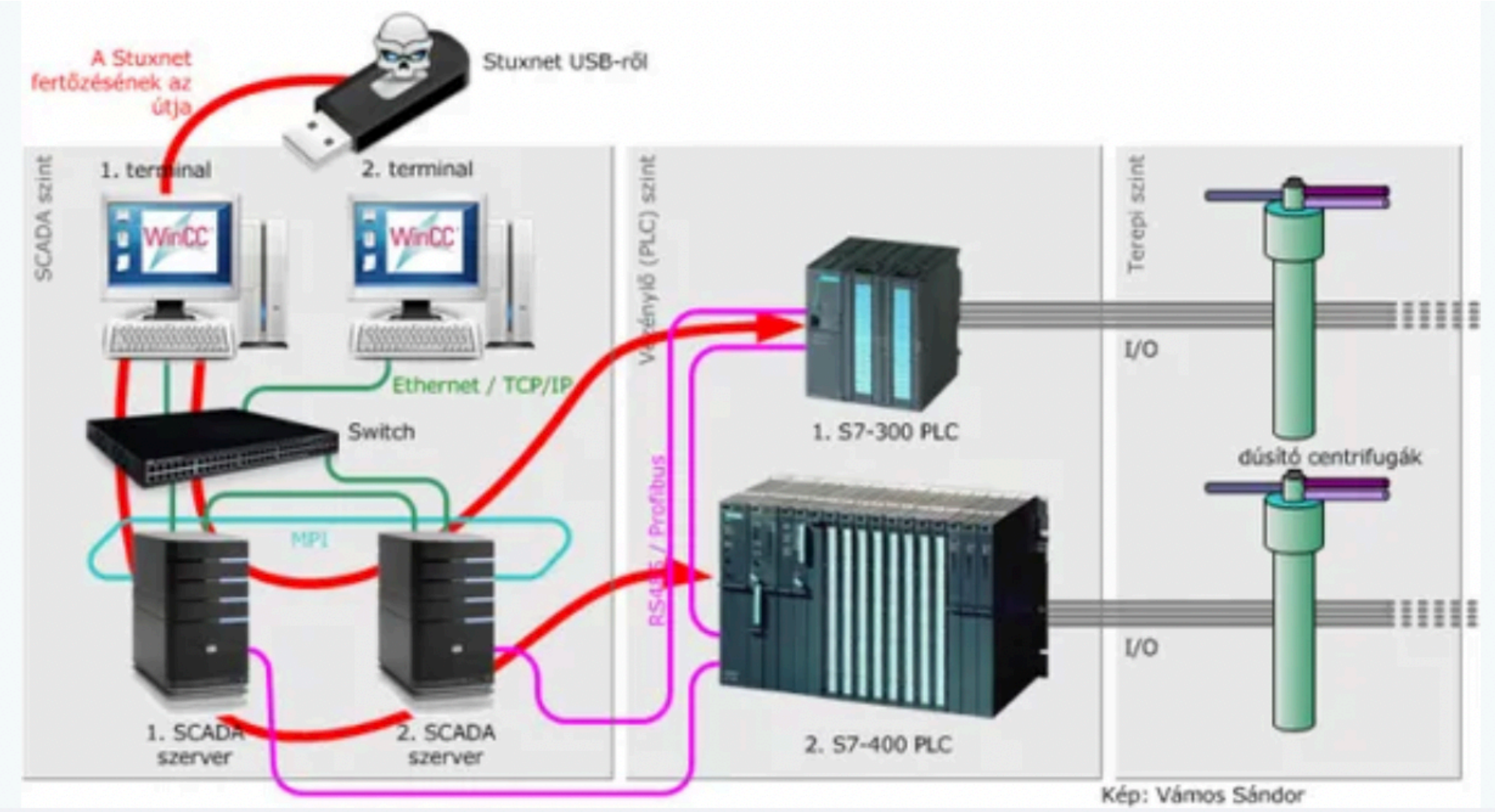
Reverse Engineering

Week 1: Introduction to Disassembly, Layout and a Little Linking

Instructor: Sergey Bratus
Contributions and guest lectures: John Berry, Travis Goodspeed, Ryan Speers, more TBA

Dartmouth College -- Winter 2022

Motivation



- “Stuxnet”, 2005?–2010
- A group infiltrated engineer workstations for Iranian nuclear centrifuges, uploaded modified code to PLC units
- Caused centrifuges to spin out of control and damaged them
  - Awesome example of a physical effect from a digital attack
- Felix ‘FX’ Lindner built RE tools for the PLC *Step7* code, from scratch
  - “27c3: Building Custom Disassemblers” https://www.youtube.com/watch?v=Q9ezff6LIoI

Motivation

Cool RE

- Vendors don’t like to give you control over your own devices
- The same is true even for tractors
- There is a group dedicated to tractor hacking that Reverse Engineers the tractor firmware so that it can be fixed by the farmers
- https://www.wired.com/story/john-deere-farmers-right-to-repair/

What is Reverse Engineering?

And is it legal?

- This isn’t a course in law, nor are any of us law experts. Seek your own legal advice.
- There are many legal uses for Reverse Engineering, but also there are potential violations of law or contracts.
- The Electronic Frontier Foundation (EFF) has a helpful guide for reference at **https://www.eff.org/issues/coders/reverse-engineering-faq**
- “Five areas of United States law are particularly relevant for computer scientists engaging in reverse engineering:
  - Copyright law and fair use, codified at 17 U.S.C. 107;
  - Trade secret law;
  - The anti-circumvention provisions of the Digital Millennium Copyright Act (DMCA), codified at 17 U.S.C. section 1201;
  - Contract law, if use of the software is subject to an End User License Agreement (EULA), Terms of Service notice (TOS), Terms of Use notice (TOU), Non-Disclosure Agreement (NDA), developer agreement or API agreement; and
  - The Electronic Communications Privacy Act, codified at 18 U.S.C. 2510 et. seq.” (-EFF)

ELF Files

How the OS interprets a binary

- Executable and Linkable Format (ELF)
- Composed of 3 main parts (ELF Header, Sections, Segments)
- We will just hit a few of the important bits

ELF Files

Header

- Provides some basic information about the file
  - Where to start executing
  - Where to find the program headers
  - Where to find the section headers.
  - Other information as well such as type, architecture, etc

ELF Files

Header

- To view header details use the `readelf -h <file>`

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14792 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
```

ELF Files

Header (Magic)

- To view header details use the `readelf -h <file>`

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14792 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
```

ELF Files

Header (Entry Point)

- To view header details use the `readelf -h <file>`

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14792 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
```

ELF Files

Header (Program Headers Start)

- To view header details use the `readelf -h <file>`

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14792 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
```

ELF Files

Header (Section Headers Start)

- To view header details use the `readelf -h <file>`

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14792 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
```

ELF Files

Segments (Program Headers)

- Used to describe how to load the executable into memory
- Provides information such as type, permissions, load address, size, etc
- 64-bit ELFs program headers have the following structure:

```
typedef struct {
    uint32_t   p_type;
    uint32_t   p_flags;
    Elf64_Off  p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    uint64_t   p_filesz;
    uint64_t   p_memsz;
    uint64_t   p_align;
} Elf64_Phdr;
```

ELF Files

Segments (Type)

```
typedef struct {
    uint32_t   p_type;
    uint32_t   p_flags;
    Elf64_Off  p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    uint64_t   p_filesz;
    uint64_t   p_memsz;
    uint64_t   p_align;
} Elf64_Phdr;
```

ELF Files

Segments (Flags)

```
typedef struct {
    uint32_t   p_type;
    uint32_t   p_flags;
    Elf64_Off  p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    uint64_t   p_filesz;
    uint64_t   p_memsz;
    uint64_t   p_align;
} Elf64_Phdr;
```

ELF Files

Segments (Offset)

```
typedef struct {
    uint32_t   p_type;
    uint32_t   p_flags;
    Elf64_Off  p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    uint64_t   p_filesz;
    uint64_t   p_memsz;
    uint64_t   p_align;
} Elf64_Phdr;
```

ELF Files

Segments (Virtual Address)

```
typedef struct {
    uint32_t   p_type;
    uint32_t   p_flags;
    Elf64_Off  p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    uint64_t   p_filesz;
    uint64_t   p_memsz;
    uint64_t   p_align;
} Elf64_Phdr;
```

ELF Files

Segments (Program Headers)

- `readelf -l <file>`

```
Elf file type is DYN (Shared object file)
Entry point 0x1060
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000600 0x0000000000000600  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000265 0x0000000000000265  R E    0x1000
```

ELF Files

Sections

- Contains the information needed for linking and relocation
- Common sections: .text; .data; .rodata; .bss

```
typedef struct {
  uint32_t   sh_name;
  uint32_t   sh_type;
  uint64_t   sh_flags;
  Elf64_Addr sh_addr;
  Elf64_Off  sh_offset;
  uint64_t   sh_size;
  uint32_t   sh_link;
  uint32_t   sh_info;
  uint64_t   sh_addralign;
  uint64_t   sh_entsize;
} Elf64_Shdr;
```

ELF Files

Sections (Section Name)

```
typedef struct {
  uint32_t   sh_name;
  uint32_t   sh_type;
  uint64_t   sh_flags;
  Elf64_Addr sh_addr;
  Elf64_Off  sh_offset;
  uint64_t   sh_size;
  uint32_t   sh_link;
  uint32_t   sh_info;
  uint64_t   sh_addralign;
  uint64_t   sh_entsize;
} Elf64_Shdr;
```

ELF Files

Sections (Section Header Address)

```
typedef struct {
  uint32_t   sh_name;
  uint32_t   sh_type;
  uint64_t   sh_flags;
  Elf64_Addr sh_addr;
  Elf64_Off  sh_offset;
  uint64_t   sh_size;
  uint32_t   sh_link;
  uint32_t   sh_info;
  uint64_t   sh_addralign;
  uint64_t   sh_entsize;
} Elf64_Shdr;
```

ELF Files

Sections (Section Header Offset)

```
typedef struct {
  uint32_t   sh_name;
  uint32_t   sh_type;
  uint64_t   sh_flags;
  Elf64_Addr sh_addr;
  Elf64_Off  sh_offset;
  uint64_t   sh_size;
  uint32_t   sh_link;
  uint32_t   sh_info;
  uint64_t   sh_addralign;
  uint64_t   sh_entsize;
} Elf64_Shdr;
```

ELF Files

Sections (.bss)

- Holds zeroed-out uninitialised data
- Used to hold global variables
- Readable and writeable

ELF Files

Sections (.data)

- Holds initialised data
- Used to hold global variables
- Readable and writeable

ELF Files

Sections (.rodata)

- Holds initialised data
- Used to hold global variables
- Read only

ELF Files

Sections (.text)

- Holds executable code
- Read/Execute only

ELF Files

Sections (.got)

- Global Offset Table
- An array of pointers used when the executable needs to call an imported function

ELF Files

Sections (.plt)

- Procedure Linkage Table
- Section of code that uses the GOT to call imported functions

ELF Files

Segments (Program Headers)

- `readelf -S <file>`

```
There are 31 section headers, starting at offset 0x39c8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.propert NOTE             0000000000000338  00000338
       0000000000000020  0000000000000000   A       0     0     8
  [ 3] .note.gnu.build-i NOTE             0000000000000358  00000358
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000037c  0000037c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003a0  000003a0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003c8  000003c8
       00000000000000a8  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000470  00000470
       0000000000000084  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           00000000000004f4  000004f4
       000000000000000e  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000000508  00000508
       0000000000000020  0000000000000000   A       7     1     8
```

ELF Files

References

- https://blog.k3170makan.com/2018/09/introduction-to-elf-format-elf-header.html
- https://wiki.osdev.org/ELF_Tutorial
- man elf

ELF Dynamic Linking

Why?

- If not then every executable has to contain every bit of code that it wants to execute
- It is better to use a common set of shared libraries.
- Don’t confuse with compile time linking

ELF Dynamic Linking

How?

- Well, its complicated
- The OS uses a “linker” that is specified in the ELF header, see **.interp** segment
- This linker looks at the ELF headers and determines what libraries need to be loaded in order for the ELF executable to run
- The address in the GOT are set to the correct location in memory where the libraries were loaded.
- So when you call printf, your code calls the location in the PLT which then will JMP to the necessary location in code.
- Way more complicated than this but it is a good overview.

ELF Dynamic Linking

ldd

```
ldd <file>

linux-vdso.so.1 (0x00007ffd121f6000)

libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff7d6e78000)

/lib64/ld-linux-x86-64.so.2 (0x00007ff7d707d000)
```

Ghidra

What is it?

- Free cross-platform reverse engineering tool written by…the NSA
    - Yes, the National Security Agency. Yes, it is free and open source.
- Method to view the assembly instructions of a compiled binary
- Also provides a decompilation view for a C-like syntax
- Makes available a number of analysis tools
- Provides a scripting interface for plugins and an intermediate language (IL)

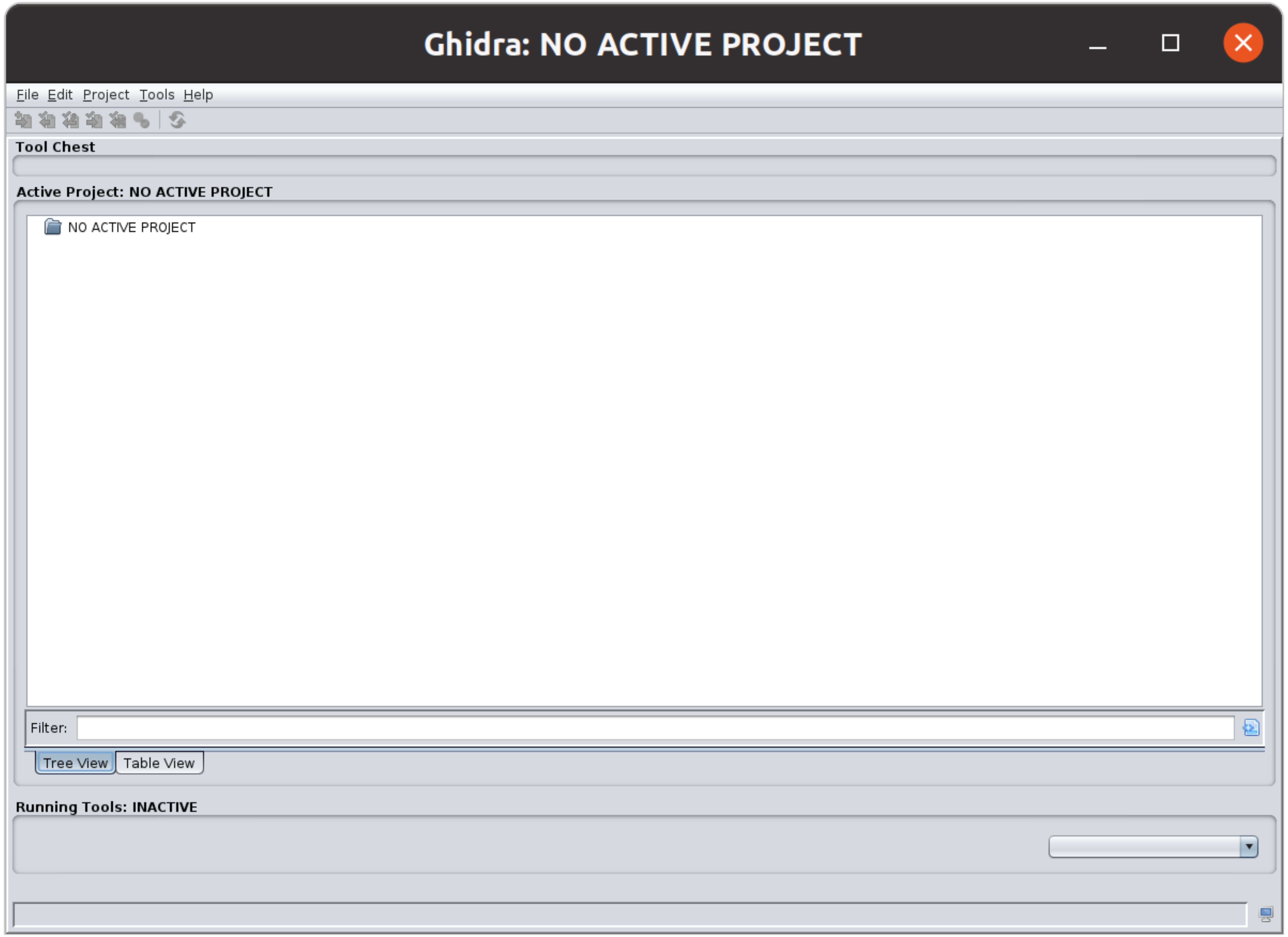
Ghidra

Launching

- You should have it downloaded and unzipped already
- Windows: Double click ghidraRun.bat
- OSX, *nix: Double click ghidra Run
  - If it doesn't launch make sure that it is executable or just run it from a terminal.

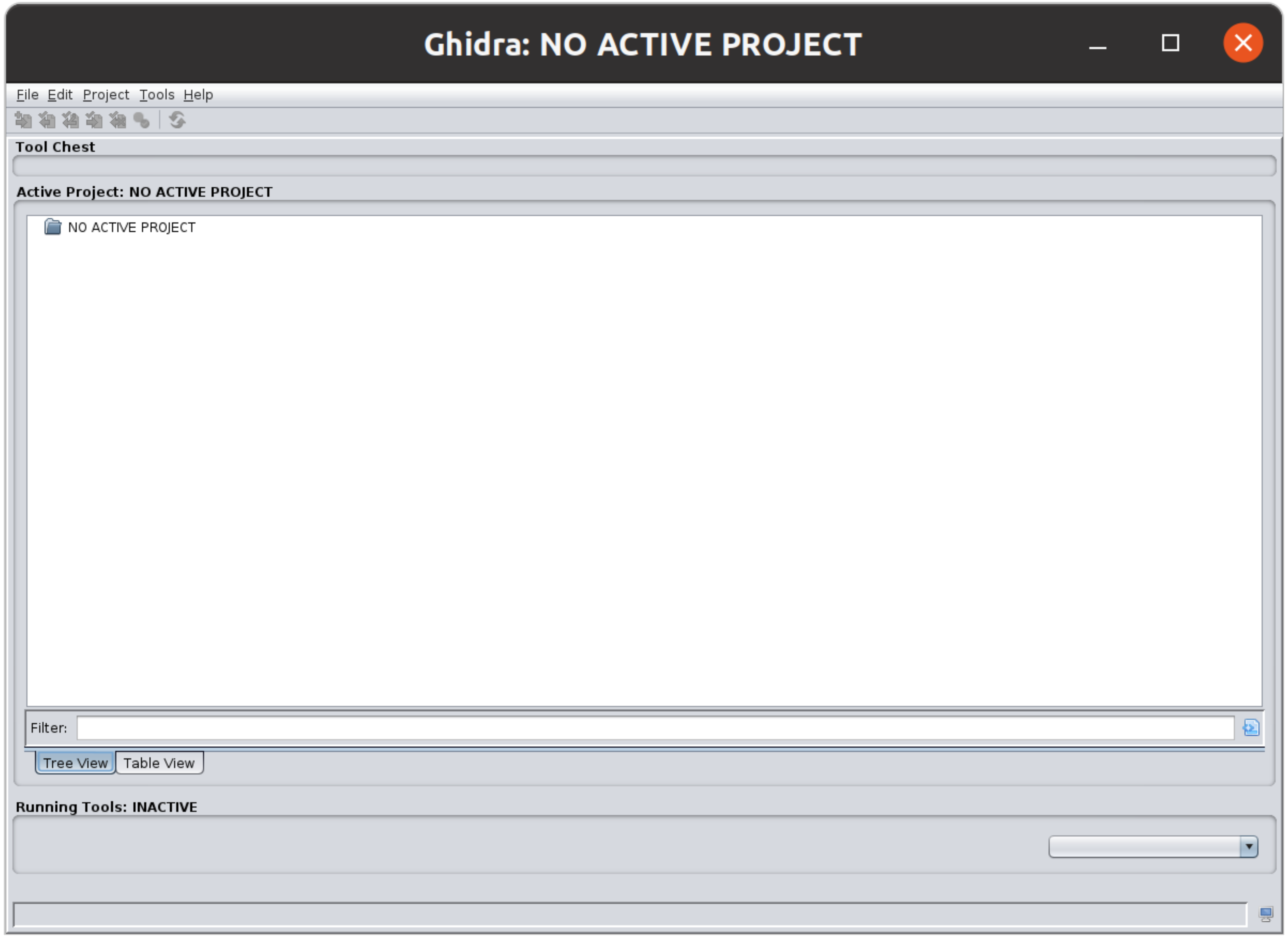
Ghidra
Launching

- When you first open Ghidra you have to open a project.
- You could just create a single project for this whole class



Ghidra
Launching

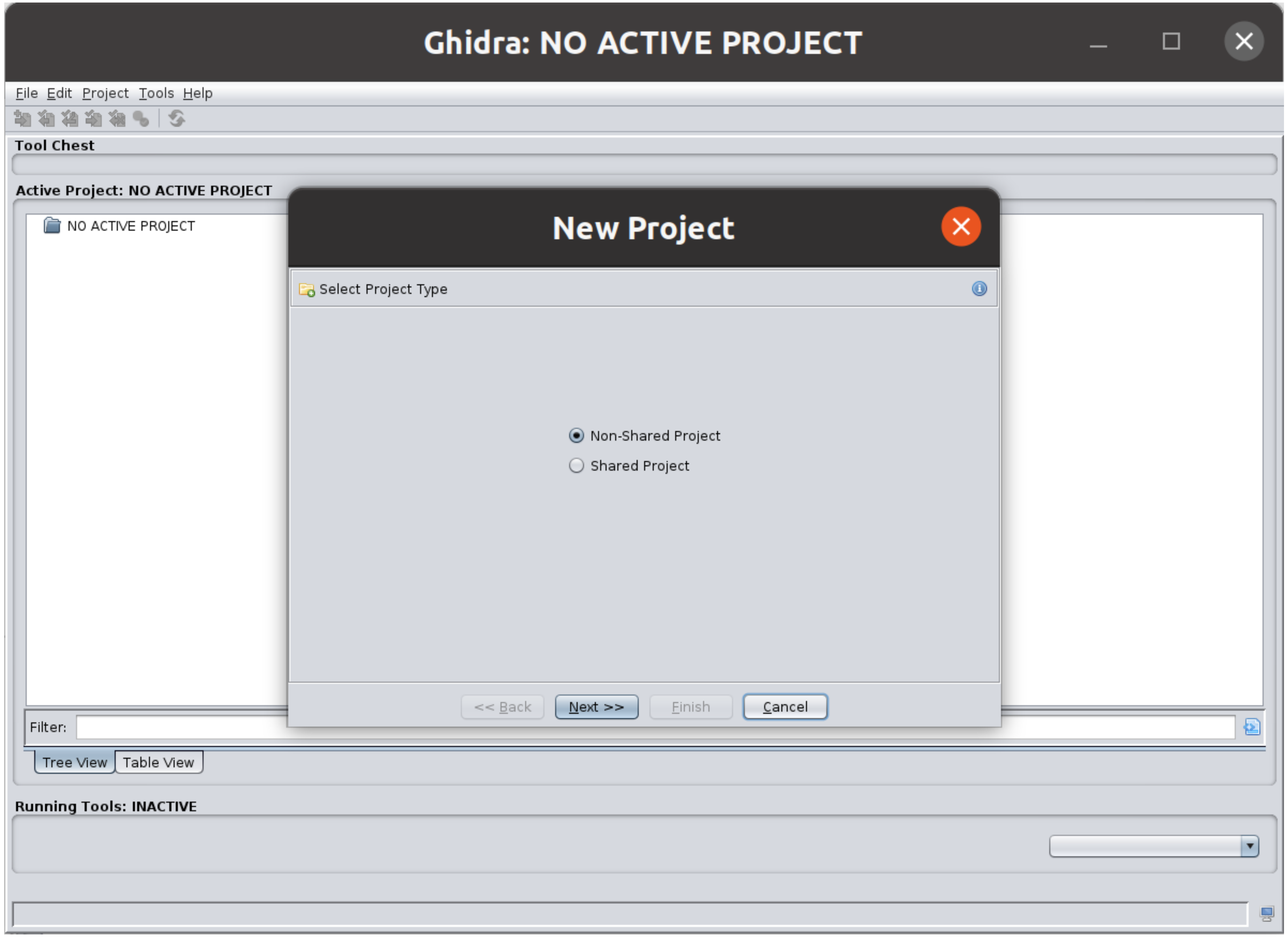
- When you first open Ghidra you have to open a project.
- You could just create a single project for this whole class
- File->New Project



Ghidra

Launching

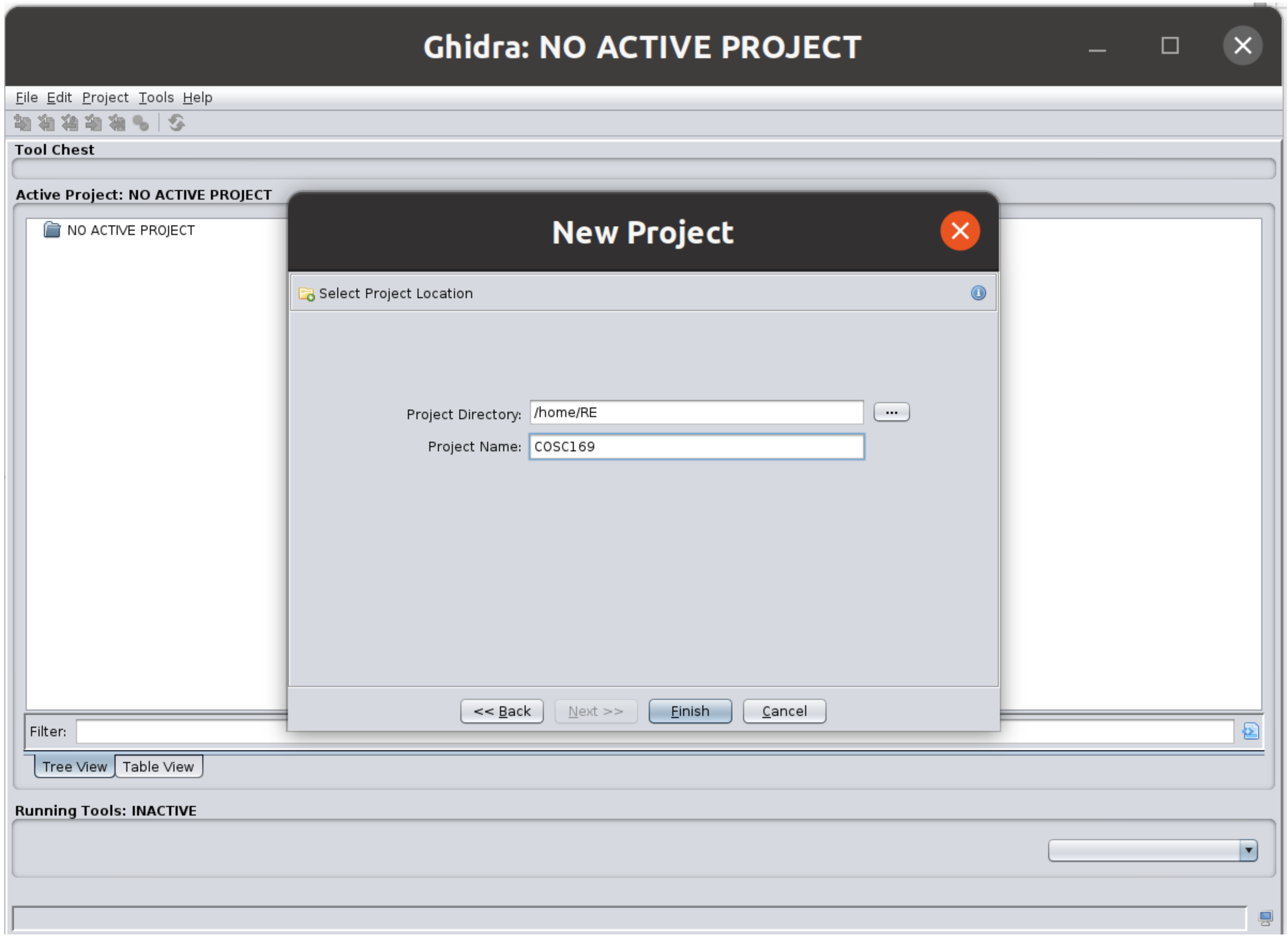
- When you first open Ghidra you have to open a project.
- You could just create a single project for this whole class
- File->New Project
- Non-Shared



Ghidra

Launching

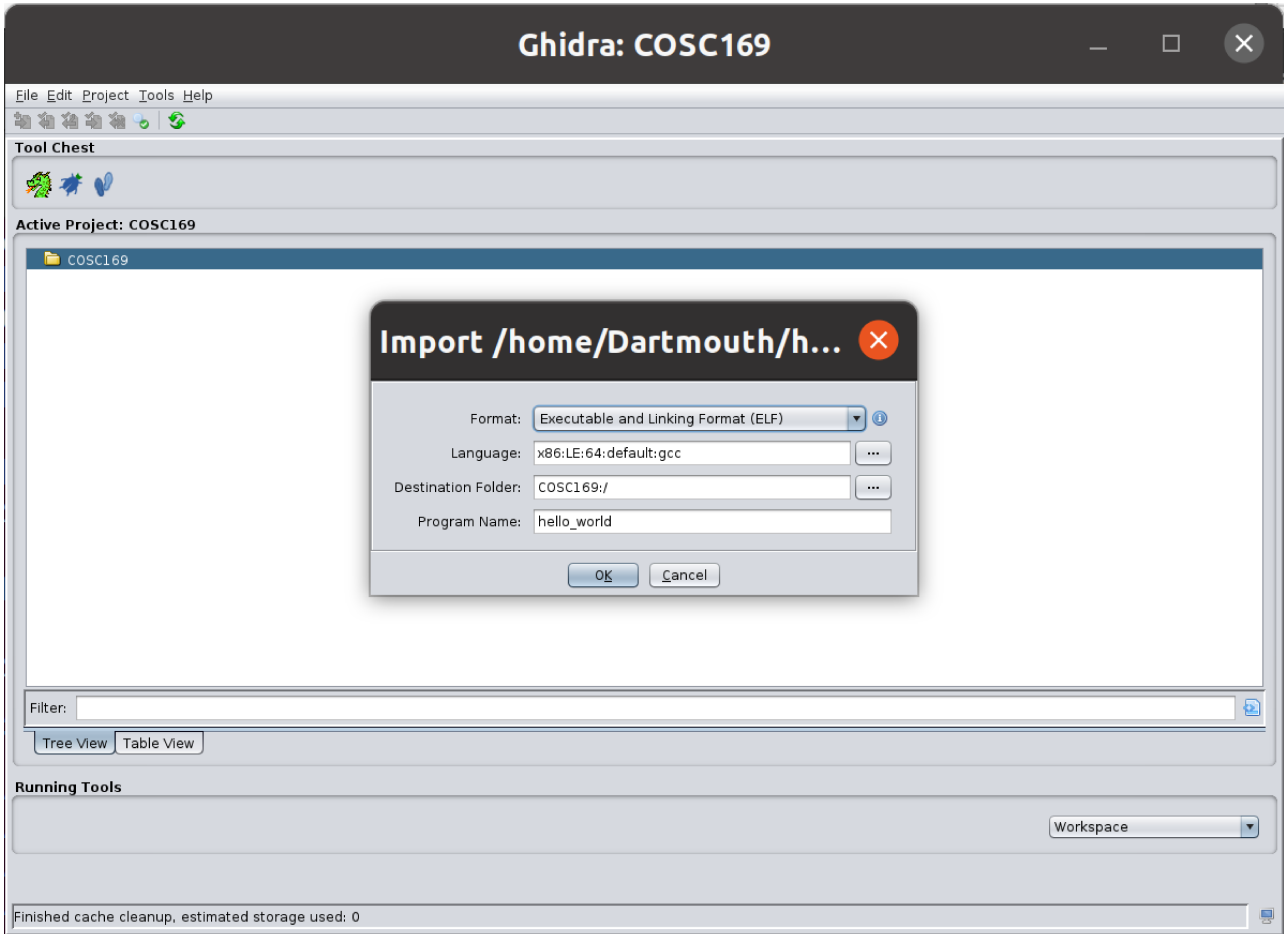
- Select a directory and give it a name
- Click Finish



Ghidra

Launching

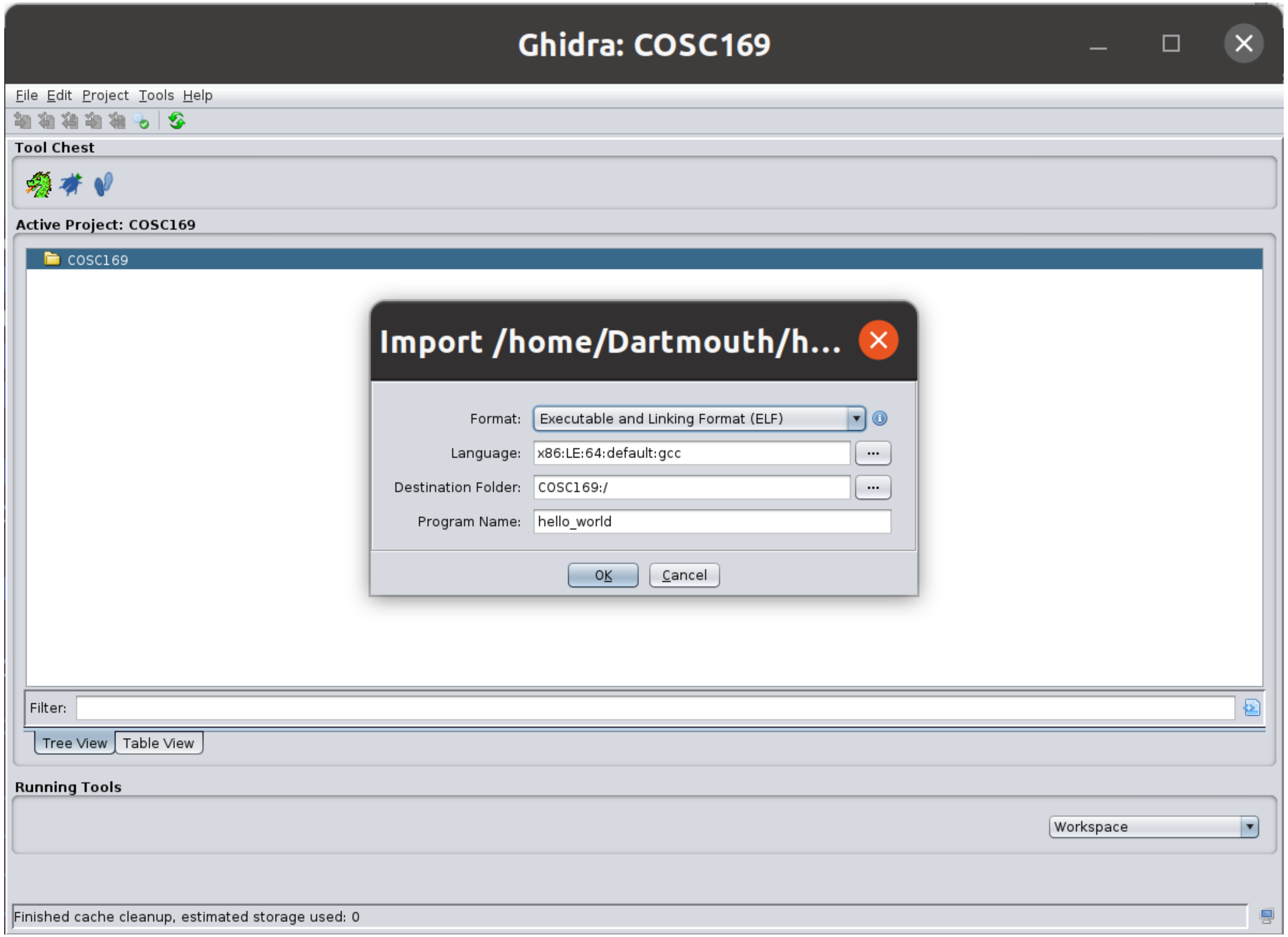
- You now have a project but nothing in it
- Let's load a file: hello_world
- File -> Import File
- Browse to where you have hello_world saved.



Ghidra

Launching

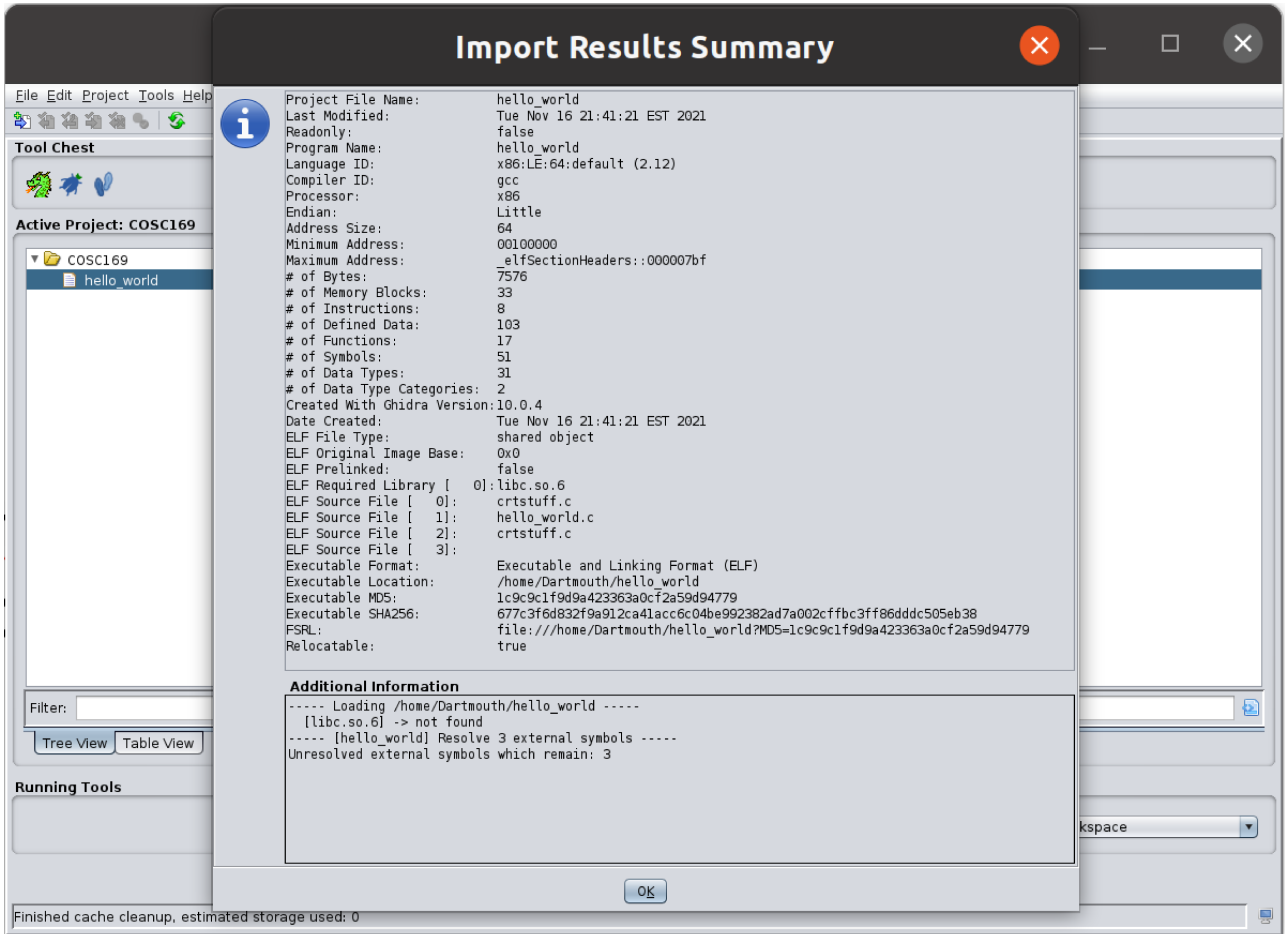
- Format: Specifies the file type. The default will likely be the correct answer.
- Language: The architecture for which the file is built.
- Destination Folder/Program Name speak for themselves



Ghidra

Launching

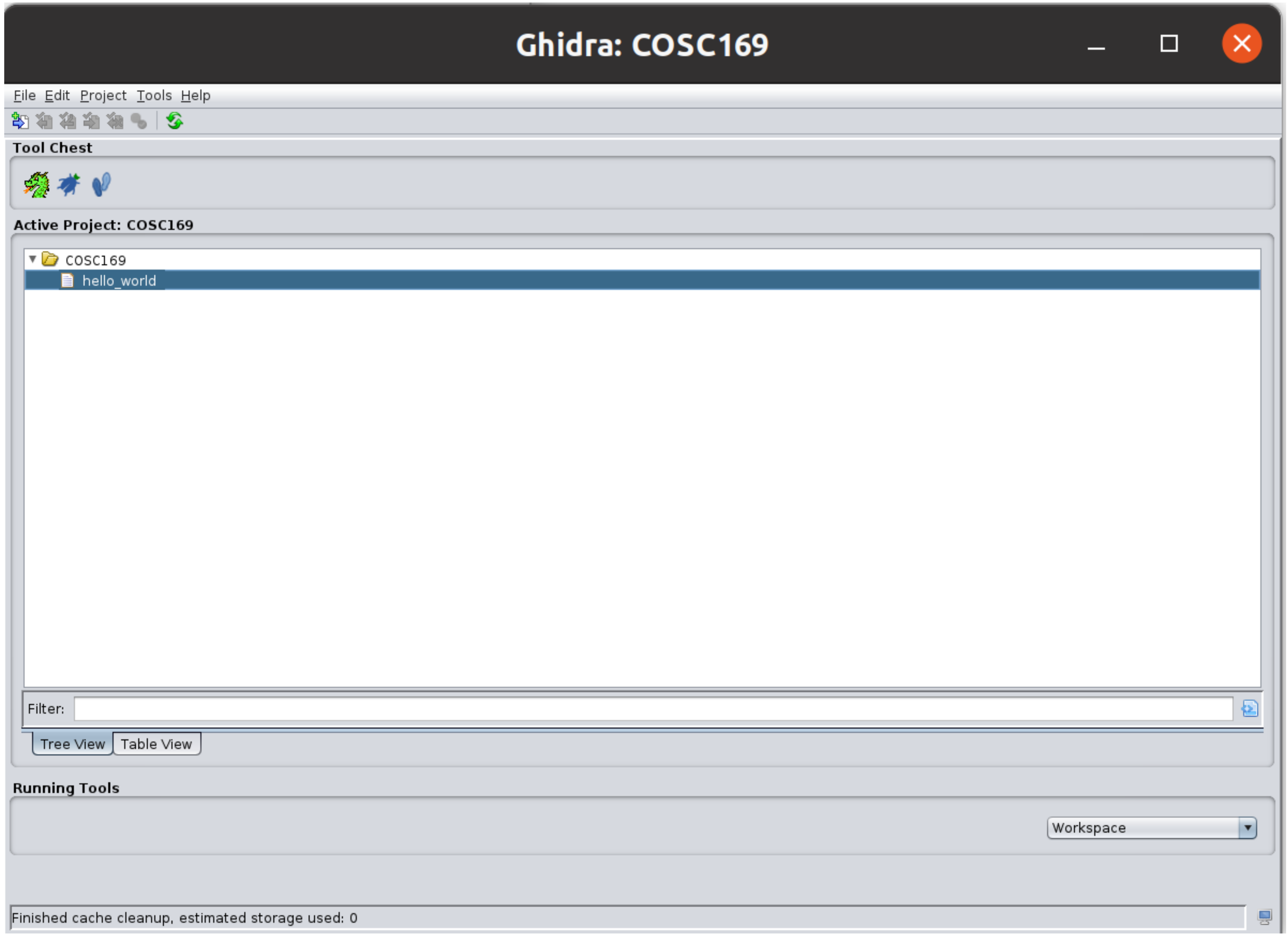
- Import Results: Interesting information but you can just click OK.



Ghidra

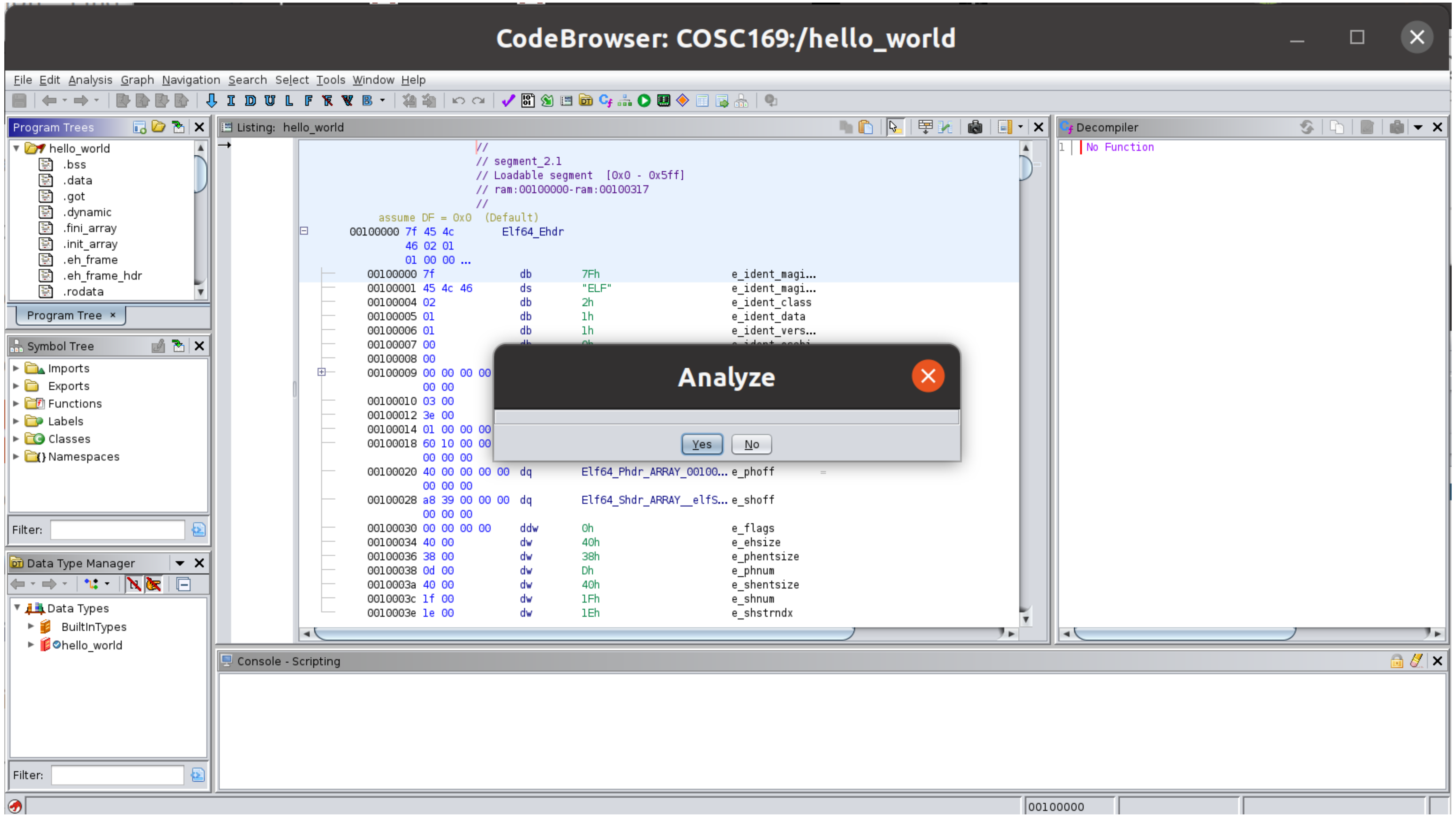
Launching

- File is imported
- Double click "hello_world" to launch the Code Browser



Ghidra

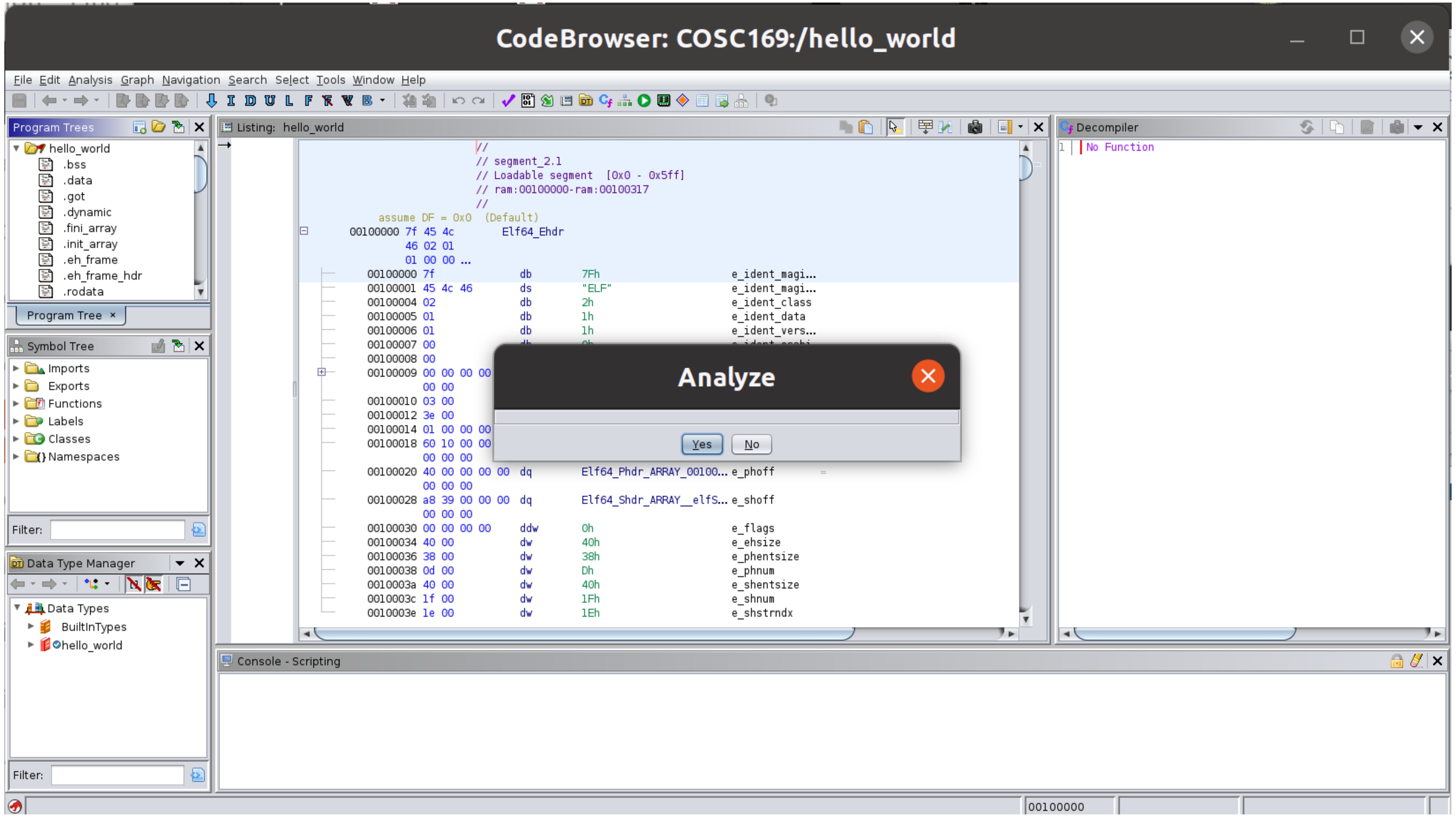
Analyzing



Ghidra

Analyzing

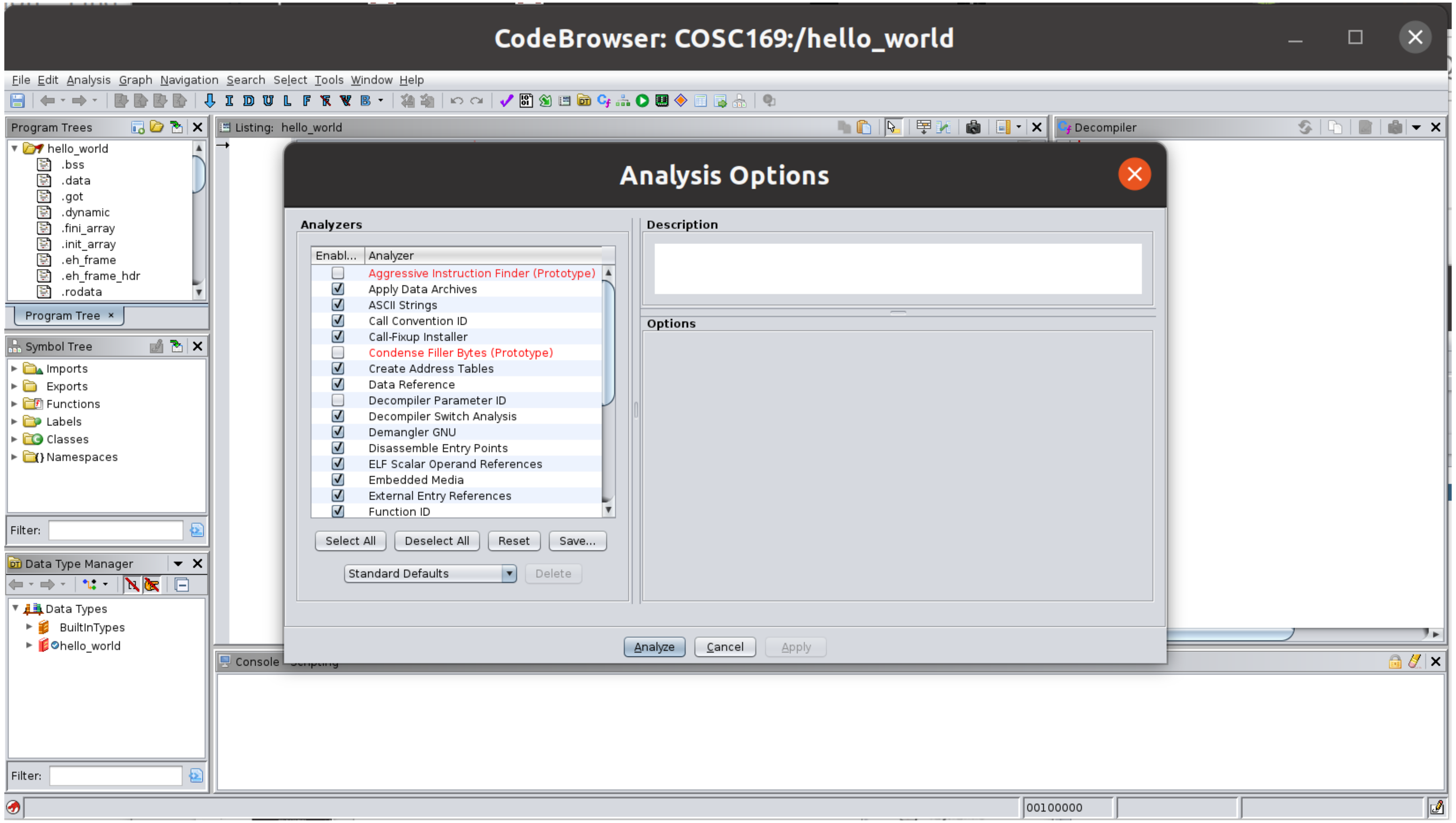
- Click Yes to start the analysis



Ghidra

Analyzing

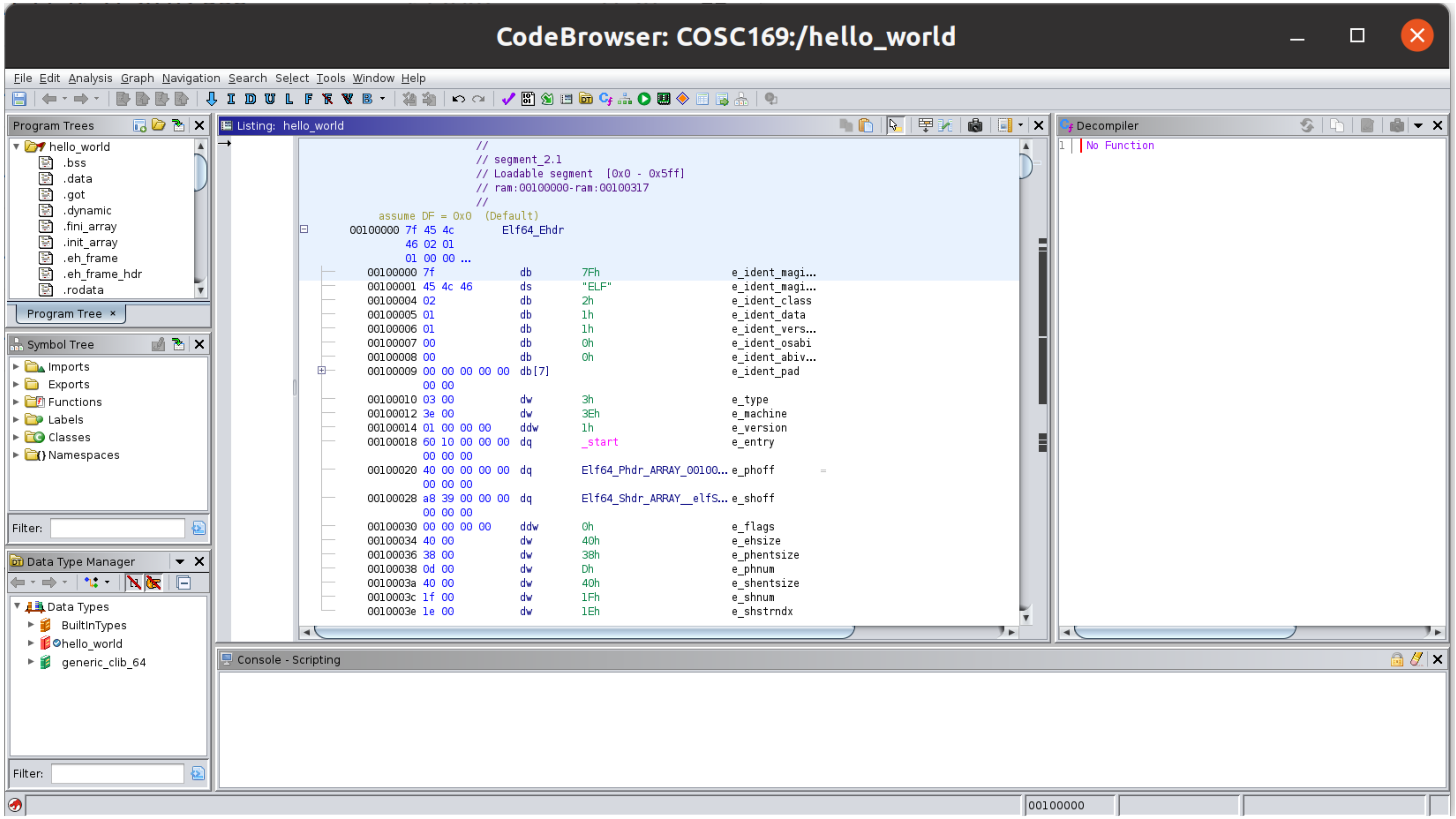
- Lots of analysis options but for now the default are sufficient. Click "Analyse"



Ghidra

Analyzing

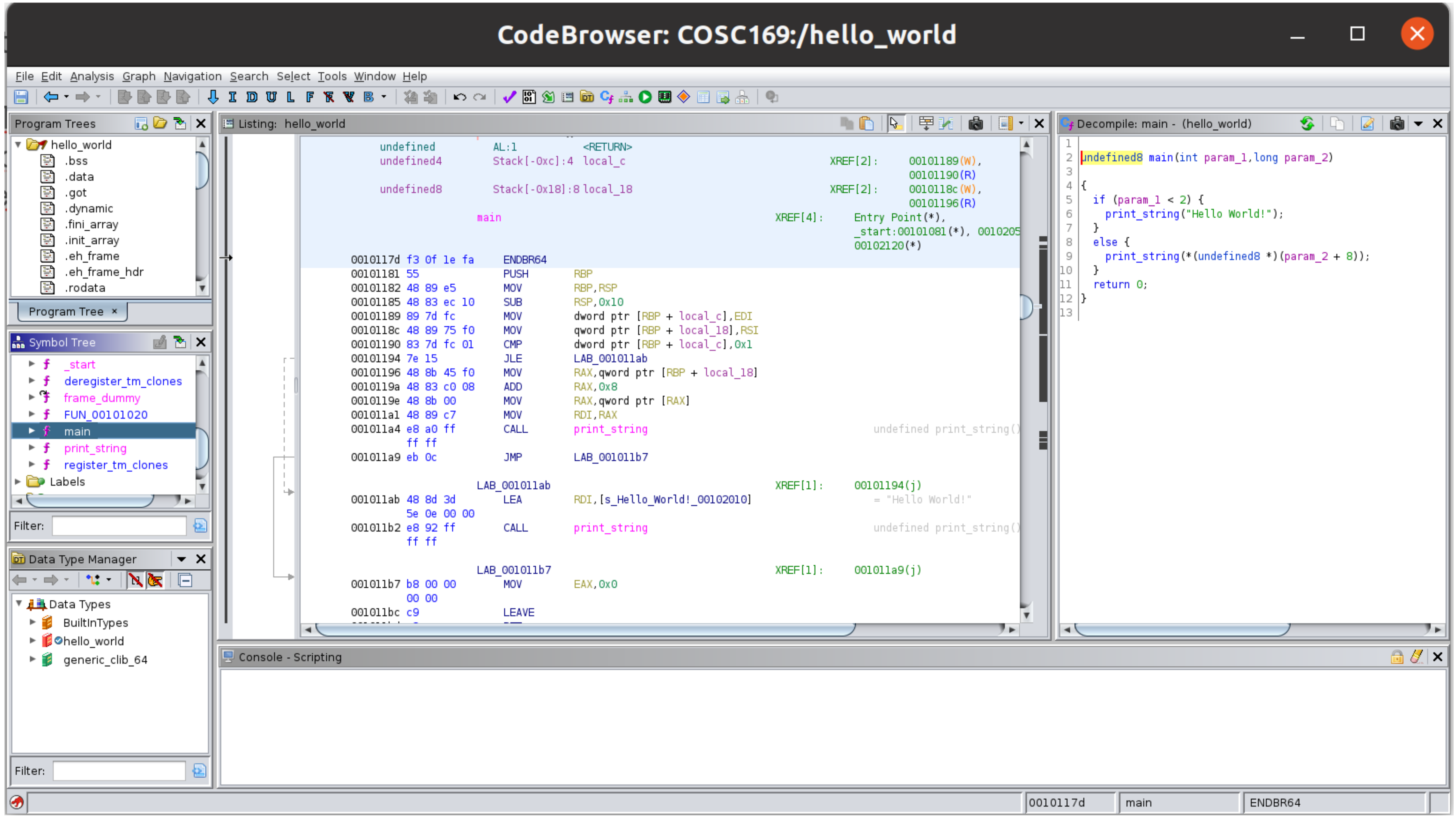
- Should be quick but once done you should be able to see the ELF header we discussed earlier.



Ghidra

Analyzing

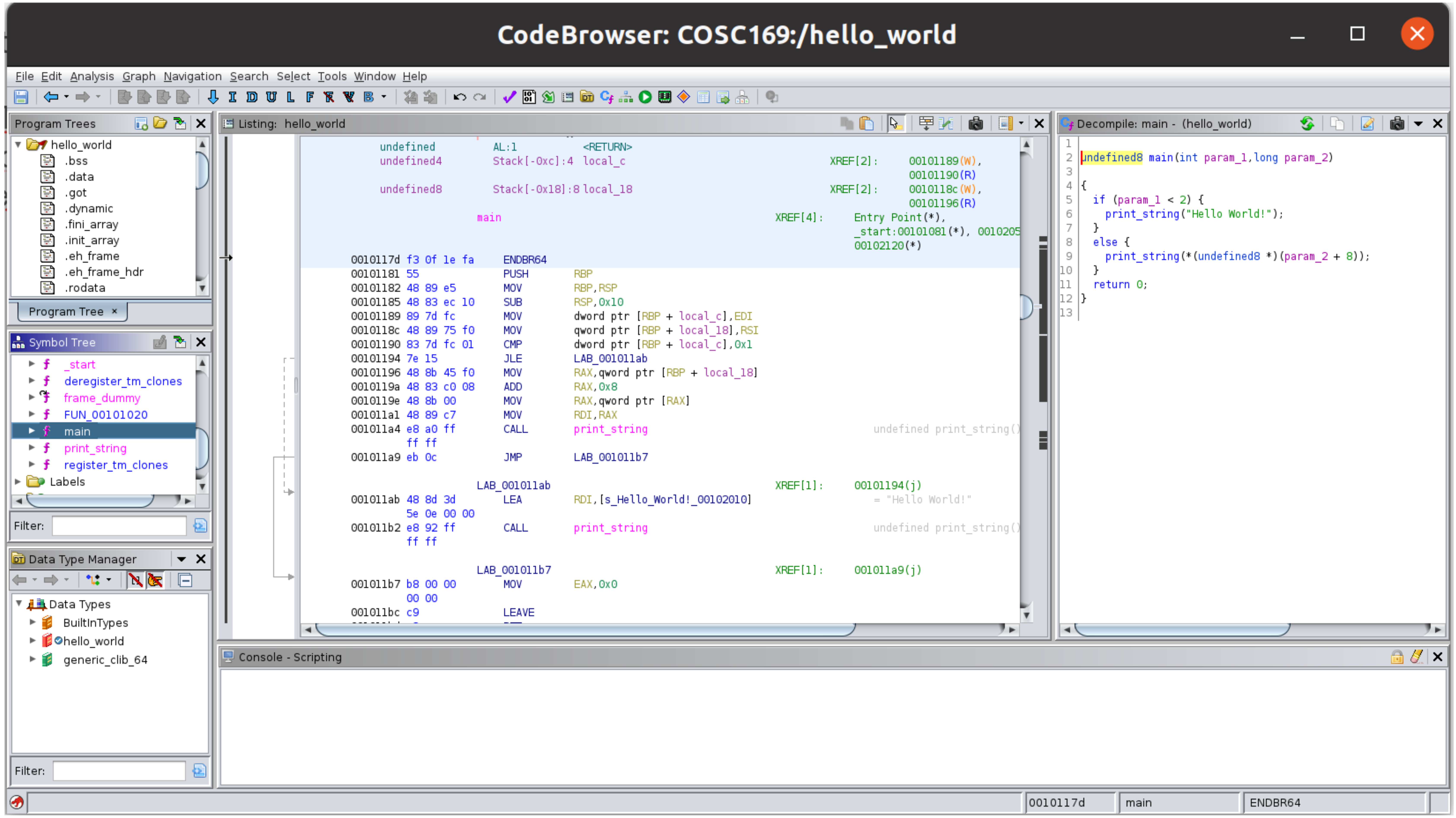
- Let's check out the main() function
- In the Symbol Tree window expand the Functions folder.
- Scroll until you find main and click on it.



Ghidra

Analyzing

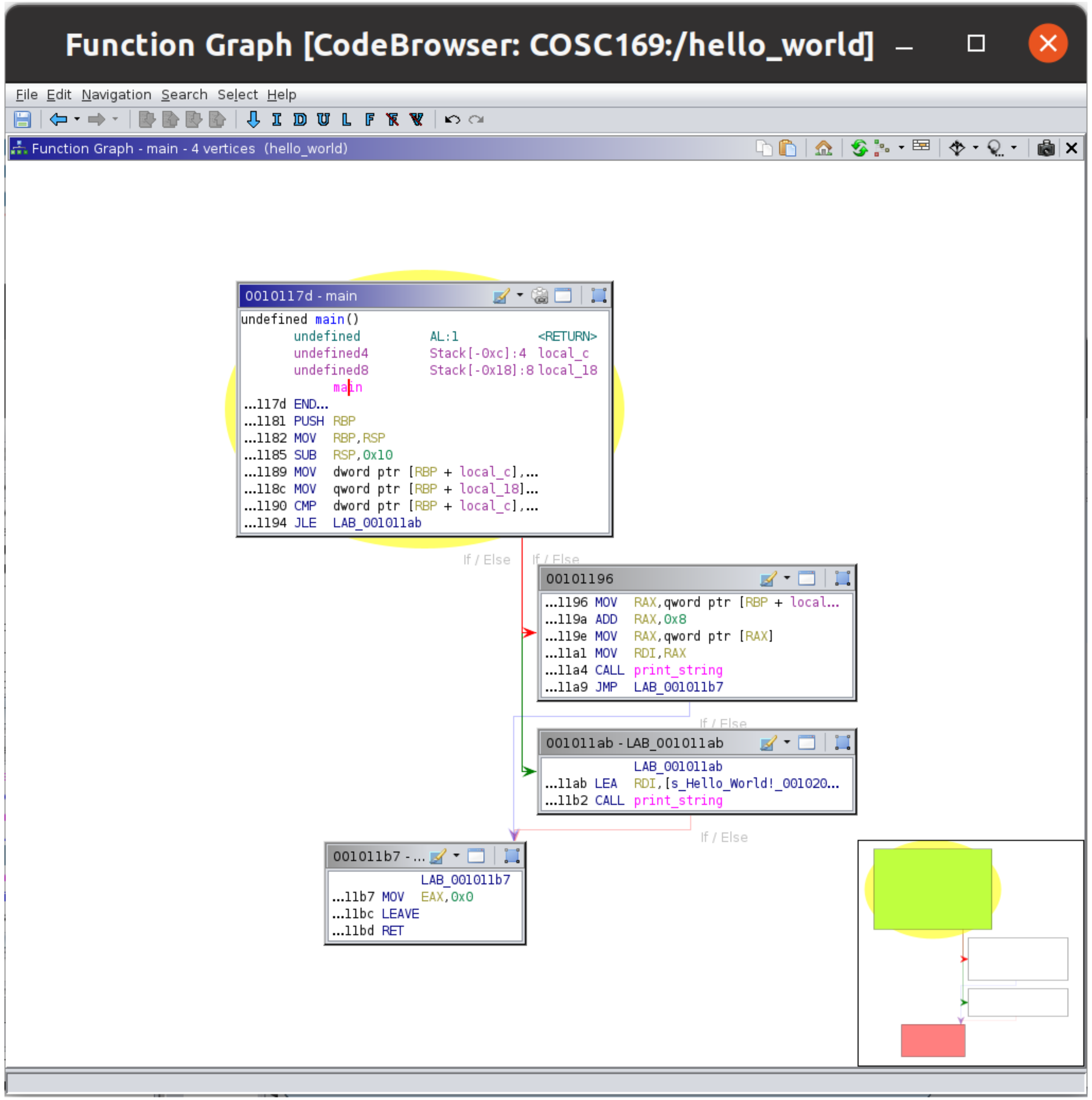
- Check out the decompilation
- How does it compare to what is inside hello_world.c?



Ghidra

Analyzing

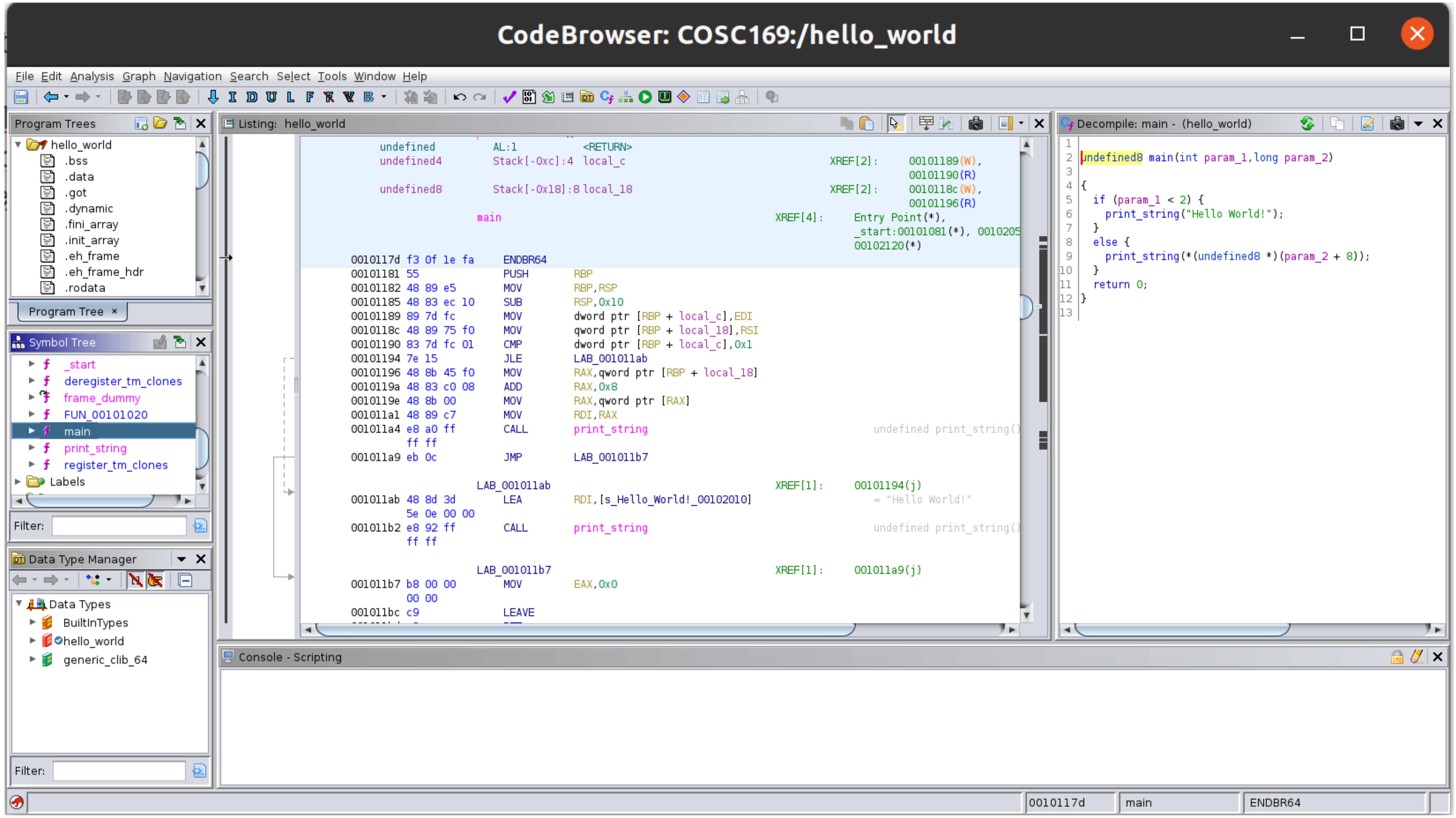
- Gives you an idea of how the execution flows through the program.
- You can see the CMP and subsequent JLE



Ghidra

Analyzing

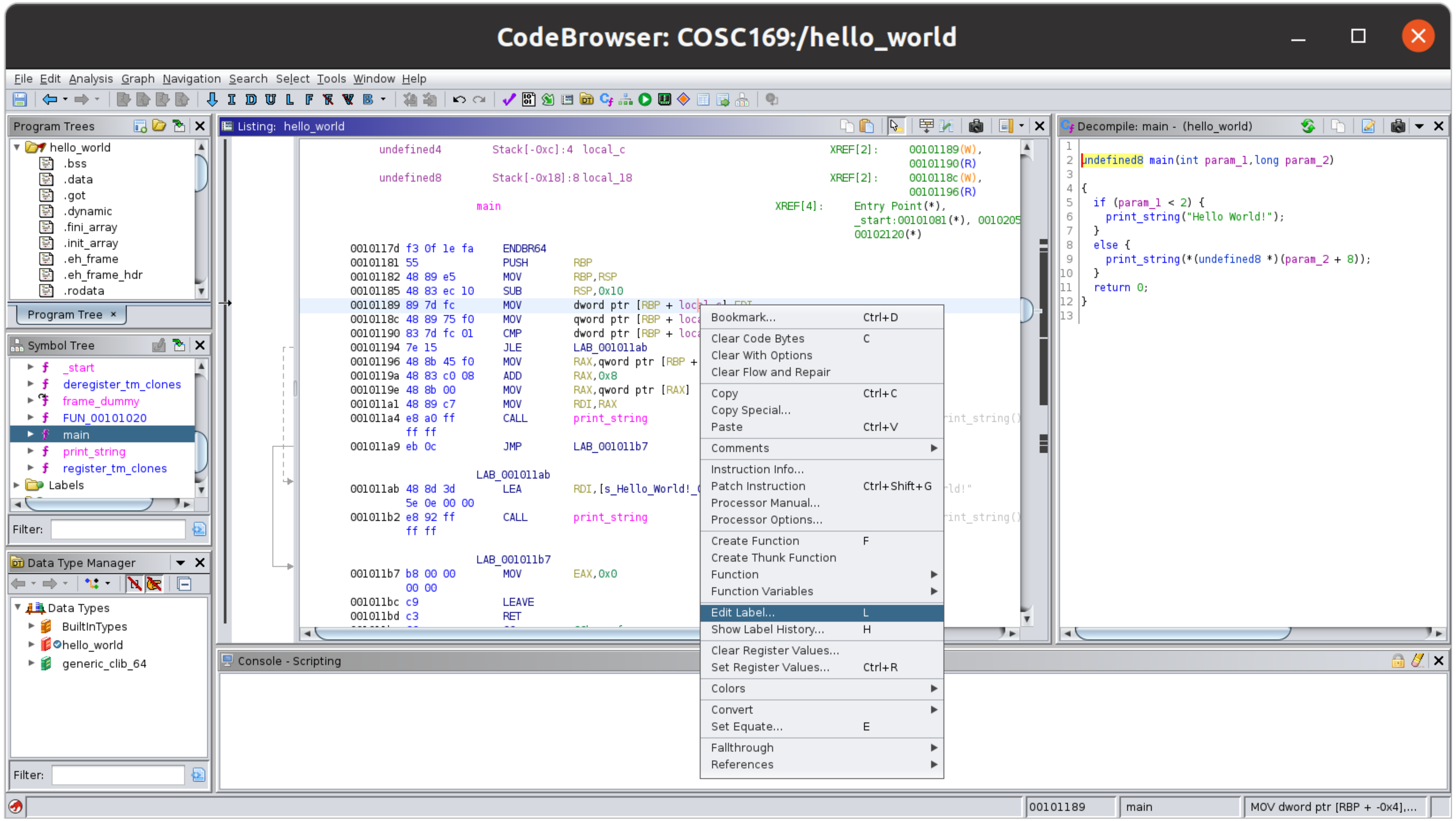
- Let's start filling out what we know about this function
- How many arguments are there to main()?
- What are they?
- Let's give them some names.
- What is happening at instruction 0x00101189?
- Name it.



Ghidra

Analyzing

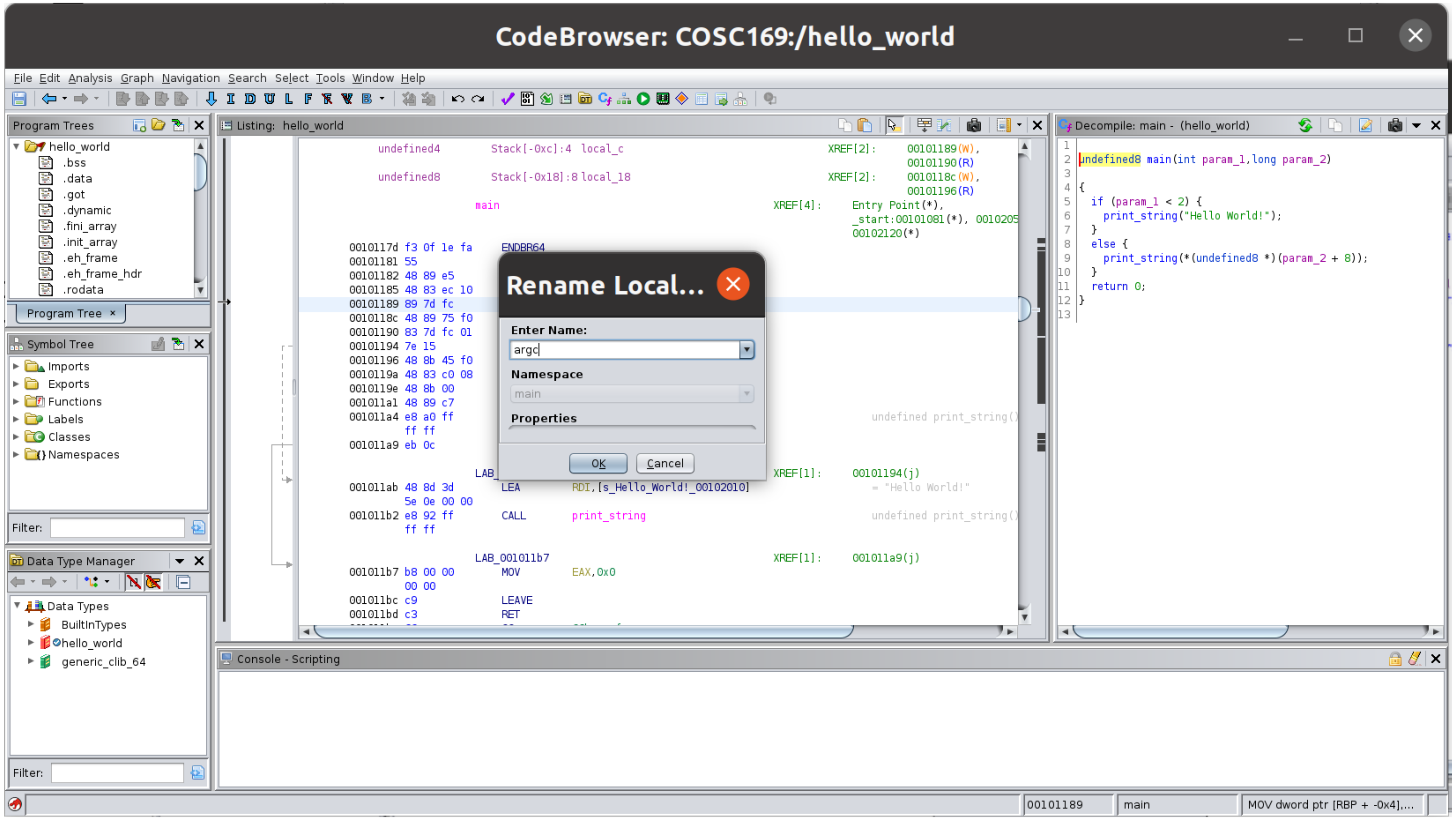
- Right or two finger click the name "local_c" -> Edit Label



Ghidra

Analyzing

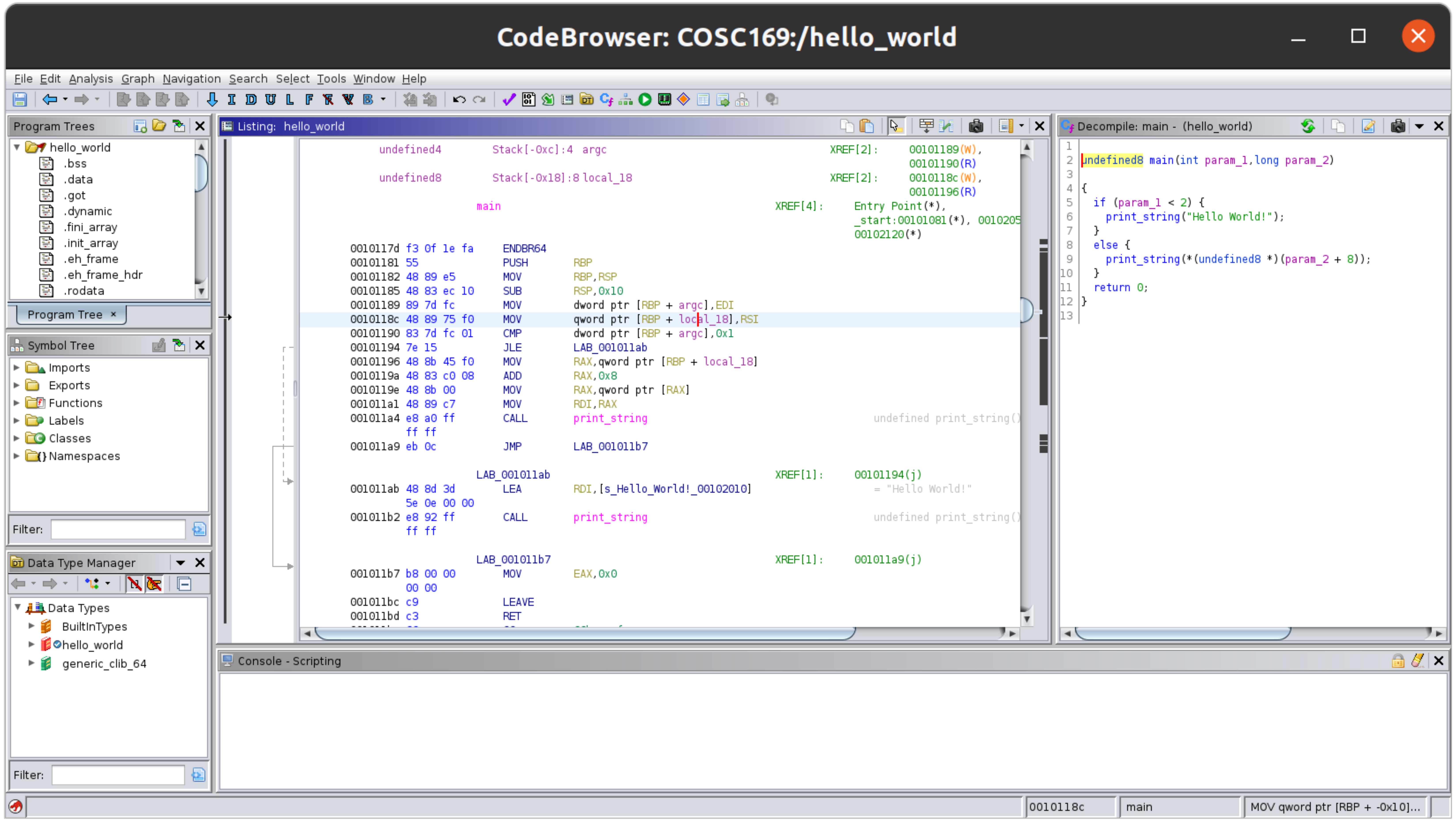
- Right or two finger click the name "local_c" -> Edit Label



Ghidra

Analyzing

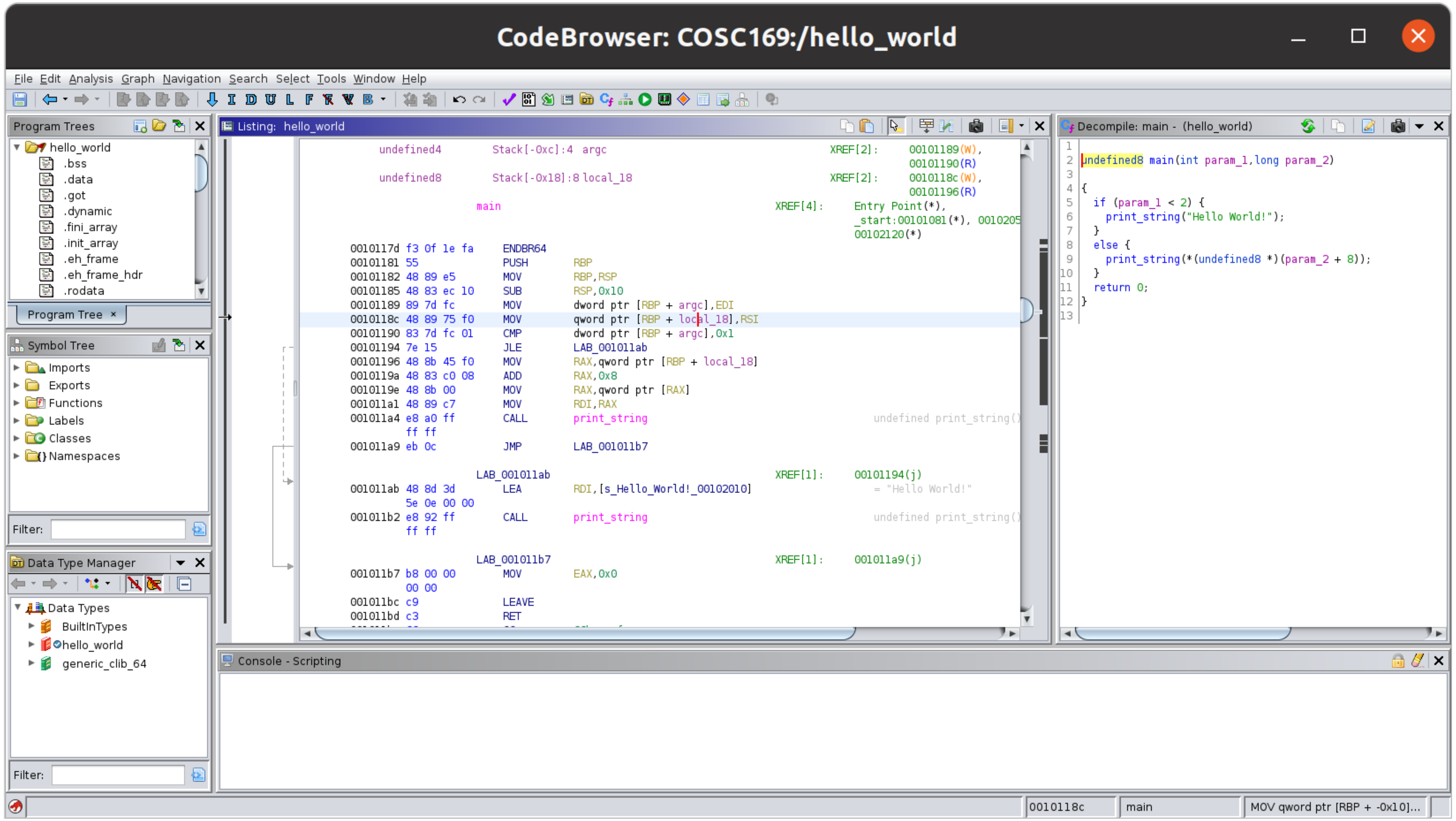
- You can see that the new name propagates
- What name should we give local_18?



Ghidra

Analyzing

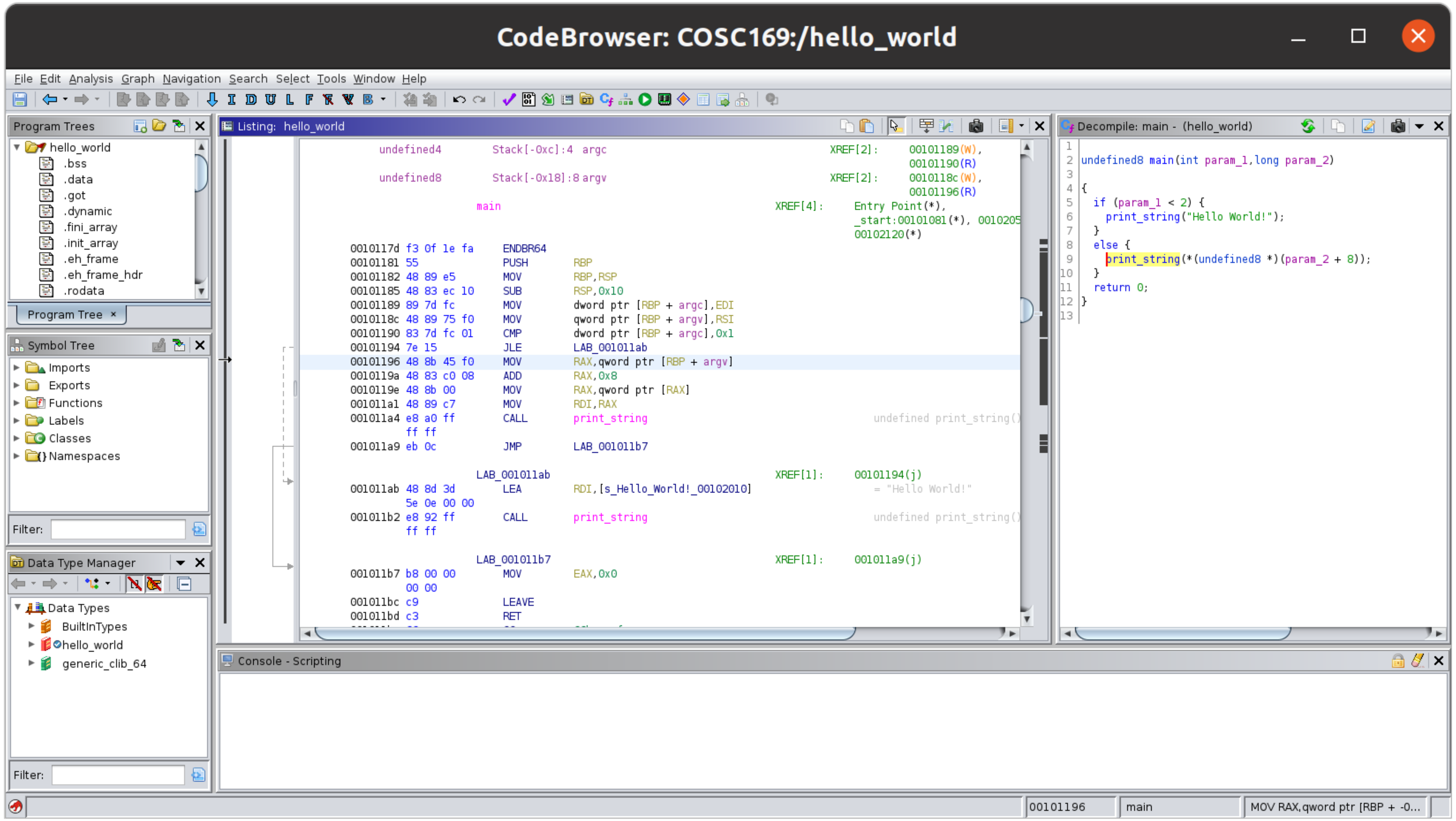
- If you said "argv" then you are correct.



Ghidra

Analyzing

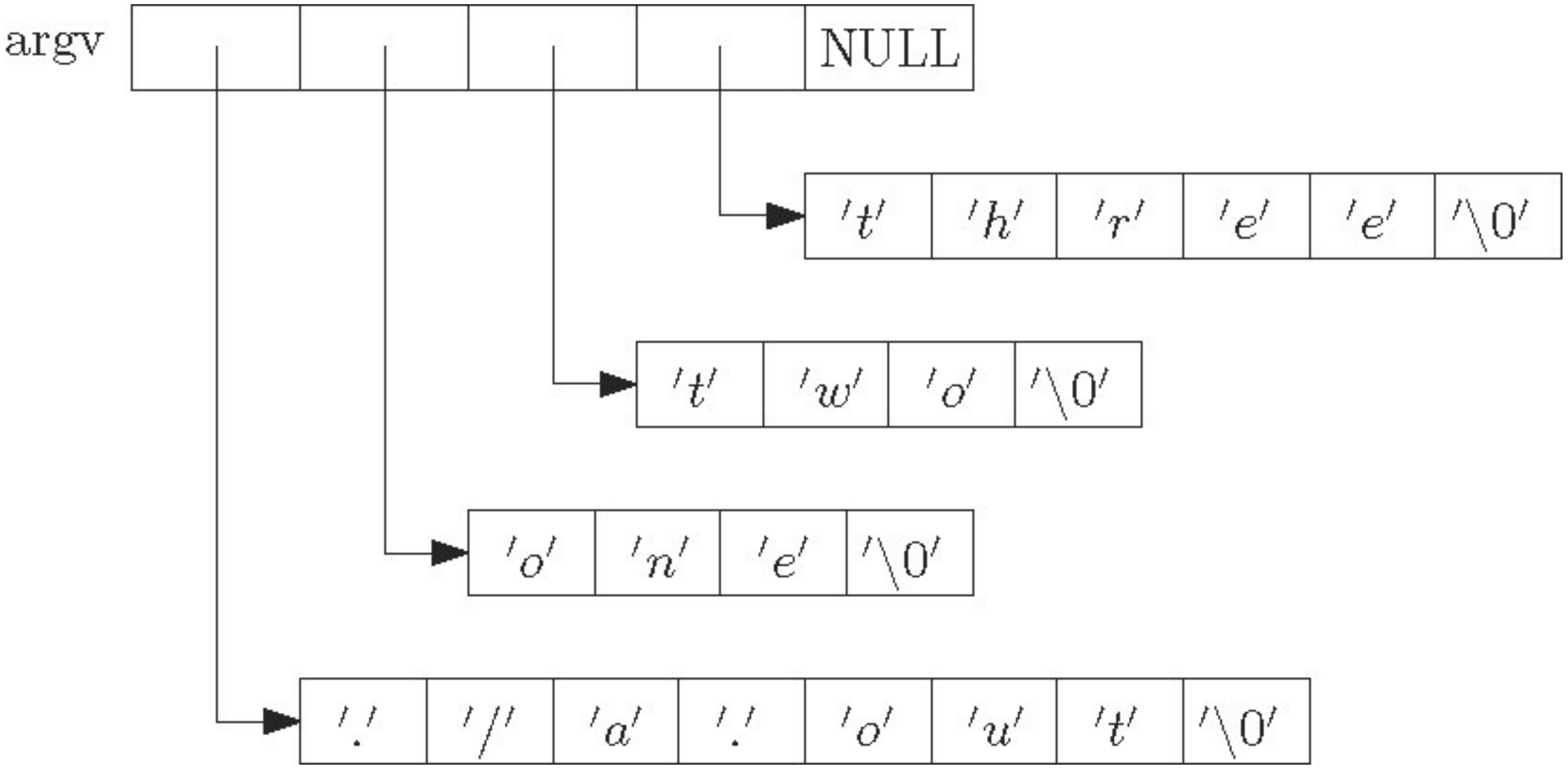
- If you said "argv" then you are correct.



argv: An aside

What is argv?

- An array of character pointers
- Each pointer points to a NULL byte delimited string
- The final entry is a NULL pointer
- On a 64-bit CPU each address is 8 bytes.

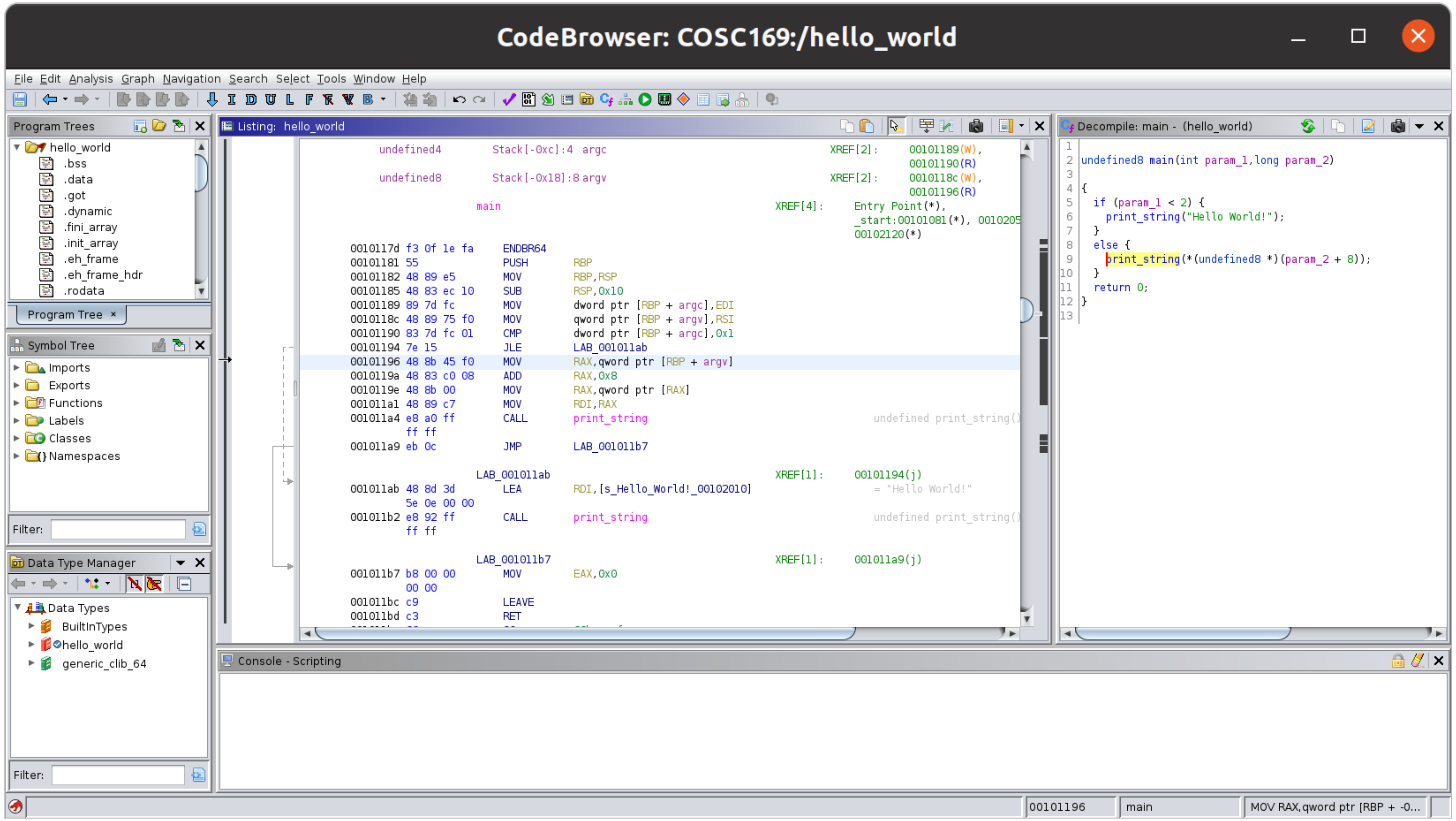


source: http://www.csc.villanova.edu/~mdamian/csc2405/assign/cmdargs.htm

Ghidra

Analyzing

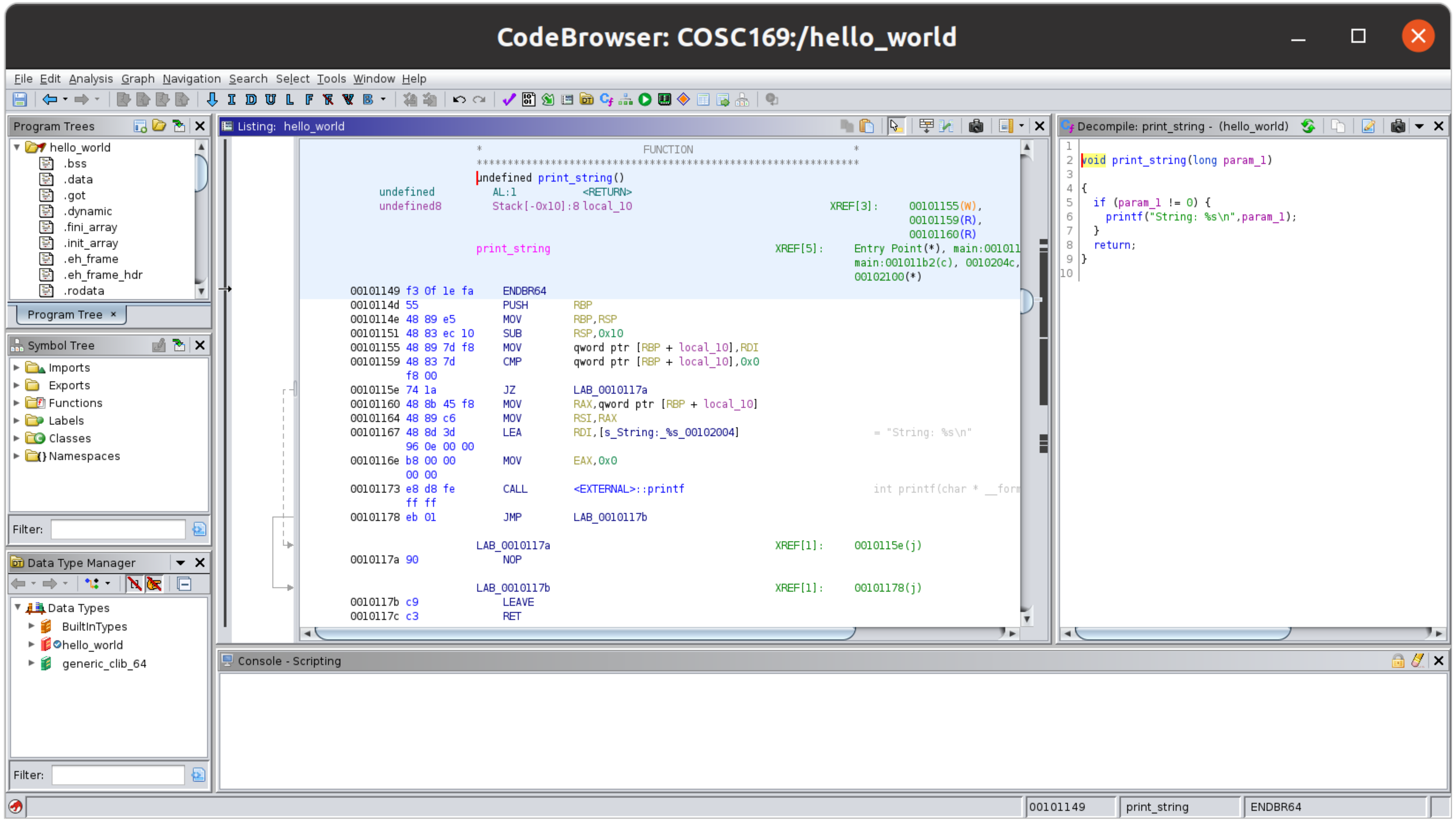
- Questions to ask:
  1. What is the compare at 0x101190 checking for?
  2. What are the instructions from 0x101196-0x1011a1 doing?
  3. Under what conditions will each block be executed?



Ghidra

Analyzing

- Double click on one of the calls to print_string
  1. How many arguments does the function take?
  2. What register(s) are used for the argument(s).
  3. What is the CMP at 0x101159 checking for?
  4. What gets printed?



Week 1 Recap

- Remember the INTEL manuals
- RE requires you to develop an intuition for how code works which only comes from practice.
- We will be teaching with Ghidra but there are other options available i.e. binary ninja, or IDA

Day 2 Homework

- You will be compiling and looking at the disassembly of 3 C programs
- If your machine isn't x86 you can log into the babylon servers and use those for compiling
- Make sure that by the next class you can connect to the babylon servers. We will be using them during class.