Lecture 14

Fall 2020

CME 211: Lecture 14

Topics:

- Compilation process
- Make for building software
- Debuggers

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 `gcc` 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, and we'll have a lecture on this in CME212.

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 -trip
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++/v1
 /usr/local/include
 /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../lib/clang/7.3.0
 /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 -dy
```

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 depreca
$ 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
# 31 "/usr/include/stdc-predef.h" 2 3 4
// ... a bunch of ommitted lines
namespace std {
  // We'll learn what the following lines mean in 212.
  typedef long unsigned int size_t;
  typedef long int ptrdiff_t;

  typedef decltype(nullptr) nullptr_t;
}
```

```
// approximately 17,500 more lines omitted!

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

If you're curious about what the first few lines beginning with # signs represent, see the documentation: https://gcc.gnu.org/onlinedocs/gcc-4.8.5/cpp/Preprocessor-Output.html. "Source file name and line number information is conveyed by lines of the form

```
# linenum filename flags
```

These are called linemarkers... They mean that the following line originated in file filename at line `linenum`... After the file name comes zero or more flags, which are '1', '2', '3', or '4'. If there are multiple flags, spaces separate them. Here is what the flags mean..."

Compilation

- Compilation is the process of translating source code (i.e. the C++ code you wrote) into assembly.
- The assembly commands are still human readable text (if the human knows assembly)!

Note that we could look at `src/hello.s`, but because we are using a library `iostream` the assembly commands become a bit harder to interpret (you can look at them on your own if you wish). Instead we'll turn to a simple addition function file: `src/add.cpp`.

```
#include <iostream>

int add(int a, int b) {
  return a + b;
}

int main(int argc, char* argv[]) {
  int a, b;
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  int c = add(a, b);
  std::cout << c << std::endl;
  return 0;
}
```

We can run compilation up through assembly by invoking `g++ -S -o src/add.s src/add.cpp`, and we can inspect a few key snippets. Let's first look at the addition procedure, i.e. our `add` function:

```
_Z3addii:
.LFB1493:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
```

Next let's see how our function gets invoked; we'll skip most of the output but print a few key lines:

```
main:
.LFB1494:
    .cfi_startproc
    pushq   %rbp
    ...
    movq    -32(%rbp), %rax  <-- Here we read the first argument from command line.
    addq    $8, %rax         <-- We have an offset from our char* array.
    movq    (%rax), %rax
    movq    %rax, %rdi
    call    atoi@PLT
    movl    %eax, -12(%rbp)
    movq    -32(%rbp), %rax  <-- Here we read the second argument from command line.
    addq    $16, %rax        <-- Note the different offset.
    movq    (%rax), %rax
    movq    %rax, %rdi
    call    atoi@PLT
    movl    %eax, -8(%rbp)   <-- Here we set up our call to add.
    movl    -8(%rbp), %edx
    movl    -12(%rbp), %eax
    movl    %edx, %esi
    movl    %eax, %edi
    call    _Z3addii         <-- Invoke add operator.
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    ...
    movl    $0, %eax         <-- Return 0 from main function.
```

Assembly

This step translates the human-readable assembly into 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
...  <-- Output omitted
__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
...  <-- Output omitted
0000000000000000 T _main
                 U _memset
                 U _strlen
```

Here is some documentation. The notable aspects are that there is a symbol associated with each subroutine. E.g. the `T _main` indicates that `main()` is a global text symbol, whereas `U _memset` indicates that this function call currently contains an unresolved reference (to be resolved by the linker in the subsequent step).

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`:

```
#include "foobar.hpp"

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

Linking example

Inspect the files:

Output:

```
$ ls src/foo* src/bar* src/foobar* src/main*
src/bar.cpp  src/foobar.hpp  src/foobar.hpp  src/foo.cpp src/main.cpp
```

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
src/bar.o  src/foo.o  src/main.o
```

What symbols are present in the object files?

Output:

```
$ nm src/foo.o
...  <-- Output omitted
                 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_EES6_
00000000000001a0 S __ZNSt3__124__put_character_sequenceIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0_EE
                 U __ZNSt3__14coutE
0000000000000090 S __ZNSt3__14endlIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0_EES7_
...  <-- Output omitted.
                 U _memset
                 U _strlen
$ nm src/bar.o
...  <-- Output omitted.
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE3putEc
                 U __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEE5flushEv
...  <-- Output omitted.
                 U _memset
                 U _strlen
$ nm src/main.o
                 U __Z3barv
                 U __Z3foov
0000000000000000 T _main
```

Notice that in the last step we still have undefined references to `foo` and `bar`. 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, by giving all of the information that the main program needs in order to execute on the instructions. 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`:

```cpp
// code omitted

#include <jpeglib.h>

#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<cha
...  <-- Output omitted.
  "_jpeg_write_scanlines", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cha
  "_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          <-- We're going to link to this file.
```

Let's find `libjpeg`, it's the library that actually contains the `jpeglib.h` header file.

```
$ locate libjpeg
/Applications/Xcode.app/Contents/Applications/Application Loader.app/Contents/itms/java/lib/libjpeg.dylib
/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              <-- We're going to link to this file.
/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/local/lib -ljpeg
$ ./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`:

```cpp
#include "sum.hpp"

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

```
  return c;
}
```

`src/ex1/sum.hpp`:

```cpp
#pragma once

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

`src/ex1/main.cpp`:

```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`:

```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! From `src/ex1/`:

```
$ 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:

```
$ 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 line and the next two are makefile comments.
#CXXFLAGS := -Wall -Wextra -Wconversion
#CXXFLAGS := -Wall -Wextra -Wconversion -g
CXXFLAGS := -Wall -Wextra -Wconversion -fsanitize=address

# We use $(VAR) to enable parameter expansion.
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
```

The last target is a `PHONY` one because it doesn't produce any files. 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

Make has automatic variables such as `$@` and `$<`, where the former specifies the name of the target of the rule, and the latter specifies the name of the first pre-requisite.

`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/LaTeX code should be built.
- Good for single platform projects.
- But be careful with dependencies. It is **very** important to understand this process for larger projects.
- Hand writing Makefile(s) for cross-platform projects is not recommended. You should consider using configuration tools such as CMake.