**Lecture 26 - Chapter 10: C Structures, Unions, Bit Manipulation – Mon Nov 20 or Tues Nov 21**

**Announcements**

Reading:

* Chapter 10

Assignments:

* Due: Assignment #10
* Assign: Assignment #11 will be assigned after Thanksgiving break

**Today’s Goals**

1. Unions
2. Bitwise Operations
3. Bit Fields
4. Enumerations Constants
5. Anonymous Structures and Unions
6. Secure C Programming

**Today’s Terminology**

**Terminology**

* Derived Data Type
  + Created from one or more types of basic data types
* Union
  + Collection of related variables under one name, but only one variable at a time contains a value
  + Derived data type
* Bitwise Operators
  + & - AND
  + | - OR
  + ^ - Exclusive OR
* Bit Fields
  + Number of bits in which an unsigned or signed integral member of a structure or union is stored
* Enumeration
  + Set of integer constants represented by identifiers

**Unions**

**Purpose**

* Derived data type
* Like structure - a **union** will allow us to create a collection of elements of **different** types
* Unlike structure – members share same memory space so **only one value** can be stored at a time!

**Why Use Unions**

* Provides a way to manipulate different kinds of data in a single area of storage
* Used in programs where certain variables are relevant in some situations while others are not relevant
* Saves space if there is a group of data where only one is used at a time
* Examples
  + Libraries that return different data – like a stack library that could return an int or float
  + A message protocol where different sized messages are sent and received

**Defining Unions**

* When defining a union, you are defining
  + The **name** of the union
  + The **members** the union contains
* General form for defining a union

**Keywor**d to define union

**union** unionName **{**

**datatype** variable**;**

**datatype** variable; **Members** **-** different values stored in the union

**};** (int, float, array, even another structure)

**Example**

* Define a union that stores a number as both an int and a double.

**Number** – names the union

**union** number { Used to declare variables

**int** integerValue;

**double** doubleValue; **Members**

}; integerValue, doubleValue

Only one can be referenced at a time

**Derived Data Type**

Constructed from one or more types

This definition **did not** reserve any memory

This definition **did** create a NEW data type – we can now define variables of this NEW type!

**Including Unions in Code**

* Same as structures!

**union** number {

**int** integerValue;

**double** doubleValue;

};

**int** **main**(**void**) {

}

**Even better is to use typedef**

**typedef** **union** {

**int** integerValue;

**double** doubleValue;

} NumberUnion;

**Defining Variables of Union Types**

Note**:** variable someUnion will be large enough

**union** number someUnion; to hold the largest type

NumberUnion someUnion;

**Union Default Values**

* Like other types, when a ***union*** is created, it is filled with garbage values NOT zero!

**int** **main**(void) {

NumberUnion someUnion;

**printf** ("IntegerValue: %d\n", someUnion.integerValue);

**printf** ("DoubleValue: %.2f\n", someUnion.doubleValue); If print these see

} **garbage values**

**Displays (varies on each system)**

IntegerValue: 6881280 **What you see may**

DoubleValue: 0.00 **be different!**

Note: if you declare someUnion as a global variable then the union is

initialized to default values! But we know defining global variables is

bad practice so we won’t be doing that!

**Union Initializers**

* Used initializer lists with arrays and structures

**int** numberList[5] = {3, 34, 44, 81, 7};

**struct** employee aEmployee = { "Joe", "Smith", 23, 'm', 15.00 };

* Use with unions as well BUT can only initialize the 1st member!

NumberUnion someValue = {10};

This causes the value in **someValue.integerValue** to be 10

NumberUnion someValue = {10.5};

This caused a warning on my system

NumberUnion someValue = {10, 25.0};

**Notes**

* Memory is allocated for the union based on its largest member
* Valid operations on unions
  + Assigning one union to another union of the **same type**
  + Taking the address (&) of a union variable
  + Accessing members of a union
* Invalid operations on unions
  + Comparing one union to another union

**Accessing Union Members**

Like structure, use the dot and arrow operator

Programmer responsibility to keep track of which type is currently stored in a union!

**Example**

* Dot operator

Globally defined

**typedef** **union** {

**int** integerValue;

**double** doubleValue;

} NumberUnion;

NumberUnion someUnion; First place value into integer

someUnion.integerValue = 81; then display both values

**printf** ("integerValue: %d\n", someUnion.integerValue);

**printf** ("doubleValue: %f\n", someUnion.doubleValue);

Second place value into double

then display both values

someUnion.doubleValue = 18.17;

**printf** ("integerValue: %d\n", someUnion.integerValue);

**printf** ("doubleValue: %f\n", someUnion.doubleValue);

**Displays** ***Implementation dependent***

output! Note how int is

Place value into integer value - display both members fine in 1st but double isn’t

integerValue: 81

doubleValue: 22962160487845746000000000 … 0000000000000.00

In this case, we are using

the double member, so int

Place value into double value - display both members is messed up

integerValue: 515396076

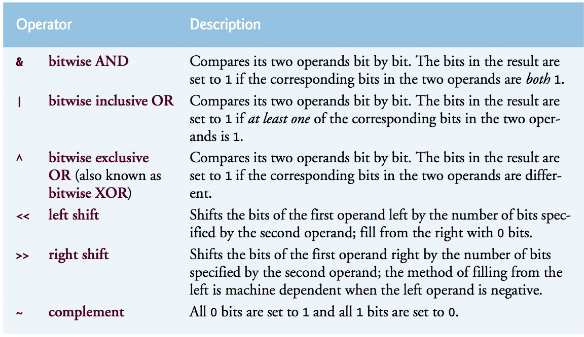
doubleValue: 18.17

**Bitwise Operations**

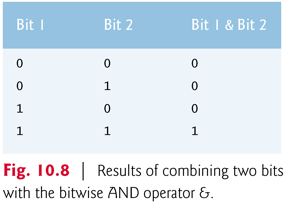
**Bitwise Operators**

* Compare two operands bit by bit
* Bitwise manipulations are machine DEPENDENT!
* Generally, don’t work at this level but there are situations where you do
  + Encryption
  + Data compression
* The operators are shown in Fig 10-6

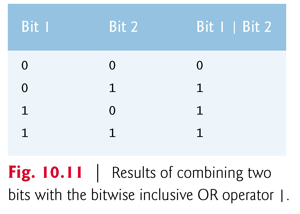
Bitwise **&, |, ^** -- compare the two operands bit by bit!



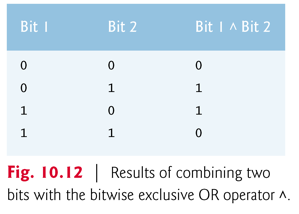
* Bitwise Operators vs Logical Operators
  + **Bitwise AND --** **Bitwise OR**
    - &, |
    - Operates at the bit level
    - Expects integral values (short, int, unsigned, char, bool, unsigned char, long)
    - Returns an integral value
  + **Logical AND** -- **Logical OR**
    - &&, ||
    - Operates at logical level
    - Expects operands that are boolean expressions
      * If operand is an integral value then in C,
        + zero is considered as false,
        + non-zero is considered as true
    - Returns a boolean value
* Bitwise AND



* Bitwise OR



* Exclusive OR



* Left Shift
  + There is code in the Fig. 10-7 ( I added the printf)

Left shift operator

**unsigned** **int** displayMask = 1 << 31;

**printf** ("displayMask = %u\n", displayMask);

231 230 229 228 …… 215 214 213 212 211 210 29 28  27 26 25 24 23 22 21 20

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

**displayMask 1 << 31**

**Shifts the bits of 1 left by 31 bits**

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (**displayMask 1 << 31**)

**Displays**

displayMask = 2147483648

Displayed as **%u** – 2,147,483,648 (this equals 231 = 2,147,483,648 which is this)

* + What happens to bits on the left that are shifted off the end?

**unsigned** **int** displayMask = 1 << 32;

**printf** ("displayMask = %u\n", displayMask);

231 230 229 228 …… 215 214 213 212 211 210 29 28  27 26 25 24 23 22 21 20

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

**displayMask = 1 << 32**

**Shifts the bits of 1 left by 32 bits – so bit was shifted off the end and is gone**

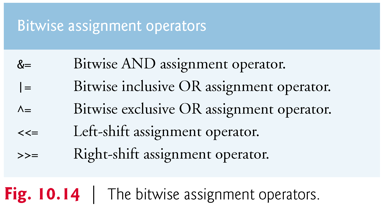
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (**displayMask 1 << 32**)

**Displays**

displayMask = 0

**Bitwise Assignment Operators**

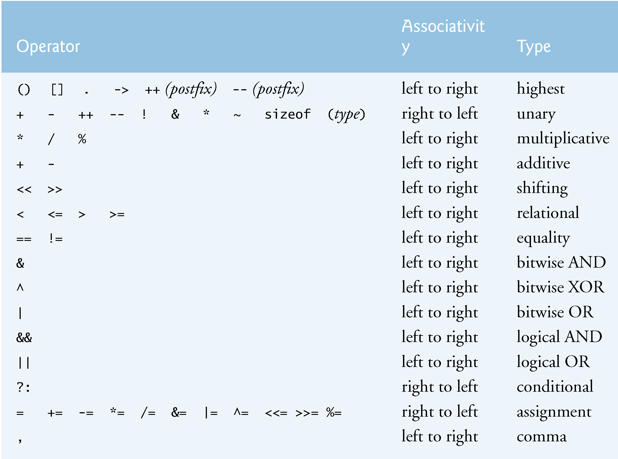
* Each bitwise operator has a corresponding bitwise assignment operator
* Fig. 10-14 shows the assignment operators for each bitwise operator



**Operator Precedence**

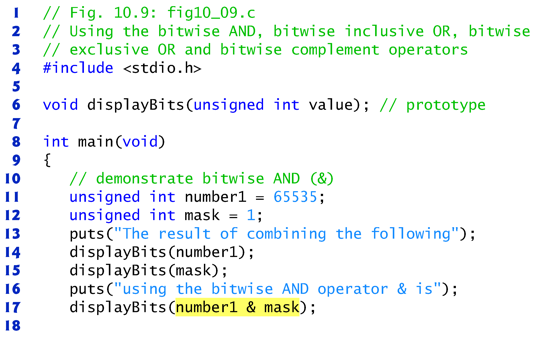
* Fig 10-15 shows the operator precedence for all the operators we have discussed

Dot operator Address operator!



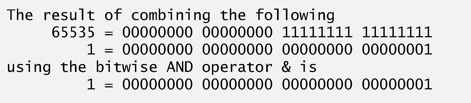
**Example**

* Example of using what is called a **mask**
* The following is from Fig 10-9



A **mask** is used to set specific bits to 1

This mask will set the bit in position 0 to 1



**Notes**

* Don’t use the bitwise operators in place of the logical operators
* The & operator can be used to check if a number is odd or even
  + (value & 1) – is non-zero if value is odd, otherwise value is even
* Using left shift and right shift operators on negative numbers results in undefined behavior

**Bit Fields**

**What is a Bit Field?**

* When you specify the number of bits for a member in a structure or union

**Why Use Bit Fields**

* Better memory utilization by storing data in the minimum number of bits required
* Avoids allocating say memory for an entire int when you only need 1 bit

**Declaring a Bit Field**

* Must be declared as int or unsigned int

**Example**

* Example from the book shows specifying the number of bits for 3 fields

Face is a bit field and its 4 bits wide

**struct** bitCard {

**unsigned** **int** face : 4;

**unsigned** **int** suit : 2; Suit is 2 bits wide and color 1-bit wide

**unsigned** **int** color : 1;

};

* With 4 bits, you can store a value from 0 to 15

23 22 21 20

8 4 2 1 8+4+2+1 = 15

Face stores 13 values – Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King

Can optimize to use only 4 bits not the full 32 bits (4 bytes) in an **unsigned int** (on my system)

**Notes**

* Bit field members are access exactly as any other structure member
* Bit field manipulations are machine dependent so need to be care if porting
* Use of bit fields save space but can generate slower executing code
  + Generates more operations!
  + Always a tradeoff between space and performance

**Enumeration Constants**

**What is an Enumeration**

* User defined type that assigns names to integral constants

**Why Use Enumerations**

* Useful when you have a longer list of items that you want to map into numeric values
* Makes programs easier to read and maintain

**Defining Enumerations**

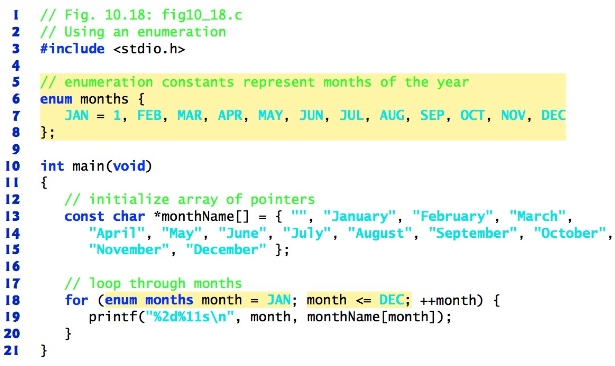
* When defining an enumeration, you are defining
  + The **name** of the enumeration
  + The **identifier** representing the values of the enumeration
* General form for defining an enumeration

**enum someEnum { identifer1, identifer2, … identifiers };**

**Example**

* Using enumeration values in a **for loop**

Note: They started the

enumeration values at 1 not 0

Array places

space in location 0

**Notes**

* Use uppercase for enumeration identifier
* Identifiers must be unique
  + This causes an error

**enum** colors {*RED*, *GREEN*, *BLUE*, *YELLOW*, *PURPLE*, *BLACK*, *ORANGE*, *BLACK*};

* If don’t assign values, compiler by default starts values at 0
* You can assign unique values to each identifier, if you don’t, complier increments by 1

**enum** colors {*FORD* = 2, *CHEVY* = 4, *TOYOTA* = 6, *VOLKSWAGEN* = 8, *MERCEDES* = 10};

**Anonymous Structures and Unions**

**Anonymous Structure and Unions**

* Unnamed structure/union
* Nest inside another structure or union
* Since not named can directly access members

**Example**

**struct** record {

**int** member1;

**int** member2;

**struct** {

**int** nestedMember1;

**int** nestedmember2;

};

};

**struct** record myRecord;

myRecord.nestedMember1 = 10;

**printf** ("%s%d\n", "The nested member is = ", myRecord.nestedMember1);

**Displays**

The nested member is = 10

**Secure C Programming**

**Secure Programming**

* To write code that uses techniques that can stand up to attacks
* This topic is an entire class so we won’t be focusing on this topic
* We will discuss some of the techniques

**CERT C Secure Coding Standard**

* CERT – Computer Emergency Response Team - [www.cert.org](http://www.cert.org)
* Publishes and promotes secure coding standards
* Standard for C
  + <https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>
* Standard for other lanaguages:
  + <https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards>

**struct**

* Size of struct is NOT necessarily the sum of its member’s sizes
* Always use sizeof to determine the number of bytes in a struct

**typedef**

* Use with derived types to create readable code!

**Bit Manipulation**

* Bitwise operations on integer types smaller than int can lead to unexpected results.
* Explicit casts are required to ensure correct results