## Advanced Features of C

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

- admin matters:
  - pick up your reading brick (PeANUt Specification)!
  - pick up a copy of the **Minute Paper** on D1–D3 and C1–C3; please fill in now (1st 5 mins) and leave in box on way out.
  - Assignment 1 is now ready!



## `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)
  - example: allocArray.c should check allocation succeeded (why?)

```
A = (int *) malloc(n * sizeof(int)); // allocates n elements
assert(A != null); // check for failure, e.g. n was too large
```

  - 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

    *exec*: *source*: *line*: *function*: Assertion *expression* failed
    Abort

  - at worst, `assert(...)` makes your program run a few milliseconds slower
  - need to `#include<assert.h>` to use it



## Examples of `assert(...)`

```
#include <assert.h>
double sqrt(double x) {
  double result;

  /* check parameter */
  assert(x>=0.0);

  /* calculate square root
     of x */
  ...

  return result;
}
```

```
#include <assert.h>
 {
  ...
   wage = rate*hours;

   /* calculate tax */
   tax = calcTax(wage);

   /* sanity check on tax */
   assert(tax >= 0.0);
   assert(tax < salary);

 }
```



## Macros

- useful for constants and simple often-used expressions
- **`#define macro_name replacement_text`**
- expanded by the C preprocessor (cpp):

  occurrences of the macro's name are replaced by the replacement text
  - can take arguments, similar to a function
- examples:

```
#define PI 3.14159        #define max(a,b)  ((a) > (b)? (a): (b))
...                          ...
area = PI*r*r;            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 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 not access fields of `*fp`, i.e. instead of accessing `fp->...`, it calls the `stdio` library to manipulate it instead
  - it is possible to completely hide the definition of the `FILE` structure from the client
  - Q: why is this important / useful in software engineering?



## File Examples: writeFile.c and readFile.c

```c
/* create and write text file */

#include<stdio.h>

int main(void) {

  FILE *fp;
  int i = 42;

  fp = fopen("myfile.txt", "w");

  fprintf(fp, "%d albatross!\n", i);

  fclose(fp);

  return 0;
}
```

```c
/* read number from text file */

#include<stdio.h>

int main(void) {

  FILE *fp;
  int i;

  fp = fopen("myfile.txt", "r");

  fscanf(fp, "%d", &i);

  fclose(fp);

  printf("%d albatross?\n", i);

  return 0;
}
```



## 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)
    - contains the function implementations (definitions)
    - the compiler checks that headers match implementations
  - useit.c:
    - contains code for main program (including `int main(...)` ) and calls to the functions provided by **myLibrary**
    - `#include "myLibrary.h"`: compiler simiarly can check that headers match function calls



## myLibrary: Header and Implementation Files

```c
/* useit.c */
#include <stdio.h>
#include "myLibrary.h"

int main(void) {

  float x = pi();
  displayFloat(x);

  return 0;

}
```

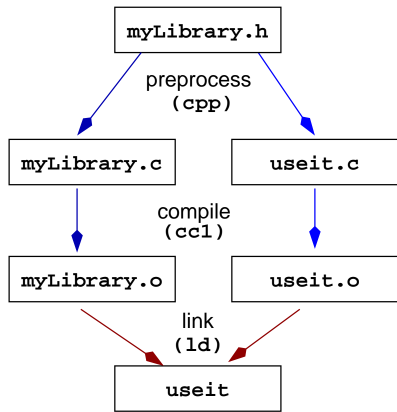
```c
/* myLibrary.h */
float pi(void);
void displayFloat(float d);

/* myLibrary.c */
#include <stdio.h>
#include "myLibrary.h"

float pi(void) {
  return 3.14159;
}

void displayFloat(float d) {
  printf("The number is: %f\n", d);
}
```

## Building `myLibrary` into an Executable




- compiled by:

```
gcc -Wall -c myLibrary.c
gcc -Wall -o useit useit.c myLibrary.o
```

- 2nd command can be separated into compile and link stages:

```
gcc -Wall -c useit.c
gcc -o useit useit.o myLibrary.o
```



## 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



## Symbol Tables and Linking

- each object file contains a symbol table listing all symbols (global variables and functions) used in the module
- also lists each symbol's address within the object file

  (or marks it as 'external' if it's not defined in this file)
- 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
  - the object files also contain a list of all places which contain a reference to an external address, together with the index of the corresp. symbol table entry
- note:
  - each symbol must be implemented once, and only once
  - unresolved, or multiply-defined symbols cause errors
- exercise: use the commands `readelf -s myLibrary.o` and `readelf -s useit.o`, noting the entries for **`pi`**, **`displayFloat`** and **`printf`**

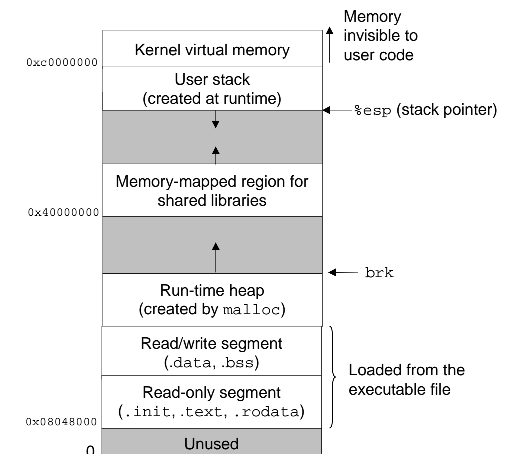
  use the commands `objdump -d myLibrary.o` and `objdump -d useit.o`



## Inside Executable Files ([O'H&Bryant, sect 7.8])

[O'H&Bryant, fig 7.13]

- constant data including string literals go into the `.rodata` segment
- global and **`static`** variables go into the `.data` and `bss` segments
  - data is valid over whole execution
- other variables local to functions go on the stack
  - data is valid only while the function is being called!
- memory areas allocated by **`malloc()`** are on the heap
  - data is valid until **`free()`** is called



## 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
  - be very careful with pointers!
- 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 next week → bring your reading brick (PeANUt Specification)