CS29206: Systems Programming Lab

Autumn 2025

Compilation using gcc

Acknowledgement

Several slides, and contents of some others, have been taken from the slides made by Prof. Abhijit Das for an earlier offering of the course

Resources

- "An Introduction to gcc" by Brian Gough (Chapters 1-4, 10-11)
- "Using the GNU Comiler Collection", by Richard Stallman and GCC developer community, Chapter 3
  - GCC online documentation at https://gcc.gnu.org/onlinedocs/
  - Open the latest manual version pdf
  - Shows you how extensive gcc really is

Bash Shell Variables

Bash shell variables

- As noted earlier, a shell is just another program
- Shell variables: a set of variables that are maintained by the shell
  - Some are created when the shell starts, defines the behavior of the shell (environment variables)
  - You can view all/any variable from the command prompt with the commands below
    - echo $<variable_name> to see the value (if set) of the variable <variable_name>
    - set to view all shell variables
      - May print a long output, use set | more to see page-by-page and see the first 1-2 pages
    - All shell variables values are stored as strings
  - You can add new variables or can change values of the existing variables from the command prompt
    - <variable_name>="…." (**no blanks** before and after "=")

Example

(showing only some selected variables for set command as output is too long)

```
$set
BASH=/bin/bash
C_INCLUDE_PATH=.
HOME=/home/faculty/agupta
HOSTNAME=cpu102
HOSTTYPE=x86_64
LOGNAME=agupta
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:.
TERM=xterm
UID=1016
USER=agupta
```

```
$echo $HOME
/home/faculty/agupta

$MYVAR ="MY_VALUE"
$echo $MYVAR
MY_VALUE
```

- Shell variables are a very powerful tool
- We will see more extensively when we do shell scripting later
- Right now we will only use a few

The PATH variable

- When you type a command (Linux command or name of any executable you created), how does the shell know where to look for the command to run?
  - Given by the shell variable PATH
  - A list of directories, separated by :, that the shell looks in order to search for the command
  - You can add directories to the PATH variable

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:.

$ PATH=$PATH:$HOME
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:.
:/home/faculty/agupta
```

Stages of Compiling a C Program

Basic stages of compiling a C program

- Preprocessing
  - Processes all # directives, removes comments
  - Example: #include: Preprocessing stage copies the contents of the files mentioned in the #include (like stdio.h for #include <stdio.h>) at that place of the program
- Compiling
  - Converts the C program to a processor specific format called assembly language
  - The assembly language code is stored in a .s file
  - Calls to external library functions (like printf()) are left unresolved
- Linking
  - Links different object files and libraries into a single executable file or library
  - Input: Multiple .o files and libraries
  - Output: a single executable file or a library

Basic stages of compiling a C program

- You can generate the output of any stage individually also
  - Preprocessing: cpp hello.c > hello.i or gcc -E hello.c > hello.i (invokes cpp)
    - Open the file hello.i. What do you see?
  - Generating the assembly file: gcc - S hello.c
    - Use ls to see a file named hello.s generated. Can you view its contents?
  - Generating object file: gcc -c hello.c
    - Use ls to see a file named hello.o generated. Can you view its contents?
  - Linking: gcc hello.o
    - Use ls to see a.out is created
    - Assumed no other libraries are needed for hello.c

- So why didn't you see all this when you ran cc/gcc in 1st year?
  - gcc automatically does everything for you (all the stages) when you run it with default options
    - gcc invokes the following commands in order
      - cpp for preprocessing
      - gcc –S for generating the .s file
      - as to convert the assembly .s file to .o file
      - ld for linking
  - Routinely we call the whole process of converting C files to an executable as "compilation"

Example

```
#include <stdio.h>
int main()
{
	/* This is a test program */
	printf("Hello World\n");
	return 0;
}
```

```
$ ls
hello.c
$ gcc -E hello.c > hello.i
$ ls
hello.c  hello.i
$ gcc -S hello.c
$ ls
hello.c  hello.i  hello.s
$ gcc -c hello.c
$ ls
hello.c  hello.i  hello.s hello.o
$ gcc hello.o
$ ls
a.out  hello.c  hello.i  hello.s hello.o
$ ./a.out
Hello World
```

```
$cat hello.i | more
…
…
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
…
…
# 4 "hello.c"
int main()
{

 printf("Hello World\n");
}
```

These lines (only few shown) are copied from stdio.h

These lines are in hello.c. Note the blank line where the comment was

Working with multiple C files

Breaking your program into multiple files

- Modular programming is a good practice, and is needed in any large coding project
- Large source files take huge time for recompilation
- If the code is broken down in pieces, then only the pieces that are changed need recompilation
- Large software development is a two-stage process
  - Generate object files from individual modules.
  - Merge the object files into a single executable file
- Sometimes object files are combined in the form of libraries
- User programs can use the functions archived in libraries during future developments

- So what will you have when you write a software in C?
  - Header files
    - Files with .h extension
  - C files
    - Files with .c extension
    - Will include the .h files using #include
  - Libraries
    - Precompiled code written by others containing ready to use function codes that you can call from your program
    - Example:
      - C standard library: contains codes for printf(), scanf() that you used and many other things
      - C Mathematical library: contains codes for sqrt(), fabs(), sin() etc. that you used
- Goal of compilation: to combine all of them into a single executable file and/or libraries

Compiling more than one C file

- Straightforward way: compile all of them with a single gcc command
  - gcc   file1.c file2.c file3.c -o files.exe
  - Requires you to compile all files again and again even if only one changes
  - Infeasible if you have a large number of files
- Better way
  - Compile each of them into .o files
  - Link them together to a single executable file
    - gcc -c file1.c
    - gcc -c file2.c
    - gcc file1.o file2.o -o files.exe
  - If one of them changes, just create that .o file only, and link again
  - Still not good to do manually if you have a large number of files
    - We will see next week how to write makefiles to address this problem

How to break your code into multiple files

- Break up into .h and .c files
  - .h file will typically contain
    - #include of other .h files
    - Type definitions (typedef, struct, …)
    - Function prototype definitions
    - Macros (#define etc.)
    - Basically, definitions only. Should not contain anything that will actually require memory when the program is run
  - .c file will typically contain
    - #include of .h files
    - Global variable declarations
    - Function codes
    - May also contain some typedefs/prototypes/macros etc. that are relevant only within this .C file
- <u>Never #include any .c file</u>
- General guideline: Keep related things together in one file

Using header files in your programs

- The system already has a large number of header files in some default include directories (usually they are mostly in /usr/include directory)
  - To include these files in your program, use #include <…>
    - Ex. #include <stdio.h>, #include <math.h>, #include <stdlib.h>, …
  - When gcc sees the "<…>" part , it looks in the default directories for the corresponding header file
- You will also write your own header files. To use them
  - Store them in a directory (different header files can be stored in different directories also depending on content type)
  - Include them in your program using #include "….."
    - Ex.: #include "myheader.h"
  - How will gcc know where myheader.h is?

- How do you tell gcc where your header files are? Different options
  - Place your header files in the default directories and include them using the "<…>" syntax, gcc will automatically look there
    - But you may not have permission to put your files in the default directories
    - What problems do you think it can cause if anyone is allowed to put their .h files in /usr/include?
  - Use the -I option while compiling
    - Ex. gcc -I/home/user/agupta/include -I. hello.c -o hello.exe
      - Tells gcc to look for any header file included with the "…" syntax in hello.c in the directories /home/user/agupta/include and in the current directory
    - Works fine if you have to include only a few directories

- Set the C_INCLUDE_PATH shell variable
  - No -I option needs to be specified every time gcc is called
  - gcc will automatically look in the directories in the value of this variable
- Setting the variable
  - export C_INCLUDE_PATH=$C_INCLUDE_PATH:/home/user/agupta/include:.
  - Use the command echo $C_INCLUDE_PATH before and after setting the value to see the difference
  - The "export" command makes the variable available to all shells opened from this shell (not to another terminal opened separately)
    - The gcc command will run in its own shell under the shell from where it is called
    - Will discuss export more later
  - Why didn't we just do $C_INCLUDE_PATH=/home/user/agupta/include:. since we just wanted to add these two directories?

Study the code given in Example-Staque.pdf for a simple example of stack and queue implementation and use. See how all we talked about are exercised there.

C Macro

Using Macros

- A symbolic name given to a value/expression/piece of code
- Defined in a program using the #define preprocessor directive
- C Preprocessor replaces the macro name used in the program with its definition before the actual compilation
  - Example: #define MAX_ARRAY_SIZE 100
    - Will replace all occurrences of the term MAX_ARRAY_SIZE in the program with 100
- <u>Replacement is literal/textual, no evaluation is done before replacement</u>
  - Example: #define DIMENSION 100*20
    - Will replace all occurrences of the term DIMENSION in your program with 100*20 (**<u>NOT by 2000</u>**)
- Macros can be parameterized
  - Example: #define square(x) (x*x) will replace
    - All occurrences of the term square(y) with y*y
    - All occurrences of the term square(z) with z*z

- Macros can be defined during compile time also using the –D option
  - gcc -DMAX_ARRAY_SIZE=100 …
    - Has the same effect as writing #define MAX_ARRAY_SIZE 100 inside the program
- Why are macros useful?
  - Allows for more readable code, by using mnemonic names
    - Example: MAX_ARRAY_SIZE for maximum array size instead of a number 100
  - Allows for smaller source code size if some small code snippets are used in many places
    - Define it as a macro and use the macro name, no need to repeat the code
  - Allows for easy maintenance of code
    - If you want to change something, just change it in the macro
      - Example: to change maximum array size of an array you use, just change the #define MAX_ARRAY_SIZE (one line change only), no need to look at every place the array size is defined in your code and change (Not just cumbersome, high chance of missing a change causing error)

- Macros can be undefined after being defined
  - #undef <macroname>
  - The macro will be undefined from the point the #undef is in the code to the end of the program (unless it is defined again with #define)
- You can check if a macro is defined or not
  - #ifdef <macroname> will be true if the macro is defined
  - #ifndef <macroname> will be true if the macro is not defined
  - #else and #endif to write if-then-else type code blocks based on a macro is defined or not

```c
#define MYFLAG
int main ()
{
   #ifdef MYFLAG
      printf("MYFLAG is defined\n");
      #undef MYFLAG
   #else
      printf("MYFLAG is not defined\n");
   #endif
   #ifndef MYFLAG
      printf("MYFLAG is undefined here\n");
   #else
   printf("MYFLAG is still defined here\n");
   #endif
   return(0);
}
```

```
$ gcc  macros.c
$ ./a.out
MYFLAG is defined
MYFLAG is undefined here
```

Now remove the #define from the C program

```
$ gcc -Wall macros.c
$ ./a.out
MYFLAG is not defined
MYFLAG is undefined here
$ gcc -Wall -DMYFLAG macros.c
$ ./a.out
MYFLAG is defined
MYFLAG is undefined here
$
```

Example: Printing different things based on a macro value

```
#ifdef ERR_LEVEL
    if (ERR_LEVEL == 1)
        printf("Printing only critical error messages\n");
    else if (ERR_LEVEL == 2)
        printf("Printing all error messages\n");
#else
        printf("No error message will be printed\n");
#endif
```

```
$gcc  macro.c
$./a.out
No error message will be printed
$ gcc -DERR_LEVEL=1 macro.c
$./a.out
Printing only critical error messages
$ gcc -DERR_LEVEL=2 macro.c
$./a.out
Printing all error messages
```

Note that # are preprocessor directive, they are handled before compilation. So if ERR_LEVEL is not defined, the part inside #ifdef till before #else will not even be there in the output of the preprocessor, will not be compiled.

To check, run cpp macro.c and see which part gets included

Example: Ensuring a header file is included only once

defs.h

```
typedef struct _node {
	int data;
	struct _node *next;
} node;
typedef node *nodep;
```

list.h

```
#include "defs.h"

typedef nodep list;

list createlist();
list insert(list, int);
list delete(list, int);
void printlist(list);
```

stack.h

```
#include "defs.h"

typedef nodep stack;

stack create();
stack push(stack, int);
stack pop(stack);
void print(stack);
```

main.c

```
#include "list.h"
#include "stack.h"
int main()
{
	stack S;
	list L;
	/* Some code here */
}
```

```
$gcc main.c
In file included from stack.h:1,
                 from main.c:3:
defs.h:3:16: error: redefinition of 'struct _node'
    3 | typedef struct _node {
      |                ^~~~~
In file included from list.h:1,
                 from main.c:2:
defs.h:3:16: note: originally defined here
    3 | typedef struct _node {
      |                ^~~~~
.....
.......
```

You will get too many errors because stack.h tries to include defs.h after list.h has already included it!! So everything is flagged as redefinition!!

Correct way to write list.h and stack.h

defs.h

```
typedef struct _node {
	int data;
	struct _node *next;
} node;
typedef node *nodep;
```

list.h

```
#ifndef DEFS
	#define DEFS
	#include "defs.h"
#endif
typedef nodep list;

list createlist();
list insert(list, int);
list delete(list, int);
void printlist(list);
```

stack.h

```
#ifndef DEFS
	#define DEFS
	#include "defs.h"
#endif
typedef nodep stack;

stack create();
stack push(stack, int);
stack pop(stack);
void print(stack);
```

main.c

```
#include "list.h"
#include "stack.h"
int main()
{
	stack S;
	list L;
	/* Some code here */
}
```

No error will be there if you compile now.
list.h and stack.h can be #included in any order in other files!

Some other gcc options you should study now

–W –Wall includes the following (among others).

Some of these have many subcategories.

–Wcomment	Warn about nested comments.
–Wformat	Warn about type mismatches in scanf and printf.
–Wunused	Warn about unused variables.
–Wimplicit	Warn about functions used before declaration.
–Wreturn-type	Warn about returning void for functions with non-void return values.

–Wall does not include the following (among others).

–Wconversion	Warn about implicit type conversions.
–Wshadow	Warn about shadowed variables.
–Werror	Convert warnings to errors.

In general, it is always advisable to compile with gcc –Wall to get most warnings. **Warnings should not be ignored**, some can be catastrophic when you run the executable

–O Set the optimization level

–O0 No optimization (default behavior, useful when debugging).

–O1, –O2, –O3 Various levels of optimization. Optimization is time-consuming, and can be used only during the last stages of development.

–Os Optimize (reduce) the size of the code.

–v Verbose mode of compilation.

– –help Print help message for usage.

– –version Print the gcc version.

For optimization options, no details are needed, just understand for now that code can be optimized for time and space

Creating Static Libraries

Introduction

- A library is a pre-compiled archive of object files
  - These can be linked to user codes during compilation or during runtime
- Example: The math library consists of the following.
  - Definition of data types: float, double, . . .
  - Prototypes of Functions: pow, sqrt, atan, cosh, abs, . . .
  - Definition of Constants: M_PI, M_E, M_LOG2E, M_SQRT2, . . .
  - A precompiled archive of implementations of the math functions defined
- The first three are in the header file math.h. You include it in your program `#include <math.h>`
- The fourth one, precompiled math library, is needed for linking to your final executable.
  - You specify the option -lm for this linking when you compile your C program

Types of Libraries

- Static libraries
  - Prefix: *lib*
  - Extension: *.a*
  - Example: the static math library has the name *libm.a*
  - Functions from static libraries are actually inserted in the final executable during linking
    - Pro: the executable can run without needing anything else when it is run
    - Con: Since all the library code is added to your code, size of executable can be large
      - Example: Even if you use only the sqrt() function from the math library in your program, the entire code of all functions in the math library will be added to the executable
- Shared (or dynamic) libraries
  - We will see this in the next class

Building a static library

- Write your programs in .h and .c files as before
  - IMPORTANT: No main() function should be there in any of the C files
    - The main() function will be there in the C program that uses this library
- Compile each of the C files using the -c option of gcc to generate .o (object) files (already seen how to do this)
- Combine all the .o files into a static library using the ar command
  - Must name the library as per the naming convention mentioned in the last slide

Example

- Suppose you have the files stack.h, queue.h, stack.c, and queue.c that implements stack and queue data structures
  - All typedefs and function prototypes for stack and queue datatype are in stack.h and queue.h respectively
  - All functions on stack and queue are in stack.c and queue.c respectively
  - No main() function in any file
- We want to build a static library libstaque.a. This will contain all the stack and queue functions defined in stack.c and queue.c

```
$ gcc -c stack.c
$ gcc -c queue.c
$ ar rcs libstaque.a stack.o queue.o
$ ls –l libstaque.a
-rw-r--r-- 1 abhij abhij 7046 Dec 24 18:25 libstaque.a
```

Using the library created

- Suppose you write a C program in a file staquecheck.c that declares variables of type stack and queue and calls the stack and queue functions in the library
  - staquecheck.c will have a main() function
  - It will include the files stack.h and queue.h using #include
  - Compile it with -l option to create the final executable file a.out
    - gcc staquecheck.c -lstaque
    - Note that you didn't have to type the full name libstaque.a. The prefix lib and the suffix .a is automatically added by gcc
  - How will the compiler know in which directory libstaque.a is?
    - You can tell the compiler with the -L option of gcc (same format as for -I for .h files)
    - You can set the LIBRARY_PATH shell variable (same way as for C_INCLUDE_PATH for header files)

- So why no -L was needed when you used -lm in your programs?
  - There are default directories that the linker automatically looks at by default
    - Similar to what happens for #include <..>
  - math library libm.a is placed in one of these default directories
- For using printf/scanf etc., which are in C library, you did not need to specify even -l option. Why?
  - C standard library is linked by default
- Actually, by default, dynamic libraries are used for both C standard library and math libraries. However, for them also, the above stands

Summary of things you must know from today

- The notion of shell variables, commands for adding/viewing/changing them
- Working with PATH, C_INCLUDE_PATH and LIBRARY_PATH
- The different stages of compilation
  - Use of cpp and gcc -E to see the output after preprocessing
  - Compilation into object files (gcc -c)
  - Linking object files to make executables (gcc, with –o option for naming the executable file)
- How to divide your code into .h and .c files
- How to compile multiple C files into a single executable file
- How to write your own header files and include them in your programs
  - Use of both <> and “” formats for including
  - Use of gcc -I and C_INCLUDE_PATH to specify header file directories

- Basic notion of a macro and its benefits
- Using #define, #undef, #ifdef, #ifndef, #else, #endif
- Additional gcc options listed for study
- Building a static library with ar command
- Linking a static library with another program using –l option of gcc

Practice in Lab

- Create two subdirectories, bin and sbin under your home directory
- Print the value of the PATH variable in your shell
- Add the two subdirectories to the path
  - Remember that other directories already there should not be deleted
- Print the value of the PATH variable again to check it is changed properly
- Change (replace) the value of the HOME shell variable to the directory $HOME/sbin
- Print the value of HOME to check it is changed (to what?)
- Do a cd ~ to go to your home directory
- Do pwd. What directory are you in?
- Change HOME again to its original value and print and check
- Open another terminal window and print the PATH variable from there. Do you see the changed PATH value (from your change above) or not? Explain what you see

- Write a simple C program hello.c to print "Hello World" (can type the one given earlier with the comment)
- Run the preprocessor cpp on hello.c directly and store the output in a file hello-1.i
- Run the preprocessor cpp on hello.c using gcc -E and store the output in a file hello-2.i
- Are the two files hello-1.i and hello-2.i exactly the same? Use the diff command to check
- Open hello-1.i in a text editor. Try going through the file
  - It is very long, so do not go through line by line. Just see what kind of things you see there
  - Go to the last few lines and explain what you see
- Open the file /usr/include/stdio.h in another editor. Do you see any relation between the contents of stdio.h and hello-1.i?
- What other files do you see in /usr/include?

- Generate the file hello.o from hello.c
- Compile hello.o to generate the executable hello.exe. Run it to see Hello World is printed correctly.
- Create a subdirectory under home directory named stq_library. Change to the subdirectory.
- Type the files defs.h stack.h queue.h stack.c queue.c and staquecheck.c from the code given, all in the stq_library subdirectory
  - Study carefully how the code is divided and match it against the guidelines mentioned
- Combine all of them into a single executable file using a single gcc command
- Run the executable and test it (easy to see from the code what it does)
- Delete the executable
- Create it again, but this time after creating each .o file separately first and then linking them finally
- Again run the executable to check, then delete it. Delete all .o files also

- Go to your home directory
- Create a subdirectory named include under it. Move (not copy) all the .h files from the stq_library subdirecory to ~/include subdirectory
- Change to stq_library subdirectory
- Compile all the files again using a single gcc command. Does the compilation succeed? Explain.
- Use the -I option to compile and solve the problem you saw above.
- Print the value of the C_INCLUDE_PATH variable and remember it
- Change it to add $HOME/include to it
- Now compile again without the -I option to generate the executable. Test again.

- Write a C program that will read in two integers x and y first. It will print x^y if a macro POW is defined, will print simply x+y otherwise.
  - Use #define to define the macro inside your program to do this
    - Test with both #define in the code and not in the code
  - Do not use #define. Use –D option during compilation to do this
- Write a C program that will print the value of a macro MY_STR. The value will be given using –D option during compile time. Compile with each of the following options and try to explain what you see. Use the pre-processor output to see what is getting included to explain. You can run like cpp –DMY_STR…
  - Use –DMY_STR=my name   to print the string “my name”
  - Use –DMY_STR=“my name”   to print the string “my name”
  - Use –DMY_STR=‘”my name”’   to print the string “my name”
- Exercise each of the –W options separately (not –Wall) of gcc by compiling hello.c. To see the effect, you need to deliberately add some mistakes in it. From the name of the options, can you try what mistakes to add to get a warning of that type?

- Go back to stq_library subdirectory again
- Remove all .o files
- Create the .o files for stack.c and queue.c only
- Combine the .o files into a static library named libstaque.a
- Compile staquecheck.c linking with libstaque.c to create an executable staquecheck.exe. Test to see it runs properly. Then delete it.
- Move (not copy) libstaque.a to ~/sbin
- Compile staquecheck.c again linking with libstaque.c to create an executable staquecheck.exe. Do you see any error?
- Use the –L option to compile without error
- Print the value of the shell variable LIBRARY_PATH
- Add ~/sbin to it
- Now compile again without the –L option and test everything.