

Announcements

- Program #1A
 - Due Today
- Program #1B
 - Due in two weeks
- Reading
 - Chapter 10 (today, skip 10.2)
 - Chapter 5 (next Tuesday)

Structures

- Syntax: `struct tag { member-list } variable-list;`
- Notes:
 - used to declare related items
 - similar to classes (without methods) in Java
- Example:

```
struct {  
    int a;  
    char b;  
    float c;  
} x;
```

- Declares a single variable x with 3 fields

Structure Examples

- Named structures

```
struct SIMPLE {  
    int a;  
    int b;  
    int c;  
};
```

creates a named structure SIMPLE

declares **no** variables of type struct SIMPLE

- Using a named structure:

```
struct SIMPLE x;  
struct SIMPLE y[20], x;
```

Combining struct and typedef

- Often useful to use typedef and struct together:

```
typedef struct {  
    int a;  
    char b;  
    float c;  
} Simple;
```

- Declarations then look like:

```
Simple x;  
Simple y[20], z;
```

- Header Files

- If structure used by more than one file, put it in a header file.
- Use `#include` to include definition in each file where used.

Accessing Fields of Structures

- Consider this definition:

```
struct {  
    int a;  
    char b;  
    float c;  
} x, y[10];
```

- Simple structure access:

- x.a = 3;
- printf("field b = %c\n", x.b);

- Accessing arrays of structures:

- y[7].c = 3.14;
- y[i].b = 'a';

Initializing Entire Structures

- Consider:

```
struct {  
    int a;  
    char b;  
    float c;  
} x, y[3];
```

- Can use { } to define and entire structure

```
x = { 2, 'z', 3.14 };  
y = { { 1, 'a', 1.0 },  
      { 2, 'b', 2.0 },  
      { 3, 'c', 3.0 } };
```

Bit Fields

- Part of the Power of C is control over data layout
 - If you need a field with exactly 13 bits, you can define it
 - Useful for:
 - defining fields that interact with hardware
 - managing pre-defined file formats
 - saving every last bit of space
 - Syntax: type variable:size;

- Examples:

```
int foo:13;
unsigned int foo:4;
typedef struct {                /* from project #1 */
    unsigned int opCode:4;
    unsigned int r1:4;
    unsigned int r2:4;
    unsigned int r3:4;
    unsigned int address:16;
} instruction;
```

Passing Structures as Arguments

```
typedef struct {  
    char productName[200];  
    int quantity;  
    float unitPrice;  
} Transaction;  
  
void printReceipt( Transaction trans)  
{  
    printf("product = %s\n", trans.productName);  
    printf("  quantity = %d\n", trans.quantity);  
    printf("  price = %f\n", trans.unitPrice);  
}
```

- Uses call by value
 - Entire structure is copied to printReceipt

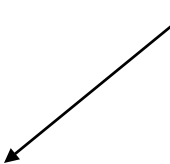
Passing a reference to a Structure

```
typedef struct {  
    char productName[200];  
    int quantity;  
    float unitPrice;  
} Transaction;
```

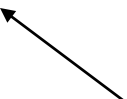
```
Transaction foo;
```

```
void printReceipt( Transaction *trans)  
{  
    printf("product = %s\n", trans->productName);  
    printf("  quantity = %d\n", trans->quantity);  
    printf("  price = %f\n", trans->unitPrice);  
}  
printReceipt(&foo);
```

Indicates reference



Indicates pass only the reference



Unions

- Syntax is similar to structs
- Rather than storing each field, only stores **one** of the fields
 - Handy when need to stores different things based on criteria
- Syntax:
 - union tag { member-list } variable-list
- Examples:

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

```
fi.f = 3.14159;
```

Variant Record Example

```
typedef struct {  
    enum { INT, FLOAT, NAME } type;  
    union {  
        int i;  
        float f;  
        char n[100];  
    } u;  
} VariableType;
```

Memory Allocation

- Structures are placed in consecutive memory locations
 - memory is rounded up to alignment
 - characters take 1 bytes
 - integers normally take 4 bytes and start at aligned addresses
- Can query the size of a structure or variable
 - sizeof(structure name) or sizeof(variable name)

0x00	0	0	0	3
0x04	?	?	?	0x61
0x08	0	0	0	0xA
0x0c				
0x10				
0x14				

```
struct {  
    int a;  
    char b;  
    int c;  
} x;  
x = { 3, 'a', 10 };
```

Project #1B

- `int getFile(char *fileName, char file[MEM_SIZE][80]);`
 - reads the passed fileName into the array file
 - returns the number of lines read
 - returns -1 on error
 - can't find file
 - too many lines
- See project web page for updates on missing info from project #1b handout.