60-141-02 LECTURE 6: STRUCTURES, UNIONS, BIT MANIPULATION AND ENUMERATIONS

Edited by Dr. Mina Maleki

Outline

- Structures
- Unions
- bit manipulation
- enumerations

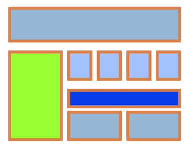
Abstract Containers and Logic

- Previously we introduced the concept and techniques related to the *array* data structure.
  - A collection of **identical** data types
    - Permits use of operations defined on the data types
  - Allocated (logically) contiguous locations in RAM
    - Permits use of relative offset to provide *direct access* to elements, based on the properties of the data types
- This raises the question: Can one create other kinds of (abstract) data structures, and types, in the C language?
  - The answer is Yes !

Structures

Structures

- A Structure is a collection of related data items, possibly of different types.
- A structure is heterogeneous in that it can be composed of data of different types.
- In contrast, array is homogeneous since it can contain only data of the same type.
- Structures hold data that belong **together**.
  - Student record: student id, name, major, gender, start year, …
  - Bank account: account number, name, currency, balance, …
  - Address book: name, address, telephone number, …
- A structure type in C is called struct.



Definition 1

- Definition of a structure:

```
struct {
    <type> <identifier_list>;
    <type> <identifier_list>;
    ...
} <struct-type>;
```

Each identifier defines a member (field) of the structure.

- Each structure definition must end with a semicolon

Common Programming Error 10.1

Forgetting the semicolon that terminates a structure definition is a syntax error.

Definition 1 …

- Example

```
struct {
   int  ID ;          // ID for this circle
   float XC ;         // Centre X coordinate
   float YC ;         // Center Y coordinate
   float Radius ;     // Radius of circle
   char Colour ;      // Colour of circle
}aCircle ;
```

5 members of different types

- There is a severe limitation with respect to the example
- We can <u>only</u> define one Circle *variable* (or a list of variables)
- Variables of the structure type may be declared only in the structure definition—not in a separate declaration.
- To overcome this, we need to define a structure ***structureName***

Definition 2

- Definition of a structure:

```
struct <structName>{
    <type> <identifier_list>;
    <type> <identifier_list>;
    ...
} ;
```

Each identifier defines a member (field) of the structure.

- The <u>type</u> of structure is called struct structName

Good Programming Practice 10.1

Always provide a structure tag name when creating a structure type. The structure tag name is convenient for declaring new variables of the structure type later in the program.

Definition 2 …

- Consider the following structure definition:

```
struct card {
    char *face;
    char *suit;
};
```

- The identifier `card` is the structure tag, which names the structure definition and is used with `struct` to declare variables of the structure type—e.g., `struct card`.
- The definition of `struct card` contains members `face` and `suit`, each of type `char *`.

Example


```
struct circleStruct {
    int ID ;            // ID for this circle
    float XC ;          // Centre X coordinate
    float YC ;          // Center Y coordinate
    float Radius ;      // Radius of circle
    char Colour ;       // Colour of circle
};
```

The "circleStruct " structure has 5 members of different types.

```
struct StudentGrade{
    char Name[15];
    char Course[9];
    int Lab[5];
    int Homework[3];
    int Exam[2];
};
```

The "StudentGrade" structure has 5 members of different array types.

Structure members can be variables of the primitive data types (e.g., int, float, etc.), or aggregates, such as arrays and other structures.

Declaration

- Structure definitions do not reserve any space in memory; rather, each definition creates a new data type that's used to define variables.
- Structure variables are defined like variables of other types.
- The declaration

```
struct Date {
    int day;
    int month;
   int year;
} date1, date2;
OR  struct Date date1, date2;
```

Example

- `struct card {`
  `char *face;`
  `char *suit;`
  `} aCard, deck[ 52 ], *cardPtr;`
- `struct card aCard, deck[52],*cardPtr;`

- declares `aCard` to be a variable of type `struct card`
- declares `deck` to be an array with 52 elements of type `struct card`
- declares `cardPtr` to be a pointer to `struct card`.

Initialization

- Structures can be initialized using initializer lists as with arrays.
- To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers.
- Example:

```
struct stdInfo {
        char Name[20];
        int Id;
        char Dept[15];
        char gender;
};
```

- struct stdInfo std1= {“Allen”,12345, “COMP”,’M’};
- Fewer initializers in the list that in the structure → the remaining members are automatically initialized to 0 (or NULL is the member is a pointer).
- struct stdInfo std2 = std1;

Common Programming Error 10.2

Assigning a structure of one type to a structure of a different type is a compilation error.

Typedef

- Create *alias* name to avoid unnecessary typing using the `typedef` compiler directive
  - typedef struct CircleStruct circleTypeName ;
- And then (after defining the struct itself) use:

```
circleTypeName  aCircle3 ;              // a single struct
circleTypeName  CircleArray[ 100 ] ;    // an array of struct’s
circleTypeName  *ptrCircle ;            // a pointer to a struct
```

Structure Operations

- The only valid operations that may be performed on structures are:
  - assigning structure variables to structure variables of the same type
  - taking the address (&) of a structure variable,
  - using the `sizeof` operator to determine the size of a structure variable.
  - accessing the members of a structure
    - the structure member operator (.)—also called the dot operator
    - the structure pointer operator (->)—also called the arrow operator.

Accessing Structure Members

- To access members of structures:
  - Structure member operator (.) called dot operator

    <struct-variable>.<member_name>;
- Example:

```
struct stdInfo std1;
strcpy(std1.Name, “Allen");
std1.Id = 12345;
strcpy(std1.Dept, "COMP");
std1.gender = 'M‘;
```

```
struct stdInfo {
  char Name[20];
  int Id;
  char Dept[15];
  char gender;
};
```

Common Programming Error 10.4

Attempting to refer to a member of a structure by using only the member’s name is a syntax error.

Accessing Structure Members …

- To access members of structures:
  - Structure pointer operator (->) called arrow operator

    <struct-variable>-><member_name>;
- Example: `struct stdInfo *stdPtr;`

```
stdPtr = &std1;

strcpy(stdPtr->name,"Reza"); //change "Allen" to "Reza"

stdPtr->Id = 10 ;   // change to a 10 instead of 12345
```

Common Programming Error 10.3

Inserting space between the - and > components of the structure pointer operator (or between the components of any other multiple keystroke operator except ?:) is a syntax error.

Accessing Structure Members …

- The expression `stdPtr->Id` is equivalent to `(*stdPtr).Id` , which dereferences the pointer and accesses the member suit using the structure member operator.

  `(*stdPtr).Id = 10 ;`   // change to a 10 instead of 12345

- The parentheses are needed here because the structure member operator (.) has a higher precedence than the pointer dereferencing operator (*).

Common Programming Error 10.5

Not using parentheses when referring to a structure member that uses a pointer and the structure member operator (e.g., `*cardPtr.suit`) is a syntax error.

Example


```c
// Fig. 10.2: fig10_02.c
// Structure member operator and
// structure pointer operator
#include <stdio.h>

// card structure definition
struct card {
   char *face; // define pointer face
   char *suit; // define pointer suit
}; // end structure card

int main( void )
{
   struct card aCard; // define one struct card variable
   struct card *cardPtr; // define a pointer to a struct card

   // place strings into aCard
   aCard.face = "Ace";
   aCard.suit = "Spades";

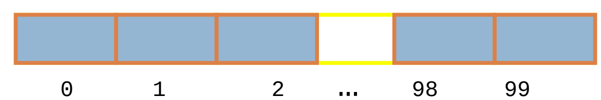
   cardPtr = &aCard; // assign address of aCard to cardPtr

   printf( "%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
      cardPtr->face, " of ", cardPtr->suit,
      ( *cardPtr ).face, " of ", ( *cardPtr ).suit );
} // end main
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Arrays of Structs

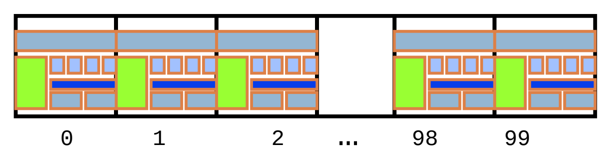
- An ordinary array: One type of data



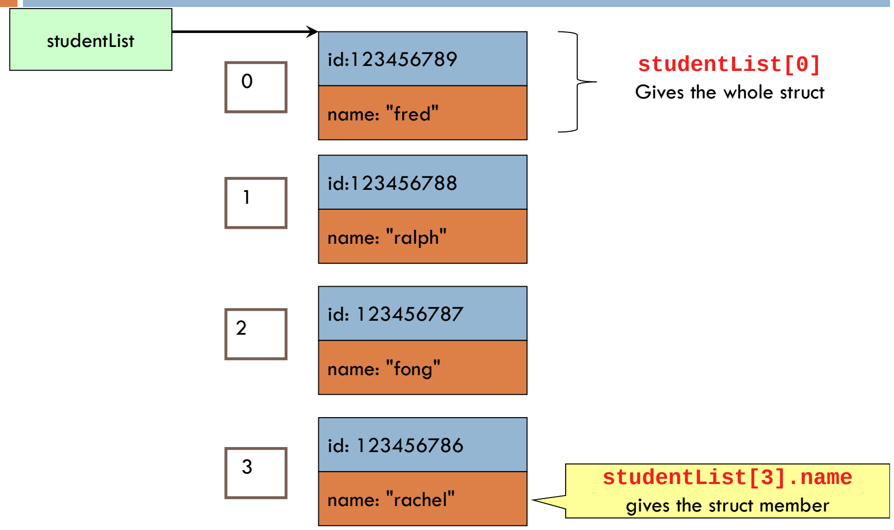
- An array of structs: Multiple types of data in each array element.
- To access a single value, you need to know which element of the array you're dealing with, *and* which member of the struct:

  `studentList[0].name`

  `studentList[i].id`



Example 1



Example 2


```
typedef struct {
      long int id;
      char name[20];
} Student;
Student std2Class[MAXCLASS];
int i;
for (i=0;i<MAXCLASS;i++) {
     printf("enter name\n");
     scanf("%s",std2Class[i].name);
     printf("enter id\n");
     scanf("%d",&(std2Class[i].id));
}
```

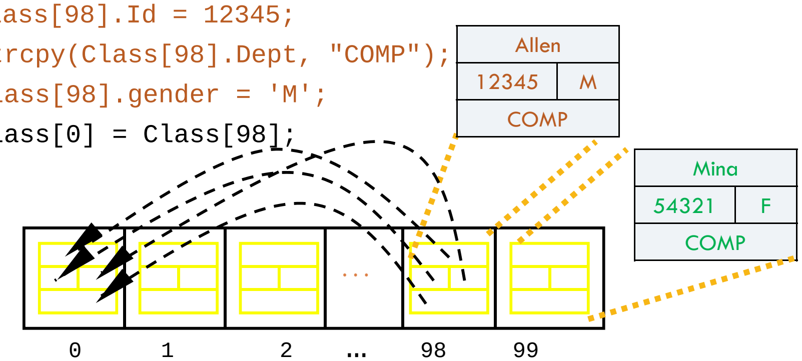
the structure at this position in the array

name of the member of the structure at that position in the array

Example 3

- We often use arrays of structures.

```
StudentRecord Class[100];
Class[99]={"Mina",54321, "COMP", 'F'};
strcpy(Class[98].Name, "Allen");
Class[98].Id = 12345;
strcpy(Class[98].Dept, "COMP");
Class[98].gender = 'M';
Class[0] = Class[98];
```



Functions and Structures

- Like any other variable, you can pass a struct as a parameter to a function
  - passing by value
  - Passing by reference

```
struct StudentRec
{
  char  lastname[MAXLEN];
  float mark;
};
```

- typedef struct StudenrRec Student;

Passing structs to Functions

25

- Pass structs by value
  - the formal parameters are copies of the actual parameters

```
void printRecord ( Student item )
{
  printf("Last name: %s\n", item.lastname);
  printf("     Mark: %.1f\n\n", item.mark);
}
```

```
main()
{
  Student studentA = {“Gauss”, 99.0};
  printRecord(studentA);
}
```

Passing structs to Functions …

26

- Pass structs by reference
  - change the value of some or all of the members
  - changes are visible in the *calling* function as well as the *called* function.
  - passing a pointer to a struct

```
void readStudent ( Student* s)
{
  printf(“Please enter name and ID\n”);
  scanf(“%s”, (*s).name) /*parens needed*/
  scanf(“%ld”, &((*s).id) ); /* Yes! */
}
```

S->name

&(S->id)

```
int main()
{
  Student studentA;
  readStudent(&studentA);
}
```

The parentheses around the (*S) are needed because of precedence

Function Returning a `struct`

27

- A “package” containing several values

```
Student readRecord ( void )
{
  Student newStudent;
  printf("Enter last name and mark: ");
  scanf("%s %f",newStudent.lastname,&(newStudent.mark));
  return newStudent;
}
```

```
main()
{
  Student studentA;
  studentA = readRecord();
}
```

Example

28

```
#include <stdio.h>
#include <stdlib.h>
#define MAXLEN  50
#define MAXN    20
struct StudentRec {
  char  lastname[MAXLEN];
  float mark;
};
typedef struct StudentRec Student;

Student readRecord ( void ){
  Student newStudent;
  printf("Enter last name and mark: ");
  scanf("%s %f", newStudent.lastname, &(newStudent.mark));
  return newStudent;
}

void printRecord ( Student item ){
  printf("Last name: %s\n", item.lastname);
  printf("     Mark: %.1f\n\n", item.mark);
}
```

Example …


```
int main(){
  int     count = 0;
  Student class[MAXN];
  int    i;
  printf("How many students? ");
  scanf("%d", &count);
  if (count > MAXN) {
    printf("Not enough space.\n");
    exit(1);
  }
  for (i=0; i < count; i++){
    class[i] = readRecord();
  }
  printf("\nClass list:\n\n");
  for (i=0; i < count; i++){
      printRecord(class[i]);
  }
  return 0;
}
```

Nested structures

- It is also possible to use struct's hierarchically, as shown in the example:
  - ```
    struct objectType {
        circleTypeName    aCircle ;
        rectangleTypeName   aRectangle ;
        ellipseTypeName    aEllipse ;
        squareTypeName   aSquare ;
    } aObject ;
    ```
  - Above, we assume that typedef's have been defined for `circleTypeName`, `rectangleTypeName`, `ellipseTypeName`, and `squareTypeName`.

Example

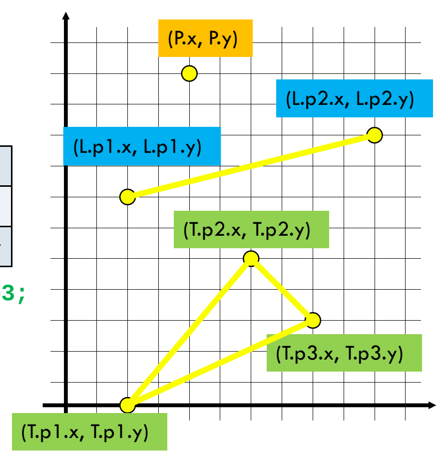

```
struct point{
    double x, y;
} P;

struct line{
    struct point p1, p2;
} L;
```

L			
p1		p2	
x	y	x	y

```
struct triangle{
    struct point p1, p2, p3;
} T;
```

T					
p1		p2		p3	
x	y	x	y	x	y



Self-Referential Structures

- One particular, important use of struct's arises when we include one or more pointer fields, including cases where the pointer points to the same struct type
  - This is called a *self-referential* data structure.
- Example:
  ```
  struct employee2 {
         char firstName[ 20 ];
         char lastName[ 20 ];
         char gender;
         double hourlySalary;
         struct employee2 person; // ERROR
         struct employee2 *ePtr; // pointer
  };
  ```
- *A structure cannot contain an instance of itself.*
  - a variable of type `struct employee` cannot be declared in the definition for `struct employee`.
  - `struct employee2` contains an instance of itself (`person`), which is an error.
- A pointer to `struct employee`, however, may be included.
  ```

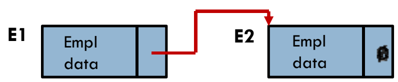
Example

33

- Example of ***self-referential*** data structure :

```
#typedef  struct Employee  Empl_t ;
#typedef  struct EmplNode  EmplNode_t ;
struct Employee {
   int ID ;
   char Fname[20] ;
   char Lname[30] ;
   float Wage ;
}
struct EmplNode {
   Empl_t  Empl ;
   EmplNode_t * ptrNext ;
}
EmplNode_t   E1, E2 ;
```

```
E1.Empl ={2548,“Roger”,“Dodger”,32.54};
E1.ptrNext = &E2 ;
E2.Empl = {2574, “Mary”,“Fleis”,32.54};
E2.ptrNext = NULL ;
```

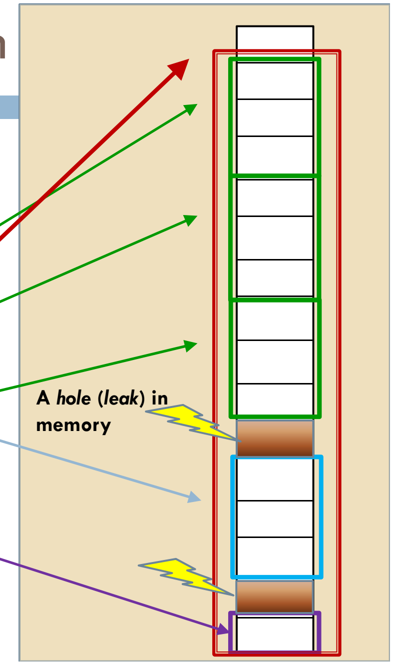


- Based on these declarations, we show how an initialization might be achieved

Memory organization

34

- The actual organization and allocation of memory (RAM) to each field, and to the struct as a whole, may not follow the conceptual definition.
  - struct {
    int ID ;
    float XC ;
    float YC ;
    float Radius ;
    char Colour ;
    } aCircle ;



Memory Organization …

35

- RAM allocation:

```
struct CircleStruct {
      int  ID ;            //  ID for this circle
      float XC ;          // Centre X coordinate
      float YC ;          // Center Y coordinate
      float Radius ;    // Radius of circle
      char Colour ;    // Colour of circle
 } aCircle ;
```

- sizeof(aCircle) >= sizeof(int) + 3*sizeof(float) + sizeof(char);
- Any difference between the left hand side and the right hand side is due to the size of all holes within the struct as it is implemented in RAM.
- Can also use the alternative syntax

`sizeof(struct CircleStruct)`

Common Mistakes

36

```
struct StudentRec
{
  char  lastname[MAXLEN];
  float mark;
};
```

Do not forget the semicolon here!

- Comparisons of two structures

```
if (studA == studB)
{
     printf(“Duplicate data.\n”);
}
```

```
if (strcmp(studA.lastname, studB.lastname) == 0
    && (studA.mark == studB.mark) )
{
     printf(“Duplicate data.\n”);
}
```

Unions

Union

- A **union** is like a **struct**, but only one of its members is stored, not all
- Memory that contains a variety of objects over time
  - It permits data of different types to be located within the same share memory space allocation
  - Only contains one data member at a time
  - Members are overlaid on top of each other
  - Only the last data member defined can be accessed
- It is *programmer's responsibility* to keep track of which type is stored in a **union** at any given time!
- Storage is big enough to hold largest member

Example


```
union TagName {
    int   N ;
    float  F ;
    double  D ;
    char * chPtr;
} aU ;
```

- `aU.N = 5 ;` // Must initialize using its first declared field type

Sizeof(union TagName) >= max(sizeof(int), sizeof(float), sizeof(double), sizeof(char*))

Union operation

- Valid **union** operations
  - Assignment to **union** of same type: **=**
  - Taking address: **&**
  - Accessing union members: **.**
  - Accessing members using pointers: **->**

Example 1


```
#include <stdio.h>
#include <string.h>
 union Data {
   int i;
   float f;
   char  str[20];
};
 int main( ) {
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming

Values of `i` and `f` members of union got corrupted because final value assigned to the variable has occupied the memory location

Example 2


```
#include <stdio.h>
#include <string.h>
 union Data {
   int i;
   float f;
   char  str[20];
};
 int main( ) {
   union Data data;
   data.i = 10;
   printf( "data.i : %d\n", data.i);
   data.f = 220.5;
   printf( "data.f : %f\n", data.f);
   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

data.i : 10
data.f : 220.500000
data.str : C Programming

Applications

- **unions** are used much less frequently than **structs**
- The **union** container is used primarily as a way of saving storage since many variables with different data types can re-use the same (roughly) allocated RAM storage
  - This is less important today than years ago due to the dramatic increases in RAM size (typically 4+GB) and lowering of costs ($/MB)
- Mostly used
  - in the inner details of operating system
  - in device drivers
  - in embedded systems where you have to access registers defined by the hardware

bit manipulation

Binary Numbers

- Representations of integer data, using bits.
  - The **unsigned int** data type has a numeric representation starting at 0 and increasing in steps of 1. In base-2, using an 8-bit container size

0	00000000	1	00000001
10	00001010	65	01000001
100	01100100	255	11111111

  - The **signed binary** type **int** : Assume an arbitrary integer, N, and we'll call the machine representation X.
    - X is formally defined (using logic) by:

X = | N | if N>=0 ,   X = ~| N | + 1 if N < 0

positive 1	0001	negative 1	1111
positive 7	0111	negative 7	1001
positive 65	01000001	Negative 65	10111111

a negative number starts with a 1 and a positive number starts with a 0.

Logic and Sequences

- Boolean concepts and operations
  - Created by George Boole
  - Bits are represented by the values 0 or 1
    - 0 represents FALSE
    - 1 represents TRUE
  - Basic operations on bits:
    - & - AND (set intersection)
    - | - inclusive OR (set union)
    - ^ - exclusive OR (set distinction)
    - ~ - Complement (set negation)

X & Y	**Y=0**	**Y=1**
X=0	0	0
X=1	0	1

X | Y	**Y=0**	**Y=1**
X=0	0	1
X=1	1	1

X ^ Y	**Y=0**	**Y=1**
X=0	0	1
X=1	1	0

~X	**~X**
X=0	1
X=1	0

Bitwise Operations

- Performing the same operations on sets of bits are referred to as *bitwise operations*.
- Assume two ordered sets of 4 bits, A and B
  - Specifically: A = 1100 and B = 1010
  - Examples:
    - A & B:

```
   A  1100
   B  1010
 A&B  1000
```

    - A | B:

```
   A  1100
   B  1010
 A|B  1110
```

    - ~A:

```
   A  1100
  ~A  0011
```

The term *bitwise operation* refers to the fact that the logic operation is carried on the bits in each corresponding position, <u>independent</u> of other bits.

A^B:

```
   A  1100
   B  1010
 A^B  0110
```

Example 1

- A0 = 00, A1 = 11, B = 10
  - Using & to *probe* or *reset* bit values
    - A1 & B : A1&B = 11 & 10 = 10 = B
    - A0 & B : A0&B = 00 & 10 = 00 All bits reset to 0
  - Using | to *probe* or *set* bit values
    - A1 | B : A1|B = 11 | 10 = 11 All bits set to 1
    - A0 | B : A0|B = 00 | 10 = 10 = B
  - Using ~ as a bit *toggle*
    - ~ B : ~B = ~ 10 = 01 Toggle each bit
  - Using ^ to *toggle* bit values
    - A1 ^ B : A1^B = 11 ^ 10 = 01 Toggle each bit
    - A0 ^ B : A0^B = 00 ^ 10 = 10 = B No effect

Example 2

Expression	Representation
a	**00000000 00000000 10000010 00110101**
b	**11111111 11111110 11010000 00101111**
a & b	
a ^ b	
a | b	
~ (a | b)	
~ a & ~ b	

Example 2 …

Expression	Representation
a	**00000000 00000000 10000010 00110101**
b	**11111111 11111110 11010000 00101111**
a & b	00000000 00000000 10000000 00100101
a ^ b	11111111 11111110 01010010 00011010
a | b	11111111 11111110 11010010 00111111
~ (a | b)	00000000 00000001 00101101 11000000
~ a & ~ b	00000000 00000001 00101101 11000000

Logic and Sequences

- Boolean Logic also admits several additional operators
  - NAND (NOT AND)
    - A nand B == not (A and B) In C: **~(A & B)**
  - NOR (NOT OR)
    - A nor B == not (A or B) In C: **~(A | B)**
  - XNOR (selective OR)
    - A xnor B == (A and B) or ((not A) and (not B)) In C: **~(A ^ B)**

Bit Manipulation-Shift

- Shift operators have two forms:
  - Left shift: << Right shift: >>
- Unsigned integers are easy to deal with
  - Left shifting: high order bits are lost + zero bits are inserted into the low order positions.
  - Right shifting: low order bits are lost + zero bits are inserted into the high order positions.
  - Examples: (Assume A = 00011010)
  - A = A << 3 ; // A = 11010000
  - A = A >> 3 ; // A = 00000011
  - A = A << 8 ; // A = 00000000

Bit Manipulation-Shift …

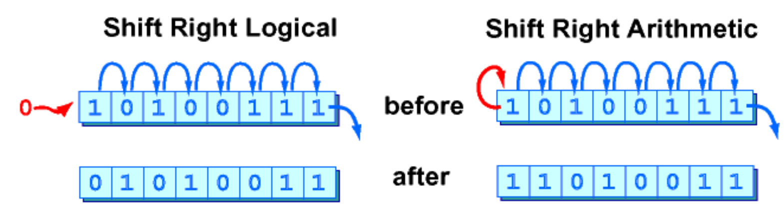
- Be careful dealing with signed binary (arithmetic shift) representations
  - The *semantic context* of negative numbers must be preserved
  - Must be careful to interpret some results
- Examples (independent statements):
  - Assume A = 00011010 (decimal 28)

    A = A << 3 ; // A = 11010000 negative ?
    A = A >> 3 ; // A = 00000011 3
    A = A << 8 ; // A = 00000000 0
  - Assume A = 11100010 (decimal -30)

    A = A << 3 ; // A = 00010000 positive ?
    A = A >> 3 ; // A = 11111100 4
    A = A << 8 ; // A = 00000000 0
    A = A >> 8 ; // A = 11111111 -1
  - Be careful in how the result is interpreted – does it make sense in the context of the operation, and the application program?

Shift …

- A **logical** right shift moves zeros into the high order bit.
  - This is desirable in some situations, but not for dividing negative integers where the high order bit is the "sign bit."
- An **arithmetic** right shift replicates the sign bit as needed to fill bit positions:

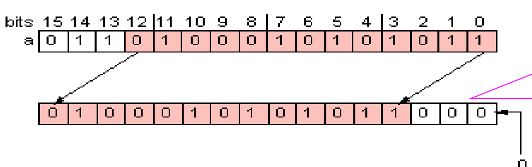


Example- Unsigned Shifts

short a = 0x68ab;

a <<= 3; /* shift left 3 bits */

Same as: a = a << 3;

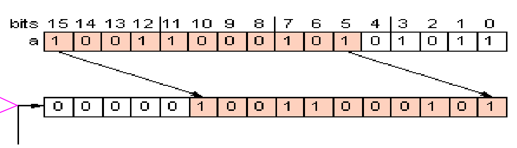


Bits positions vacated by shift are filled with zeros

unsigned short a = 0x98ab;

a >>= 5; /* shift right 5 bits */

For unsigned data type, bits positions vacated by shift are filled with zeros.

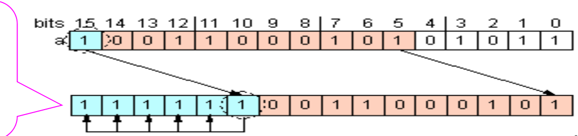


Example – Signed Shifts

short a = 0x98ab;

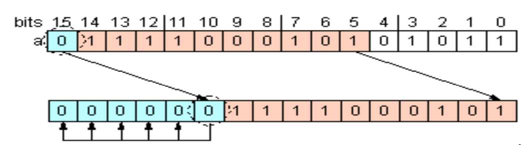
a >>= 5; /* shift right 5 bits */

Bit positions vacated by shifting is filled with a copy of the highest (sign) bit for signed data type



short a = 0x78ab;

a >>= 5; /* shift right 5 bits */



Implementation Note

57

- x<<n is equivalent to multiplication by 2^n.
- x>>n is equal to $x/2^n$

0x0001	0000000000000001
0x0002	0000000000000010
0x0004	0000000000000100
0x0008	0000000000001000

- Shifting is much faster than actual multiplication (*) or division (/)

Operator Precedence

58

(a+b << 12*a >> b)

(((a+b)<<(12*a))>>b)

Operators	Associativity
() [] . -> ++(postfix) --(postfix)	left to right
+(unary) -(unary) ++(prefix) --(prefix) ! sizeof(type) &(address) *(dereference) ~	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= &= >>= etc	right to left

Printing the bits of an integer

59

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

int main(void){
  int num=0;
  do{
     printf("Enter an integer:\n");
     scanf("%d", &num);
     bit_print(num);
     putchar('\n');
  } while (num!=0);
  return 0;
}
```

Prints the binary representation of an integer.
E.g: 00000000 00000000 00000000 00000111

Printing the bits of an integer ...

60

```
void bit_print(int a) {
  int  i;
  int  n = sizeof(int) * CHAR_BIT;  /* #define CHAR_BIT 8 (in <limits.h>)*/
  int mask = 1 << (n - 1);   /* mask = 100...0 */
  for (i = 1; i <= n; ++i) {
    putchar((a & mask)? '1' : '0');
    a <<= 1;
    if (i % CHAR_BIT == 0 && i < n)
       putchar(' ');
  }
}
```

n is the number of bits in an integer

Prints the most significant bit of a

(condition) ? (if true) : (if false)

```
if (a&mask == 1)
    putchar(`1`);
else
    putchar(`0`);
```

Prints a space between the bytes

Enumerations

Enumeration

- Enumeration of sequences
  - Provides for user defined data sequences (eg. days, months)
  - Elements are assigned integer values in a (closed) range
  - These allow for conceptual simplification of related programming tasks
- Defined by the C language statement

  enum eName { *idlist* } ;
- Examples:
  - enum Workday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
  - enum Colour { Red, Green, Blue, Violet };

Enumeration …

- Each of the symbols are assigned <u>default</u> values by the compiler, beginning with the first symbol (usually 0) and then incrementing by 1 the value of each successive symbol in the list
- This can be modified by programmers
  - enum Colour { Red=1, Green, Blue, Violet }; // Start at 1
  - enum Colour { Red=1, Green=3, Blue=4, Violet=2 }; // Redefine ordering
- Can check the values assigned to each symbol using a printf() statement.

Enumeration …

- Enumerated lists provide a self-documenting approach to useful symbols.
  - Consider the following

    ```
    enum Colour { Red=1, Green, Blue, Violet }; // Start at 1
    char ColText [ 5 ] [10 ] = { "ERROR", "Red", "Green", "Blue", "Violet" } ;

    char * ptrColour ;
    ptrColour = &ColText [ Green ] ;    // Using enumerated constant
    printf( "The colour is %s\n", ptrColour ) ; // outputs the string "Green"
    ```
  - It is the ability to write code using common, intuitive terminology that makes the job easier for programmers
    - Working with numbers and pointers can be confusing in simple applications

Example


```
#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday,
           thursday,friday, saturday};
int main(){
   enum week today;
   today=wednesday;
   if ( today == wednesday)
      printf("%d day",today+1);
   return 0;
}
```

4 day

```
If ( ! strcmp(today, Wednesday))  //incorrect
If ( ! strcmp(today, “Wednesday”)) //incorrect
```

Lecture 6: Summary

- Structures
- Unions
- Bit Manipulation
- Enumerated constants
- Reading – Chapter 10
  - Review all sections and end of chapter problems.
- Reading – Chapter 11: File Processing
  - Moving beyond RAM to include data on persistent storage in the file system.
- Assignment
  - Deadline of third assignment is March. 12.
  - The sixth lab assignment has been posted on Blackboard..