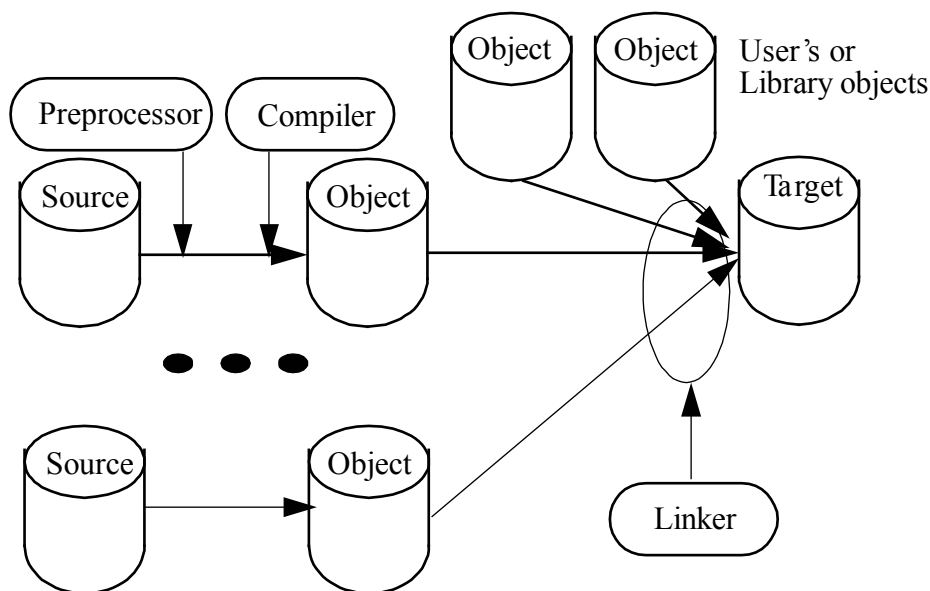# Review of Program Architecture in C

Modern Modular Compiler

- Multiple *sources*/*modules*, often worked on by different team members
    - Note difference between modules (linked together into single executable) and components accessed at runtime via agreed API.
- When each source/module is being complied, information about what to expect in the other modules in needed
    - Provided via *headers*
    - Older languages require the programmer to provide them, newer languages process implicit
    - Needed information on "what is exported" to other modules, that is what other modules can use
        - Functions
        - Variables
        - Classes
- Linker combines together references from multiple modules, user provided or libraries
    - *Externa linkage*
- A single source needs to link multiple internal elements
    - *Internal linkage*

- A larger project can be made of multiple targets communicating somehow via defined API

Source and Header

- *Source* - file containing some instructions or definitions, among other things, that have to be compiled.
  In the C language, it will have the extension `.c`
  Examples of what should go to source (in some languages this can be different)
    - Statements   // typically must be in source
    - Function definitions   // typically must be in source
    - Variable definitions   // typically must be in source
    - Type definitions, macro definitions
        - should be in a source only if this source is the only one using this type or macro,
        - otherwise should be in a header file to be included in multiple sources
            - types definitions and macro definitions are not liked

  Source files should NOT be included (except for some linker optimization)

- *Header* file is a file (explicit or more recently implicit) containing only code that is informational or processed by the preprocessor
  In C, it will have the extension `.h`
    - `Typedef`   // could be in source if for single source
    - Macros   // could be in source if for single source
    - `extern` variables declarations (variable prototypes)
    - function prototypes
    - no variable definitions nor function definitions (other than inline)
    - **No variables** other than `extern`

Program Structure

- Any nontrivial project will be developed in multiple files and by multiple people
- `main()` should be one application source typically by itself
- The remaining code (functions, variables) are implemented in different sources based on how they fit together
    - *Cohesion* (internal match) and *coupling* (external match)
    - Generally code/functions operating on the same data go into the same source
        - Object-oriented languages push this further by making such functions methods of a class and placing the data in the same class
    - So a source is once function (class) or a group of related functions such as functions designed to manipulate the same data
- Another way to create architecture
    - Start will all functions needed to accomplish the objectives
    - Start with one function per file
    - Combine some functions into same file
        - Functions operate on the same data

- Functions need helpers that are not be linked/exposed to other files
  - These are static functions/global variables in the source but NOT prototyped in header
- Functions will communicate with each other using global data hidden from other files
  o Decide on any data that needs to have global external linkage
    - Data needs to be placed in some source and declarations need to be placed in the corresponding header file
- File name reflects the contents of the file
  o Could be the name of the function (class)
- Generally, each source has a header file with the same name
  o `source.c` -> `source.h`
  o The header file `source.h` will have
    - Multiple inclusion prevention
    - List of functions prototypes and variable declarations that are exported out of the source (have *external linkage*)
  o Exceptions
    - The source containing `main()` will generally not export anything and thus will not have header
    - There may be headers without sources, such as containing macros or `typedef`

Multiple Inclusion Prevention

- Assume header file *filename*.h

```
#ifndef FILENAME_H
#define FILENAME_H

// the needed contents

#endif
```

Linkage, Internal vs. External

- Elements at the global level in a source (function, global variable) can be made available to other sources
  o *external linkage*
    - Header file declares these elements, does not provide the actual linkage
    - Linker provides actual linkage
    - Declarations are important for proper compilation is proper linkage
  o `static` keyword prevents external linkage and makes *internal linkage* (available in this source only)

- this is used for "helper" functions and variables, that is functions and variables needed in this file only and not available to other sources

Example: Program architecture

Assume `main()`, `f1()`, `f2()`, `int x`, and structure type `node_t` needed. Assume each function implemented in separate separate, `x` goes in the file with `f2()`, and interfaces as shown.

Note

- Source can include its own header file for cross-checking, not shown
- Multiple inclusion prevention mechanism not shown.

```
appl.c                    f1.c                   f2.c
    //include system      #include "node.h"     #include "node.h"
    #include "node.h"     #include "f2.h"
    #include "f1.h"
    #include "f2.h"
node.h                                          int x;
                          node_t *f1()          void f2(node_t*p)
typedef struct    main()  {                     { // something
{ // members      { node_t *p;      f2(...);     }
                      p=f1();       }
} node_t;             f2(p);
                      x=1;
                  }
                          f1.h                   f2.h
                          #include "node.h"     #include "node.h"

                                                extern int x;
                          node_t *f1();         void f2(node_t*);
```

```
> gcc appl.c f1.c f2.c
```

Summary

- Functions and global variables are in sources
  - They are linked across modules by the linker unless restricted to a file (static)
  - Header files only provide information not linkage
- Types and macros can be in sources or headers
  - Never linked because they are processed before or at compile
  - Put in source for only this source
  - Put in header and include for multiple sources – each source will get its own included
- Header provide information to compiler
  - Function prototypes
  - Extern variables declarations
  - Types and macros to use in more than one module