CSE 230 Intermediate Programming in C and C++

Unions and Bit Fields

Fall 2017

Stony Brook University

Instructor: **Shebuti Rayana**

http://www3.cs.stonybrook.edu/~cse230/

Union

- Like structures, unions are derived types.
- They follow the same syntax as structures.
- But union members share storage.
- It defines a set of alternative values that may be stored in a shared portion of memory.
- The programmer is responsible for interpreting the stored values correctly.

Union: Declaration

```
union int_or_float{
    int i;
    float f;
};
```

- `union` is a keyword, `int_or_float` is the union tag name, and the variables `i` and `f` are members
- Creates the derived data type `union int_or_float`. The declaration can be thought of as a template; it creates the type, but no storage is allocated.
- The tag name, along with the keyword union, can now be used to declare variables of this type.

```
union int_or_float a,b,c;
```

Union: Declaration

- After declaration compiler allocates storage for the identifiers.
- For each variable the compiler allocates a piece of storage that can accommodate the largest of the specified members.
- The notation to access a member is identical to structure.

Example

```
typedef union int_or_float {
        int i;
        float f;
} number;

int main (void){
        number n;
        n.i = 4444;
        printf("i: %10d   f: %16.10e\n",n.i, n.f);
        n.f = 4444.0;
        printf("i: %10d   f: %16.10e\n",n.i, n.f);
        return 0;
}
```

causes the system to allocate 4 byte for n.

```
Output: i:       4444   f: 6.227370375e-41
        i: 1166729216   f: 4.4440000000e+03
```

Why?

Example

- The point is that the system will interpret the same stored values according to which member component is selected. It is the programmer's responsibility to choose the right one.
- Unions are used in applications that require multiple interpretations for a given piece of memory.
  - You use a union when your "thing" can be one of many different things but you use *only one at a time.*
  - You use a structure when your "thing" should be a group of other things.
- They are used to conserve storage by allowing the same space in memory to be used for a variety of types.

Union: Usage

- The members of a union can be structures or other unions, and a structure can have union members

```
struct flower{
        char *name;
        enum {red, white, blue} color;
};
```

```
struct fruit{
        char *name;
        int calories;
};
```

```
struct vegetables{
        char *name;
        int calories;
        int cooking_time; //in mins
} ;
```

```
union flower_fruit_or_vegetable{
        struct flower flw;
        struct fruit    frt;
        struct vegetables veg;
};
```

Union: Usage

- to assign a value to the member `cooking_time` of the member `veg` in the `union ffv`

```
ffv.veg.cooking_time = 7;
```

A useful example

- Networking APIs sometimes define a union for IPv4 addresses, e.g.,

```
union ipv4addr {
    unsigned address;
    char octets[4];
};
```

- Most code just wants to pass around the 32-bit integer value, but some code wants to read the individual octets (bytes). This is all doable with masking, but it's a bit easier, more self-documenting, and hence slightly safer to use a union in such a fashion.

Bit Fields

- An int or unsigned member of a structure or union can be declared to consist of a specified number of bits. Such a member is called a bit field, and the number of associated bits is called its width.
  - The width is specified by a nonnegative constant integral expression following a colon.
  - The width is at most the number of bits in a machine word.
  - Typically, bit fields are declared as consecutive members of a structure, and the compiler packs them into a minimal number of machine words.

```
struct pcard{
    unsigned pips : 4;
    unsigned suit : 2;
};
```

Bit Fields (cont.)

```
struct pcard{
    unsigned pips : 4;
    unsigned suit : 2;
};
```

- A variable of type struct pcard has a 4-bit field called pips that is capable of storing the 16 values 0 to 15, and a 2-bit field called suit that is capable of storing the values 0, 1, 2, and 3, which can be used to represent clubs, diamonds, hearts, and spades, respectively.
- thirteen pips values and the four suit values needed for playing cards can be represented compactly with 6 bits.

Bit Fields (cont.)

```
struct pcard c;
```

- To assign to c the nine of diamonds, "ve can write

```
c.pips =  9;
c.suit = 1;
```

- Whether the compiler assigns the bits in left-to-right or right-to-left order is machine dependent.
- On a machine with 4-byte words, the declaration

```
struct abc{
    int a : 1, b : 16, c : 16;
}x;
```

- This would cause x to be stored in two words, with a and b stored in the first and c stored in the second.

Bit Fields (cont.)

- Only nonnegative values can be stored in unsigned bit fields.
- For int bit fields, what happens is system-dependent. On some systems the high-order bit in the field is treated as the sign bit.
  - In most applications, unsigned bit fields are used.
- The chief reason for using bit fields is to conserve memory.
  - On machines with 4-byte words, we can store 32 1-bit variables in a single word. Alternatively, we could use 32 char variables.
  - The amount of memory saved by using bit fields can be substantial.
- There are some restrictions:
  - Arrays of bit fields are not allowed.
  - address operator & cannot be applied to bit fields. This means that a pointer cannot be used to address a bit field directly, although use of the member access operator -> is acceptable.
- Unnamed bit fields can be used for padding and alignment purposes.
  - Suppose our machine has 4-byte words, and suppose we want to have a structure that contains six 7-bit fields with three of the bit fields in the first word and three in second.

Bit Field Alignment

```
struct small_integers {
    unsigned i1 : 7, i2 : 7, i3 : 7,
             : 11, /* align to next word */
             i4 : 7, i5 : 7, i6 : 7;
} ;
```

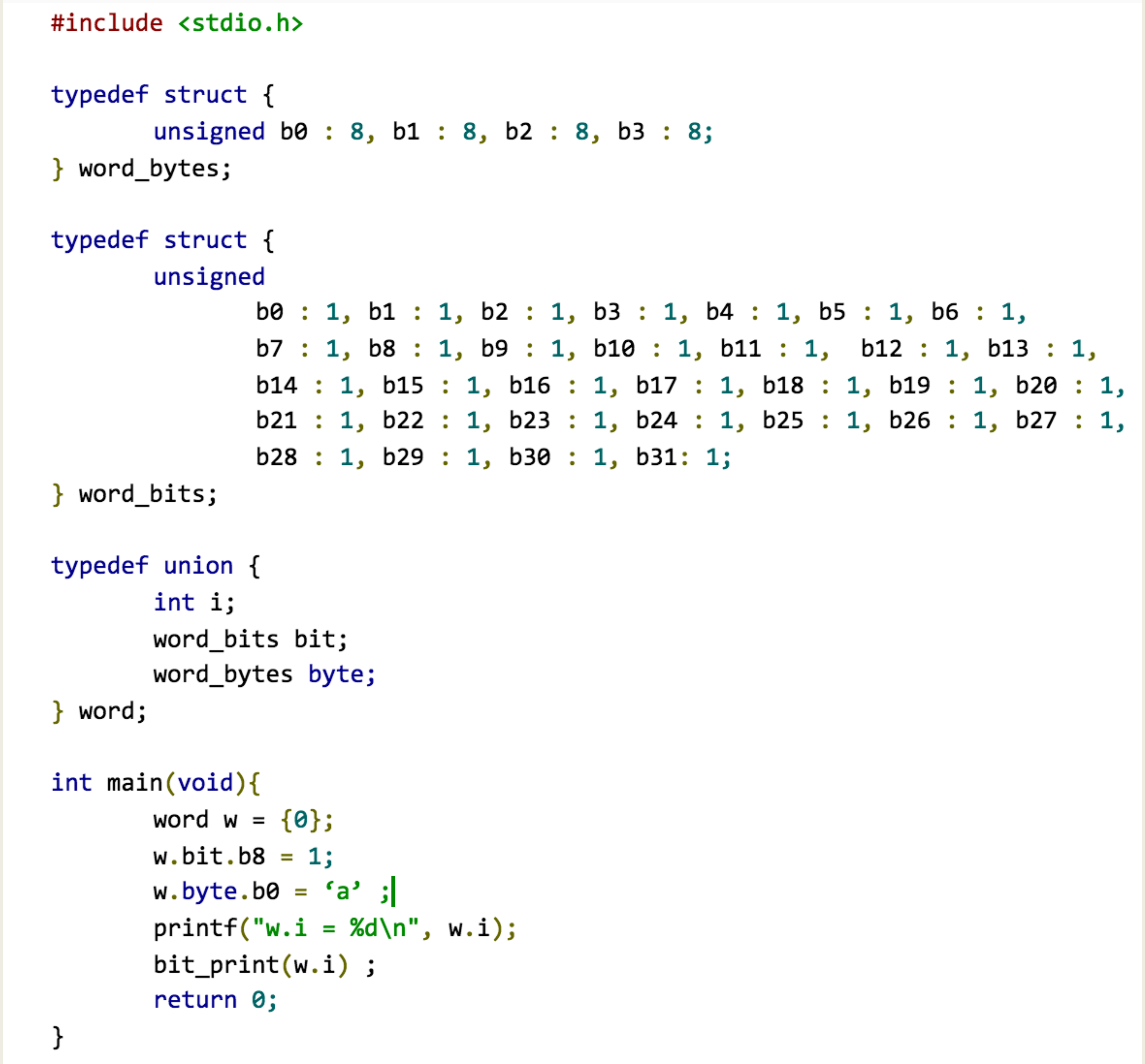
- Another way to create alignment to the next word is to use a unnamed bit field with a zero width

```
struct abc{
    unsigned a : 1, : 0, b : 1, : 0, c : 1;
};
```

- This creates three 1-bit fields in three separate words.

Example: Accessing bits and bytes

- How the bits and bytes of a word in memory can be accessed?
- Using,
  - Bitwise operators and expressions
  - Masks
  - Using bit fields



```c
#include <stdio.h>

typedef struct {
        unsigned b0 : 8, b1 : 8, b2 : 8, b3 : 8;
} word_bytes;

typedef struct {
        unsigned
                b0 : 1, b1 : 1, b2 : 1, b3 : 1, b4 : 1, b5 : 1, b6 : 1,
                b7 : 1, b8 : 1, b9 : 1, b10 : 1, b11 : 1,  b12 : 1, b13 : 1,
                b14 : 1, b15 : 1, b16 : 1, b17 : 1, b18 : 1, b19 : 1, b20 : 1,
                b21 : 1, b22 : 1, b23 : 1, b24 : 1, b25 : 1, b26 : 1, b27 : 1,
                b28 : 1, b29 : 1, b30 : 1, b31: 1;
} word_bits;

typedef union {
        int i;
        word_bits bit;
        word_bytes byte;
} word;

int main(void){
        word w = {0};
        w.bit.b8 = 1;
        w.byte.b0 = ‘a’ ;
        printf("w.i = %d\n", w.i);
        bit_print(w.i) ;
        return 0;
}
```

Example: Accessing bits and bytes

- On one machine, this program caused the following to be printed:

```
w.i = 353
```

00000000 00000000 00000001 01100001

- whereas on another machine

```
w.i 1635778560
```

01100001 10000000 00000000 00000000

- Because machines vary with respect to word size and with respect to how bits and bytes are counted, code that uses bit fields may not be portable.

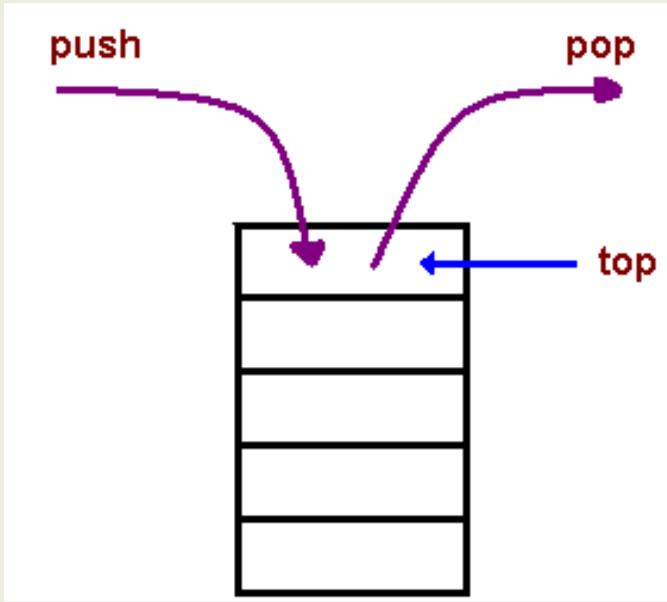
Abstract Data Type

- The term abstract data type (ADT) is used in computer science to mean a data structure together with its operations, without specifying an implementation.
  - Suppose we wanted a new integer type, one that could hold arbitrarily large values. The new integer type together with its arithmetic operations is an ADT.
- Programmer-defined types are frequently implemented with structures. For example, stack is an ADT which is one of the most useful standard data structures.

Stack

- A stack is a data structure that allows insertion and deletion of data to occur only at a single restricted element, the top of the stack. This is the last-in-first-out (LIFO) discipline.
  - a stack behaves like a pile of trays that pops up or is pushed down when trays are removed or added.
- The typical operations that can be used with a stack are push, pop, top, empty, full, and reset.

The push operator places a value on the stack
The pop operator removes a value off the stack
The top operator returns the top value from the stack
The empty operator tests if the stack is empty.
The full operator tests if the stack is full.
The reset operator clears the stack, or initializes it.
The stack, along with these operations, is a typical ADT.



Stack: Implementation

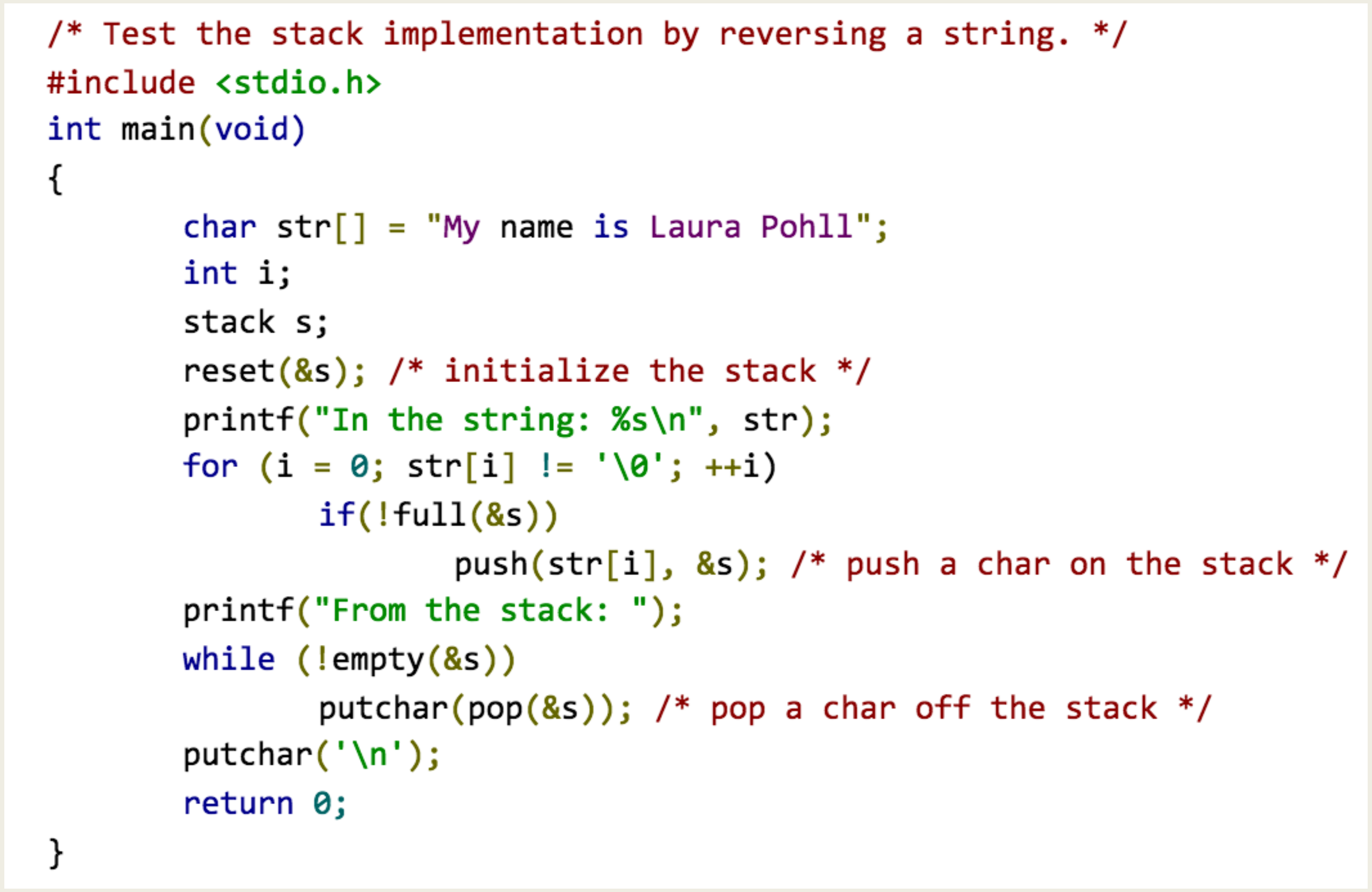
- We will use a fixed-length char array to store the contents of the stack
- The top of the stack will be an integer-valued member named top.
- The various stack operations will be implemented as functions, each of whose parameter lists includes a parameter of type pointer to stack
- By using a pointer, we avoid copying a potentially large stack to perform a simple operation.

Stack: Implementation

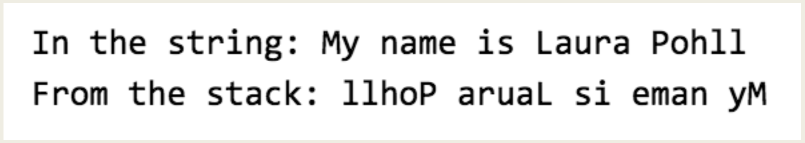
```
/* An implementation of type stack. */

#define MAX_LEN 1000
#define empty -1
#define FULL MAX_LEN-1

typedef enum boolean{false, true} boolean;

typedef struct stack{
        char s[MAX_LEN];
        int top;
}stack;

void reset(stack * stk){
        stk->top = empty;
}

void push(char c, stack * stk){
        stk->top++;
        stk->s[stk->top] = c;
}

char pop(stack *stk){
        return stk->s[stk->top--];
}
```

```
char top(const stack *stk){
        return stk->s[stk->top];
}

boolean empty(const stack *stk){
        return ((boolean)(stk->top == EMPTY));
}

boolean full(const stack *stk){
        return ((boolean)(stk->top == FULL));
}
```

Stack: Test Code to Reverse String



```c
/* Test the stack implementation by reversing a string. */
#include <stdio.h>
int main(void)
{
        char str[] = "My name is Laura Pohll";
        int i;
        stack s;
        reset(&s); /* initialize the stack */
        printf("In the string: %s\n", str);
        for (i = 0; str[i] != '\0'; ++i)
                if(!full(&s))
                        push(str[i], &s); /* push a char on the stack */
        printf("From the stack: ");
        while (!empty(&s))
                putchar(pop(&s)); /* pop a char off the stack */
        putchar('\n');
        return 0;
}
```



```
In the string: My name is Laura Pohll
From the stack: llhoP aruaL si eman yM
```