C Primer

CS 351: Systems Programming
Melanie Cornelius

Slides and course content obtained with permission from Prof. Michael Lee, <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Reminders

- Piazza
- Watch for emails from:
  - Fourier
  - IIT VPN
  - GitHub
- CS:APP, read CH 1 and 2

Survey

Agenda

1. Overview
2. Basic syntax & structure
3. Compilation
4. Visibility & Lifetime

Not a Language Course!

- Resources:
  - K&R (*The C Programming Language*)
  - comp.lang.C FAQ (c-faq.com)
  - UNIX man pages (kernel.org/doc/man-pages/)

>man strlen

```
NAME
     strlen - find length of string

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <string.h>

     size_t
     strlen(const char *s);

DESCRIPTION
     The strlen() function computes the length of the string s.

RETURN VALUES
     The strlen() function returns the number of characters that precede the
     terminating NUL character.

SEE ALSO
     string(3)
```

Overview

Language Philosophies

C: “Make it efficient and simple, and let the programmer do whatever she wants”

Java: “Make it portable, provide a huge class library, and try to protect the programmer from doing stupid things.”

C is …

- imperative (state changes)
- statically typed (can’t change)
- weakly type checked (functions between types)
- procedural (step-wise execution)
- low level (pointers but machine abstractions!)

C	Java
Procedural	Object-oriented
Source-level portability	Compiled-code portability
Manual memory management	Garbage collected
Pointers reference addresses	Opaque memory references
Manual error code checking	Exception handling
Manual namespace partitioning	Namespaces with packages
Small, low-level libraries	Vast, high-level class libraries

Basic syntax & structure

Primitive Types

- char: one byte integer (e.g., for ASCII)
- int: integer, *at least* 16 bits
- float: single precision floating point
- double: double precision floating point

Integer type prefixes

- signed (default), unsigned
  - same storage size, but sign bit on/off
- short, long
  - sizeof (short int) ≥ 16 bits
  - sizeof (long int) ≥ 32 bits
  - sizeof (long long int) ≥ 64 bits

Recall C’s weak type-checking…

```
/* types are implicitly “converted” */

char  c  = 0x41424344;
short s  = 0x10001000;
int   i  = 'A';
unsigned int u = -1;

printf("'%c', %d, %X, %X\n", c, s, i, u);
```

```
'D', 4096, 41, FFFFFFFF
```

Basic Operators

- Arithmetic: +, -, *, /, %, ++, --, &, |, ~
- Relational: <, >, <=, >=, ==, !=
- Logical: &&, ||, !
- Assignment: =, +=, *=, …
- Conditional: *bool* **?** *true_exp* **:** *false_exp*

True/False

- 0 = False
- **Everything else** = True
  - But *canonical* True = 1

Boolean Expressions

```
!(0)          → 1
0 || 2        → 1
3 && 0 && 6   → 0
!(1234)       → 0
!!(-1020)     → 1
```

Control Structures

- if-else
- switch-case
- while, for, do-while
  - continue, break

Variables

- Must declare before use
- Declaration implicitly allocates storage for underlying data

Functions

- C’s *top-level* modules
- Procedural language vs. OO: no classes!

Declaration vs. *Definition*

- *Declaration* (aka *prototype*): name + arg & ret types
- *Definition*: declaration + body
- A function can be *declared many times* but only *defined once*

Declarations reside in *header* (.h) files,
Definitions reside in *source* (.c) files

(Suggestions, not really requirements)

hashtable.h

```
unsigned long hash(char *str);
hashtable_t *make_hashtable(unsigned long size);
void  ht_put(hashtable_t *ht, char *key, void *val);
void *ht_get(hashtable_t *ht, char *key);
void  ht_del(hashtable_t *ht, char *key);
void  ht_iter(hashtable_t *ht, int (*f)(char *, void *));
void  ht_rehash(hashtable_t *ht, unsigned long newsize);
int   ht_max_chain_length(hashtable_t *ht);
void  free_hashtable(hashtable_t *ht);
```

← "API"

hashtable.c

```
#include "hashtable.h"



unsigned long hash(char *str) {
  unsigned long hash = 5381;
  int c;
  while ((c = *str++))
    hash = ((hash << 5) + hash) + c;
  return hash;
}

hashtable_t *make_hashtable(unsigned long size) {
  hashtable_t *ht = malloc(sizeof(hashtable_t));
  ht->size = size;
  ht->buckets = calloc(sizeof(bucket_t *), size);
  return ht;
}

. . .
```

hashtable.h

```
unsigned long hash(char *str);
hashtable_t *make_hashtable(unsigned long size);
void  ht_put(hashtable_t *ht, char *key, void *val);
void *ht_get(hashtable_t *ht, char *key);
void  ht_del(hashtable_t *ht, char *key);
void  ht_iter(hashtable_t *ht, int (*f)(char *, void *));
void  ht_rehash(hashtable_t *ht, unsigned long newsize);
int   ht_max_chain_length(hashtable_t *ht);
void  free_hashtable(hashtable_t *ht);
```

← "API"

main.c

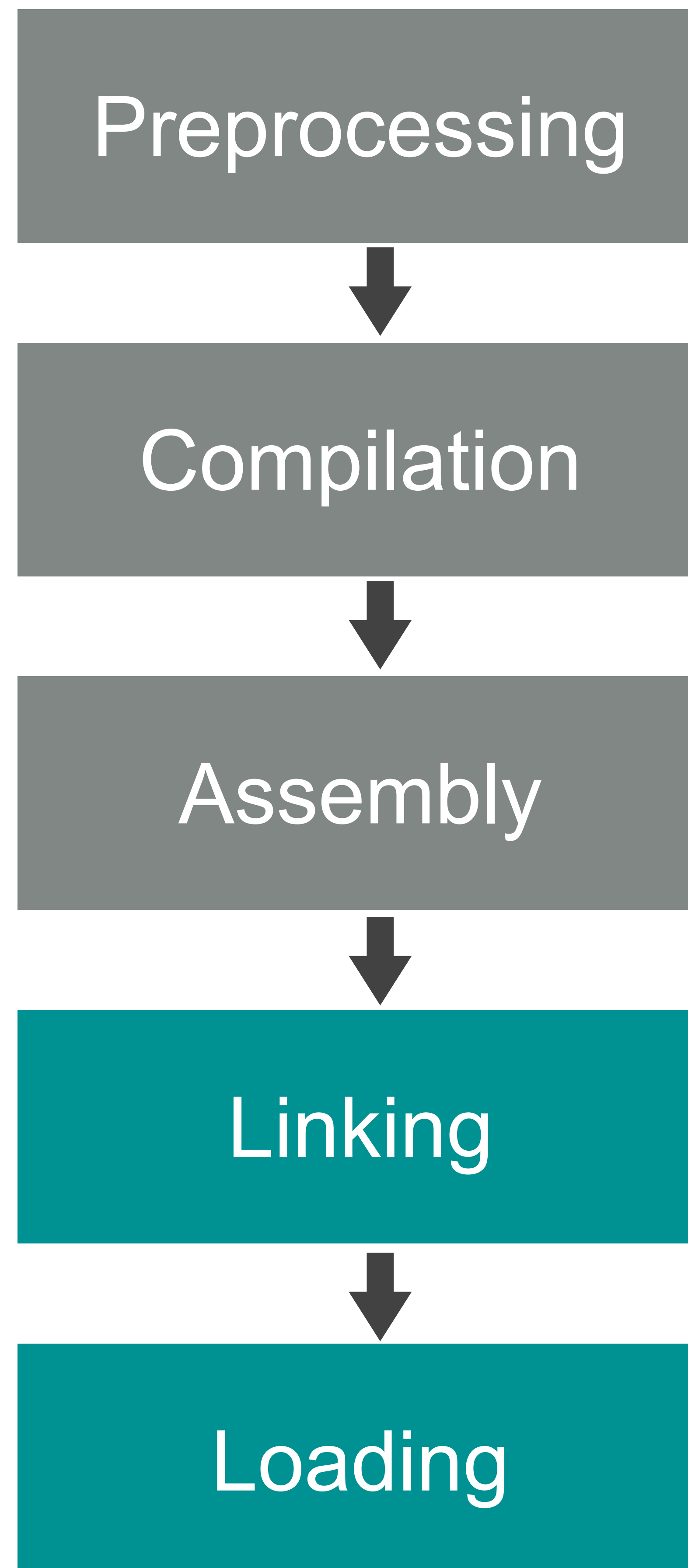
```
#include "hashtable.h"




int main(int argc, char *argv[]) {
  hashtable_t *ht;
  ht = make_hashtable(atoi(argv[1]));
  ...
  free_hashtable(ht);
  return 0;
}
```

Compilation

main.c

```c
#include <stdio.h>


int main () {
    printf("Hello world!\n");
    return 0;
}
```

```
$ gcc main.c -o prog
$ ./prog
Hello world!
```


Preprocessing
Compilation
Assembly
Linking
Loading

greet.h

```c
void greet(char *);
```

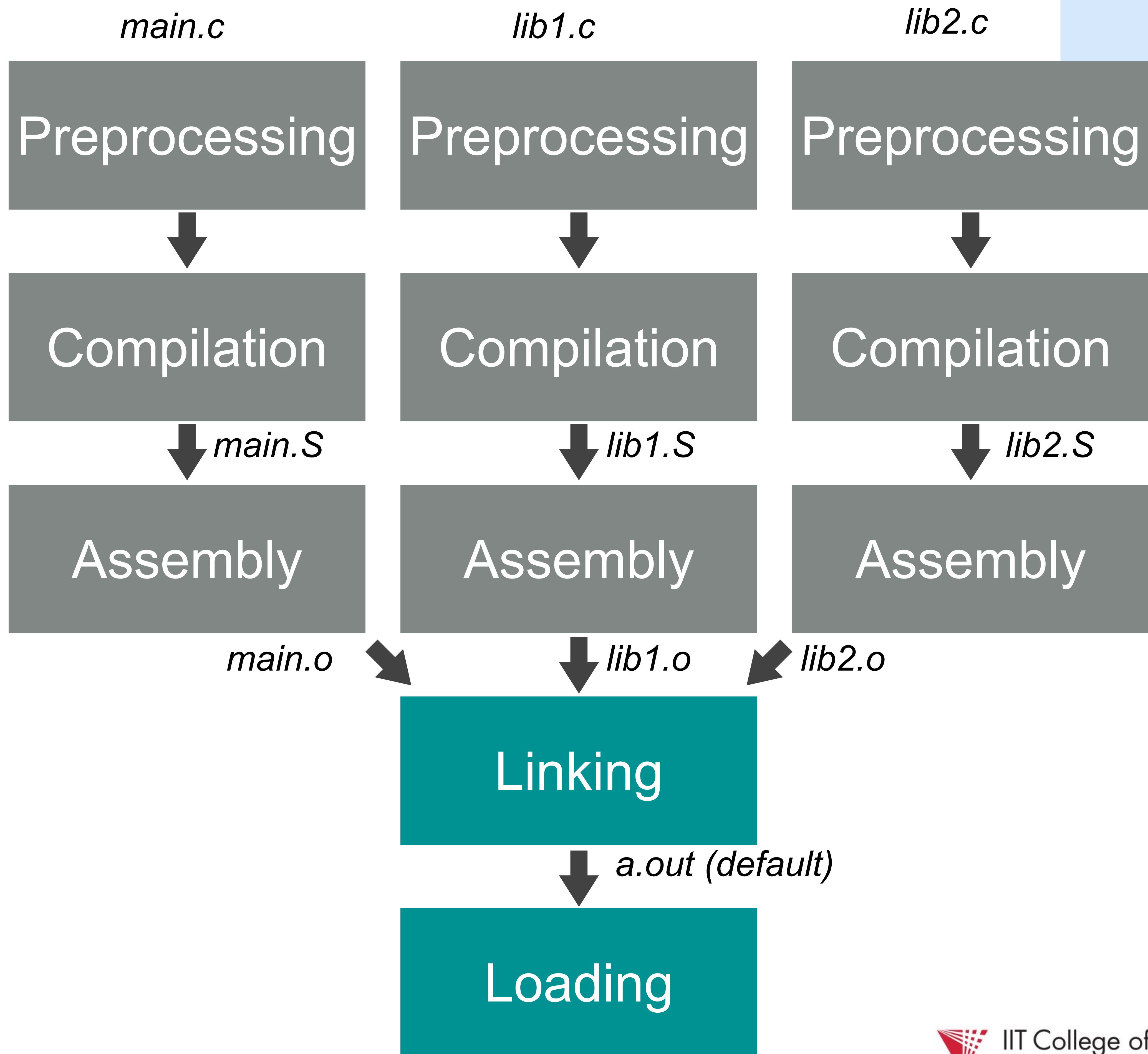
greet.c

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

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

main.c

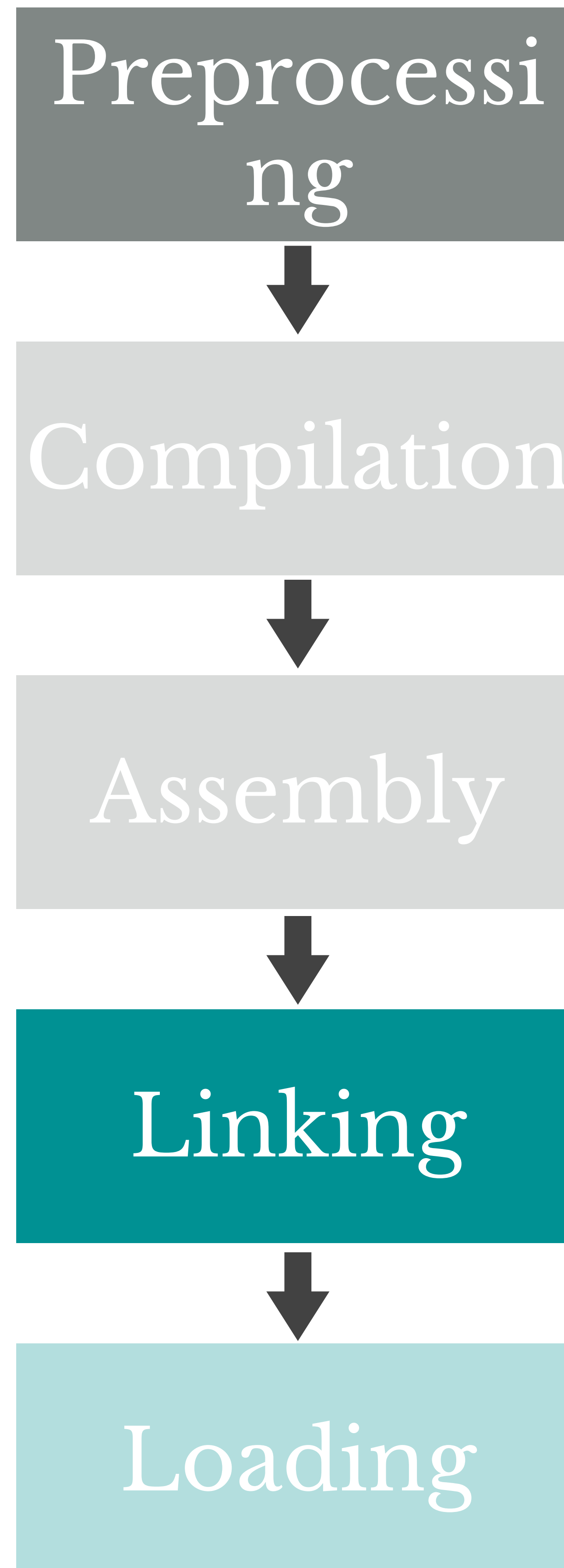
```c
#include "greet.h"

int main() {
  greet("Michael");
  return 0;
}
```

```
$ gcc -c greet.c     -o greet.o
$ gcc -c main.c      -o main.o
$ gcc greet.o main.o -o prog
$ ./prog
Hello, Michael
```



Computer Science

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY


Preprocessing
Compilation
Assembly
Linking
Loading

“Preprocessing”

- preprocessor *directives* exist for:
    - text substitution
    - macros
    - conditional compilation
- directives start with ‘#’

greet.h

```
void greet(char *);
```

greet.c

```
#include "greet.h"

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

```
$ gcc -E greet.c

void greet(char *);

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

```c
#define msg "Hello world!\n"


int main () {
    printf(msg);
    return 0;
}
```

```
$ gcc -E hello.c

int main () {
    printf("Hello world!\n");
    return 0;
}
```

```c
#define PLUS1(x) (x+1)

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

```
$ gcc -E plus1.c

int main () {
    int y;
    y = y * (y+1);
    return 0;
}
```

```
#define PLUS1(x) (x+1)

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

```
#define PLUS1(x) x+1

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

same effect?

```
$ gcc -E plus1.c

int main () {
    int y;
    y = y * (y+1);
    return 0;
}
```

```
$ gcc -E plus1b.c

int main () {
    int y;
    y = y * y+1;
    return 0;
}
```

no!

macros *blindly* manipulate *text*!

```
int main () {
  int f0=0, f1=1, tmp;

  for (int i=0; i<20; i++) {
#ifdef VERBOSE
    printf("Debugging: %d\n", f0);
#endif
    tmp = f0;
    f0 = f1;
    f1 = tmp + f1;
  }
  return 0;
}
```

```
$ gcc -E fib.c

int main () {
  int f0=0, f1=1, tmp;

  for (int i=0; i<20; i++) {
    tmp = f0;
    f0 = f1;
    f1 = tmp + f1;
  }
  return 0;
}
```

create preprocessor definition

```
$ gcc -D VERBOSE —E fib.c

int main () {
  int f0=0, f1=1, tmp;

  for (int i=0; i<20; i++) {
    printf("Debugging: %d\n", f0);
    tmp = f0;
    f0 = f1;
    f1 = tmp + f1;
  }
  return 0;
}
```

“Linking”

- Resolving symbolic references (e.g., variables, functions) to their definitions
  - e.g., by placing final target addresses in jump/call instructions
- Both *static* and *dynamic* linking are possible; the latter is performed at run-time

greet.h

```c
void greet(char *);
```

greet.c

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

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

main.c

```c
#include "greet.h"

int main() {
  greet("Michael");
  return 0;
}
```

```
$ gcc -c greet.c     -o greet.o
$ gcc -c main.c      -o main.o
```

```
$ objdump -d main.o
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   bf 00 00 00 00          mov    $0x0,%edi
   9:   e8 00 00 00 00          callq  e <main+0xe>
   e:   b8 00 00 00 00          mov    $0x0,%eax
  13:   5d                      pop    %rbp
  14:   c3                      retq
```

```
$ objdump -d greet.o
0000000000000000 <greet>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
   c:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  10:   48 89 c6                mov    %rax,%rsi
  13:   bf 00 00 00 00          mov    $0x0,%edi
  18:   b8 00 00 00 00          mov    $0x0,%eax
  1d:   e8 00 00 00 00          callq  22 <greet+0x22>
  22:   90                      nop
  23:   c9                      leaveq
  24:   c3                      retq
```

placeholder addresses

greet.h

```c
void greet(char *);
```

greet.c

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

void greet(char *name) {
  printf("Hello, %s\n", name);
}
```

main.c

```c
#include "greet.h"

int main() {
  greet("Michael");
  return 0;
}
```

```
$ gcc -c greet.c     -o greet.o
$ gcc -c main.c      -o main.o
$ gcc greet.o main.o -o prog
$ ./prog
Hello, Michael
```

```
$ objdump -d prog

0000000000400400 <printf@plt>:
  400400:   ff 25 12 0c 20 00       jmpq   *0x200c12(%rip) # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
  400406:   68 00 00 00 00          pushq  $0x0
  40040b:   e9 e0 ff ff ff          jmpq   4003f0 <_init+0x28>

0000000000400526 <main>:
  400526:   55                      push   %rbp
  400527:   48 89 e5                mov    %rsp,%rbp
  40052a:   bf e4 05 40 00          mov    $0x4005e4,%edi
  40052f:   e8 07 00 00 00          callq  40053b <greet>
  400534:   b8 00 00 00 00          mov    $0x0,%eax
  400539:   5d                      pop    %rbp
  40053a:   c3                      retq

000000000040053b <greet>:
  40053b:   55                      push   %rbp
  40053c:   48 89 e5                mov    %rsp,%rbp
  40053f:   48 83 ec 10             sub    $0x10,%rsp
  400543:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
  400547:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  40054b:   48 89 c6                mov    %rax,%rsi
  40054e:   bf ec 05 40 00          mov    $0x4005ec,%edi
  400553:   b8 00 00 00 00          mov    $0x0,%eax
  400558:   e8 a3 fe ff ff          callq  400400 <printf@plt>
  40055d:   90                      nop
  40055e:   c9                      leaveq
  40055f:   c3                      retq
```

“Linking”

- The linker allows us to create large, multi-file programs with complex variable/function cross-referencing
- Pre-compiled libraries can be “linked in” (statically or dynamically) without rebuilding from source

"Linking"

- But, we don't always *want* to allow linking a call to a definition!
  - e.g., to hide implementations and build *selective* public APIs

Visibility & Lifetime

Visibility: *where* can a symbol (var/fn) be seen from, and how do we refer to it?

Lifetime: *how long* does allocated storage space (e.g., for a var) remain useable?

sum.c

```c
int sumWithI(int x, int y) {
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
sum.c: In function `sumWithI':
sum.c:2: error: `I' undeclared (first use in this function)
main.c: In function `main':
main.c:6: warning: implicit declaration of function `sumWithI'
```

sum.c

```
int sumWithI(int x, int y) {
    int I;
    return x + y + I;
}
```

main.c

```
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
-1073743741
```

problem: variable *declaration* & *definition* are implicitly tied together

note: definition = *storage allocation* + possible *initialization*

`extern` keyword allows for declaration *sans definition*

sum.c

```c
int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
13
```

... and now global variables are visible from *everywhere*.

Good/Bad?

`static` keyword lets us limit the *visibility* of things

sum.c

```c
int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

static int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
Undefined symbols:
  "_I", referenced from:
      _sumWithI in ccmvi0RF.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

sum.c

```c
static int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
Undefined symbols:
  "_sumWithI", referenced from:
      _main in cc9LhUBP.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

static also forces the *lifetime* of variables to be equivalent to global

(i.e., stored in static memory vs. stack)

sum.c

```c
int sumWithI(int x, int y) {
    static int I = 10; // init once
    return x + y + I++;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int main() {
    printf("%d\n", sumWithI(1, 2));
    printf("%d\n", sumWithI(1, 2));
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
13
14
15
```

recap:

- by default, variable *declaration* also results in *definition* (storage allocation)
- `extern` is used to declare a variable but use a separate definition

recap:

- by default, functions & global vars are visible within *all* linked files
- `static` lets us limit the visibility of symbols to the defining file

recap:

- by default, variables declared inside functions have *local lifetimes* (stack-bound)
- `static` lets us change their storage class to static (aka “global”)