Programming in C

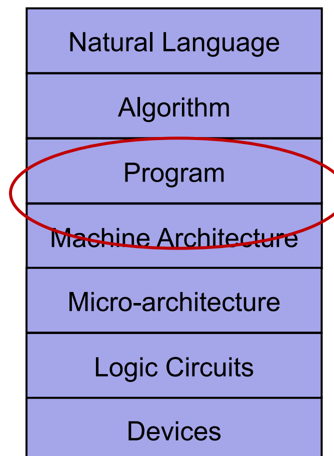
(Chapters 11-13)

1

Problem Transformation
- levels of abstraction

How do user programs get executed?

Translation from C to Assembly
- **Scope of variables**
- **Function calls**
- **Recursion**
- **Pointers & Arrays**
- **Data Structures**
- **Memory Allocation**



2

Programming Languages

- Assembly language is low-level.
  - It exposes machine instructions and details of the ISA to the programmer.
  - It is ISA-specific.
  - It is useful when the programmer needs fine-grained control of instruction flow and memory usage.

- A high-level language provides a computational abstraction that is machine-independent.
  - Symbolic names (variables) instead of registers and memory locations.
  - High-level operators: multiply, divide, shift, ...

3

Why use a High-Level Language? 1

- Expressiveness: say more with less effort, closer to human-level thinking

```
a = b * c;
```

C statement

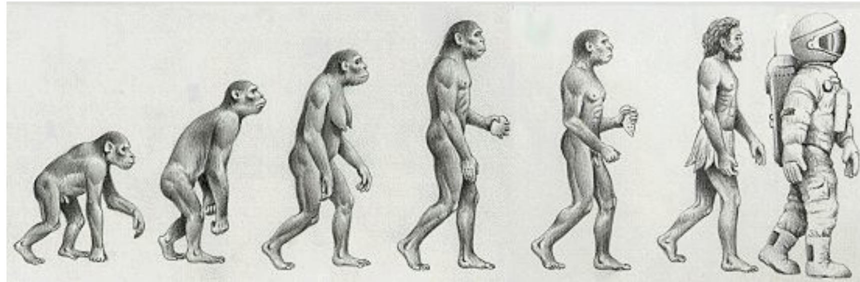
Equivalent LC-3 code

```
        AND R2,R2,#0
        AND R3,R3,#0
        LDR R0,R5,#-1   ; b
        BRz L3
        BRp L1
        NOT R3,R3
        NOT R0,R0
        ADD R0,R0,#1
L1      LDR R1,R5,#-2   ; c
        BRz L5
        BRp L2
        NOT R3,R3
        NOT R1,R1
        ADD R1,R1,#1
L2      ADD R2,R2,R0    ; b * c
        ADD R1,R1,#-1
        BRp L2
        ADD R3,R3,#0
        BRp L3
        NOT R2,R2
        ADD R2,R2,#1
L3      STR R2,R5,#0    ; store to a
```

- Both multiply two values together –
- which is easier to understand?

4

The Evolution of Programming



Machine Language (binary) | Assembly Language | C | Java | Python | Haskell | What's Next??

Simula/C++/etc

5

History: 1969 AT&T Bell Labs drops out of MULTICS project.

>Ken Thompson develops UNICS

>Ken Thompson writes interpreted language: B

>Dennis Ritchie and Brian Kernighan improve on "B" and called it "C"

A picture purportedly of Dennis Ritchie and Brian Kernighan two of the key developers of C with a PDP-11 minicomputer

6

Programming in C == Working without a net

A C program is like a fast dance on a newly waxed dance floor by people carrying razors." — *Waldi Ravens.*

7

Compilation vs. Interpretation

- Different ways of translating high-level language
- *Interpretation*
  - interpreter = program that executes program statements
    - Called a Virtual Machine
  - generally one line/command at a time
  - limited processing
  - easy to debug, make changes, view intermediate results
  - languages: BASIC, LISP, Perl, *Java*, Matlab, Python, C-shell
- *Compilation*
  - translates statements into machine language
    - does not execute, but creates executable program
  - performs optimization over multiple statements
  - change requires recompilation
    - can be harder to debug, since executed code may be different
  - languages: C, C++, Fortran, Pascal

8

Compilation vs. Interpretation

•Consider the following algorithm:

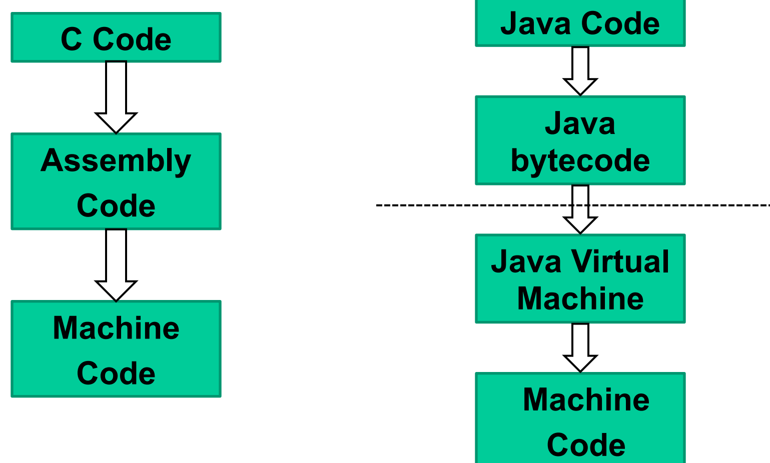
```
• Get W from the keyboard.
• X = W + W
• Y = X + X
• Z = Y + Y
• Print Z to screen.
```

•If interpreting, how many arithmetic operations occur?

•If compiling, how many arith operations needed ?

•In a compiler we can analyze the entire program and possibly reduce the number of operations. Can we simplify the above algorithm to use a single arithmetic operation?

C vs Java



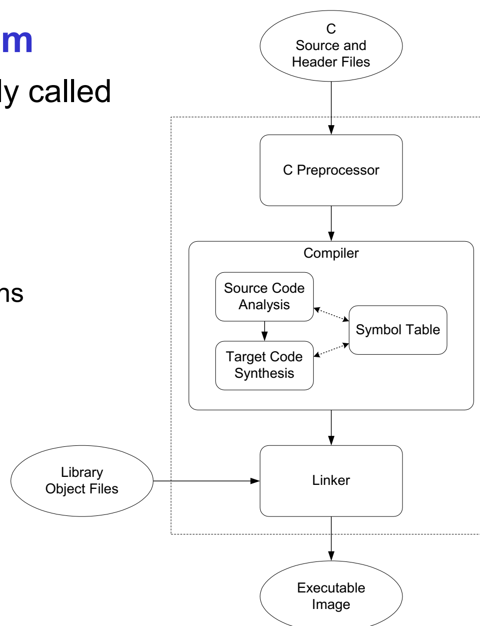
Compiling a C Program

•Entire mechanism is usually called the "compiler"

•Preprocessor
- macro substitution
- conditional compilation
- "source-level" transformations
  - output is still C

•Compiler
- generates object file
  - machine instructions

•Linker
- combine object files (including libraries) into executable image



11

Compiler

•Source Code Analysis….you will learn this in Foundations
- "front end"
- parses programs to identify its pieces
  - variables, expressions, statements, functions, etc.
- depends on language (not on target machine)

•Code Generation… we will cover it implicitly in this course
- "back end": generates machine code from analyzed source
- may optimize machine code to make it run more efficiently
- very dependent on target machine
- *We will play the role of the code generation component as we discuss how different C concepts are implemented in LC3*
  - This is what the compiler backend does

•Symbol Table
- map between symbolic names and items
- like assembler, but more kinds of information

12

A Simple C Program

```
#include <stdio.h>
#define STOP 0
int scale=10;
int square(){      /* function square */{
      int x,y; /* two local variables */
      x = scale*scale;
      return(x);}
/* Function: main  */
/* Description: gets value and squares it */
main()
{  /* variable declarations */
    int number, value, temp; /* three integer local
variables */
    /* prompt user for input */
    printf("Enter a positive number: ");
    scanf("%d", &value);  /* read into value */
    /* scale input and print */
    number = value*square();
    printf(”number is %d\n", number);
}
```

13

Preprocessor Directives

- `#include <stdio.h>`
  - Before compiling, copy contents of header file (stdio.h) into source code.
  - Header files typically contain descriptions of functions and variables needed by the program.
    - no restrictions -- could be any C source code

- `#define STOP 0`
  - Before compiling, replace all instances of the string "STOP" with the string "0"
  - Called a *macro*
  - Used for values that won't change during execution, but might change if the program is reused. (Must recompile.)

14

Symbol Table

•Like assembler, compiler needs to know information associated with identifiers

- in assembler, all identifiers were labels and information is address

•Compiler keeps more information

- Name (identifier)
- Type
- Location in memory
- Scope

Name	Type	Offset	Scope
scale	int	0	global
number	int	0	main
value	int	-1	main
temp	int	-2	main
x	int	0	square
y	int	-1	square

15

Loader..

•The Loader loads the executable image, generated by linker, into the memory and executes the program

•Loader is part of O/S

•What about the addresses ??

- Program starts at x3000, refers to memory at x3100, etc.
- Does this mean the loader loads exactly into x3000 ??
- Concept of user space
  - Memory management system does the actual mapping from user space to physical addresses

16

Linking

- Linking is the process whereby a program that you wrote is combined with all of the libraries it calls to produce the final executable program.
- To do this the linker needs to find the code for every function that is called in the program and link them together appropriately to produce the final executable binary file.
  - You can get a taste of linking if implemented the calculator program as one large assembly program

18

Makefiles

- The **make** tool was developed to simplify and automate the process of multi-file development.
- The makefile specifies dependencies between various source files in your project and indicates how they should be compiled to produce the final result.
- See the following URL for a simple introduction to Makefiles
  - http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- You will have to create and use makefiles
  - So go through the tutorial

19

Static vs Dynamic Linking

- For most modern OSes, Windows, MacOSX, Linux etc., commonly used functions, printf, rand, open, close etc. are stored in dynamically linked libraries. On Windows these have the suffix .DLL
- The idea here is that instead of copying the function code into every executable program that uses them at compile time the function is instead dynamically linked at run time.
- This saves space, makes for smaller executables and allows the OS maker to change the implementation of those functions in future releases without requiring programmers to recompile or relink their code.
  - However, it does raise the prospect of changing the implementation in such a way that it breaks programs that used to run just fine before you changed the underlying library.
  - *Also a great target for worms and viruses.*

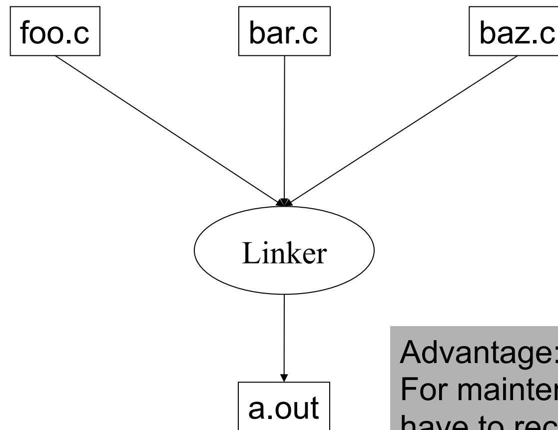
20

Multi File development

- C offers some basic facilities to support multi-file development.
- It is common to write a set of functions that implement a piece of functionality in one C (.c) file and then expose that functionality to other C files that may want to use it as a header file (.h)
  - See for example stdio.h, stdlib.h, string.h etc.
- In order to support this pattern the C compilers support partial compilation where a file that does not have a main routine can be compiled down to an object file containing the code for the routines using the –c flag
  - gcc –c my_functions.c
- The resulting object (.o, .obj) files can, optionally, be archived together into a library, (.a, .lib) that your program can then be linked against.

21

Multiple Files



Advantage:
For maintenance only have to recompile affected files

22