gcc Cheat Sheet

Oft used gcc commands
======================

```
$g++ --version           Returns the version of gcc
$g++ -c sourceFile.cpp   Compiles a single source file into an object file
$g++ -o Program.exe source1.o source2.o source3.o    Links a series of object fi
les into
                                                     a single executable.
                                                     -o means "output" and is fo
llowed by a filename
To include debug symbols, use -g during compiling:
$g++ -c -g sourceFile.cpp    compiles a single soruce file with debug info into
an object file
```

For the following examples, assume this directory structure:

```
/Dir
/Dir/source   [Contains:  File1.cpp   File2.cpp]
/Dir/include  [Contains:  File1.h     File1.h]
/Dir/lib      [will contain output libraries]
/Dir/obj      [temporarily used to contain .o object files
```

Create a static (.a) library from two source files:

```
g++ -c -I./include ./source/File1.cpp -o ./obj/File1.o
g++ -c -I./include ./source/File2.cpp -o ./obj/File2.o
ar rc ./lib/libLib1.a ./obj/File1.o ./obj/File2.o
```

Create a shared (.so) library from two source files:

```
g++ -c -fPIC -o ./obj/File1.o -I./include ./source/File1.cpp
g++ -c -fPIC -o ./obj/File2.o -I./include ./source/File2.cpp
g++ -shared -Wl,-soname,./lib/libLib1.so.1 -o ./lib/libLib1.so.1.0 ./obj/File1.o
 ./obj/File2.o
```

Note: the linker will prefer (use first) shared libraries (.so) if found
over static (.a) libraries, unless explicitly directed to use static libs.

Create an executable binary from a source file + **static** library, use the
-static switch. Will increase size, but allow exe to run on systems w/out
a shared library.
First compile the file, then create the exe by linking in Lib1,
placing the binary in ./bin/main.exe

```
g++ -c -I./include -I../Lib1/include ./source/main.cpp -o ./obj/main.o
g++ -static ./obj/main.o -L../Lib1/lib -lLib1 -o ./bin/main.exe
```

Create an executable binary from a source file + **dynamic** library:
Note: the .so library must be present during compile and run time, unless
a shared .a version is found. If a shared .a version is found (after searching
and failing to find a .so library), it will be linked in statically.
This means that this particular .so library need not be present at run
time, as it is part of the exe. This increases the size (often times considerab
ly)
of the resultant exe.

First compile the file, then create the exe by linking in Lib1,

```
 placing the binary in ./bin/main.exe
-----
g++ -c -I./include -I../Lib1/include ./source/main.cpp -o ./obj/main.o
g++ ./obj/main.o -L../Lib1/lib -lLib1 -o ./bin/main.exe

-------------------------
```

Notes on building, order, undefined reference, etc...

Assume you have two libraries and an exe project: Lib1, Lib2, and exe.
  Lib2 depends on Lib1 (ie, uses functions / classes from Lib1), and the
  exe project depends on both Libs.

You can build Lib1 error free even if you don't define all the functions
  that have been declared within it.

You can build Lib2 error free even if Lib1 hasn't defined all symbols. This
  is true whether or not Lib2 expicitly makes calls to the missing functions.

Once you try to link the Libs into the exe, however, you will get the
  error "undefined reference".

If the function is undefined in Lib1, and then used in Lib2 in a function, and
  then that function from Lib2 is used in exe, it will reference Lib2.

If it is undefined in Lib1 and used directly from exe (without being used in
  Lib2), it will reference only the missing reference without mentioning Lib2.

If the function is undefined and never used, you will not receive any errors.

If an undefined function from Lib1 is used in Lib2, and that function from
  Lib2 is *not* used in the exe, then you will not receive any errors.

You can build either Lib1 or Lib2 first, no matter which library depends on
  which. But both must be built before building the exe.

```
$ldd   - a CL utility to determine and print shared library dependencies

$ld.so - the dynamic linker / loader
```