CME 211: Lecture 22

Topics:

- Compilation process
- Make for building software

Compilation

- Although you can go from source code to an executable in one command, the process is actually made up of 4 steps
- Preprocessing
- Compilation
- Assembly
- Linking
- `g++` and `clang++` (and `gcc` or `clang` for C code) are driver programs that invoke the appropriate tools to perform these steps
- This is a high level overview. The compilation process also includes optimization phases during compilation and linking.

Behind the scenes

We can inspect the compilation process in more detail with the `-v` compiler argument. `-v` typically stands for "verbose".

Output:

```
$ g++ -v -Wall -Wextra -Wconversion src/hello1.cpp -o src/hello1
Apple LLVM version 7.3.0 (clang-703.0.31)
Target: x86_64-apple-darwin15.6.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/clang" -cc1 -t
clang -cc1 version 7.3.0 (clang-703.0.31) default target x86_64-apple-darwin15.6.0
ignoring nonexistent directory "/usr/include/c++/v1"
#include "..." search starts here:
#include <...> search starts here:
 /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/
 /usr/local/include
 /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../lib/clang/7.
 /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include
 /usr/include
 /System/Library/Frameworks (framework directory)
 /Library/Frameworks (framework directory)
End of search list.
 "/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/ld" -demangle
```

Splitting up the steps manually

GNU compiler flags:

- `-E`: preprocess
- `-S`: compile
- `-c`: assemble

Output:

```
$ cat src/hello1.cpp
#include <iostream>

int main() {
  std::cout << "Hello, CME 211!" << std::endl;
  return 0;
}
$ g++ -E -o src/hello1.i src/hello1.cpp
$ g++ -S -o src/hello1.s src/hello1.i
clang: warning: treating 'cpp-output' input as 'c++-cpp-output' when in C++ mode, this behavior is depr
$ g++ -c -o src/hello1.o src/hello1.s
$ g++ -o src/hello1 src/hello1.o
$ ./src/hello1
Hello, CME 211!
```

Preprocessing

- The preprocessor handles the lines that start with `#`
- `#include`
- `#define`
- `#if`
- etc.
- You can invoke the preprocessor with the `cpp` command

Preprocessed file

From `src/hello1.i`:

```
# 1 "hello1.cpp"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 30 "/usr/include/stdc-predef.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/predefs.h" 1 3 4
# 31 "/usr/include/stdc-predef.h" 2 3 4
# 1 "<command-line>" 2
# 1 "hello1.cpp"
# 1 "/usr/include/c++/4.8/iostream" 1 3
# 36 "/usr/include/c++/4.8/iostream" 3

// approximately 17,500 lines omitted!

int main()
{
std::cout << "Hello" << std::endl;
return 0;
}
```

Compilation

- Compilation is the process of translating source code to assembly commands
- The assembly commands are still human readable text (if the human knows assembly)

From `src/hello.s`:

```
main:
.LFB1020:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0, %esi
    movl    $_ZSt4cout, %edi
    call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
    movl    $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_, %esi
    movq    %rax, %rdi
    call    _ZNSolsEPFRSoS_E
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

Assembly

- This step translates the text representation of the assembly instructions into the binary machine code in a `.o` file
- `.o` files are called object files
- Linux uses the Executable and Linkable Format (ELF) for these files
- If you try to look at these files with a normal text editor you will just see garbage, intermixed with a few strings
- Sometimes it is helpful to inspect object files with the `nm` command to see what symbols are defined:

Output:

```
$ nm ./src/hello1.o
0000000000000cac s GCC_except_table2
0000000000000cec s GCC_except_table3
0000000000000d9c s GCC_except_table5
                 U __Unwind_Resume
                 U __ZNKSt3__16locale9use_facetERNS0_2idE
                 U __ZNKSt3__18ios_base6getlocEv
0000000000000c80 S __ZNSt3__111char_traitsIcE11eq_int_typeEii
0000000000000ca0 S __ZNSt3__111char_traitsIcE3eofEv
00000000000005d0 S __ZNSt3__111char_traitsIcE6lengthEPKc
                 U __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEE6__initEmc
                 U __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEED1Ev
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE3putEc
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE5flushEv
```

```
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE6sentryC1ERS3_
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE6sentryD1Ev
00000000000005f0 S __ZNSt3__116__pad_and_outputIcNS_11char_traitsIcEEEENS_19ostreambuf_iteratorIT_T0_EE
00000000000001a0 S __ZNSt3__124__put_character_sequenceIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0
                 U __ZNSt3__14coutE
00000000000000a0 S __ZNSt3__14endlIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0_EES7_
                 U __ZNSt3__15ctypeIcE2idE
                 U __ZNSt3__16localeD1Ev
                 U __ZNSt3__18ios_base33__set_badbit_and_consider_rethrowEv
                 U __ZNSt3__18ios_base5clearEj
0000000000000050 S __ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc
                 U __ZSt9terminatev
0000000000000c70 S ___clang_call_terminate
                 U ___cxa_begin_catch
                 U ___cxa_end_catch
                 U ___gxx_personality_v0
0000000000000000 T _main
                 U _memset
                 U _strlen
```

Linking

- Linking is the process of building the final executable by combining (linking) the `.o` file(s), and possibly library files as well
- The linker makes sure all of the required functions are present
- If for example `foo.o` contains a call to a function called `bar()`, there has to be another `.o` file or library file that provides the implementation of the `bar()` function

Linking example

src/foobar.hpp:

```
#pragma once

void bar(void);
void foo(void);
```

src/foo.cpp:

```
#include <iostream>

void foo(void) {
  std::cout << "Hello from foo" << std::endl;
}
```

src/bar.cpp:

```
#include <iostream>

void bar(void) {
  std::cout << "Hello from bar" << std::endl;
}
```

src/main.cpp:

```cpp
#include "foobar.hpp"

int main() {
  foo();
  bar();
  return 0;
}
```

Linking example

Inspect the files:

Output:

```
$ ls src
bar.cpp
bar.o
ex1
ex2
ex3
ex4
foo.cpp
foo.o
foobar.hpp
hello1
hello1.cpp
hello1.i
hello1.o
hello1.s
hw6
hw6.cpp
hw6.hpp
main
main.cpp
main.o
stanford.jpg
test.jpg
```

Compile and assemble source files, but don't link:

Output:

```
$ g++ -c src/foo.cpp -o src/foo.o
$ g++ -c src/bar.cpp -o src/bar.o
$ g++ -c src/main.cpp -o src/main.o
```

Let's inspect the output:

Output:

```
$ ls src/*.o
ls: src/*.o: No such file or directory
```

What symbols are present in the object files?

Output:

```
$ nm src/foo.o
0000000000000d2c s GCC_except_table2
```

```
0000000000000d6c s GCC_except_table3
0000000000000e1c s GCC_except_table5
                 U __Unwind_Resume
0000000000000000 T __Z3foov
                 U __ZNKSt3__16locale9use_facetERNS0_2idE
                 U __ZNKSt3__18ios_base6getlocEv
0000000000000d00 S __ZNSt3__111char_traitsIcE11eq_int_typeEii
0000000000000d20 S __ZNSt3__111char_traitsIcE3eofEv
0000000000000610 S __ZNSt3__111char_traitsIcE6lengthEPKc
                 U __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEE6__initEmc
                 U __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEED1Ev
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE3putEc
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE5flushEv
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE6sentryC1ERS3_
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE6sentryD1Ev
0000000000000630 S __ZNSt3__116__pad_and_outputIcNS_11char_traitsIcEEEENS_19ostreambuf_iteratorIT_T0_EES
00000000000001a0 S __ZNSt3__124__put_character_sequenceIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0_
                 U __ZNSt3__14coutE
0000000000000090 S __ZNSt3__14endlIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0_EES7_
                 U __ZNSt3__15ctypeIcE2idE
                 U __ZNSt3__16localeD1Ev
                 U __ZNSt3__18ios_base33__set_badbit_and_consider_rethrowEv
                 U __ZNSt3__18ios_base5clearEj
0000000000000040 S __ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc
                 U __ZSt9terminatev
0000000000000cf0 S ___clang_call_terminate
                 U ___cxa_begin_catch
                 U ___cxa_end_catch
                 U ___gxx_personality_v0
                 U _memset
                 U _strlen
$ nm src/bar.o
0000000000000d2c s GCC_except_table2
0000000000000d6c s GCC_except_table3
0000000000000e1c s GCC_except_table5
                 U __Unwind_Resume
0000000000000000 T __Z3barv
                 U __ZNKSt3__16locale9use_facetERNS0_2idE
                 U __ZNKSt3__18ios_base6getlocEv
0000000000000d00 S __ZNSt3__111char_traitsIcE11eq_int_typeEii
0000000000000d20 S __ZNSt3__111char_traitsIcE3eofEv
0000000000000610 S __ZNSt3__111char_traitsIcE6lengthEPKc
                 U __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEE6__initEmc
                 U __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEED1Ev
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE3putEc
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE5flushEv
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE6sentryC1ERS3_
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE6sentryD1Ev
0000000000000630 S __ZNSt3__116__pad_and_outputIcNS_11char_traitsIcEEEENS_19ostreambuf_iteratorIT_T0_EES
00000000000001a0 S __ZNSt3__124__put_character_sequenceIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0_
                 U __ZNSt3__14coutE
0000000000000090 S __ZNSt3__14endlIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0_EES7_
                 U __ZNSt3__15ctypeIcE2idE
                 U __ZNSt3__16localeD1Ev
```

```
                 U __ZNSt3__18ios_base33__set_badbit_and_consider_rethrowEv
                 U __ZNSt3__18ios_base5clearEj
0000000000000040 S __ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc
                 U __ZSt9terminatev
0000000000000cf0 S ___clang_call_terminate
                 U ___cxa_begin_catch
                 U ___cxa_end_catch
                 U ___gxx_personality_v0
                 U _memset
                 U _strlen
$ nm src/main.o
                 U __Z3barv
                 U __Z3foov
0000000000000000 T _main
```

What happens if we try to link `main.o` into an executable with out pointing to the other object files?

Output:

```
$ g++ src/main.o -o src/main
Undefined symbols for architecture x86_64:
  "bar()", referenced from:
      _main in main.o
  "foo()", referenced from:
      _main in main.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Ahhh, linker errors! Let's do it right:

Output:

```
$ g++ src/main.o src/foo.o src/bar.o -o src/main
$ ./src/main
Hello from foo
Hello from bar
```

Libraries

- Libraries are really just a file that contain one or more `.o` files
- On Linux these files typically have a `.a` (static library) or `.so` (dynamic library) extension
- `.so` files are analogous to `.dll` files on Windows
- `.dylib` files on Mac OS X and iOS are also very similar to `.so` files
- Static libraries are factored into the executable at link time in the compilation process.
- Shared (dynamic) libraries are loaded up at run time.

JPEG Example

From `src/hw6.cpp`:

```
// code omitted

#include <jpeglib.h>
```

```cpp
#include "hw6.hpp"


void ReadGrayscaleJPEG(std::string filename, boost::multi_array<unsigned char,2> &img)
{
  /* Open the file, read the header, and allocate memory */

  FILE *f = fopen(filename.c_str(), "rb");
  if (not f)
  {
    std::stringstream s;
    s << __func__ << ": Failed to open file " << filename;
    throw std::runtime_error(s.str());
  }
  // code omitted
}

// code omitted

#ifdef DEBUG
int main()
{
  boost::multi_array<unsigned char,2> img;
  ReadGrayscaleJPEG("stanford.jpg", img);
  WriteGrayscaleJPEG("test.jpg", img);

  return 0;
}
#endif /* DEBUG */
```

Let's try to compile:

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/hw6.cpp -o src/hw6
Undefined symbols for architecture x86_64:
  "_jpeg_CreateCompress", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<
  "_jpeg_CreateDecompress", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cl
  "_jpeg_destroy_decompress", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cl
  "_jpeg_finish_compress", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<
  "_jpeg_finish_decompress", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cl
  "_jpeg_read_header", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cl
  "_jpeg_read_scanlines", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cl
  "_jpeg_set_defaults", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<
  "_jpeg_set_quality", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<
  "_jpeg_start_compress", referenced from:
```

```
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<
  "_jpeg_start_decompress", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cl
  "_jpeg_std_error", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cl
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<
  "_jpeg_stdio_dest", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<
  "_jpeg_stdio_src", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cl
  "_jpeg_write_scanlines", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<
  "_main", referenced from:
     implicit entry/start for main executable
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

That did not work. The linker looks for the `main` symbol when trying to build and executable. This linker also cannot find all of the symbols from the JPEG library.

Let's find the `jpeglib.h` header file:

Output:

```
$ locate jpeglib.h
/usr/local/Cellar/jpeg/8d/include/jpeglib.h
/usr/local/include/jpeglib.h
```

Let's find `libjpeg`:

Output:

```
$ locate libjpeg
/Applications/Xcode.app/Contents/Applications/Application Loader.app/Contents/itms/java/lib/libjpeg.dyli
/usr/local/Cellar/jpeg/8d/lib/libjpeg.8.dylib
/usr/local/Cellar/jpeg/8d/lib/libjpeg.a
/usr/local/Cellar/jpeg/8d/lib/libjpeg.dylib
/usr/local/Homebrew/Library/Taps/homebrew/homebrew-core/Aliases/libjpeg
/usr/local/Homebrew/Library/Taps/homebrew/homebrew-core/Aliases/libjpeg-turbo
/usr/local/lib/libjpeg.8.dylib
/usr/local/lib/libjpeg.a
/usr/local/lib/libjpeg.dylib
/usr/local/lib/python3.5/site-packages/PIL/.dylibs/libjpeg.9.dylib
```

Note that the library files may be in a different location on your system.

Now let's compile:

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/hw6.cpp -o src/hw6 -DDEBUG -I/usr/local/include -L/usr/l
$ ./src/hw6
```

- `-I/usr/local/include`: look in this directory for include files (optional in this case)
- `-L/usr/local/lib`: look in this directory for library files (optional in this case, maybe required on Ubuntu)
- `-ljpeg`: link to the `libjpeg.{a,so}` file (not optional here)

Make

- Utility that compiles programs based on rules read in from a file called Makefile
- Widely used on Linux/Unix platforms
- Setup and maintenance of Makefile(s) can become rather complicated for major projects
- We will look at a few simple examples

Example source files

`src/ex1/sum.cpp`:

```
#include "sum.hpp"

double sum(double a, double b) {
  double c = a + b;
  return c;
}
```

`src/ex1/sum.hpp`:

```
#pragma once

double sum(double a, double b);
```

`src/ex1/main.cpp`:

```
#include <iostream>

#include "sum.hpp"

int main() {
  double a = 2., b = 3., c;

  c = sum(a,b);
  std::cout << "c = " << c << std::endl;

  return 0;
}
```

Example `makefile`

`src/ex1/makefile`:

```
main: main.cpp sum.cpp sum.hpp
    g++ -Wall -Wextra -Wconversion -o main main.cpp sum.cpp
```

Anatomy of a `make` rule:

```
target: dependencies
    build_command
```

- `target`: is the thing you want the rule to create. The target should be a file that will be created in the file system. For example, the final executable or intermediate object file.
- `dependencies`: space separated list files that the target depends on (typically source or header files)

- `build_command`: a **tab-indented** shell command (or sequence) to build the target from dependencies.

Let's run the example

Let's run make!

```
$ ls
main.cpp  makefile  sum.cpp  sum.hpp
$ make
g++ -Wall -Wextra -Wconversion -o main main.cpp sum.cpp
$ ls
main  main.cpp  makefile  sum.cpp  sum.hpp
$ make
make: 'main' is up to date.
$
```

File changes

Make looks at time stamps on files to know when changes have been made and will recompile accordingly (from `src/ex1` directory):

```
$ make
make: 'main' is up to date.
$ touch main.cpp
$ make
g++ -Wall -Wextra -Wconversion -o main main.cpp sum.cpp
$ touch sum.hpp
$ make
g++ -Wall -Wextra -Wconversion -o main main.cpp sum.cpp
$ make
make: 'main' is up to date.
```

Make variables, multiple targets, and comments

`src/ex2/makefile`:

```
# this is a makefile variable, note := for direct assignment
CXX := g++

# this is a makefile comment
#CXXFLAGS := -Wall -Wextra -Wconversion
#CXXFLAGS := -Wall -Wextra -Wconversion -g Commented out
CXXFLAGS := -Wall -Wextra -Wconversion -fsanitize=address

main: main.cpp sum.cpp sum.hpp
    $(CXX) $(CXXFLAGS) -o main main.cpp sum.cpp

# here is a target to clean up the output of the build process
.PHONY: clean
clean:
    $(RM) main
```

Output (from `src/ex2` directory):

```
$ ls
main.cpp  makefile  sum.cpp  sum.hpp
$ make
g++ -Wall -Wextra -Wconversion -fsanitize=address -o main main.cpp sum.cpp
$ ls
main  main.cpp  makefile  sum.cpp  sum.hpp
$ make clean
rm -f main
$ ls
main.cpp  makefile  sum.cpp  sum.hpp
```

Individual compilation of object files

src/ex3/makefile:

```
CXX := g++
CXXFLAGS := -O3 -Wall -Wextra -Wconversion -std=c++11

TARGET := main
OBJS := main.o sum.o foo.o bar.o
INCS := sum.hpp foobar.hpp

$(TARGET): $(OBJS)
    $(CXX) -o $(TARGET) $(OBJS)

# this is a make pattern rule
%.o: %.cpp $(INCS)
    $(CXX) -c -o $@ $< $(CXXFLAGS)

.PHONY: clean
clean:
    $(RM) $(OBJS) $(TARGET)
```

Output (from src/ex3 directory):

```
$ ls
bar.cpp  foobar.hpp  foo.cpp  main.cpp  makefile  sum.cpp  sum.hpp
$ make
g++ -c -o main.o main.cpp -O3 -Wall -Wextra -Wconversion -std=c++11
g++ -c -o sum.o sum.cpp -O3 -Wall -Wextra -Wconversion -std=c++11
g++ -c -o foo.o foo.cpp -O3 -Wall -Wextra -Wconversion -std=c++11
g++ -c -o bar.o bar.cpp -O3 -Wall -Wextra -Wconversion -std=c++11
g++ -o main main.o sum.o foo.o bar.o
$ ls
bar.cpp  bar.o  foobar.hpp  foo.cpp  foo.o  main  main.cpp  main.o  makefile  sum.cpp  sum.hpp  sum.o
$ make clean
rm -f main.o sum.o foo.o bar.o main
$ ls
bar.cpp  foobar.hpp  foo.cpp  main.cpp  makefile  sum.cpp  sum.hpp
```

Linking to a library & `run` targets

src/ex4/makefile:

```
# conventional variable for c++ compiler
CXX := g++

# conventional variable for C preprocessor
CPPFLAGS := -DDEBUG

# conventional variable for C++ compiler flags
CXXFLAGS := -O3 -std=c++11 -Wall -Wextra -Wconversion

# conventional variable for linker flags
LDFLAGS := -ljpeg

TARGET := hw6
OBJS := hw6.o
INCS := hw6.hpp

$(TARGET): $(OBJS)
    $(CXX) -o $(TARGET) $(OBJS) $(LDFLAGS)

%.o: %.cpp $(INCS)
    $(CXX) -c -o $@ $< $(CPPFLAGS) $(CXXFLAGS)

# use .PHONY for targets that do not produce a file
.PHONY: clean
clean:
    rm -f $(OBJS) $(TARGET) *~

.PHONY: run
run: $(TARGET)
    ./$(TARGET)
```

Output (from `src/ex4` directory):

```
$ ls
hw6.cpp  hw6.hpp  makefile  stanford.jpg
$ make
g++ -c -o hw6.o hw6.cpp -DDEBUG -O3 -std=c++11 -Wall -Wextra -Wconversion
g++ -o hw6 hw6.o -ljpeg
$ ./hw6
$ make clean
rm -f hw6.o hw6 *~
$ make run
g++ -c -o hw6.o hw6.cpp -DDEBUG -O3 -std=c++11 -Wall -Wextra -Wconversion
g++ -o hw6 hw6.o -ljpeg
./hw6
$ ls
hw6  hw6.cpp  hw6.hpp  hw6.o  makefile  stanford.jpg  test.jpg
```

Make

- Automation tool for expressing how your C/C++/Fortran code should be compiled
- Good for small projects
- But be careful with dependencies. It is **very** important to understand this process for larger projects.

- Some people would not recommend hand writing Makefile(s) for larger projects (use CMake or similar)
- With discipline, I believe that Make is a good tool for large projects. This is what I use. Sometimes CMake and other tools make it harder to build projects.