C: Program Structure

Department of Computer Science
College of Engineering
Boise State University

September 11, 2017

Scope

- Variables and functions are visible from the point they are defined until the end of the source file.
- Creating function prototypes at the start of the source file (or in an included header file) makes them visible throughout the source file.
- Add modifier `static` in front of a function declaration makes the scope of the function to be limited to the containing source file (similar to `private` modifier in Java).

Scope (2)

- Global variables are not visible to code in another source file. To make a global variable be visible to multiple source files, define it normally in only one source file and then declare it as `extern` in all the other source files.
- The *declaration* of an external variable announces the properties of the variable (e.g. type, name).
- The *definition* of an external variable will actually *create and set aside storage for the variable*.
- There must be only one *definition* of an external variable among *all* files that make up the source program.

```
/* file1.c */
int moomoo = 10;
```

```
/* file2.c */
extern int moomoo;
```

```
/* file3.c */
extern int moomoo;
```

Example

- Examples: ourhdr.h, scope1.c, scope2.c.

Convention for writing header files

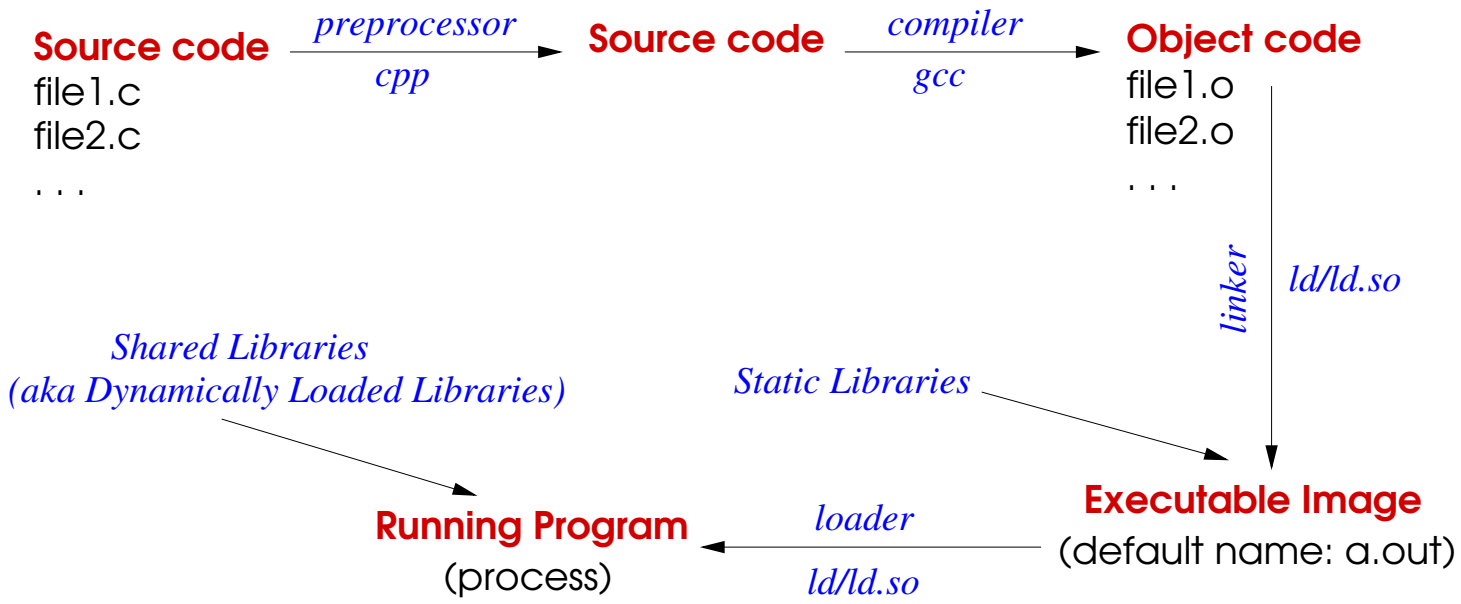
- Note that by using the ifdef's to wrap the header file is a standard way of declaring header files. This prevents the contents of the header file from being included multiple times (which would cause compilation errors)
- The convention is to use two leading underscores, convert the letters in the name of the header file to all uppercase and convert periods to underscores. You should always follow this convention.

```
/* C-examples/scope/ourhdr.h */
#ifndef __OURHDR_H
#define __OURHDR_H

/* declarations */

#endif /* __OURHDR_H */
```

Compiling, Linking and Running Programs



Note: The above illustrates the GCC compiler tool chain but the concepts are the same for any C/C++ compiler.

GCC: The GNU C/C++ Compiler options

An integrated C/C++ open-source compiler, invoked as `gcc`. Available on a wide variety of platforms. The following shows some commonly used command line. See the man page for more options.

- `-c` Compile but do not link.
- `-Wall` Print all warnings. *Always use this option...some of the warnings can save you hours of debugging!*
- `-O` Optimize option. Usually speeds up the execution time significantly.
- `-g` Enable debugging. Need this option if you want to use a debugger. Also gcc allows you to use the optimize option (-O) at the same time as the debug option.
- `-o <filename>` Name the output executable file to the given filename. Otherwise the default file name is `a.out`.
- `-I dir` Add the directory `dir` to the list of directories to be searched for header files. Directories named by `-I` are searched before the standard system include directories.

The GNU C/C++ compiler options

- `-D<macro>=[<value>]` Define a macro. The C file is processed with the macro defined. For example `gcc -Wall -DDEBUG ex1.c` would define the DEBUG macro to be true for the file being compiled.
- `-E` Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.
- `-S` Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.
- `-version` Display the version number and copyrights of the invoked GCC

Example: Design Using Functions and Files

- We will study a postfix calculator example from C-examples/scope/calculator
- Uses a stack data structure.
- Divides the program into logical files (similar to Java classes)

In-class Exercise

- Revisit the remove blanks example here: https://github.com/BoiseState/CS253-resources/tree/master/C-examples/intro/remove-blanks
- Refactor the program to use one function per state transition. Replace the code in the `main` to use calls to the functions you have created.
- Recompile and retest.

C Preprocessor

- File inclusion. The command is `#include`. Files in the standard library can be specified using angle brackets. Files specified in double quotes are relative to the current directory. For example:

```
#include <stdio.h>
#include "myheader.h"
```

- Defining constants

```
#define MAX 323239283
```

- Macro substitution. The command is `#define`. Macros expand into in-line code. There is no compile-time check for type correctness. For example:

```
#define max(a,b) ((a)>(b)? (a) : (b))
```

- Example: C-examples/c-preprocessor/macro.c.

Conditional Inclusion (1)

```
#ifdef __linux__
    printf("I am the Happy Penguin!\n");
#elif _WIN32
    printf("Welcome to MS Windows ( I rule!).\n");
#elif __APPLE__&&__MACH__
    printf("Welcome to I am cool!\n");
#else
    printf("Uh! who am i?\n");
#endif
```

- The above `ifdef` strings are standard ways of detecting the Operating System the code is being compiled on.
- For full list of identifying strings for various systems, see http://sourceforge.net/p/predef/wiki/OperatingSystems
- Example: C-examples/c-preprocessor/ex2.c.

Conditional Inclusion (2)

- `ifdef` statements are also used for toggling debug statements on and off.

```
#ifdef DEBUG
        printf("nc=%d\n",nc);
#endif
```

- Three ways of triggering the DEBUG flag
  - The DEBUG preprocessor flag can be defined at the top of the source file (or in a common header file) but then we have to change the code if we want to turn the debug output on, which isn't desirable.
  - We can define the DEBUG flag in the command line arguments to the C compiler as shown below:

    ```
    gcc -Wall -DDEBUG=1 ex1.c
    ```

  - We can define the DEBUG flag in the command line arguments to the `make` command and have it pass that on to the C compiler. For example:

    ```
    make "DEBUG=-DDEBUG=1"
    ```

    See example below for more details on this option.
- Example: C-examples/c-preprocessor/ex1.c.