Filmuniversity Babelsberg KONRAD WOLF
Creative Technologies MA | CC2 Lecture | SoSe23, SoSe24
Authors: Michael Witt, Angela Brennecke

Compilation Deep Dive

The compilation toolchain include steps for preprocessing, compilation, assembly and linking. If we build our project using an IDE all these steps are executed automatically for us. However to get a better understanding about the processes involved we want to look at each step in this example for the following code:

```
//
//  main.cpp
//  Created by Michael Witt on 23.04.23.
//

#include <iostream>

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

int main(int argc, char** argv) {
    int a = 100;
    int b(200);
    int c{300};

    std::cout << "The result is " << addTwoNumbers(a, b) << std::endl;
}
```

Preprocessing

The proprocessor investigates all preprocessing directives (you can identify them by the starting `#` character) and processes them. You already saw `#include` and other directives like `#define` or `#pragma once`. You can find a reference for preprocessor commands here: https://en.cppreference.com/w/cpp/preprocessor.

We can invoke the preprocessor on the command line:

System	Hints	Command
Linux	Use a Terminal	`g++ -std=c++17 -E CPPFILE > output.i`

System	Hints	Command
macOS	Use a Terminal and install the Command Line Tools for XCode	`clang++ -std=c++17 -E CPPFILE > output.i`
Windows	Use Visual Studio Community and install the Desktop C++ development package. Start a `Developer Command Prompt` from your app menu	`cl /EHsc /E CPPFILE > output.i`

Since the preprocessor outputs all results to the console we need to redirect the output to a file called `output.i` in the example commands. The file ending `.i` is the common file extension for `C/C++` code files that have been preprocessed.

If you look at these files you see that at the bottom is our main code where the thousends of lines before that where merged by the preprocessor into our code by following all `#include` directives.

Compiling

The compiler takes the preprocessed input and generates assembly code. Assembler languages are low-level languages that no longer support sophisticated high-level features like data types, loops, objects etc.

We can compile our source code on the command line to assembly with the following commands:

System	Command
Linux	`g++ -std=c++17 -S CPPFILE`
macOS	`clang++ -std=c++17 -S CPPFILE`
Windows	`cl /EHsc /FA CPPFILE`

This will generate a file with the same name as our code file but with the `.s` file extension on Linux and macOS and `.asm` on Windows. If you look at the assembly source file you can find our main function (search for `main` in the file) and can see our variable initializations:

On macOS the assembly looks similar to this:

```
_main:                                  ; @main
    .cfi_startproc
; %bb.0:
    sub sp, sp, #64
    .cfi_def_cfa_offset 64
    stp x29, x30, [sp, #48]             ; 16-byte Folded Spill
    add x29, sp, #48
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    stur    w0, [x29, #-4]
    stur    x1, [x29, #-16]
```

```
    mov w8, #100                        ; Move value 100 to CPU register
w8
    stur    w8, [x29, #-20]             ; Store content of w8 to the
address in x29 -20 bytes
    mov w8, #200                        ; Move value 200 to CPU register
w8
    str w8, [sp, #24]                   ; Store content of w8 to the
address of the stack pointer + 24 bytes
    mov w8, #300                        ; Move value 300 to CPU register
w8
    str w8, [sp, #20]                   ; Store content of w8 to the
address of the stack pointer + 20 bytes
```

You can see that the assembler can determine the position in memory where to store each variable beacuse we provided static datat types (`ìnt` in our example with 4 bytes size)

The assembly code can look very different depending on the assembler used to convert it to machine code in the next step. However you should be able to find the `mov` operations to move our static variable initializations to the desired places in memory.

Assembly & Machine Code

To convert the assembler code to machine code you use the following commands:

System	Command
Linux	`g++ -std=c++17 -c CPPFILE`
macOS	`clang++ -std=c++17 -c CPPFILE`
Windows	`cl /EHsc /FA CPPFILE` (same command as above)

This will generate `.o` files on Linux and macOS and an `.obj` file on windows. This file contains the processor specific instructions in binary form generated by the assembler. We can still use a hexdump cammand to look at the single bytes. If the processor architecture is known, a *disassembler* could rebuild the assembly code from the machine code.

While the translation of assembly code -> machine code can be reversed by a *disassembler*, it is generally not possible to regenerate high-level language code from assembly code since all high level features were lost during the compilation step.

```
...
00000230: e10b 00b9 e80f 40b9 e90b 40b9 0001 090b  ......@...@.....
00000240: ff43 0091 c003 5fd6 ff03 01d1 fd7b 03a9  .C...._......{..
00000250: fdc3 0091 a0c3 1fb8 a103 1ff8 880c 8052  ...............R
00000260: a8c3 1eb8 0819 8052 e81b 00b9 8825 8052  .......R.....%.R
00000270: e817 00b9 0000 0090 0000 40f9 0100 0090  ..........@.....
00000280: 2100 0091 0000 0094 e007 00f9 a0c3 5eb8  !.............^.
00000290: e11b 40b9 0000 0094 e103 00aa e007 40f9  ..@...........@.
000002a0: 0000 0094 0100 0090 2100 40f9 0000 0094  ........!.@.....
```

```
000002b0: 0000 8052 fd7b 43a9 ff03 0191 c003 5fd6  ...R.{C........_.
...
```

Although the machine code instructions are no longer readable for us there might be some human readable information still present in the object code file. E.g. dynamic library function call names as well as static strings from the source code remain in the object file:

```
...
00001dd0: 005f 5f5a 5374 3974 6572 6d69 6e61 7465  .__ZSt9terminate
00001de0: 7600 5f5f 5a4e 5374 335f 5f31 3869 6f73  v.__ZNSt3__18ios
00001df0: 5f62 6173 6533 335f 5f73 6574 5f62 6164  _base33__set_bad
00001e00: 6269 745f 616e 645f 636f 6e73 6964 6572  bit_and_consider
00001e10: 5f72 6574 6872 6f77 4576 005f 5f5a 4e53  _rethrowEv.__ZNS
00001e20: 7433 5f5f 3131 3362 6173 6963 5f6f 7374  t3__113basic_ost
00001e30: 7265 616d 4963 4e53 5f31 3163 6861 725f  reamIcNS_11char_
00001e40: 7472 6169 7473 4963 4545 4535 666c 7573  traitsIcEEE5flus
00001e50: 6845 7600 5f5f 5a4e 5374 335f 5f31 3131  hEv.__ZNSt3__111
00001e60: 6368 6172 5f74 7261 6974 7349 6345 3365  char_traitsIcE3e
00001e70: 6f66 4576 005f 5f5a 4e4b 5374 335f 5f31  ofEv.__ZNKSt3__1
00001e80: 3869 6f73 5f62 6173 6536 6765 746c 6f63  8ios_base6getloc
00001e90: 4576 005f 5f5a 4e53 7433 5f5f 3131 375f  Ev.__ZNSt3__117_
00001ea0: 5f63 6f6d 7072 6573 7365 645f 7061 6972  _compressed_pair
00001eb0: 494e 535f 3132 6261 7369 635f 7374 7269  INS_12basic_stri
00001ec0: 6e67 4963 4e53 5f31 3163 6861 725f 7472  ngIcNS_11char_tr
...
```

Assembly code thus provides a more accessible way for humans to write and understand the instructions that a CPU will execute, whereas machine code is the actual set of binary instructions that the CPU understands and processes directly. Both are specific to a CPU architecture, but assembly language bridges the gap between human-readable code and machine-readable instructions.

Tools to Understand Object Code

While you cannot directly read a .o file like a .cpp file, there are tools such as **nm**, **objdump**, and **readelf** that can help you extract and analyze information from an object file. These tools are particularly useful for debugging and reverse engineering, but they do not provide a direct translation of the machine code back into the original C++ source code:

1. `nm`: Displays the symbol table of an object file. This can help you see the names of functions and variables that are defined or referenced in the file.

   ```
   $: nm file.o
   ```

2. `objdump`: A powerful tool that can display various information about an object file, including disassembling the machine code back into assembly.

   ```
   ```

```
$: objdump -d file.o  # disassembles the file
```

3. `readelf`: Another tool that can display details about the ELF (Executable and Linkable Format) headers and sections of the object file.

```
$: readelf -a file.o
```

Linker & Executable

The final step is done by the linker which binds together all object files of your project and generates an executable application for your target operating system. The file format for each operating system is different and structures required information in different ways. Windows uses `PE` format, Linux binaries are in `ELF` format and macOS utilitzes `MACH-O` (https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats).

An executable is the final output of the linking process and the actual program that users run. It is a complete, standalone file that can be run directly by the operating system. Executable files contain the machine code for the entire program, including code from all linked object files and libraries. They also include information for the operating system to load and execute the program.

An executable can be created by linking object files and libraries using a linker such as examplified below:

```
$ g++ file1.o file2.o -o program
```

Bibliography / Web References

- Compiling and Linking in C++
- GCC and Make
- Introduction to the compiler...
- What is Linking?
- Compiling and Linking C++ Applications
- Wikipedia: Linking
- ChatGPT