Agenda

Building an application: preprocessing, compiling, and linking

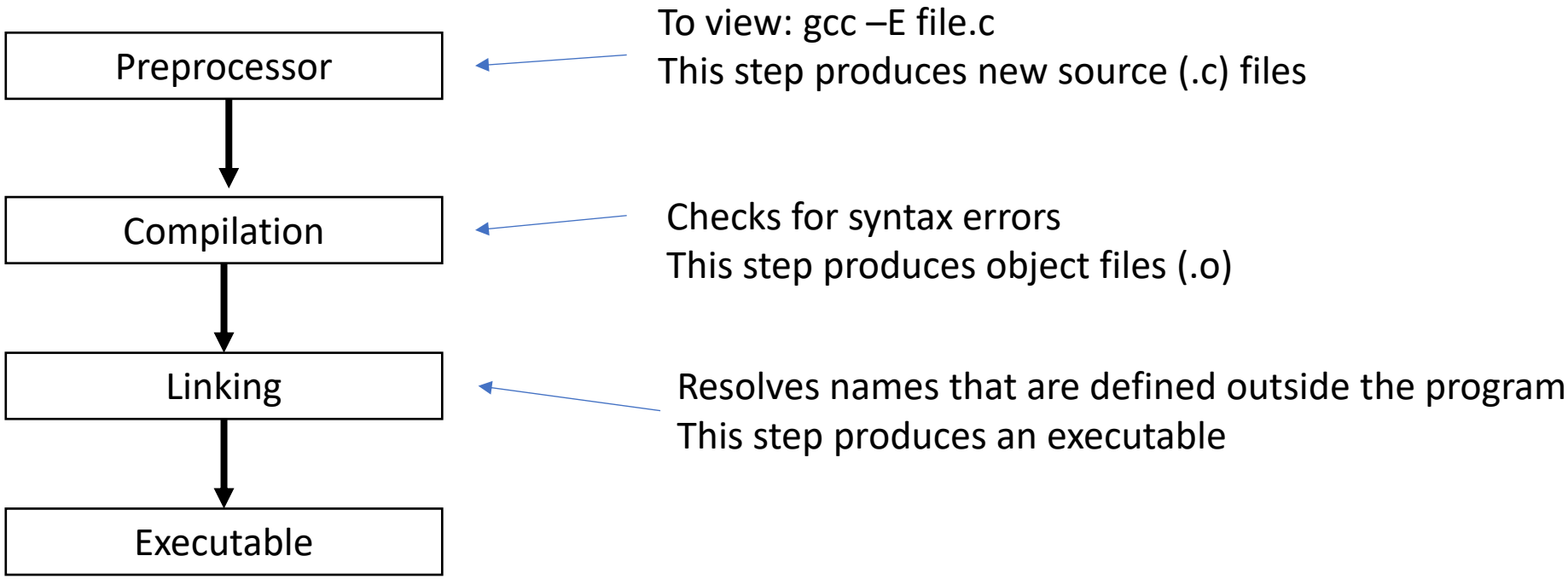
Compiler optimizations

What can’t be optimized?

Profiling tools

Overview: Building an application

Consists of several steps:



Compiler optimization

Perform basic optimizations to

- reduce code size
- reduce execution time

```
$ gcc -O1 -o program program.c
```

- demo

What can compilers optimize?

Constant folding

```
#define N 5
int debug = N - 5;
```

NOTE: Macro Expansion replaces all instances of a macro with its value

What can compilers optimize?

Constant propagation

```
int debug = 0;

//sums up all the elements in an array
int doubleSum(int *array, int length){
   int i, total = 0;
   for (i = 0; i < length; i++){
      total += array[i];
      if (debug) {
         printf("array[%d] is: %d\n", i, array[i]);
      }
   }
   return 2 * total;
}
```

What can compilers optimize?

Dead code elimination

```
int debug = 0;

//sums up all the elements in an array
int doubleSum(int *array, int length){
   int i, total = 0;
   for (i = 0; i < length; i++){
      total += array[i];
      if (0) { //debug replaced by constant propagation by compiler
         printf("array[%d] is: %d\n", i, array[i]);
      }
   }
   return 2 * total;
}
```

What can compilers optimize?

Simplifying expressions

```
//declaration of debug removed through dead-code elimination

//sums up all the elements in an array
int doubleSum(int *array, int length){
   int i, total = 0;
   for (i = 0; i < length; i++){
      total += array[i];
      //if statement removed through data-flow analysis
   }
   return 2*total;
}
```

Examples

- total * 2 -> total + total
- total * 8 -> total << 3
- total % 8 -> total & 7

What can compilers optimize?

Loop unrolling, **-**funroll-loops

Idea: Reduce the number of loop iterations

```
int isPrime(int x) {
   int i;
   int max = sqrt(x)+1;

   // no prime number is less than 2
   for (i = 2; i < max; i+=2) {
      // if the number is divisible by i or i+1
      if ( (x % i == 0) || (x % (i+1) == 0) ) {
         return 0; // it's not prime
      }
   }
   return 1; // otherwise it is
}
```

What can compilers optimize?

Function inlining, -finline-functions

Idea: Replace function calls with their code (remove overhead of function call)

```
int max_val(int x, int y) {
    return x > y? x : y;
}

int main() {
   int a = atoi(argv[1]);
   int b = atoi(argv[2]);
   int value = max_val(a, b);
}
```

```
int main() {
   int a = atoi(argv[1]);
   int b = atoi(argv[2]);
   int value = a > b? a : b;
}
```

What can’t compilers optimize?

Optimize pointer arithmetic

```
void shiftAdd(int *a, int *b){
  *a = *a * 10; //multiply by 10
  *a += *b; //add b
}
```

Why?

Memory aliasing: when two variables point to the same address in memory

Memory aliasing: Function stack example

```
void shiftAdd(int *a, int *b){
  *a = *a * 10; //multiply by 10
  *a += *b; //add b
}
void shiftAddOpt(int *a, int *b){
  *a = (*a * 10) + *b;
}
int main() {
  int x = 5;
  int y = 6;
  shiftAdd(&x, &y);
  printf("shiftAdd produces: %d\n", x);

  x = 5; //reset x
  shiftAddOpt(&x, &y);
  printf("shiftAddOpt produces: %d\n", x); // draw stack here
  return 0;
}
```

Memory aliasing: Function stack example

```
void shiftAdd(int *a, int *b){
  *a = *a * 10; //multiply by 10
  *a += *b; //add b
}
void shiftAddOpt(int *a, int *b){
  *a = (*a * 10) + *b;
}
int main() {
  int x = 5;
  shiftAdd(&x, &x);
  printf("shiftAdd produces: %d\n", x);

  x = 5; //reset x
  shiftAddOpt(&x, &x);
  printf("shiftAddOpt produces: %d\n", x); // draw stack here
  return 0;
}
```

What can’t compilers optimize?

Fix loop-invariant code involving function calls

```
int main() {
  char* test = "hello";
  for (int i = 0; i < strlen(test); i++) {
    printf("%c\n", test[i]);
  }
}
```

Why?

Because functions can have **side-effects**

A **side-effect** is a change that persists after the function completes (ex. strtok modifies its argument)

Because function values might change between iterations

What can’t compilers optimize?

Improve algorithms

Ex: To find primes, the “Sieve of Eratosthenes” algorithm will perform better than a naive algorithm that needlessly recomputes values

Watch out: Limits of -O

Compiling with optimization flags can introduce bugs

- Can’t compile with both debug symbols (-g) and optimization enabled

Why? Undefined behaviors can be handled differently depending on optimization strategy.

```
int silly(int a) {
   int b = a + 1;
   return b > a;
}
```

```
$ gcc silly.c
$ ./a.out
0

$ gcc -O silly.c
$ ./a.out
1
```

Watch out: Limits of -O

Compiling with optimization flags does not always improve performance!

Table 3. Time in Seconds to Produce Prime numbers between 2 and 5,000,000

Unoptimized	-01	-02	-03
3.86	2.32	2.14	2.15

Profiling

Profiling is the process of analyzing the performance of your program

```
$ gcc -o optExample optExample.c -lm

$ time -p ./optExample 5000000
Time to allocate: 5.5e-05
Time to generate primes: 3.85525
348513 primes found.
real 3.85
user 3.86
sys 0.00

$ valgrind --tool=callgrind ./optExample 100000

$ callgrind_annotate --auto=yes callgrind.out.3243456
```

Strategy for performance optimization:

- Identify hot spots, e.g. expensive code that is called often
- Improve performance of the hotspot

Demo: Finding primes

Find the expensive loop in `findPrimes.c`

Code Optimization Best Practices

Prioritize readability

Use built-in functions

Optimize based on performance statistics, not feelings

Split code into small, well-defined functions (facilitates function inlining)

Don’t forget memory