Errors During Compilation and Execution – Background Information

Preprocessor Directives and Compilation

- #define <name> <body> - defines a macro, identified by <name>. During compilation, all instances of the macro name are replaced by its <body>.
  - ex: #define M_PI 3.141592653 defines a macro constant for pi. Everywhere M_PI is used, the text “M_PI” will be replaced with the decimal.
- The flag –D<identifier>=<body> can be used to define macros. Like macros defined in source files, the body is optional. -D<identifier> will define an empty macro just like #define <identifier> will.

Preprocessor Directives and Compilation

- Conditional Compilation
  - You can conditionally compile portions of code using preprocessor directives.
  - #if <expression>, #else, #endif can be used to conditionally include to be compiled based on the value of <expression>. #elif can also be used, similar to ‘else if’, to chain optional blocks.
  - #ifdef <identifier> checks if a given macro identifier is already defined.
  - #ifndef <identifier> checks if a given macro identifier is *not* already defined.

Preprocessor Directives and Compilation

- #include - used to include source files. The text of the included file is copied into the current translation unit.
- Include search path – the set of directories searched by the compiler to locate included source files
  - A filename enclosed in ‘<>’ will search through the paths defined in the include search path (ie #include <string>).
  - A filename enclosed in “” will search in the directory the current translation unit is in before searching through the include search path (ie #include “myfile.h”).

Preprocessor Directives and Compilation

- Default search paths depend on your platform, but they will include all the headers for system-provided libraries (like /usr/include, usr/local/include, etc).
- The flag –I <path to header directory> can be used to add a directory to the include search path, which will then be searched when looking for included files.

Preprocessor Directives and Compilation

- Header Guards
  - Used to prevent multiple inclusion (which would then lead to many linker errors, mainly errors from multiple definitions)
  - Example:

```
#ifndef SOME_UNIQUE_NAME
#define SOME_UNIQUE_NAME
// contents of the header file
#endif
```

Preprocessor Directives and Compilation

- Example of conditionally compiling code. Include one source file is on Windows, another if on a posix-like platform.

```
#ifdef __WIN32
  #include <Windows.h>
#else
  #include <unistd.h>
#endif
```

Preprocessor Directives and Compilation

- -E: print the output from the preprocessor and halt compilation.
    - This would contain the code as it was *after* all the #if, #include, etc were processed.

Preprocessor Directives and Compilation

- Other useful preprocessor directives:
  - #undef: undefine a previously defined macro
  - #error <message>: generate a compiler error with a custom message
  - ‘#’ will stringify the given token
  - ‘##’ will concat two given tokens
  - __FILE__: will expand to the full path of the file
  - __LINE__: will always be defined as the current line number
  - #pragma once: can be used as an alternative to a header guard. Supported by all major compilers (GCC version 3.4+)

GCC shared library code-gen flags

- -fpic: emits position independent code. Necessary for shared libraries.

- -shared: emits a shared library instead of an executable. This would not look for a ‘main’ as an entry point, but rather compile the code to be used in another program.

Linking with GCC

- Linking is the process of creating an executable from multiple object files (.o) and external libraries
- To perform linking, the compiler invokes a separate program called a linker (ld on Linux)
- Libraries can be dynamic or static:
  - Static (.a files) - Library functions are copied into your compiled program
  - Dynamic (.so files) - Library functions are stored in separate files and loaded at runtime

Linking with GCC

- To link a library, you must tell the compiler the name of the library you want to link and where to look for it
- Linker search path – The set of directories in which the linker searches for libraries
- The search path includes several locations by default where system-provided libraries are stored, for example /usr/lib and /usr/local/lib
- -L path/to/lib/dir – adds the specified path to the set of directories to search for libraries

Linking with GCC

- -l<library_name> - tells the compiler to link a particular library
- The linker searches the linker search path for a file named either lib<library_name>.so (dynamic) or lib<library_name>.a (static)
- The dynamic library gets preference, unless the -static flag was passed to the compiler

- Example: -lm links /usr/lib/libm.so

Linking error example

```
/tmp/ccVMKuR6.o: In function
`main':linkfail.c:(.text+0xf): undefined reference to
`library_function'collect2: error: ld returned 1 exit
status
```

Runtime linking

- When an executable requires a shared library, the runtime linker is invoked to locate and load that library
- LD_LIBRARY_PATH - the list of directories that the runtime linker searches to find shared libraries. Same format as PATH. By default includes the system directories (/usr/lib, /usr/lib/local, etc)
- If a required library can’t be found, the program will crash

objdump

- objdump is a tool for analyzing compiled executables, libraries, object files, etc

- -D flag prints a disassembly of a compiled binary

- See the man page for the other available options