

File Access

- reading and writing to text files is similar to doing so from the standard input (e.g. keyboard) and standard output (e.g. screen)
- use functions from stdio library:
 - file pointer variable: FILE *fp;
 - open file: fp = fopen(name, mode); with mode either "r" for reading, "w" for writing or "a" for appending
 - read from file: fscanf(FILE *fp, char *format, ...);
 - write to file: fprintf (FILE *fp, char *format, ...);
 - close file: fclose(fp);
- an example of an abstract data type in C using an 'opaque structure'
 - client program does access fields of *fp, i.e. instead of accessing fp->..., it call stdio library to manipulate it instead it
 - it is possible to completely hide the definition of the FILE structure from the client
 - Q: why is this important / useful in software engineering?

Multi-module Programs

- putting code in different modules is useful for several reasons:
 - make a library of commonly needed functions which can then be used in other programs (e.g. stdio, math, string)
 - enable several programmers to work on the same project at once
 - make the program easier to understand (e.g. database functionality in one module, graphics into another)
 - proprietary libraries need not release source code
- e.g. MyLibrary: a C Library
 - /MyLibrary.h: contains 'headers' (declarations) for the functions which the module provides externally
 - MyLibrary.c:
 - ◆ #include "MyLibrary.h" (note: "." rather than <...> instructs the compiler to look in current directory first)
 - ◆ the compiler checks headers match implementations
 - ◆ contains the function implementations (definitions)
 - useit.c:
 - ◆ #include "MyLibrary.h" compiler checks headers match function calls
 - ◆ contains code for main program (including int main(...)) and calls to the functions provided by MyLibrary

File Examples: writefile.c and readfile.c

```
1: /* create and write text file */
2:
3: #include <stdio.h>
4:
5: int main(void) {
6:
7:     FILE *fp;
8:     int i = 42;
9:
10:    fp = fopen("myfile.txt", "w");
11:
12:    fprintf(fp, "%d_albatross!\n", i);
13:
14:    fclose(fp);
15:
16:    return 0;
17: }

1: /* read number from text file */
2:
3: #include <stdio.h>
4:
5: int main(void) {
6:
7:     FILE *fp;
8:     int i;
9:
10:    fp = fopen("myfile.txt", "r");
11:
12:    fscanf(fp, "%d", &i);
13:
14:    fclose(fp);
15:
16:    printf("%d_albatross?\n", i);
17:
18:    return 0;
19: }
```

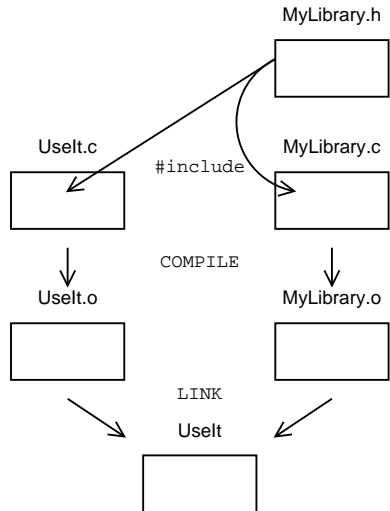
MyLibrary: Header and Implementation Files

```
1: /* useit.c */
2:
3: #include <stdio.h>
4: #include "MyLibrary.h"
5:
6: int main(void) {
7:
8:     double x = pi();
9:     display_double(x);
10:
11:    return 0;
12:
13: }

1: /* MyLibrary.h */
2:
3: double pi(void);
4: void display_double(double d);

1: /* MyLibrary.c */
2:
3: #include <stdio.h>
4: #include "MyLibrary.h"
5:
6: double pi(void) {
7:     return 3.14159;
8: }
9:
10: void display_double(double d) {
11:     printf("The_number_is:_%f\n", d);
12: }
```

Building MyLibrary into an Executable



Symbol Tables

- a table at the beginning of each object file, listing all symbols (global variables and functions) used in the module
- also lists each symbol's address within the object file, or external if it's used but not implemented in this module
- when the linker builds an executable, it resolves these external symbols (linking references to the symbol with their final address), as it adds each object file into the executable
- note:
 - each symbol must be implemented once, and only once
 - unresolved, or multiply-defined symbols cause errors

Compilation and Linking

- compilation:
 1. the C preprocessor:
 - processes #include directives
 - expands macros (#define ...)
 2. syntax and semantic checking
 - printf@17'
 - also builds the symbol tables for the modules
 3. translation into relocatable machine code, i.e. producing the object files (.o)
- linking:
 - linking is the process of combining object files into an executable file
 - the object files contain a symbol table, and relocatable machine code
 - each function used must be implemented in one, and only one, module
 - when the object files are combined, the linker links the function calls to the function implementation
 - ◆ the symbol table in each object file gives it the information needed for this

Inside Object Files (O'H&Bryant sect 7.8)

MyLibrary.o	useit.o
pi 0AEx	main 0AEx
display_double 0CCx	pi external
printf external	diplay_double external
100111000010110111001001	100100111010010101011101
100100111010011000100101	100111010100100010101010
...	...
100100111010010101011101	100100110110001000110010
001101110111 call#2 1101	11001010 call#1 00100110
110100100....	11 call#2 10101101101011
...	...

