信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
C Language

Instructors:
Siting Liu & Yuan Xiao
Course website: https://faculty.shanghaitech.edu.cn/liust/courses/CS110.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2026/3/10

Administrative

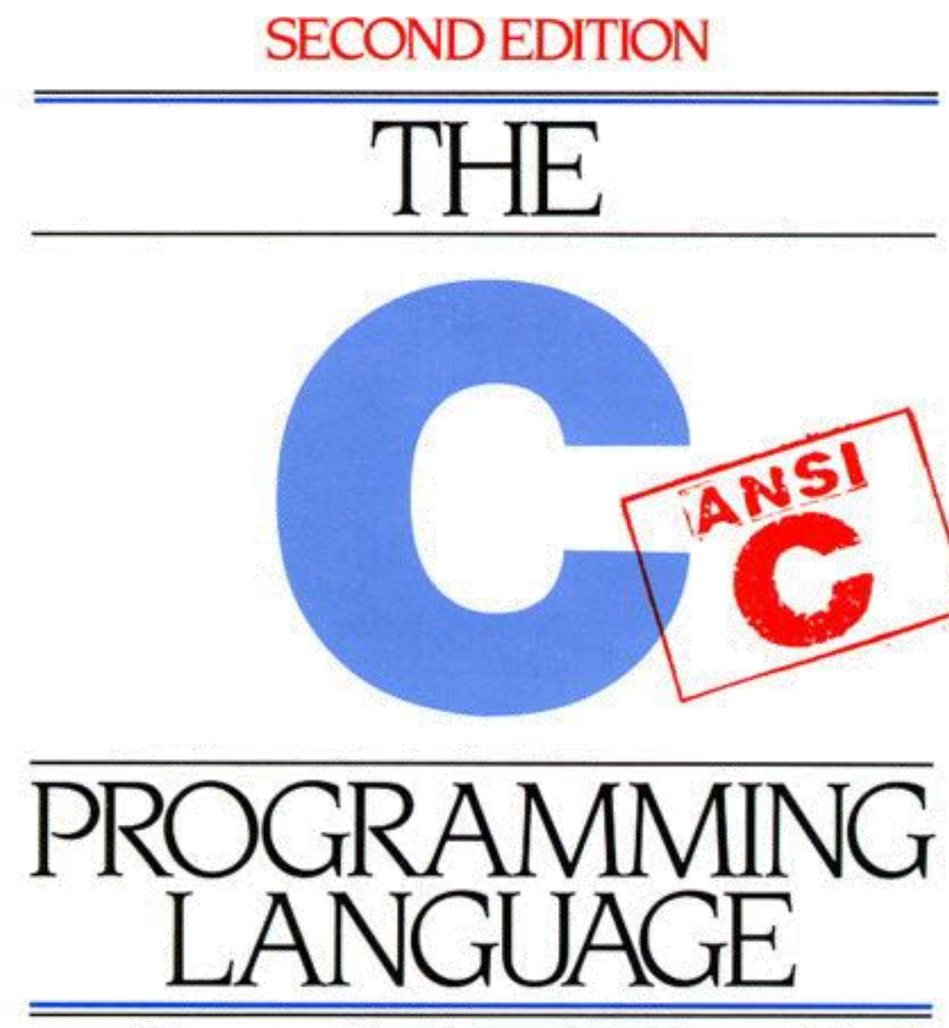
- HW1 due **today**!
- Lab1 to check this week during the Lab sessions.
- Discussion this week on git, gdb, etc. useful for HW2

  Fri. 18:00-, SPST 4-122 by TA Yuxuan Li
- Discussion recording last week available on course website.
- HW2 will be released later today, deadline Mar. 24th.
  - will use github classroom (see Lab 2 for git tutorial)
- Lab2 will be released later, checked next week

Outline

- Introduction to C
- How C works?
- C language review
- C memory management

Introduction to C
“The Universal Assembly Language”



History

- Appear early 1970s as a system implementation language for the nascent Unix operation system (Dennis Ritchie, group lead primarily by Ken Thompson, Bell Lab.)
- Influenced largely by BCPL and B language, named as NB (“new B”) in 1971 and renamed as C in 1972
- <*The C Prgramming Language*, 1st edition> in 1978
- Standardized by the ANSI X3J11 committee in the middle 1980s
- <*The C Prgramming Language*, 2nd edition> in 1988
- The ANSI C standard published (C89)
- Accepted as ISO/IEC 9899:1990 (C90) in 1990
- Updated as ISO/IEC 9899:1990/Amd.1:1995 (C95) in 1995
- C99, C11, C17, etc.

Ritchie, Dennis M. "The development of the C language." ACM Sigplan Notices 28.3 (1993): 201-208.

Some features of C

- General-purpose imperative, procedural language
  - BCPL, B and C all “close to the machine”; differ syntactically, but similar broadly
- Linker resolve external names (later B and all those of C)
  - BCPL uses a “global vector” for communicating between separately compiled programs
  - Covered later in CALL lecture
- Introduction of “fuller type” & better memory management
  - B and BCPL are typeless, or rather have a single data type, the “word” or “cell”, a fixed-length bit pattern
  - Values of array type are converted into pointers to the first of the objects making up the array; Pointers are integer indices in B and BCPL
  - Composed type (e.g., pointer to pointer)

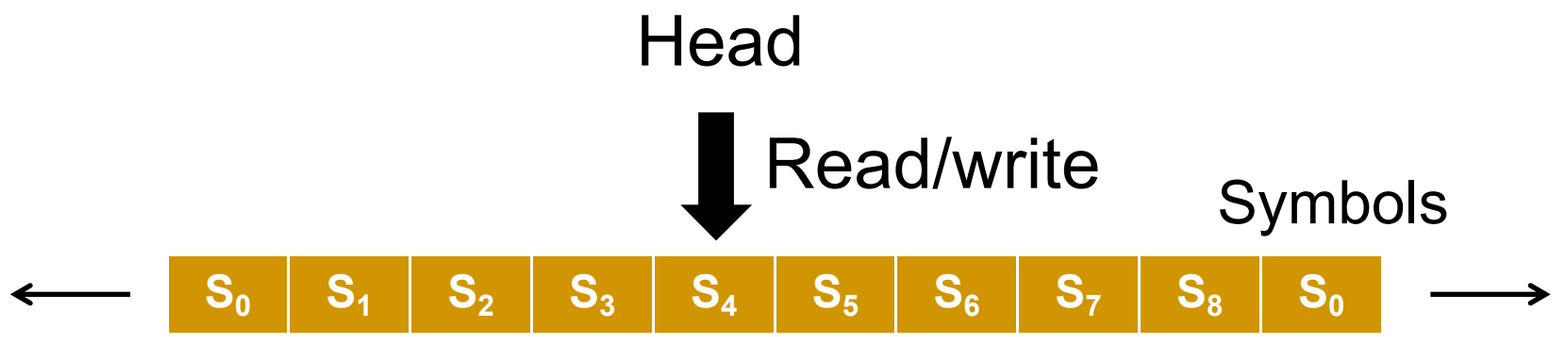
Ritchie, Dennis M. "The development of the C language." ACM Sigplan Notices 28.3 (1993): 201-208.

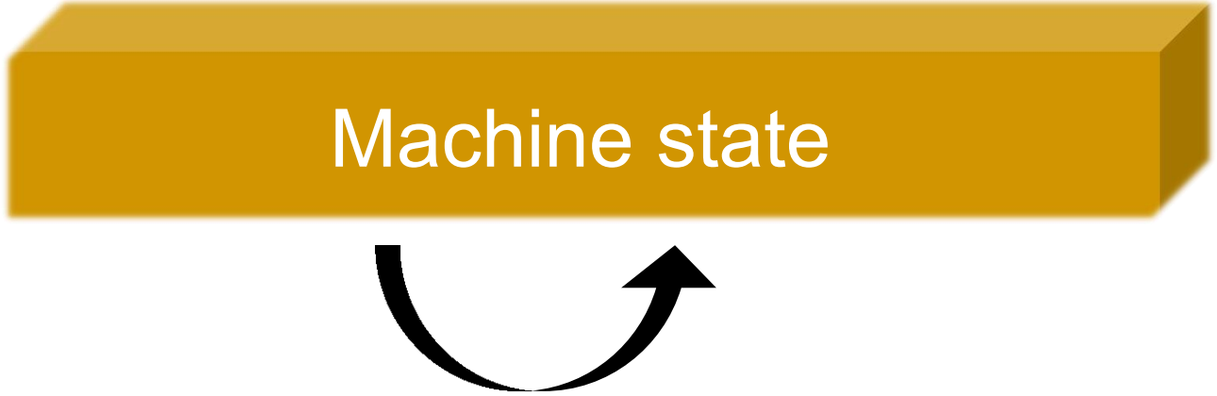
Disclaimer

- You will not learn how to fully code in C in these lectures! You'll still need your C reference for this course
  - K&R C is a recommendation
    - Check online for more sources
    - ANSI/ISO C standard manual (RTFM)
  - Key C concepts: pointers, arrays, memory management

Turing Machine & Machine Structure

Backgrounds to understand how C works

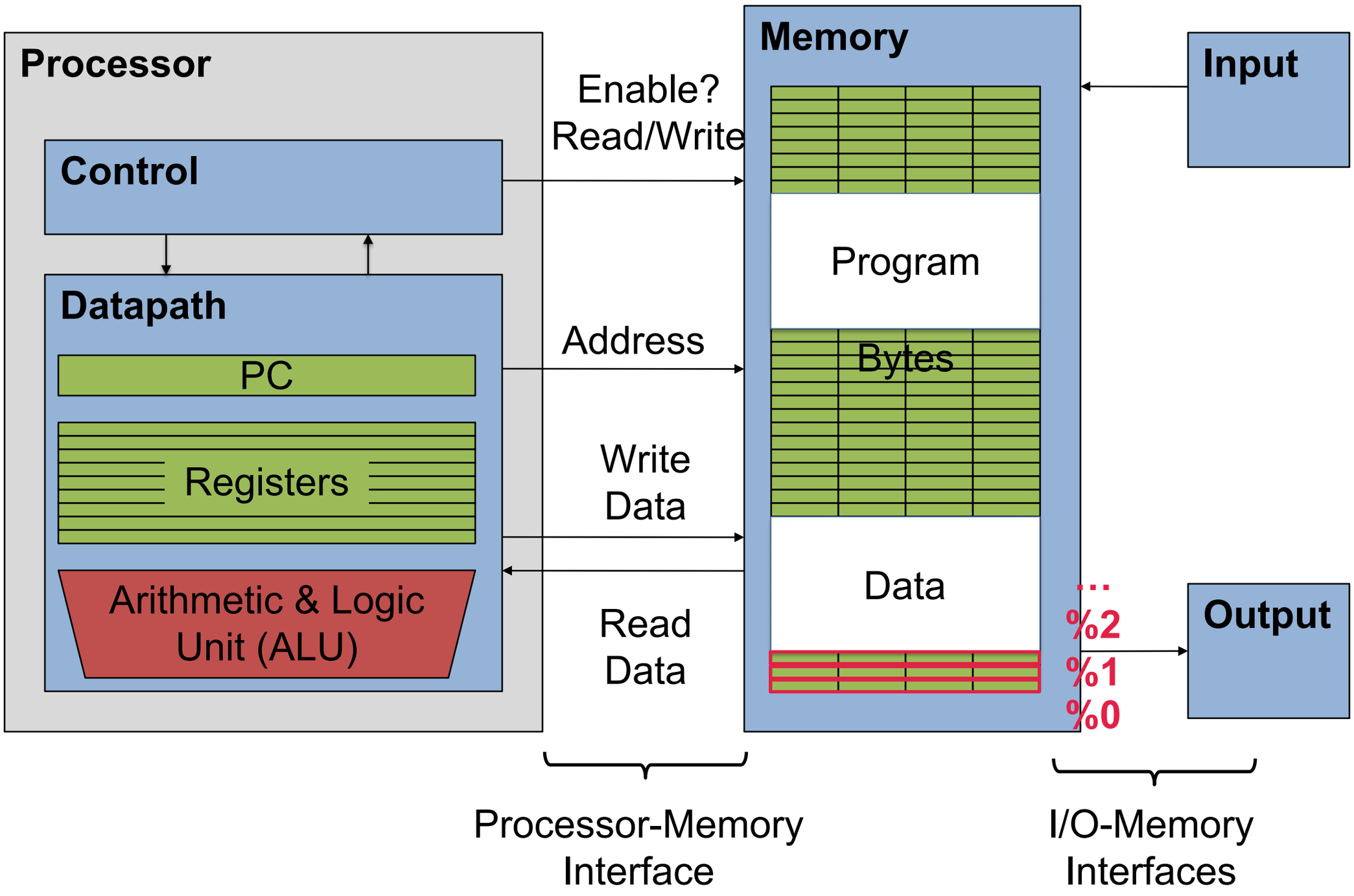




Update state according to the symbol being read and the current state

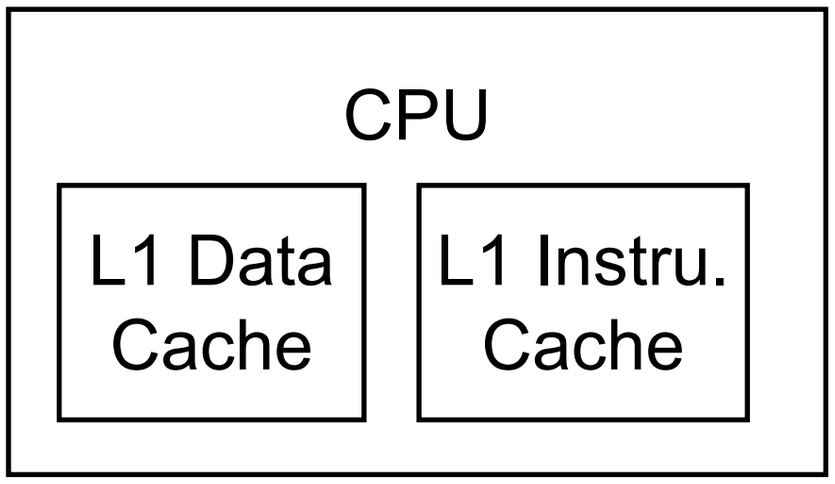
Machine Structure

Backgrounds to understand how C works



Organization of Computers

- Von Neumann Architecture
  - A.K.A. Princeton architecture
  - Uniform memory for data & program/instruction
- Harvard Architecture
  - Separated memory for data & program
  - E.g. MCU, DSP, L1 Cache



Outline

- Introduction to C
- How C works?
- C language review
- C memory management

How C program works?

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

We are here!

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw   t0, 0(s2)
lw   t1, 4(s2)
sw   t1, 0(s2)
sw   t0, 4(s2)
```

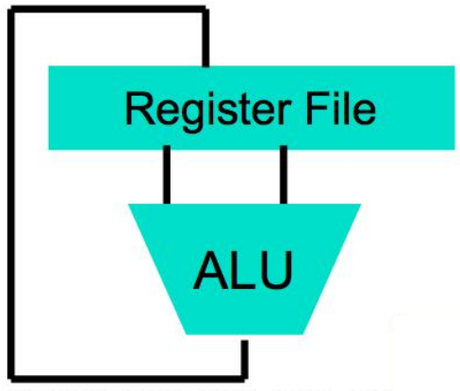
Anything can be represented as a *number*, i.e., data or instructions

Assembler

Machine Language Program (RISC-V)

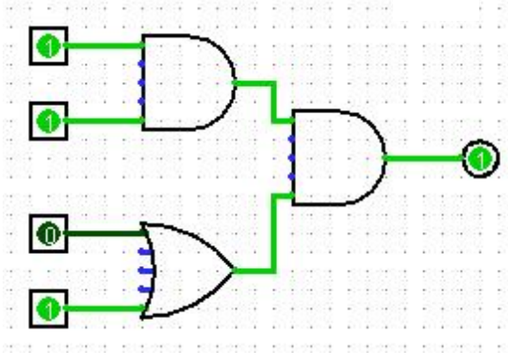
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Machine Interpretation



Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation



Logic Circuit Description (Circuit Schematic Diagrams)

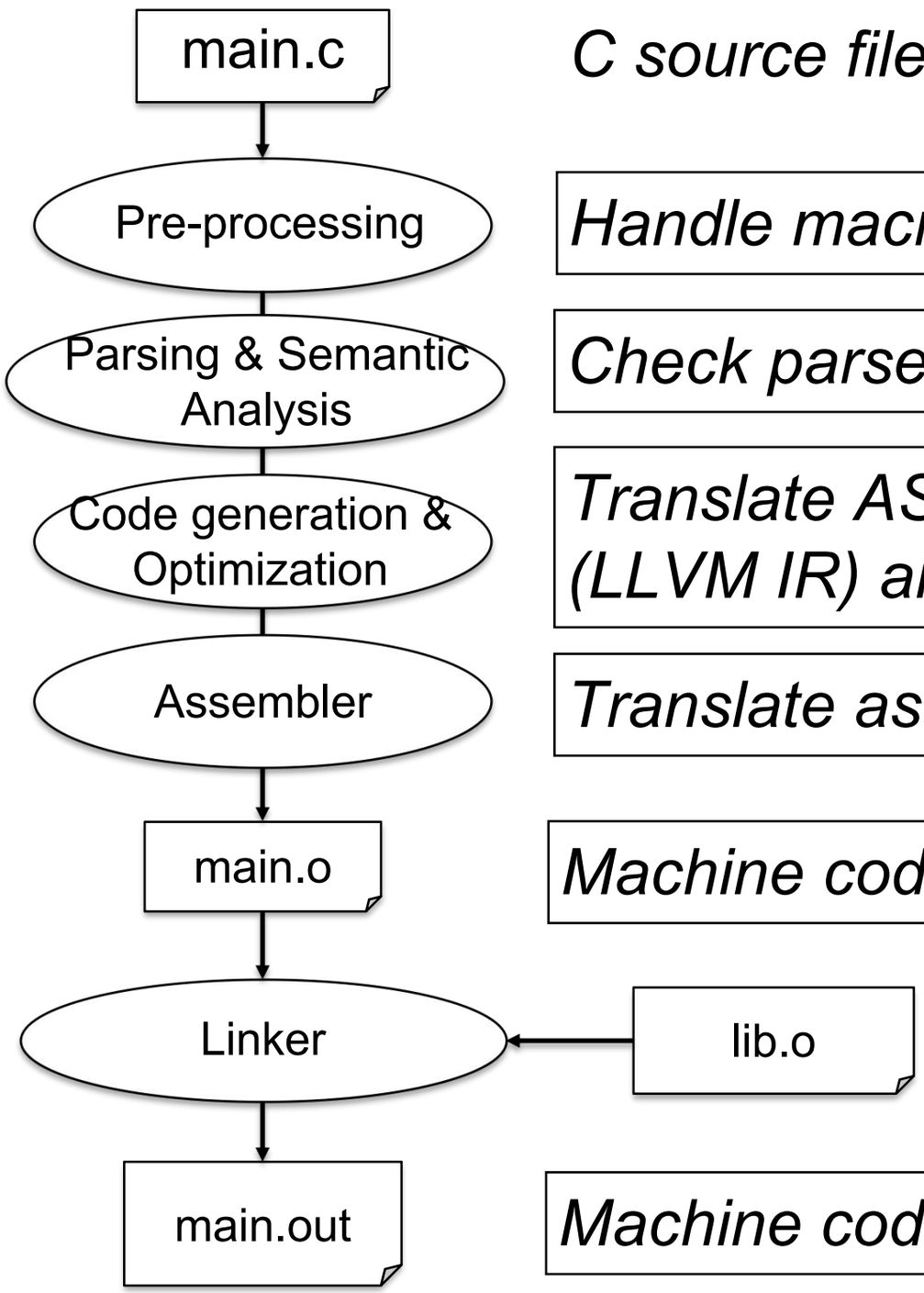
Compilation: Overview

- C compilers map C programs into architecture (OS & ISA)-specific machine code (strings of 1's and 0's)
  - Unlike Java, which converts to architecture-independent bytecode
  - Unlike Python environments, which interpret the code
  - These differ mainly in exactly when your program is converted to low-level machine instructions ("levels of interpretation")
- For C, generally a two part process of compiling .c files to .o files, then linking the .o files into executables;
- Assembling is also done (but is hidden, i.e., done automatically, by default); we'll talk about that later

Compilation: Advantages

- Excellent run-time performance: generally much faster than Python or Java for comparable code (because it optimizes for a given architecture)
- Reasonable compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled
- Mainstream C compiler in Linux:
  - GNU Compiler Collection (gcc, not only for C family);
  - clang/LLVM (for C language family)
  - In terminal/command line tool/shell, “man clang/gcc”

C Compilation Simplified Overview



C source files (text)

Handle macro expansion, #include, etc.

Check parse errors, generate "Abstract Syntax Tree" (AST)

Translate AST to lower level intermediate representation (LLVM IR) and then generate assembly (.s)

Translate assembly to machine code (binary)

Machine code object files (.o)

Pre-built object file libraries

Machine code executable file (.out)

C Pre-Processing (CPP)

- C source files first pass through CPP, before compiler sees code (mainly text editing)
- CPP replaces comments with a single space
- CPP commands begin with "#"
- #include "file.h" /* Inserts file.h into output */
- #include <stdio.h> /* Looks for file in standard location */
- #define M_PI (3.14159) /* Define constant */
- #if/#endif /* Conditional inclusion of text */
- Use -save-temps (-E) option to gcc to see result of preprocessing

Function-Like Macro

- #define MAG(x, y) (sqrt((x)*(x) + (y)*(y)))

```
#include <stdio.h>
#include <math.h>
#define MAG0(x, y) sqrt(x*x + y*y)
#define MAG(x, y) (sqrt((x)*(x) + (y)*(y)))
#define MAG2(x,y) ({double a=x; double b=y; sqrt(a*a + b*b);})
#define MSG "Hello \
World!\n"
int main() {
#ifdef MSG
  printf(MSG /* "hi!\n" */);
#endif
    printf("%f\n",MAG(3.0,4.0));
    double i=2, j=3, k0, k1, k2, k3;
    double c=2, d=3;
    k0=MAG0(i+1,j+1);
    k1=MAG(i+1,j+1);
    k2=MAG(++i,++j);
    k3=MAG2(++c,++d);
    printf("%f\n",k0);
    printf("%f\n",k1);
    printf("%f\n",k2);
    printf("%f\n",k3);
  return 0;
}
```

%clang/gcc -E introC_1_0.c

```
k0=sqrt(i+1*i+1 + j+1*j+1);
k1=(sqrt((i+1)*(i+1) + (j+1)*(j+1)));
k2=(sqrt((++i)*(++i) + (++j)*(++j)));
k3=({double a=++c; double b=++d; sqrt(a*a + b*b);});
```

=> Convention: put parenthesis EVERYWHERE!

CPP Macro II

- Avoid using macros whenever possible
- NO or very tiny speedup.
- Instead use C functions – e.g. inline function:

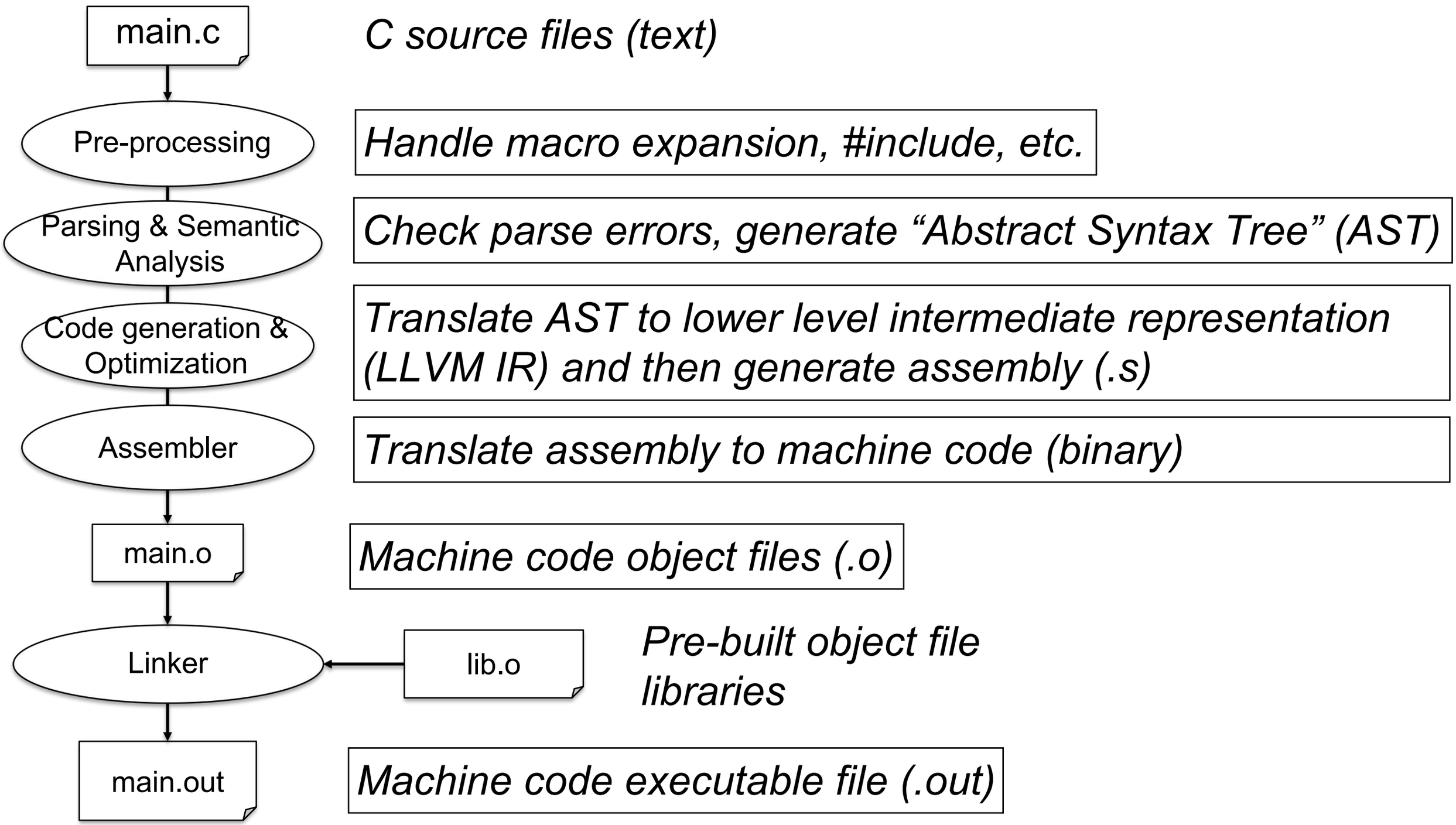
```
double mag(double x, double y);

double inline mag(double x, double y)
{  return sqrt( x*x + y*y ); }
```

Read more…

RTFM: https://gcc.gnu.org/onlinedocs/cpp/Macros.html

https://chunminchang.gitbooks.io/cplusplus-learning-note/content/Appendix/preprocessor_macros_vs_inline_functions.html

C Compilation Simplified Overview



Parser & Semantic Analysis

- Recognize each code word as a “token” (identifiers/symbols, C keywords, constant, comma, semicolon, etc.)
- Record the location of each token

```
%clang -fsyntax-only -Xclang -dump-tokens introC_1_1.c
```

```
#include <stdio.h>
int main() {//compute 1234 + 4321
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```

Lexer

```
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<introC_1_1.c:3:3>
identifier 'x'   [LeadingSpace] Loc=<introC_1_1.c:3:7>
equal '='        [LeadingSpace] Loc=<introC_1_1.c:3:9>
numeric_constant '1234'  [LeadingSpace] Loc=<introC_1_1.c:3:11>
comma ','               Loc=<introC_1_1.c:3:15>
identifier 'y'   [LeadingSpace] Loc=<introC_1_1.c:3:17>
equal '='        [LeadingSpace] Loc=<introC_1_1.c:3:19>
numeric_constant '4321'  [LeadingSpace] Loc=<introC_1_1.c:3:21>
semi ';'                Loc=<introC_1_1.c:3:25>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<introC_1_1.c:4:3>
identifier 'z'   [LeadingSpace] Loc=<introC_1_1.c:4:7>
equal '='        [LeadingSpace] Loc=<introC_1_1.c:4:9>
identifier 'x'   [LeadingSpace] Loc=<introC_1_1.c:4:11>
plus '+'                Loc=<introC_1_1.c:4:12>
identifier 'y'          Loc=<introC_1_1.c:4:13>
semi ';'                Loc=<introC_1_1.c:4:14>
identifier 'printf'      [StartOfLine] [LeadingSpace]  Loc=<introC_1_1.c:5:3>
l_paren '('             Loc=<introC_1_1.c:5:9>
string_literal '"z=%d/n"'               Loc=<introC_1_1.c:5:10>
comma ','               Loc=<introC_1_1.c:5:18>
identifier 'z'          Loc=<introC_1_1.c:5:19>
r_paren ')'             Loc=<introC_1_1.c:5:20>
semi ';'                Loc=<introC_1_1.c:5:21>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<introC_1_1.c:6:3>
numeric_constant '0'     [LeadingSpace] Loc=<introC_1_1.c:6:10>
semi ';'                Loc=<introC_1_1.c:6:11>
r_brace '}'      [StartOfLine]  Loc=<introC_1_1.c:7:1>
```

Parser & Semantic Analysis

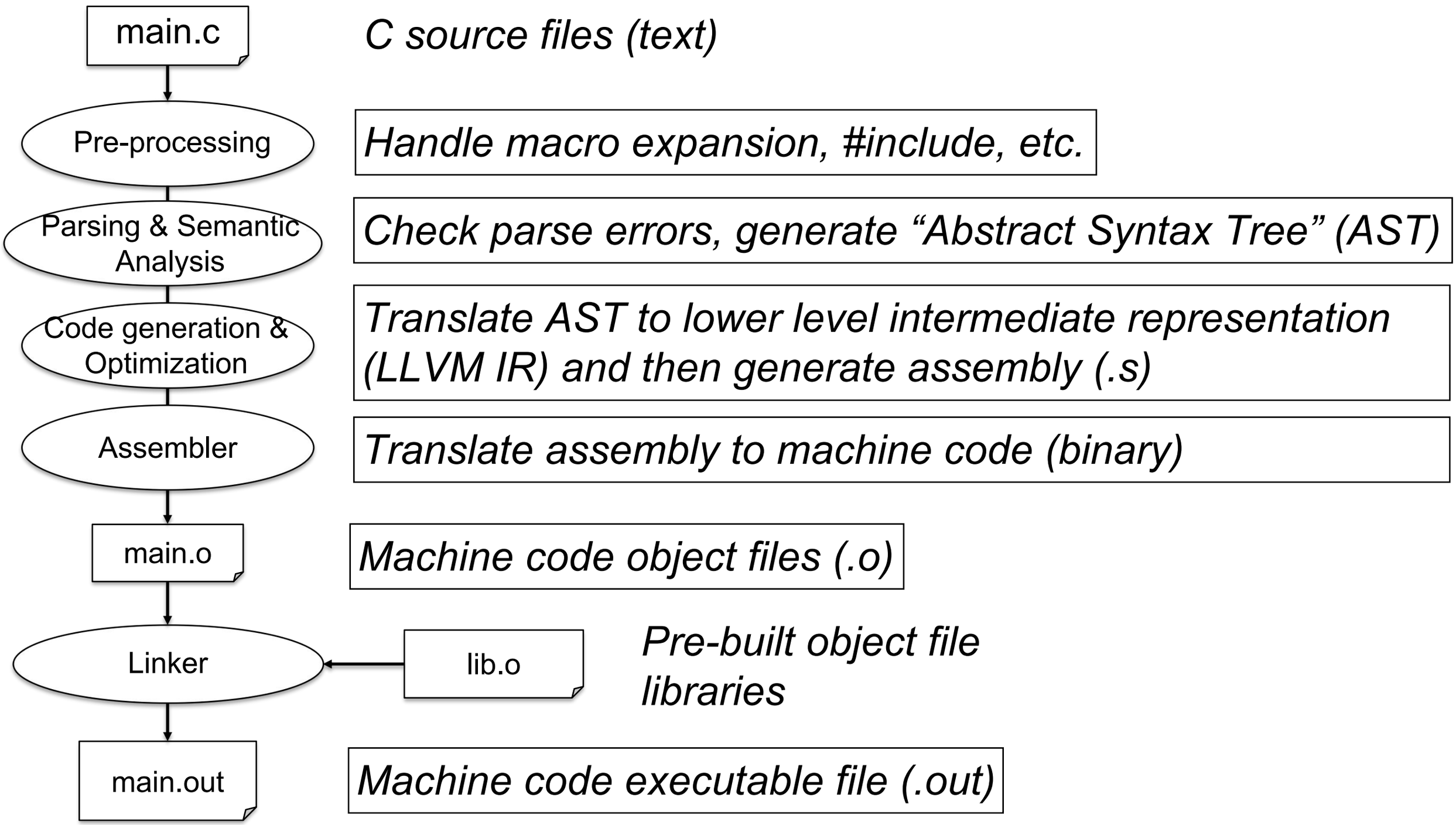
- Organize tokens as “AST” tree
- Report errors

```
% clang -fsyntax-only -Xclang -ast-dump introC_1_1.c
```

```
#include <stdio.h>
int main() {//compute 1234 + 4321
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```

```
`-FunctionDecl 0x1590f8600 <introC_1_1.c:2:1, line:7:1> line:2:5 main 'int ()'
  `-CompoundStmt 0x1590f8aa0 <col:12, line:7:1>
    |-DeclStmt 0x1590f87f8 <line:3:3, col:25>
    | |-VarDecl 0x1590f86b8 <col:3, col:11> col:7 used x 'int' cinit
    | | `-IntegerLiteral 0x1590f8720 <col:11> 'int' 1234
    | `-VarDecl 0x1590f8758 <col:3, col:21> col:17 used y 'int' cinit
    |   `-IntegerLiteral 0x1590f87c0 <col:21> 'int' 4321
    |-DeclStmt 0x1590f8920 <line:4:3, col:14>
    | `-VarDecl 0x1590f8828 <col:3, col:13> col:7 used z 'int' cinit
    |   `-BinaryOperator 0x1590f8900 <col:11, col:13> 'int' '+'
    |     |-ImplicitCastExpr 0x1590f88d0 <col:11> 'int' <LValueToRValue>
    |     | `-DeclRefExpr 0x1590f8890 <col:11> 'int' lvalue Var 0x1590f86b8 'x' 'int'
    |     `-ImplicitCastExpr 0x1590f88e8 <col:13> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0x1590f88b0 <col:13> 'int' lvalue Var 0x1590f8758 'y' 'int'
    |-CallExpr 0x1590f89f8 <line:5:3, col:20> 'int'
    | |-ImplicitCastExpr 0x1590f89e0 <col:3> 'int (*)(const char *, ...)' <FunctionToPointerDeca
y>
    | | `-DeclRefExpr 0x1590f8938 <col:3> 'int (const char *, ...)' Function 0x1590dd388 'printf
' 'int (const char *, ...)'
```

C Compilation Simplified Overview



Code Generation & Optimization

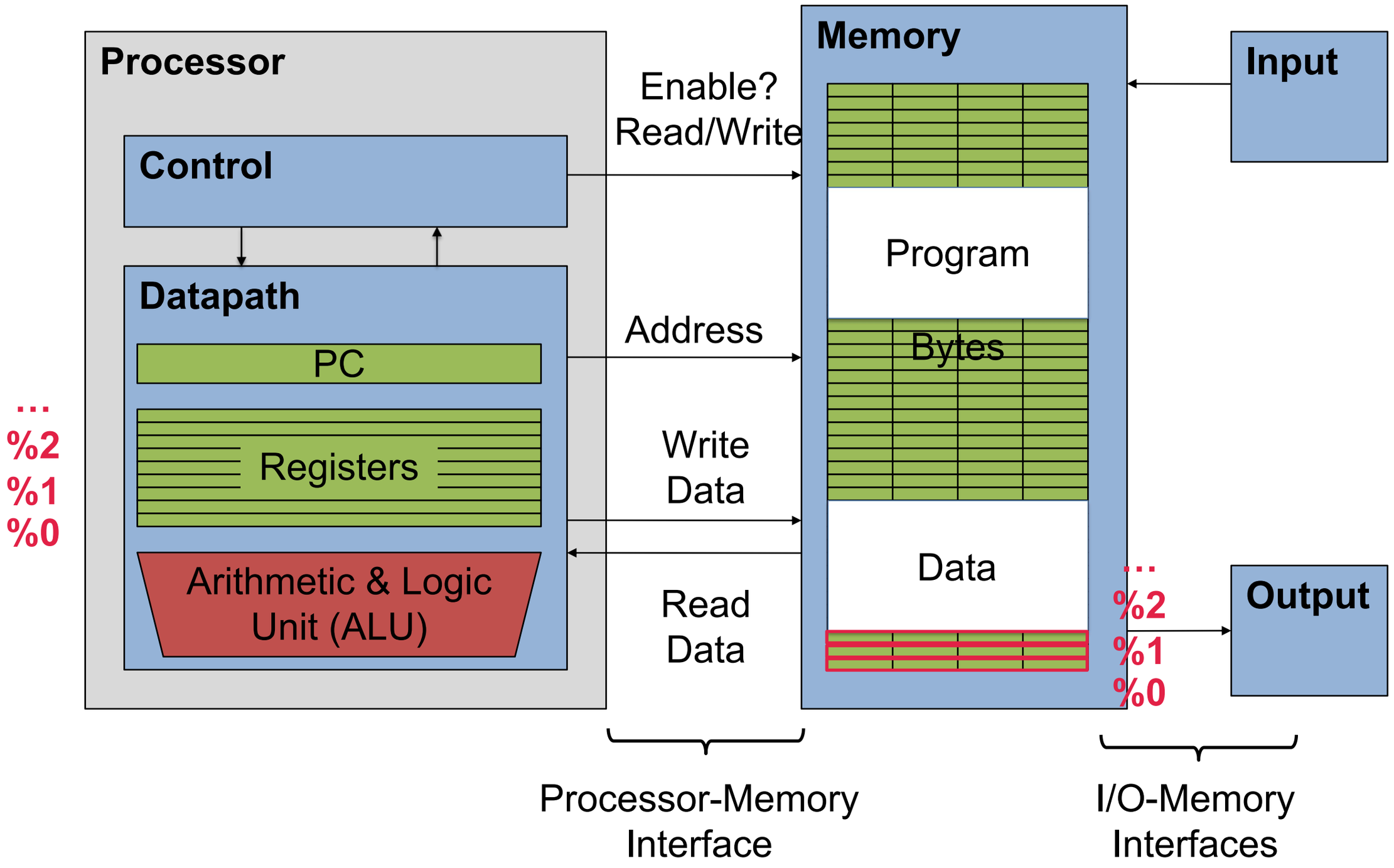
- Generate intermediate representation (IR)
  - LLVM IR for clang/LLVM
  - GIMPLE for gcc

%clang -S -emit-llvm introC_1_1.c -o introC_1_1.ll

```
#include <stdio.h>
int main() {//compute 1234 + 4321
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```

```
; ModuleID = 'introC_1_1.c'
source_filename = "introC_1_1.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx12.0.0"

@.str = private unnamed_addr constant [7 x i8] c"z=%d/n\00", align 1

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1234, i32* %2, align 4
  store i32 4321, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %5, %6
  store i32 %7, i32* %4, align 4
  %8 = load i32, i32* %4, align 4
  %9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([7 x i8]
0), i32 %8)
  ret i32 0
}
```

Optimization



IR to Assembly

% clang -S introC_1_1.c -o introC_1_1.s

```c
#include <stdio.h>
int main() {//compute 1234 + 4321
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```

Original code

LLVM IR

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1234, i32* %2, align 4
  store i32 4321, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %5, %6
  store i32 %7, i32* %4, align 4
  %8 = load i32, i32* %4, align 4
  %9 = call i32 (i8*, ...) @printf(i8*
getelementptr inbounds ([7 x i8], [7 x i8]*
@.str, i64 0, i64 0), i32 %8)
  ret i32 0
}
```

```
        .section    __TEXT,__text,regular,pure_instructions
        .build_version macos, 12, 0   sdk_version 13, 1
        .globl _main                    ; -- Begin function main
        .p2align    2
_main:                                  ; @main
        .cfi_startproc
; %bb.0:
        sub   sp, sp, #48
        stp   x29, x30, [sp, #32]       ; 16-byte Folded Spill
        add   x29, sp, #32
        .cfi_def_cfa w29, 16
        .cfi_offset w30, -8
        .cfi_offset w29, -16
        mov   w8, #0
        str   w8, [sp, #12]             ; 4-byte Folded Spill
        stur  wzr, [x29, #-4]
        mov   w8, #1234
        stur  w8, [x29, #-8]
        mov   w8, #4321
        stur  w8, [x29, #-12]
        ldur  w8, [x29, #-8]
        ldur  w9, [x29, #-12]
        add   w8, w8, w9
        str   w8, [sp, #16]
        ldr   w9, [sp, #16]
```

ARM Assembly

Translated to machine code defined by ISA

Assembly to Machine Code

(later details in CALL chapter)

% clang -c introC_1_1.c -o introC_1_1.o

% objdump -d introC_1_1.o

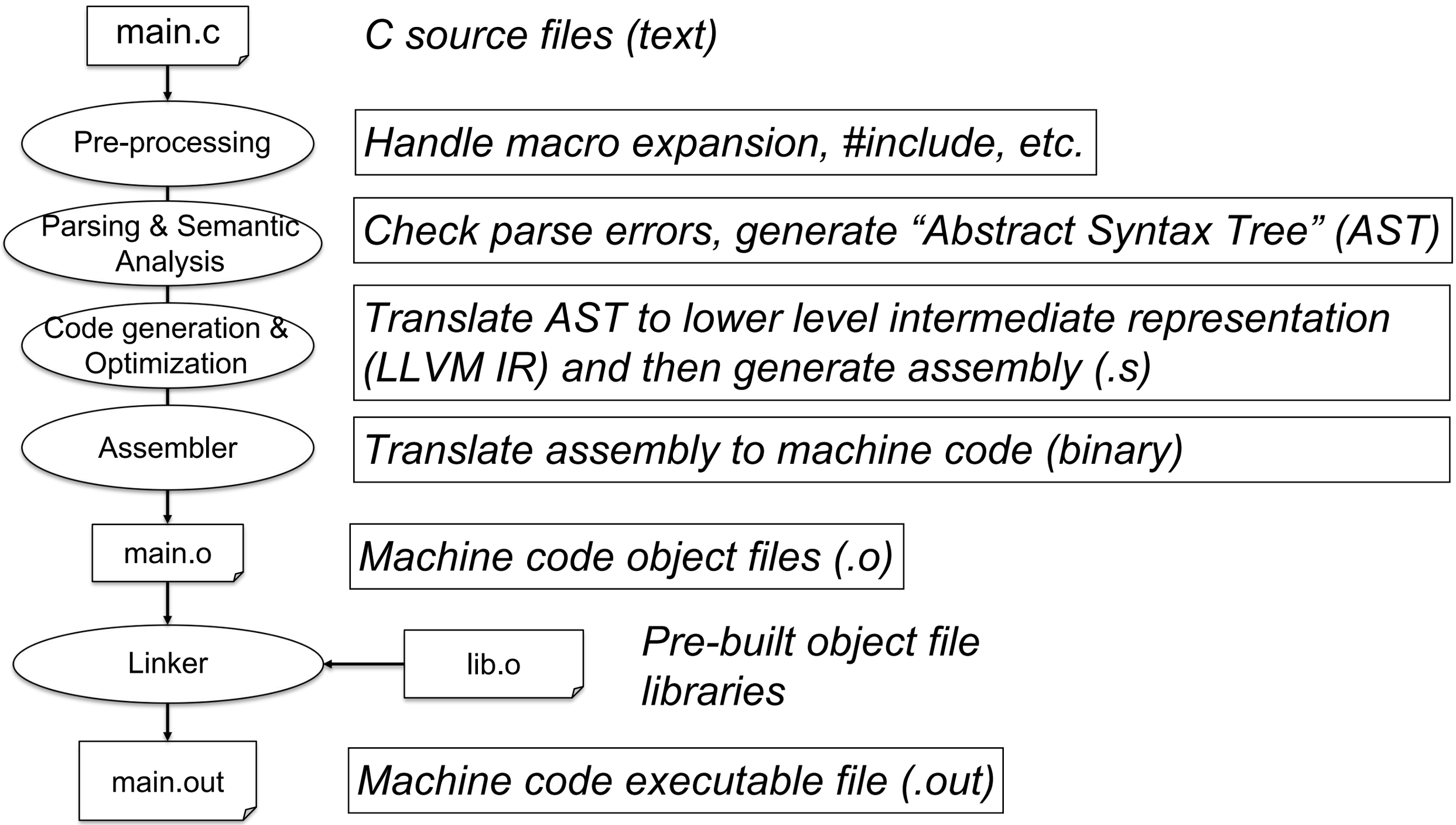
```
Disassembly of section __TEXT,__text:

0000000000000000 <ltmp0>:
       0: ff c3 00 d1   sub sp, sp, #48
       4: fd 7b 02 a9   stp x29, x30, [sp, #32]
       8: fd 83 00 91   add x29, sp, #32
       c: 08 00 80 52   mov w8, #0
      10: e8 0f 00 b9   str w8, [sp, #12]
      14: bf c3 1f b8   stur wzr, [x29, #-4]
      18: 48 9a 80 52   mov w8, #1234
      1c: a8 83 1f b8   stur w8, [x29, #-8]
      20: 28 1c 82 52   mov w8, #4321
      24: a8 43 1f b8   stur w8, [x29, #-12]
      28: a8 83 5f b8   ldur w8, [x29, #-8]
      2c: a9 43 5f b8   ldur w9, [x29, #-12]
      30: 08 01 09 0b   add w8, w8, w9
      34: e8 13 00 b9   str w8, [sp, #16]
      38: e9 13 40 b9   ldr w9, [sp, #16]
      3c: e8 03 09 aa   mov x8, x9
      40: e9 03 00 91   mov x9, sp
      44: 28 01 00 f9   str x8, [x9]
      48: 00 00 00 90   adrp x0, 0x0 <ltmp0+0x48>
      4c: 00 00 00 91   add x0, x0, #0
      50: 00 00 00 94   bl 0x50 <ltmp0+0x50>
      54: e0 0f 40 b9   ldr w0, [sp, #12]
      58: fd 7b 42 a9   ldp x29, x30, [sp, #32]
      5c: ff c3 00 91   add sp, sp, #48
      60: c0 03 5f d6   ret
```

Machine Code

Stored program or instructions

ARM Assembly

C Compilation: Simplified Overview



Wrap-it-up

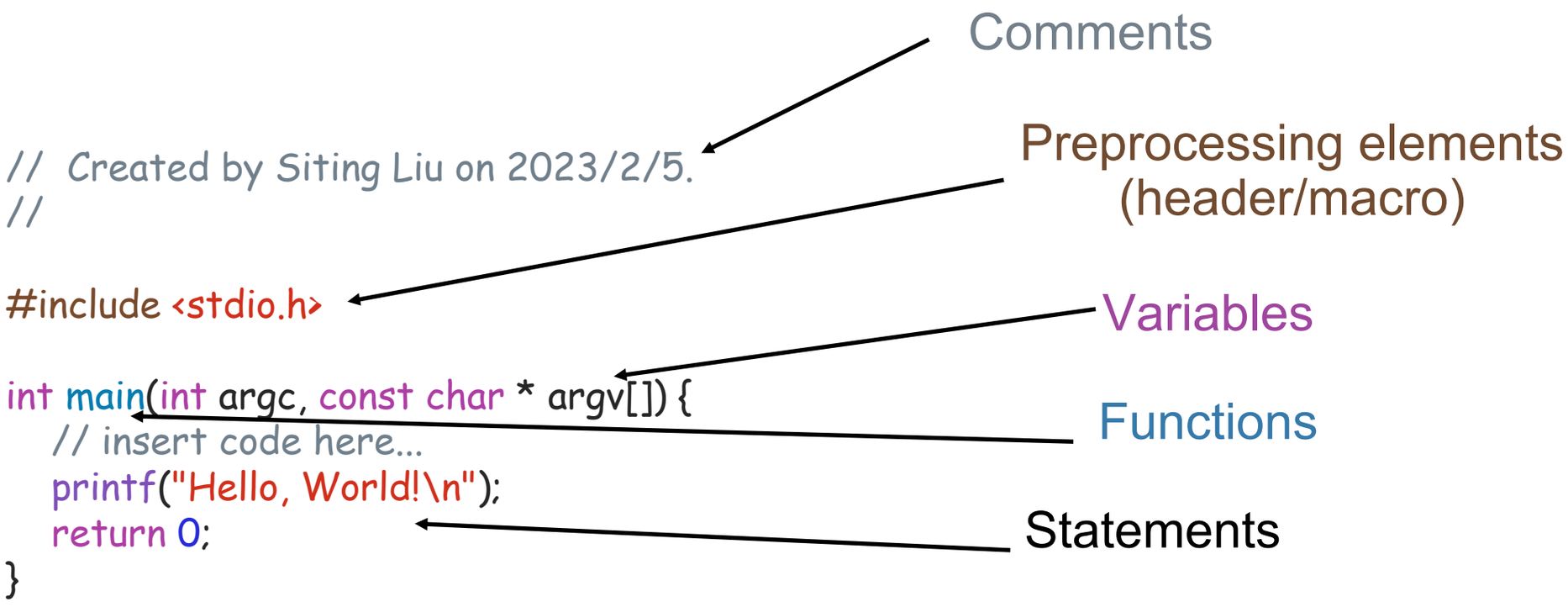
- From C to machine code (clang/gcc *.c ⟶ *.out & ./*.out)
  - Pre-processing (macro, function-like macro, text editing, #include)
    - Use “()” whenever necessary, or use “function” directly
  - Parser & Semantic Analysis (tokenization & generate AST, basic operations)
  - Translate to IR & optimize (machine structure)
  - Translate to assembly and then machine code, executed by hardware (Some details covered in future lectures, CALL)
  - Clang manual: https://releases.llvm.org/14.0.0/tools/clang/docs/UsersManual.html
  - GCC: https://gcc.gnu.org/

Outline

- Introduction to C
- How C works?
- C review
- C memory management

C Review

- Typical C program



- Must C program start with main()? (see C standard)

Variables

- Typed Variables in C

```
int   variable1  = 2;
float variable2  = 1.618;
char  variable3  = 'A';
```

Must declare the type of data a variable will hold;

Initialize, otherwise it holds garbage

Type	Description	Examples
int	integer numbers, including negatives	0, 78, -1400
unsigned int	integer numbers (no negatives)	0, 46, 900
long	larger signed integer	-6,000,000,000
(un)signed char	single text character or symbol	'a', 'D', '?'
float	floating point decimal numbers	0.0, 1.618, -1.4
double	greater precision/big FP number	10E100

C standard defines a lot of “Undefined Behavior”s. It means the code may produce unpredictable behavior. It may

- Produce different results on different computers/OS;
- Produce different results among multiple runs;
- Very difficult to re-produce and debug

Integers

- Typed Variables in C

Language	sizeof(int)
Python	>=32 bits (plain ints), infinite (long ints)
Java	32 bits
C	Depends on computer; 16 or 32 or 64 bits

- C: int should be integer type that target processor works with most efficiently
- Generally: sizeof(long long) ≥ sizeof(long) ≥ sizeof(int) ≥ sizeof(short)
  - Also, short >= 16 bits, long >= 32 bits
  - All could be 64 bits

Integer Constants

```
#include <stdio.h>
int main() {
   printf((6-2147483648)>(6)?"T\n":"F\n");
   printf((6-0x80000000)>(6)?"T\n":"F\n");
   return 0;
}
```

Semantics: The value of a decimal constant is computed base 10; that of an octal constant base 8; that of a hexadecimal constant base 16. The lexically first digit is the most significant.
The type of an integer constant is the first of the corresponding list in which its value can be represented. Unsuffixed decimal **int, long int, unsigned long int;** unsuffixed octal or hexadecimal: **int, unsigned int, long int, unsigned long int;** suffixed by the letter u or U: **unsigned int, unsigned long int**; suffixed by the letter l or L: **long int, unsigned long int**; suffixed by both the letters u or U and 1 or L: **unsigned long int**.

Range of each type defined in <limits.h> (INT_MAX, INT_MIN)

Consts. and Enums. in C

- Constant is assigned a typed value once in the declaration; value can't change during entire execution of program

```
const float golden_ratio = 1.618;
const int days_in_week = 7;
```

- You can have a constant version of any of the standard C variable types
- Enums: a group of related integer constants. Ex:

```
enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
enum color {RED, GREEN, BLUE};

enum color c = RED;
```

Excercise

Compare "`#define PI 3.14`" and "`const float pi=3.14`" – which is true?

A: Constants "PI" and "pi" have same type

B: Can assign to "PI" but not "pi"

C: Code runs at about the same speed using "PI" or "pi"

D: "pi" takes more memory space than "PI"

E: Both behave the same in all situations

C Syntax: Variable Declarations

- All variable declarations must appear before they are used (e.g., at the beginning of the block)
- A variable may be initialized in its declaration;
  if not, it holds garbage!
- Examples of declarations:
  - Correct:
```
{
    int a = 0, b = 10;
    ...
```
  - Incorrect (in C89):
```
    for (int i = 0; i < 10; i++)
}
```

Newer C standards are more flexible about this…

C Syntax: True or False

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of pointer: more on this later)
- No explicit Boolean type (use `stdbool.h`)

- What evaluates to TRUE in C?
  - Anything that isn't false is true
  - Same idea as in Python: only 0s or empty sequences are false, anything else is true!

C Operators

- arithmetic: +, -, *, /, %
- assignment: =
- augmented assignment: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- bitwise logic: ~, &, |, ^
- bitwise shifts: <<, >>
- boolean logic: !, &&, ||
- equality testing: ==, !=
- subexpression grouping: ( )
- order relations: <, <=, >, >=
- increment and decrement: ++ and --
- member selection: ., ->
- conditional evaluation: ? :

Typed C Functions

- You need to declare the return type of a function when you declare it (plus the types of any arguments)
- You also need to declare functions before they are used
  - Usually in a separate header file, e.g.

```
int number_of_people();
float dollars_and_cents();
int sum(int x, int y);
```

- void type means “returns nothing”

```
int number_of_people()
{  return 3;}

float dollars_and_cents ()
{ return 10.33; }

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

Summary

- Compiler: translate C code into machine code
  - Pre-processing (text processing/replacement)
- Corner cases in C
- Basic C operators