CS C341/ IS C361 – Data Structures and Algorithms
Compilation and linking hands-on reference

Typographical conventions

We use the following conventions in this guide:

gcc	The name of a specific command or file
filename	You should replace *file* with a specific name
WARNING	Output that you see on the screen

Getting started with gcc

- Log into the Linux server using your respective ids/ Boot to linux

- You can compile the file with the following command:

  gcc *sourcefilename.c*

- Observe the executable formed is called a.out. Execute the file using the command:

  ./a.out

  (Here ./ implies that the path of a.out is the current directory)

- Once again compile the source code with the following command:

  gcc –o *outputfilename sourcefilename.c*

  Where *outputfilename* is the name of the executable file. After this command gets executed successfully, the name of the executable is not a.out but whatever is given as *outputfilename*

  Execute the output file using the command:
  ./outputfilename

Object files

- Once again compile the source code with the following command:

  gcc –c *sourcefilename.c*

  Observe using ls that an object file called *sourcefilename.o* is created in the directory

  What is the difference between an executable and an object file?

  Now, create the executable from the object file using the following command:

  gcc *sourcefilename.o*

CS C341/ IS C361 – Data Structures and Algorithms
Compilation and linking hands-on reference

Preprocessor directives

- What is the result of programs after preprocessing? Note that you only have to concentrate till the preprocessed stage. After thinking the answers, preprocess your program with the following command to see the actual results.

  gcc –E *sourcefilename.c*

Additional gcc switches

- Now, compile the program with the –Wall switch. Observe the warnings that are now given by the compiler.

- Now, compile the program with the –ansi switch. Observe the warnings that are now given by the compiler.

Compiler's output

- Compile the programs using gcc with the -S option such that only compilation takes place and assembling does not take place.

- Observe that variables are represented as addresses and not as names in the resulting low-level code

Linking

- Suppose you write a program linking1.c, which has main function. Inside main there is a call to another function which is defined in another file linking2.c

- Now Compile the file linking1.c using the following command:

  gcc –c –Wall linking1.c

  Observe the warnings received

  Again compile the file linking1.c using the following command:

  gcc –Wall linking1.c

  Observe the warnings received

- Now create a header file prototype.h, which contains the prototype of the function defined in linking2.c. Include this file in linking1.c using the following statement:

  #include <prototype.h>

  and again compile linking1.c using the following command:

  gcc –I. –c –Wall linking1.c

The –I switch tells gcc where the included header files have to be searched. Thus –I. implies that the current directory needs to be searched by gcc

Compare the result with the earlier compilation using the same command

- Similar effect can be achieved by including the file prototype.h using the following command

  #include "prototype.h"

  and then compiling using the following command

  gcc –c –Wall linking1.c

  If the included file name is given in double quotes, searching for the file typically starts where the source program was found. If it is not found there, then searching follows the paths given by the –I flag and implementation defined paths.

  If the included file name is given in <>, searching follows the paths given by the –I flag and implementation defined paths.

- Now after all the compilation warnings have been removed, link the 2 files together:

  ```
  gcc –c –Wall linking1.c
  gcc –c –Wall linking2.c
  gcc linking1.o linking2.o
  ```

- Do not include the header file prototype.h containing the prototype of the function in the file linking1.c.
  Instead of that include the file linking2.c in the file linking1.c using the following statement:
  #include "linking2.c"

  Now repeat the following three commands:

  ```
  gcc –c –Wall linking1.c
  gcc –c –Wall linking2.c
  gcc linking1.o linking2.o
  ```

  Observe the error received.

  Thus we see that the inclusion of a C source file in another C source file is NOT a correct way of writing programs. Imagine a scenario where there are 100 source files each having function calls to functions defined in other files. It would be a chaos trying to include files in each other, with no guarantee that the files would compile successfully

Q. Is it right to include C function definitions instead of their prototypes in a header file?

- From the above explanation, it follows that header files should only contain declarations (prototypes) of functions and not their definitions

Q. Is it right to include C global variable definitions in a header file?

- If a global variable is defined in a header file, and that header file is included in multiple source files linked together, then it would give an error of multiple definitions for a single variable. Thus global variables should be defined in a C source file, while its extern declaration may be included in a header file. This header file may then be included in all the other source files accessing that global variable

- Do not include the file linking2.c in linking1.c. Instead of that include the file prototype.h and proceed as mentioned.

- A shell script is a collection of commands that can be executed on the shell. Create a script doCompilation, which should contain the commands to compile both the C source files separately and then link them together as mentioned earlier.

  Execute the script with the following command:

  sh doCompilation