Language Design and Compilers

Muchang Bahng

Summer 2025

Contents

Duck typing, Protocols @dataclass, @property macros Interfaces: what is a contract, More of "invocation", determined by which methods you have

A lot of this is gotten from the Gang of Four's *Design Patterns (1994)*. We are concerned with how programming languages are designed and then how we compile them into machine code.

The reason you define types and classes is to control how you interpret/encode data. Fundamentally all programming is manipulating bits. Safety measure on ourselves and convenience.

Definition 0.1 (Object)

An **object**[a] o is a run-time entity that packages both

1. data, also called *attributes*, and
2. the procedures—also called *methods*—that operate on that data.

[a] There is a bit of a circular definition with classes, but we'll ignore this for now.

Definition 0.2 (Signature)

The **signature** of a method f defines its name, parameters, and the return value.

Definition 0.3 (Interface)

An **interface** of an object o is set of all signatures defined by its operations. The interface describes the set of requests to which an object can respond.

Definition 0.4 (Type)

A **type** is the name of a particular interface.

1. A type A **subtype** of another B if its interface contains the interface of B.
2. A type B **supertype** of another A if A's interface contains the interface of B.

Note that while we have talked about interfaces through signatures, we haven't actually described how the functions are actually *implemented*. In object-oriented design, we want to decouple these two concepts, and objects are known only through their interfaces.

Definition 0.5 (Implementation)

An **implementation** of

1. a method f describes the actual computations done within the function.
2. an object o describes the implementations of all of its methods

Example 0.1 (Multiplication)

A function that multiplies two numbers might have a signature that looks like

```
int multiply(int x, int y)
```

But the implementations may differ.

```c
int multiply(int x, int y) {
  return x * y;
}
.
.
.
.
```

```c
int multiply(int x, int y) {
  int res = 0;
  for (int i = 0; i < y; i++) {
    res += x
  }
  return res;
}
```

Definition 0.6 (Class)

A **class** $C(a_1, \ldots, a_n, m_1, \ldots, m_k)$ is defines an object's interface and implementation. Objects are created by **instantiating** a class, and the object is said to be an **instance** of the class.

1. An **abstract class** is a class whose primary purpose is to define an interface and defers some or all of its implementation to subclasses. An abstract class cannot be instantiated. Its methods are called **abstract operations**, which is an operation that declares a signature but doesn't implement it.[1]
2. A **concrete class** is a class which has no abstract operations and can be instantiated.

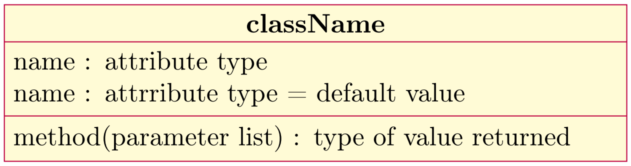
className
name : attribute type
name : attrribute type = default value
method(parameter list) : type of value returned

Figure 1: Class diagram depicts classes, their internal structure and operations, and the static relationships between them. The top box is the class name, the middle contains the attributes, and the bottom contains the methods.

In this sense, an abstract class is closer to an interface than an actual class, and we should view it as such.

This is why when you look at Python's docs, you see *type* hierarchy that defines interfaces (e.g. `numbers.Real`), which are abstract base classes.[2]

[2] Python models interfaces with ABCs.

1 Jan 7

Will be working in tiger.

1. *Lexer.* Convert a sequence of characters into a sequence of tokens. Need to talk about DFA, NFA, and regular expressions.
2. *Parsing.* Building the abstract syntax tree. e.g. LL parsing (what we are doing) vs LR parsing. MLyac. Trees make everything explicit and so is easier to work with.
3. *Type Checking.*
4. *IR.* Top of the mountain.
5. *Instruction Selection.*
6. *Liveness Analysis.* Data flow analysis, which is at the core of a lot of optimization.
7. *Register Allocation.* This gives the MIPS assembly, which is text.

Expression evaluates to a value, while a statement is ... SMl-NJ, sort of functional.

2 Regular Expressions

Example 2.1

The syntax for all numbers might look something like

```
0 | (-? [1-9][0-9])
```

To compute on regular expressions (i.e. matching a string with a regular expression), it is generally hard.

A little side track to DFA. Can we turn regular expressions into DFA? Yes. We do it through NFA.

3 Lexers

4 Jan 14

Last time, we found out that regexs are hard to compute, so we need DFAs and NFAs to compute. We can turn an NFA into a DFA. We like NFAs since it's easier to convert a regex into an NFA and then into a DFA. This allows us to write a nice declarative.

5 Encapsulation

Definition 5.1 (Encapsulation)

The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object. Operations are the only way to access and modify an object's representation.

public, private, protected + getter/setter methods Python doesn't have (true) encapsulation (or some people say it does, but it lacks access control), since the attributes can be accessed, even when there is name mangling.

6 Inheritance

But so far, these classes are isolated from one another.

This is bad because ideally, we want the classes to actually have relationships with each other

Definition 6.1 (Polymorphism)

The ability to substitute objects of matching interface for one another at run-time. The idea of *single interface, multiple implementations*.

Definition 6.2 (Toolkit)

A **toolkit** is a set of related and reusable classes designed to provide useful, general-purpose functionality.

Definition 6.3 (Framework)

A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software. It provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.

There are in general two ways: inheritance and object composition.

Definition 6.4 (Interface Inheritance)

Interface inheritance defines a new interface in terms of one or more existing interfaces.

Definition 6.5 (Implementation Inheritance)

Implementation inheritance defines a new implementation in terms of one or more existing implementations.

Class inheritance combines both interface and implementation inheritance, since the subclass.

Inheritance is not polymorphism. In inheritance, you get polymorphism when you cast it back to the base class.

subclassing vs inheritance. Invasive vs non-invasive.

abcs vs duck typing in python (PEP 3119).

7 Composition

Sometimes, inheritance may not be the right way. There can be a lot of independent properties of a certain class that we would like to model, so we can have child classes across different cross sections.

Example 7.1 (Visible, Solid, and Movable Objects)

The example in Wikipedia summarizes it nicely. Say you have some an interface to represent any object in the game, and you define three subtypes describing the properties it has.

```cpp
class Object {
public:
  virtual void update() {}
  virtual void draw() {}
  virtual void collide(Object objects[]) {}
};

class Visible : public Object
{
    Model* model;

public:
    virtual void draw() override {
        // code to draw a model at the position of this object
    }
};

class Solid : public Object
{
public:
    virtual void collide(Object objects[]) override {
        // code to check for and react to collisions with other objects
    }
};

class Movable : public Object
{
public:
    virtual void update() override {
        // code to update the position of this object
    }
}
```

If we want to implement the following concrete classes:

1. class `Player` which is `Solid`, `Movable`, and `Visible`,
2. class `Cloud`, which is `Movable` and `Visible` but not `Solid`,
3. class `Building` which is `Solid` and `Visible` but not `Movable`,
4. class `Trap` which is `Solid` but neither `Visible` not `Movable`.

Multiple inheritance is dangerous as we've seen before since it can lead to the diamond problem. One solution to this is the create classes such as `VisibleAndSolid`, `VisibleAndMovable`, etc. for all combinations, but this leads to repetitive code.

There must be a better way to organize this, and indeed composition comes to our rescue. The general idea is that rather than modeling the classes with *is-a* relationships, it is better to compose what an object can do with a *has-a* relationship.

Definition 7.1 (Object Composition)

Object composition is the principle that objects should contain instances of other classes that implement the desired functionality.

Definition 7.2 (Delegation)

In fact, object composition is so popular that there is a popular saying of *composition over inheritance.* It generally leads to more flexible and modular designs, leveraging the idea of building classes out of *components* rather than trying to find some commonality between them and creating a family tree.

Compiler	Interpreter
Takes more time to analyze source code but execution time is faster.	Takes less time to analyze source code but execution time is slower.
Debugging is harder since the compiler generates an error message after the entire scan.	Debugging is easier since the interpreter continues translating the program until an error is met.
Requires a lot of memory for generating object codes.	Requires less memory because no object code is generated.
Generates intermediate object code.	No intermediate object code is generated.

8 Program Lifecycle Phases

First, we review some definitions. More on program lifecycle phases here. Programming languages are broadly classified into two types. **High-level languages** are the familiar programming languages that we work with today (that allow much more abstraction), while **low-level languages** are very close to the hardware, such as machine language and assembly language. Programmers write programs in **source code** (usually high-level languages), which are then inputted into **language processors** that translate them into **object code** (usually **machine code** consisting of binary). The duration in which the source code of the program is being edited is called the **edit time**, while the **compile time** is when the source code is translated into machine code by a language processor. There are three types of language processors.

1. A **compiler** is a language processor that reads the complete source program written in high-level language as a whole in one go and translates it into an equivalent program in machine language. The source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of the compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again. (e.g. C, C++, C#, Java)

2. An **assmebler** is used to translate the program written in Assembly language (basically a low-level language with very strong correspondence between the instructions in the language and the machine code instructions) into machine code. The assembler is basically the 1st interface that is able to communicate humans with the machine. We need an assembler to fill the gap between human and machine so that they can communicate with each other. Code written in assembly language is some sort of mnemonics (instructions) like ADD, MUL, MUX, SUB, DIV, MOV and so on, and the assembler is basically able to convert these mnemonics into binary code.

3. An **interpreter** translates a single statement of the source program into machine code and executes immediately before moving on to the next line. If there is an error in the statement, the interpreter terminates its translating at that statement and displays an error message. The interpreter moves on to the next line for execution only after the removal of the error. An interpreter directly executes instructions written source code without previously converting them to an object code or machine code. (e.g. Python, Pearl, JavaScript, Ruby)

A quick compare and contrast.

The result of a successful compilation is an executable, which is a program in the form of a file containing millions of lines of very simple machine code instructions (e.g. add 2 numbers or compare 2 numbers), also called **processor instructions**. This executable can be stored somewhere in the computer drive for future use or it may be copied immediately in a faster memory state, such as the RAM. The **load time** is when the OS takes the program's executable from storage and puts it into an active memory (e.g. RAM) in order to begin execution.

The CPU understands only a low level machine code language (aka native code), which is contained within the executable. The language of the machine code is hardwired into the design of the CPU hardware; it is not something that can be changed at will. Each family of compatible CPUs (e.g. the popular Intel x86 family) has its own, idiosyncratic machine code which is not compatible with the machine code of other CPU families. More information here. Once the instruction bytes are copied from storage to RAM, the CPU can run through the steps/lines at the rate of about 2 billion lines/steps per second. This execution phase, when the CPU executes the instructions until normal termination or a crash, is called the **runtime**.

8.1 More on Executables

More specifically, an **executable** is a file that contains a list of instructions and data to cause a computer's CPU to perform indicated tasks, as opposed to the data files, which are fundamentally strings of data that must be interpreted (parsed) by a program to be meaningful. Executables usually have extension names `.exe` or `.bat`, and they can generally be run (invoked) in two ways:

1. The executable file can be run by simply double clicking on the file name, opening it, and having the user type commands in an interactive session of an interpter (like inputting commands in terminal window or a python shell).
2. Alternatively, we can start writing a program, complete writing it, and then have this program compiled into an executable to be invoked.

Some common examples of executables are:

1. python.exe - used to run python scripts that have the .py extension, located at

   ```
   C: \Users\bahng\AppData\Local\Programs\Python\Python39
   ```

2. pythonw.exe - used to run .pyw files for GUI programs
3. terminal.exe (on MacOS)
4. cmd.exe (on Windows OS)
5. py.exe - an executable used to run the python.exe executable like a shortcut, located at

   ```
   C:\windows\py.exe
   ```

8.2 Static vs Dynamic Languages

Type-checking is the process of checking and verifying the type of a construct (constant, variable, array, list, object) and its usage context. It helps in minimizing the possibility of type errors in the program, and type checking may occur either at compile-time (static checking) or at run-time (dynamic checking).

1. **Statically-Typed Languages**: Since we type check during compilation, every detail about the variables and all the data types must be known before we do the compiling process. Once a variable is assigned a type, it can't be assigned to some other variable of a different type, and so the data type of a declared variable is fixed. This makes sense since in Java, C, C++, etc., the programmer must specify what the data type of each variable is by writing something like `int myNum = 15`.
2. **Dynamically-Typed Languages**: Since we type-check during runtime, there is no need to specify the data type of each variable while writing code, which improves writing speed. These languages have the capability to identify the type of each variable during run-time, so we do not need to declare the data types of variables. In these languages, variables are bound to objects at run-time using assignment statements, and most modern languages (e.g. JavaScript, Python, PHP, etc.) are dynamically typed.

9 Compiling and Linking

Now let's talk about how this compiling actually happens. *Compiling* is actually an umbrella term that is misused. Turning at C file into an executable file consists of multiple intermediate steps, one of which is actually compiling, but the whole series is sometimes referred to as compiling. A more accurate term would be *building.* Before we get onto it, there are two types of compilers.

Definition 9.1 (GCC, CLang)

The two mainstream compilers used is GCC (with the gdb debugger) and Clang (with lldb). For now, the difference is that

1. gcc is more established.
2. clang is newer and has more features.

A useful flag to know is that we can always specify the name of the (final or intermediary) output file with the `-o` flag.

Definition 9.2 (Complete Build Process)

To actually turn a C file into an executable file, we need to go through a series of steps. We start off with the C code, which are the `.c`, `.cpp`, or `.h` files.

1. **Preprocessing**: The precompiler step expands the *preprocessor directives* (all the `#include` and `#define` statements) and removes comments. This results in a `.i` file. The preprocessor will replace these macros with the actual code. This results in a `.i` file.

   ```
   clang/gcc -E main.c -o main.i
   ```

2. **Compiling**: We take these and generate assembly code. This results in a `.asm` or `.s` file.

   ```
   clang/gcc -S main.c -o main.s
   ```

3. **Assembler**: We take the assembly code and generate machine code in the form of relocatable binary object code (this is machine code, not assembly). This results in a `.o` or `.obj` file.

   ```
   clang/gcc -c main.c -o main.o
   ```

4. **Linking**: We take these object files and link them together to form an executable file. This results in a `.exe` or `.out` file.

The GCC or CLang compiler automates this process for us. For example, `gcc -c hello.c` generates an object file, taking care of the preprocessing, compiling, and assembling code. Then, `gcc hello.o` links the object file to generate an executable file.

There are a lot of questions to be asked here, and we will go through them step by step.

9.1 Precompiling Stage

Just like how Python package managers like conda have specific directories that they find package in, the C library also has a certain directory.

Definition 9.3 (Standard Library Directory)

In Linux systems, there are two main directories you look at:

1. `/usr/include` contains the standard C library headers.
2. `/usr/local/include` contains the headers for libraries that you install yourself.

In Mac Silicon, these directories are a little bit more involved. You must first install the xcode command line developer tools, which will then create these directories.

1. The standard C library headers are in

   `/Library/Developer/CommandLineTools/SDKs/MacOSX*.sdk/usr/include`.

In here, we can find all the relevant import files like `stdio.h` and such. When we precompile, the output `.i` file represents a precompiled C file. It still has C code, but it has been optimized to

1. Remove comments.
2. Replace all the `#include` statements with the actual code.
3. Replace all the global variables declared in `#define` with the actual value.

Between x86 and ARM, there are no significant differences in how C files are precompiled.

Example 9.1

Take a look at the following minimal example.

```c
#include "second.h"
#define a 3

int add(int x, int y) {
  return x + y;
}

int main() {
  // test comment
  int b = 5;
  int c = add(a, b);
  int d = subtract(a, b);
  return 0;
}
```

```c
int subtract(int a, int b) {
  return a - b;
}
.
.
.
.
.
.
.
.
.
.
.
```

Figure 2: I have included a `main.c` file that imports statements from a `second.h` file.

Now, I run `gcc -E main.c -o main.i` to generate the precompiled file, which gives me the following.

```
# 1 "main.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 418 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "main.c" 2
# 1 "./second.h" 1
int subtract(int a, int b) {
  return a - b;
}
# 2 "main.c" 2


int add(int x, int y) {
  return x + y;
}

int main() {

  int b = 5;
  int c = add(3, b);
  int d = subtract(3, b);
  return 0;
}
```

Figure 3: The precompiled file.

Notice a few things:

1. The header file `second.h` has been replaced with the actual code.
2. The comments have indeed been removed.
3. The global variable `a` has been replaced with the actual value 3.

This leaves us with the question of what all the rest of the lines that start with a `#` are for. They are called *preprocessor directives.*

Definition 9.4 (Preprocessor Directives)

Preprocessor directives are commands that are executed before the actual compilation begins. These directives allow additional actions to be taken on the C source code before it is compiled into object code. Directives are not part of the C language itself, and they are always prefixed with a `#` symbol.

1. `#include` is used to include the contents of a file into the source file. It selects portions of the file to include based on the file name.
2. `#define` is used to define a macro, which is a way to give a name to a constant value or a piece of code.
3. `#ifdef`, `#ifndef`, `#else`, and `#endif` are used for conditional compilation.
4. `#error` is used to generate a compilation error.
5. `#pragma` is used to give the compiler specific instructions.

9.2 Compiling Stage

Once we have precompiled, we can compile the code into assembly code. For the following two examples, we will parse through the general syntax of assembly code. It is quite different between x86 and ARM, so we will use the minimal C code

```
int add(int x, int y) {
  return x + y;
}

int main() {
  int a = 3;
  int b = 5;
  int c = add(a, b);
  return 0;
}
```

for both examples.

Example 9.2 (x86 Compiled Assembly Language)

The assmebly code is shown.

```
.
  .file  "main.c"
  .text
  .globl add
  .type  add, @function
add:
.LFB0:
  .cfi_startproc
  endbr64
  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
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
.LFE0:
  .size  add, .-add
  .globl main
  .type  main, @function
main:
.LFB1:
  .cfi_startproc
  endbr64
  pushq  %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq   %rsp, %rbp
  .cfi_def_cfa_register 6
  subq   $16, %rsp
  movl   $3, -12(%rbp)
```

```
	movl	$5, -8(%rbp)
	movl	-8(%rbp), %edx
	movl	-12(%rbp), %eax
	movl	%edx, %esi
	movl	%eax, %edi
	call	add
	movl	%eax, -4(%rbp)
	movl	$0, %eax
	leave
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE1:
	.size	main, .-main
	.ident	"GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
	.section	.note.GNU-stack,"",@progbits
	.section	.note.gnu.property,"a"
	.align 8
	.long	 1f - 0f
	.long	 4f - 1f
	.long	 5
0:
	.string	 "GNU"
1:
	.align 8
	.long	 0xc0000002
	.long	 3f - 2f
2:
	.long	 0x3
3:
	.align 8
4:
```

Example 9.3 (ARM Compiled Assembly Language)

The assembly code is shown.

```
.
	.section	__TEXT,__text,regular,pure_instructions
	.build_version macos, 14, 0	sdk_version 14, 4
	.globl	_add                            ; -- Begin function add
	.p2align	2
_add:                                   ; @add
	.cfi_startproc
; %bb.0:
	sub	sp, sp, #16
	.cfi_def_cfa_offset 16
	str	w0, [sp, #12]
	str	w1, [sp, #8]
	ldr	w8, [sp, #12]
	ldr	w9, [sp, #8]
	add	w0, w8, w9
	add	sp, sp, #16
	ret
	.cfi_endproc
```

```
                                        ; -- End function
  .globl _main                           ; -- Begin function main
  .p2align  2
_main:                                  ; @main
  .cfi_startproc
; %bb.0:
  sub sp, sp, #48
  .cfi_def_cfa_offset 48
  stp x29, x30, [sp, #32]             ; 16-byte Folded Spill
  add x29, sp, #32
  .cfi_def_cfa w29, 16
  .cfi_offset w30, -8
  .cfi_offset w29, -16
  mov w8, #0
  str w8, [sp, #12]                   ; 4-byte Folded Spill
  stur   wzr, [x29, #-4]
  mov w8, #3
  stur   w8, [x29, #-8]
  mov w8, #5
  stur   w8, [x29, #-12]
  ldur   w0, [x29, #-8]
  ldur   w1, [x29, #-12]
  bl  _add
  mov x8, x0
  ldr w0, [sp, #12]                   ; 4-byte Folded Reload
  str w8, [sp, #16]
  ldp x29, x30, [sp, #32]             ; 16-byte Folded Reload
  add sp, sp, #48
  ret
  .cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

We can see that in both examples, there are generally two types of codes.

1. The regular CPU operations with registers and memory.
2. Some code starts off with some code that starts with a `.`. Every line that starts with a `.` are called *assembler directives*.

Let's elaborate more on what these directives are.

Definition 9.5 (Assembler Directives)

An **assembler directives** are instructions in assembly language programming that that give commands to the assembler (which then converts this to an object file) about various aspects of the assembly process, but they do not represent actual CPU instructions that execute in the program. Unlike typical assembly language instructions that directly manipulate registers and execute arithmetic or logical operations, directives are used to organize, control, and provide necessary information for the assembly and linking of binary programs. They can manage memory allocation, define symbols, control compilation settings, and much more.

There are general types of directives that are common in both x86 and ARM that we should be aware about:

1. Section directives.
2. Data allocation directives.

3. Symbol definition directives.
4. Macro and Include directives.
5. Debugging and error handling directives.

Example 9.4 (x86 Assembly Directives)

Let us elaborate on the specific directives in the x86 assembly code, some of which are in the example above.

1. `.file "main.c"` is a directive that tells the assembler that the following code is from the file `main.c`. It is a form of metadata.
2. `.text` is a directive that tells the assembler that the following code is the text section (the text/code portion of memory) of the program. This is where the actual code is stored.
3. `.globl add` is a directive that tells the assembler that the following code is a global function called `add`.
4. `.type add, @function` is a directive that tells the assembler that the following code is a function.

Example 9.5 (ARM Assembly Directives)

You also see that there are symbols that represent memory addresses. Let's elaborate on what symbols mean.

Definition 9.6 (Symbol)

A **symbol** is a name that is used to refer to a memory location. It can be a function name, a global variable, or a local variable.

1. Global symbols are symbols that can be referenced by other object files, e.g. non-static functions and global variables.
2. Local symbols are symbols that are only visible within the object file, e.g. static functions and local variables. The linker won't know about these types.
3. External symbols are referenced by this object file but defined in another object file.

9.3 Objdump

Since we will be using the `objdump` package quite a lot, it is worth mentioning the different commands you will use and store them here as a reference. For first readers, don't expect to know what each of them do, but rather look back at this for a reference.

9.3.1 ELF and Mach-O Formats

Objdump is a command line utility that is used to display information about object files, which are often outputted in a specific format. The two main output file types are called ELF (Executable and Linkable Format) and Mach-O (Mach Object).

Definition 9.7 (ELF)

The **Executable and Linkable Format** (ELF) is a common standard file format for executables, object code, shared libraries, and core dumps. It is analogous to a book, with the following parts:

1. **Header**, which is like the cover of the book. It contains metadata about the file, such as the architecture, the entry point, and the sections.
2. **Sections**, which are like chapters. Each section contains the content for some given purpose or

use wthin the program. e.g. `.binary` is just a block of bytes, `.text` contains the machine code, `.data` contains initialized data, and `.bss` contains uninitialized data.

3. **Symbol Table**, is like a detailed table of contents of all defined symbols such as functions, external (global) variables, local maps, etc.
4. **Relocation records**, which is like the index of the book that lists references to symbols.

The format is generally as such when you run `objdump -d -r hello.o` (d represents disassembly and r represents relocation entries).

```
ELF header         # file type

.text section
  - code goes here

.rodata section
  - read only data

.data section
  - initialized global variables

.bss section
  - uninitialized global variables

.symtab section
  - symbol table (symbol name, type, address)

.rel.text section
  - relocation entries for .text section
  - addresses of instructions that will need to be modified in the executable.

.rel.data section
  - relocation info for .data section
  - addresses of pointer data that will need to be modified in the merged executable.

.debug section
  - info for symbolic debugging (gcc -g)
```

Definition 9.8 (Mach-O)

9.3.2 Objdump Commands

Theorem 9.1 (File Headers with Objdump)

Given that you have an object file, the first thing you might want to do is see the file header. You do with this `objdump -f main.o`.

```
main.o:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
```

Theorem 9.2 (Section with Objdump)

To look at the section headers to get a closer overview, you use `objdump -h main.o`.

```
main.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000004b  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  0000008b  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  0000008b  2**0
                  ALLOC
  3 .comment      0000002c  0000000000000000  0000000000000000  0000008b  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000b7  2**0
                  CONTENTS, READONLY
  5 .note.gnu.property 00000020  0000000000000000  0000000000000000  000000b8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000d8  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

Theorem 9.3 (Disassembly with Objdump)

Now you might actually want to look at the disassembly of the code, which is what we often use it for. To do this, you use `objdump -D main.o` to get the entire output.

1. The leftmost column represents the address of the instruction.
2. The next column represents the machine code of the instruction.
3. The next column represents the assembly code of the instruction.

```
main.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <add>:
   0: f3 0f 1e fa             endbr64
   ...
  17: c3                      retq

0000000000000018 <main>:
  18: f3 0f 1e fa             endbr64
  ...
  4a: c3                      retq

Disassembly of section .comment:

0000000000000000 <.comment>:
   0: 00 47 43                add    %al,0x43(%rdi)
   ...
  2a: 30 00                   xor    %al,(%rax)

Disassembly of section .note.gnu.property:

0000000000000000 <.note.gnu.property>:
   0: 04 00                   add    $0x0,%al
  ...

Disassembly of section .eh_frame:

0000000000000000 <.eh_frame>:
   0: 14 00                   adc    $0x0,%al
  ...
```

If you just want to look at the contents of the executable sections, then you can use `objdump -d main.o`.

```
main.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <add>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 89 7d fc                mov    %edi,-0x4(%rbp)
   b: 89 75 f8                mov    %esi,-0x8(%rbp)
   e: 8b 55 fc                mov    -0x4(%rbp),%edx
  11: 8b 45 f8                mov    -0x8(%rbp),%eax
  14: 01 d0                   add    %edx,%eax
  16: 5d                      pop    %rbp
  17: c3                      retq

0000000000000018 <main>:
  18: f3 0f 1e fa             endbr64
  1c: 55                      push   %rbp
  1d: 48 89 e5                mov    %rsp,%rbp
  20: 48 83 ec 10             sub    $0x10,%rsp
  24: c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  2b: c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  32: 8b 55 f8                mov    -0x8(%rbp),%edx
  35: 8b 45 f4                mov    -0xc(%rbp),%eax
  38: 89 d6                   mov    %edx,%esi
  3a: 89 c7                   mov    %eax,%edi
  3c: e8 00 00 00 00          callq  41 <main+0x29>
  41: 89 45 fc                mov    %eax,-0x4(%rbp)
  44: b8 00 00 00 00          mov    $0x0,%eax
  49: c9                      leaveq
  4a: c3                      retq
```

If you want to see the source code intermixed with disassembly, then you can use the `-S` flag, but make sure that the object file is a generated with debugging information, i.e. use `gcc -c -g main.c -o main.o`.

```
main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <add>:
int add(int x, int y) {
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 89 7d fc                mov    %edi,-0x4(%rbp)
   b: 89 75 f8                mov    %esi,-0x8(%rbp)
  return x + y;
   e: 8b 55 fc                mov    -0x4(%rbp),%edx
  11: 8b 45 f8                mov    -0x8(%rbp),%eax
  14: 01 d0                   add    %edx,%eax
}
  16: 5d                      pop    %rbp
  17: c3                      retq

0000000000000018 <main>:

int main() {
  18: f3 0f 1e fa             endbr64
  1c: 55                      push   %rbp
  1d: 48 89 e5                mov    %rsp,%rbp
  20: 48 83 ec 10             sub    $0x10,%rsp
  int a = 3;
  24: c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  int b = 5;
  2b: c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  int c = add(a, b);
  32: 8b 55 f8                mov    -0x8(%rbp),%edx
  35: 8b 45 f4                mov    -0xc(%rbp),%eax
  38: 89 d6                   mov    %edx,%esi
  3a: 89 c7                   mov    %eax,%edi
  3c: e8 00 00 00 00          callq  41 <main+0x29>
  41: 89 45 fc                mov    %eax,-0x4(%rbp)
  return 0;
  44: b8 00 00 00 00          mov    $0x0,%eax
}
  49: c9                      leaveq
  4a: c3                      retq
```

Figure 4: Disassembly of the object file back into assembly using `objdump -d -S main.o`.

Note that you can always see this disassembly with debuggers like gdb or lldb, but objdump generally works for all architectures.

Theorem 9.4 (Symbol Table)

If you want to look at all the symbols existing within the object file, you use `objdump -t main.o` (t for table of symbols).

1. The leftmost column represents the address of the symbol.

2. The next column represents the type of the symbol. The `g` and `l` represent global and local symbols, respectively. The `O` and `F` represent object and function symbols, while the `UND` and `ABS` represent undefined and absolute symbols.
3. The next column represents the section that the symbol is in.
4. The next column represents the size of the symbol.
5. The last column represents the name of the symbol.

```
main.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*   0000000000000000 main.c
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack  0000000000000000 .note.GNU-stack
0000000000000000 l    d  .note.gnu.property  0000000000000000 .note.gnu.property
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     F .text   0000000000000018 add
0000000000000018 g     F .text   0000000000000033 main
```

Theorem 9.5 (Relocation Table)

If you want to look then at the relocation table, then you use `objdump -r main.o`.

1. The leftmost column represents the offset of the relocation (i.e. the location within the section where this relocation needs to be applied).
2. The second column represents the type of relocation.
3. The third column represents the symbol that this relocation references.

```
main.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000003d R_X86_64_PLT32    add-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
0000000000000040 R_X86_64_PC32     .text+0x0000000000000018
```

9.4 Assembling Stage and Object Files

Now, once you have gotten the object file, you cannot simply open it up in a text edit as it is in machine code. To actually interpret anything from it, you must **disassmble** it, meaning that you convert the machine code back into assembly code. The main software that you use to do this is `objdump`. Let's take a look again at the object file.

```
Disassembly of section .text:

0000000000000000 <add>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 89 7d fc                mov    %edi,-0x4(%rbp)
   b: 89 75 f8                mov    %esi,-0x8(%rbp)
   e: 8b 55 fc                mov    -0x4(%rbp),%edx
  11: 8b 45 f8                mov    -0x8(%rbp),%eax
  14: 01 d0                   add    %edx,%eax
  16: 5d                      pop    %rbp
  17: c3                      retq

0000000000000018 <main>:
  18: f3 0f 1e fa             endbr64
  1c: 55                      push   %rbp
  1d: 48 89 e5                mov    %rsp,%rbp
  20: 48 83 ec 10             sub    $0x10,%rsp
  24: c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  2b: c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  32: 8b 55 f8                mov    -0x8(%rbp),%edx
  35: 8b 45 f4                mov    -0xc(%rbp),%eax
  38: 89 d6                   mov    %edx,%esi
  3a: 89 c7                   mov    %eax,%edi
  3c: e8 00 00 00 00          callq  41 <main+0x29>
  41: 89 45 fc                mov    %eax,-0x4(%rbp)
  44: b8 00 00 00 00          mov    $0x0,%eax
  49: c9                      leaveq
  4a: c3                      retq
```

Figure 5: Disassembly of the object file back into assembly using `objdump -d main.o`.

Let's note a couple things.

1. The functions are organized by their starting address followed by their name, e.g.

   ```
   0000000000000000 <add>:
   ```

   Within each function, each line of assembly code is shown. To find the total memory the function takes up, you can just take the address of the last line and subtract it from the address of the first line. Or you can literally count the number of bytes in each line (remember 2 hex is 1 byte).

2. The line that calls the `add` function is `0x0` (`00 00 00 00`), with is the *relative target address* intended to be filled in by the linker. The actual assembly line just says that the function continues on to the next line at address `0x41`. This is because the object file is not aware of where it will be loaded into memory, and all lines with this opcode `e8 00 00 00 00` is intended to be filled in by the linker.

3. Look at address `0x3c`. It is calling another function, but the values starting from address `0x3d` is `00 00 00 00`, which is not the actual address of the function but also a dummy address. This is because the object file is not aware of where the function is located in memory.

9.5 Linking Stage and Relocation

9.5.1 Relocation

If the object file is already in machine code, then why do we need a separate linking stage that converts `main.o` into `main` the binary? The reason is stated in the previous section: because the object files uses relative memory addressing and does not know about which memory is accessed in other object files, we need to **relocate** the symbols in the object file to their proper addresses. So how does the linker actually know how to relocate these symbols into their proper addresses? It uses the *relocation table*, which contains information about the addresses that need to be modified in the object file.

```
main.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000003d R_X86_64_PLT32    add-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
0000000000000040 R_X86_64_PC32     .text+0x0000000000000018
```

Figure 6: Relocation table for `main.o` object file.

Let's talk about how to actually read this table. We can look at the first entry, which shows an offset of `0x3d`. This represents the offset from the beginning of the `.text` section where the relocation needs to be applied. Looking back at the disassembly file, this address `0x3d` is precisely where there was a dummy address `00 00 00 00`. We want to replace this with the actual address defined in the `VALUE` column, which is `add` (with a slight offset of `0x4`, which is typically used to compensate for the PC-relative addressing mode where the CPU might be adding the length of the instruction to the program counter (PC) before the relocation value is applied). The type of relocation won't be covered in our scope. Let's go through each relocation entry:

1. The first entry is for the `add` function. If we look at the disassembly, within the `main` function, the address `0x3d` is where the `add` function is called. The linker will replace the dummy address with the actual address of the `add` function.

```
Disassembly of section .text:

0000000000000000 <add>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 89 7d fc                mov    %edi,-0x4(%rbp)
   b: 89 75 f8                mov    %esi,-0x8(%rbp)
   e: 8b 55 fc                mov    -0x4(%rbp),%edx
  11: 8b 45 f8                mov    -0x8(%rbp),%eax
  14: 01 d0                   add    %edx,%eax
  16: 5d                      pop    %rbp
  17: c3                      retq

0000000000000018 <main>:
  18: f3 0f 1e fa             endbr64
  1c: 55                      push   %rbp
  1d: 48 89 e5                mov    %rsp,%rbp
  20: 48 83 ec 10             sub    $0x10,%rsp
  24: c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  2b: c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  32: 8b 55 f8                mov    -0x8(%rbp),%edx
  35: 8b 45 f4                mov    -0xc(%rbp),%eax
  38: 89 d6                   mov    %edx,%esi
  3a: 89 c7                   mov    %eax,%edi
  3c: e8 00 00 00 00          callq  41 <main+0x29>     <-- here
  41: 89 45 fc                mov    %eax,-0x4(%rbp)
  44: b8 00 00 00 00          mov    $0x0,%eax
  49: c9                      leaveq
  4a: c3                      retq
```

2. The second and third entries are for the `.eh_frame` section. We can see that the offset of `0x20` and `0x40` represents the following lines below. They also have dummy addresses that need to be replaced. They are replaced by the address `.text`, which represents the first address in the `.text` section, i.e. the address of the `add` function, and the address `.text+0x18`, which represents the address of the `main` function.

```
Disassembly of section .eh_frame:

0000000000000000 <.eh_frame>:
   0: 14 00                   adc    $0x0,%al
   2: 00 00                   add    %al,(%rax)
   4: 00 00                   add    %al,(%rax)
   6: 00 00                   add    %al,(%rax)
   8: 01 7a 52                add    %edi,0x52(%rdx)
   b: 00 01                   add    %al,(%rcx)
   d: 78 10                   js     1f <.eh_frame+0x1f>
   f: 01 1b                   add    %ebx,(%rbx)
  11: 0c 07                   or     $0x7,%al
  13: 08 90 01 00 00 1c       or     %dl,0x1c000001(%rax)
  19: 00 00                   add    %al,(%rax)
  1b: 00 1c 00                add    %bl,(%rax,%rax,1)
  1e: 00 00                   add    %al,(%rax)
  20: 00 00                   add    %al,(%rax)     <-- here for 2nd entry
  22: 00 00                   add    %al,(%rax)
  24: 18 00                   sbb    %al,(%rax)
  26: 00 00                   add    %al,(%rax)
  28: 00 45 0e                add    %al,0xe(%rbp)
  2b: 10 86 02 43 0d 06       adc    %al,0x60d4302(%rsi)
  31: 4f 0c 07                rex.WRXB or $0x7,%al
  34: 08 00                   or     %al,(%rax)
  36: 00 00                   add    %al,(%rax)
  38: 1c 00                   sbb    $0x0,%al
  3a: 00 00                   add    %al,(%rax)
  3c: 3c 00                   cmp    $0x0,%al
  3e: 00 00                   add    %al,(%rax)
  40: 00 00                   add    %al,(%rax)     <-- here for 3rd entry
  42: 00 00                   add    %al,(%rax)
  44: 33 00                   xor    (%rax),%eax
```

Therefore, we can see that the object file generates a “skeleton” code that contains all the instructions, with some dummy addresses that need to be replaced. The relocation table T tells us exactly where these dummy addresses are in the code and what they need to be replaced with. Therefore, if we want to call a function `printf` that is in the text section at address `0x30`, then we can actually look at the value at `T[30]` to see where the actual address is. At this point, note that we still do not know the actual memory address of `add`. This is determined by the linker.

9.5.2 Linking with One Object File

Now let's see what happens once we link the object file `main.o` into the final executable `main`. If we disassemble it, then we can see a few things:

1. The addresses of all the functions have been changed. `add` starts on address `0x1129` rather than `0x0` and `main` starts on address `0x1141` rather than `0x18`.
2. The dummy address `0x0` of the call to function `add` in `main` have been replaced with the actual addresses `0x1129`.

```
0000000000001129 <add>:
  1129:  f3 0f 1e fa             endbr64
  112d:  55                      push   %rbp
  112e:  48 89 e5                mov    %rsp,%rbp
  1131:  89 7d fc                mov    %edi,-0x4(%rbp)
  1134:  89 75 f8                mov    %esi,-0x8(%rbp)
  1137:  8b 55 fc                mov    -0x4(%rbp),%edx
  113a:  8b 45 f8                mov    -0x8(%rbp),%eax
  113d:  01 d0                   add    %edx,%eax
  113f:  5d                      pop    %rbp
  1140:  c3                      retq

0000000000001141 <main>:
  1141:  f3 0f 1e fa             endbr64
  1145:  55                      push   %rbp
  1146:  48 89 e5                mov    %rsp,%rbp
  1149:  48 83 ec 10             sub    $0x10,%rsp
  114d:  c7 45 f4 03 00 00 00    movl   $0x3,-0xc(%rbp)
  1154:  c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
  115b:  8b 55 f8                mov    -0x8(%rbp),%edx
  115e:  8b 45 f4                mov    -0xc(%rbp),%eax
  1161:  89 d6                   mov    %edx,%esi
  1163:  89 c7                   mov    %eax,%edi
  1165:  e8 bf ff ff ff          callq  1129 <add>     <-- replaced with actual address
  116a:  89 45 fc                mov    %eax,-0x4(%rbp)
  116d:  b8 00 00 00 00          mov    $0x0,%eax
  1172:  c9                      leaveq
  1173:  c3                      retq
  1174:  66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  117b:  00 00 00
  117e:  66 90                   xchg   %ax,%ax
```

9.5.3 Global vs External Symbols

So far, we have talked about using the `#include` as a precompiling command that says "put all the text from this other file right here." Take the following code for instance.

```c
// file1.c
#include "sum.h"

int array[2] = {1, 2};

int main() {
  int val = sum(array, 2);
  return val;
}
```

```c
// sum.h
int sum(int *a, int n) {
  int i, s = 0;
  for (i = 0; i < n; i++) {
    s += a[i];
  }
  return s;
}
.
```

Figure 7: Including a header file in `file1.c` to import functions and variables.

However, there is another way to do this. We can use *external symbols* to access. Rather than simply copying and pasting the code into the file, the `extern` keyword marks that the variable or function exists externally to this source file and does not allocate storage for it.

```c
// main.c
extern int sum(int *array, int n);

int array[2] = {1, 2};

int main(void) {
  int val = sum(array, 2);
  return val;
}
```

```c
// sum.c
int sum(int *array, int n) {
  int i, s = 0 ;
  for (int i = 0; i < n; i++) {
      s += array[i];
    }
  return s;
}
.
```

Figure 8: Using external symbols to access functions and variables.

One is not a replacement for the other, so what advantage does this have? Well, as we will see, if we have multiple object (source) files, say `A.c`, `B.c`, and `C.c`, that need to reference the same function or variable `var` in `ext.c`, then how would we do this? If we simply put `#include "ext.h"` in all the files, then we would have multiple copies of the same code. This means that for each source there would be its own copy of `var` created and the linker would be unable to resolve this symbol. However, if we put `extern int var;` at the top of each source file, then only one copy of `var` would be created (in `ext.c`), which creates a single instance of `var` for the linker to resolve. [3]

Therefore, there are three types of symbols (variables, functions, etc.) that we need to consider:

1. **Global symbols** that are defined in the global scope of a C file.
2. **Local symbols** that are defined in the local scope of a C file, e.g. within functions, loops, etc.
3. **External symbols** that are defined in another C file referenced by the `extern` keyword.

Linkers will only know about global and external symbols, and will have no idea that any local symbols exist. With the information of these two types of symbols and the relocation tables of each object file, the linker can then resolve the addresses of all the symbols in the final binary.

The two types of symbols that the linker will know about are the global and external symbols. We can see that external symbols can be problematic if the object files don't know about each other.

[3] https://stackoverflow.com/questions/1330114/whats-the-difference-between-using-extern-and-including-header-files

Example 9.6 (Global and Local Symbols)

Consider the following code where the left file includes the right file.

```c
// main.c
#include "sum.h"

int array[2] = {1, 2};

int main() {
  int val = sum(array, 2);
  return val;
}
```

```c
// sum.h
int sum(int *a, int n) {
  int i, s = 0;
  for (i = 0; i < n; i++) {
    s += a[i];
  }
  return s;
}
.
```

In the left file,

1. We define the global symbol `main()`.
2. Inside main, `val` is a local symbol so the linker knows nothing about it.
3. The `sum` function is an external symbol, and it references a global symbol that's defined in `sum` the right file.
4. The `array` is a global symbol that is defined in the right file.

In the right file, the linker knows nothing of the local symbols `i` or `s`.

9.5.4 Linking with Multiple Object Files

We have seen the case of linking when we simply have one object file. The relocation was simple since the `.text` section is contiguous and so we needed simple translations of addresses to relocate `add` and `main`, along with whatever other sections and files. Now let's consider the case where we have multiple object files.

```c
// main.c
extern int sum(int *array, int n);

int array[2] = {1, 2};

int main(void) {
  int val = sum(array, 2);
  return val;
}
```

```c
// sum.c
int sum(int *array, int n) {
  int i, s = 0 ;
  for (int i = 0; i < n; i++) {
      s += array[i];
    }
  return s;
}
.
```

Now they have their own object files shown below, where I also put the source code lines to make it easier to parse. Note that again, in `main.o` the call to function `sum` is a dummy address that needs to be replaced. Furthermore, in both `main.o` and `sum.o`, the `.text` section is at address `0x0`, where the addresses of the function `main` and `sum` are, respectively. This causes an overload in the address space.

To demonstrate what happens, we look at how the disassembly, symbol tables, and relocation tables are updated before (with the object files) and after (in the binary) linking.

Example 9.7 (Disassembly of Object Files)

In here, note that both the `array` and `sum` are not initialized and are therefore set to dummy addresses.

```
main.o:     file format elf64-x86-64
Disassembly of section .text:

0000000000000000 <main>:
extern int sum(int *array, int n);

```

```
int array[2] = {1, 2};

int main(void) {
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 83 ec 10             sub    $0x10,%rsp
  int val = sum(array, 2);
   c: be 02 00 00 00          mov    $0x2,%esi
  11: 48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 18 <main+0x18>  <-- dummy
    address
  18: e8 00 00 00 00          callq  1d <main+0x1d>                          <-- dummy
    address
  1d: 89 45 fc                mov    %eax,-0x4(%rbp)
  return val;
  20: 8b 45 fc                mov    -0x4(%rbp),%eax
}
  23: c9                      leaveq
  24: c3                      retq
```

```
sum.o:     file format elf64-x86-64
Disassembly of section .text:

0000000000000000 <sum>:
int sum(int *array, int n) {
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 89 7d e8             mov    %rdi,-0x18(%rbp)
   c: 89 75 e4                mov    %esi,-0x1c(%rbp)
  int i, s = 0;
   f: c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  for (int i = 0; i < n; i++) {
  16: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  1d: eb 1d                   jmp    3c <sum+0x3c>
    s += array[i];
  1f: 8b 45 fc                mov    -0x4(%rbp),%eax
  22: 48 98                   cltq
  24: 48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  2b: 00
  2c: 48 8b 45 e8             mov    -0x18(%rbp),%rax
  30: 48 01 d0                add    %rdx,%rax
  33: 8b 00                   mov    (%rax),%eax
  35: 01 45 f8                add    %eax,-0x8(%rbp)
  for (int i = 0; i < n; i++) {
  38: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
  3c: 8b 45 fc                mov    -0x4(%rbp),%eax
  3f: 3b 45 e4                cmp    -0x1c(%rbp),%eax
  42: 7c db                   jl     1f <sum+0x1f>
  }
  return s;
  44: 8b 45 f8                mov    -0x8(%rbp),%eax
}
  47: 5d                      pop    %rbp
  48: c3                      retq
```

1. In `main.o` at address `0x0`, we have the `main` function and this is because everything is stored relatively to the start of main. Once we have linked, `main` shows the absolute addresses of all the instructions.
2. In instruction 11 in `main.o` we can see that `48 8d 3d` is the `lea` instruction, which is the same as that in `main`. However, the address that is was acting on is `0x0` since the array has not been initialized yet. We can see in `main` that the address is now `0x00002ecf`.
3. The comment in `main` indicates that the final relocated address used to access the `array` is `0x4010`. To see relocated addresses in general, just look for the comments and shift them accordingly.

```
main:     file format elf64-x86-64

0000000000001129 <main>:
    1129:   f3 0f 1e fa             endbr64
    112d:   55                      push   %rbp
    112e:   48 89 e5                mov    %rsp,%rbp
    1131:   48 83 ec 10             sub    $0x10,%rsp
    1135:   be 02 00 00 00          mov    $0x2,%esi
    113a:   48 8d 3d cf 2e 00 00    lea    0x2ecf(%rip),%rdi        # 4010 <array>
    1141:   e8 08 00 00 00          callq  114e <sum>
    1146:   89 45 fc                mov    %eax,-0x4(%rbp)
    1149:   8b 45 fc                mov    -0x4(%rbp),%eax
    114c:   c9                      leaveq
    114d:   c3                      retq

000000000000114e <sum>:
    114e:   f3 0f 1e fa             endbr64
    1152:   55                      push   %rbp
    1153:   48 89 e5                mov    %rsp,%rbp
    1156:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
    115a:   89 75 e4                mov    %esi,-0x1c(%rbp)
    ...
```

Example 9.8 (Symbol Tables of Object Files)

Let's take a look at the symbol table of each file as well. Again, all of the addresses of each symbol are 0s since they are using relative addressing. The `array` and `main` are global symbols since they reside in the global scope, while the `sum` function is an external and undefined symbol.

```
main.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*   0000000000000000 main.c
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack  0000000000000000 .note.GNU-stack
0000000000000000 l    d  .note.gnu.property  0000000000000000 .note.gnu.property
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     O .data   0000000000000008 array
0000000000000000 g     F .text   0000000000000025 main
0000000000000000         *UND*   0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*   0000000000000000 sum
```

```
sum.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*   0000000000000000 sum.c
0000000000000000 l    d  .text   0000000000000000 .text
0000000000000000 l    d  .data   0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack  0000000000000000 .note.GNU-stack
0000000000000000 l    d  .note.gnu.property  0000000000000000 .note.gnu.property
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     F .text   0000000000000049 sum
```

When we have the linked binary, note a few things.

1. In `main.o`, the numbers on the left represents the address of the symbol (all 0s since we haven't linked yet and their final addresses aren't known), while the addresses in `a.out` are all known.
2. In `main.o`, the `sum` function is an external symbol and is undefined. The linker will need to know where this is. In `main`, note that the `sum` function is now a global symbol and is defined, along with the size. We can now see that all the final addresses of each symbol is known, along with their sizes, and the `UND` marker is now gone as well.
3. Only the size of the global variable is known in `main.o` since we have defined it within the code. However, in `main`, the linker has now assigned an address to it.
4. To see the size in bytes of the array, you can look at the address and how much size it takes up.

```
main:     file format elf64-x86-64

SYMBOL TABLE:
...
0000000000004008 g     O .data     0000000000000000              .hidden __dso_handle
000000000000114e g     F .text     0000000000000049              sum
0000000000002000 g     O .rodata   0000000000000004              _IO_stdin_used
00000000000011a0 g     F .text     0000000000000065              __libc_csu_init
0000000000004020 g       .bss      0000000000000000              _end
0000000000001040 g     F .text     000000000000002f              _start
0000000000004018 g       .bss      0000000000000000              __bss_start
0000000000001129 g     F .text     0000000000000025              main
0000000000004018 g     O .data     0000000000000000              .hidden __TMC_END__
...
```

Example 9.9 (Relocation Tables)

Ignoring the `.eh_frame`, in `main.o` the relocation table contains entries for `array` and `sum` that must be relocated.

```
main.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000014 R_X86_64_PC32     array-0x0000000000000004
0000000000000019 R_X86_64_PLT32    sum-0x0000000000000004

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

```
sum.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

We can see a couple things. Namely, there is nothing to be relocated in `a.out` since everything has been relocated already by the linker. So let's focus on the relocation for `main.o`. In here, we can see that in the `.text` section, there are two things being relocated:

1. The reference to the global variable `array` is being relocated. In this object file, we look at the offset `0x14` from the beginning of the `.text` section, which contains the instruction that needs to access `array`. This relocation record tells the linker to calculate the 32-bit offset from the instruction (at offset `0x14`) to the start of `array`, then adjust it by subtracting 4 bytes.
2. The reference to the `sum` function is being relocated. In this object file, we look at the offset `0x19` from the beginning of the `.text` section, which contains the instruction that needs to access `sum`. This relocation record tells the linker to calculate the 32-bit offset from the instruction (at offset `0x19`) to the start of the `.plt` section, then adjust it by subtracting 4 bytes.

```
main:     file format elf64-x86-64
```

9.6 Compiler Optimization

We have learned the complete process of compilers, but compilers can be a little smarter than just translating code line by line. They also come with flags that can optimize the code.

Definition 9.9 (gcc Optimization)

The gcc compiler can optimize the code with the `-O` flag. To run level 1 optimization, we can write

```
gcc -O1 -o main main.c
```

The level of optimizations are listed:

1. Level 1 perform basic optimizations to reduce code size and execution time while attempting to keep compile time to a minimum.
2. Level 2 optimizations include most of GCC's implemented optimizations that do not involve a space-performance trade-off.
3. Level 3 performs additional optimizations (such as function inlining) and may cause the program to take significantly longer to compile.

Let's see what common implementation are.

Definition 9.10 (Constant Folding)

Constants in the code are evaluated at compile time to reduce the number of resulting instructions. For example, in the code snippet that follows, macro expansion replaces the statement `int debug = N-5` with `int debug = 5-5`. Constant folding then updates this statement to `int debug = 0`.

```c
#define N 5
int debug = N - 5; //constant folding changes this statement to debug = 0;
```

Definition 9.11 (Constant Propagation)

Constant propagation replaces variables with a constant value if such a value is known at compile time. Consider the following code segment, where the `if (debug)` statement is replaced with `if (0)`.

```c
int debug = 0;

int doubleSum(int *array, int length){
    int i, total = 0;
    for (i = 0; i < length; i++){
        total += array[i];
        if (debug) {
            printf("array[%d] is: %d\n", i, array[i]);
        }
    }
    return 2 * total;
}
```

Definition 9.12 (Dead Code Elimination)

Dead code elimination removes code that is never executed. For example, in the code snippet that follows, the `if (debug)` statement and its body is removed since the value of `debug` is known to be 0.

```c
int debug = 0;

int doubleSum(int *array, int length){
    int i, total = 0;
    for (i = 0; i < length; i++){
        total += array[i];
        if (debug) {                                    // remove
            printf("array[%d] is: %d\n", i, array[i]);    // remove
        }                                               // remove
    }
    return 2 * total;
}
```

Definition 9.13 (Simplifying Expressions)

Some instructions are more expensive than others, so things like

1. `2 * total` may be replaced with `total + total` because addition instruction is less expensive than multiplication.
2. `total * 8` may be replaced with `total « 3`
3. `total % 8` may be replaced with `total & 7`

Note that these optimization techniques are in no way a guarantee that the code will run faster since there are many factors and always edge cases (for example, maybe some localities are lost). Furthermore, compiler optimization will never be able to improve runtime complexity (e.g. by replacing bubble sort with quicksort).