Specifying Branch and Call Targets

- In assembly language, we indicate (direct) jumps and calls with a symbolic syntax.
  - `JMP Label`
  - `Jcc Label`
  - `CALL Label`
- The machine-level ISA reference specifies these instructions as having an offset operand.

Specifying Branch and Call Targets

- In assembly language, we indicate (direct) jumps and calls with a symbolic syntax.
  - `JMP Label`
  - `Jcc Label`
  - `CALL Label`
- The machine-level ISA reference specifies these instructions as having an offset operand.

- This offset can be encoded in one of two ways.
  - As an absolute offset ("Go to this address").
  - As a relative offset ("Go so far from the address of the current instruction") — actually, from the address of *the sequential successor of* the current instruction.

Example: PC-Relative Offset

```
loop:   movq    %rdi, %rax
        jmp     .L2
.L3:    sarq    %rax
.L2:    testq   %rax, %rax
        jg      .L3
```

Example: PC-Relative Offset

```
loop:   movq    %rdi, %rax      0:   48 89 F8   movq    %rdi, %rax
        jmp     .L2             3:   EB 03      jmp     8 <loop+0x8>
.L3:    sarq    %rax            5:   48 D1 F8   sarq    %rax
.L2:    testq   %rax, %rax      8:   48 85 C0   testq   %rax, %rax
        jg      .L3             B:   7F F8      jg      5 <loop+0x5>
                                D:
```

Example: PC-Relative Offset

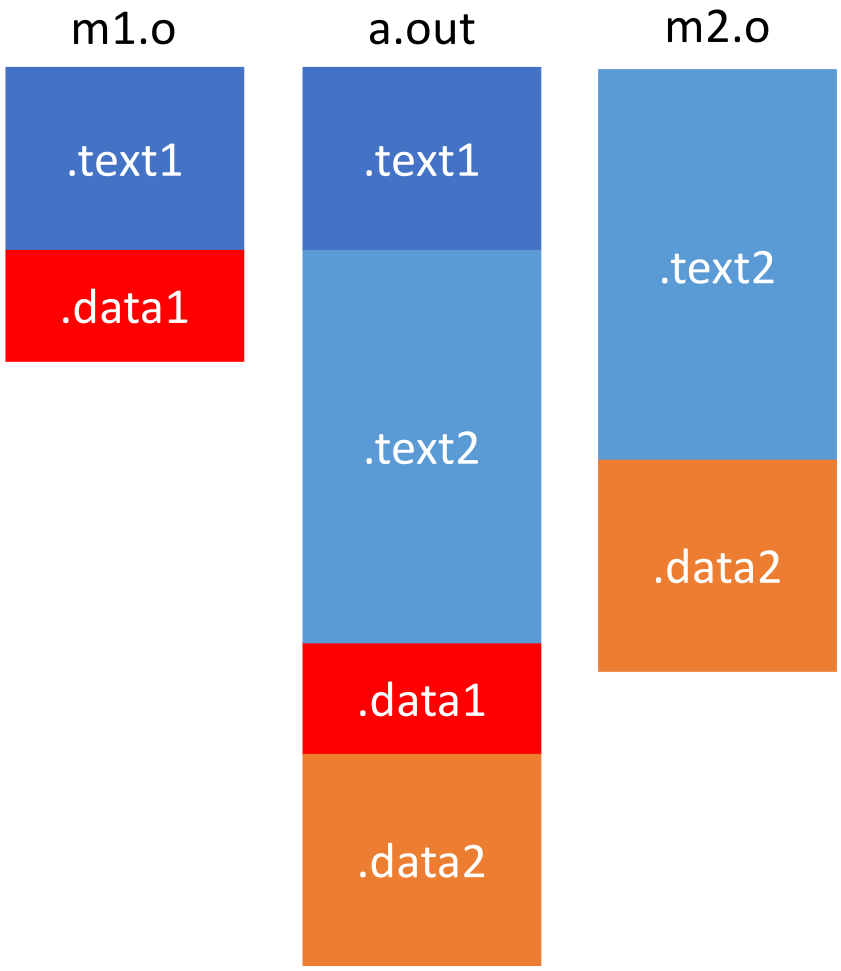
```
loop:   movq    %rdi, %rax   0:   48 89 F8   movq    %rdi, %rax
        jmp     .L2          3:   EB 03      jmp     8 <loop+0x8>  0x5+0x03 = 0x8
.L3:    sarq    %rax         5:   48 D1 F8   sarq    %rax
.L2:    testq   %rax, %rax   8:   48 85 C0   testq   %rax, %rax
        jg      .L3          B:   7F F8      jg      5 <loop+0x5>  0xD+0xF8 = 0x5
                             D:
```

Example: PC-Relative Offset

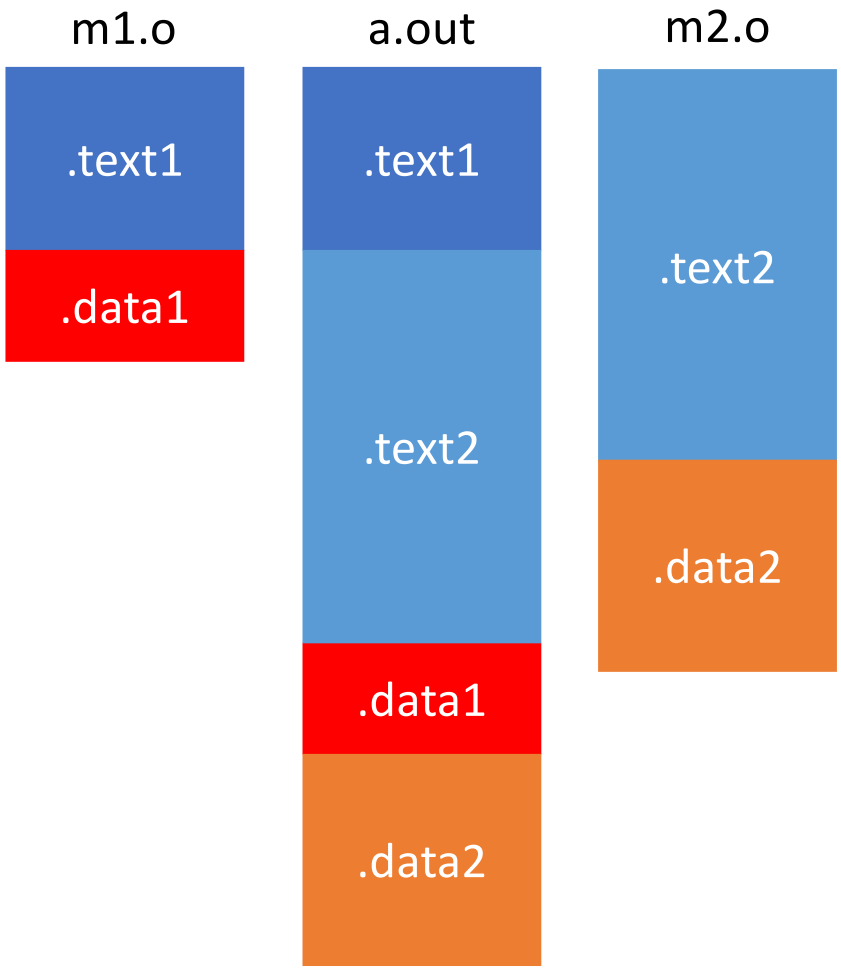
```
loop:   movq    %rdi, %rax   0:   48 89 F8   movq    %rdi, %rax
        jmp     .L2          3:   EB 03      jmp     8 <loop+0x8>  0x5+0x03 = 0x8
.L3:    sarq    %rax         5:   48 D1 F8   sarq    %rax
.L2:    testq   %rax, %rax   8:   48 85 C0   testq   %rax, %rax
        jg      .L3          B:   7F F8      jg      5 <loop+0x5>  0xD+0xF8 = 0x5
                             D:
```

- The encoding remains unchanged even if the numerical value of `loop` changes.
- This is the most basic form of Position-Independent Code (PIC).
  - The non-sequential control transfer edge is “frozen” at compile-time and can be freely relocated to any memory address.
  - The offsets in the two jump instructions do not need to be patched at link-time.

Which Edges Are Position-Independent?

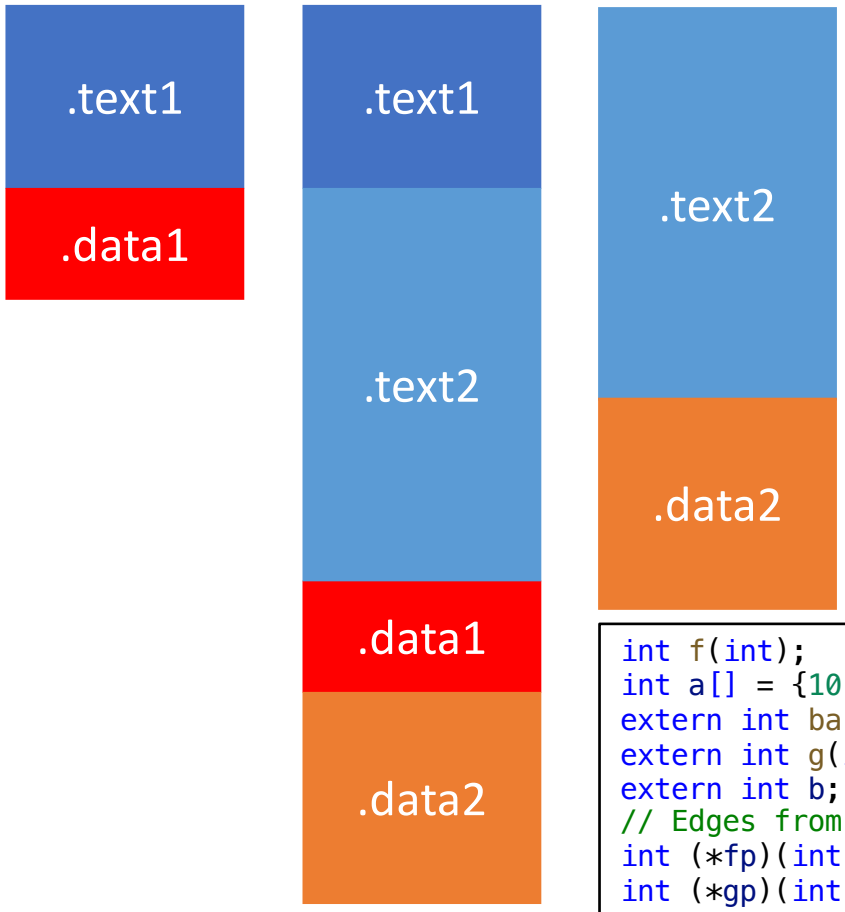


Which Edges Are Position-Independent?



From	To	Example	PIC?
.text1	.text1	Intra-module call	Yes
.text1	.text2	Inter-module call	No
.text1	.data1	Ref to global data	No
.text1	.data2	Ref to external data	No
.data1	.text1	Init to global fn ptr	No
.data1	.text2	Init to external fn ptr	No
.data1	.data1	Init to global data	Yes
.data1	.data2	Init to external data	No

Which Edges Are Position-Independent?



From	To	Example	PIC?
.text1	.text1	Intra-module call	Yes
.text1	.text2	Inter-module call	No
.text1	.data1	Ref to global data	No
.text1	.data2	Ref to external data	No
.data1	.text1	Init to global fn ptr	No
.data1	.text2	Init to external fn ptr	No
.data1	.data1	Init to global data	Yes
.data1	.data2	Init to external data	No

m1.c

```
int f(int);
int a[] = {10, 20};
extern int bar(int*, int);
extern int g(int);
extern int b;
// Edges from .data1
int (*fp)(int) = f; // To .text1
int (*gp)(int) = g; // To .text2
int *ap = a; // To .data1
int *bp = &b; // To .data2
int f(int n) {return n*n;}
// Edges from .text1 to .data1 and to .text2
static int foo(void) {return bar(a, 2);}
// Edges from .text1 to .data2 and to .text1
// Also from .text1 through .data1 to .text1, .text2,
.data1, and .data2
int main(void) {return b+foo()+fp(*ap)+gp(*bp);}
```

m2.c

```
int b = 42;
int bar(int *a, int n) {
int sum = 0;
for (int i = 0; i < n; i++) sum += a[i];
b -= sum;
return sum;
}
int g(int n) {return 2*n+1;}
```

Why PIC Matters

- Consider a large library, e.g., the C standard library `libc`.
  - The library exposes a large number of definitions.
  - Any single program likely references a very small number of these definitions (e.g., file I/O but not signals). However, there may be a lot of references to these definitions.

Why PIC Matters

- Consider a large library, e.g., the C standard library `libc`.
  - The library exposes a large number of definitions.
  - Any single program likely references a very small number of these definitions (e.g., file I/O but not signals). However, there may be a lot of references to these definitions.
- If we link `libc` *statically* with application programs, we resolve and patch these references prior to load-time.
  - The entire library is embedded into each executable, increasing its size on disk and its virtual memory footprint.
  - The OS has to manage many distinct copies of the library, putting a greater load on the VM system.
    - In principle, at least the read-only segment could be shared among all executables linking against `libc` that are running simultaneously.

Why PIC Matters

- Consider a large library, e.g., the C standard library `libc`.
  - The library exposes a large number of definitions.
  - Any single program likely references a very small number of these definitions (e.g., file I/O but not signals). However, there may be a lot of references to these definitions.
- If we link `libc` *statically* with application programs, we resolve and patch these references prior to load-time.
  - The entire library is embedded into each executable, increasing its size on disk and its virtual memory footprint.
  - The OS has to manage many distinct copies of the library, putting a greater load on the VM system.
    - In principle, at least the read-only segment could be shared among all executables linking against `libc` that are running simultaneously.
- If we link `libc` *dynamically* with application programs, we defer patching the references to load-time (or possibly even run-time).
  - Multiple references to the same `libc` definition still need to be patched individually, resulting in a time overhead. Can we mitigate this overhead?
  - Can we also arrange to share a single copy of the read-only segment?