Whole-program optimization

- dividing a program into subroutines is good and bad froma purely compiler perspective
- good: it allows compilation of subroutines individually
- bad: adds the call/return overhead to code/execution
- bad: the subroutine abstractions make it difficult to optimize what happens inside the subroutine with what happens outside the subroutine
- for optimizations, will look at inlining functions and at subroutine placement in the executable

inlining functions

- replace a function call with the actual body of the function, with suitable substitutions for the variables/parameters

```
// original
int f(int a, int b) {
   int sum=0, x=a;
   while (x < b)
      sum += x++;
   return sum;
}
...inside caller...
result = foo(i,j);
```

```
// revised caller segment with new var
// names that don't clash with rest

int f_sum=0, f_x=i;

while (f_x < j)
   f_sum += f_x++;

result = f_sum
```

inlining pros/cons

- good: the inlined code can present better opportunities for local optimization
- good: the inlined code runs faster (no call/return)
- bad: the inlined code produces a larger executable
- bad: the inlined code generates a larger number of temporary variable names (can be troublesome when it comes to mapping to registers)

heuristics: when should it inline?

- might lean towards inlining if:
  - callee is smaller thant the cost of call/return
  - the call is made frequently, e.g. within nested loops
  - the callee is only called from one place
  - constant valued parameters used (inline for constant folding)
  - callee is a leaf
  - callee represents significant amount of total execution time
- avoid inlining if there isn't a good reason to inline, and avoid inlining if the caller is already large

Subroutine placement

- much like the block placement section of subroutine optimization
- if function f calls function g, try to place them next to one another in the executable to improve paging
- with lots of functions calling each other it is generally difficult to make this work for all caller/callee pairs
- will perform an analysis phase using a call graph, then use that information to determine the order of subroutines in the executable

Analysis phase: build call graph

- nodes in our graph represent subroutines
- directed edge from n1 to n2 where n1 calls n2
- the weight of each edge will be estimate of how frequently that call is performed
- ignore recursive calls
- if there are multiple edges from ni to nj (i.e. function f contains multiple calls to function g) combine them into a single edge and add up their edge weights

Analysis algorithm

queue all the weighted edges, highest weights first

give each node, n, a placement list initialized to { n }

while the queue of edges isn't empty:

    take next edge, e = (x,y):w from the queue

    for each remaining edge of form (y, z)

        replace it with edge (x,z), keeping same weight

    for each remaining edge of form (z,y)

        replace it with edge (z,x), keeping same weight

    append y's placement list to the end of x's

    delete edge (x,y) and node y from the graph

When to do whole-prog optimize?

- from compiler optimization perspective, seperating compilation then linking is a bad thing
  - whole program optimization easiest if it's one big file (but that's not going to happen in modern software development)
- if compiler is part of an IDE then compiler can be notified whenever anything changes, and update relevant optimization portions
- some whole program optimizations might best be done at link-time, when all the parts are in place together

Algorithm

- note that with each edge we remove from the queue, we delete one node and one edge from the graph, and rewrite any of the other relevant edges
- each of these also moves one placement list onto the end of one of the remaining ones
- thus after N passes (where N is number of nodes) we have reduced it to a single placement list that has all the nodes in it
- by taking most-used edges first we ensure the most-used calls have their functions grouped together