

Announcements

- Program #1
 - Due Tuesday (2/13/07)
- Reading
 - Chapter 9,10 (Today)
 - skip 9.3
 - Chapter 5 (Tuesday)

Character Functions

- Prototypes in ctype.h
- Classifying characters
 - parameter is int, but it's a character
 - int isspace(int ch);
 - returns true if ch ' ', '\n', '\t', form feed, or carriage return
 - int isdigit(int ch);
 - returns true if its 0 through 9
 - int islower(int ch), isupper(int ch)
 - return true if it's a-z, A-Z
 - int isalpha(int ch)
 - returns true if it's a-z or A-Z
- Transformation
 - int toupper(int ch), int tolower(int ch)
 - converts to upper/lower case

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 } };
```

Abstract Data Types

- Key Idea: Hide implementation from users
 - lets implementation evolve without changing use
 - makes it easier to understand code
 - simply know what a function should do not how
- Static keyword helps with this
 - hides global variables from other modules
 - hides functions that should not be called from other modules

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

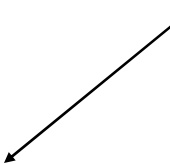
Passing a reference to a Structure

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

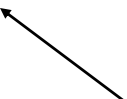
```
Transaction foo;
```

```
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 };
```

Standard I/O Library

- `#include <stdio.h>`
 - includes prototypes for standard I/O routines
- Most I/O is stream based
 - buffered to allow efficient operations
 - mostly to disks or networks
 - interactive I/O is normally flushed at useful points
 - `printf("foo\n");` /* `\n` causes a flush */
- `FILE` type
 - used to describe an active I/O connection
 - three default ones supplied on entry to main:
 - `stdin` - an input stream for default input (often keyboard)
 - `stdout` - default output stream (often terminal window)
 - `stderr` - default output stream for errors

Opening Files

- `FILE *fopen(char *name, char *mode)`
 - name specifies the name of the file to open
 - mode specifies the access mode:
 - "r" - read (file must exist)
 - "w" - write (write, existing file deleted)
 - "a" - append (write, existing file left alone)
 - Check return code:
 - A NULL return implies an error
 - use `perror(...)` to report the cause of the error

- **Example:**

```
FILE *fp;  
fp = fopen("foo", "r");  
if (!fp) {  
    perror("foo");  
    exit(-1);  
}
```

Operations on FILE

- `int fclose(FILE *fp);`
 - close a file (flushing any output if needed)
 - on success, returns 0
 - **CAN FAIL!:** On AFS (e.g., WAM) files not written to server until close.
- **Character I/O**
 - `int fgetc(FILE *stream);`
 - return the next character from input (as an int)
 - returns EOF on end of file
 - EOF doesn't fit into a char
 - check for EOF **before** assigning result to char
 - `int getchar(void);`
 - like fgetc but reads from stdin.

Character I/O (cont.)

- `int fputc(int character, FILE *stream);`
 - write character to the destination named in stream
 - return EOF on failure, 0 on success
- `int putchar(int character);`
 - like fputc but always uses stdout.
- `int ungetc(int character, FILE *stream);`
 - if you read something, and want to put it back
 - next read will get this character
 - all implementations will support at least one ungetc
 - may support more, but not guaranteed
 - moving the file pointer will discard these
 - see fseek, etc. later in this lecture

Line I/O

- Often useful to read and process an entire line
- `char *fgets(char *buffer, int bufferSize, FILE *stream);`
 - read a line into buffer (at most bufferSize-1 characters)
 - null byte added at end of buffer
 - reads from stream
 - returns NULL on error or end of file
 - on success returns buffer
- **gets: never use this function!**
 - it leads to buffer overflow problems
 - class assignments will fail if you use it
- `int fputs(char *buffer, FILE *stream);`
 - write a line of output to stream
 - returns EOF on error, non-negative value on success

Converting Input

- When you read data it is characters
- `int atoi(const char *str);`
 - Convert characters in str into an integer
 - Handles sign
 - Ignores leading white space characters
 - Ignores non-numbers after integer
- `long atol(const char *str);`
 - Converts characters in str into a long integer
 - Ignores leading white space characters
 - Ignores non-numbers after integer
- `double atof(const char *str);`
 - Converts characters in str into a double
 - Ignores leading white space characters