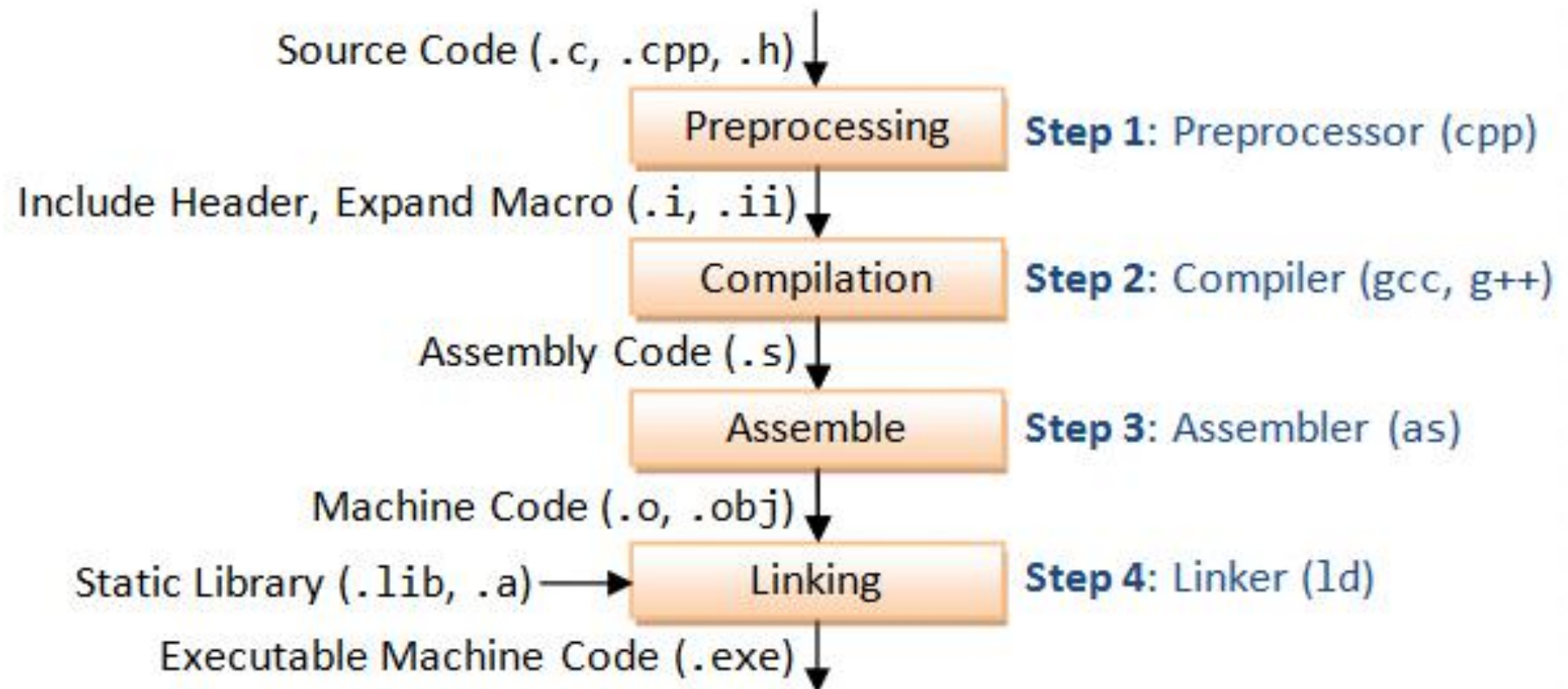


The GNU Compiler Collection (GCC)



The Preprocessor – Object Macros

```
#define BUFFER_SIZE 1024
```

```
foo = (char *) malloc (BUFFER_SIZE);
```

The `#define` directive can be used to set up symbolic replacements in the source.

The Preprocessor – Function Macros

```
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
```

```
#define twice(X) (2*(X))
```

```
y = min(1,2);
```

```
y = twice(1+1);
```

The Preprocessor – Imports

header.h

```
char *test(void)
```

program.c

```
#include "header.h"
```

```
int x;
```

```
int main(int argc, char *argv[]) {  
    puts(test());  
}
```

The `#include` directive just pastes in the text from the given file.

The Preprocessor – Demo

```
gcc -E -o hello.i hello.c
```

Preprocess `hello.c`, store output in `hello.i`

The Compiler – Demo

```
gcc -S hello.i
```

Compile preprocessed `.i` code into assembly instructions

The Assembler – Demo

```
as -o hello.o hello.s
```

Assemble object code from `hello.s`

The Assembler – ELF

a rare depiction of an Elf made by Tolkien



ELF: the Executable and Linkable Format

The Assembler – ELF

ELF: the Executable and Linkable Format

Cross-platform, used across multiple operating systems to represent components (object code) of a program. This comes in handy for linking and execution across different computers.

The Assembler – ELF

ELF: the Executable and Linkable Format

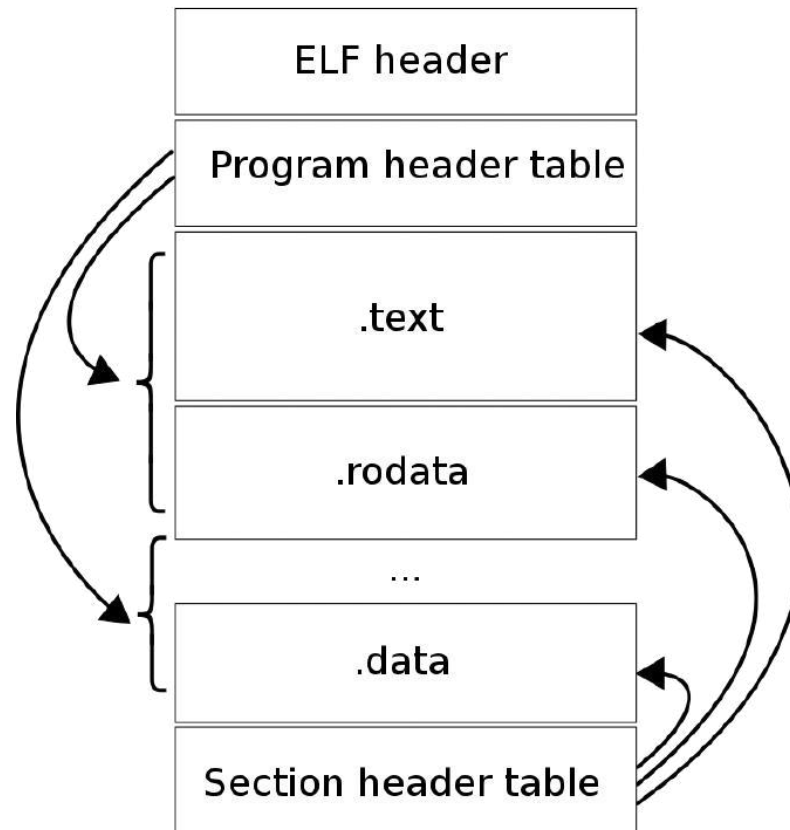
```
readelf -e hello.o
```

Actually read `hello.o`!
“-e” flag is for printing headers out only

The Assembler – ELF

Section	Contents	Code Example
<code>.text</code>	Executable code (x86 assembly)	<code>mov -0x8(%rbp),%rax</code>
<code>.data</code>	Any global or static vars that have a pre-defined value and can be modified	<code>int val = 3</code> (as global var)
<code>.rodata</code>	Variables that are only read (never written)	<code>const int a = 0;</code>
<code>.bss</code>	All uninitialized data; global variables and static variables initialized to zero or or not explicitly <code>static int i;</code> initialized in source code	<code>static int i;</code>
<code>.comment</code>	Comments about the generated ELF (details such as compiler version and execution platform)	

The Assembler – ELF



The Assembler

```
nm hello.o
```

Dump the variables and functions in hello and see what sections they belong to!

The Linker – Shared vs. Static Libraries

Static Linking

1. When your program uses static linking, the machine code of external functions used in your program is copied into the executable.
2. A static library has file extension of ".a" (archive file) in Unix.

Dynamic Linking

1. When your program is dynamically linked, only an offset table is created in the executable. The operating system loads the machine code needed for external functions during execution —a process known as dynamic linking.
2. A shared library has file extension of ".so" (shared objects) in Unix.

The Linker

```
ld --dynamic-linker /lib64/ld-linux-x86-64.so.2 hello.o  
-o hello -lc --entry main
```

1. **--dynamic-linker** is used to specify the linker we must use to load stdlib.
2. **-lc** tells the linker to link to the standard C library.
3. **--entry main** specifies the entry point of the program (the method "main").

Note: You may not get this command working, because it will be slightly different on different Linux distributions

Using Multiple Files

myadd.h (called a header file)

```
#ifndef _MYADD_H_
#define _MYADD_H_

int add(int x, int y);

#endif
```

myadd.c

```
#include "myadd.h"
int add(int x, int y)
{
    return x + y;
}
```

main.c

```
#include "myadd.h"

int main(int argc, char **argv)
{
    int a = 1;
    int b = 2;

    c = add(a,b);


    printf("%d + %d = %d", a, b, c);
}
```


Makefiles

Makefile

- *Makefile*: A list of *rules*.
- *Rule*: Tells Make the *commands* to build a *target* from 0 or more *dependencies*

`target: dependencies...`

 `commands`
`...`

Must indent with '\t', not spaces

Makefiles

Makefile = List of Rules

- *Rule*: Tells Make how to get to a ***target*** from ***source files***

```
target: dependencies...  
    commands  
...
```

“If dependencies have changed or don’t exist, rebuild them...
Then execute these commands.”

Generic Makefile

```
# A simple makefile for building a program composed of C source files.
#
PROGRAMS = hello

all:: $(PROGRAMS)

# It is likely that default C compiler is already gcc, but explicitly
# set, just to be sure
CC = gcc

# The CFLAGS variable sets compile flags for gcc:
# -g          compile with debug information
# -Wall       give verbose compiler warnings
# -O0         do not optimize generated code
# -std=gnu99  use the GNU99 standard language definition
CFLAGS = -g -Wall -O0 -std=gnu99

# The LDFLAGS variable sets flags for linker
# -lm        says to link in libm (the math library)
LDFLAGS = -lm

$(PROGRAMS): %:%.c
    $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

.PHONY: clean all

clean::
    rm -f $(PROGRAMS) *.o
```

Example – Makefile

```
# This Makefile should be used as a template for future Makefiles.
# It's heavily commented, so hopefully you can understand what each
# line does.

# We'll use gcc for C compilation and g++ for C++ compilation
CC = gcc
CXX = g++

# Let's leave a place holder for additional include directories
INCLUDES =

# Compilation options:
# -g for debugging info and -Wall enables all warnings
CFLAGS = -g -Wall $(INCLUDES)
CXXFLAGS = -g -Wall $(INCLUDES)

# Linking options:
# -g for debugging info
LD_FLAGS = -g

# List the libraries you need to link with in LDLIBS
# For example, use "-lm" for the math library
LDLIBS =

# The 1st target gets built when you type "make".
# It's usually your executable. ("main" in this case.)
#
# Note that we did not specify the linking rule.
# Instead, we rely on one of make's implicit rules:
#
# $(CC) $(LD_FLAGS) <all-dependent-.o-files> $(LDLIBS)
#
# Also note that make assumes that main depends on main.o,
# so we can omit it if we want to.
```

```
main: main.o myadd.o

# main.o depends not only on main.c, but also on myadd.h because
# main.c includes myadd.h. main.o will get recompiled if either
# main.c or myadd.h get modified.
#
# make already knows main.o depends on main.c, so we can omit main.c
# in the dependency list if we want to.
#
# make uses the following implicit rule to compile a .c file into a .o
# file:
#
# $(CC) -c $(CFLAGS) <the-.c-file>
#
main.o: main.c myadd.h

# And myadd.o depends on myadd.c and myadd.h.
myadd.o: myadd.c myadd.h

# Always provide the "clean" target that removes intermediate files.
# What you remove depend on your choice of coding tools
# (different editors generate different backup files for example).
#
# And the "clean" target is not a file name, so we tell make that
# it's a "phony" target.
.PHONY: clean
clean:
    rm -f *.o a.out core main

# "all" target is useful if your Makefile builds multiple programs.
# Here we'll have it first do "clean", and rebuild the main target.
.PHONY: all
all: clean main
```