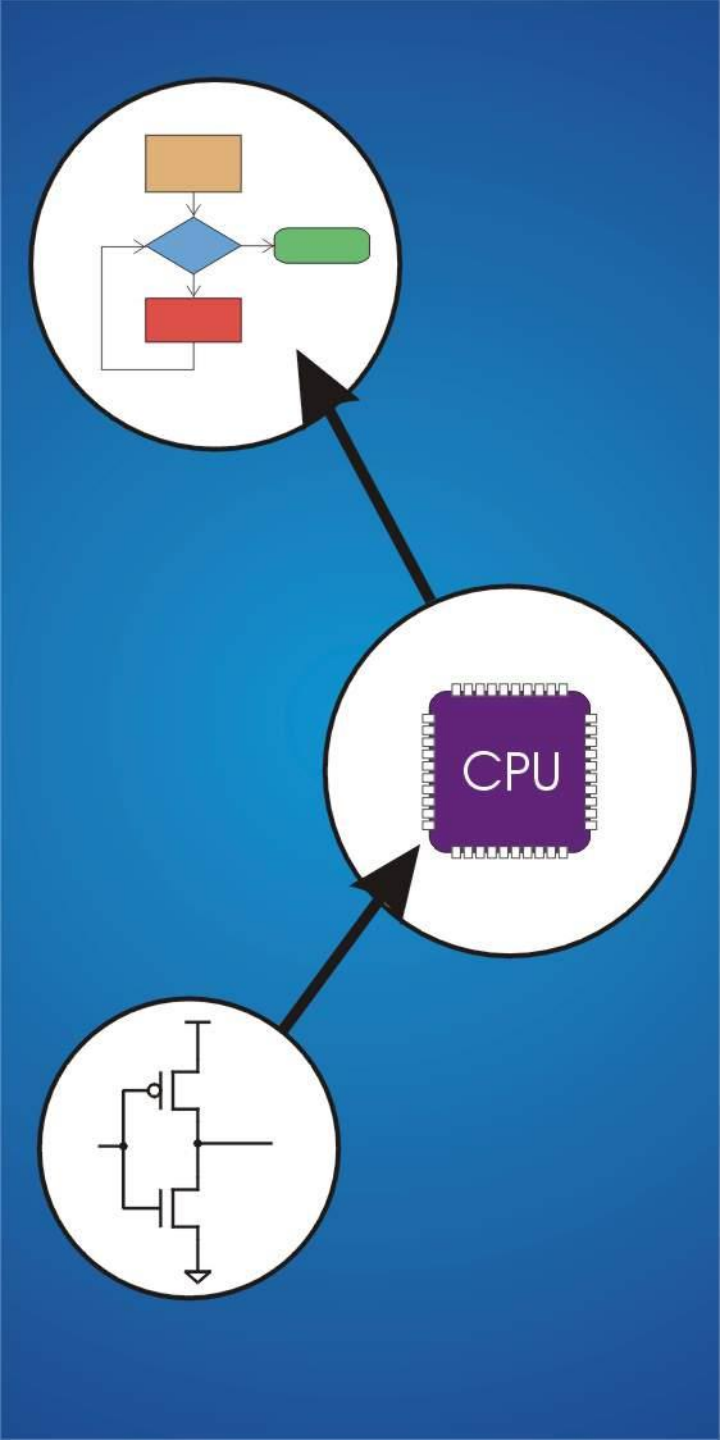


Chapter 11
Introduction to Programming in C

C: A High-Level Language

Gives symbolic names to values

- don't need to know which register or memory location

Provides abstraction of underlying hardware

- operations do not depend on instruction set
- example: can write "a = b * c", even though LC-3 doesn't have a multiply instruction

Provides expressiveness

- use meaningful symbols that convey meaning
- simple expressions for common control patterns (if-then-else)

Enhances code readability

Safeguards against bugs

- can enforce rules or conditions at compile-time or run-time

Compilation vs. Interpretation

Different ways of translating high-level language

Interpretation

- **interpreter = program that executes program statements**
- **generally one line/command at a time**
- **limited processing**
- **easy to debug, make changes, view intermediate results**
- **languages: BASIC, LISP, Perl, Java, Matlab, 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**

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 <u>interpreting</u>, how many arithmetic operations occur?

If <u>compiling</u>, 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?

Compiling a C Program

Entire mechanism is usually called the "compiler"

Preprocessor

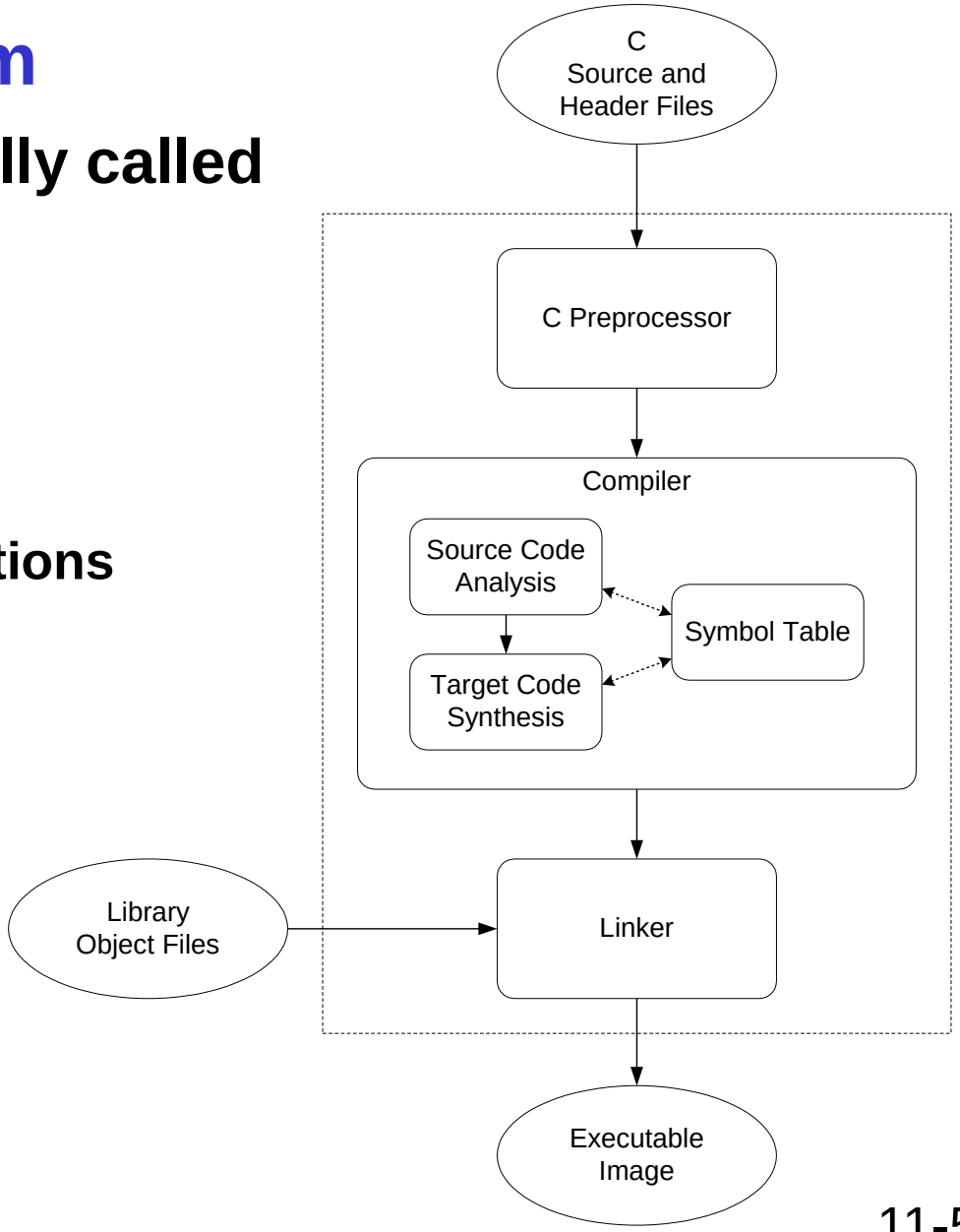
- **macro substitution**
- **conditional compilation**
- **"source-level" transformations**
  - **output is still C**

Compiler

- **generates object file**
  - **machine instructions**

Linker

- **combine object files (including libraries) into executable image**



Compiler

Source Code Analysis

- "front end"
- parses programs to identify its pieces
  - variables, expressions, statements, functions, etc.
- depends on language (not on target machine)

Code Generation

- "back end"
- generates machine code from analyzed source
- may optimize machine code to make it run more efficiently
- very dependent on target machine

Symbol Table

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

A Simple Java Program

```
import java.lang;
public class Simple {

  /* Function: main */
  /* Description: count down from user input to STOP */
  public static void main(String[] args)
  {
    /* variable declarations */
    static final int STOP = 0;
    int counter;  /* an integer to hold count values */
    int startPoint; /* starting point for countdown */

    /* prompt user for input, assumes scanner */
    System.out.printf("Enter a positive number: ");
    startPoint = in.nextInt();

    /* count down and print count */
    for (counter=startPoint; counter>=STOP; counter--)
      System.out.printf("%d\n", counter);
  }
}
```

C vs. Java: some differences

Java	C
1990s	1970s
Object Oriented	Function oriented
Compilation: byte code	Compilation: machine code
No pointers	Pointers
Automatic allocation/deallocation	Dynamic memory allocation malloc/free
	Array's don't know their own size
…	…

A Simple C Program

```
#include <stdio.h>
#define STOP 0

/* Function: main */
/* Description: counts down from user input to STOP */
int main(int argc, char *argv[])
{
  int counter;       // an integer to hold count values
  int startPoint;    // starting point for countdown

  /* prompt user for input */
  printf("Enter a positive number: ");
  scanf("%d", &startPoint);  /* read into startPoint */

  /* count down and print count */
  for (counter=startPoint; counter>=STOP; counter--)
    printf("%d\n", counter);

      return 0;
}
```

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.)**

Comments

Begins with `/*` and ends with `*/`

Can span multiple lines

Cannot have a comment within a comment

Comments are not recognized within a string

- **example: "my/*don't print this*/string"**
  would be printed as: my/*don't print this*/string

As before, use comments to help reader, not to confuse or to restate the obvious

`main` Function

Every C program must have a `main()` function:

The main function contains the code that is executed when the program is run.

As with all functions, the code for main lives within brackets:

```
int main(int argc, char *argv[])
{
/* code goes here */
}
```

Java is similar, but C needs the size of array (argc) since C has no length member.

`main` Function

`main()` returns an `int`

- **Really**
- **"I tried `void main()`, and it worked!"**
- **This is an example of undefined behavior, which cannot be refuted by experimentation.**

Variable Declarations

Variables are used as names for data items.

Each variable has a *type*,
which tells the compiler how the data is to be interpreted (and how much space it needs, etc.).

```
int counter;
int startPoint;
```

`int` **is a predefined integer type in C.**

Input and Output

Variety of I/O functions in *C Standard Library*.

Must include `<stdio.h>` to use them.

```
printf("%d\n", counter);
```

- **String contains characters to print and formatting directions for variables.**
- **This call says to print the variable `counter` as a decimal integer, followed by a linefeed (`\n`).**

```
scanf("%d", &startPoint);
```

- **String contains formatting directions for looking at input.**
- **This call says to read a decimal integer and assign it to the variable `startPoint`. (Don't worry about the `&` yet.)**

More About Output

Can print arbitrary expressions, not just variables

```
printf("%d\n", startPoint - counter);
```

Print multiple expressions with a single statement

```
printf("%d %d\n", counter,
        startPoint - counter);
```

Different formatting options:

- `%d` **decimal integer**
- `%x` **hexadecimal integer**
- `%c` **ASCII character**
- `%f` **floating-point number**

Examples

This code:

```
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\n", 43+59);
printf("43 plus 59 in hex is %x.\n", 43+59);
printf("43 plus 59 as a character is %c.\n", 43+59);
```

produces this output:

```
43 is a prime number.
43 + 59 in decimal is 102.
43 + 59 in hex is 66.
43 + 59 as a character is f.
```

Examples of Input

Many of the same formatting characters are available for user input.

```
scanf("%c", &nextChar);
```

- **reads a single character and stores it in nextChar**

```
scanf("%f", &radius);
```

- **reads a floating point number and stores it in radius**

```
scanf("%d %d", &length, &width);
```

- **reads two decimal integers (separated by whitespace), stores the first one in length and the second in width**

Must use ampersand (`&`) for variables being modified.

(Explained in Chapter 16.)

Compiling and Linking

- Various compilers available
  - gcc, c99, c11, clang
  - includes preprocessor, compiler, and linker
  - Warning: some features are implementation dependent!
- Lots and lots of options
  - level of optimization, debugging
  - preprocessor, linker options
  - usually controlled by makefile
  - intermediate files --
    object (.o), assembler (.s), preprocessor (.i), etc.

Remaining Chapters on C

A more detailed look at many C features.

- **Variables and declarations**
- **Operators**
- **Control Structures**
- **Functions**
- **Data Structures**
- **I/O**

Emphasis on how C is implemented by LC-3 assembly language.

Also see C Reference in Appendix D.