Separate Compilation

Class 19

Clarification: Defining Variables

- why is this illegal in C89?

```
void foo(void)
{
  for (unsigned index = 0; index < MAX; index++)
  {
    bar(index);
  }
}
```

Clarification: Defining Variables

- why is this illegal in C89?

```
void foo(void)
{
  for (unsigned index = 0; index < MAX; index++)
  {
    bar(index);
  }
}
```

- must do this:

```
void foo(void)
{
  unsigned index;
  for (index = 0; index < MAX; index++)
  {
    bar(index);
  }
}
```

Clarification: Defining Variables

- this is legal in C89

```
void foo(void)
{
  unsigned result = 0;
  unsigned index;
  for (index = 0; index < MAX; index++)
  {
    unsigned temp_value = bar(index);
    result += temp_value;
  }
}
```

Clarification: Defining Variables

- this is legal in C89

```
void foo(void)
{
  unsigned result = 0;
  unsigned index;
  for (index = 0; index < MAX; index++)
  {
    unsigned temp_value = bar(index);
    result += temp_value;
  }
}
```

- variables can be declared at the top of any block, including a nested block

Testing

this is a major way I tested your programs:

```
#!/bin/bash

program=$1
for hex in 0x{0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}{0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}
do
    forward=$(./$1 $hex | cut -d" " -f4)
    backward=$(./$1 $forward | cut -d" " -f4)

    if [[ $backward = $hex ]]
    then
        echo "$hex: ok"
    else
        echo "bad: $hex $forward $backward"
    fi
done
```

Modularity

- real programs are not contained in a single file
- real C programs are modularized so that related resources are grouped together in one .c file

- how to best group resources into modules is an art form
- why to group resources into modules:
  - easier to handle smaller pieces than huge pieces
  - information hiding: prevent the client of a resource from making assumptions based on internal implementation details (data structures or algorithms)
  - force formal analysis and documentation of interfaces
  - allow parallel development with a team of developers
  - facilitate reuse

Example: tree

- tree is a simple, very useful tool (not installed on all systems)
- tree source code has the following modules
- color.c (504) colorize the output
- file.c (284) some of the main guts of reading directories
- hash.c (115) maintain index of already-seen to avoid infinite loops
- html.c (454) output in HTML format
- json.c (318) output in JSON format
- strverscmp.c (158) some weird stuff of alphabetizing names with embedded digits
- tree.c (1324) main program, parse options, print error messages
- unix.c (261) some things specific to unix filesystems
- xml.c (304) output in XML format

Example: nano

- an extremely small, fast text editor

- browser.c (770) file browser and associated dialogs
- chars.c (641) character set support
- color.c (412) syntax highlighting
- cut.c (667) this many lines to cut, copy, and paste
- files.c (2597) locking, backups
- global.c (1490) global variables, messages
- help.c (579) context-sensitive help dialogs
- history.c (604) maintain history of commands for undo, searches, etc
- move.c (597) navigate within a file
- nano.c (2613) main, options
- prompt.c (785) mouse and keyboard support
- rcfile.c (1672) support for saving stuff between sessions
- search.c (989) searching, regular expression support
- text.c (3342) indent, comment out
- utils.c (528) spell checking, measuring lines, words, and characters in regions, etc
- winio.c (3651) figure out what kind of terminal is in use

mycrypt

- a simple demonstration system that encrypts and decrypts files
- `./mycrypt -<de> -<keysize> [filename]`
  -d: decrypt
  -e: encrypt
  keysize must be between 0 and 255 inclusive
  if no filename is given, read standard input
  output to standard output

Files

- main.c
  - the main function
  - usage() function
  - no global variables (of course)
- mycrypt.c
  - functions encrypt() and decrypt()
  - the symbols ENCRYPT and DECRYPT
- options.c
  - parse_command_line() and utility function dump_options() for debugging
  - responsible for parsing the command line
  - defines the maximum allowable length of a file name (for safety)
  - defines a structure for holding the various pieces of the command line, as well as several error indicators

Dependencies

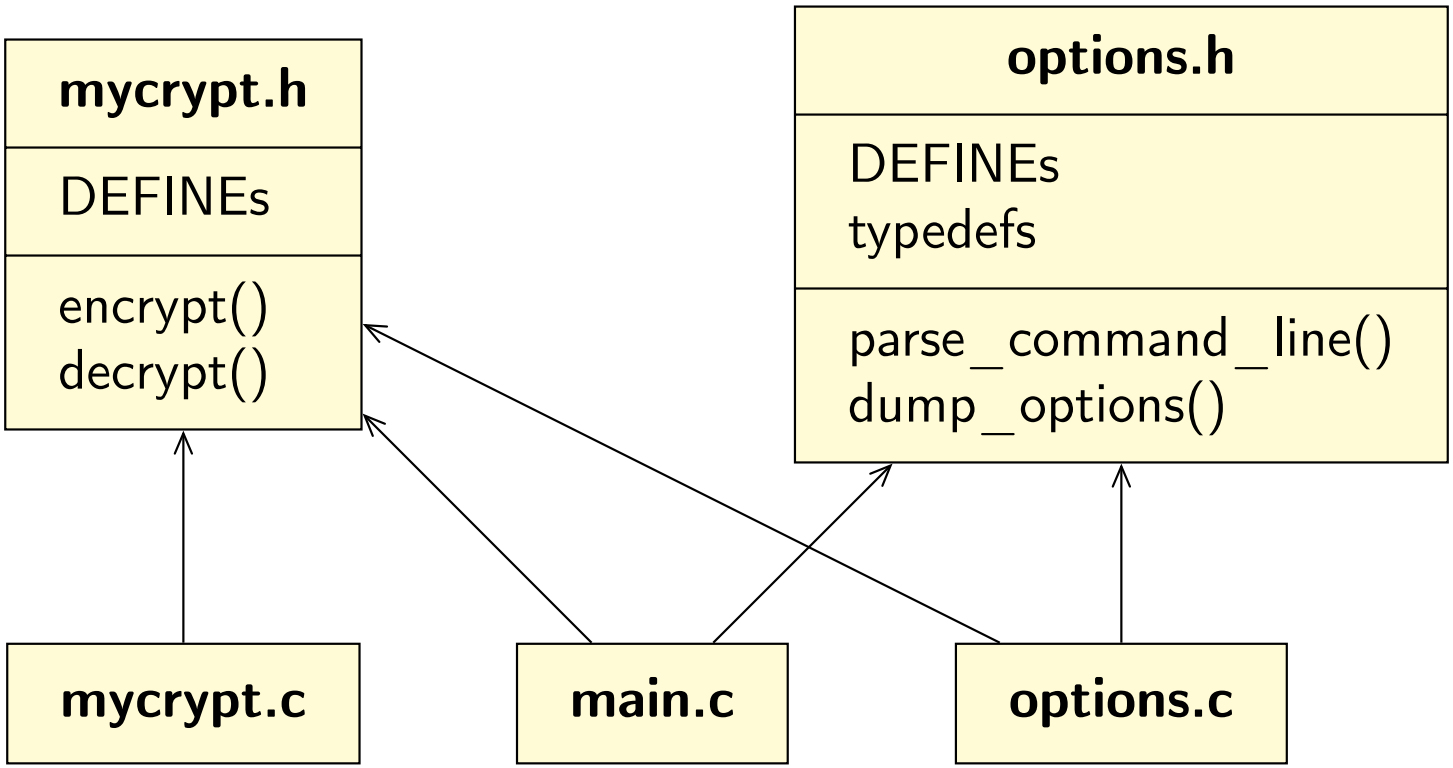
- main depends on stuff provided by both mycrypt and options
- options depends on stuff provided by mycrypt (symbols)
- mycrypt is standalone and could be directly reused, without any dependencies, in another program

- the "normal" organization in C is to have a .h file for every .c file that provides a resource
- the .h file advertises all of the things the corresponding .c file provides
  - global variables
  - functions
  - symbols and enums
  - structure definitions, typedefs

Information Hiding

- a .c file contains the implementation of its functionality
  - internal data structures (e.g., is the stuff stored internally as a linked list or as a tree?)
  - algorithm details (e.g., is the sorting routine quicksort, heapsort, or shellsort?)
- the corresponding .h file is the interface that a client must understand to use the .c functionality
  - names of functions
  - types and order of parameters, and the return type
  - this is where documentation goes
  - so that a potential client can see what the implementation provides, without having to read the implementation
  - only describes what the .c provides, nothing about how it is provided

Includes


mycrypt.h
DEFINEs
encrypt()
decrypt()
options.h
DEFINEs
typedefs
parse_command_line()
dump_options()
mycrypt.c
main.c
options.c

```
 1  /*
 2   * encryption and decryption routines, written for the mycrypt system
 3   * Jon Beck
 4   */
 5
 6  #ifndef MYCRYPT_H
 7  #define MYCRYPT_H
 8  #include <stdio.h>
 9  #include <stdint.h>
10
11  /**
12   * flags available to determine whether we are running in
13   * encrypt or decrypt mode
14   */
15  #define ENCRYPT 0
16  #define DECRYPT 1
17
18  /**
19   * encrypt a stream using the given key, with results going to stdout
20   * @param input_file the open stream
21   * @param key the encryption key to use
22   */
23  void encrypt(FILE* input_file, uint8_t key);
24
25  /**
26   * decrypt a stream using the given key, with results going to stdout
27   * @param input_file the open stream
28   * @param key the encryption key to use
29   */
30  void decrypt(FILE* input_file, uint8_t key);
31
32  #endif
```

```
1  /*
2   * command line parser, written for the mycrypt system
3   * Jon Beck
4   */
5
6  #ifndef OPTIONS_H
7  #define OPTIONS_H
8  #include <stdint.h>
9
10 #define MAX_FILE_NAME 255
11
12 typedef struct
13 {
14   unsigned direction;
15   uint8_t key;
16   char filename[MAX_FILE_NAME];
17   unsigned direction_error;
18   unsigned key_error;
19 } Options;
20
21 /**
22  * parse the command line, putting the results (and any error
23  * conditions detected) into the return struct
24  * @param argc the number of argument strings provided
25  * @param argv the array of strings to be parsed
26  */
27 Options parse_command_line(size_t argc, const char** argv);
28
29 /**
30  * for debugging purposes, dump the contentes of an Options struct
31  * to stdout. not really designed for production use
32  * @param options the struct whose contents to dump
33  */
34 void dump_options(Options options);
35
36 #endif
```

.c Files

the top of mycrypt.c:

```
1 #include "mycrypt.h"
2
3 void encrypt(FILE* input_file, uint8_t key)
```

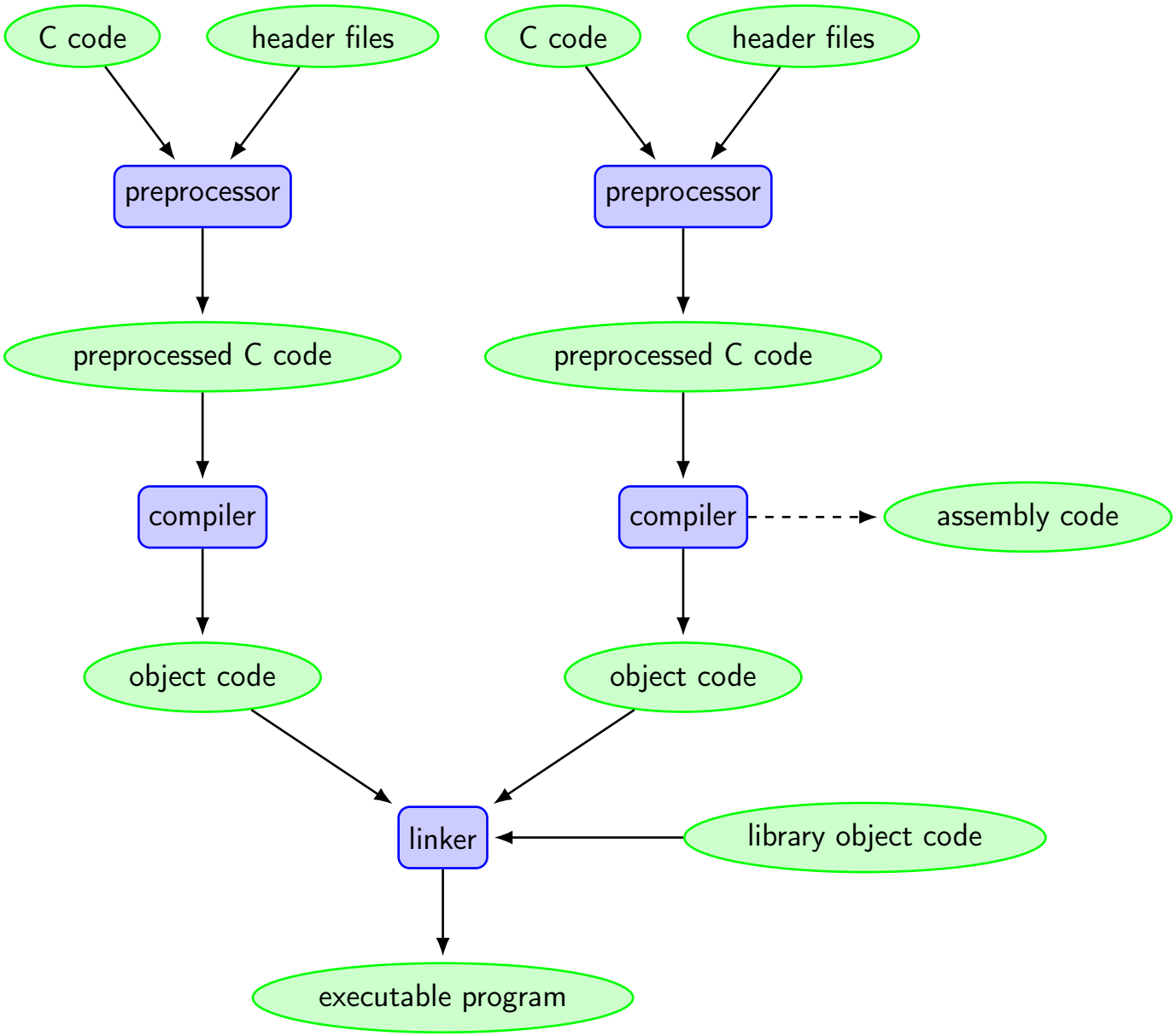
the top of options.c

```
1 #include <ctype.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "options.h"
5 #include "mycrypt.h"
```

the top of main.c

```
1 #include <stdio.h>
2 #include <stdint.h>
3 #include "options.h"
4 #include "mycrypt.h"
```

Compilation



Compilation Process

- we have a .c file and its associated .h file
- the preprocessor puts them together into a single unit
- the compiler now is supposed to create object code

- the compiler command for all-in-one compiling is

```
$ clang -Weverything -std=c89 -pedantic-errors \
        -o filename filename.c
```

- this takes you straight from .c (and included .h's) to executable

Separate Compilation

- to enable separate compilation, we need to explicitly generate object code

```
$ clang -Weverything -std=c89 -pedantic-errors \
        -c filename.c
```

show example

Compilation Process

- a .h file describes (actually, declares) what is provided by the corresponding .c file
- but does not actually provide it

- another .c file can't be directly used
- the resources described by the .h file reside in the .o file (for local files)

- there are actually two types of .h files
  - local: `#include "foo.h"`
  - system: `#include <foo.h>`

System Libraries

- a very common directive:
  `#include <stdio.h>`

- what does that do?
- same thing we've been talking about: it describes what is provided by a file named stdio.c
- where is that?

- the file stdio.h is at /usr/include/stdio.h
- you can look at it with less or open it in an editor

System Libraries

- `#include <stdio.h>`

- but the file stdio.c isn't even on your system (unless you downloaded the sources — it's free software)
- instead, your system includes the pre-compiled object file, stdio.o

- actually, it's a bit more complicated than that
- there is no file named stdio.o
- instead, many .o files are packaged together into a library archive
- the stuff that stdio.h describes is in the library `/usr/lib/x86_64-linux-gnu/libc.a`
  (the location is different on Macs: `/usr/lib/libc.dylib`)

System Libraries

- libc.a is a huge file that has many .o files packaged inside of it
- we can view its table of contents with the archive tool ar:
  `$ ar -t /usr/lib/x86_64-linux-gnu/libc.a | less`
  (Mac: `$ llvm-objdump -a /usr/lib/libc.dylib | less`)

- unfortunately, you still won't find stdio.h there
- instead, you'll find object code for all the stdio functions: putchar.o, printf.o, etc etc
- as well as object code for all the other standard system .h stuff

- stdio.h describes some of the functions in libc.a
- stdint.h describes others
- string.h still others

- the compiler knows where to find something referred to in stdio.h

Linking

- so now we have our local .o object code
- and we know where to find .o code in the system library archives
- now we're ready to invoke the linker to put them together into an executable program
- exactly one .o file must contain a function named main, with the correct parameter list

- we call the linker:

```
$ clang -Weverything -std=c89 -pedantic-errors \
      -o filename file1.o file2.o ... filen.o
```