

Various C

- assert (...)
- #define (macros)
- file access
- multi-module programs
- compilation and linking revisited
- the last word on C

assert (...)

- `assert(cond)` is used for checking that condition *cond* is true
 - useful for defensive programming, having the program check (at least in some small way) that it is doing what you expect
 - also can serve as useful documentation (expresses your intentions)
 - at best, `assert (...)` will stop your program with a meaningful error message, that gives you some clue as to where to look for the error
Assertion failed: *expression*, file *filename*, line *nnn*
Segmentation Fault: Core dumped
 - at worst, `assert (...)` makes your program run a few milliseconds slower
 - need to `#include <assert.h>` to use it

Macros

- useful for constants and simple often-used expressions

- `#define macro_name replacement_text`

- expanded by the C preprocessor:

occurrences of the macro's name are replaced by the replacement text

■ can take arguments, similar to a function

- examples:

```
#define PI 3.14159
```

```
...
```

```
area = PI * r * r;
```

```
#define max(a,b) ((a) > (b)? (a): (b))
```

```
...
```

```
printf("%f_ %d\n", max(1.0, x), max(0, i));
```

- what is difference of this from:

```
const double PI = 3.14159 ;
```

```
int max(int a, int b) {  
    return (a > b? a: b);  
}
```


File Examples: writefile.c and readfile.c

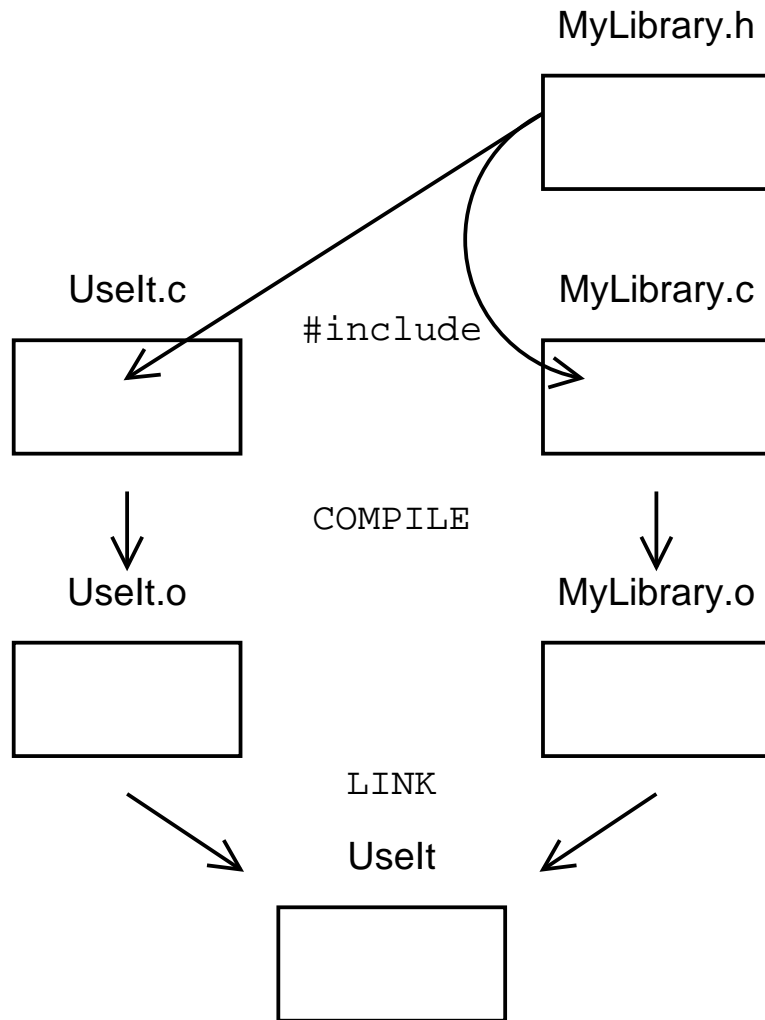
```
1:  /* create and write text file */
2:
3:  #include <stdio.h>
4:
5:  int main(void) {
6:
7:      FILE *fp;
8:      int i = 42;
9:
10:     fp = fopen("myfile.txt", "w");
11:
12:     fprintf(fp, "%d_albatross!\n", i);
13:
14:     fclose(fp);
15:
16:     return 0;
17: }
```

```
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: }
```

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

Building MyLibrary into an Executable



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

```
pi          0AEx
display_double 0CCx
printf      external
100111000010110111001001
100100111010011000100101
```

... ..

```
100100111010010101011101
001101110111 call#2 1101
110100100....
```

... ..

useit.o

```
main          0AEx
pi            external
diplay_double external
100100111010010101011101
100111010100100010101010
```

... ..

```
100100110110001000110010
11001010 call#1 00100110
11 call#2 10101101101011
```

... ..

Last Word on C

- much more to the C language than we've seen
 - what have we missed
- C takes a week to learn but a lifetime to master
- we have covered the basics of C, and this should enable you to:
 - understand most C programs you come across
 - write simple C programs
 - list C on your resume!
- if you're programming C, a good manual or reference book is a necessity
- next week (week 4):
 - another C lab (plus chance to work on assignment)
 - PeANUt lectures will start on Thursday → bring your reading brick (PeANUt Specification)
 - Assignment 1 is due in end week 5: **start working on it NOW!!**