Introduction to
the C language

(Textbook chapter 11)

The C language - history

- Late '60s: MIT, GE, and Bell Labs partner up to build MULTICS, to provide computational power on a grid, just like electrical power, using the B programming language
- Early '70s: from B, Dennis Ritchie develops C, used by Ken Thompson to rewrite UNIX (at Bell Labs)
- "The white book" by Brian Kernighan and Dennis Ritchie:
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

C: A High-Level Language (cont.)

- 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
Interpretation

- Interpretation and compilation are different ways of translating high-level languages.

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

Compilation

- **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
  - compilers can optimize
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, 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? (Ex. 11.8)

Compiling a C Program

- The 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

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

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

• Begin with `/*` and end 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`

• Use comments to help the reader understand and make the program look "nice".
(Think about yourself trying to figure out what your own program does in two years time...!)
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 curly braces:

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

Structure of a C program

- A C program is a set of functions, all at the same level, always including `main()` - the execution starting point.

```c
/********************
main()
{
}
/********************
functionA()
{
}
/********************
functionB()
{
}
/********************
```
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.).
• \texttt{int} \texttt{counter;}
• \texttt{int} \texttt{startPoint;}
• \texttt{int} is a predefined integer type in \texttt{C}.

Input and Output

• Variety of I/O functions in \texttt{C Standard Library}.
• Must include \texttt{<stdio.h>} to use them.

\texttt{printf("\%d\n", \texttt{counter};}}
• This call says to print the variable \texttt{counter} as a decimal integer, followed by a linefeed (\texttt{\\textbackslash n}).
• The string contains characters to print as well as formatting directions for variables.

\texttt{scanf("\%d", \&\texttt{startPoint});}
• This call says to read a decimal integer and assign it to the variable \texttt{startPoint}. (The \& means “the address of \texttt{startPoint}, it is a pointer.)
• String contains formatting directions for looking at input.
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:
  
  ```c
  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:
  
  ```c
  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);
  ```
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 one in `length` and the second in `width`

- Must use ampersand (`&`) for variables, to pass the pointer to the variable.

Compiling and Linking

- Various compilers available
  
  - `cc`, `gcc` (in UNIX)
    - include 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.

- Ex:
  
  `>gcc -O3 -o myProg myProg.c`
Recommended exercises

- Ex 11.1 to 11.8 good, simple review questions
- Ex 11.10, 11.11, 11.12 simple code samples