- **Multiple source files and the linker**
- **Intro**
  - **Real C and C++ programs have a large number of source files, not just one**
    - E.g. a typical Windows application might have something like 100 source files
  - **each file corresponds to a "module" - a piece of code that forms a logical unit of some sort**
    - consists of things that "go together"
    - can be re-used in another project as a unit
    - can be developed, tested, modified, etc by itself
  - **keeps projects much more managable.**
  - **How do you do this?**
  - **Start with sanity check on how headers and compilations work**

- **Headers, prototypes, and the linker**
  - **Intro example: scary C**
    - <beginning of file>
      ```
      int main(void)
      {
              int i = 2;
              double x;
              x = sqrt(i);
              /* what is the value of x? */
      }
      ```
  - **What's in a header file?**
    - common misconception - library code is not present, just declarations
    - mostly function prototypes - in C
      - e..g. #include math.h, get function declaration (prototype)
        - double sqrt(double);
        - means: "sqrt" is the name of a function that takes one double argument, and returns one double argument.
        - compiler can tell what needs to be pushed onto the stack, and where to find the return value
          - generates the corresponding code
    - sometimes, global variable declarations
      - objects used for I/O - like stdin, stdout (in C), cin, cout (in C++)
    - also templates in C++
      - library code is present, but as a template
  - **function prototypes**
    - Why supplied?
    - How does the compiler use a function prototype?

- **Review how the compilation process works in C and C++**
  - **start with source file, .c or .cpp**
  - **preprocessing**
    - preprocessor strips out comments
    - preprocessor looks for # lines and acts on them
    - #include filename means copy the file into this place in the source file
    - library header files have class declarations and prototypes in them
    - result is a bunch of c/c++ source code - no comments, no more # statements
    - result is a TRANSLATION UNIT - what compiler actually works on
  - **each TRANSLATION UNIT compiled completely independently of everything else**
    - once #includes done, compiler should have all the information needed to generate machine code
  - **result is object file - can find in varieous prog env.**
    - contains a notation for every function call
      - a function called "foo" called here
    - another for every function definition
      - a function named "foo" starts here
  - **libraries are standard collections of object modules that have been precompiled for you**
    - prog. env. normally knows or is told where the standard library is
  - **linker stitches these together for you**
    - assembles object modules, decides layout in memory, where everything is going to be
    - finds every function call, sticks in address of where the function will be
    - goes only by function name
    - common error: forget to define a function - linker gives you "undefined symbol" error
      - need to learn to tell the difference, because the fix is very different from undefined errors produced by the compiler
      - you called foo, but linker can't find the code for foo
      - most IDEs say "link error", but gcc says "ld" before the error message.
  - **result is executable file - application, etc. What you actually run.**
  - **to execute, loader puts in memory, makes any additional address adjustments needed**
  - **then goes to start-up code in executable, eventually calls your main function**
- **You can have your own multiple source code files**
  - **Each one usually called a MODULE**
    - contains a class or a group of related classes
    - contains a set of functions that go together - e.g. math library
  - **compile each one separately, object modules can then be linked together**
  - **rules:**
    - Must be able to compile each one separately, get separate object module
    - Linker has to know about every object module you want it to use
      - e.g. library files are usually "automatically" known to the linker

    - you tell it in various ways for your own, depending on programming environment
  - Linker has to be able to match every function call with a function definition
  - All functions that are called must be defined somewhere in the whole program, and only once in the whole program
- **how do you enable one module to call functions or use classes in another?**
  - Point from before: class declarations and function prototypes are supposed to tell the compiler everything it needs to know in order to compile code that uses the class or calls the function.
  - So idea: put declarations in a "header file" to communicate between modules.

- **Concept of header file and .c or .cpp "implementation" file**
  - **choose structs, classes, functions that make up a module**
  - **put struct or class declarations and function prototypes in the header file**
    - (also any #includes needed to process the declarations)
  - **put function definitions in the .c or .cpp or implementation file**
    - #include the .h file to bring in the declarations and prototypes
  - **in IDE, add the .c or .cpp file to the project - will get compiled along with everything else**
    - use makefile in Unix
  - **Any other module that needs the module simply #includes the header file!**
    - the header file contains all the declarations needed to enable another module to call this one
    - do not have to recompile the .c or .cpp file unless the code changes.
- **What you have to get right to make this work**
  - **The .h file has to have every declaration in it that's needed about the other module**
    - remember the compiler only knows what is in the header file - nothing else
  - **Each function definition must appear once, and only once in the whole program**
    - One-Definition Rule (ODR): it is an error to declare twice a struct or class of the same name, and it is an error to define twice a function (in C) of same name or a function of same signature (in C++)!
    - Special cases:
      - OK to have multiple declarations (prototypes) of a function as long as they agree.
        - Simply makes it a lot easier to use libraries (both standard and your own)
      - OK to have multiple declarations of global variables as long as:
        - In the whole program there is exactly one **defining declaration**, e.g.
          - int x = 0; // no extern, an initializer
        - All other declarations are **referencing declarations** with extern, e.g.
          - extern int x;
  - **The complete translation unit must not have duplicate conflicting declarations or definitions in it**
    - C/C++ compiler doesn't want to have to figure out whether two or more declarations are consistent with each other - can be a mess!
    - Makes for the only serious complication in doing multiple source files.
- **Getting duplicate declarations**
  - **easy to happen: say three modules, a, b, and c**
    - a uses b and c
    - b uses c
    - main modules needs all three
      - a.h
        ```
        #include "b.h"
        #include "c.h"
        ```
    - b.h
      ```
          #include "c.h"
      ```
    - main
      ```
          #include a.h  // brings in b.h and c.h stuff
      ```

```
#include b.h // brings in c.h
#include c.h
// uniquely c brought in three times
// uniquely b brought in two times
// uniquely a stuff only once
compiler is VERY UPSET about b and c!
```

- **Other issues problem:**
  - redundant includes
    - a.h includes b.h
    - b.h includes c.h
    - c.h includes a.h - but we've already done that!
  - Circular - infinite regression, preprocessor eventually gives up
    - a.h includes b.h
    - b.h includes c.h
    - c.h includes a.h

Preventing duplicate includes

- **First, remember you can #define a symbol**
  - #define PI 3.14
  - preprocessor does text-editor replacement - everywhere it finds "PI", changes it to "3.14"
  - in C++ we use const variables instead for this sort of thing
  - but preprocessor keeps a symbol table showing symbol and substitute value for it
    - PI   3.14
  - now you can also just define a symbol without providing a substitute value - goes into the symbol table
    - #define XYZ
    - PI 3.14
    - XYZ
  - it's a convention to make preprocessor symbols all upper case to distinguish them from real variables
- **Using a feature of the preprocessor - CONDITIONAL COMPILATION**
  - #if expression
  - stuff
  - #endif
  - if the expression is true, then stuff is LEFT IN and processed
  - if the expression is false, then stuff is LEFT OUT and ignored
  - stuff will be the declarations
  - can use this to control whether the declarations are left in are not
  - So common and important that a special form of if has been provided to use together with a preprocessor symbol:
    - #ifndef - IF NOT DEFINED
    - #ifndef XYZ - true if XYZ is not in the symbol table
  - Here's the pattern: called an INCLUDE GUARD - guards against duplicate includes

-
  ```
  #ifndef XYZ
  #define XYZ
  declarations
  #endif
  ```
- if symbol XYZ is not defined, then process everything up to the endif, otherwise IGNORE everything up to the endif
- If this is the first time the preprocessor has seen this, then
  - XYZ is not defined (not in the symbol table) so:
  - define XYZ - add XYZ to the preprocessor symbol table
  - keep the declarations in the translation unit
- if the preprocessor sees this again, then
  - XYZ is now defined (in the symbol table)
  - IGNORE the declarations
  - continue reading after the #endif
- So here is the pattern for a .h file:

  -
    ```
    #ifndef YOUR_PREPROCESSOR_SYMBOL
    #define YOUR_PREPROCESSOR_SYMBOL
    declarations
    #endif
    ```
- What to use for YOUR_PREPROCESSOR_SYMBOL?
- A good choice - something that you can consistently cook up that will be unique to each header file
- my personal choice: the header file name spelled out in all caps
  - MEETING_H
- NOTE: DO NOT USE LEADING UNDERSCORES IN PREPROCESSOR SYMBOLS
  - two leading underscores are reserved for the C/C++ implementation developers to use (or one followed by an Upper-case letter
    - implementation-specific symbols - not standardized
  - you risk a name collision - will produce an extremely hard-to-find bug
  - you risk confusing the reader that you are interacting in a non-standard way with the library code
- So, the declarations get processed by the compiler only ONCE, no matter how many times the Meeting.h file gets #included
- System header files <math.h>, <cmath> already have their own include guards, so you don't have to worry about them

- **Global variables within and between modules**
  - **A global variable is defined at the "file level" - not inside any functions**
    - its scope is the rest of the file - name is known to apply to this piece of memory in the rest of the code in the translation unit.
    - compiler/linker/loader arrange to put it somewhere where it will last for whole program execution
      - not recycled memory like stack
      - value placed in it remains as long as the program is running
      - special start-up code initializes global variables (along with all other "statically allocated" memory) before execution starts at main function
    - <beginning of file>

```
...
int gv = 3; /* iniitialzed before program starts execution */
...
void foo()
{
        printf("global value is %d\n", gv); /* gv known here */
}

int main ()
{
        foo();
        gv = 5; /* known here */
        foo();
}

etc
```

  - **now suppose we split this into modules: one with main, other with foo**
    - ODR: can only be one piece of memory for this global variable name
    - so the **defining declaration** of gv can only be in ONE of the two modules
    - we need a **referencing declaration** of it in the other module
      - We have to tell compiler whether or not we are defining the actual variable here, or just referring to it in some other module
    - main.c

```
...
extern int gv; /* declaration: gv is an int defined EXTERNally to this module */
...
int main ()
{
        foo();
        gv = 5; /* known here */
        foo();
}
```

    - Compiler notes that gv is an int, but it doesn't know where in memory it is going to be, so annotate object module and let linker match up this "gv" with the actual gv and plug in the address
    - module a.c

```
...
int gv = 3; /* definition & initialization of gv */
...
```

```
void foo()
{
	printf("global value is %d\n", gv); /* gv known here */
}
```

- The compiler sees the definition/iniitialization, and arranges to set aside memory space;
- Object module is annotated to show that there is a global variable named gv defined in this module
- Linker will match up the defined gv in module a with the named gv in main
- if you leave off the extern, then you are trying to define a second gv - violates the one-definition rule! Linker error.
  - different compiler/linkers will behave differently in C - most will merge into one definition, force you to follow the ODR even if you didn't realize it.
- if you have header files, involved, then do it this way:
  - the C++ way, works well in C.
- a.h
  ```
  etc
  extern int gv; this is a declaration only
  etc
  ```
- a.c
  ```
  #include "a.h"
  etc
  int gv = 3;  /* here is the definition - ok, matches the declaration in a.h */
  etc
  ...
  void foo()
  {
  	printf("global value is %d\n", gv); /* gv known here */
  }
  ```
- main.c
  ```
  #include "a.h"
  ...
  int main ()
  {
  	foo();
  	gv = 5; /* known here */
  	foo();
  }
  ```

- **Summary for modules**
  - **Multiple modules:**
    - As many modules as you want
    - a .h and .c or .cpp file for each one
    - main module - .c or .cpp file that has main function in it
      - often doesn't need a .h file
  - **See Handout - assigned soon - for a Guideline summary of what should be in the files.**

- **Final concept: Linkage: internal and external**
  - **Linkage means whether the symbol can be used by the linker across object modules**
    - Internal linkage - only within the module
    - external linkage - can be used across modules
  - **Functions and file-global variables normally have "external" linkage**
    - Ordinary functions are "global" - with external linkage - by default - can be called from anywhere - compiler just needs the prototype to compile the call correctly, then linker will hook it up with no further action on your part.
      - note that linker can hook it up without the prototype - prototype is for the benefit of the compiler
    - The compiler puts an annotation in the object module that these names are available outside the module for the linker to match up with calls or extern declarations.
  - **You can give them "internal" linkage if you want**
    - Changes the compiler annotation to say that the name is not available outside of the module - linker can not use it to match up with names used in other modules.
      - remember here that a module corresponds to a translation unit.
    - So if a global variable or function has internal linkage, it can only be referred to in the same source file!
    - Same concept as "private" in a class, except it applies to a source file. global, but private to this file
  - **Do it with the keyword "static" - another stupid use of this overworked keyword**
    - module a.c
      ```
      ...
      static int gv = 3; /* definition & initialization of gv, internal linkage only */
      ...
      void foo()
      {
              printf("global value is %d\n", gv);
      }
      ```
    - main.c
      ```
      ...
      int main ()
      {
              foo();
              gv = 5; /* NOT known here*/
              foo();
      }
      ```
  - **can also do with functions:**
    - a.c
      ```
      ...
      static void foo() /* or prototype appears first */
              {whatever}
      goo ()
      {
              foo(); /* ok has internal linkage in this module */
      }
      ```
    - main.c
      ```
      ...
      ```

```
int main ()
{
        foo(); /* NOT OK foo() does not have external linkage - link error will result */
}
```

- **Using "static" for internal linkage is not used as often with C++ because public/private class concept is so much more flexible and powerful**
  - Also can use namespaces to produce the same effect.
- **What is the original meaning of the static keyword? Original usage, occasionally useful**
  - You want a local variable that is not on the stack - lasts for the whole duration of the program.
    - it is "statically" allocated - stays in the same place, instead of being allocated and deallocated on the stack as the function is entered and left.
  - Example: a function that counts how many times it is called

```
void count_calls()
{
        static int counter = 0;  // gets set to zero only on first call
        counter++;  // add one
        cout << counter << endl;
}
```

    - compiler inserts some special code to keep track of whether to do the initialization or not.
- **Another use - class-wide member variables and functions - see the handout - occasionally useful, often used to replace global variables**