SCHOOL OF ENGINEERING

Software and Embedded Systems Laboratory 2 (ELEE08022)
C language part

Bit Operation & Dynamic Memory Allocation

1. Bit Operations

The operators we have used so far have dealt with the value of complete variables. One of the attractive features of C, for engineering applications, is its provision of *operators* for manipulating the *individual bits* of *integer* and *character* variables. Bitwise operators are often used with **`unsigned`** variables but are equally applicable to *signed* variables. The C language forbids their use in floating point expressions.

A table showing the *bit-wise* operators is given below:

Bitwise operator	Function
`&`	Bit-wise AND
`|`	Bit-wise OR
`^`	Bit-wise EXCLUSIVE OR
`~`	Bit-wise Complement (Inversion)
`<<`	Left Shift
`>>`	Right Shift

Beware not to confuse the *bit-wise AND* and *OR* operators `&` and `|` with the *logical AND* and *OR* operators `&&` and `||`. All the *bit-wise operators* operate on *individual* bits in one or both *operands*.

1.1. The Bit-Wise AND Operator (`&`)

This performs a Boolean **AND** of *corresponding pairs of bits* in the two operands and only *sets* the corresponding bit in the *result* if the relevant bits in *both* operands are also set (i.e. `1`). Thus, the result of the *bit-wise* **AND** of `10101100` with `10011001` is `10001000`. The bit-wise **AND** operator is most useful when we wish to clear certain bits in a variable to `0` and leave the other bits unaltered. A common application of this is where we wish to select just certain bits from a variable. For example, if we wish to select just the least significant (lower) `4` bits (*nibble*) of an `8`-bit byte we can **AND** it with a *mask* value of `0x0F`. Similarly we can select just the least significant (lower) byte of a `16`-bit number by **AND**ing it with a mask value of `0xFF`, as shown in the following program.

Program 1

```
#include <stdio.h>
#define LOW_NIBB 0xF       /* mask for ANDing to get lower nibble */
#define LOW_BYTE 0xFF        /* mask for ANDing to get lower byte */

int main(void)
{
unsigned char one_byte; /* create an unsigned char, usually 8 bits*/
unsigned short two_byte;  /* create a short integer, often 16 bits*/
one_byte = 0x4E;                    /* initialise value to 4E hex */
two_byte = 0x5141;                /* initialise value to 5141 hex */
one_byte = one_byte & LOW_NIBB;       /* select only lower 4 bits */
two_byte &= LOW_BYTE;                 /* select only lower 8 bits */
printf("Lower 4 bits of 'char' = %x, Lower 8 bits of 'short' = %x
\n", one_byte, two_byte);
return 0;
}
```

This gives:
`Lower 4 bits of 'char' = E, Lower 8 bits of 'short' = 41`
Note that the combination of the bit-wise and assignment operators (e.g. **`&=`**, as for the arithmetic operators) is very convenient when carrying out bit-wise operations.

1.2. The Bit-Wise OR Operator (`|`)

This performs a Boolean **OR** of corresponding pairs of bits in the two operands and *sets* the corresponding bit in the result if the bit is set in *either* operand. Thus, the bit-wise **OR** of **`10101100`** with **`10011001`** is **`10111101`**. The bit-wise **OR** operator is most useful when we wish to set certain bits in a variable to **`1`** and leave the other bits unaltered.

1.3. The Bit-Wise Exclusive OR Operator (`^`)

This performs a Boolean **exclusive OR** of corresponding pairs of bits in the two operands and *sets* the corresponding bit in the result if *only one* of the operands has the bit set. Thus, the bit-wise **exclusive OR** of **`10101100`** and **`10011001`** is **`00110101`**. The bit-wise **exclusive OR** operator is most useful when we wish to *complement* an individual bit of a variable. Exclusive ORing a bit with **`1`** will clear the bit if it was set and set it if it was clear. Exclusive ORing a bit with **`0`** will leave the bit value unchanged.

1.4. The Complement Operator (`~`)

This is a *unary* operator which simply *inverts each* bit in the variable. Thus, the bit-wise **complement** of **`10101100`** is **`01010011`**. The complement operator is commonly used with *hexadecimal* or *octal* constants in conjunction with other bit-wise operators. The following statement:

```
num1 &= ~0xF;
```

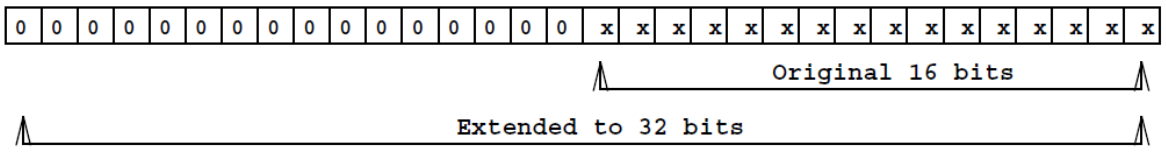
clears the last *four* bits of the variable **`num1`** to **`0`**, irrespective of their previous value. This works whatever the *size* of the variable **`num1`**, whereas using *only* the **AND** operator for this function e.g:

```
num1 &= 0xFFF0;
```

requires knowledge of the *exact* size of the variable, so that the correct *size of mask* is used.

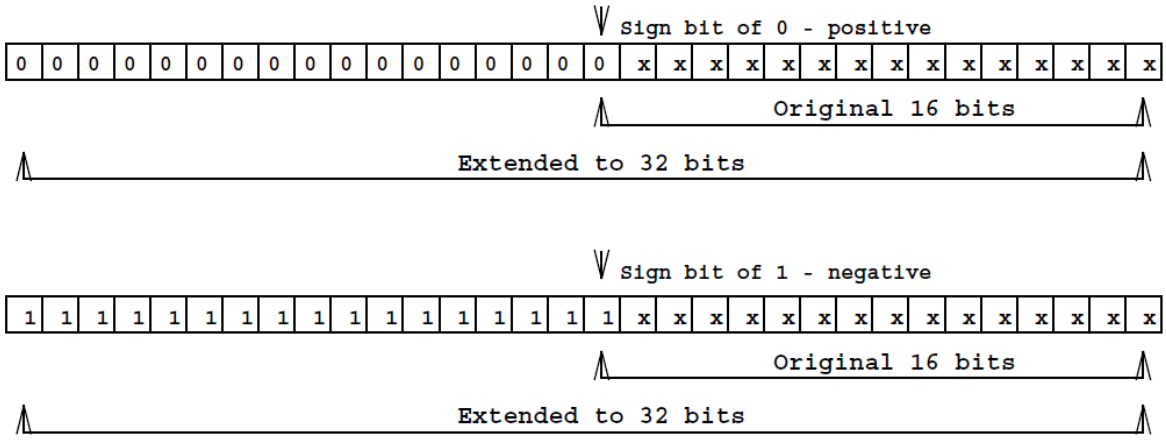
1.5. Different Size Operands and Sign Extension

As we have seen in session four, when different *sizes* (*types*) of *operands* are mixed in C expressions, the "smaller" (shorter) types are normally converted (*promoted*) to the larger type. The rules for this are clearly defined and usually we do not need to know the details of *how* this is accomplished. However, when we are interested in the individual bits in a number (e.g. when using the bit-wise operators **`&`**, **`|`** and `^`) we need to understand exactly what takes place. For example, in a bit-wise operation, if one of the operands is a **`16-bit`** number and the other is a **`32-bit`** number, then the **`16-bit`** number will be *extended* to a **`32-bit`** number before the operation takes place. If the original number is **`unsigned`** then the extension takes place by filling the *upper* **`16`** bits of the new **`32-bit`** number with zeros (**`0`**) placing the original number in the *lower* **`16`** bits as shown below:



If the original (**`16-bit`**) number is *signed* then *sign extension* takes place. In this case the *upper* **`16`** bits of the new number are filled with **`0`**s if the original number is *non-negative* (MSB - *negative weight*

bit is **0**) and with **1**s if the original number is negative (MSB - *negative weight bit* is **1**). In other words, the number is extended by *replicating* the Most Significant Bit (MSB) of the original (signed) number the required number of times (**16** in this case). This is shown below:

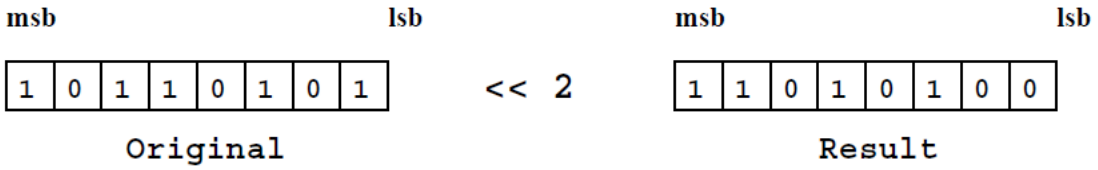


1.6. The Shift Operators (<< and **>>)**

These operators are have the effect of moving the bits in their *first operand* LEFT (**<<**) or RIGHT (**>>**) by the number of places specified in the *second* operand. For example,

```
num = num << 2;
```

shifts the bits in the variable **num** *left* (*towards the MSB*) by **2** bit positions and fills these bit positions (the two *rightmost*) with **0**'s. The 2 bits originally at the furthest left positions are lost from the number as shown below:

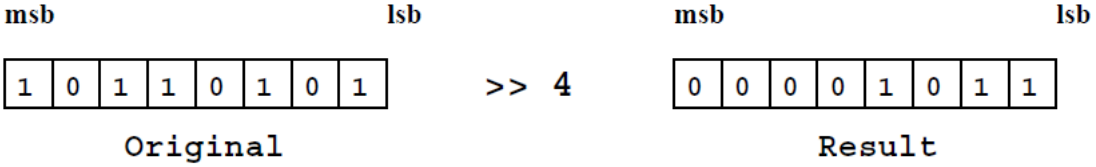


If the variable (e.g. **num**) is **unsigned** each shift **left** by *one* bit position corresponds to a multiplication by **2**. This is also true for a signed variable unless a *left* shift results in a change in the value of the leftmost bit (MSB - sign bit) whereupon the sign of the number will change!

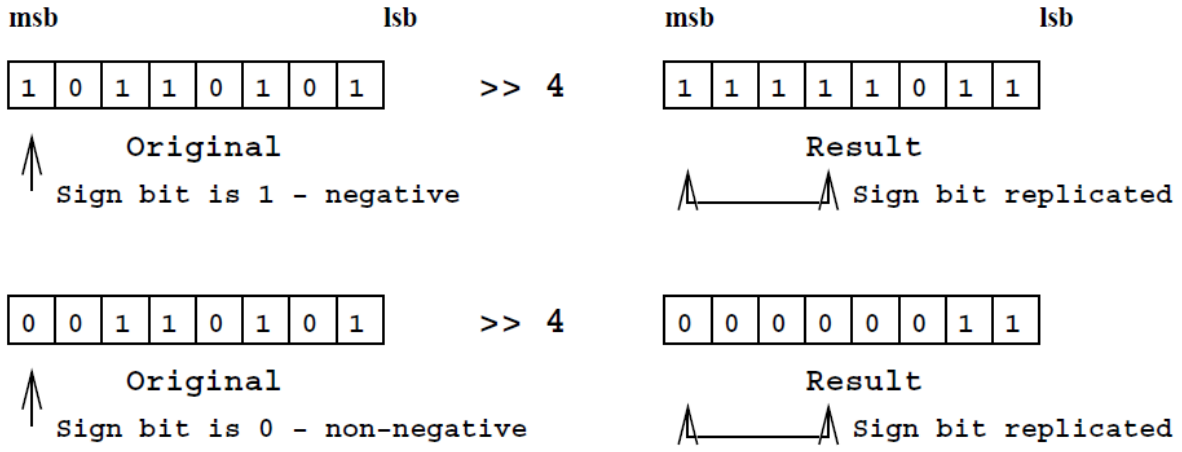
For a **RIGHT** shift, for example:

```
num = num >> 4;
```

the bits in the variable **num** are shifted *right towards the Least Significant Bit (LSB*) by **4** bit positions. If **num** is an **unsigned** variable, the *leftmost* bits are filled with **0**'s as shown below:



If **num** is a *signed* variable then *sign extension* takes place. Thus, if the MSB is **0** (a *non-negative* number), the *leftmost* bits are filled with **0**'s as in the **unsigned** case. If the MSB is **1** (**a** *negative* number), the *leftmost* bits are filled with **1**'s to preserve the sign of the number. This is illustrated below:



In both *signed* and **unsigned** cases each *right* shift by **1** bit corresponds to division by 2. Also, the *rightmost* bits are shifted off the end of the number and so are lost. The shift operators are very useful when manipulating parts of a variable. For example, we can modify Program 1 to select *only* the *upper* nibble of a byte or the upper byte of a **16**-bit word but still be able to print them out using **printf** by adding only two statements, using the right shift operator **>>**):

Program 2

```
#include <stdio.h>
#define LOW_NIBB 0xF        /* mask for ANDing to get lower nibble */
#define LOW_BYTE 0xFF         /* mask for ANDing to get lower byte */
int main(void)
{
unsigned char one_byte; /* create an unsigned char, usually 8 bits*/
unsigned short two_byte; /* create a short integer, often 16 bits */
one_byte = 0x4E;                    /* initialise value to 4E hex */
two_byte = 0x5141;                /* initialise value to 5141 hex */
one_byte = one_byte >> 4;           /* shift upper nibble to lower */
one_byte = one_byte & LOW_NIBB;       /* select only lower 4 bits */
two_byte = two_byte >> 8;           /* shift upper byte to lower */
two_byte &= LOW_BYTE;                 /* select only lower 8-bits */

printf("Upper 4 bits of 'char' = %x, Upper 8 bits of 'short' = %x
\n", one_byte, two_byte);
return 0;
}
```

This gives:

```
Upper 4 bits of 'char' = 4, Upper 8 bits of 'short' = 51
```

2. Dynamic Memory Allocation

We are, by now, very familiar with the need to *declare a variable before we make use of it*. When we declare and define a variable, the compiler and hardware reserve space for it in a particular part of the memory hardware, according to the requirements of that variable for the *size* and *persistence* of the information to be held in it.

Sometimes this reservation of memory takes place *before* execution of the program begins, and the same space is held for use by the same variable until the program ends. This is *static* allocation of memory. It is used for global variables, and others where the **static** keyword has been used. It is, generally, a poor way of using the hardware memory.

We usually declare variables as *automatic*, by putting the declaration inside a function body or parameter list (including the **`main()`** program). Automatic variables have their space reserved *dynamically*, while the program is executing. As execution of a function is started, before instructions for the executable C statements in the body of the function are executed, special instructions are executed to reserve hardware memory space for the automatic variables. At the end of the function's execution, just before the function **`return`**s, other special instructions release the hardware memory space used by that function's automatic variables. That released space is now free to be used by automatic variables in the next function called.

This process of allocating and freeing hardware memory space as a series of software functions are executed by the hardware, generally makes very good use of the hardware memory. If the creation and use of stored information in the program follows the flow of function execution, then automatic variables, with their dynamic creation and destruction, are all we need.

However, there are programs where some information *must* be created (or gathered) and stored in one function, but must also be retained long after that function has **`return`**ed. This might be dealt with by copying the data through function **`return`**s, or by use of *pointer* arguments. However, there are situations where these two methods become very cumbersome and inefficient.

You may now be thinking "Aha! A case for using global variables at last". **Think again**. The very few situations, where a global variable is the right answer, do not arise in this course. Furthermore, the *size* of the storage needed to hold the information may not be known until the program is running (after it has been compiled). In such cases, we would have to declare a size to the compiler which would as big as the biggest size which *we could ever imagine* being needed. Such a size will always end up being smaller than reality, in which case the program will terminate execution in an uncontrolled fashion (crash), or such a size will be too big for the hardware to physically support, in which case the program will not compile.

What we need is a way of allocating memory (creating space for variables) which has the *dynamic allocation* property of automatic variables, but which has *persistence of existence* more like a static variable, along with the ability to re-cycle the memory once we have finished with the information it holds.

2.1. `malloc()` and `free()`

C allows us to reserve memory for variable storage during the execution of a program by a technique called *dynamic memory allocation*. Most computer systems which support C provide an *operating system* function called **`malloc()`**, which takes one argument - the amount of space (in *bytes*) which the program requires at some point in its execution, and *returns* the *address* of some suitably allocated hardware memory, i.e. a **pointer** value for the start of the allocated memory. If **`malloc()`** cannot allocate the memory (because no more is available in the hardware) then it returns a **NULL** pointer to indicate the problem. A good programmer will arrange for the program to deal with such a problem in a graceful (controlled) fashion.

`malloc()` allocates memory in bytes (unit of **`sizeof`**) and it returns a pointer to storage type **`void`** (see session 14 notes). The value of a pointer to **`void`** can legitimately be assigned to a pointer of any other type. Indeed, it must be assigned to a non-void pointer type before it can be used to get at the allocated memory space. Since **`malloc()`** returns an address it is obvious that we must use pointer notation to access variables which are declared dynamically. Since these will usually be arrays or structures then that is quite normal.

In addition to making efficient use of memory, by allocating just the right amount of memory at the point when the program needs it, we can also *re-use* memory by *freeing* memory we no longer require. This is accomplished by the system function **`free()`**. This also takes only one argument which *must be a pointer to memory previously allocated by* **`malloc()`**.

We must not try to free memory not obtained from `malloc()`! Also, we must keep track of all allocated memory and free it when it is no longer required, otherwise the hardware memory resource will *leak* away over time. Memory leaks can be extremely hard to de-bug, it is very much easier to prevent them, by careful house-keeping (program *design*).

Thus, with **`malloc()`** and **`free()`**, we use just the right amount of hardware memory for just the period over which it is needed. This minimises the likelihood of the program failing due to the hardware resources being insufficient to support its operation in the real world.

The prototypes for the system functions **`malloc()`** and **`free()`** are declared in a *header file* called **`stdlib.h`**. This file should be specified (just like **`stdio.h`**) if we want to use these functions – and many others.

2.2. Example program

As an example, of dynamic memory allocation let us consider the structure we developed to store student records. The template for this structure is given below:

```
struct studrec        /* structure template for student record */
{
  char name[30];                  /* Name as character string */
  long matric;                /* Matriculation number as long */
  char addr[30];                    /* Term Address as string */
};
```

This structure is composed of two character arrays each of size **`30`** and one **`long`** integer. If the name or address is much shorter than **`29`** characters then parts of the arrays will contain no useful information. If a name or address is longer than **`29`** characters then we will be unable to store all of it. If we use dynamic memory allocation to reserve space for the required character arrays *after* we know the number of characters in the string to be stored then this would be much more efficient. These character arrays would no longer be stored "inside" the structure but the structure would instead contain **pointers** to the character arrays allocated to store the strings. A possible structure could then be:

```
struct stud_rec  /* new structure template for student record */
{
  char *name;      * POINTER to character string holding Name */
  long matric;               /* Matriculation number as long */
  char *addr;       /* POINTER to string holding Term Address */
};
```

```
Program 3
          /* Example program input student records and output */
#include <stdio.h>
#include <stdlib.h>             /*include library header file*/
#include <string.h>
#define MAXREC 5             /* Max number of student records */
#define LEN 80               /* maximum characters in address */
struct studrec                 /* template for student record */
{
  char *name;                             /* Name as string */
  long matric;                /* Matriculation number as long */
  char *addr;                       /* Term Address as string */
 };
```

```
int main(void)    /* read in and store records then print out */
{
 int i;
 struct studrec group[MAXREC];            /* array of records */
 void input_rec (struct studrec *);   /* input fun. Prototype */
 void output_rec(struct studrec *);  /* output fun. Prototype */
       /* read student records until structure array is full */
 printf("Enter name, matriculation No. & address on a separate
         line\n");
 for (i = 0; i < MAXREC; ++i) {/* get records of ALL students */
  input_rec(&group[i]);   /* call fun. to pass record pointer */
 }

 printf("\nNAME              MATRIC No.  TERM ADDRESS\n");
 for (i = 0; i < MAXREC; ++i) {/*print records of ALL students*/
  output_rec(&group[i]);  /* call fun. to pass record pointer */
 }
 return 0;
}
```

Function **input_rec( )** prompts, reads in a single student record, and stores it in the structure pointed to by the argument passed to the function. We will construct this function to use this new student record structure, and also to read input from the keyboard and print the prompts to the display. The operation of allocating memory (using **malloc()**) and then copying an original string into this piece of memory. We should always check if **malloc()** succeeded, by testing that it did not return **NULL**.

```
     /* Function to prompt for and read a single student record */
void input_rec(struct studrec *s_p)
{
  char temp[LEN];                    /* temp buffer to get input */
  printf("Name: ");   fflush(stdout);
  scanf("%s", temp);                              /* get name line */
  s_p->name = malloc(strlen(temp) + 1);     /* +1 byte for '\0' */
  if (NULL != s_p->name)                 /* check space is found */
    strcpy(s_p->name, temp);    /* copy string from temp buffer */

  printf("Matriculation No: ");    fflush(stdout);
  scanf("%ld", &(s_p->matric));       /* get Matriculation line */

  printf("Term Address: ");   fflush(stdout);
  scanf("%s", temp);                           /* get address line */
  s_p->addr = malloc(strlen(temp) + 1);      /* +1 byte for '\0'*/
  if (NULL != s_p->addr)
    strcpy(s_p->addr, temp);    /* copy string from temp buffer */
}
```

Notice that when we request the allocation of a piece of memory large enough to hold a string we must remember the extra byte needed to hold the terminating **'\0'** at the end of a string.

output_rec( ) is much simpler. Through structure pointer, each structure member is accessed and printed. At the end, memories requested using **malloc()** are released using **free()** function. This is all for C language part.

```
void output_rec(struct studrec *s_p)
{
 printf("%-20s%07ld%-20s\n", s_p->name, s_p->matric, s_p->addr);

 free(s_p->name);                          /* release name memory */

 free(s_p->addr);                       /* release address memory */
}
```

The End

Thank you