

Chapter 11

Introduction to Programming in C

C: A High-Level Language

Gives symbolic names for containers of 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 (can be implemented as an “invisible” subroutine)

Provides expressiveness

- use meaningful symbols that convey meaning
- simple expressions for common control structures

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
- less efficient than compilation
- easy to debug, make changes, view intermediate results
- languages: BASIC, LISP, Perl, Matlab, C-shell

Compilation

- translates statements into machine language
 - does not execute, but creates executable program
- more efficient than interpretation
- change requires recompilation
 - can be harder to debug, since executed code may be different
- languages: C, C++, Fortran, Pascal

A SIDE-BY-SIDE COMPARISON OF COMPILED LANGUAGES AND INTERPRETED LANGUAGES

A look at how compilers and interpreters work, and how their differences affect memory, runtime speed, and computer workload.

	A COMPILER	AN INTERPRETER
Input	... takes an entire program as its input.	... takes a single line of code, or instruction, as its input.
Output	... generates intermediate object code.	... does not generate any intermediate object code.
Speed	... executes faster.	... executes slower.
Memory	... requires more memory in order to create object code.	... requires less memory (doesn't create object code).
Workload	... doesn't need to compile every single time, just once.	... has to convert high-level languages to low-level programs at execution.
Errors	... displays errors once the entire program is checked.	... displays errors when each instruction is run.

Compilation vs. Interpretation

• Interpretation/Compilation Hybrid

- **Java**
 - Java language is compiled into JVM (Java Virtual Machine) byte code.
 - The JVM byte code can be either interpreted or compiled into native machine code by a JIT (Just in Time) or AOT (Ahead of Time) compiler.
- **Python**
 - Python program is converted into byte code like Java language – PVM (Python Virtual Machine) for CPython (the reference implementation of Python) and JVM for Jython.
 - The byte code can be either interpreted or compiled into native machine code by a JIT (Just in Time) or AOT (Ahead of Time) compiler.

Compilation vs. Interpretation

Consider the following simple program:

- `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, 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

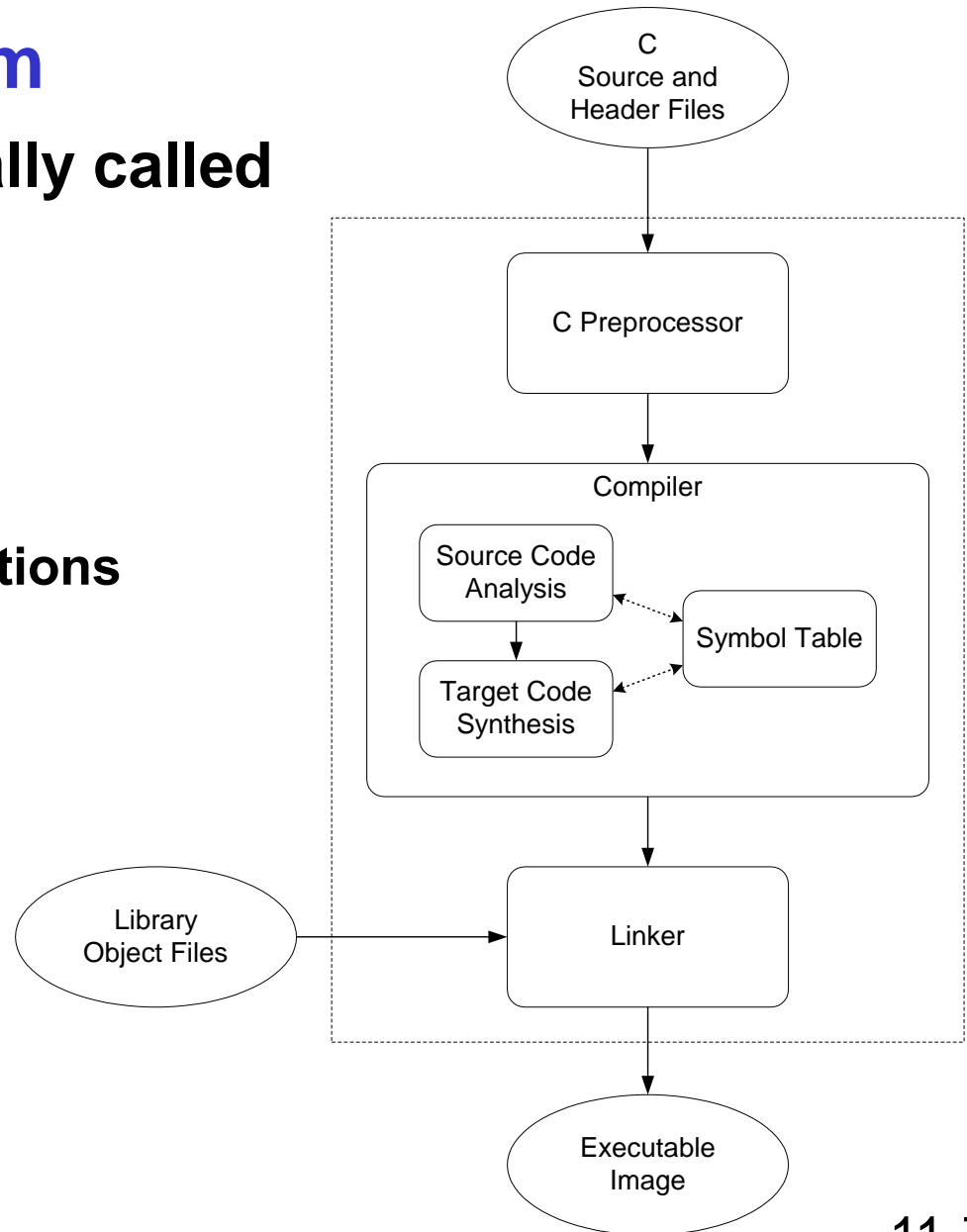
- macro substitution
- conditional compilation
- “source-level” transformations
 - output is still C

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 C Program

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

/* Function: main */
/* Description: counts down from user input to STOP */
main()
{
    /* variable declarations */
    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);
}
```

Preprocessor Directives

`#include <stdio.h>`

- Before compiling, copy contents of header file (stdio.h) into source code.
- Header files typically contain constants, 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 function called `main()`.

This is the code that is executed when the program is run.

The code for the function lives within brackets:

```
main()  
{  
    /* code goes here */  
}
```

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

- **cc, gcc**
- **includes preprocessor, compiler, and linker**

Lots and lots of options!

- **level of optimization, debugging**
- **preprocessor, linker options**
- **intermediate files --
object (.o), assembler (.s), preprocessor (.i), etc.**

Remaining Chapters

A more detailed look at many C features.

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

Emphasis on how C is converted to LC-3 assembly language.

Also see C Reference in Appendix D.

꼭 기억해야 할 것

- Interpretation vs. Compilation and its hybrids
- High-Level Programming Language
 - provides a high-level abstraction of machine language
 - variables (registers and memory locations)
 - operations (arithmetic / logic instructions)
 - control structures (branch / jump instructions)
 - sequence
 - conditional
 - iterative