An Overview
of
Object Files
And
Linking

Harry H. Porter III

Portland State University
cs.pdx.edu/~harry

April 9, 2015

Goals of this Paper

Audience:
- CS-201 students

Coverage:
- Compiling
- Linking
- Object files
- External Symbols
- Relocatable Code
- Segments (.text, .data., .bss)

Overview

In order to create an executable file, a program must be compiled and linked. In this paper, I'll use the "C" programming language as an example, but this discussion applies to other compiled languages like C++. Languages that are interpreted (like Java or Python) do things differently and linking does not apply to them. We'll also discuss the Linux/Unix system, but other OSes are similar.

A program begins as a human readable source text file, such as "**hello.c**". The program must first be compiled and this step produces a human readable text file in assembly code.

Then the assembly code version is assembled and this produces an object file. Finally, the object file is linked and the executable file is produced. At some later time, the OS will load the executable file into memory run it.

Many programs are large and these programs are broken into several "**.c**" files. We'll look at an example involving two files, "**hello.c**" and "**there.c**". Each of the **.c** files must be compiled and assembled, but these steps can be done independently. In other words, we can compile and assemble **hello.c** before we even create the **there.c** file.

After both **hello.c** and **there.c** have be compiled and assembled, we use the linker to combine the two pieces and produce the executable file. The linking step will combine the two object files (**hello.o** and **there.o**) to produce the executable file.

Programs generally use library functions that were written long ago by someone else. For example, the program might call "**printf**" to perform output and might call "**cos**" to compute the mathematical cosine function. These functions were compiled and assembled previously. When linking the **hello** program, the linker must locate the code for **printf** and **cos** and include it in the executable file it creates.

The previously compiled code for functions like **printf**, **cos**, and hundreds more is kept in libraries. Each library will contain many (perhaps several hundred) functions. The linker will include only the functions your program actually uses.

The Assembly Code File

The output of the compiler is assembly code. This output is human readable and can be saved in a file. The "**.s**" extension is used for assembly code. Normally, the name of an assembly file will be the same as the "**.c**" file that it came from.

Each machine has its own assembly language, but they are all quite similar. For example, each type of computer will have an ADD instruction, but they differ in details. Here are some examples:

```
IA32
     addl    %ebx,%eax
X86-64:
     addq    %rbx,%rax
BLITZ
     add     r3,r5,r8
Arduino
     add     r18,r19
ARM
     add     r3,r5,r8
```

Assembly code files are normally produced by the compiler but you can also create your own **.s** file from scratch. Not many programs are written in assembly code any more. The assembly code created by the compiler is human readable… sort of. It is computer-generated and you need to know assembly language to make any sense out of it. But it is a text file that can be viewed.

Typically, we do not produce the **.s** file. Instead, the assembly language version of the program is sent straight from the compiler to the assembler. Thus, the assembly code exists only for a short time during the compiling process and is not normally saved.

However, we might create a **.s** file in order to look at the output of the compiler, which might be necessary when optimizing a program or trying to find a bug in the "C" code or even in the compiler itself. We also look at the assembly code to understand what the compiler is doing and how the whole thing works.

The Object File

The output of the assembler is placed into an "object file". An extension of "**.o**" is used for object files. Object files primarily contain the machine code. In other words, they contain the binary representations of the instructions. The object file also contains data,

such as strings and integer values that appear in the program. Together these are the bytes that will be loaded into memory when the program is executed.

Object files contain bytes that are interpreted by the processor as instructions. Therefore, the files are not text files. And they are not human readable, unless you can make sense of very long sequences of binary numbers.

A Unix/Linux command called **objdump** can be used to pick apart an object file. You might type this to see what's in an object or executable file:

```
objdump -xd hello.o
```

In addition to instructions and data bytes, the object file also contains some additional information that the linker needs.

Normally, the name of an object file will be the same as the **.s** file that it came from. So

hello.c → hello.s → hello.o

The Executable File

The linker will combine several **.o** object files to produce an executable file. The default name of the executable file is "**a.out**" but this name is normally overridden. In our example, we'll call the executable file "**hello**" (executable files rarely have extensions).

The format of the executable file is quite similar to the object files. In particular, the executable file contains a lot of bytes which will be loaded into memory when the program is executed. These bytes, which come from the object files, each represent the machine instructions and data values.

There is also additional material in the executable file. This material can be used by the debugger at runtime.

Sometimes you'll hear these terms:

"Relocatable Object File"
(means "object file")
"Executable Object File"
(means "executable file")

The Object Files Must Be Linked

The linker is a program named "**ld**". The linker takes as input several object files and produces a single executable file as its output. The linking step essentially glues the object files together to produce the executable.

Linking is necessary because instructions in one file can use symbols that are defined in other files. For example, some code in **hello.c** can call a function defined in file **there.c**. When assembling **hello.c**, the assembler has no idea where the function is, so the assembler cannot complete building the CALL instruction.

As another example, some code in **hello.c** might make use of a variable defined in a different file. The assembler must create instructions to read or write that variable, but the assembler does not know where that variable will be located in memory.

The linking phase will patch up and complete these instructions. The linker will decide where the routines are to be placed in memory. After this, the linker can fill in the missing address in the call instruction. Likewise, it will decide where the variables are to be placed. Then later the linker can fill in the missing addresses in the instructions that read and write variables.

The "gcc" Command

The **gcc** command can be used to preprocess, compile, assemble, and link files. It will invoke the preprocessor, the C compiler, the assembler, and the linker tools as necessary. It can do some or all of these tasks.

preprocessor → compiler → assembler → linker

The **gcc** command is a compiler driver script and invokes other Unix tools. The program names of these tools are "**cpp**", "**cc1**", "**as**", and "**ld**". We can summarize like this:

cpp → cc1[1] → as → ld

Recall the normal extensions of the files:

"C" source code:

[1] Some systems name the compiler **cc1** and others name it **cc**.

hello.c

After preprocessing

hello.i

Assembly Language

hello.s

Object file

hello.o

Executable

hello

So here is what **gcc** can do:

`hello.c` → *preprocessor* → **`hello.i`** → *compiler* →
`hello.s` → *assembler* → **`hello.o`** → *linker* → **`hello`**

Normally the input is a source file and the output is an executable file, but **gcc** can start anywhere in this sequence and it can end anywhere.

The **gcc** command will take one or more input files so the command will have several file names. The **gcc** command will look at the extensions on these names and from that deduce the sort of file it is. This will tell **gcc** where in the process to begin. For example, **gcc** is run on a file named **hello.s**, it will skip the preprocess and compile steps and begin by assembling the file.

You can tell **gcc** where to stop by using command line arguments.

`-E` stop after preprocessing and produce a **.i** file
`-S` stop after compiling and produce an assembly file
`-c` stop after assembly and produce an object file

The output file will be given the default name. The following option can be used to force **gcc** to name the output file what you want:

`-o` *<name>*

For example:

```
gcc hello.c -o hello
```

If the output is an executable file, it is typical to rename the output file, since the name "**a.out**" is not particularly meaningful. For **.i**, **.s**, and **.o** output, the default name is usually just fine and you won't need to override it.

Other Useful Options to gcc

One option will cause the C compiler to print warnings for code that is questionable. You should always use this option and fix your code so that no warnings are printed:

```
gcc hello.c –Wall –o hello
```

The "**-O**" option is for optimization. This is the capital letter "oh", not "zero". This option is followed by a digit telling how much optimization the compiler should perform.

More optimization may make the program run faster, but may also change the program's behavior in subtle ways.

-O0 No optimization. This is the default
-O1 Some optimization.
-O2 Even more optimization.
-O3 The highest amount of optimization

-Og Do some optimization, but not enough to cause confusion during gdb

I recommend using **-O1**.

```
gcc hello.c -Wall -O1 -o hello
```

The **-g** option will cause the linker to include information in the executable file that can be used by the debugger at run time. Include options **-g –Og** if you intend to use **gdb** or a similar debugger. If you run into a bug in your code and want to use **gdb** to find it, you may need to recompile your program with these options.

If you are planning to use **gdb**, you need to avoid optimization. Some optimization will cause the compiler to change your code around in unexpected ways. Since the correspondence between your original code and executable code can be complex, it may make it difficult to understand what is happening when you use **gdb** to look closely at the code.

The **-l** option (that is "el" not "one") tells which libraries to use during linking. The option should be followed immediately by some characters, like this:

```
gcc hello.c -lxyz
```

The characters "**xyz**" will have "**lib**" added to the front and "**.a**" added to the end. Thus, the linker will use the library named "**libxyz.a**".

There are two important libraries:

`libc.a` contains lots of necessary functions, such as **printf**
`libm.a` contains math-related functions, like **cos**.

With all "C" programs, the first library is necessary. The option to use the first library would be **`-lc`**, but since it is assumed by default, you don't need to specify it.

If your program uses a function from the math library, then you'll need to use this option:

```
gcc hello.c -lm
```

The linker processes libraries and **`.o`** files in order they appear on the command line. This can sometimes cause a frustrating problem. The best practice is always to place the **`-l`** options before the **`.o`** files on the **gcc** command line.

The assembler is capable of producing a listing showing what machine instructions (i.e., the actual bytes) it produced for each line of the assembly code program.

`-Wa,-al` Produce an assembler listing

(There is no space before or after the comma.)

There are lots of other options to **gcc**.

Memory at Runtime

To run a program, the OS will load the program from a file into memory. The executable file contains the bytes that represent instructions and data. These will be copied into the virtual address space. Then the OS will create a thread and begin executing instructions within that new address space.

The object file contains other material, besides the instruction and data bytes. This extra material includes such information as:

- The number of bytes of instructions

- The number of bytes of data
- Where in memory to put the bytes
- Where to begin execution (i.e., the address to jump to)
- Whether this file is an "executable" or "relocatable" object file
- Symbol table:
  - Which symbols are defined in this file
  - What is the numerical value of each symbol
- Relocation records:
  - Which symbols are used in this file, but not defined
  - Which bytes need to be modified when we know their values
- Additional info for **gdb** (optional):
  - What are the names of the variables
  - What type is each variable (char, int, pointer, double, etc.)
  - Where in memory is each variable stored
  - What are the names of the functions
  - Where is each function in memory

The symbol table and the relocation records are included in relocatable object files. These are used by the linker when it combines the object files. Once the linking is complete, the symbol table and relocation records are not needed. (At least with static linking; dynamic linking is more complex.)

The additional information for **gdb** is ignored when the file is loaded for execution. It is only used when **gdb** is running the program. In this case, **gdb** will retrieve this information and use it while you are running and try to debug the executable.

The virtual address space is divided into "pages". Each page is 4K bytes, at least for most versions of Linux/Unix. The memory consists of a number of pages, one after the other.

The OS will mark each page in a virtual address space as either:
- Read-only
- Read/Write
- Invalid

If the program tries to read a byte from a page that is marked "read-only" or "read/write" there is no problem.

If the program tries to write data to a byte in a page marked "read/write" there is no problem. But if the program tries to write into a page marked "read-only" the OS will

detect the violation and the program will be terminated. (More precisely, the OS will raise the seg-fault signal, SIGSEGV.)

And if the program tries to read or write a byte within a page that is marked "invalid", the OS will terminate the program. (That is, the OS will send the SIGSEGV signal to indicate a segmentation error which normally causes the program to be terminated.)

A running program will contain some material that can be placed in read-only pages. For example, the code itself will never be modified and can go into read-only pages. Also, the program might contain some fixed constants, such as string constants, that cannot be changed, so these too can go into read-only pages.

The program will also contain variables whose values will change during the program's execution. These variables must go into read-write pages.

The program will also use a runtime stack. Every time a function is called, a stack frame (also called an "activation record") will be pushed onto the stack. The stack frame will contain the function's local variables and the return address. When the function returns, the stack frame will be popped off the top of the stack, effectively destroying the functions local variables. The stack must also be placed in pages marked read-write.

These three areas of a program's address space are called "segments." They are

.text segment	Read-only	Program instructions and constant data
.data segment	Read-write	Variables and data that may be updated
.stack segment	Read-write	The stack of activation records

A segment consists of one or more pages.

Most programs have very few variables and their **.data** segments will often be as small as one page. But a large program might contain huge arrays and so its **.data** segment would contain many pages.

The executable file contains no information about the program's **.stack** segment since the stack is non-existent (i.e., empty) before the program begins execution. You may have been told that every program begins execution by executing the **main()** function. In fact, the linker always quietly adds some additional functions to your program. One of these functions will actually be the first code executed. This function will initialize the stack and then it will subsequently call your **main()** function.

In any case, a program's **.stack** segment starts small, generally fitting within a single page, but the **.stack** segment may need to grow as the program calls more functions and pushes frames onto the stack.

The **.text** segment does not change size at runtime and the **.data** segment does not normally change size at runtime. However, some programs use a heap and will call **malloc()** or **calloc()** to get more memory. Such programs may need additional **.data** pages at runtime. The code within **malloc()** will ask the OS to add more pages to the **.data** segment when needed.

Therefore, during the program's execution, additional pages may be added to the virtual address space. (This can also occur as a result of calls to the OS explicitly made by your code, in addition to the calls made be **malloc()**.)

From the executing program's point of view, the OS merely changes some pages from "invalid" to "read-only" (or to "read-write"). However, the OS will also need to allocate physical memory frames and map these newly accessible pages into physical memory locations.

For example, if the **.stack** segment is about to grow by one page, the OS will need to find a block of 4K bytes of physical memory to use. The OS will change the page table for the executing program. The new page will be located somewhere in the virtual address space. A new **.stack** page will be located directly before the lowest page currently in the **.stack** segment (since the stack grows downward). So the page table entry for this page will be altered to make the page read-write. Also, the entry will be changed to point to a block of 4K bytes of physical memory. As the stack grows into the new page, this is where the data will be stored when new stack frames are pushed onto the stack.

There is also a segment called "**.bss**". Sometimes programs contain variables that are uninitialized. These variables have no values before the program begins execution, so there is no reason to waste space in the object files or in the executable files.

In my opinion, a program should never have any uninitialized variables. In order to reduce bugs and write more reliable code, you should get into the habit of initializing all variables at the point where they are declared. But realistically, this does not happen and almost every C program contains uninitialized variables.

A typical program will have only a few uninitialized variables so there is really very little wasted space and the waste is insignificant. However, some programs contain huge arrays — say, of 1 gigabyte in size — and the potential waste is significant. The executable files

would need to contain these initial values; at execution time, the OS would need to load all those bytes into memory; and when linking the object files, the linker would need to move and copy all those bytes. The idea with the **.bss** segment is to avoid this wasted time and space.

The **.bss** segment is for uninitialized variables. This segment is a group of bytes, without values. Within the object and executable files, the **.bss** segment takes almost no space. Essentially, all we need is a single number telling how many bytes are required. When loading, the OS will create pages for this segment but will not waste any time moving bytes from disk into memory.

Each object file contains three segments: **.text**, **.data**, and **.bss**. Each executable file also contains these three segments. These files contain some other material, but the main thing they contain is these segments.

For the **.text** and **.data** segments, the object file contains the bytes that will go into memory. For the **.bss** segment, the object file contains only the length.

Within the object file, there is no information about where the bytes must be placed in memory. The linker will determine where in the address space the segments are to be placed. Within the executable file, there will be information about where each segment must be placed. So the executable file will also contain a "starting address" for each of the segments.

The Main Goal of the Linker

When linking together several object files in order to produce an executable file, the linker must take the **.text** segment from each of the object files and concatenate them to form a single **.text** segment. The linker appends all the **.text** segments from the input object files, building a single large **.text** segment, which it then places in the output, the executable file it is creating.

Likewise, the linker takes a .**data** segment from each of the object files and appends them together to create one big **.data** segment. This combined **.data** segment is then placed in the executable file.

Similarly, for the **.bss** segments. Since there are no data bytes for the **.bss** segments, all the linker really does is just add their lengths together to compute the length of the combined **.bss** segment.[2]

The linker must also determine where in memory the three segments will go. The linker will place the **.text** segment at some fixed, predetermined starting addresses. All programs generally begin at the same address so this is not something that changes. The pages for the **.text** segment will be marked read-only when the OS loads the **.text** segment.

The linker will place the **.data** segment directly following (i.e., in higher addresses) the **.text** segment. More precisely, the linker will round up to the next page boundary and then place the **.data** segment at that address.

Finally, the linker will place the **.bss** segment after the **.data** segment.

The linker determines where the segments are to be placed in memory and adds these "starting addresses" to the executable file. Then, at execution time, the OS will copy the data bytes from the executable file into the new address space, placing them at whatever starting addresses that the linker has determined.

There is a special function is named "**_start**" which will be executed first, before **main()** is called. The **_start()** function is included by the linker in every executable file regardless of whether the program was written in "C", "C++" or whatever.

The executable file contains some header information which includes a number called the "starting address". The starting address is where execution will begin. The starting address is the exact same address at which the **_start()** function is placed. Execution begins in **_start()**.

After the OS loads the segments into virtual memory pages, it will mark the pages containing the **.text** segment "read-only" and the pages containing the **.data** and **.bss**

---

[2] The virtual address space contains several "segments". To be more precise in our terminology, we should say that an executable file contains *images* of these segments which will be loaded at runtime into the corresponding memory segments. Each object file contains three "sections" called **.text**, **.data**, and **.bss**. Since each section is only a piece or a chunk of the entire segment, we should probably not call it a segment. As we will see later, the linker will concatenate the **.text** "sections" from several relocatable object files to produce the single **.text** "segment" that will exist at runtime. Similarly for the **.data** and **.bss** segments. I also use the term "chunk" to mean "section", since "section" sounds too much like "segment" and might cause confusion.

segments "read-write". The OS will also allocate a page or two for the **.stack** segment. The OS will mark all other pages as "invalid" and will begin the program's "thread of execution" by jumping to the starting address in the **.text** segment.

At this moment the process begins. The first instruction executed will be the instruction located at the "starting address". This is address of the first byte of the **_start()** function. The **_start()** function does a number of things (such as dealing with dynamically loaded libraries and calling initialization code for C++ programs). But finally **_start()** will call your **main()** function and, if **main()** returns, it will then call **exit()**. Whatever value your **main()** function returns will get passed to **exit()**.[3] Here is the relevant line of code:

```
exit(main(argc, argv, envp));
```

The **_start()** function will never return; it can't return because it was not called like a normal function. The OS just began executing instructions at the beginning of the **_start()** function. But that's okay because the exit function will never return.

Global and Local Variables

Here is some "C" code:

foo.c:

```
int x = 123;
int y;
int foo (int i) {
    int j;
    ...
}
```

The variables **i** and **j** are local to the function **foo**. The variable **i** is a parameter to **foo**, while **j** is just a plain-old local variable.

Local variables do not take up memory space until the function is actually called. If the function is never called, the local variables will never occupy memory. If the function is called recursively, there will be a copy of **i** and **j** for each invocation that is active and has not yet returned.

[3] This discussion is simplified and some details are left out.

Local variables are allocated on the stack. The actual addresses of **i** and **j** will not be determined until the moment **foo** is called. Local variables go into stack frames.

For example, if **foo** calls itself recursively and, at some moment during execution, there are 37 activations of **foo**, then the stack will contain 37 stack frames, one for each currently running invocation of **foo**. Thus, there will be 37 versions of **i** and 37 versions of **j**, and each will probably have a different value. When the most recently invoked invocation of **foo** returns, one stack frame will be popped from the stack and then there will only be 36 copies of **i** and **j**.

The variables **x** and **y** are global variables. Sometimes we use the term "static" variables to mean the same thing. Any variable that is declared outside of a function is global while any variable inside a function is local.[4]

The variables **x** and **y** will occupy memory, even if they are never used. They will be put into the **.data** segment and you can see this if look at the assembly code. (It might be the case that **y** will be put into the **.bss** segment since it is uninitialized; you can determine this by looking at the assembly code generated by the compiler.)

The actual addresses of **x** and **y** will be determined and fixed at link-time when the linker determines where the **.data** and **.bss** segments will be placed. The **.data** and **.bss** segments will presumably contain many other variables, so the exact locations within the segments will depend on what else is there and the order that they happen to be in.

For example, the variable **x** might be located at the address 0x104 bytes from the beginning of the **.data** segment. If the linker places the **.data** segment at address 0x00300000, then **x** will be at address 0x00300104 in the address space.

Instructions That Access Local Variables

Within the function **foo**, we can access local variables.

```
int foo (int i) {
    int j;
```

[4] There is a minor exception. In "C", a variable declared inside a function can be marked with the keyword "static". This will cause the variable to become a global variable. In that case, the variable will go into the **.data** segment and not go into the stack frame.

```
   ...
   j = i + 200;
   ...
}
```

What instructions will the compiler generate to access local variables?

Since these variables are located within the stack frame at the top of the stack, they can be accessed using the stack top pointer. Here are the x86-64 instructions you might see:

```
movl    -20(%rbp), %eax
addl    $200, %eax
movl    %eax, -4(%rbp)
```

The register **%rbp** is pointing to the stack frame so the first instruction moves something from the stack (namely **i**) into register **%eax**. The second instruction adds 200. The last instruction moves the result from register **%eax** back to another location in the stack (namely **j**).

Here is the BLITZ code for the same sequence:

```
load    [r14+8],r1
add     r1,200,r1
store   r1,[r14-12]
```

In the BLITZ instruction sequence, **r14** is being used to address variables in the stack frame.

The important thing to notice is that when local variables are used (either read or written) their actual addresses do not appear in the code. Instead, local variables are always accessed using offsets relative to the stack pointer.

This means the linker does not need to do anything for these instructions; they will work fine regardless of where the code is located and regardless of where in memory the stack frames happen to be when the instructions are executed.

Instructions that Access Global Variables

Next, let's look at some code that accesses global (i.e., static) variables.

```
int x = 123;
int y;
int foo (…) {
   …
   y = x + 400;
}
```

Here are the x86-64 instructions for this assignment statement. We also show the 17 bytes produced by the assembler for these 3 instructions.

Machine Code	**Assembler Code**
8B0500000000	movl x(%rip), %eax
0590010000	addl $400, %eax
890500000000	movl %eax, y(%rip)

First look at the add instruction. Performing a little arithmetic, we see that 400 (decimal) = 0x190 (hex). The Intel machine is Little Endian; notice that

`00000190` (Big Endian) = `90010000` (Little Endian)

The op-code for the **addl** instruction is 05. Putting this together, we see that the **addl** instruction assembles to:

```
05 90010000
```

Next look at the first instruction, which loads a 32-bit word from memory to register **%eax**. The op-code is

```
8B05
```

The instruction uses relative addressing. In other words, at runtime an offset will be added to the current instruction pointer (the register **%rip**) to yield the address of **x**. This is the meaning of:

```
x(%rip)
```

However, if we look at the bytes corresponding to the offset, we see

```
00000000
```

Likewise, the instruction to store from register **%eax** into memory has a meaningful op-code

```
8905
```

but the offset for **y** is also

```
00000000
```

The reason is that the assembler does not know where in memory **x** and **y** will be placed.

Here are the instructions for the BLITZ machine. The BLITZ code sequence is longer, in part because it takes two instructions to get a word from memory. The **set** instruction moves an address into a register and then a separate **load** instruction goes to memory to fetch the word.

Machine Code	**Assembler Code**
c0100000	set x,r1
c1100000	
6b110000	load [r1],r1
80110190	add r1,400,r1
c0200000	set y,r2
c1200004	
6f120000	store r1,[r2]

The Intel sequence required 17 bytes and the BLITZ machine required 28 bytes.

The Intel machine uses relative addressing to access **x** and **y** byte specifying offsets from the current instruction pointer. The BLITZ machine uses absolute addressing.

But like the Intel code, the BLITZ assembler is unable to fill in the address. The set instruction is unfinished. Below we have replaced the 32-bit address of **x** with "---- ----" to show where the address will go.

Machine Code	**Assembler Code**
c010----	set x,r1
c110----	

It is the job of the linker to determine where to place variables **x** and **y** and then to go back to the instructions that are incomplete. The linker must update the instructions that are incomplete, filling in the missing values.

In the case of the BLITZ machine, the linker must move the correct address of **x** into the first **set** instruction.

In the case of the Intel machine, the linker must first compute an offset (by subtracting the address of the **movl** instruction from the address of **x**) and then move the resulting offset into the **movl** instruction.

Looking next at the variable **y**, we see that the BLITZ assembler produces this code:

Machine Code	**Assembler Code**
c0200000	set y,r2
c1200004	

The assembler has placed the value 00000004 in the code. Why is it not zero? We just said that the assembler does not know where **x** or **y** will go. However, the assembler builds a piece or chunk of the **.data** segment. Each object file will contribute a chunk of the **.data** segment and the linker will combine all these chunks together to produce the final **.data** segment. Within the **.data** segment chunk contributed by this object file, there will be two variables, **x** and **y**. The assembler has placed **x** at the beginning of this chunk of the **.data** segment and has placed **y** directly after **x**. Since these variables are 4 bytes in length, the relative offsets are:

```
x    0
y    4
```

The linker will relocate this chunk of the **.data** segment when it combines it with other chunks. Thus, the addresses of **x** and **y** will need adjusting. So when the linker knows where this chunk of the **.data** segment will be located, it can adjust the actual addresses of **x** and **y** by adding some fixed value to both.

An Example Involving Two Object Files

Consider these two input files:

hello.c

```
int x = 123;
int main (…) {
        …
}
```

there.c

```
extern x;
int foo (…) {
        y = x + 123;
        …
}
```

The **hello.c** file defines a variable **x** and gives it a value. The **there.c** file uses that variable. The "extern" keyword in C lets the compiler know that the variable **x** exists and tells the compiler what the type of **x** is. This information allows the compiler to generate the correct code when **x** is used within **there.c**.

However, when **there.c** is compiled and assembled, there will be no space allocated for the variable. It will not be until linking that the linker can determine where **x** was declared.

Here is the assembly code and what bytes the assembler produces for the variable **x** from the file **hello.c**:

Machine Code	Assembler Code	
	.data	
	.globl	x
	.align	4
	.type	x,@object
	.size	x,4
	x:	
7B000000	.long	123

The first line

```
        .data
```

indicates that everything that follows should be placed in the **.data** segment.

The second line

```
                    .globl   x
```

tells the linker that the symbol **x** is defined in this file and should be made available to other object files.

The next line

```
                    .align   4
```

will ensure that the next thing is aligned to a 4 byte address boundary.

The next line

```
                    .type    x,@object
```

indicates that **x** is a data object. This directive would contain "`@function`" if **x** were the name of a function instead of a variable.

The next line

```
                    .size    x,4
```

tell the linker the size of the variable **x**.

Finally, we have the last two lines

```
                x:
7B000000            .long    123
```

The pseudo-op "**.long**" means "32-bit constant value". Note that

123 (decimal) = 0x 0000007B (hex)

The label "**x:**" indicates that the address of the next thing that follows will be an address known as **x**. In other words, the symbol **x** is defined at that place as having that address.

It is important to note that **x** is a "C" variable that will have some value at runtime. This value will be held in memory. The value is not placed in memory until runtime and of course the value of variable **x** will change during runtime.

At the same time, the *symbol* **x** is something a little different. A symbol also has a value, but this value is known before runtime. The value of the symbol **x** is the address of the variable **x** and this is value will not change. In some cases, the value of a symbol might be known to the assembler during assembly but in many cases the value of a symbol can not be determined until link time. For example, the assembler does not know exactly where variable **x** will be placed in memory, so the value of the symbol **x** (i.e., the address of variable **x**) will have to be determined by the linker.

For the file **there.c**, the assembler simply uses the symbol **x** in the instructions it generates. There are no additional assembler directives. Since symbol **x** is not defined in the file, it is assumed to be defined in some other file, but the assembler has no information about how **x** is defined or even which file it is defined in.

Segment Chunks

Each object file provide a chunk of the **.text** segment, a chunk of the **.data** segment and a chunk of the **.bss** segment.[5]

A chunk is just a bunch of bytes. In the case of the **.text** and **.data** chunks, the object file contains the actual byte values. In the case of the **.bss** chunk, the object file just has the length, with no actual data.

The linker takes several object files as input. For example, imagine that we are linking together 7 object files to produce an executable. Thus, the linker will have 7 **.text** chunks, 7 **.data** chunks, and 7 **.bss** chunks coming in.

The linker will combine all 7 **.text** chunks, concatenating them one after the other to create the final **.text** segment. Likewise, the linker will concatenate all 7 **.data** chunks to produce the final **.data** segment. And it will concatenate all 7 **.bss** chunks to produce the final **.bss** segment.

[5] As mentioned earlier, these "chunks" are also called "sections".

The assembler creates the **.text**, **.data** and **.bss** chunks but doesn't know exactly where in memory they will end up. However, the linker will figure this out. The linker will decide exactly where each chunk will be placed in memory.

The assembler will know where in each chunk various things are. For example, the assembler might know that the function named **foo** will begin at offset 480 within the **.text** chunk that it produces. As another example, the assembler might place variable **x** at offset 00000000 in the **.data** chunk and **y** at offset 00000004 in the **.data** chunk.

When the linker runs and determines where each chunk will be placed, the linker will then be able to determine exactly where each function and variable is placed. The linker can then go back and patch up the code, filling in and/or adjusting the bytes as necessary.

The Symbol Table and the Relocation Records

Each relocatable object file contains a symbol table and a list of relocation records, in addition to the **.text**, **.data**, and **.bss** segment chunks.

The symbol table lists each symbol that is defined or used in the object file. For symbols that are defined in the object file, the symbol table gives information about them including which chunk (**.text**, **.data**, or **.bss**) and where in the chunk they are located.

The list of relocation records contains a record for each piece of code that needs to be updated by the linker, once the actual locations are known.

An Example

Consider this source file:

foo.c

```
extern int x1;
extern int x2;
extern int x3;

int y1=1111;
int y2=2222;
int y3=3333;
```

```
void foo3 (int i) {
  y1 = x1 + 1234;
  y2 = x2 + 5678;
  y1 = x1 + 2468;
}
```

The **extern** keyword indicates that **x1**, **x2**, and **x3** are defined elsewhere. So some other object file will be responsible for allocating their storage. However, **y1**, **y2**, and **y3** are defined here and given initial values.

Within the code, we see the use of variables that are defined here (**y1** and **y2**) as well as variables that are defined elsewhere (**x1** and **x2**).

Here is the assembly code file. I've simplified it a bit:

foo.s

```
        .data

y1:     .long  1111
y2:     .long  2222
y3:     .long  3333

        .text

foo3:   pushq  %rbp
        movq   %rsp, %rbp
        movl   %edi, -4(%rbp)
        movl   x1(%rip), %eax
        addl   $1234, %eax
        movl   %eax, y1(%rip)
        movl   x2(%rip), %eax
        addl   $5678, %eax
        movl   %eax, y2(%rip)
        movl   x1(%rip), %eax
        addl   $2468, %eax
```

```
        movl   %eax, y1(%rip)
        popq   %rbp
        ret
```

The assembly code says to put into the **.data** chunk the variables **y1**, **y2**, and **y3**. Each is defined to be a **long**, which means 4 bytes in length and each is given an initial value.

The assembly code says to put the instructions into the **.text** chunk. Within the code, we see references to both **y1** and **y2** (which are defined here) and references to **x1** and **x2** (which are defined elsewhere).

Next, look the actual code generated:

```
Offset  MachineCode     AssemblyCode
                                .data

0000    57040000        y1:     .long  1111
0004    AE080000        y2:     .long  2222
0008    050D0000        y3:     .long  3333

                                .text

0000    55              foo3:   pushq  %rbp
0001    4889E5                  movq   %rsp, %rbp
0004    897DFC                  movl   %edi, -4(%rbp)
0007    8B05 00000000           movl   x1(%rip), %eax
000d    05D2040000              addl   $1234, %eax
0012    8905 00000000           movl   %eax, y1(%rip)
0018    8B05 00000000           movl   x2(%rip), %eax
001e    052E160000              addl   $5678, %eax
0023    8905 00000000           movl   %eax, y2(%rip)
0029    8B05 00000000           movl   x1(%rip), %eax
002f    05A4090000              addl   $2468, %eax
0034    8905 00000000           movl   %eax, y1(%rip)
003a    5D                      popq   %rbp
```

```
003b    C3                      ret
```

We see that variables **y1**, **y2**, and **y3** are placed at the beginning of the **.data** chunk at offsets 0, 4, and 8. Their values are initialized. Note that

```
1111 (decimal) = 00000457 (hex)
2222 (decimal) = 000008AE (hex)
3333 (decimal) = 00000D05 (hex)
1234 (decimal) = 000004D2 (hex)
5678 (decimal) = 0000162D (hex)
2468 (decimal) = 000009A4 (hex)
```

We see that the code begins at offset 0 in the **.text** chunk. Note that the offset information in the code for **x1**, **x2**, **y1**, and **y2** is appears as 00000000. These places are underlined.

Next let's look at the **.o** file. The file is not a text file so it's in binary and not human readable, but we can use the **objdump** utility to display it in a human readable way.

The object file begins like this. (I've simplified the display to show the most relevant parts.)

```
file format:
     elf
architecture:
     x86-64
Sections:
     Name          Size
  0 .text         0000003c
  1 .data         0000000c
  2 .bss          00000000
```

Next the .o file contains the symbol table:

```
SYMBOL TABLE:
0000000000000000 g     O .data    0000000000000004 y1
0000000000000004 g     O .data    0000000000000004 y2
0000000000000008 g     O .data    0000000000000004 y3
0000000000000000 g     F .text    000000000000003c foo3
0000000000000000         *UND*    0000000000000000 x1
0000000000000000         *UND*    0000000000000000 x2
```

Look at the third line, which describes variable **y3**. The symbol table entry tells us that the variable is located at offset 8 within the **.data** chunk in this file. It also indicates that it has a size of 4 bytes.

For the symbol **foo3**, we see that it is located at offset 0 in the **.text** chunk. It has a length of 3C (hex).

We also see entries for variables **x1** and **x2**. These symbols are used in this file but they are defined elsewhere. The "*UND*" indicates that they are to be defined elsewhere and the linker needs to find them in some other file. Although the source file mentioned variable **x3**, it was never used so there is nothing about **x3** in the object file.

Finally, the object file contains a list of relocation records:

```
RELOCATION RECORDS FOR [.text]:

OFFSET           TYPE              VALUE
0000000000000009 R_X86_64_PC32     x1-0x0000000000000004
0000000000000014 R_X86_64_PC32     y1-0x0000000000000004
000000000000001a R_X86_64_PC32     x2-0x0000000000000004
0000000000000025 R_X86_64_PC32     y2-0x0000000000000004
000000000000002b R_X86_64_PC32     x1-0x0000000000000004
0000000000000036 R_X86_64_PC32     y1-0x0000000000000004
```

Each of these records indicates where in the **.text** chunk we need to update. Each of the 6 underlined spots above is a place that needs to be filled in. There are 6 records here, which is one record for each place.

The offset indicates where in the **.text** segment to bytes are. For example, the third line indicates offset 1A, which is the location underlined here:

```
0018    8B05<u>00000000</u>            movl   x2(%rip), %eax
001e    052E160000                    addl   $5678, %eax
```

The "type" field indicates that this is a PC-relative offset. In other words, the linker will not plug in the actual address of **x2**.

In an instruction using PC-relative addressing, the location of **x2** is relative to the instruction pointer at the time the instruction is executed. More precisely, it is relative to the address of the next instruction. The instruction pointer (often called the Program

Counter or PC) is **%rip**. So at the time this instruction is executed, **%rip** will point to the next instruction (the **addl** instruction), which is at offset 001E.

So the linker will compute the actual address of the 4 bytes to be updated and it will know the actual address of **x2**. By subtracting them, the linker computes a relative address. But it must be adjusted by 0004 since **%rip** will not point to the 4 bytes in question, but will be pointing at the **addl** instruction when the **movl** is executed. So the meaning of x2-0x0004 is that 4 will be subtracted from the difference, giving the value that must be filled in.

Looking at the result of linking, we can see the modified code.

```
Addr     MachineCode     AssemblyCode
4004ED:  55              foo3:   pushq   %rbp
4004EE:  4889E5                  movq    %rsp,%rbp
4004F1:  897DFC                  movl    %edi,-0x4(%rbp)
4004F4:  8B054A0B2000            movl    x1(%rip),%eax
4004FA:  05D2040000              addl    $1234,%eax
4004FF:  8905330B2000            movl    %eax,y1(%rip)
400505:  8B053D0B2000            movl    x2(%rip),%eax
40050B:  052E160000              addl    $5678,%eax
400510:  8905260B2000            movl    %eax,y2(%rip)
400516:  8B05280B2000            movl    x1(%rip),%eax
40051C:  05A4090000              addl    $2468,%eax
400521:  8905110B2000            movl    %eax,y1(%rip)
400527:  5D                      popq    %rbp
400528:  C3                      retq
```

Here are the locations of the variables:

```
y1      601038
y2      60103C
y3      601040
x1      601044
x2      601048
x3      60104C
```

Look at the following instructions:

```
400505:  8B053D0B2000              movl    x2(%rip),%eax
40050B:  052E160000                addl    $5678,%eax
```

The offset that the linker has filled in is `00200B3D`. At the time the **movl** instruction is executed **%rip** will be pointing to the next instruction, located at: `0040050B`. Adding these:

```
   00200B3D
+  0040050B
   00601048
```

This is the address of variable **x2**, as it should be.

What Can Go Wrong?

In the next sections we discuss potential errors that can occur in the linking phase.

Failure to Declare a Variable

In a "C" program, every variable must have some sort of "declaration". If you fail to declare a variable in any file, the compiler will detect the problem when compiling that file. The compiler will give you a message such as:

```
foo.c: In function 'foo':
foo.c:10:8: error: 'x undeclared (first use in this
function)
   x = 45;
   ^
```

The problem could be that you simply misspelled the variable name, either in the declaration or at the place the variable is used. From the compiler's perspective, misspellings cannot be distinguished from undeclared variables. The compiler simply sees that a variable has been used but there is no declaration of a variable with exactly the same name.

Note that the linker is not involved at all. The compiler detects this problem and the compiler does it when looking only at file **foo.c** (and perhaps some additional **#include** header files). It doesn't matter if the variable is declared in some other **.c** file; every variable must have some sort of declaration in the file during compilation (before the linking step).

These remarks apply to function names as well as variables. If a function is used (that is, if it is called) then the file must include some sort of declaration (either a function definition or a function prototype). These rules ensure that the compiler can check that the variable or function is used appropriately.

Multiple Declarations of a Symbol

Consider these two source code files:

foo.c

```
int x = 111;

foo (...) {
  ...
```

bar.c

```
int x = 222;

bar (...) {
  ...
```

The problem is that there are two declarations of the variable **x**. The compiler cannot catch this problem, but the linker will. It will issue this error message:

```
foo.o:(.data+0xc): multiple definition of `x'
bar.o:(.data+0x8): first defined here
```

Hiding Variables

In the above example with two declarations of the variable **x** the linker detected a problem. But perhaps you intended for these two variables to be different. You might

solve the problem by renaming one of them to prevent the conflict. But perhaps there is some reason you can't rename either one. What else can you do?

Another solution is to add the keyword "static" to a variable's declaration. For example, by changing one of the files, the error can be eliminated.

bar.c

```
static int x = 222;

bar (...) {
  ...
```

Now the variable named **x** in **bar.c** is a private variable. Uses of **x** within **bar.c** will refer to the variable initialized to 222. References to **x** within all other files will refer to the **x** defined in **foo.c**.

What if the Declarations Are Identical?

Here we have modified the files by removing the initializing expression:

foo.c

```
int x;       // Previous version included "=111"

foo (...) {
  ...
```

bar.c

```
int x;       // Previous version included "=222"

bar (...) {
  ...
```

Strangely enough, this works! The linker does not complain. Why?

Strong and Weak Declarations

The difference was the information that the linker has, which is: "Please set aside an area of 4 bytes (an int is 32 bits in "C") and call it **x**. Two separate files request this.

This sort of a declaration is called a "weak" declaration. However, a declaration with the initializing expression is a "strong" declaration. The rule is this:

"There can be multiple weak declarations, but at most only one strong declaration."

What if we have two declarations but the initializing expressions are identical. For example,

foo.c

```
int x = 3333;

foo (...) {
  ...
```

bar.c

```
int x = 3333;

bar (...) {
  ...
```

This is still an error. There are two strong declarations for **x** and that is an error. The linker doesn't look at the actual initializing bytes and doesn't realize that they are equal.

Mismatched Declarations: An Undetected Error

Consider this example:

foo.c

```
double x;

foo (...) {
```

```
x = 123.456;
...
```

bar.c

```
int x;

bar (...) {
  printf ("%d", x);
  ...
```

Here we see two weak declarations of for **x**, but their types don't match! Within each file, the variable is used correctly, so the compiler does not complain.

Unfortunately, the linker does not complain either! The linker doesn't know about types. It sets some space aside for **x** and patches up the code in both programs to refer to that address.

This program prints out:

```
446676599
```

which is undoubtedly not what the programmer intended. It's a bug.

To see why it printed this value, we'd need to look at the bit-level representation of 123.456 as a double precision floating point number. Then we'd need to interpret those same bits as a 32-bit binary number.

There is something else to note about this example. In "C", an **int** is 32 bits and a **double** is 64 bits. These variables are not even the same size! It is conceivable that the assignment

```
x = 123.456;
```

had the effect of overwriting some other unrelated variable. This could be a very difficult bug to find.

The moral of this beware! It is best to put your variable declarations into **.h** header files and **#include** them in the files that need them. This way, all files will see the same declarations.

Whenever you change a header file, it is important to recompile all the files that use the header file.

Generally programmers will use a **makefile** to make sure that programs are recompiled as necessary. However, it's very important to keep track of which files use which header files. When composing a **makefile**, be sure to make the file dependent on both the **.c** source file and any (and all!) files that are **#include**d.

As an example of what can go wrong, consider a **makefile** that fails to include a dependency. Imagine that a variable such as **x** is declared as **int** in a header file. During program development, it becomes necessary to change the declaration of **x** to a **double**. Running **make** will recompile all the programs that need to be recompiled.

But if the **makefile** fails to list all the dependencies correctly, some source file might not get recompiled. This is the exactly the error discussed above! The linker will not catch the problem. And it will be very, very difficult for the programmer to catch it.

For example, the file might contain a line like this:

```
a = x * 4.0;
```

When debugging, the programmer checks the header file and sees that **x** is a **double**. Then the programmer looks at this assignment statement and it looks good.

Yet the code is not doing what the programmer thinks. This assignment statement was compiled earlier, when the header file said that **x** was an **int**. The problem is that the compiler believed that **x** was an **int** at the time is was compiled, so it quietly inserted code to convert the **int** value into a double-sized floating point value. Well, **x** was not an **int**, so that code will essentially mangle the value of **x** before doing the multiplication! The programmer thinks this assignment multiplies **x** by 4, but it really messes things up and sets **a** to something completely wrong!

The programmer might run **make** again to make sure everything is up-to-date, but if the dependency is not listed correctly, the relevant source is still not recompiled.

If you suspect something like this might be going on, then one trick is to **touch**[6] all the **.c** source files and run **make** again. This will force everything to be recompiled. If this touch-trick modifies the program's behavior, it is a clue that the **makefile** might be missing some dependencies.

[6] The command **touch** can be used to update a file's time-of-last-access to the current time.