

## 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!



## 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);
}
```







# 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 similarly can check that headers match function calls





# 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:

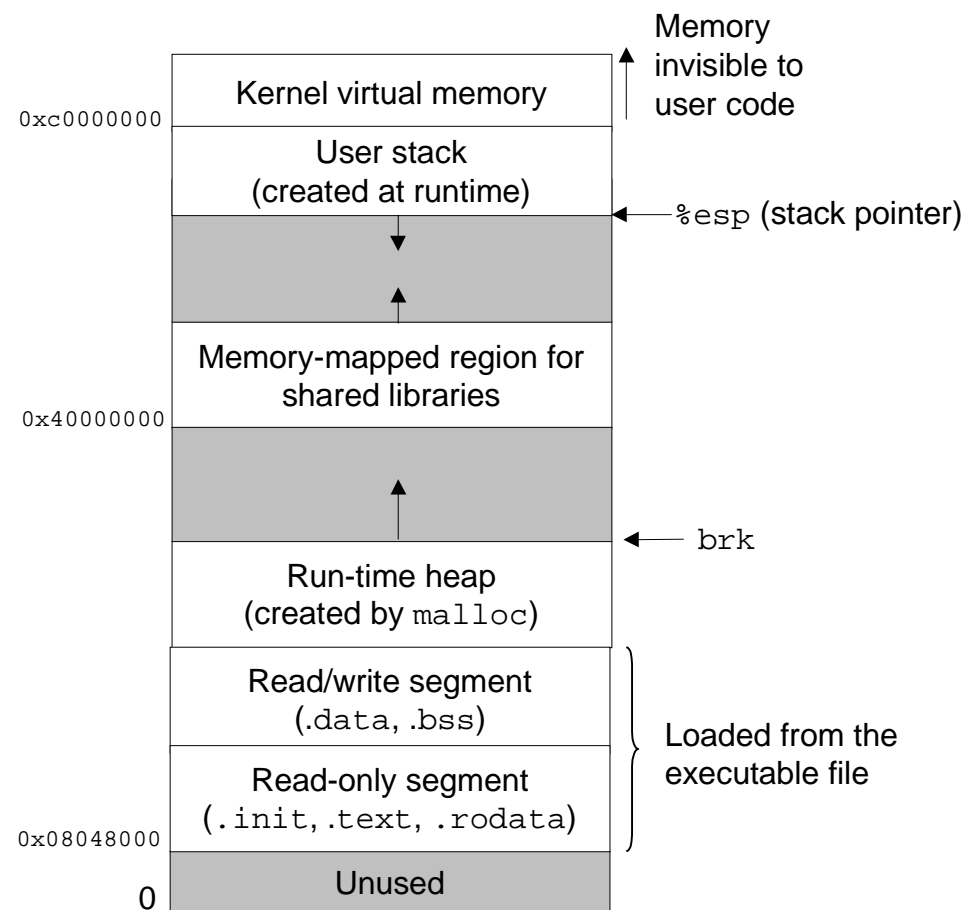
- linking is the process of combining object files into an executable file
- the object files contain a symbol table, and relocatable machine code
- each function used must be implemented in one, and only one, module
- when the object files are combined, the linker links the function calls to the function implementation
  - ◆ the symbol table in each object file gives it the information needed for this



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

- 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

[O'H&Bryant, fig 7.13]



## 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)