MULTIPLE-FILE PROGRAMS

CS 23200

Big Picture

- ✓ Developing programs on *nix computers
- C Language
  - ✓ Familiar aspects of C (variables, operators, basic I/O, control flow, functions)
  - ✓ Pointers
  - ✓ Structures and related constructs
  - ✓ File operations
  - Multi-file programs
  - Standard library functions
- *nix tools

Outline

- File organization
- Preprocessor

- Steps from multiple source files to an executable
  - Preprocessor
  - Compiling
  - Linking
  - Libraries

Moving to Larger Projects

- Projects and examples thus far have been small
  - Accomplish a single purpose
  - Only a few functions
  - Not many lines of code

- When these properties are not present, divide the code into multiple files
  - Division is a design decision
  - Strongly related entities should be in the same file
  - Weakly related entities: separate file when a single file gets too big

Multiple Files on the Server

- Helpful to have multiple windows/files visible simultaneously
- Options:
  - Open multiple C9 editor windows
  - For real servers, open multiple putty windows
  - Copy files locally and edit using multiple windows (e.g., vi/emacs/sublime text on Windows or notepad)
  - WinSCP to automatically synchronize files

Order Matters in C

- Every function or structure should be declared or defined earlier in the file than it is first used
- What is wrong here?

```
int fnTwo(int x);
int fnOne(int i){
  return fnTwo(2 * i);
}
int fnTwo(int x){
  return x * x + 1;
}
```

This declaration allows fnOne to call fnTwo properly

- Good practice: every function (except main) has a declaration and a separate definition
  - All declarations before any definitions

Example: List of Images

- Linked list entities can go in their own file(s)
  - Header (list.h) and C file (list.c)
- Image entities can go in their own file(s)
  - Header (image.h) and C file (image.c)

- Main function can use both to make lists of images

File Organization: **list.h**

list.h ***Notice: no executable program statements***

```
#include <stdlib.h>

struct listnode{
   void* data;
   struct listnode *pNext;
};
struct list{
   struct listnode *pHead;
   ...
};

void insertNode(struct listnode *pNode,
                      struct list *pList);
```

includes

structure template declarations

function declarations

File Organization: **list.c**

list.c

```
#include "list.h"

void insertNode(struct listnode *pNode,
                  struct list *pList){
   /* find insertion point */
   struct listnode *pBefore, *pAfter;
   pBefore = pList->pHead;
   pAfter = pBefore->pNext;
   ...
}

...
```

include the corresponding header file (and nothing else!)

function definitions

image.h

```
#include <stdlib.h>
struct pixel{
   unsigned char red;
   unsigned char blue;
   unsigned char green;
};
typedef struct pixel color_t;
struct image{
   unsigned int width;
   unsigned int height;
   struct pixel **image;
};

struct image*
  createImage(const int width, const int height);

void clearImage(struct image *pImage,
                  const color_t *pBackgroundColor);
void destroyImage(struct image *pImage);
```

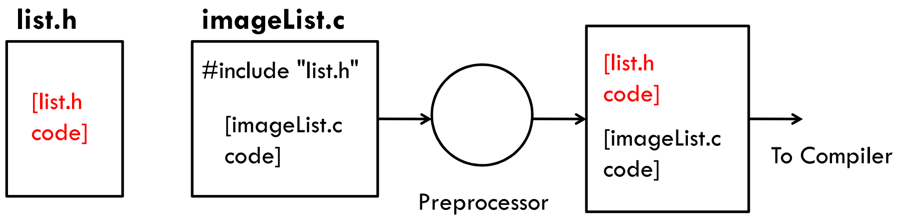
File Organization

imageList.c

```
#include <stdlib.h>
#include <stdio.h>
#include "list.h"
#include "image.h"

int main(){
   const int width = 800;
   const int height = 600;
   struct list imageList;
   ...
   struct image *pImage = createImage(width,
height);
   ...
```

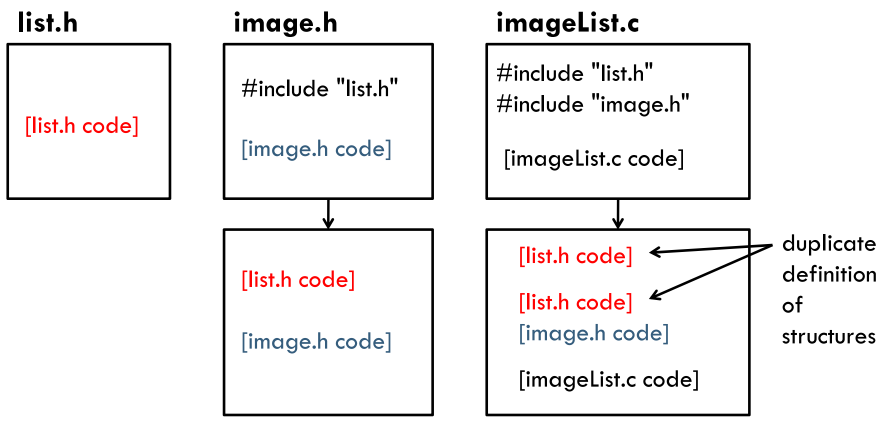
#include

- Preprocessor handles the #include directives
- Use angle brackets for standard library includes, quotes for headers found in the local directory
- #include effectively copies the code from the included file to the point of the #include statement



#include

- What does imageList.c look like after preprocessing?



Other Preprocessor Commands

```
#define IDENTIFIER
...
#ifdef IDENTIFER
/* included:
   IDENTIFIER is
   defined (even
   though it has
   no value */
...
#else
/* not included */
...
#endif
```

```
#define IDENTIFIER 0
...
#ifndef IDENTIFER
/* not included:
   IDENTIFIER is defined */
...
#endif

#if IDENTIFIER
/* not included:
   IDENTIFIER is 0 */
...
#else
/* included */
...
#endif
```

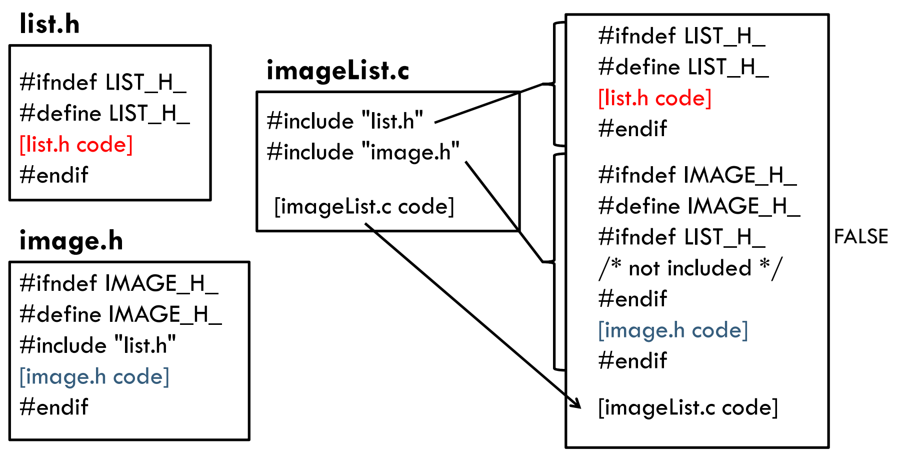
Header Guards

- Typical way to prevent multiple inclusion of headers
- First time the header is included, LIST_H_ is undefined, so the body is included
  - First statement in the body defines LIST_H_
- Other times the header is included, LIST_H_ is already defined, so the body is not included

```
#ifndef LIST_H_
#define LIST_H_

#include <stdlib.h>

struct listnode{
   int data;
   struct listnode *pNext;
};

...

#endif
```

Header Guards

- The expansion of the includes:


list.h

```
#ifndef LIST_H_
#define LIST_H_
[list.h code]
#endif
```

image.h

```
#ifndef IMAGE_H_
#define IMAGE_H_
#include "list.h"
[image.h code]
#endif
```

imageList.c

```
#include "list.h"
#include "image.h"

[imageList.c code]
```

```
#ifndef LIST_H_
#define LIST_H_
[list.h code]
#endif

#ifndef IMAGE_H_
#define IMAGE_H_
#ifndef LIST_H_
/* not included */
#endif
[image.h code]
#endif

[imageList.c code]
```

Header Guards

- After preprocessing:

```
#ifndef LIST_H_           TRUE
#define LIST_H_
[list.h code]
#endif

#ifndef IMAGE_H_          TRUE
#define IMAGE_H_
#ifndef LIST_H_           FALSE
...
#endif
[image.h code]
#endif

[imageList.c code]
```

$\longrightarrow$

```
[list.h code]

[image.h code]

[imageList.c code]
```

Summary of Header Guards

- For each header file you make, the very first and last parts of the file should be the header guards for that file

- When set up correctly, header guards let you include whatever files you need without worrying about duplicate includes
  - Example: you can include <stdio.h> in image.h, list.h, and imageList.c

- Write **headers** for the following structures and functions, grouping into files as appropriate
  - A structure for a string
  - A structure for a text document
  - A function that takes two strings and searches for one in the other
    - The return value should be -1 if the string is not found and the index of the first occurrence otherwise
  - A function that concatenates two strings, returning a new string
  - A function that takes a string and a document and returns how many times that string occurs in the document
  - A function that takes a document and returns another document that lists the word counts of the original document

Outline

- File organization
- Preprocessor

- Steps from multiple source files to an executable
  - Preprocessor
  - Compiling
  - Linking
  - Libraries

For a single source file, gcc -g source.c -o source does all these steps

For multiple files, compiling and linking are often carried out separately

Compiling

- Object file
  - Output from compiling
  - Typically has a .o extension
  - Not a text file (a binary file)
  - Not a complete executable
    - Allowed to have **references** to functions and variables that are not in this object file

Compiling

- Compiling with gcc

-c flag for "compile" output should have .o extension

```
gcc -g -c foo.c -o foo.o
```

Other (less common) usages:

```
gcc -g -c foo.o bar.c -o foobar.o

gcc -g -c file1.c file2.c file3.c -o files.o
```

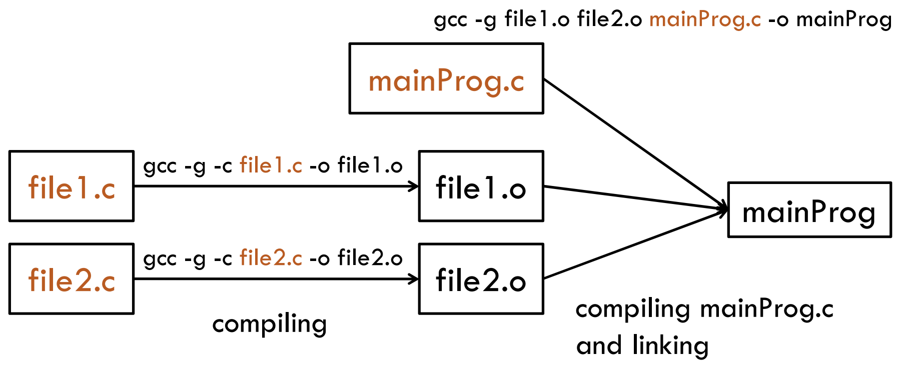
Linking

- Resolves references of object files
- Sets the entry point (at the main function)
- Creates an executable
- Use gcc without the -c flag to call the linker
- Typical usage:
  - Compile a file prog.c that has the main function
  - Link with other object files

```
gcc -g foo.o bar.o prog.c -o prog
```

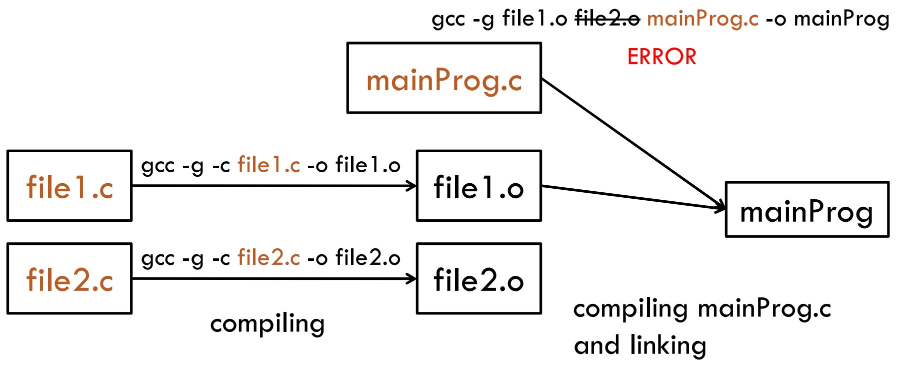
Compiling and Linking

- file1.c and file2.c have functions that could be used in multiple programs, so we compile them to object files separately


gcc -g file1.o file2.o mainProg.c -o mainProg
mainProg.c
file1.c
gcc -g -c file1.c -o file1.o
file1.o
mainProg
file2.c
gcc -g -c file2.c -o file2.o
file2.o
compiling
compiling mainProg.c and linking

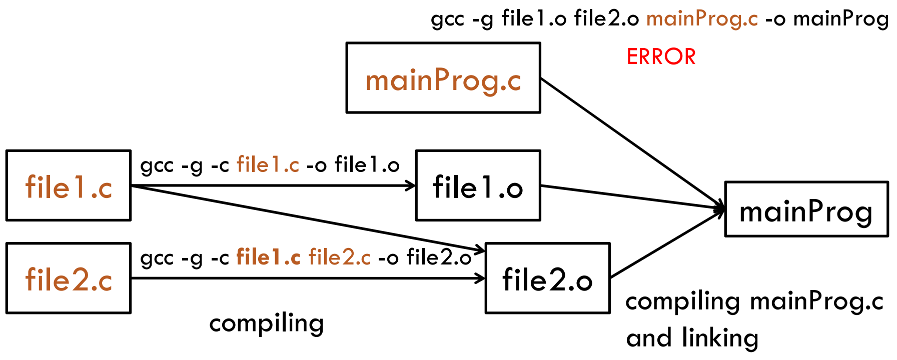
Common Problems with Compiling and Linking Multiple Files

- "undefined reference to fnFromFile2..."
- Including a header, but not linking with the corresponding object file


gcc -g file1.o ~~file2.o~~ mainProg.c -o mainProg
ERROR
mainProg.c
file1.c
gcc -g -c file1.c -o file1.o
file1.o
mainProg
file2.c
gcc -g -c file2.c -o file2.o
file2.o
compiling
compiling mainProg.c and linking

Common Problems with Compiling and Linking Multiple Files

- "multiple definitions of fnFromFile1..."
- Compiling a .c file into multiple object files that are linked together


gcc -g file1.o file2.o mainProg.c -o mainProg
ERROR
mainProg.c
file1.c
gcc -g -c file1.c -o file1.o
file1.o
mainProg
file2.c
gcc -g -c **file1.c** file2.c -o file2.o
file2.o
compiling
compiling mainProg.c and linking

A Word About Header Files

- The header file needs to be included to declare the function prototypes, structure templates, etc.

- A header file can be included into multiple object files which can be linked together
  - Unlike .c files
  - Because the header file does not contain any variable declarations or function definitions

- Tell gcc where to look for header files using `-Ipath`

```
gcc -g -c image.c -o image.o -I../headers
  -I../../otherHeaders
```

Example

- Text document C files:
  - myString.c
  - myDoc.c
  - myProg.c : contains main
- Commands to...
  - Compile the .c files to object files

```
gcc -g -c myString.c -o myString.o
gcc -g -c myDoc.c -o myDoc.o
```

  - Link the object files with a program written in myProg.c

```
gcc -g myString.o myDoc.o myProg.c -o myProg
```

Outline

- File organization
- Preprocessor
- Steps from multiple source files to an executable
  - Preprocessor
  - Compiling
  - Linking
  - **Libraries**

Libraries

- A library is a collection of functions that are already compiled
  - e.g., DLLs on Windows are "Dynamic Link Libraries"
- To use a (non-standard) library, the linker must be told to link with that library
  - Common example: the math library
    - Includes functions like `sqrt`, `sin`, `cos`

Libraries

- To use math library functions:
  - `#include <math.h>` in the source file
  - Link with the math library
    - Add -lm AFTER all the other flags in the compilation command

```
gcc -g myProg.c -o myProg -lm
```

- More information about libraries:
  - "Linking with Libraries" section of Chapter 4 in *Programming with GNU Software*
  - http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

Summary

- File organization
- Preprocessor
- Steps from multiple source files to an executable
  - Preprocessor
  - Compiling
  - Linking
  - Libraries