1 Introduction

Meh, I don't like writing narratives. We are here to start with turning code into software. As a starting place, I will cover the basics of c++ compilation without the use of any sort of IDE.

2 gcc

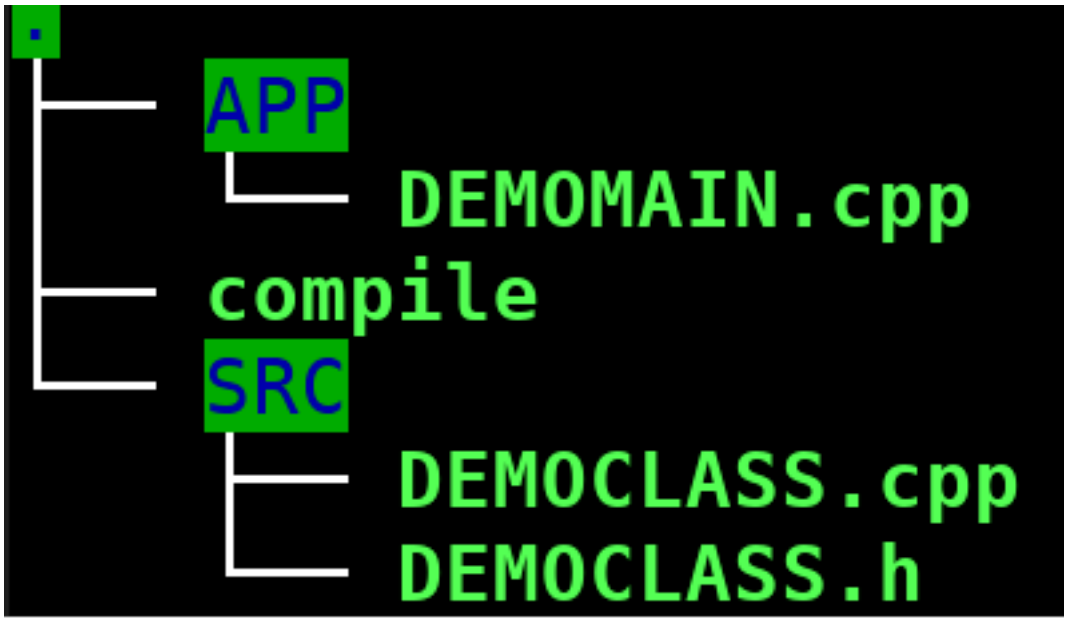
The formal description of the gcc tool is:

> GCC stands for "GNU Compiler Collection". GCC is an integrated distribution of compilers for several major programming languages. These languages currently include C, C++, Objective-C, Objective-C++, Fortran, Ada, D, Go, and BRIG (HSAIL).

The gcc tool is a very complex tools with all kinds of uses, most of which I won't cover here. This is ultimately unnecessary since there is this neat resource with has been written by the people who maintain this tool. Here is the reference: Super Awesome Reference.
You will need to know which version of gcc you have. To learn this simply execute: `gcc --version`
Given these things, you should be able to figure out everything you would ever need to know about gcc. To cover some basic usage of this command, I will demonstrate it's use.

Example: Oh, that's Wow, Morty. Wow. What an exciting life you lead. Let's go. In and out, 20 minutes adventure.

Given a directory structure like so:



Then our compile script holds the simple command:

```
gcc -x c++ APP/DEMOMAIN.cpp SRC/DEMOCLASS.h SRC/DEMOCLASS.cpp -o DEMOEXE -lstdc++
```

Lets look at this command. The -x option allows us to select the language for the following input files (rather than letting the compiler choose a default based on the file name suffix). This is necessary as by default, the gcc tool will treat all files as c files. Next we have n paths to different source files. In this case we have 3. Next we have the -o flag which allows us to name the executable file. Finally, since the stdc++ library isn't linked to by default with the gcc compile, we have to state it explicitly. With all that out of the way, if the compile file has executable permission, then we can run it to compile the files.

3 Includes

When compiling a program using the `-v` we can see things like where the compiler is looking to find libraries that you might link to. Adding the option `-Wl,--trace` to the end of the gcc command will tell you where the libraries are which you are linking to.

4 Make

Next, we come to make files.
Makefiles are like my compile script, however it's a standardized way for handling larger systems of code. Once again we are playing with a pretty complicated tool here however it also has a great reference maintained by the people who maintain the make tool. Here is the reference: Dank Reference Time
So to use a Makefile, generally what you would call is just: `make` while in a directory containing a file called Makefile. This will create an compile command like I demonstrated earlier and execute it.
Next up is a demo of a Makefile which lives in the same directory as our compile script though totally independent of that script.

Example: Yo, I heard you like compilation automation.

```
FLAGS=-x c++
LIBS=-lstdc++

all : DEMOMAIN.o DEMOCLASS.o
gcc -o DEMOEXE DEMOMAIN.o DEMOCLASS.o $(LIBS)

DEMOMAIN.o : APP/DEMOMAIN.cpp
gcc $(FLAGS) -c APP/DEMOMAIN.cpp $(LIBS)

DEMOCLASS.o : SRC/DEMOCLASS.h SRC/DEMOCLASS.cpp
gcc $(FLAGS) -c SRC/DEMOCLASS.h SRC/DEMOCLASS.cpp $(LIBS)

clean :
rm DEMOMAIN.o DEMOCLASS.o DEMOEXE
```

Here, both FLAGS and LIBS are called macros and are not themselves executed. They are variables you can set and call using the `$(VAR)` syntax.
The `all` tag is the first target and therefore the default target.
The general syntax for targets is:
`target: prerequisites`
`<TAB> recipe`
The tab is necessary, latex just didn't want to add that information with how I am linking the file. One great advantage of handling the compilation this way is that if a file changes, make will only rebuild that part of the system and not the entire thing. This doesn't matter when dealing with smaller problems but is extremely helpful when dealing with larger projects. Each of these targets can be called individually by using the following syntax:
`make target1 target2 ... targetN`
However if just make is called then it will run the default target.

5 CMake

This is great and all, but what if you didn't want to generate the makefile yourself and instead wanted a way to just list the files and libraries you want to use and have the Makefile generated for you? Here comes cmake with the cure for what ails you. Cmake is another tool which is very complex and with our complex tools comes a sweet reference you can use to learn all kinds of stuff about it: Sweet Reference Time. We are here for demonstrations however and so here we go.

Example 1: I put some automation in your automation so you can automate while you automate.

For this you will need a file called: `CMakeLists.txt` in the same directory as the other compilation files.

```
cmake_minimum_required(VERSION 3.8.2)
project(DEMO0)
set(CMAKE_CXX_STANDARD 17)

add_executable(DEMOEXE APP/DEMOMAIN.cpp
   SRC/DEMOCLASS.h SRC/DEMOCLASS.cpp)

target_link_libraries(DEMOEXE)
```

The whitespace format is messed up again here.
Calling `cmake .` here will generate everything needed to build the project.
After that you can call `make` and it will generate the executable. This is a much simpler file format and can be expanded to reference other libraries needed.

Example 2: Return of the King

This is an actual example which I am currently using to build all the source code for my main project DSYS.

```
cmake_minimum_required(VERSION 3.8.2)
project(DLRGSFTSYS)
set(CMAKE_CXX_STANDARD 17)

#DKINECT STUFF
set(THREADS_USE_PTHREADS_WIN32 true)
find_package(Threads)
include_directories(${THREADS_PTHREADS_INCLUDE_DIR})

find_path(LIBUSB_INCLUDE_DIR NAMES libusb.h PATH_SUFFIXES "include" "libusb" "libusb-1.0")
find_library(LIBUSB_LIBRARY NAMES usb PATH_SUFFIXES "lib" "lib32" "lib64")

#find_package(libfreenect REQUIRED)
#include_directories( ${libfreenect_INCLUDE_DIRS} )
#include_directories( /run/media/drake/Seagate Expansion Drive/BackupDesktop/Code/DSYS/DRIVERS
#include_directories( /run/media/drake/Seagate Expansion Drive/BackupDesktop/Code/DSYS/DRIVERS

#Add arrayfire
```

```
find_package(ArrayFire REQUIRED)

#Add GTK3
FIND_PACKAGE(PkgConfig REQUIRED)
PKG_CHECK_MODULES(GTK3 REQUIRED gtk+-3.0)

INCLUDE_DIRECTORIES(${GTK3_INCLUDE_DIRS})
LINK_DIRECTORIES(${GTK3_LIBRARY_DIRS})

#Add Boost Libraries
find_package( Boost REQUIRED COMPONENTS program_options regex filesystem graph)
include_directories( ${Boost_INCLUDE_DIRS} )

#Add OpenGL Libraries
find_package(OpenGL REQUIRED)
find_package(GLUT REQUIRED)
include_directories( ${OPENGL_INCLUDE_DIRS} ${GLUT_INCLUDE_DIRS} ${LIBUSB_INCLUDE_DIR})

#Add mysql Libraries


add_executable(exe DSRC/DOBJ/DOBJ.h DSRC/DOBJ/DOBJ.cpp
   DSRC/DGTKWINDOW/DGTKWINDOW.h DSRC/DGTKWINDOW/DGTKWINDOW.cpp
   DSRC/DGLUTWINDOW/DGLUTWINDOW.h DSRC/DGLUTWINDOW/DGLUTWINDOW.cpp DSRC/DGLUTWINDOW/GLUTHELPER
#    DSRC/DKINECT/DKINECT.h DSRC/DKINECT/DKINECT.cpp
   DSRC/DNETWORK/DNETWORK.h DSRC/DNETWORK/DNETWORK.cpp
   DSRC/DSERVERTCP/DSERVERTCP.h DSRC/DSERVERTCP/DSERVERTCP.cpp
   DSRC/DCLIENTTCP/DCLIENTTCP.h DSRC/DCLIENTTCP/DCLIENTTCP.cpp
   DSRC/DSERVERUDP/DSERVERUDP.h DSRC/DSERVERUDP/DSERVERUDP.cpp
   DSRC/DCLIENTUDP/DCLIENTUDP.h DSRC/DCLIENTUDP/DCLIENTUDP.cpp
   DSRC/DML/DML.h DSRC/DML/DML.cpp DSRC/DML/DOP.cpp
#    DSRC/DSQL/DSQL.h DSRC/DSQL/DSQL.cpp
   DSRC/DLLC/DLLC.h DSRC/DLLC/DLLC.cpp
   DSRC/DGRAPHICS/DGRAPHICS.h DSRC/DGRAPHICS/DGRAPHICS.cpp
   DAPP/main.cpp )

target_link_libraries(exe
  ArrayFire::afcpu
#   freenect
  ${GTK3_LIBRARIES}
  ${OPENGL_LIBRARIES}
  ${GLUT_LIBRARY}
  ${Boost_LIBRARIES}
  ${CMAKE_THREAD_LIBS_INIT}
  ${MATH_LIB} ${LIBUSB_LIBRARY}
#   ${libfreenect_LIBRARIES}
    mysqlclient
    mysqlcppconn
    crypto
    ssl
```

```
  resolv
)
```