

Various C

- `assert(...)`
- Multi-module programs
- `#define` (Macros)
- Compilation and linking revisited
- The last word on C

Examples of `assert(...)`

```

1: #include <assert.h>
2:
3: double sqrt(double x) {
4:     double result;
5:
6:     /* Check parameter */
7:     assert(x>=0.0);
8:
9:     /* Calculate square root
10:        of x */
11:     ...
12:
13:     return result;
14: }

```

```

1: #include <assert.h>
2:
3: {
4:     ...
5:     wage = rate*hours;
6:
7:     /* Calculate tax */
8:     tax = calc_tax(wage);
9:
10:    /* Sanity check on tax */
11:    assert(tax >= 0.0);
12:    assert(tax < salary);
13:
14: }

```

`assert(...)`

- `assert(Cond)` is used for checking that condition *Cond* is true
 - This is useful for defensive programming, having the program check (at least in some small way) that it is doing what you expect
 - 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

`#define` (Macros)

- Useful for constants and simple often used code (see Afzal 18.1)
- `#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));

```

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)

5

COMP2300, 2006

Multi-module example

```

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: }

```

7

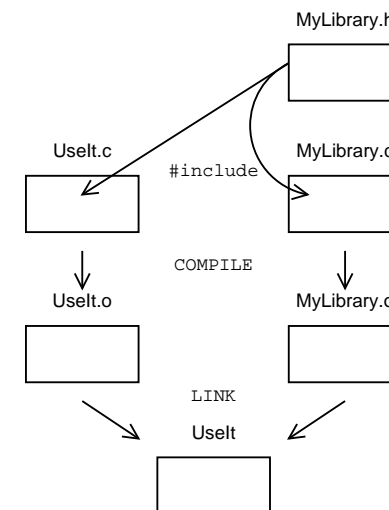
COMP2300, 2006

MyLibrary

- `MyLibrary.h`
 - Contains prototype (declarations) for the functions which the module provides to the main program (or other modules)
- `MyLibrary.c`
 - `#include "MyLibrary.h"` (note `..` rather than `<...>` instructs compiler to look in your directory first). The compiler checks prototypes match implementations
 - Contains the function implementations (definitions)
- `useit.c`
 - `#include "MyLibrary.h"`
 - Compiler checks prototypes match function calls
 - Contains code for main program (including `int main(...)`) and calls to the functions provided by `MyLibrary`

6

COMP2300, 2006



8

COMP2300, 2006

Compilation

1. Preprocessor
 - Processes `#include` directives
 - Expands macros (`#define...`)
2. Syntax and semantic checking
 - `printf@17'`
 - Also builds the **Symbol Table** for the modules
3. Translation into relocatable machine code
i.e. producing the object files (`.o`)

Symbol table

- A table at the beginning of each object file, listing all symbols (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 its implementation) as it adds each object file into the executable
- Note:
 - Each symbol must be implemented once, and only once
 - Unresolved, or multiple defined symbols cause linkage errors

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 (See Tanenbaum 7.4.2)

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

Last word on C

- Much more to the C language than we've seen
 - What have we missed from Afzal table 1.3
- 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 Monday
 - Bring your reading brick (*PeANUt specifications*)
- Assignment 1 is due in week 7 (Wednesday 5 April, 12:00 noon)
Start working on it NOW!!