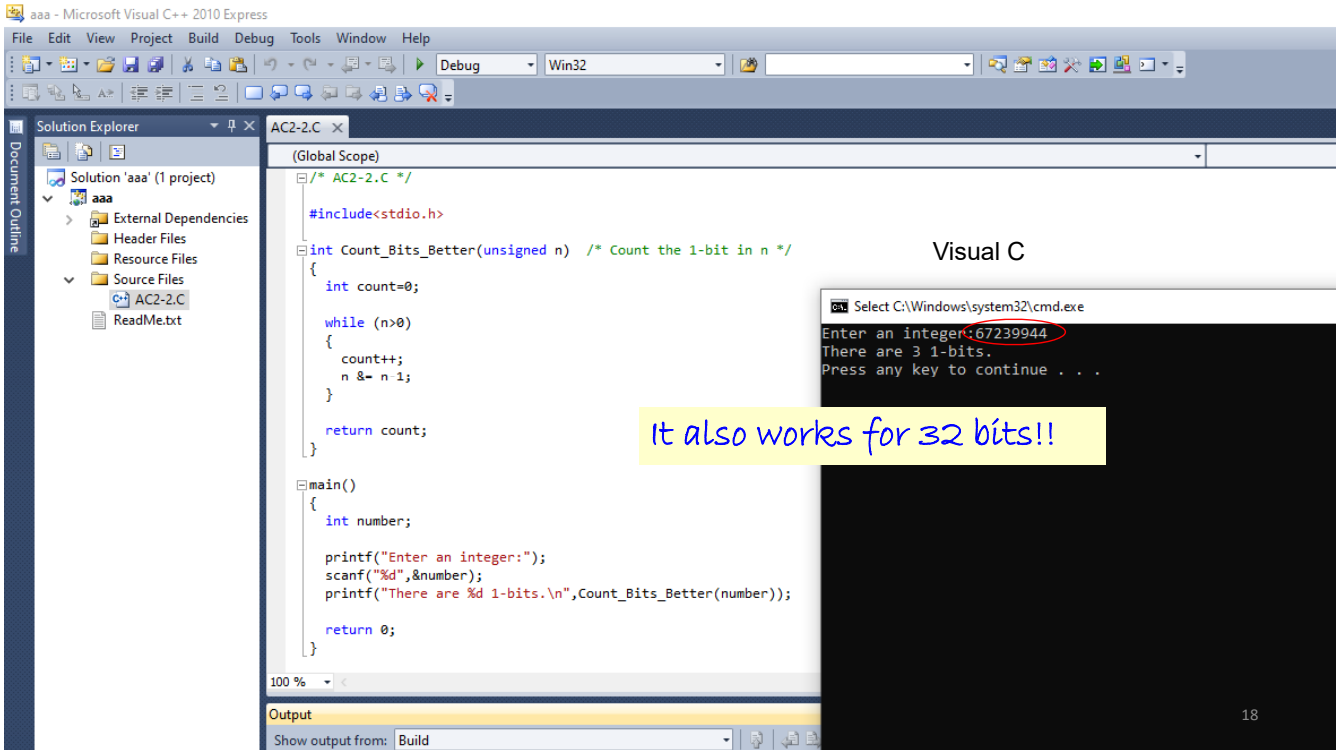
```
/* AC2-2.C */
#include<stdio.h>

int Count_Bits_Better(unsigned n)   /* Count the 1-bit in n */
{
  int count=0;

  while (n>0)
  {
    count++;
    n &= n-1;    /* n = n & (n-1); */
  }
  return count;
}
```

```
main()
{
  int number;
  printf("Enter an integer:");
  scanf("%d",&number);
  printf("There are %d 1-bits.\n",
      Count_Bits_Better(number));
  return 0;
}
```

Screen Output

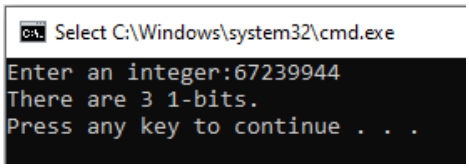
Enter an integer:*78*
There are 4 1-bits.

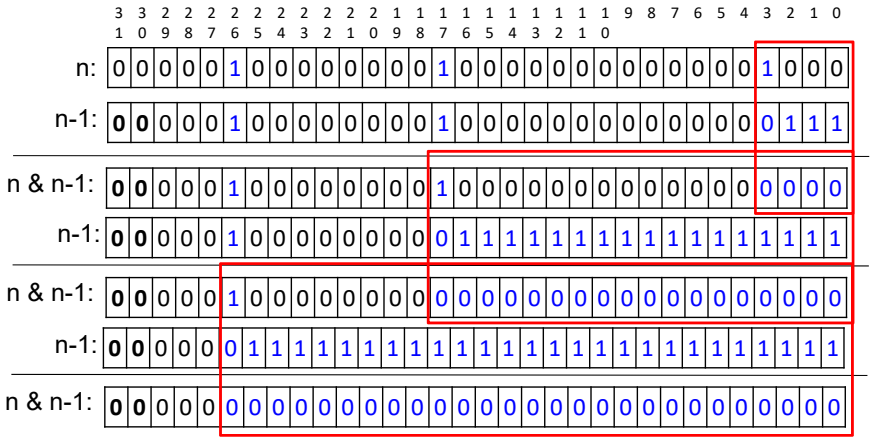

DEV C
/* AC2-2.C */
#include<stdio.h>
int Count_Bits_Better(unsigned n) /* Count the 1-bit in n */
{
int count=0;
while (n>0)
{
count++;
n &= n-1;
}
return count;
}
main()
{
int number;
printf("Enter an integer:");
scanf("%d",&number);
printf("There are %d 1-bits.\n",Count_Bits_Better(number));
return 0;
}
Enter an integer:67239944
There are 3 1-bits.
Process exited after 418 seconds with return value 0
Press any key to continue . . .


aaa - Microsoft Visual C++ 2010 Express
Visual C
Solution Explorer
Solution 'aaa' (1 project)
aaa
External Dependencies
Header Files
Resource Files
Source Files
AC2-2.C
ReadMe.txt
/* AC2-2.C */
#include<stdio.h>
int Count_Bits_Better(unsigned n) /* Count the 1-bit in n */
{
int count=0;
while (n>0)
{
count++;
n &= n-1;
}
return count;
}
main()
{
int number;
printf("Enter an integer:");
scanf("%d",&number);
printf("There are %d 1-bits.\n",Count_Bits_Better(number));
return 0;
}
Select C:\Windows\system32\cmd.exe
Enter an integer:67239944
There are 3 1-bits.
Press any key to continue . . .
It also works for 32 bits!!

67239964 = 67108864+131072+8

$= 2^{26} + 2^{17} + 2^{3}$





3 iterations only!!

4. Bit Rotations

-Slightly different from bit shifting.

-Dropped bits are appended to the other end.

E.g.,

0001 *0001 0000 0001* left-rotated 4 bits becomes

0001 0000 0001 0001

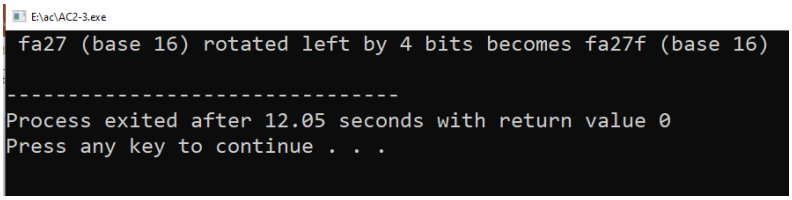
But for shift operation, bits are dropped !!!!

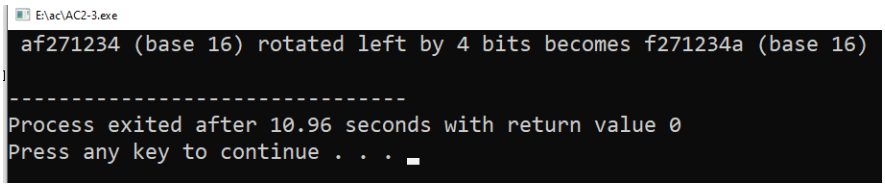
E.g.,

0001 *0001 0000 0001* << 4 = *0001 0000 0001* 0000

-No rotate operator in C language.

In the following program, a rotate_l function is written to rotate an integer x to the left by n places. A main function is also included to test the function.





Program

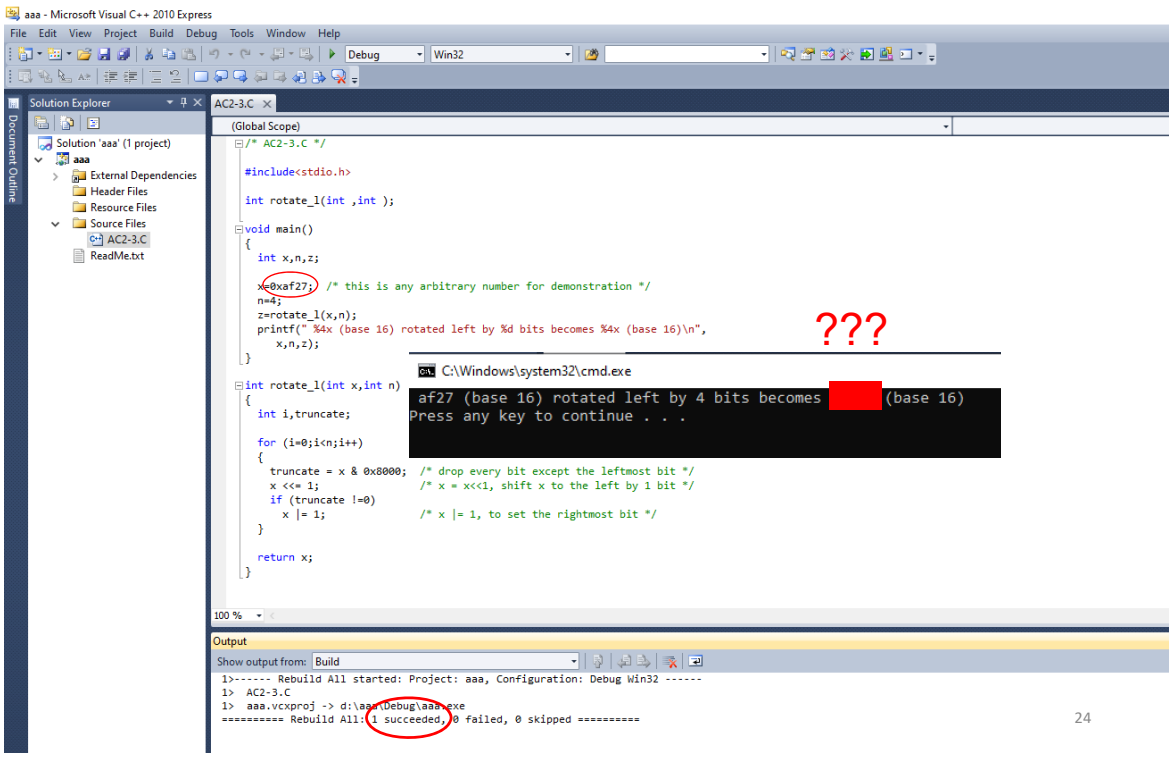
```
/* AC2-3.C */
#include<stdio.h>
int rotate_l(int ,int );
int main()
{
   int x,n,z;
   x=0xfa27;  /* this is any arbitrary number for demonstration */
   n=4;
   z=rotate_l(x,n);
   printf(" %4x (base 16) rotated left by %d bits becomes %4x (base 16)\n", x,n,z);

  return 0;
}

int rotate_l(int x,int n)
{
   int i,truncate;
   for (i=0;i<n;i++)
   {
      truncate = x & 0x8000;    /* drop every bit except the leftmost bit */
     x <<= 1;                   /* x = x<<1, shift x to the left by 1 bit */
     if (truncate !=0)  x |= 1;     /* x |= 1, to set the rightmost bit */
  }
  return x;
}
```

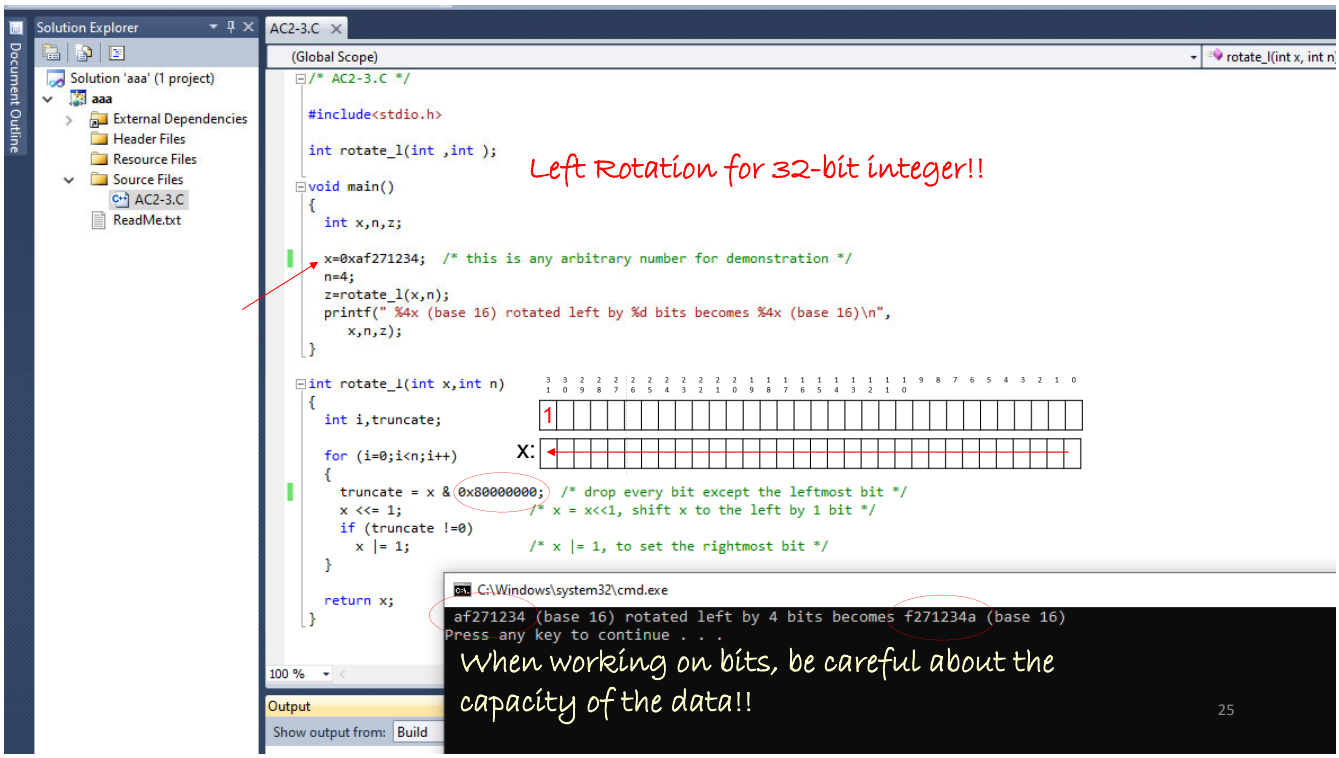
truncate = x & 0x8000;

X:	?															
& 0x8000:	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	?	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Screen Output

fa27 (base 16) rotated left by 4 bits
becomes a27f (base 16)





5. Case Study

A 21st-century date can be written with integers in the form day/month/year. An example is 24/1/03, which represents 24 January 2003. A simple way to store the date is using a struct definition as follows :

```
struct date
{
  int day;
  int month;
  int year;
};
```

Storage Efficiency Consideration:

- Too much storage overhead; we only require 31 different values for the day, 12 different values for the month and 100 different values for the year (2001 to 2100).
- So we need only 5 bits to represent the day (2^5=32), 4 bits to represent the month (2^4=16), and 7 bits to represent the year (1 to 100, 2^7=128). The total number of bits is 5+4+7= 16 bits.

The next program packs 2 dates in one 32-bit unsigned integers.

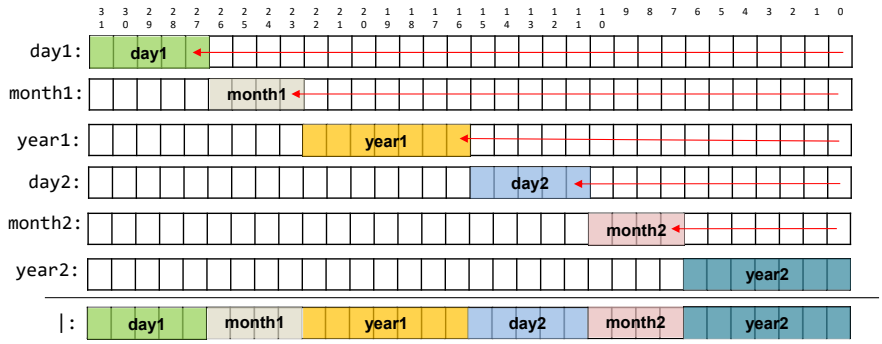
```c
/* AC2-4.C */
// For 32 bits

#include<stdio.h>

unsigned pack_date(int day1,int month1,int year1,
                   int day2,int month2,int year2)
{
  unsigned packed;

  day1 <<= 27;
  month1 <<= 23;
  year1 %= 100;
  year1 <<=16;

  day2 <<= 11;
  month2 <<= 7;
  year2 %= 100;

  packed = day1 | month1 | year1 | day2 | month2 | year2;

  return packed;
}


void print_bit(unsigned word)
{
  int n=sizeof(int)*8,i;
  unsigned mask=1<<(n-1);

  for (i=0;i<n;i++)
  {
    if (word & mask)
      printf("1");
    else
      printf("0");
    word <<= 1;
  }
  putchar('\n');
}
```

```c
void unpack_date(unsigned packed)
{
  int day,month,year;

  day = packed >> 27;
  month = (packed & 0x07800000) >> 23;
  year = (packed & 0x007f0000) >> 16;
  printf("Date 1: %02d/%02d/%02d\n", day,month,year);

  day = (packed & 0x0000f800) >> 11;
  month = (packed & 0x00000780) >> 7;
  year = packed & 0x007f;
  printf("Date 2: %02d/%02d/%02d\n", day,month,year);
}


main()
{
  int d1,m1,y1,d2,m2,y2;
  unsigned packed;

  printf("Enter the first date in the form dd/mm/yyyy  : ");
  scanf("%d/%d/%d",&d1,&m1,&y1);

  printf("Enter the second date in the form dd/mm/yyyy : ");
  scanf("%d/%d/%d",&d2,&m2,&y2);

  packed=pack_date(d1,m1,y1,d2,m2,y2);
  printf("The packed bit pattern is : ");
  print_bit(packed);
  unpack_date(packed);

  return 0;
}
```

```
C:\Windows\system32\cmd.exe
Enter the first date in the form dd/mm/yyyy  : 19/11/2009
Enter the second date in the form dd/mm/yyyy : 31/3/2012
The packed bit pattern is : 10011101100010011111100110001100
Date 1: 19/11/09
Date 2: 31/03/12
Press any key to continue . . .
```

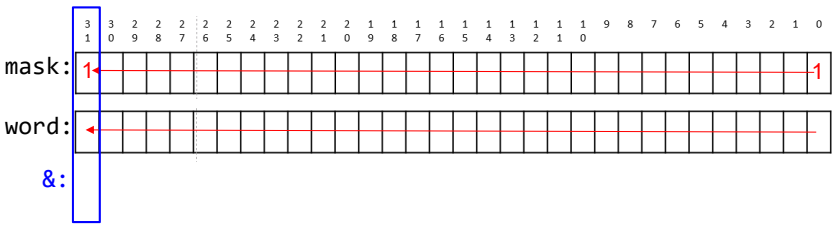
19 11 9 31 3 12


```c
unsigned pack_date(int day1,int month1,int year1,
                   int day2,int month2,int year2)
{
  unsigned packed;

  day1 <<= 27; month1 <<= 23; year1 %= 100; year1 <<=16;
  day2 <<= 11; month2 <<= 7; year2 %= 100;
  packed = day1 | month1 | year1 | day2 | month2 | year2;
  return packed;
}
```

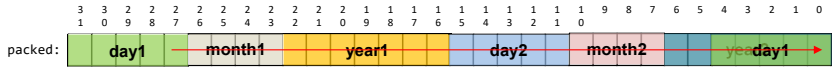


```
void print_bit(unsigned word)
{
  int n=sizeof(int)*8,i;
  unsigned mask=1<<(n-1);

  for (i=0;i<n;i++)
  {
    if (word & mask) printf("1");
    else printf("0");
    word <<= 1;
  }
  putchar('\n');
}
```

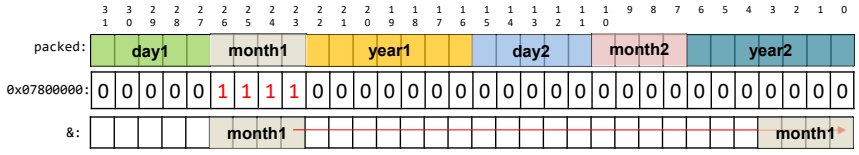
```
C:\Windows\system32\cmd.exe
Enter the first date in the form dd/mm/yyyy  : 19/11/2009
Enter the second date in the form dd/mm/yyyy : 31/3/2012
The packed bit pattern is : 10011101100010011111100110001100
Date 1: 19/11/09
Date 2: 31/03/12
Press any key to continue . . .
```



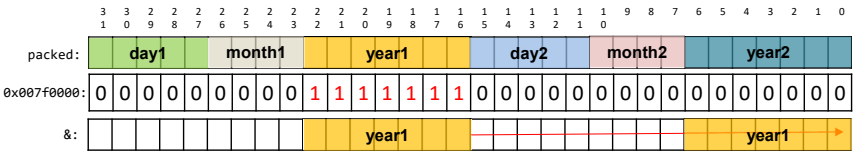

```
void unpack_date(unsigned packed)
{
  int day,month,year;

  day = packed >> 27;
```




```
  month = (packed & 0x07800000) >> 23;
```

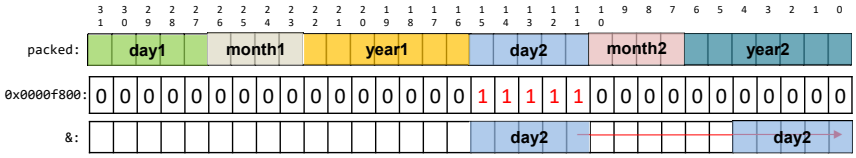



```
  year = (packed & 0x007f0000) >> 16;
```

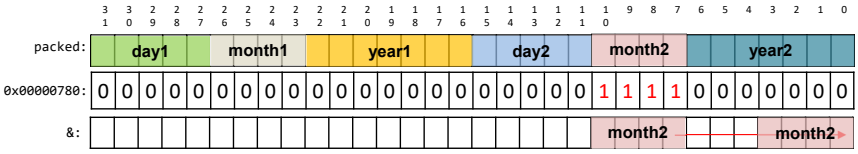



```
  printf ("Date 1: %02d/%02d/%02d\n", day,month,year);
```

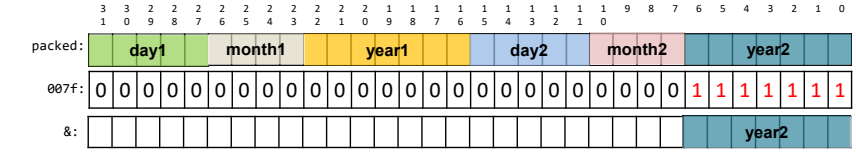
```
day = (packed & 0x0000f800) >> 11;
```




```
month = (packed & 0x00000780) >> 7;
```




```
year = packed & 0x007f;
```




```
  printf("Date 2: %02d/%02d/%02d\n", day,month,year);
}
```