Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: March 26, 2014 at 15:26
Symbols are lexical entities that name functions and variables.
Each symbol has a value (typically a memory address).
Code consists of symbol definitions and references.
References can be either local or external.

```c
int e = 7;  // def of local e

int main() {
    int r = a();  // ref to external symbol a
    exit(0);  // ref to external symbol exit
    // (defined in libc.so)
}
```
a.c

extern int e;

int *ep = &e;       // def of local ep, ref to external symbol e
int x = 15;        // def of local x
int y;             // def of local y

int a() {
    return *ep + x + y; // refs of locals ep, x, y
}

CS49 Slideset 22: 3  Linking II
```c
int e = 7;

int main() {
    int r = a();
    exit(0);
}
```

Source: objdump

### Disassembly of section .text

00000000 <main>:

0: 55 pushl %ebp
1: 89 e5 movl %esp, %ebp
3: e8 fc ff ff ff call 4<main+0x4>
4: R_386_PC32 a
8: 6a 00 pushl $0x0
a: e8 fc ff ff ff call b<main+0xb>
   b: R_386_PC32 exit
f 90 nop

### Disassembly of section .data

00000000 <e>:

0: 07 00 00 00
```c
extern int e;

int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + y;
}
```

## Disassembly of section .text

```
00000000 <a>:
  0: 55          pushl %ebp
  1: 8b 15 00 00 00 movl 0x0, %edx
  6: 00
  3: R_386_32 ep
  7: a1 00 00 00 00 movl 0x0, %eax
  8: R_386_32 x
  c: 89 e5        movl %esp, %ebp
  e: 03 02        addl (%edx),%eax
 10: 89 ec        movl %ebp, %esp
 12: 03 05 00 00 00 addl 0x0, %eax
 17: 00
 14: R_386_32 y
 18: 5d          popl %ebp
 19: 3c          ret
```
extern int e;

int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + y;
}
## Executable After Relocation

### After Relocation and External Reference Resolution (.text)

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Assembly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>08048530</td>
<td>&lt;main&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048530</td>
<td>55</td>
<td>pushl %ebp</td>
<td></td>
</tr>
<tr>
<td>8048531</td>
<td>89 e5</td>
<td>movl %esp, %ebp</td>
<td></td>
</tr>
<tr>
<td>8048533</td>
<td>e8 08 00 00 00</td>
<td>call 8048540 &lt;a&gt;</td>
<td></td>
</tr>
<tr>
<td>8048538</td>
<td>6a 00</td>
<td>pushl $0x0</td>
<td></td>
</tr>
<tr>
<td>804853a</td>
<td>e8 35 ff ff ff</td>
<td>call 8048474 &lt;_init+0x94&gt;</td>
<td></td>
</tr>
<tr>
<td>804853f</td>
<td>90</td>
<td>nop</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Assembly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>08048540</td>
<td>&lt;a&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048540</td>
<td>55</td>
<td>pushl %ebp</td>
<td></td>
</tr>
<tr>
<td>8048541</td>
<td>8b 15 1c a0 04</td>
<td>movl 0x804a01c, %edx</td>
<td></td>
</tr>
<tr>
<td>8048546</td>
<td>08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048547</td>
<td>a1 20 a0 04 08</td>
<td>movl 0x804a020, %eax</td>
<td></td>
</tr>
<tr>
<td>804854c</td>
<td>89 e5</td>
<td>movl %esp, %ebp</td>
<td></td>
</tr>
<tr>
<td>804854e</td>
<td>03 02</td>
<td>addl (%edx), %eax</td>
<td></td>
</tr>
<tr>
<td>8048550</td>
<td>89 ec</td>
<td>movl %ebp, %esp</td>
<td></td>
</tr>
<tr>
<td>8048552</td>
<td>03 05 d0 a3 04</td>
<td>addl 0x804a3d0, %eax</td>
<td></td>
</tr>
<tr>
<td>8048557</td>
<td>08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048558</td>
<td>5d</td>
<td>popl %ebp</td>
<td></td>
</tr>
<tr>
<td>8048559</td>
<td>c3</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Executable After Relocation

After Relocation and External Reference Resolution (.data)

m.c

```c
int e = 7;

int main() {
    int r = a();
    exit(0);
}
```

a.c

```c
extern int e;

int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + ; y;
}
```

Disassembly of section .data

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0804a018</td>
<td>&lt;e&gt;</td>
</tr>
<tr>
<td>804a018</td>
<td>07 00 00 00</td>
</tr>
<tr>
<td>0804a01c</td>
<td>&lt;ep&gt;</td>
</tr>
<tr>
<td>804a01c</td>
<td>18 a0 04 08</td>
</tr>
<tr>
<td>0804a020</td>
<td>&lt;x&gt;</td>
</tr>
<tr>
<td>804a020</td>
<td>0f 00 00 00</td>
</tr>
</tbody>
</table>
Program symbols are either *strong* or *weak*.

**strong**: procedures and initialized globals

**weak**: uninitialized globals

This doesn't apply to local variables.

```
# p1.c
int foo = 5;  // foo: strong
p1() {        // p1: strong
...
}

# p2.c
int foo;      // foo: weak here
p2() {        // p2: strong
...
}
Rule 1: A strong symbol can only appear once.

Rule 2: A weak symbol can be overridden by a strong symbol of the same name.
    - References to the weak symbol resolve to the strong symbol.

Rule 3: If there are multiple weak symbols, the linker can pick one arbitrarily.
What happens in each case?

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int x;</code></td>
<td><code>p1() {}</code></td>
<td></td>
</tr>
<tr>
<td><code>p1() {}</code></td>
<td><code>p1() {}</code></td>
<td></td>
</tr>
<tr>
<td><code>int x;</code></td>
<td><code>int x;</code></td>
<td></td>
</tr>
<tr>
<td><code>p1() {}</code></td>
<td><code>p2() {}</code></td>
<td></td>
</tr>
<tr>
<td><code>int x;</code></td>
<td><code>double x;</code></td>
<td></td>
</tr>
<tr>
<td><code>int y;</code></td>
<td><code>p2() {}</code></td>
<td></td>
</tr>
<tr>
<td><code>p1() {}</code></td>
<td><code>p2() {}</code></td>
<td></td>
</tr>
<tr>
<td><code>int x=7;</code></td>
<td><code>double x;</code></td>
<td></td>
</tr>
<tr>
<td><code>int y=5;</code></td>
<td><code>p2() {}</code></td>
<td></td>
</tr>
<tr>
<td><code>p1() {}</code></td>
<td><code>p2() {}</code></td>
<td></td>
</tr>
<tr>
<td><code>int x=7;</code></td>
<td><code>int x;</code></td>
<td></td>
</tr>
<tr>
<td><code>p1() {}</code></td>
<td><code>p2() {}</code></td>
<td></td>
</tr>
</tbody>
</table>
Think carefully about each of these.

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x; p1() {}</td>
<td>p1() {}</td>
<td>Link time error: two strong symbols (p1)</td>
</tr>
<tr>
<td>int x; p1() {}</td>
<td>int x; p2() {}</td>
<td>References to (x) will refer to the same uninitialized int. What you wanted?</td>
</tr>
<tr>
<td>int x; int y; p1() {}</td>
<td>double x; p2() {}</td>
<td>Writes to (x) in p2 might overwrite (y)! That’s just evil!</td>
</tr>
<tr>
<td>int x=7; int y=5; p1() {}</td>
<td>double x; p2() {}</td>
<td>Writes to (x) in p2 might overwrite (y)! Very nasty!</td>
</tr>
<tr>
<td>int x=7; p1() {}</td>
<td>int x; p2() {}</td>
<td>References to (x) will refer to the same initialized variable.</td>
</tr>
</tbody>
</table>

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.
How to package functions commonly used by programmers? (Math, I/O, memory management, string manipulation, etc.)

Awkward, given the linker framework so far:

- **Option 1**: Put all functions into a single source file.
  - Programmers link big object file into their programs.
  - Space and time inefficient.
- **Option 2**: Put each function in a separate source file.
  - Programmers explicitly link appropriate binaries into their programs.
  - More efficient, but burdensome on the programmer.
Solution: *static libraries* (.a archive files)

- Concatenate related relocatable object files into a single file with an index (called an archive).
- Enhance the linker so that it tries to resolve unresolved external reference by looking for symbols in one or more archives.
- If an archive member resolves the reference, link into the executable.
Static Libraries (archives)

libc.a is a static library (archive) of relocatable object files concatenated into one file.

The output p is an executable object file that only contains code and data for libc functions called from p1.c and p2.c.

This further improves modularity and efficiency by packaging commonly used functions, e.g., C standard library (libc) or math library (libm).

The linker includes only the .o files in the archive that are actually needed by the program.
Command: `ar rs libc.a atoi.o printf.o ... random.o`

Archiver allows incremental updates: Recompile a function that changes and replace the .o file in the archive.
Commonly Used Libraries

**libc.a** (the C standard library)
- 8MB archive of 900 object files
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

**libm.a** (the C math library)
- 1MB archive of 226 object files
- floating point math (sin, cos, tan, log, exp, sqrt, ...)

```bash
% ar -t /usr/lib/libc.a | sort
... fork.o ...
... printf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
...
```

```bash
% ar -t /usr/lib/libm.a | sort
... e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
...
```
Linker’s algorithm for resolving external references:

- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.
- If there are any entries in the unresolved list at the end of the scan, then error.

Problem:

- Command line order matters.
- Moral: put libraries at the end of the command line.

```
> gcc -L libtest.o -lmine
> gcc -L -lmine libtest.o
libtest.o: In function ‘main’:
libtest.o(.text+0x4): undefined reference to ‘libfun’
```
Executable object file for example program p:

<table>
<thead>
<tr>
<th>ELF header</th>
<th>Virtual addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header tables (required for executables)</td>
<td>0x080483e0</td>
</tr>
<tr>
<td>.text section</td>
<td>0x08048494</td>
</tr>
<tr>
<td>.data section</td>
<td>0x0804a010</td>
</tr>
<tr>
<td>.bss section</td>
<td>0x0804a3b0</td>
</tr>
<tr>
<td>.symtab</td>
<td></td>
</tr>
<tr>
<td>.rel.txt</td>
<td></td>
</tr>
<tr>
<td>.rel.data</td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
</tr>
</tbody>
</table>

Section header table (required for relocatables)

Loaded segments:

- Process image:
  - init and shared lib segments
  - .text segment (r/o)
  - .data segment (initialized r/w)
  - .bss segment (uninitialized r/w)
Shared Libraries

Static libraries have some disadvantages:

- Potential for duplicating lots of common code in the executable files on a file system. (e.g., every program needs the standard C library).
- Potential for duplicating lots of code in the virtual memory space of many processes.
- Minor bug fixes of system libraries require each application to explicitly relink.
Solution:

- **Shared libraries** (dynamic link libraries DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.

- Dynamic linking can occur when an executable is first loaded and run. (The common case for Linux, handled automatically by `ld-linux.so`.)

- Dynamic linking can also occur after the program has begun.
  - In Linux, this is done explicitly by user with `dlopen()`.
  - Basis for High-Performance Web Servers.

- Shared library routines can be shared by multiple processes.
Dynamically Linked Shared Libraries

Processes: m.c and a.c are loaded, linked, and libc.so functions called by m.c and a.c are loaded, linked, and (potentially) shared among processes.

- Translators (cc1, as)
  - m.c
  - a.c
  - m.o
  - a.o
- Linker (ld)
  - p
  - libc.so
- Loader/Dynamic Linker (ld−linux.so)
  - Fully linked executable p’ (in memory)
  - Fully linked executable p’ (in memory)
- Partially linked executable p (on disk)

Shared library of dynamically relocatable object files