

Announcements

- Program #1A
 - Due Today at 8pm
- Program #1B
 - Due in two weeks
 - Will be available by 8pm tonight
- Reading
 - Chapter 10 (for today, skip 10.2)
 - Chapter 5 (by next Tuesday)

Structures

- Syntax: `struct [tag] { member-list } [variable-list];`
 - `[ ]` indicate optional portions depending on the version used
 - `{ }` and `;` are required
- 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 using tag

- Named structures (using the tag)

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

creates a named structure SIMPLE

declares **no** variables of type **struct SIMPLE**

- Using a named structure to create objects of that type:

```
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.
 - If you want programmers who will be using your implementation to see this struct, put it in the 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:

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

- Like arrays you can use { } to initialize and entire structure

```
sType x = { 2, 'z', 3.14 };  
sType y[3] = { { 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
 - size must be less than or equal to the default size for that type
 - Useful for:
 - defining fields that interact with hardware
 - managing pre-defined file formats
 - saving every last bit of space
 - Syntax: **type variablename: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);  
}  
  
int main(void){  
    Transaction foo;  
    printReceipt(foo);  
}
```

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

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);  
    trans.unitPrice += 100;  
}  
  
int main(void){  
    Transaction foo;  
    printReceipt(foo);  
}
```

- Uses call by value
 - Entire structure is copied to printReceipt
 - trans.unitPrice in the caller function won't change

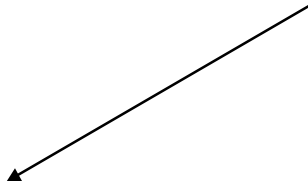
Passing a reference to a Structure

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

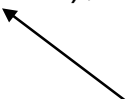
```
void initReceipt(Transaction *trans){  
    trans->productName[0] = '\0';  
    trans->quantity = 0;  
    trans->unitPrice = 12.50;  
}
```

```
int main(void){  
    Transaction foo;  
    initReceipt(&foo);  
}
```

Indicates pointer type variable



Indicates pass the reference (the address)



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];`
 - again, the things in `[ ]` are optional
 - the `{ }` and `;` are required

- Examples:

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

```
objname.f = 3.14159;
```

Variant Record Example

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

How to use it
as an individual variable:

```
varTy    var;  
var.type = INT;  
var.u.i = 5;
```

```
typedef enum {INT, FLOAT, NAME} eTy;
```

```
typedef union {  
    int i;  
    float f;  
    char n[100];  
} uType;
```

```
typedef struct{  
    eTy type;  
    uType u;  
} VarTy;
```

or in an array:

```
varTy    arr[20];  
if (arr[1].type == INT)  
    arr[1].u.i += 7;
```

Memory Allocation

- Structures are placed in consecutive memory locations
 - memory is rounded up to alignment
 - characters take 1 byte
 - 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 #1 B

- `int getFile(char *fileName, char file[MEM_SIZE][80]);`
 - reads the passed fileName into the array file
 - returns the number of lines read
 - returns a -1 on error
 - can't find the file
 - too many lines