ELF1 1E Common Symbol

Young W. Lim

2022-09-12 Mon

Outline

Based on

"Study of ELF loading and relocs", 1999
http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Compling 32-bit program on 64-bit gcc

- gcc -v
- gcc -m32 t.c
- sudo apt-get install gcc-multilib
- sudo apt-get install g++-multilib
- gcc-multilib
- g++-multilib
- gcc -m32
- objdump -m i386

uninitialized global variable (1)

gcc, in C mode:

- uninitialised globals which are not declared extern
  are treated as common symbols, not weak symbols.
- common symbols are merged at link time
  so that they all refer to the same storage;
  - if more than one object attempts to *initialise* such a symbol,
    you will get a *link-time error*
  - if they are not explicitly initialised anywhere,
    they will be placed in the BSS, i.e. initialised to 0.

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-

uninitialized global variable (2)

gcc, in C++ mode:

- not the same as in c mode
- there is no common symbols in C++
- Uninitialised globals which are not declared extern
  are implicitly initialised to a default value
  (0 for simple types, or default constructor).

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

uninitialized global variable (3)

- In either case,
  a weak initialized symbol can be overridden
  by a non-weak initialised symbol of the same name
  at link time

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

uninitialized global variable (4)

1. init.c

```
int global = 999;

int main(void) { ... }
```

2. uninit.c

```
int global;

int main(void) { ... }
```

3. extern.c

```
extern int global;

int main(void) { ... }
```

4. weak.c

```
int global __attribute__((weak))
    = 999;

int main(void) { ... }
```

5. another.c

```
int global = 1234;
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

uninitialized global variable (5)

1. init.c

```
#include <stdio.h>
int global = 999;
int main(void) {
  printf("%d\n", global);
  return 0;
}
```

2. uninit.c

```
#include <stdio.h>
int global;
int main(void) {
  printf("%d\n", global);
  return 0;
}
```

3. extern.c

```
#include <stdio.h>
extern int global;
int main(void) {
  printf("%d\n", global);
  return 0;
}
```

4. weak.c

```
#include <stdio.h>
int global __attribute__((weak))
    = 999;
int main(void) {
  printf("%d\n", global);
  return 0;    }
```

5. another.c

```
int global = 1234;
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

uninitialized global variable (6)

1.	init.c	`int global=999;`
2.	uninit.c	`int global;`
3.	extern.c	`extern int global;`
4.	weak.c	`int global __attribute__((weak))=999;`
5.	another.c	`int global = 1234;`

case 1	uninit.c	0
	uninit.c another.c	1234
case 2	init.c another.c	multiple definition
case 3	extern.c	undefined reference
	extern.c another.c	1234
case 4	weak.c	999
	weak.c another.c	1234

uninitialized global variable (7)

1. init.c	`int global=999;`	
2. uninit.c	`int global;`	
3. extern.c	`extern int global;`	
4. weak.c	`int global __attribute__((weak))=999;`	
5. another.c	`int global = 1234;`	

case 1	gcc -o test uninit.c && ./test	0
	gcc -o test uninit.c another.c && ./test	1234
case 2	gcc -o test init.c another.c && ./test	mulitple definition
case 3	gcc -o test extern.c && ./test	undefined
	gcc -o test extern.c another.c && ./test	1234
case 4	gcc -o test weak.c && ./test	999
	gcc -o test weak.c another.c && ./test	1234

Case 1 uninit

1.	init.c	`int global=999;`
2.	uninit.c	`int global;`
3.	extern.c	`extern int global;`
4.	weak.c	`int global __attribute__((weak))=999;`
5.	another.c	`int global = 1234;`

1. uninit

```
$ gcc -o test uninit.c && ./test
0

$ gcc -o test uninit.c another.c && ./test
1234
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-

Case 2 init

1.	init.c	`int global=999;`
2.	uninit.c	`int global;`
3.	extern.c	`extern int global;`
4.	weak.c	`int global __attribute__((weak))=999;`
5.	another.c	`int global = 1234;`

2. init

```
$ gcc -o test init.c another.c && ./test
/tmp/cc5DQeaz.o:(.data+0x0): multiple definition of ‘global’
/tmp/ccgyz6rL.o:(.data+0x0): first defined here
collect2: ld returned 1 exit status
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

Case 3 extern

1.	init.c	`int global=999;`
2.	uninit.c	`int global;`
3.	extern.c	`extern int global;`
4.	weak.c	`int global __attribute__((weak))=999;`
5.	another.c	`int global = 1234;`

3. extern

```
$ gcc -o test extern.c && ./test
/tmp/ccqdYUIr.o: In function ‘main’:
main_uninit_extern.c:(.text+0x12): undefined reference to ‘global’
collect2: ld returned 1 exit status#+end_src

$ gcc -o test extern.c another.c && ./test
1234
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

Case 4 weak

1.	init.c	`int global=999;`
2.	uninit.c	`int global;`
3.	extern.c	`extern int global;`
4.	weak.c	`int global __attribute__((weak))=999;`
5.	another.c	`int global = 1234;`

4. weak

```
$ gcc -o test weak.c && ./test
999

$ gcc -o test weak.c another.c && ./test
1234
```

https://stackoverflow.com/questions/3691835/why-uninitialized-global-variable-is-w

Common symbols (1)

- common symbols allow a programmer
  to *define* several variables
  of the same name in different source files
- the other way is to define a variable
  once in *one* source file, and
  reference it everywhere else
  in *other* source files,
  using `extern`

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

Common symbols (2)

- when common symbols are used,
  the linker will merge all symbols of the same name
  into a single memory location
- the size of which is the largest type of
  the individual common symbol definitions.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

Common symbols (3)

- `fileA.c` defines an uninitialized 32-bit integer `myint`
- `fileB.c` defines an 8-bit character `myint`,
- then in the final executable,
  references to `myint` from both files
  will point to the same memory location (common location),
  and the linker will reserve 32 bits for that location.

`https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml`

Common symbols (4)

- COMMON symbols are contained
  only in relocatable object files,
  not in executable object files.
- they are generated by the compiler / assembler
  when creating an object file from a single source file.
- later, the linker will need to interpret these symbols.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

Common symbols (5)

- Remember that ELF reserves
  a special section header table index
  for referring to a COMMON section:
- the index COM
  - just like the special indices ABS and UND,
  - these sections do not physically exist in the file
- common symbols defined in the symbol table
  of relocatable object files
  have their section index member set to COM

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

Common symbols (6)

- common symbols first appeared
  as a feature of the FORTRAN language.
- common symbols are present only
  for backward-compatibility
  with *old* source files
  where `extern` is not used
- nowadays, the best practice is to make use of
  only one definition of a variable,
  and use `extern` in all *other source files*
  that reference it

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

Common symbols (7)

- if a global variable is not initialized
  in a C source file,
- after compiling, we would expect the variable to go to
  the .bss section in the relocatable object file
- However, *by default*, GCC will put the symbol
  in the COMMON section of the file;
- that is, the option -fcommon is the default behaviour.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(1a) COMMON

- *by default*, the uninitialized variable `un_a`
  is put in the common section.
  `int un_a` → `COM un_a` (common)
- when compile with `-fno-common`,
  the section of `un_a` is now at index 3,
  which is the `.bss` section
  of the `main.o` relocatable object file
  `int un_a, -fno-common` → `3 un_a` (`.bss`)

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(1b) COMMON

main.c

```
int un_a;      // common

int main() {
     return 0;
}
```

script

```
gcc -c -o main.o main.c
readelf -s main.o

gcc -c -o main.o main.c -fno-common
readelf -s main.o
```

results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a
$> gcc -c -o main.o main.c -fno-common
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(2a) .data

- if the variable is initialized to a certain value,
  then it is placed in the .data section in the output file
  by the compiler (actually the assembler)
- note that section index 2 corresponds
  to the .data section here:
  int un_a=9 → 2 un_a (.data)
- use -SW options to readelf to list all sections

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(2b) .data

main.c

```
int un_a = 9;      // .data

int main() {
     return 0;
}
```

script

```
$> gcc -c -o main.o main.c

$> readelf -s main.o
```

results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3a) .bss

- If we define a global variable,
  and explicitly initialise it to zero,
  then it will be put in the .bss section
  `int un_a=0` → 2 `un_a` (.bss)
- although it is initialized and
  logically should go into .data,
  the compiler knows it is optimal to put it in the .bss,
  as in any case it will become initialized to zero at runtime,
  and in .bss will not consume *file space*

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3b) .bss

main.c

```
int un_a = 0;      // .bss

int main() {
     return 0;
}
```

script

```
$> gcc -c -o main.o main.c

$> readelf -s main.o
```

results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

Two object files defining the same symbol

	main.c main.o	swap.c swap.o	 prog
case 1	int un_a COM	int un_a=0 .bss	 .bss
case 2	int un_a COM	extern int un_a UND	 .bss
case 3	int un_a COM	int un_a COM	 .bss
case 4	int un_a=0 .bss	int un_a=9 .data	 *multiple definition*
case 5	int un_a=10 .data	extern int un_a UND	 .data

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(1a) COMMON, .bss

1. int un_a;, int un_a=0; (COMMON, .bss)

main.c

```
int un_a;      // common

int main() {
     return 0;
}
```

swap.c

```
int un_a=0;    // .bss

int swap() {
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(1b) COMMON, .bss

1. int un_a;, int un_a=0; (COMMON, .bss)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 un_a

$> gcc -o prog main.o swap.o
$> readelf -s prog | grep 'un_a'
    49: 0000000000601030     4 OBJECT  GLOBAL DEFAULT   25 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(1c) COMMON, .bss

1. `int un_a;`, `int un_a=0;` (COMMON, .bss)
   - the linker creates the final variable `un_a`
     in the `.bss` section of the executable object file
     (index 25 is the index of `.bss` as shown by `readelf -SW`)
   - In the file `swap.c`, `un_a` were initialised with a non-zero value,
     the variable would have been in the `.data` of the relocatable `swap.o`,
     and the linker would have then placed it
     in the `.data` section of the executable.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(2a) COMMON, undefined

2. int un_a;, extern int un_a; (COMMON, undefined)

main.c

```
int un_a;

int main() {

     return 0;
}
```

swap.c

```
extern int un_a;

int swap() {
     int a = un_a;
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(2b) COMMON, undefined

2. int un_a;, extern int un_a; (COMMON, undefined)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND un_a

$> gcc -o prog main.o swap.o
$> readelf -s prog | grep 'un_a'
   49: 0000000000601030     4 OBJECT  GLOBAL DEFAULT   25 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(2c) COMMON, undefined

2. `int un_a;`, `extern int un_a;` (COMMON, undefined)
   - As we see, in the final executable, our variable is located in the `.bss` section.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3a) COMMON, COMMON

3. int un_a;, int un_a; (COMMON, COMMON)

main.c

```
int un_a;

int main() {
     return 0;
}
```

swap.c

```
int un_a;

int swap() {
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3b) COMMON, COMMON

3. int un_a;, int un_a; (COMMON, COMMON)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     8: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM un_a

$> gcc -o prog main.o swap.o
$> readelf -s prog | grep 'un_a'
    49: 0000000000601030     4 OBJECT  GLOBAL DEFAULT   25 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(3c) COMMON, COMMON

3. `int un_a;`, `int un_a;` (COMMON, COMMON)
   - We see here also, the variable is located in `.bss` of final executable.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(4a) .bss, .data

4. int un_a=0;, int un_a=9; (.bss, .data)

main.c

```
int un_a=0;

int main() {
     return 0;
}
```

swap.c

```
int un_a=9;

int swap() {
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(4b) .bss, .data

4. int un_a=0;, int un_a=9; (.bss, .data)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 un_a
$> gcc -o prog main.o swap.o
swap.o:(.data+0x0): multiple definition of `un_a'
main.o:(.bss+0x0): first defined here
collect2: error: ld returned 1 exit status
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(4c) .bss, .data

4. `int un_a=0;`, `int un_a=9;` (`.bss`, `.data`)
   - a linking error.
     The codes from both files each reference
     their own data location, initialized differently.
     - if it takes the value 9 and puts it in the `.data` section
       for symbol `un_a`, then the code in `main.c` may behave *incorrectly*
       as it was written assuming the initial value of `un_a` to be `0`.
     - if it were to choose to put `un_a` in `.bss`,
       such that at runtime the initial value of `un_a` will be `0`,
       the code in `main.c` will work correctly
       but code from `swap.c` will likely not function well.

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml

(5a) .data, undefined

5. int un_a=10;, extern int un_a; (.data, undefined)

main.c

```
int un_a=10;      // .data

int main() {
     return 0;
}
```

swap.c

```
extern int un_a;    // undefined

int swap() {
     int a = un_a;
     return 108;
}
```

script

```
$> gcc -c -o main.o main.c
$> gcc -c -o swap.o swap.c
$> gcc -o prog main.o swap.o
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(5b) .data, undefined

5. int un_a=10;, extern int un_a; (.data, undefined)

script and results

```
$> gcc -c -o main.o main.c
$> readelf -s main.o
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 un_a

$> gcc -c -o swap.o swap.c
$> readelf -s swap.o
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND un_a

$> gcc -o prog main.o swap.o
$> readelf -s prog | grep 'un_a'
    49: 000000000060102c     4 OBJECT  GLOBAL DEFAULT   24 un_a
```

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-symb

(5c) .data, undefined

5. `int un_a=10;`, `extern int un_a;` (.data, undefined)
   this example shows a normal and common case.
   We see that finally the variable is defined
   in the .data of the executable (section index 24).

https://binarydodo.wordpress.com/2016/05/09/investigating-linking-with-common-syml