Systems Programming

Code Optimization and Linking

Textbook coverage:
Ch 5: Optimizing Program Performance
Ch 7. Linking

Byoungyoung Lee
Seoul National University
byoungyoung@snu.ac.kr
https://lifeasageek.github.io

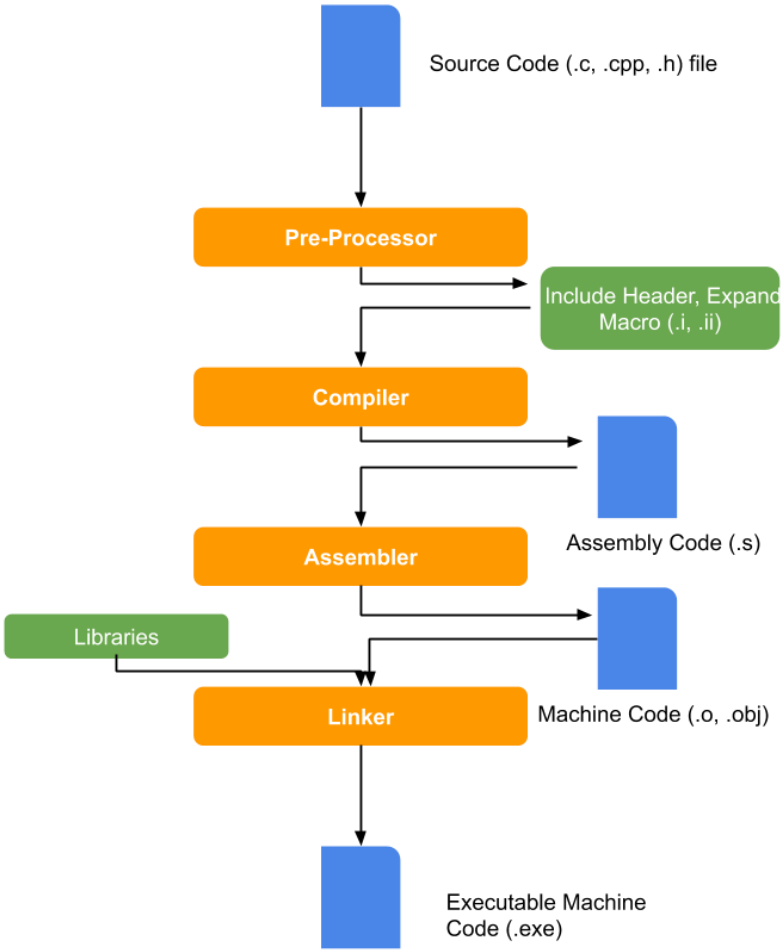
Lecture slides are prepared based on materials provided by CSAPP authors.

Today

- **Basics of compiler optimization**
  - Principles and goals
  - Some example optimizations
  - Obstacles to optimization
- **Linking: combining object files into programs**
  - Symbols and symbol resolution
  - Relocation
  - Static libraries
  - Dynamic libraries

What does it mean to compile code?

- **The CPU only understands *machine code* directly**
- **All other languages must be either**
  - ***interpreted:*** executed by software
  - ***compiled:*** translated to machine code by software



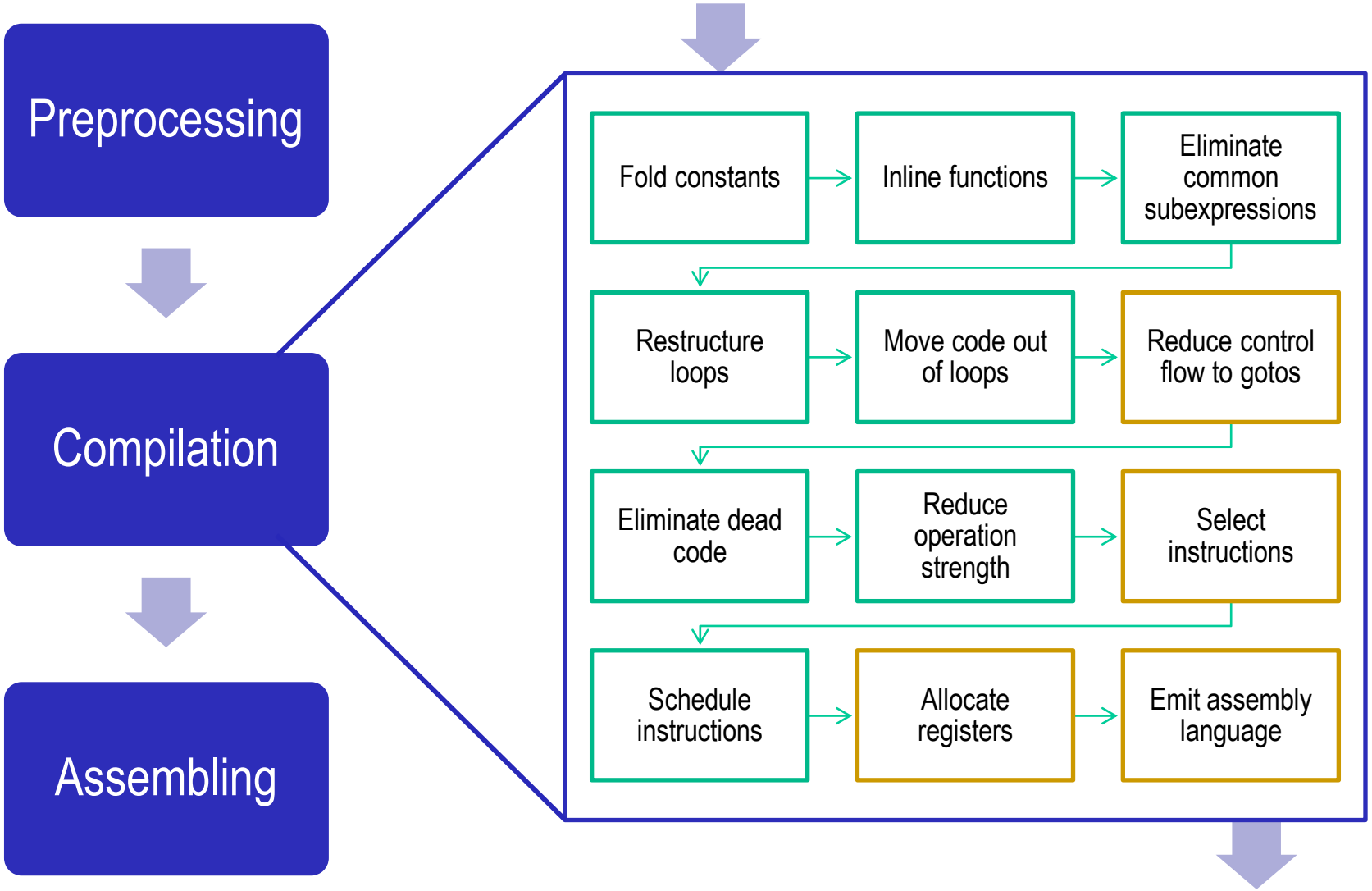
Goals of compiler optimization

- **Minimize number of instructions**
  - Don’t do calculations more than once
  - Don’t do unnecessary calculations at all
  - Avoid slow instructions (division; multiplication is cheap on modern CPUs)
- **Avoid waiting for memory**
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns
  - Load data from memory early, and only once
- **Avoid branching**
  - Don’t make unnecessary decisions/branches at all
  - Make it easier for the CPU to predict branch destinations
  - “Unroll” loops to spread cost of branches over more instructions

Limits to compiler optimization

- **Generally cannot improve algorithmic complexity**
  - Only constant factors, but those can be worth 10x or more for some cases
- **Must not cause *any* change in program behavior**
  - Programmer expect the program runs as they developed/tested
  - Note: language may declare some changes acceptable
    - e.g., “Undefined behavior” (signed integer overflow)
- **Usually only analyze one function at a time**
  - Whole-program analysis is expensive, so it is opt-in: -flto (LTO) is now routine for the Linux kernel, Chrome, Firefox
- **Cannot know the run-time inputs**
  - “Worst case” performance can be just as important as “normal case”
  - Especially for code exposed to *attacker-controlled* input (e.g. network servers, cryptocurrency networks)
  - Exception: profile-guided optimization (PGO, -fprofile-use) feeds measured runs back into the compiler

Compilation is a pipeline



Two kinds of optimizations

- **Local optimizations**
  - Work inside a single *basic block*
  - Constant folding, strength reduction, (local) CSE, …
- **Global optimizations**
  - Process the entire *control flow graph* of a function
  - Loop nest optimization, code motion, (global) CSE, dead code elimination, …

Constant Folding

- **Do arithmetic in the compiler**

```
long mask = 0xFF << 8;
```

➔

```
long mask = 0xFF00;
```

- **Any expression with constant inputs can be folded**

- **Might even be able to remove library calls…**

```
size_t namelen = strlen("Harry Bovik");
```

➔

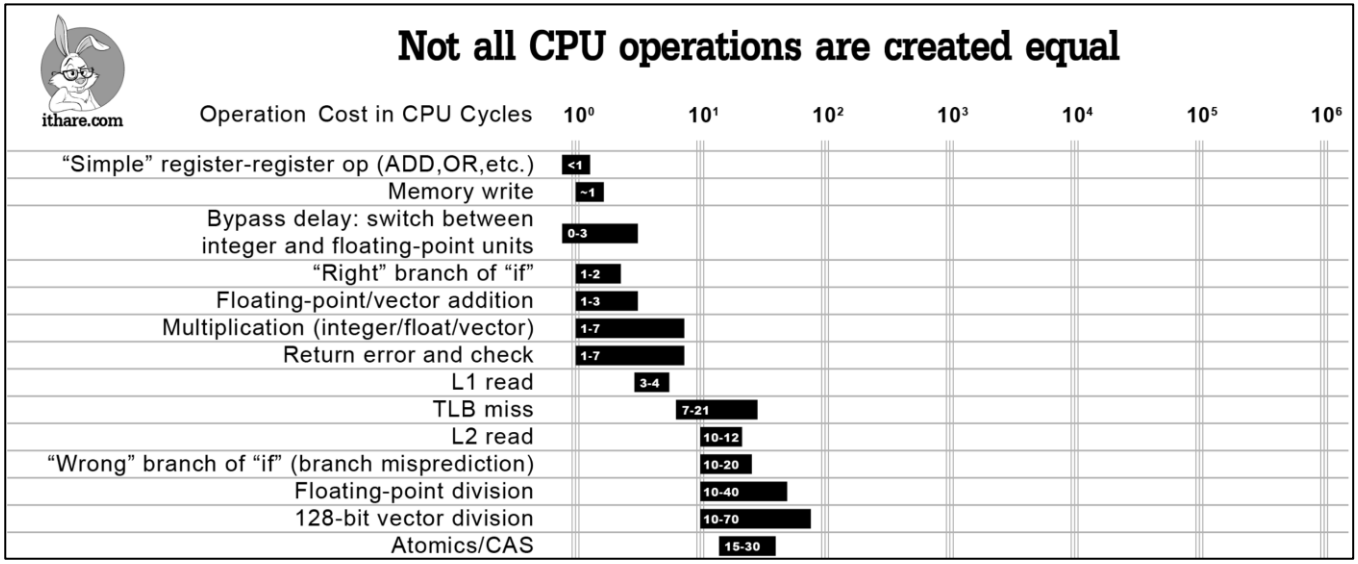
```
size_t namelen = 11;
```

Strength reduction

- **Replace expensive operations with cheaper ones**

```
   long a = b * 5;
➔  long a = (b << 2) + b;
```

- **Multiplication and division are the usual targets**



http://ithare.com/wp-content/uploads/part101_infographics_v08.png

Dead code elimination

- **Don't emit code that will never be executed**

```
if (0) { puts("Kilroy was here"); }
if (1) { puts("Only bozos on this bus"); }
```

- **Don't emit code whose result is overwritten**

```
x = 0;
x = 23;
```

- **These may look silly, but...**
  - Can be produced by other optimizations
  - Assignments to x might be far apart
  - Involve complex control-flows/data-flows

Common Subexpression Elimination (CSE)

- **Factor out common expression, only evaluate them once**

```
norm[i] = v[i].x * v[i].x + v[i].y * v[i].y;
```



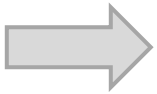
```
elem = &v[i];
x = elem->x;
y = elem->y;
norm[i] = x*x + y*y;
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```



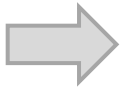
```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```



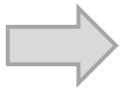
```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Always true **Does nothing** **Can constant fold**

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```



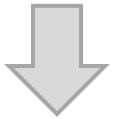
```
int func(int y) {
  int tmp = 0;
  if (y != 0) tmp = y - 1;

  if (y != -1) tmp += y;
  return tmp;
}
```

Code Motion

- **Move calculations out of a loop**
- **Only valid if every iteration would produce same result**
  - So called “loop invariant”

```
long j;
for (j = 0; j < n; j++)
    a[n*i+j] = b[j];
```

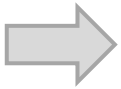


```
long j;
int n_i = n*i;
for (j = 0; j < n; j++)
    a[n_i+j] = b[j];
```

Loop Unrolling

- Amortize cost of loop condition by duplicating body
- Creates opportunities for CSE, code motion, etc.
- Can hurt performance by increasing code size
- Needs a remainder loop; and beware size_t: nelts - 4 underflows when nelts < 4

```
for (size_t i = 0; i < nelts; i++) {
    A[i] = B[i]*k + C[i];
}
```



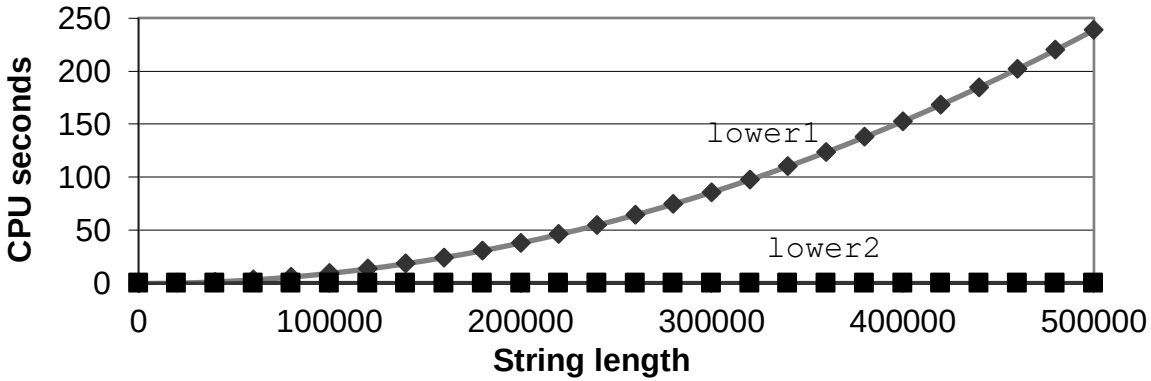
```
size_t i;
for (i = 0; i + 4 <= nelts; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
for (; i < nelts; i++)   /* remainder */
    A[i] = B[i]*k + C[i];
```

When the compiler can't move something

Q. What's the asymptotic complexity of lower1() and lower2()?
Q. Can the compiler optimize the code from lower1() to lower2()?

```
void lower1(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
void lower2(char *s)
{
  size_t i, n = strlen(s);
  for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```



Question: Undefined Behavior

```
1  long long_cmp_opt(const int a, const int b)
2  {
3    if (a > 0) {                    // a is positive
4      do_something();
5      if (b < 0) {                  // b is negative
6        do_something_else();
7        if ((a - b) > 0)            // always true or can it be false?
8          do_another_thing();
9      }
10   }
11 }
```

Q1. Is the condition check at line 7 always true?
Q2. Should compiler remove the condition check at line 7?

A. Signed overflow optimization hazards in the kernel (https://lwn.net/Articles/511259/)

Design Rationale:
behind Undefined Signed Integer Overflow

- Enable aggressive optimization!

```
// A naive attempt to check if x + 100 overflows
bool is_safe_to_add(int x) {
    if (x + 100 < x) {
        return false; // Overflow happened! (e.g., INT_MAX + 100 is negative)
    }
    return true;      // Safe
}
```

```
// What the compiler actually turns your code into:
bool is_safe_to_add(int x) {
    return true;
}
```

Design Rationale: behind Undefined Signed Integer Overflow

- Enable aggressive optimization!

```
int count_iterations(int start) {
    int loops = 0;
    for (int i = start; i < start + 10; i++) {
        loops++;
    }
    return loops;
}
```

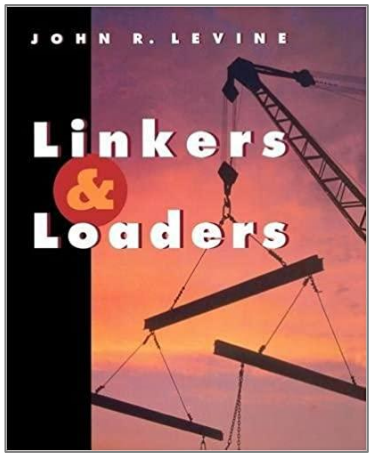
```
// What the compiler actually turns your code into:
int count_iterations(int start) {
    return 10;
}
```

Compiler vs. LLM Optimization

- **A compiler never changes your algorithm and never changes behavior (except where the language says “undefined”)**
- **An LLM will happily rewrite bubble sort into quicksort, and guarantees nothing about behavior**
- **Everything in the last dozen slides is what -O2 already does for you**
  - An LLM hand-applying strength reduction, CSE, or unrolling is wasted diff; measure -O2 first
  - Spend human (or LLM) effort where compilers cannot: algorithms, data layout, the strlen-in-the-loop bug
- **LLM-written C is full of undefined behavior**
  - Overflow checks like if (x + 100 < x), size_t i; … i >= 0 loops, shifts >= width
  - Passes the tests at -O0, then “breaks” at -O2: it did not break, the UB was always there
- **Verify: -O2 -Wall -Wextra -fsanitize=undefined, and benchmark before accepting a “faster” version**

Today

- **Basics of compiler optimization**
  - Principles and goals
  - Some example optimizations
  - Obstacles to optimization
- **Linking: combining object files into programs**
  - Symbols and symbol resolution
  - Relocation
  - Static libraries
  - Dynamic libraries



Linkers and Loaders, The Morgan Kaufmann Series in Software Engineering and Programming, 1st Edition, John R. Levine

Example C Program

- Two source files

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

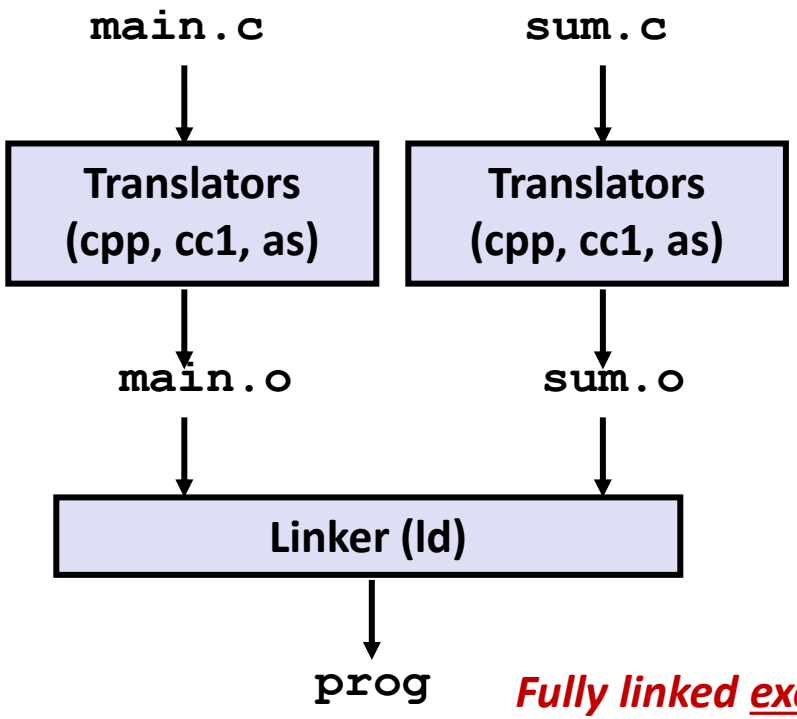
```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Linking

- **Programs are translated and linked using a *compiler driver*:**
  - `$ gcc -Og -o prog main.c sum.c`
  - `$ ./prog`



Source files

Separately compiled relocatable object files

Fully linked executable object file

Why should you know “Linking”

- **To maintain a build system for a large-scale project**
  - The Linux kernel? PyTorch? llama.cpp? vLLM?
  - To write a complex Makefile/CMake, or to read the “undefined reference” errors from the one a coding agent wrote, you need to understand the things behind the scene
- **To clearly understand the important systems concepts**
  - Shared libraries
  - Loading
  - Memory mapping
  - Virtual memory
  - The lifetime of “process”

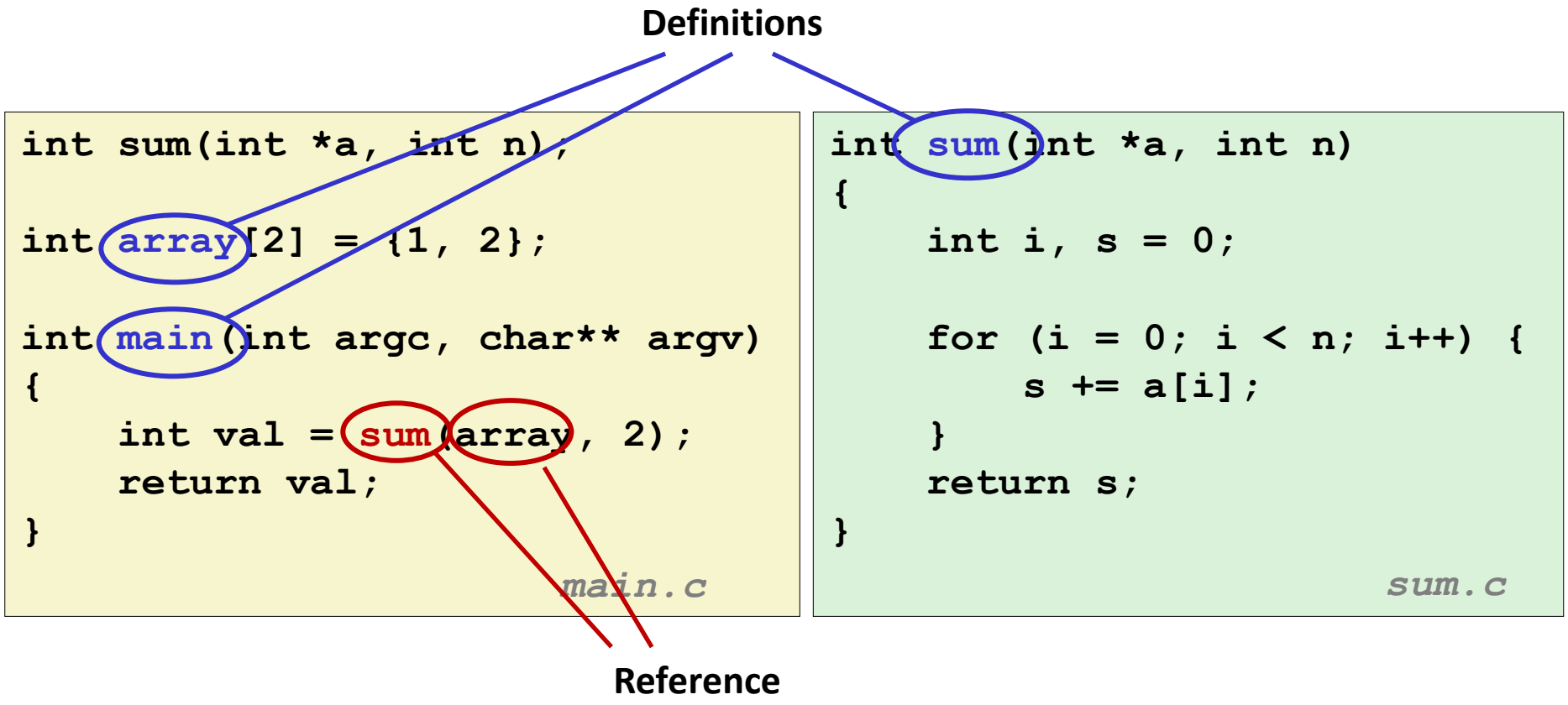
What Do Linkers Do?

- Step #1: Constructing symbol tables (assembler)
- Step #2: Symbol resolution (linker)
- Step #3: Relocation (linker)

Step #1: Constructing symbol tables

- **Step #1: Constructing symbol tables**
  - **Symbol**
    - Definition and references of global variables and functions
      - `void swap() {…} /* define symbol swap */`
      - `swap();  /* reference symbol swap */`
      - `int *xp = &x;  /* define symbol xp, reference x */`
  - **Symbol table**
    - Symbol table stores symbol definitions, which are an array of entries
    - Each entry includes symbol name, size, and location of symbol.
  - Symbol resolution: the linker associates each symbol reference with the symbol definition

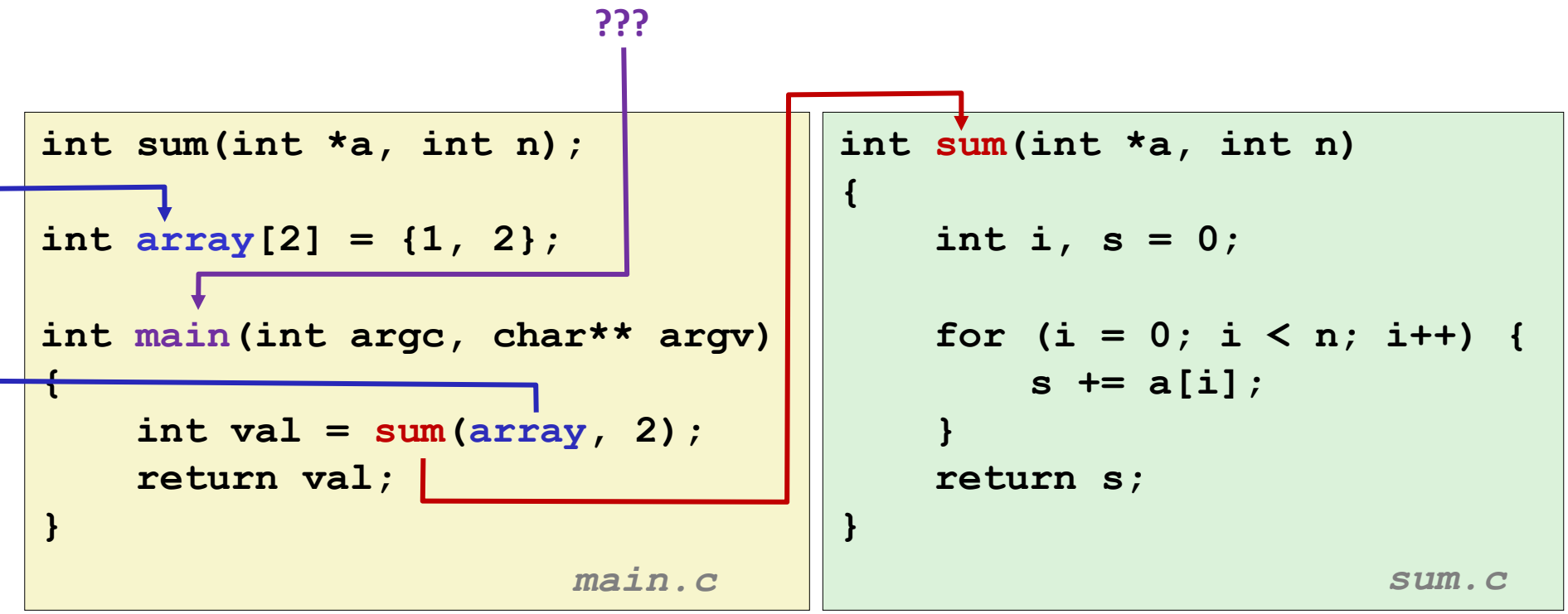
Symbols in Example C Program



Linker Symbols

- **Every object file *m* has a table of symbols it defines or needs.**
- **Three types of symbols**
  - Global definitions
    - Symbols defined by *m* that can be referenced by other files.
    - In C, non-**`static`** functions and global variables.
  - Local definitions
    - Symbols that are defined by *m* but *cannot* be referenced by other files.
    - In C, functions and global/local variables defined with **`static`**.
    - **Note: this is different from local variables (in stack)**
  - External references
    - Symbols that *m* references but does not define.
    - These must be defined by some other module.

Step #2: Symbol Resolution



Strong vs. Weak Symbols

- **What if several object files define the same global symbol?**
- **Strong symbols: functions and initialized global variables**
- **Weak symbols: uninitialized global variables (int x;), historically "common" symbols**
- **Linker rules**
  - Multiple strong symbols with the same name: link error (multiple definition)
  - One strong + any number of weak: the strong one wins
  - Only weak ones: pick any, so int x; in one file and double x; in another silently share memory

Relocation Entries in a object file

```c
int array[2] = {1, 2};

int main(int argc, char**
argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

Each relocation entry instructs how each reference should be relocated using the symbol table.

Output from gcc 13 on Ubuntu 24.04, which builds PIE by default: array is referenced PC-relative (lea) and sum through the PLT

$ objdump -r -d main.o

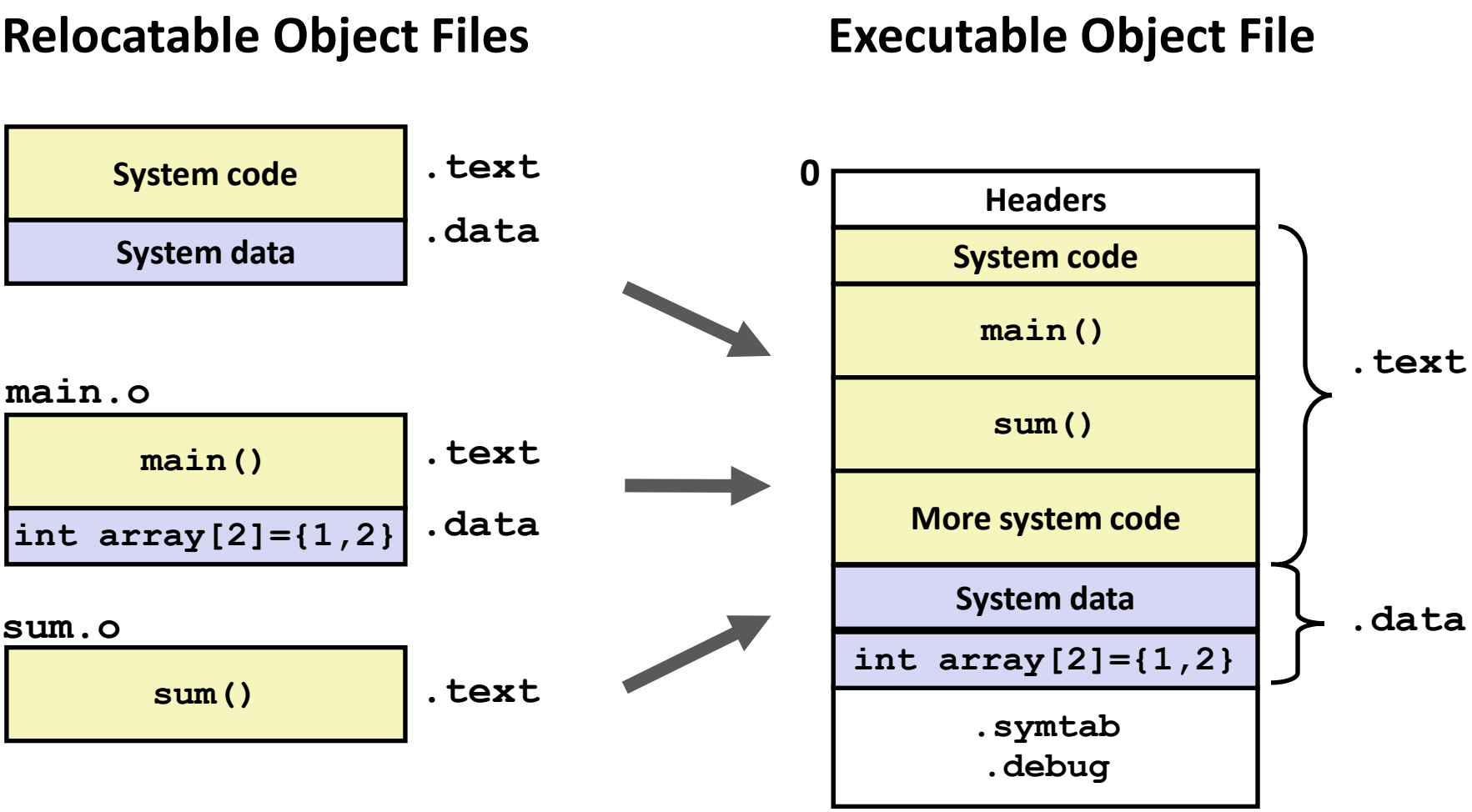
```
0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   48 83 ec 08             sub    $0x8,%rsp
   8:   be 02 00 00 00          mov    $0x2,%esi
   d:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi # %rdi = &array
                        10: R_X86_64_PC32 array-0x4   # Relocation entry
  14:   e8 00 00 00 00          call   19 <main+0x19>  # sum()
                        15: R_X86_64_PLT32 sum-0x4    # Relocation entry
  19:   48 83 c4 08             add    $0x8,%rsp
  1d:   c3                      ret
```

Step #3. Relocation

- **Step #3. Relocation**

  - Takes place when linking
    - e.g., merging multiple object files (with multiple code/data sections) into a single executable file (with single code/data section)
  - Relocate symbols:
    - from their relative locations in the `.o` files
    - to their final memory locations in the executable
    - With PIE (today's default) "final" means relative to the load base; the dynamic loader adds the base at run time

Step #3: Relocation



Relocated .text section (executable)

```
$ objdump -d prog
```

```
0000000000001129 <main>:
    1129:  f3 0f 1e fa           endbr64
    112d:  48 83 ec 08           sub    $0x8,%rsp
    1131:  be 02 00 00 00        mov    $0x2,%esi
    1136:  48 8d 3d d3 2e 00 00  lea    0x2ed3(%rip),%rdi  # 4010 <array>
    113d:  e8 05 00 00 00        call   1147 <sum>         # sum()
    1142:  48 83 c4 08           add    $0x8,%rsp
    1146:  c3                    ret
0000000000001147 <sum>:
    1147:  f3 0f 1e fa           endbr64
    114b:  ba 00 00 00 00        mov    $0x0,%edx
    1150:  b8 00 00 00 00        mov    $0x0,%eax
    1155:  eb 09                 jmp    1160 <sum+0x19>
    1157:  48 63 c8              movslq %eax,%rcx
    115a:  03 14 8f              add    (%rdi,%rcx,4),%edx
    115d:  83 c0 01              add    $0x1,%eax
    1160:  39 f0                 cmp    %esi,%eax
    1162:  7c f3                 jl     1157 <sum+0x10>
    1164:  89 d0                 mov    %edx,%eax
    1166:  c3                    ret
```

`call` uses PC-relative addressing for sum(): `0x1147 = 0x1142 + 0x5`

`lea` does too, for array (PIE: no absolute addresses): `0x4010 = 0x113d + 0x2ed3`

Libraries: Packaging a Set of Functions

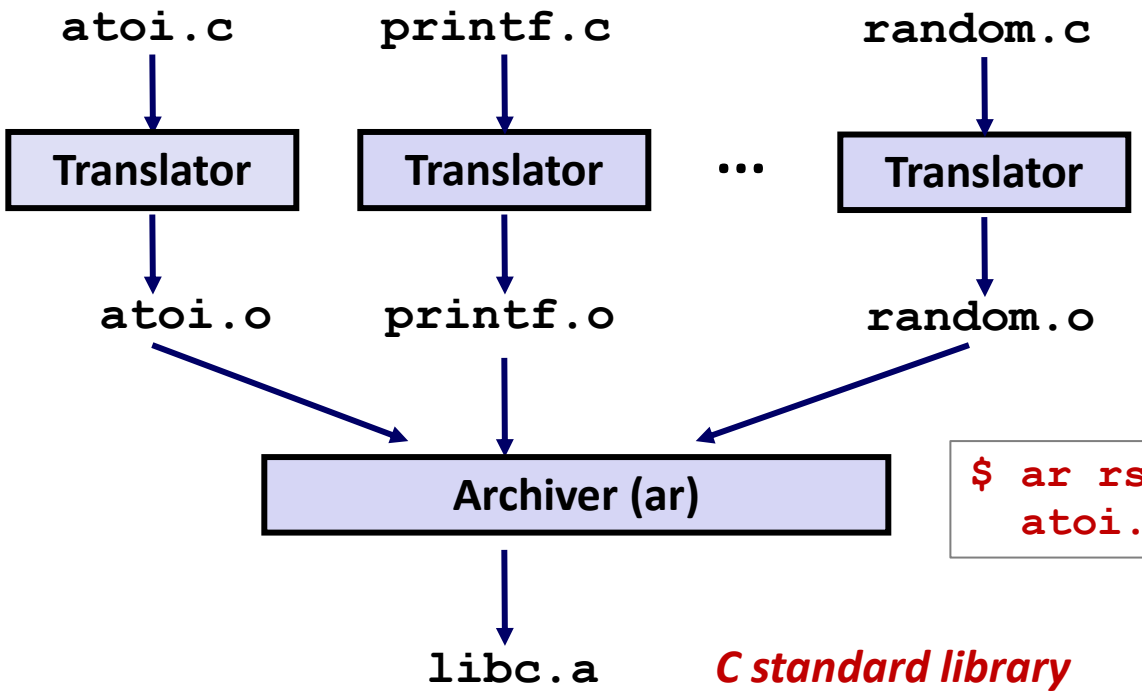
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **Awkward, given the linker framework so far:**
  - **Option 1:** Put all functions into a single object file (e.g., a single lib file)
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate object file (e.g., multiple lib files)
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Static Libraries

- **Static libraries (`.a` archive files)**
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Linker resolves unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Creating Static Libraries




```
$ ar rs libc.a \
  atoi.o printf.o … random.o
```

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

`libc.a` (the C standard library)

- 6.1 MB archive of 2211 object files (Ubuntu 24.04, glibc 2.39)
- I/O, memory allocation, signal handling, string handling, date and time, random numbers, integer math

`libm.a` (the C math library)

- 2.6 MB archive of 801 object files (libm.a itself is now a linker script pointing to libm-2.39.a)
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
$ cd /usr/lib/x86_64-linux-gnu
$ ar -t libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
…
```

```
$ cd /usr/lib/x86_64-linux-gnu
$ ar -t libm-2.39.a | sort
…
e_acosf128.o
e_acosf.o
e_acoshf128.o
e_acoshf.o
e_acoshl.o
e_acosh.o
e_acosl.o
e_acos.o
…
```

Linking with Static Libraries

```c
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char**
argv)
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n",
           z[0], z[1]);
    return 0;
}
```

main2.c

libvector.a

```c
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

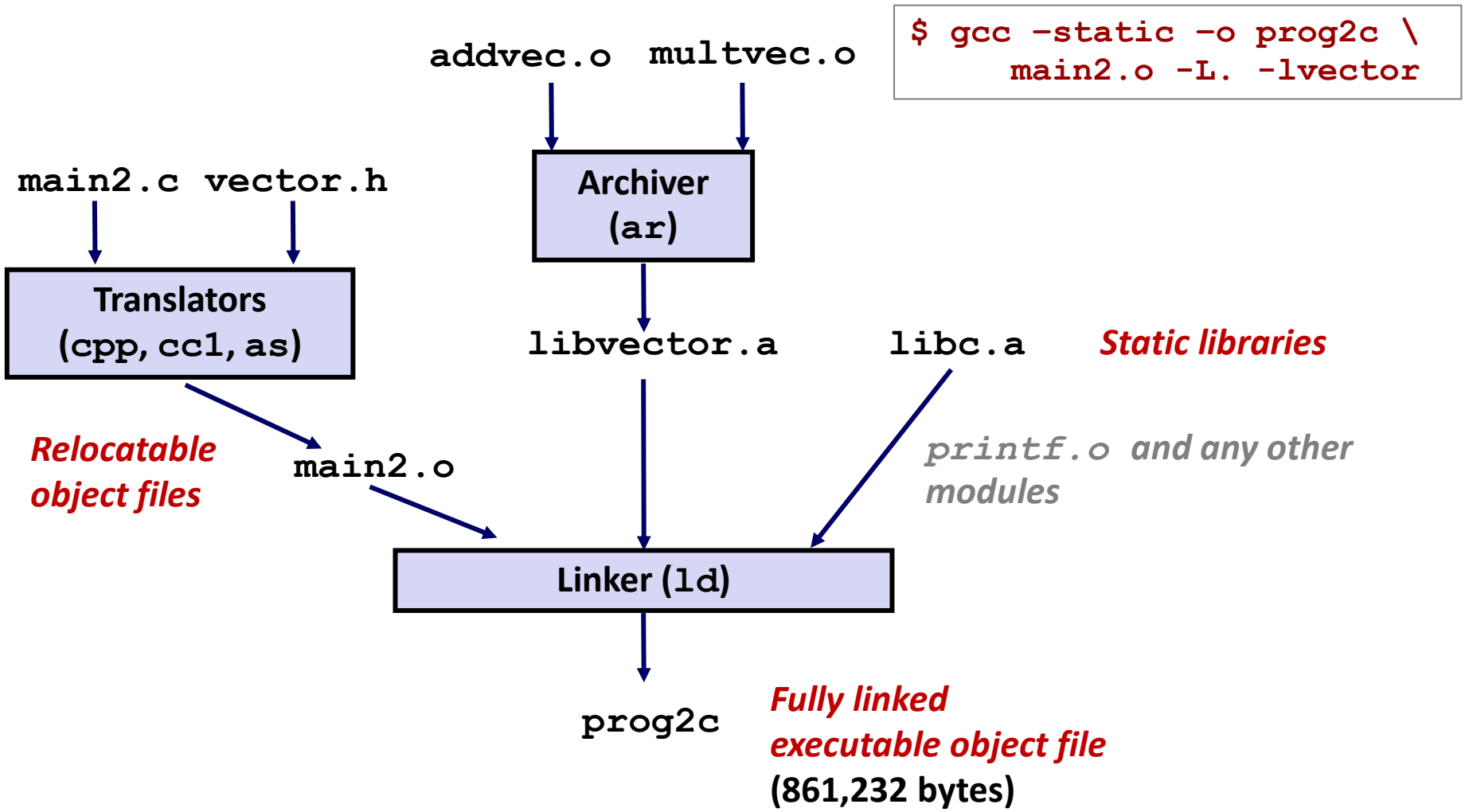
addvec.c

```c
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c

Linking with Static Libraries



"c" for "compile-time"

Shared Libraries

- **Limitations of Static libraries**
  - Duplication in the linked executables (every program needs libc)
  - Duplication in the running executables
    - Each statically linked program carries its own copy of libc in memory; nothing is shared across processes
  - Minor bug fixes of system libraries require each application to explicitly relink
    - Stack overflow in glibc's getaddrinfo() (CVE-2015-7547)
      - If linking glibc statically, an executable has to be rebuilt
      - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

- **Shared libraries (or dynamic library, or dynamic linking)**
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: DLLs, `.so` files

Shared Libraries (cont.)

- **Load-time linking: Dynamic linking can occur when executable is first loaded and run**
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`)
  - Standard C library (`libc.so`) is usually dynamically linked
  - Calls into a shared library go through the PLT/GOT: bound lazily on first call, or all at load time (-z now, the Ubuntu default, so lazy binding is off)
- **Run-time linking: Dynamic linking can also occur after program has begun**
  - In Linux, this is done by calls to the `dlopen()` interface
  - dlsym() looks up a symbol by name: plugins, JIT engines, the CUDA kernels PyTorch loads

Case Study: the xz Backdoor

- **March 2024: xz-utils 5.6.0/5.6.1 shipped a backdoor in liblzma (CVE-2024-3094), planted by a maintainer over two years**
- **Target: sshd. But sshd never calls liblzma…**
  - Debian and Fedora patch sshd to link libsystemd, and libsystemd links liblzma
  - So ld-linux.so loads the malicious liblzma into every sshd process
- **The hook: liblzma's IFUNC resolver runs during symbol resolution, before main(), and redirected OpenSSL's RSA_public_decrypt to the backdoor**
- **Found by accident: a PostgreSQL developer noticed ssh logins took 500 ms longer and profiled sshd**
- **Lesson: every transitively linked .so runs code inside your process before main(); ldd, nm, and the GOT are how you audit what got in**

Static Linking vs. Dynamic Linking

- **Static Linking (.a archives)**
  - Library code is copied into the executable at compile time
  - Executable is self-contained: no external dependencies at runtime
  - Larger binaries; every program gets its own copy of library code
  - Library bug fixes require recompiling all affected programs

- **Dynamic Linking (.so / .dll)**
  - Library code is loaded at load-time or run-time, not copied into the executable
  - All programs share one copy in memory, saving disk space and RAM
  - Updating a shared library patches all programs that use it
  - Executable depends on the correct library version being present at runtime

- **Key tradeoff: portability (static) vs. efficiency (dynamic)**

What dynamic libraries are required?

- **Use "`ldd`" to find out:**

```
$ ldd prog
    linux-vdso.so.1 (0x00007ffebb5be000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000758ef6a00000)
    /lib64/ld-linux-x86-64.so.2 (0x0000758ef6d37000)
```

Caution: ldd works by actually running the dynamic loader on the binary

- Never run it on an untrusted executable; use objdump -p prog | grep NEEDED instead
- Model checkpoints, CTF binaries, and "helper" tools from the internet all count as untrusted