Agenda

Unions

Introduction

- A union is a derived data type, like structure, with members that share the same storage space. For different situations in a program, some variables may not be relevant, but other variables are so a union shares the space instead of wasting storage on variables that are not being used.

- The members of a union can be of any data type. The number of bytes used to store a union must be at least enough to hold the largest member. In most cases, unions contain two or more data types. Only one member, and thus one data type, can be referenced at a time. It's your responsibility to ensure that the data in a union is referenced with the proper data type.

- A union definition has the same format as a structure definition. The union definition

```
union number{
  int x;
  double y;
}; // end union number
```

indicates that number is a union type with members int x and double y. The union definition is normally placed in a header and included in all source files that use the union type.

Unions

Initializing Unions in Declarations

- In a declaration, a union may be initialized with a value of the same type as the first union member. For example, with the union in the statement below;

```
union number value = { 10 };
```

is a valid initialization of union variable value because the union is initialized with an int, but the following declaration would truncate the floating-point part of the initializer value and normally would produce a warning from the compiler.

Unions

Initializing Unions in Declarations

- The program below uses the variable value of type union number to display the value stored in the union as both an int and a double.

```
1   // Displaying the value of a union in both member data types
2   #include <stdio.h>
3
4    union number{
5       int x;
6       double y;
7    }; // end union number
8   // function main begins program execution
9   int main( void )
10  {
11     union number value;
12     value.x = 100;
13     printf( “int:%d\tdouble:%f\n”, value.x, value.y );
14     value.y = 100.0;
15     printf( “int:%d\tdouble:%f\n”, value.x, value.y );
16     return 0;
17  } // end main
```

Bitwise Operators
Introduction

- Computers represent all data internally as sequences of bits. Each bit can assume the value 0 or the value 1. On most systems, a sequence of 8 bits forms a byte, the typical storage unit for a variable of type char. The bitwise operators are used to manipulate the bits of integral operands, both signed and unsigned.
- The bitwise operators are bitwise AND (&), bitwise inclusive OR (|), bitwise exclusive OR (^), left shift (<<), right shift (>>) and complement (~).
- The bitwise AND operator sets each bit in the result to 1 if the corresponding bit in both operands 1.
- The bitwise inclusive OR operator sets each bit in the result to 1 if corresponding bit in either (or both) operands is 1.
- The bitwise exclusive OR operator sets each bit in the result to 1 if corresponding bit in each one operand is 1.
- The left-shift operator shifts the bits of its left operand to the left by the number of bits specified in its right operand.
- The right-shift operator shifts the bits in its left operand to the right by the number of bits specified in its right operand.
- The bitwise complement operator sets all 0 bits in its operand to 1 in the result and sets all 1 bits to 0 in the result.

Bitwise Operators

Displaying an Unsigned Integer in Bits

- The program below prints an unsigned int in its binary representation in groups of eight bits each for readibility.

```
1   // Displaying an unsigned int in bits
2   #include <stdio.h>
3    void displayBits( unsigned int value );
4
5   int main( void )
6   {
7      unsigned int x;
8      printf( "%s", "Enter a nongenative int:" );
9      scanf( "%u", &x );
10     displayBits( x );
11     return 0;
12  } // end main
13
14  void displayBits( unsigned int value ){
15     unsigned int c;
16     unsinged int displayMask = 2 << 30;
17     printf( "%10u = ", value );
18     for( c = 1; c <= 32; c++ ){
19        putchar( value & displayMask ? '1' : '0' ); value <<= 1;
20        if( c % 8 == 0 ){
21           putchar( ' ' );
22        }
23     }
24     putchar( '\n' );
25  }
```

Bitwise Operators

Displaying an Unsigned Integer in Bits

- The program below prints an unsigned int in its binary representation in groups of eight bits each for readibility.

```
1   // Displaying an unsigned int in bits
2   #include <stdio.h>
3    void displayBits( unsigned int value );
4
5   int main( void )
6   {
7      unsigned int x = 1510;
8      unsigned int y = 3493;
9      displayBits( x );
10     displayBits( y );
11     displayBits( x & y );
12     return 0;
13  } // end main
14  void displayBits( unsigned int value ){
15     unsigned int c;
16     unsinged int displayMask = 2 << 30;
17     printf( "%10u = ", value );
18     for( c = 1; c <= 32; c++ ){
19        putchar( value & displayMask ? '1' : '0' ); value <<= 1;
20        if( c % 8 == 0 ){
21           putchar( ' ' );
22        }
23     }
24     putchar( '\n' );
25  }
```

Enumeration Constants
Introduction

- An enumeration introduced by the keyword enum, is a set of integer enumeration constants represented by identifiers. Values in an enum start with 0, unless specified otherwise, and are incremented by 1. For example;

```
enum months{
  JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}; // end enum months
```

creates a new type, enum months, in which the identifiers are set to the integers 0 to 11, respectively. To number the months 1 to 12, use the following enumeration:

```
enum months{
  JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}; // end enum months
```

- The identifiers in an enumeration must be unique. The value of each enumeration constant of an enumeration can be set explicitly in the definition by assigning a value to the identifier. Multiple members of an enumeration can have the same constant value.

Enumeration Constants
Introduction

- In the program below, the enumeration variable mont is used in a for statement for print the months of the year from the array monthName.

```
1   // Using an enumeration
2   #include <stdio.h>
3    enum months {
4     JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
5    }
6
7   int main( void )
8   {
9      enum months month;
10     const char *monthName [] = { “January”, “February”, “March”, “April”, “May”,
11     “June”, “July”, “August”, “September”, “October”, “November, ”December” }
12
13     for( month = JAN; month <= DEC; month++ )
14        printf( “%-3d\t%s\n”, month, monthName[ month ] );
15     }
16     return 0;
17
18  } // end main
```