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
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    ...
    movl    $0, %eax
```

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_EES6_
00000000000001a0 S __ZNSt3__124__put_character_sequenceIcNS_11char_traitsIcEEEERNS_13basic_ostreamIT_T0_EE
                 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`:

```cpp
#pragma once

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

`src/foo.cpp`:

```cpp
#include <iostream>

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

`src/bar.cpp`:

```cpp
#include <iostream>

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

`src/main.cpp`:

```cpp
#include "foobar.hpp"
```

```cpp
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_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_
                 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_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_
                 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`:

```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
  "_jpeg_CreateDecompress", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char
  "_jpeg_destroy_decompress", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char
  "_jpeg_finish_compress", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cha
  "_jpeg_finish_decompress", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char
  "_jpeg_read_header", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char
  "_jpeg_read_scanlines", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char
  "_jpeg_set_defaults", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cha
  "_jpeg_set_quality", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cha
  "_jpeg_start_compress", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cha
  "_jpeg_start_decompress", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char
  "_jpeg_std_error", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cha
  "_jpeg_stdio_dest", referenced from:
      WriteGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<cha
  "_jpeg_stdio_src", referenced from:
      ReadGrayscaleJPEG(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char
  "_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
```

Let's find `libjpeg`:

Output:

```
$ 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
/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/loca
$ ./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`:

```
#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

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.

Debugging with command line debuggers

Command line debuggers such as GDB and LLDB come without graphical bells and whistles, but can be as effective when you get some experience with them. Once you learn how to use a console-based debugger, it will be fairly straightforward to learn almost any graphics-based one.

Let's use this simple C++ code in file student.cpp to introduce LLDB basics:

```
#include <string>


// Definition of the class Student
class Student
{
public:
  // Constructor
  Student(std::string name, int studentID)
  {
    name_      = name; // set break point here
```

```
    studentID_ = studentID;
  }

  // Destructor
  ~Student()
  {
    studentID_ = 0;
  }

private:
  std::string name_; // Student's name
  int studentID_;    // Student's ID number
};


int main()
{
  // The instance of Student on the stack.
  Student icmeStudent("Terry Gilliam", 123444);

  // The instance of Student on the heap.
  Student* pGeographyStudent = new Student("Terry Jones", 123555);

  delete pGeographyStudent;

  return 0;
}
```

Compiling code for debugging

In order to enable debugging, we need to compile the code with `-g` flag to tell compiler to enable debugging symbols in the executable. It is also highly desirable to turn off optimization with flag `-O0`, so that the debugger can keep track of what line in the source code is being executed. We can build our code by

```
clang++ -g -O0 -o student student.cpp
```

This code produces no output, so to see what is going on inside we need to use debugger.

Starting debugger

To start debugger simply enter `lldb` followed by the executable name on the command line:

```
$ lldb student
(lldb) target create "student"
Current executable set to 'student' (x86_64).
(lldb)
```

To run the executable, enter `run` at the debugger command prompt.

```
(lldb) run
Process 27818 launched: '/Users/peles/lectures/classes/debug' (x86_64)
Process 27818 exited with status = 0 (0x00000000)
(lldb)
```

This tells us that the code ran without any errors. By the way, most difficult bugs to find are those when your code produces believable results and returns no errors. These are situations when you need a debugger the most.

Setting break points

Running the code in a debugger by itself does not give you much information. You typically want to pause at certain places in the code and review what is going on there. You can set up break points (i.e. places for quiet reflection) in your code by using debugger `breakpoint` command. We want to set a breakpoint inside the constructor of the Student class at line 11:

```
(lldb) breakpoint set --file student.cpp --line 11
Breakpoint 1: where = student`Student::Student(std::__1::basic_string<char,
std::__1::char_traits<char>, std::__1::allocator<char> >, int) + 158 at
student.cpp:11, address = 0x0000000100000d0e
(lldb)
```

Note that GDB command for setting the break point is different: `break student.cpp:11`. Now, when we run the code, we stop at the break points we set. The first is the constructor for `icmeStudent` instance of class `Student`.

```
(lldb) run
Process 27952 launched: '/Users/peles/lectures/classes/debug' (x86_64)
Process 27952 stopped
* thread #1: tid = 0x29cb6f, 0x0000000100000cfe student`Student::Student(this=
0x00007fff5fbffa58, name="Terry Gilliam", studentID=123444) + 158 at student.cpp:11,
queue = 'com.apple.main-thread',
stop reason = breakpoint 1.1
    frame #0: 0x0000000100000cfe student`Student::Student(this=0x00007fff5fbffa58,
    name="Terry Gilliam", studentID=123444) + 158 at student.cpp:11
   8      // Constructor
   9      Student(std::string name, int studentID)
   10     {
-> 11       name_      = name; // set break point here
   12       studentID_ = studentID;
   13     }
   14
(lldb)
```

Now that we stopped the code execution at the place we wanted, we would like to inspect variable values there. You can view the variables in the curret scope by typing

```
(lldb) frame variable
(Student *) this = 0x00007fff5fbffa58
(std::__1::string) name = "Terry Gilliam"
(int) studentID = 123444
(lldb)
```

In `gdb` there are separate commands for local arguments in the frame `info args` and local variables in the frame `info locals`. To continue execution of the code simply type `continue` or `c` at the debugger command prompt. That gets us to the next break point inside the constructor for the Georgraphy student instance of the class `Student` (who happens to be Terry Jones).

```
(lldb) continue
Process 27952 resuming
Process 27952 stopped
* thread #1: tid = 0x29cb6f, 0x0000000100000cfe student`Student::Student(
this=0x0000000100104b40, name="Terry Jones", studentID=123555) + 158
at student.cpp:11, queue = 'com.apple.main-thread', stop reason =
breakpoint 1.1
    frame #0: 0x0000000100000cfe student`Student::Student(
    this=0x0000000100104b40, name="Terry Jones", studentID=123555)
    + 158 at student.cpp:11
   8      // Constructor
   9      Student(std::string name, int studentID)
```

```
   10     {
-> 11       name_      = name; // set break point here
   12       studentID_ = studentID;
   13     }
   14
(lldb)
```

We can take a look at the varibales again:

```
(lldb) frame variable name
(std::__1::string) name = "Terry Jones"
(lldb) frame variable name_
(std::__1::string) name_ = ""
(lldb)
```

At this point in the code execution, the constructor argument `name` is set to `Terry Jones`, while `Student` member variable `name_` has been initialized to an empty string, but it has not yet been assigned the value passed to the constructor.

Navigating through the code

We saw that command `continue` resumes the code execution and gets us to the next break point. Command `next` will execute the current and stop at the next line of the code. In this situation, the command `step` will do the same.

```
(lldb) next
   ...
   (some cryptic stuff)
   ...
   9      Student(std::string name, int studentID)
   10     {
   11       name_      = name; // set break point here
-> 12       studentID_ = studentID;
   13     }
   14
   15     // Destructor
(lldb) frame variable name_
(std::__1::string) name_ = "Terry Jones"
(lldb) step
   ...
   (some cryptic stuff)
   ...
   9      Student(std::string name, int studentID)
   10     {
   11       name_      = name; // set break point here
   12       studentID_ = studentID;
-> 13     }
   14
   15     // Destructor
(lldb)
```

The difference between commands `next` and `step` is if the next line of the code is a function the command `next` will call the function and stop at the next line of the code.

```
(lldb) next
   (...)
   32     // The instance of Student on the heap.
   33     Student* pGeographyStudent = new Student("Terry Jones", 123555);
   34
-> 35     delete pGeographyStudent;
```

```
   36
   37     return 0;
   38   }
(lldb) next
   (...)
   34
   35     delete pGeographyStudent;
   36
-> 37     return 0;
   38   }
(lldb)
```

The command `step`, on the other hand, will step into the function. You can use command `finish` to get out of the function and back to the parent scope.

```
(lldb) next
   (...)
   32     // The instance of Student on the heap.
   33     Student* pGeographyStudent = new Student("Terry Jones", 123555);
   34
-> 35     delete pGeographyStudent;
   36
   37     return 0;
   38   }
(lldb) step
   (...)
   14
   15     // Destructor
   16     ~Student()
-> 17     {
   18       studentID_ = 0;
   19     }
   20
(lldb) finish
   (...)
   32     // The instance of Student on the heap.
   33     Student* pGeographyStudent = new Student("Terry Jones", 123555);
   34
-> 35     delete pGeographyStudent;
   36
   37     return 0;
   38   }
(lldb) step
   (...)
   34
   35     delete pGeographyStudent;
   36
-> 37     return 0;
   38   }
(lldb)
```

Let us quickly summarize execution commands:

- `run` – launches the code execution.
- `continue` – continues code execution from the current position in the code.
- `next` – executes the current line of the code and moves to the next line.
- `step` – executes the current line of the code and steps into the function if the current line is a function call.
- `finish` – leaves current scope and moves to the next line in the parent scope.

Backtrace

This is another quite useful debugging command. Sometimes, you will mess up your pointers and your code will crash with segmentation fault. One such example is given in `student2.cpp` file. This code crashes with a segmentation fault:

```
$ ./student2
Segmentation fault: 11
```

We recompile the code with proper flags and launch it in the debugger.

```
(lldb) run
Process 28588 launched: '/Users/peles/cme212/debugging/student2' (x86_64)
Process 28588 stopped
   (... lots of stuff ...)
   1661
   1662     _LIBCPP_INLINE_VISIBILITY
   1663     bool __is_long() const _NOEXCEPT
-> 1664         {return bool(__r_.first().__s.__size_ & __short_mask);}
   1665
   1666 #if _LIBCPP_DEBUG_LEVEL >= 2
   1667
(lldb)
```

This is not very helpful, so we use `backtrace`.

```
(lldb) thread backtrace
   ( ... even more stuff ... )
    frame #2: 0x0000000100001c07 student`std::__1::basic_string<char,
    std::__1::char_traits<char>, std::__1::allocator<char> > std::__1::operator
    +<char, std::__1::char_traits<char>, std::__1::allocator<char> >
    (__lhs="", __rhs=" ") + 359 at string:3978
    frame #3: 0x0000000100001462 student2`Name::getName(this=0x0000000000000000)
    const + 50 at student2.cpp:25
    frame #4: 0x000000010000117f student2`Student::getName(this=0x0000000100104b40)
    const + 31 at student2.cpp:62
    frame #5: 0x0000000100000ce8 student2`main + 456 at student2.cpp:77
    frame #6: 0x00007fff970965c9 libdyld.dylib`start + 1
(lldb)
```

We get even more mess than before, but here we can recognize some things. Reading the backtrace output from the bottom up, we first find that the problem started in function `main` on line 77 in file `student2.cpp`. We then find that the issue was in the call to `Student::getName` on line 62 and `Name::getName` on line 25. Since we now narrowed down the problem to accessing the student's name, we have much better chances of finding the actual bug on line 45, where we accidentally set pointer to `Name` class to `nullptr` value.

Reading

- Glossary of LLDB and GDB commands
- LLDB Tutorial
- Debugging with GDB