

Software and Embedded System Lab 2 (ELEE08022)

Bit Operation & Dynamic Memory Allocation in C language

Dr Jiabin Jia

Email: jiabin.jia@ed.ac.uk

8 bits -> 1 byte

- Computers hold information in memory as patterns of *binary logic* bits: true (**1**) or false (**0**)
- **8** bits of storage is usually referred to as one *byte*, which is the smallest chunk of memory which can be addressed
- All of the C variable types we have used so far are ***signed***
- C does allow us to add the **unsigned** keyword before the *type* name if we wish to declare an ***unsigned variable*** to store ***non-negative*** values

Use of Unsigned Variables

In `printf()` use `%u` to print *decimal value* of `unsigned int`

- use `%o` to print *bits in octal format* `%x` for hexadecimal format
- add a `#` *format modifier* (e.g. `%#x`) for a leading `0x`

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

```
int main(void)                /* use of unsigned & oct & hex */
{
    int i;
    unsigned int ui;
    char c;
    unsigned char uc;

    i  = -475;
    ui = 0xF4E7;               /* unsigned hexadecimal number */
    c  = 'a';
    uc = 045;                  /* unsigned character in octal format */
    printf("i=%d, ui=%u, c=%c, uc=%#x\n", i, ui, c, uc);
    return 0;
}
```

The output of this program is:

i=-475, ui=62695, c=a, uc=0x25

Bitwise (bit-by-bit) Operations

C bit-wise operators act on *individual bits* of *integer* type variables e.g. `int`, `long`, or `char`

Often used with `unsigned` variables, applicable to *signed* variables but forbidden for use with `float` types

Bitwise operator Function

<code>&</code>	Bit-wise AND
<code> </code>	Bit-wise OR
<code>^</code>	Bit-wise EXCLUSIVE OR
<code>~</code>	Bit-wise Inversion
<code><<</code>	Left Shift
<code>>></code>	Right Shift

Logical operator Function

<code>&&</code>	AND
<code> </code>	OR
<code>!</code>	Inversion

`172 & 153` → `136`

`10101100 & 10011001` → `10001000`

`172 && 153` → `1`

BITWISE AND Operator &

Boolean **AND** of *corresponding pairs of bits* in operands
corresponding result bit **set only if both** operand bits set
corresponding result bit **clear if either** operand bit *clear* :

bitwise AND of 10101100 and 10011001 is **10001000**

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

bitwise AND useful for:

- **clearing** (forcing to **0**) *particular* bits in a variable:
 - result of **0** AND x is ALWAYS **0**
 - result of **1** AND x is ALWAYS **x**
- hence **select** particular bits in a variable (*other bits made 0*)

Bitwise AND Operator Program

To select only the least significant (lower) **4-bits** (*nibble*) of an **8-bit** byte we can **AND** it with a *mask* value of **0x0F**. Select only least significant (lower) byte of **16-bit** number by **ANDing** it with a mask value of **0xFF**:

```
#include <stdio.h>
#define LOW_NIBB 0xF /*AND mask for lower nibble 00001111*/
#define LOW_BYTE 0xFF /* AND mask for lower byte 11111111*/
int main(void)
{
    unsigned char one_byte;      /* unsigned char - 8-bits */
    unsigned short two_byte;     /* short integer - 16-bits */
    one_byte = 0x4E;             /* initialise value to 4E hex */
    two_byte = 0x5141;           /* initialise value to 5141 hex */
    one_byte = one_byte & LOW_NIBB; /* get lower 4 bits */
    two_byte &= LOW_BYTE;         /* select only lower 8-bits */
    printf("Low 4-bits char = %x, Low 8-bits short = %x\n",
           one_byte, two_byte);

    return 0;
}
```

Low 4-bits char = E , Low 8-bits short = 41

Combination of the bit and assignment operators (eg. **&=**) is very convenient when carrying out *bitwise* operations.

BITWISE OR Operator |

Boolean **OR** of *corresponding pairs of bits* in operands
corresponding result bit **set if either** operand bit set
corresponding result bit **clear only if both** operand bits *clear* :

bitwise OR of 10101100 and 10011001 is **10111101**

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

bitwise OR useful for:

- **setting** (forcing to **1**) *particular* bits in a variable:
 - result of **1** OR x is ALWAYS **1**
 - result of **0** OR x is ALWAYS **x**

BITWISE EXCLUSIVE OR Operator ^

Boolean **XOR** of *corresponding pairs of bits* in operands
corresponding result bit **set if** operand *bits differ*
corresponding result bit **clear if** operand *bits same*:

bitwise **XOR** of 10101100 and 10011001 is **00110101**

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

bitwise **XOR** useful for:

- **complementing** (inverting) *particular* bits in a variable:
 - result of 1 XOR 1 is 0
 - result of 0 XOR 0 is 0
 - result of 1 XOR 0 is 1

BITWISE COMPLEMENT Operator ~

A *unary* operator (only one operand) ~

Boolean **NOT** (inversion) of *each bit* in the operand

bitwise **complement** of 10101100 is **01010011**

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

bitwise **COMPLEMENT** useful for constants (expressed in hex or octal) to be used with other bitwise operators:

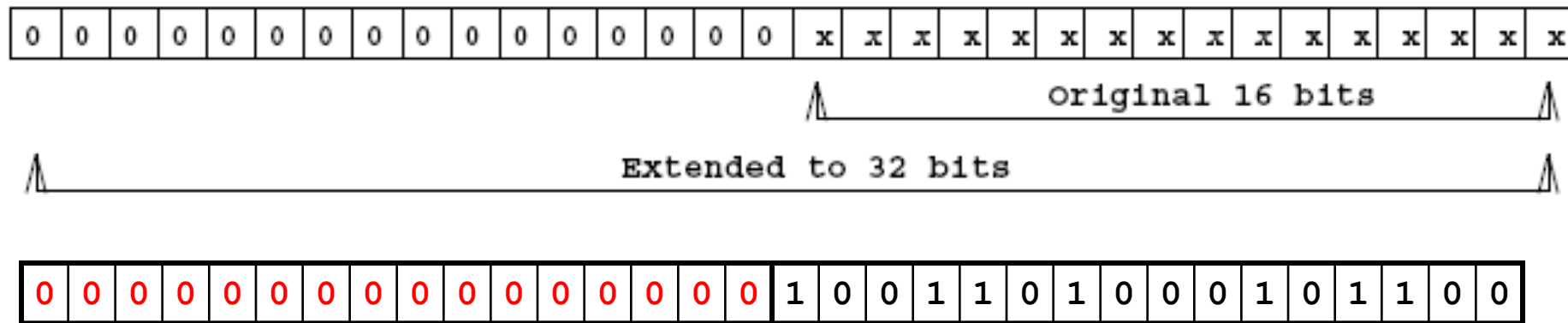
```
num1  &=  ~0xF;
```

clears *lower four bits* of num1 *irrespective of size* of num1

Compatibility of Different Size Operands - Promotion

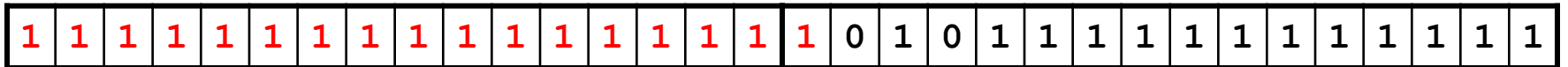
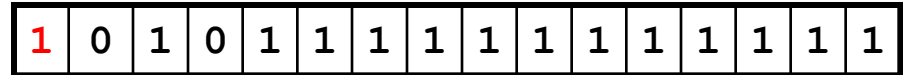
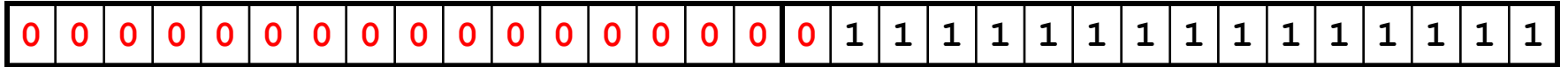
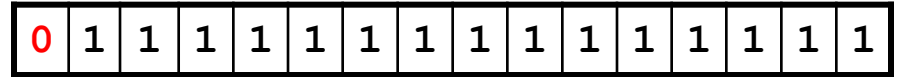
When different *sizes (types)* of *operands* are mixed, the smaller (shorter) types are converted (*promoted*) to the larger type

- if *one* operand is **16-bit** and *other* is **32-bit** then **16-bit** will be *extended* to **32-bit** *before* operation takes place
- if *original* is **unsigned** extend by filling the *upper 16* bits of new **32-bit** number with zeros (0)



Promotion of Signed Types - Sign Extension

- if original (**16-bit**) is **signed** then *sign extension* takes place
- by *replicating* the most significant bit (MSB) of the original (signed) number the required number of times (**16** in this case)

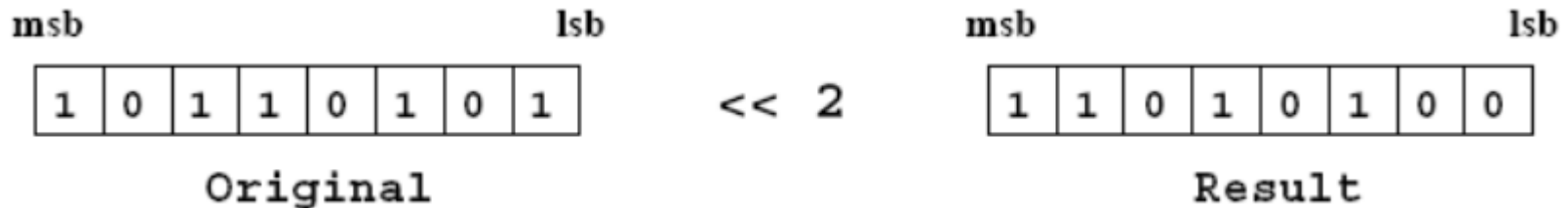


SHIFT Operators << and >>

Move bits in *first operand* **LEFT** (<<) or **RIGHT** (>>) by number of places given by *second operand*. eg.:

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

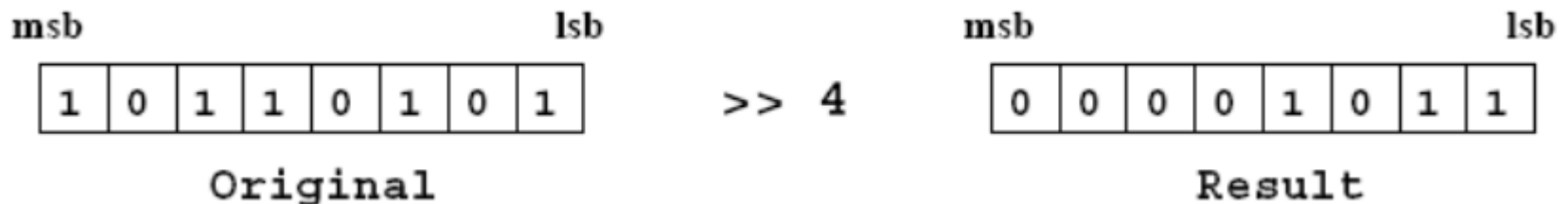
shifts bits in **num** *left* (toward MSB) by two positions and fills rightmost two bits with 0s. Two bits originally at the furthest left are lost



- For **unsigned** variable each shift **LEFT** by *one* bit is arith. doubling

For a **RIGHT** shift, eg. : **num = num >> 4;**
bits in **num** are shifted toward LSB by **four places**.

If **num** is **unsigned**, the *leftmost* bits are filled with 0s



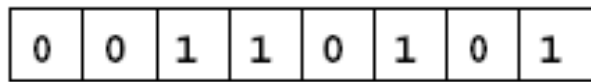
- each shift **RIGHT** by *one* bit corresponds to arithmetic division by 2

SHIFT Operators << and >>

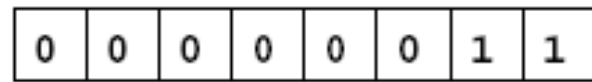
For a **RIGHT** shift, e.g. : `num = num >> 4;`
bits in **num** are shifted *right* by **4** bit positions.

If **num** is **signed** then *sign extension* takes place:

- if the MSB is **0** the *leftmost* bits are filled with **0s**



>> 4



↑
Original

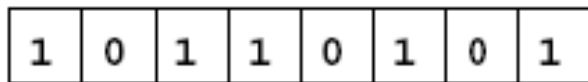
Sign bit is 0 - positive

Result

↑ ↑
Sign bit replicated

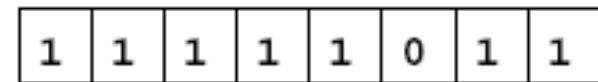
- if the MSB is **1** the *leftmost* bits are filled with **1s**

msb lsb



>> 4

msb lsb



↑
Original

Sign bit is 1 - negative

Result

↑ ↑
Sign bit replicated

Shift operators are useful when manipulating small groups of bits.
Eg modify Program 2 to select *only* the *upper* nibble of a byte or upper byte of a **16-bit** word but still be able to print out using **printf** by adding only two statements, using the right shift operator **>>**:

```
#include <stdio.h>
#define LOW_NIBB 0xF          /* AND mask for lower nibble */
#define LOW_BYTE 0xFF        /* AND mask for lower byte */
int main(void)
{
    unsigned char one_byte;    /* unsigned char - 8-bits */
    unsigned short two_byte;   /* short integer - 16-bits */

    one_byte = 0x3E;           /* initialise value to 3E hex */
    two_byte = 0x5141;         /* initialise value to 5141 hex */

    one_byte = one_byte >> 4;  /* shift upper nibble down */
    one_byte = one_byte & LOW_NIBB; /* get lower 4 bits */

    two_byte = two_byte >> 8;   /* shift upper byte down */
    two_byte &= LOW_BYTE;       /* select only lower 8-bits */
    printf("Upper 4 = %x, Upper 8 = %x \n", one_byte, two_byte);
    return 0;
}
```

Upper 4 = 3, Upper 8 = 51

Dynamic Memory Allocation

Normal variables:

we must ***declare*** the variable ***before we use it***

- For **auto** / **local** variables
 - CPU reserves memory to hold the information
 - During program execution
 - value storage persists for **FUNCTION** execution period
 - storage size variable during run time
- For **static** / **global** variables
 - Compiler reserves memory to hold the information
 - Before program starts
 - value storage persists for **PROGRAM** execution period
 - storage size set at compile time

Dynamic Memory Allocation

Dynamic memory allocation reserve memory for array or structure just as information arrives, then release/re-use storage as soon as information no longer needed

Reservation of memory for variables **during execution**

- Like an auto local variable
- take memory only when actually needed

Value storage persists for as long as we wish

- Even better than a global: space can be re-cycled

Scope restricted by programmer control (pointer passing)

- better than either global or local for program structure

Dynamic Memory Allocation

Void pointer

- A general purpose pointer
- Does not have any data type associated with
- Can store address of any type of variable

Declare a void pointer: **void * pointer_name**

```
void *ptr;
```

```
char cnum; int inum; float fnum;
```

```
ptr = &cnum; ptr = &inum; ptr = &fnum;
```

Library function **malloc()**

One argument - size of storage space needed, **in bytes**

Return value - start **address** of allocated memory (**pointer value**)

- return is **void pointer**, OK to assign it to other pointer type
- returns **NULL** pointer value to indicate error (request too big)

Library function **free()**

Re-cycle memory for future use when no longer required for current purpose:

- **library** function: **free(void *)**
 - *argument must be **pointer** previously supplied by **malloc()***
 - MUST NOT try to **free()** memory not obtained from **malloc()** !
- Must track all allocated memory and free it when it is no longer required.
- Otherwise memory resource will *leak away* over time.
- It is hard to debug but easier to prevent.

```

/* Example program of dynamic memory allocation */
#include <stdio.h>
#include <stdlib.h> /* include this header file to use */
int main(void) /* malloc() and free() */
{
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: "); fflush(stdout);
    scanf("%d", &n);
    ptr = malloc(n * sizeof(int)); /* address assignment */
    if(ptr == NULL)
    {
        /* if memory cannot be allocated, terminate program */
        printf("Error! memory not allocated.");
        exit (0);
    }
    for(i = 0; i < n; ++i) {
        printf("Enter elements: "); fflush(stdout);
        scanf("%d", ptr + i); /* ptr is a pointer, no & is needed */
        sum += *(ptr + i);
    }
    printf("Sum = %d", sum);
    free(ptr); /* release memory */
    return 0;
}

```

Dynamic Memory Allocation

- Prototypes for **malloc** () and **free** () are declared in the *header file* called **stdlib.h**

Consider student record structure in previous sessions:

```
struct studrec /* structure template for student record */
{
    char name[30];           /* Name as character string */
    long matric;             /* Matriculation number as long */
    char addr[50];           /* Term Address as string */
};
```

- Array space wasted when actual name or address less than 29 or 49 characters assumed as worst case
- If a name or address is longer than 29 or 49 characters, unable to store all of it

Dynamic Memory Allocation

- More efficient to use *dynamic memory allocation* to reserve space for strings *after we know the number of characters*
- Structure could contain *pointers* to character arrays
- Memory of arrays *allocated separately* from the structures

```
/* 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;         /* Matric number as long */
    char *addr;          /* Term Address as string */
};
```

`main()` program declares function prototype, prompts messages, calls input and output functions

```
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 */
    }

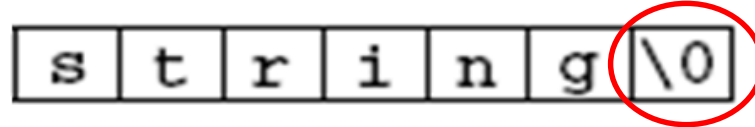
    printf("\nNAME                MATRIC No.   TERM ADDRESS\n");

    for (i = 0; i < MAXREC; ++i) { /*print records of ALL students*/
        output_rec(&group[i]);    /* call fun. to pass record pointer */
    }

    return 0;
}
```

Something you need to know

- String stored in memory as array of characters terminated by a special character **NUL** (`'\0'`) to indicate the end of the string



- Function `strlen()` returns the length of the string, excluding the NUL (`'\0'`)
- Function `strcpy(string1, string2)` copies `string2` to `string1`. `string1` must be large enough to hold `string2`;
- Include `<string.h>` header file to use string functions
- **NULL** is a special “**NO pointer**” pointer
- **NULL** can match with **any type** of pointer
- But **NULL** points to **nowhere**

```

    /* Function to prompt 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 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 for '\0' */
    if (NULL != s_p->addr)
        strcpy(s_p->addr, temp); /* copy string from temp buffer */
}

```



```
                /* Function to output student record */  
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);    /* Do not forget to release name memory */  
  
    free(s_p->addr); /* Do not forget to release address memory */  
}
```

The End

Thank you!

