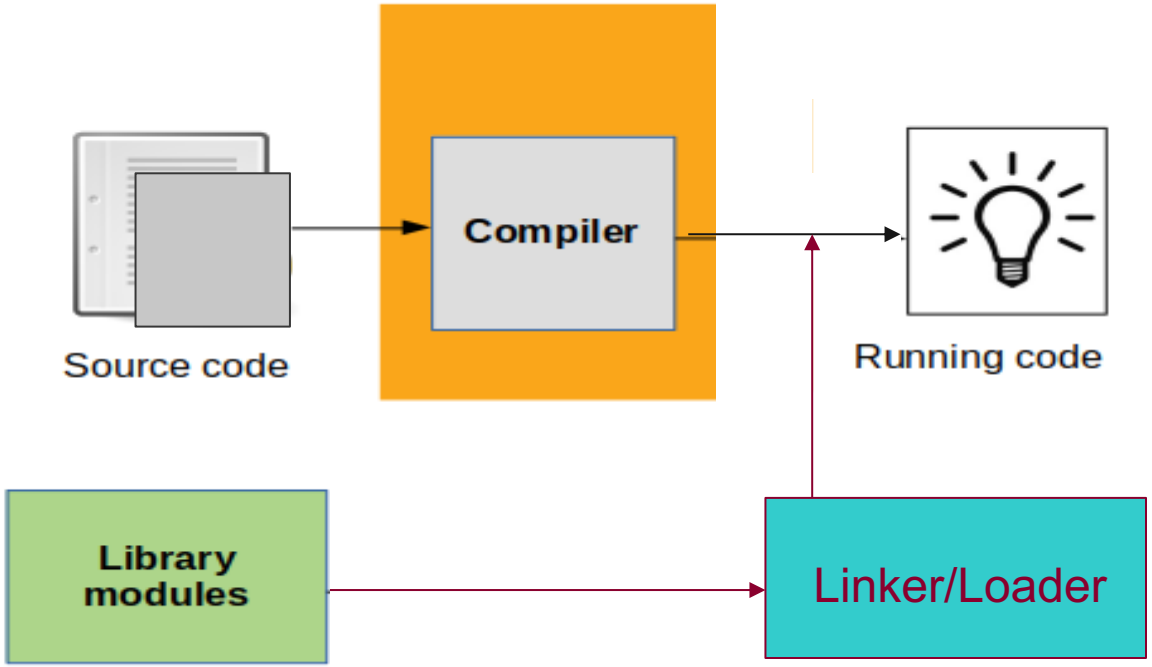
(1 - 2) C Language Elements
H&K Chapter 2

Instructor – Beiyu Lin
CptS 121 (May 7th, 2019)
Washington State University

Review - Algorithm

- Computer Science?
- Algorithm?
  - A well ordered collections
  - Unambiguous and effectively computable operations
  - Producing a result
  - Halts in a finite amount of time
- Use in daily life?
  - count the number of people in a room

Review – High Level Language



Introducing C… (1)

- Developed in 1972 by Dennis Ritchie (b. 1941 – 2011) at AT&T Bell Labs
- Considered an imperative procedural programming language versus an object-oriented language (like Java, C++, C#)
- Originally designed as language in which to program UNIX operating system

Introducing C… (2)

- Originally geared toward operating systems programming
- A very popular language from which many other languages are derived
- Has evolved into a general-purpose programming language
- Has a reputation for being on the low end of the "high-level" programming language spectrum

Introducing C… (3)

- Most computer architectures support a C compiler
- Designed to support cross-platform programming
- Available on platforms ranging from embedded systems to supercomputers
- Great for manipulating low level memory constructs

C Language Elements (1)

- Implementation (in C)

```
#include <stdio.h> /* Needed for printf (), scanf () */
#define PI 3.14159 /* Constant macro */

int main (void)
{
        int height = 0, radius = 0;
        double volume = 0.0;

        printf ("Enter height of cone as integer: "); /* Displays prompt message */
        scanf ("%d", &height); /* Gets the value from the user/keyboard */
        printf ("Enter radius of base of cone as integer: ");
        scanf ("%d", &radius);

        /* Compute the volume of the given cone */
        volume = ((double) 1 / 3) * PI * radius * radius * height ;

        /* Display the resultant volume of the given cone */
        printf ("Volume of cone with radius %d and height %d is %lf.\n", radius, height, volume);

        return 0;
}
```

Comment

Preprocessor Directive

Variable Declaration

Reserved Word

Standard Identifier / Library Function

Special Symbol / Operator

Punctuation

C Language Elements (2)

- Preprocessor Directives
  - Begin with "#“
  - Do NOT end with a semicolon “;”
  - Tell the preprocessor to modify the program before compilation
    - A *preprocessor* is a system program that modifies a C program before compilation
- ANSI (American National Standards Institute) C defines several standard libraries
  - To use a library, you must include its header file in your code using `#include`
  - A *library* is a collection of functions and symbols needed to perform specific tasks or operations
- `#define` can be used to instruct preprocess to replace each occurrence of a textual constants (e.g., `PI`) with a value (e.g., 3.14159)
  - *Convention: constants are in all caps*

C Language Elements (3)

- Function `main`
  - All C programs must define a main function
  - It is the place where program execution begins
  - Contains a series of declarations and executable statements separated by punctuation to help compiler translate the statements
  - "Drives" the rest of the program
- Reserved words
  - Always in lowercase
  - Have special meaning (e.g., `int`, `double`)
  - Show up in blue within MS Visual Studio

C Language Elements (4)

- Standard Identifiers
  - Have special meaning in C
  - Represent names of operations (i.e. `printf` and `scanf`)
  - Can be redefined, but not recommended
  - If redefined, can't be used for original purpose

C Language Elements (5)

- User-Defined Identifiers
  - Name memory cells used for computations ("variables")
  - Name our own custom algorithms (functions)
  - Must consist only of letters, numbers, and underscores
  - Must not begin with digits
  - Must not conflict with reserved words
  - Should not redefine standard C library identifiers

C Language Elements (6)

- Notes
  - C is case sensitive (pay attention to upper and lower case)
  - Choose identifier names wisely
    - They should be meaningful, indicating the role of the variables they are naming
      - E.g., *average*, not *a*
    - Make them descriptive but not excessively long
    - Make sure that identifier names are sufficiently different, so that you don't accidentally mix them up
    - Use underscores (_) or camel caps/case to distinguish between words, i.e. average_score or averageScore

Variable Declarations

- Declaring a variable reserves memory space for a value
- It also associates a name with a memory cell
  - These names are user-defined identifiers
- Variable data type precedes variable name:
  - `double miles; /* will store # of miles */`
  - `int count; /* will store # of zeros found */`
  - `char initial; /* will store first initial */`
- Every variable must be declared!
- All variables in C must be declared before any executable statements!!! i.e. before statements like printf (), scanf (), etc.

Data Types (1)

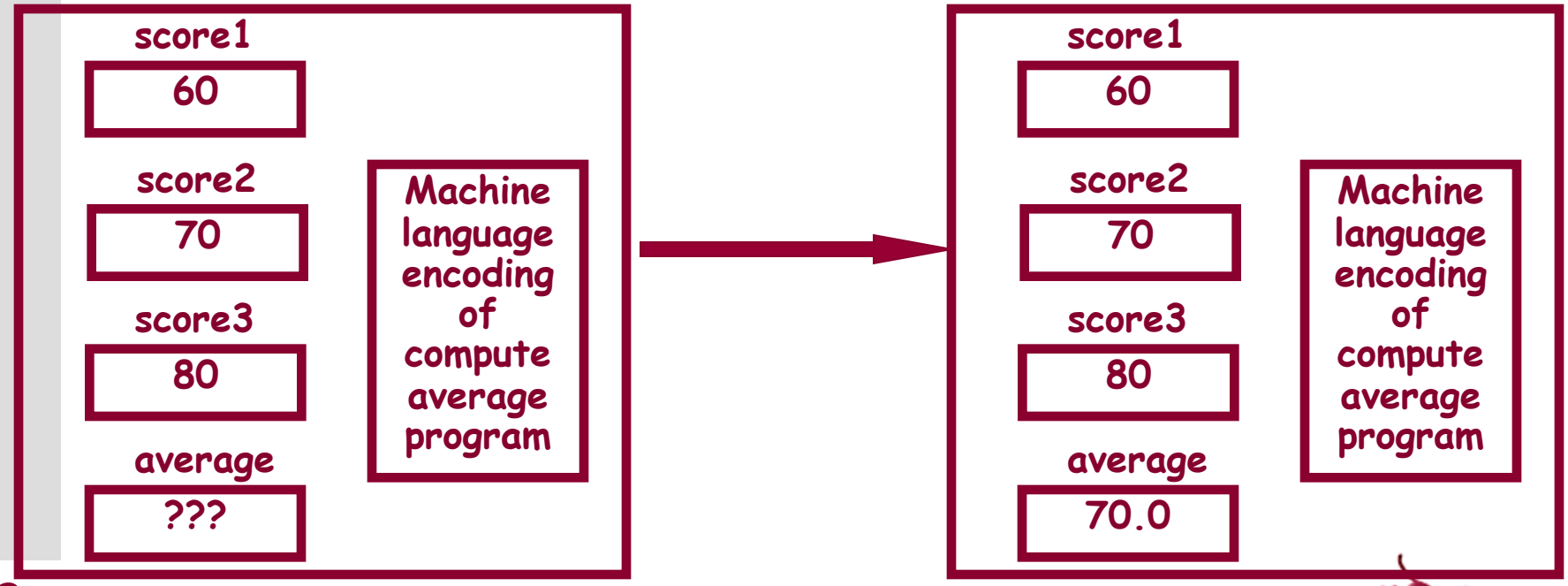
- Data type = set of values + set of operations on those values
- Can be associated with variables (changeable) or constants (non-changeable)
- C defines several standard data types
  - `int`
    - Models the integers (at least -32767 to 32767 (16-bit), but most machines define 32-bit integers)
    - Several operations are defined, including +, -, *, /, and % (mod), and comparisons (>, <, <=, >=, ==, !=)

Data Types (2)

- C defines several standard data types (cont.)
  - `double`
    - models real numbers (must include a decimal point)
    - not all real numbers can be modeled because of space limitations (64 bits)
    - Several operations are defined, including +, -, *, and /, and comparisons (>, <, <=, >=, ==, !=)
    - Scientific notation can be used to write double values (e.g., 15.0e-3, 314e-01)
  - `char`
    - Models individual ASCII characters (8 bits)
    - Can compare characters (>, <, <=, >=, ==, !=)
    - When defining char variables in a program, use single quotes: `'c',' ','"'`, etc.

Executable Statements (1)

- Follow variable and constant declarations
- Do the work of the algorithm by transforming inputs into outputs



Executable Statements (2)

- Assignment statements
  - Store a computational result into a variable
  - The = operator does the assignment
  - The *, -, +, /, operators perform the computation
  - Example:
    volume = (1/3) * PI * radius * radius * height; /* always 0 because of 1 / 3 */
  - Note: above will yield an int for 1 / 3 instead of a double, so we need to perform a *cast:*

```
volume = ((double) 1 / 3) * PI * radius * radius * height;
```


Executable Statements (3)

- Assignment Statements (cont.)
  - We can also assign the value of one variable to another:

```
y = x; /* assigns y to the value of x  */
y = -x; /* computes the negation of x,
  assigning
           it to y */
```

- Input/Output Statements
  - It is extremely useful to obtain input data interactively from the user, and to display output results to the user
  - How can this be done in C?

Executable Statements (4)

- Input/Output Statements (cont.)
  - The C input/output library defined in `<stdio.h>` (which you must `#include` if you want to use it) includes several *functions* that perform input and output
  - <Digression: Functions>
    - A function is a set of statements that perform a task.
    - A function performs the task, hiding from you the details of *how* it performs the task (they're irrelevant)
    - We'll study functions in depth!

    <End Digression>

Executable Statements (5)

- Input/Output Statements (cont.)
  - The `printf` function can be used to output results to the user's display
  - Example

```
printf("The student's average is %f.",average);
```

  - Notes
    - `%f` is a placeholder for `double` (`%lf` in `scanf`); `%d` is a placeholder for `int`; and `%c` is a placeholder for `char`
    - Multiple placeholders can exist within a single format string (see next example)

Executable Statements (6)

- Input/Output Statements (cont.)
  - Another example:

```
printf("Wow, %c%c%c%c %d sure is
cool!\n",letter_1, letter_2, letter_3,
letter_4, course_num);
```

would display

```
Wow, CptS 121 sure is cool!
```

Assuming that `letter_1` is `'C'`, `letter_2` contains `'p'`, `letter_3` contains `'t'`, `letter_4` contains `'S'`, and `course_num` contains `121`.

Note that `'\n'` prints a newline (return), causing the cursor to advance to the next line.

Executable Statements (7)

- Input/Output Statements (cont.)
  - The `scanf` function reads an input value from the keyboard
  - Its format is similar to that of `printf`
  - Example:
    ```
    scanf("%d",&score1);
    ```
    forces the program to pause until the user enters a value from the keyboard and hits the return key.
  - Notes
    - `scanf` interprets the input as an `int` (`%d` is a placeholder for an `int`).
    - The `int` value is then stored into the variable `score1`. The `&` ("address of") operator tells `scanf` *where* to store the inputted value.
    - If the `&` were omitted, `scanf` would only know the value of `score1`, not where in memory it is located

Executable Statements (8)

- Input/Output Statements (cont.)
  - `scanf` should always be used in conjunction with a `printf` statement that displays a prompt, so that the user knows that an input value is expected
  - Example:

```
printf ("Enter radius of base of cone as integer: ");
scanf ("%d", &radius);
```

  - Notes
    - User may separate values with either spaces or returns
    - If more values are entered than are specified in a `scanf`, they are saved ("buffered") for the next call to `scanf`

Executable Statements (9)

- `return` statement
  - In C, most functions, including `main`, *return* a value (just like a mathematical function)
  - The `return` statement specifies the value to be returned
  - The type of the return value must match the declared return value of the function
    - `int main(void) { … }` indicates that an `int` is to be returned; hence, `main` must return an `int`
  - In the `main` function, `return(0)` (a return value of 0) tells the operating system that the program executed without error

General Form of a C Program (1)

- General template of a C program is as follows:

```
comment block
preprocessor directives
main function heading
{
    declarations
    executable statements
}
```


General Form of a C Program (2)

- Statements may extend over a single line (return is treated as a space)
  - Exception: Do not break up a a statement in the middle of a reserved word, constant, or quoted format string
- More than one statement can go on a single line
- The programs you write should adhere to good C style
  - Makes them more readable, but does not affect compilation

General Form of a C Program (3)

- What is good C style?
  - When you write C programs for this class, refer to the "Recommended C Style and Coding Standards" found here
  - Insert blank space before and after commas and operators such as +, =, /, *

General Form of a C Program (4)

- What is good C style?
  - Liberally comment your programs
    - Document the purpose of each variable where it is declared:
    - Begin programs with a header section that indicates
      - Programmer's name
      - Date of current version
      - Brief description of what program does
      - The course ("CptS 121")
      - Assignment number (e.g., "Programming Assignment #1")

General Form of a C Program (5)

- What is good C style? (cont.)
  - Liberally comment your programs (cont.)

```
/*
 * Programmer:
 * Class: CptS 121, Spring 2018
 * Programming Assignment #0
 * Date:
 *
 * Description: This program computes the
 * volume of a cylinder.
 */
```


Next Lecture…

- More C language elements: Arithmetic expressions, number formatting, file input and output

References

- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (8th Ed.)*, Addison-Wesley, 2016

Collaborators

- Chris Hundhausen
- Andrew O'Fallon