12.2 Symbol resolution

- For each **local symbol**, the compiler guarantees exactly one definition. The name is modified to be unique (*e.g.* `count` above).
- If *the compiler* **finds no definition**, it expects it to come from another module, and leaves it to the linker, (*e.g.* `buf` above).
- When **the linker** resolves *global* symbols, several conditions can occur:
  - **No definition** is found in the symbol table of any input object file.
  - **Multiple definitions** are found in different object files, and one must be chosen.

Example No `main` function, and `buf` undefined.

```
$ pk-cc swap.o # without -c, try to build an executable
.../lib/crt1.o: In function ‘_start':
(.text+0x20): undefined reference to ‘main'
swap.o: In function ‘swap':
.../swap.c:12: undefined reference to ‘buf'
swap.o:(.data+0x0): undefined reference to ‘buf'
collect2: error: ld returned 1 exit status
```

- The linker tries to link with `crt1.o`, wich refers to the `main` function.

Choosing one among multiple definitions

- The linker distinguishes weak and strong symbols:
  - Functions and initialized global variables are **strong symbols**.
  - Uninitialized global variables are **weak symbols**.
- If a conflict arises, the strategy is as follows:
  - Multiple strong symbols → raise error.
  - One strong, and multiple weak symbols → choose the strong one.
  - Multiple weak symbols → choose an arbitrary one.

Example Two strong symbols

```
$ pk-cc -c foo1.c bar1.c          # compilation is fine
$ pk-cc foo1.o bar1.o
bar1.o: In function 'main':
.../bar1.c:2: multiple definition of 'main'
foo1.o:.../foo1.c:2: first defined here
collect2: error: ld returned 1 exit status
```

foo1.c

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

bar1.c

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

Question What will happen here?

foo2.c

```c
#include <stdio.h>

void f(void);
int x = 12345;

int main(void)
{
    f();
    printf("x = %d\n", x);

    return 0;
}
```

bar2.c

```c
int x = 54321;

void f(void)
{
    x++;
}
```

Example One strong, and one weak symbol.

foo3.c

```c
#include <stdio.h>

void f(void);

int x = 12345;  /* strong */

int main(void)
{
    f();
    printf("x = %d\n", x);

    return 0;
}
```

bar3.c

```c
int x;  /* weak */

void f(void)
{
    x = 54321;
}
```

The result is probably expected:

```
$ pk-cc -c foo3.c bar3.c
$ pk-cc foo3.o bar3.o
$ ./a.out
x = 54321
```

- Maybe check out the symbol tables? `readelf -s {foo,bar}3.o`

Example Two weak symbols.

foo4.c

```
#include <stdio.h>

void f(void);

int x;  /* weak */

int main(void)
{
    x = 12345;
    f();
    printf("x = %d\n", x);

    return 0;
}
```

bar4.c

```
int x;  /* weak */

void f(void)
{
    x = 54321;
}
```

Again, no surprise:

```
$ pk-cc -c foo4.c bar4.c
$ pk-cc foo4.o bar4.o
$ ./a.out
x = 54321
```

Note The linker has unified both variables `x`, and makes references to both symbols address the same space in memory.

Question What about this one?

foo5.c

```c
#include <stdio.h>

void f(void);

int x = 12345;  /* strong */
int y = 54321;  /* strong */

int main(void)
{
    f();
    printf("x = %d, y = %d\n", x, y);

    return 0;
}
```

bar5.c

```c
double x;  /* weak */

void f(void)
{
    x = 1;
}
```

Explain this:

```
$ pk-cc -c foo5.c bar5.c
$ pk-cc foo5.o bar5.o
/usr/bin/ld: Warning: alignment 4
of symbol ‘x' in foo5.o is smaller
than 8 in bar5.o
$ ./a.out
x = 0, y = 1072693248
```

Notes

- Not long ago, the linker would give **no warning** at all.
- It is extremely difficult to **debug** such code.

What else?

- After resolving symbols, the linker knows which definition belongs to each symbol.

Recall

- Machine code does not use variable names any more.
- The compiler produced code that accesses variables and functions only by their **memory addresses**.

⇒ How does this go together with separate compilation and symbol resolution?

12.3 The program in memory

How does a program start?

- When a program is run, it is **copied into memory** by the **loader**.
  - Copy **text segment**, *i.e.*, the actual machine code,
  - copy **initialized data**,
  - **initialize** uninitialized data,
  - *etc.*
- We want to minimize the amount of data to be copied!
  - Only load parts that are **actually required**,
  - and only load them **when** they are needed.
- Wa want to save memory!
  - Do not load the same code into memory multiple times.
  - **Share** already loaded code between processes.
- Avoid expensive **transformations**
  - Store program on disk in a **format** that allows fast setup of the process image.

Virtual memory

- VM is a mapping from the process' **virtual address space** into the machine's physical address space (organized in **pages**).
- The VM system may flag pages as, *e.g.*, **read only**, **executable**, or **private**, *cf.* `mmap`(2).
- A physical page may reside on disk, until **loaded on demand**.
  - So we compile the memory layout into the executable file,
  - the loader just **maps the file** into the process' virtual address space, and
  - the VM system gets the pages into memory when actually referenced.
- Multiple running instances of a program share their text (machine code) through a *read only* mapping to **the same physical** address space.

Note To achieve all this, the structure of the program file depends on the process' memory layout!

Example

- The mapping of virtual memory can be observed in `/proc/$PID/maps`.
- I have several `xterm`s running, one with PID 23172.

```
$ cat /proc/23172/maps                                    # this is on a 64bit machine
# address                 perms offset   dev   inode     pathname
00400000-00472000          r-xp 00000000 00:10 987773    /usr/bin/xterm
00672000-00673000          r--p 00072000 00:10 987773    /usr/bin/xterm
00673000-0067c000          rw-p 00073000 00:10 987773    /usr/bin/xterm
0067c000-0067e000          rw-p 00000000 00:00 0
006f7000-00718000          rw-p 00000000 00:00 0         [heap]
# ...
7f05d802c000-7f05d81cc000 r-xp 00000000 00:10 953240    /usr/lib/libc-2.18.so
# ...
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

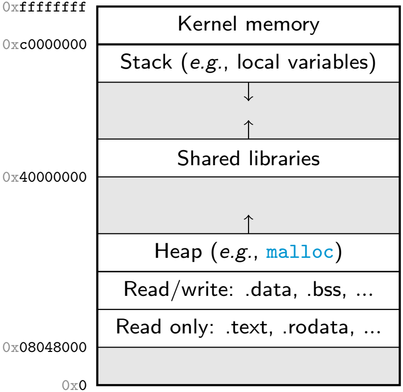
- The executable pages are readonly,
- the writable area is private (*i.e.*, copy-on-write).
- Other instances of `xterm` have the same mapping for the binary `xterm`,
- but may have others for the shared libraries.

(*cf.* `proc`(5) for more information)

Process memory layout

When running, a process has the following **virtual memory layout**.

(This is for 32bit Linux)



- Kernel memory (1GiB) is not accessible by the process.
  - **Shared** among all processes.
- The stack maintains local variables and function calls.
- Shared libraries may even be added at runtime.
- The heap contains allocated memory.
- `.data` and `.bss` store global variables.
- `.text` and `.rodata` are marked `ro`, so can be shared with other processes.

12.4 Object file layout

- There are various formats to store binary programs.
- Linux uses ELF, the **Executable and Linking Format**.
- COFF and `a.out` are others, the latter coined the name used by `gcc` for default binaries (in ELF on Linux!).
- All formats have the concept of **sections** in common.
- A section is the unit of organization in a binary.

Some section names (but there are many more)

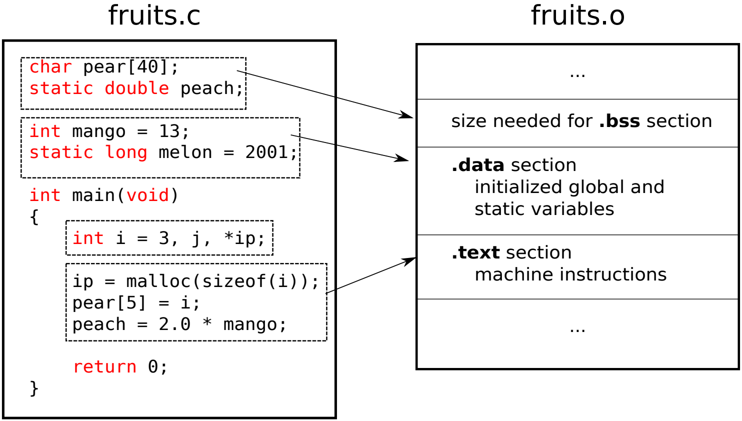
`.text` The program code, *i.e.*, processor instructions.

`.rodata` Read-only data, *e.g.*, string literals.

`.data` *Initialized global* variables.

`.bss` *Uninitialized global* variables.

`.symtab` The symbol table, displayed with `readelf -s`.


fruits.c
char pear[40];
static double peach;
int mango = 13;
static long melon = 2001;
int main(void)
{
int i = 3, j, *ip;
ip = malloc(sizeof(i));
pear[5] = i;
peach = 2.0 * mango;
return 0;
}
fruits.o
...
size needed for .bss section
.data section
initialized global and
static variables
.text section
machine instructions
...

A typical *relocatable* object file

This is what the compiler produces out of the **individual C files**.

	ELF header
Sections	`.text`
	`.rodata`
	`.data`
	`.bss`
	`.symtab`
	`.rel.text`
	`.rel.data`
	`.debug`
	`.line`
	`.strtab`
	Section header table

- The **ELF header** describes word size, endian, object file type, machine type, offset and format of the section header table, and other information.
- The **section header table** describes the locations of the various sections.
- Try

```
$ pk-cc -c -m32 swap.c
$ readelf -S swap.o
```

A typical *executable* object file

That is what we want to have in the **final binary program**.

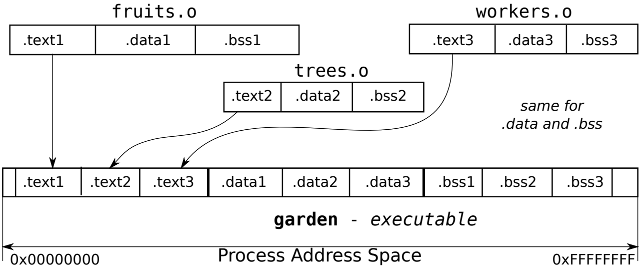
	ELF header
	Segment header table
Code segment mapped read-only	.init
	.text
	.rodata
Data segment mapped read/write	.data
	.bss
Symbol table and debug info are not loaded into memory	.symtab
	.debug
	.line
	.strtab
	Section header table

- The **segment header table** describes the mapping of contiguous file sections into memory.
- Try this

```
$ pk-cc -m32 -static swap.c \
> main.c
$ readelf -l a.out
```

12.5 Relocation

- So after resolving the symbols, the linker needs to put all the code from the individual object files' sections into the final program's sections:



- This process is called **Relocation**.

Relocation involves **two tasks**:

1. Relocating **sections** and symbol **definitions**.
   - Merge sections of the same type.
   - Assign **run-time memory addresses** to the new aggregate sections, the input sections, and each symbol defined in the input.

   ⇒ After this step, every global symbol has a known run-time memory address.
2. Relocating symbol **references** everywhere in the code.
   - Modification of each reference in `.text` and `.data`, so that they point to correct location.

Relocation entries

- The assembler does not know where data and code will be stored ultimately,
- nor does it know addresses of the external objects.

⇒ In such situations a *relocation entry* is generated by the assembler.

Relocation entries

```
typedef struct {
  int offset;       /* Offset of the reference to relocate. */
  int symbol: 24,   /* Symbol the reference should point to. */
      type:8;       /* Relocation type. */
} Elf32_Rel;
```

- ELF defines 11 different relocation types
- We will look at `R_386_32` only.
  This is used to relocate 32bit absolute addresses.

Example: Relocation at work

- Function `f` simply assigns `0xbeef` to the global variable `x`.
- The final memory location of `x` is not yet known.
- Relocation will fix the runtime address of `x`.

foo6.c

```
int x = 0xdead;

void f(void)
{
    x = 0xbeef;
}
```

- First we **compile** `foo.c` into a *relocatable object file*:

```
$ pk-cc -m32 -c foo6.c
```

- Have a look at the `.data` section:

```
$ objdump -d -j.data foo6.o
Disassembly of section .data:
00000000 <x>:
   0:   ad de 00 00                                        ....
```

- Variable `x` appears at address `0x0` in the `.data` section.
- The value `0xdead` is stored there.

- Have a look at the `.text` section:

```
1 $ objdump -d -j.text foo6.o
2 Disassembly of section .text:
3 00000000 <f>:
4    0:   55                      push   %ebp
5    1:   89 e5                   mov    %esp,%ebp
6    3:   c7 05 00 00 00 00 ef    movl   $0xbeef,0x0
7    a:   be 00 00
8    d:   5d                      pop    %ebp
9    e:   c3                      ret
```

  - In line 6, the value `0xbeef` is copied to address `0x0`.
  - This address `0x0` appears at **offset 5** in the `.text` section.

- These are the relocation entries:

```
$ objdump -r -j.text foo6.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000005 R_386_32          x
```

  - So on relocation of symbol `x`, the absolute 32bit address at **offset 5** in the `.text` section must be updated.

- Then we **link** foo.o with something that uses f.

```
$ pk-cc -m32 foo6.o bar6.c                /* main in bar6.c simply calls f in foo6.c */
```

- Have a look at the .data section **after relocation**:

```
$ objdump -d -j.data a.out
Disassembly of section .data:
08049698 <x>:
 8049698:       ad de 00 00                                            ....
```

  - Variable x has been moved to address 0x8049698 in the .data section.
  - The value 0xdead is stored there.

- Have a look at the .text section **after relocation**:

```
$ objdump -d -j.text a.out | grep -C3 beef
080483cd <f>:
 80483cd:       55                      push   %ebp
 80483ce:       89 e5                   mov    %esp,%ebp
 80483d0:       c7 05 98 96 04 08 ef    movl   $0xbeef,0x8049698
 80483d7:       be 00 00
 80483da:       5d                      pop    %ebp
 80483db:       c3                      ret
```

  - The reference to variable x has been **updated to address** 0x8049698.

Example: Relocation in the .data section

- Sometimes, updating references in the .text section is not enough.
- Here we have a global variable xp initialized with an address!
- The compiler **cannot even fix a value** for xp!

foo7.c

```
int x = 0xdead;
int *xp = &x;

void f(void)
{
    *xp = 0xbeef;
}
```

- Again, we compile and link our object files:

```
$ pk-cc -m32 -c foo7.c
$ pk-cc -m32 foo7.o bar6.c
```

- Relocation in the `.data` section.

```
$ objdump -d -j.data foo7.o
Disassembly of section .data:
00000000 <x>:
   0:   ad de 00 00                                    ....
00000004 <xp>:
   4:   00 00 00 00    # This value has to be updated on relocation of x!
$ objdump -r -j.data foo7.o    # Note: in .data this time!
RELOCATION RECORDS FOR [.data]:
OFFSET   TYPE              VALUE
00000004 R_386_32          x
$ objdump -d -j.data a.out
Disassembly of section .data:
08049698 <x>:
 8049698:       ad de 00 00                                    ....
0804969c <xp>:
 804969c:       98 96 04 08                                    ....
```

- Relocation in the `.text` section:

```
$ objdump -d -j.text foo7.o
Disassembly of section .text:
00000000 <f>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   a1 00 00 00 00          mov    0x0,%eax
   8:   c7 00 ef be 00 00       movl   $0xbeef,(%eax)
   e:   5d                      pop    %ebp
   f:   c3                      ret
$ objdump -r -j.text foo7.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_386_32          xp
$ objdump -d -j.text a.out | grep -C3 beef
 80483cd:       55                      push   %ebp
 80483ce:       89 e5                   mov    %esp,%ebp
 80483d0:       a1 9c 96 04 08          mov    0x804969c,%eax
 80483d5:       c7 00 ef be 00 00       movl   $0xbeef,(%eax)
 80483db:       5d                      pop    %ebp
 80483dc:       c3                      ret
```

- Recall: `0x804969c` is the address of the variable `xp`!