Problem Solving and Program Design - Chapter 2

Cory L. Strope

- C Language Elements
- Variable Declarations and Data Types
- Executable Statements
- General Form of a C Program
- Arithmetic Expressions
- Formatting Numbers in Program Output
- Interactive Mode, Batch Mode, and Data Files
- Common Programming Errors

Overview of C Programming

This chapter introduces C – a high-level programming language developed in 1972 by Dennis Ritchie at AT&T Bell Laboratories. This chapter describes the elements of a C program and the types of data that can be processed by C. It also describes C statements for performing computations, for entering data, and for displaying results.

Mistakes in Chapter 2

- Page 35, Figure 2.1 Line 9 needs to end with */
- Page 39, 1Letter needs to say "begins with a digit"

C Language Elements

- Preprocessor Directives
- Syntax Displays for Preprocessor Directives
- "int main()" Function
- Reserved Words
- Standard Identifiers
- User-Defined Identifiers
- Uppercase and Lowercase Letters
- Program Style

Preprocessor Directives

- The *C preprocessor* modifies the text of the C program before it is passed to the compiler.
- Preprocessor directives are C program lines beginning with a # that provide instructions to the C preprocessor.
- Preprocessor directives begins with a #, either #include or #define.
- Predefined libraries are useful functions and symbols that are predefined by the C language (standard libraries).

#include and #define

- `#include` <library> gives the program access to a library
  - `stdio.h` (standard input and output) has definitions for input and output, such as `printf` and `scanf`.
- `#define` *NAME value* associates a constant macro.
  - `#define KMS_PER_MILE 1.609`
  - `#define PI 3.14159`

Comments

Comments provide supplementary information making it easier for us to understand the program, but comments are ignored by the C preprocessor and complier.

- `/* */` - anything between them with be considered a comment, even if they span multiple lines.
- `//` - anything after this and before the end of the line is considered a comment.

Function main

- The point at which a C program begins execution is the *main* function:

```
int
main(void)
```

- **Every** C program must have a main function.
- The main function (and every other function) body has two parts:
  - Declarations - tell the compiler what memory cells are needed in the function
  - Executable statements - (derived from the algorithm) are translated into machine language and later executed

Function main

- The main function contains *punctuation* and *special symbols*
  - Punctuation - Commas separate items in a list, semicolons appear at the end of each statement
  - Special Symbols - * and $=$, braces ($\{$, $\}$) mark the beginning and end of the body of function main.

Reserved Words

- A word that has special meaning in C
  - `int` - Indicates the main function returns an integer value,
  - `double` - Indicates the memory cells used to store these values will store real numbers.
- Always lower case,
- Can not be used for other purposes,
- Page 819 Appendix E has a full listing of reserved words (ex: `double, int, if , else, void, return` ...)

Standard Identifiers

- Standard identifiers have a special meaning in C (assigned by standard libraries).
- Standard identifiers can be redefined and used by the programmer for other purposes
  - Not recommended If you redefine a standard identifier, C will no longer be able to use it for its original purpose.
- Examples - `printf, scanf`

User-Defined Identifiers

We choose our own identifiers to name memory cells that will hold data and program results and to name operations that we define (more on this in Chapter 3).

- An identifier must consist only of letters, digits, and underscores.
- An identifier cannot begin with a digit.
- A C reserved word cannot be used as an identifier.
- An identifier defined in a C standard library should not be redefined.

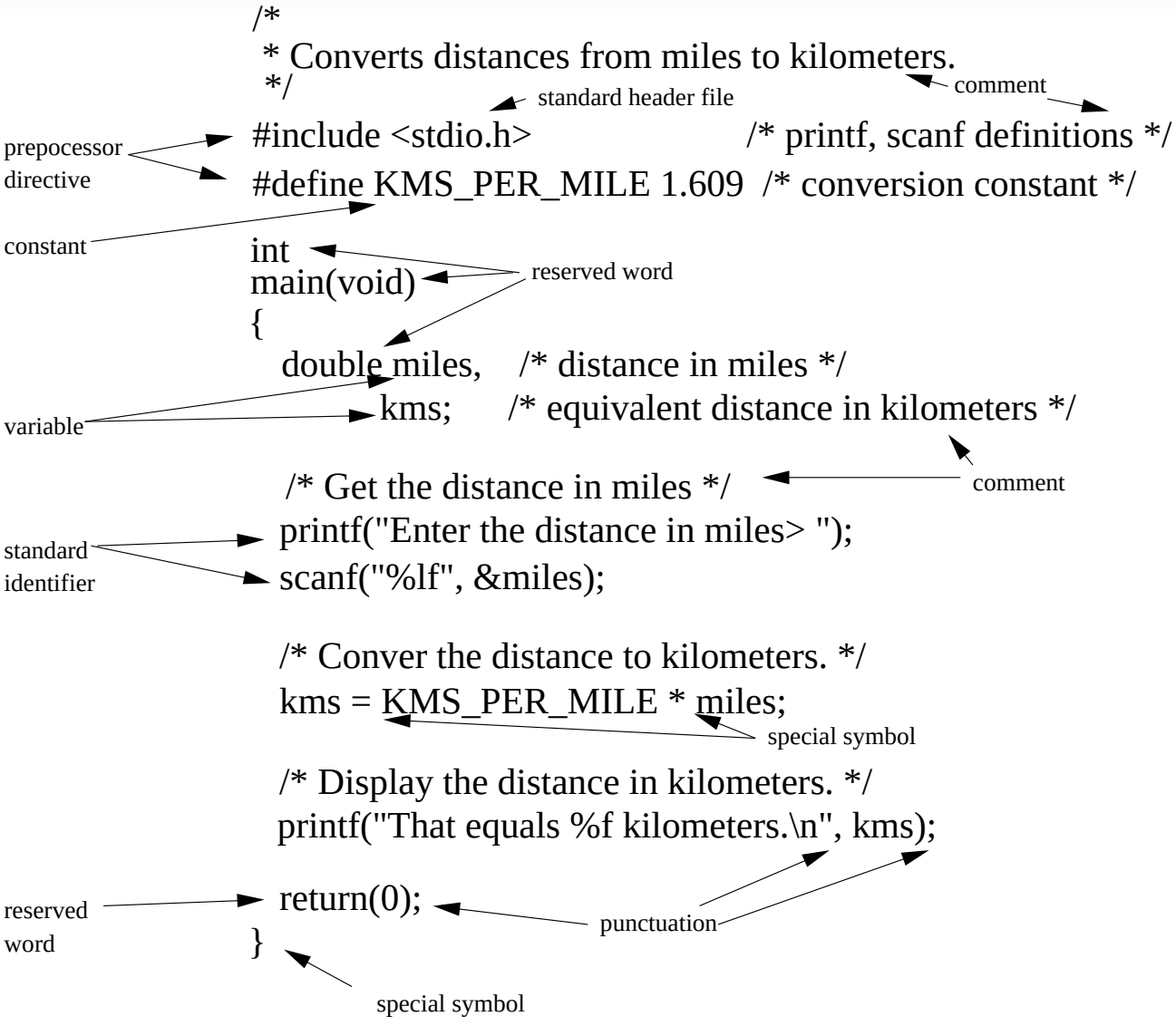
User-Defined Identifiers

- Examples: letter_1, Inches, KMS_PER_MILE
- Some compliers will only see the first 31 characters
- Uppercase and lowercase are different (Variable $\neq$ variable $\neq$ VARIABLE)
- choosing identifer names:
  - Choose names that mean something,
  - should be easy to read and understand,
  - shorten only if possible
- Don't use Big, big, and BIG as they are easy to confuse
- All caps are usually used for preprocessor-defined identifiers (#define)

Program Style

A program that "looks good" is easier to read and understand than one that is sloppy (i.e. good spacing, well-named identifiers).

In industry, programmers spend considerably more time on program maintenance than they do on its original design or coding.


/*
* Converts distances from miles to kilometers.
*/
comment
standard header file
prepocessor directive
#include <stdio.h> /* printf, scanf definitions */
#define KMS_PER_MILE 1.609 /* conversion constant */
constant
int
main(void)
reserved word
{
double miles, /* distance in miles */
variable
kms; /* equivalent distance in kilometers */
/* Get the distance in miles */
comment
standard identifier
printf("Enter the distance in miles> ");
scanf("%lf", &miles);
/* Conver the distance to kilometers. */
kms = KMS_PER_MILE * miles;
special symbol
/* Display the distance in kilometers. */
printf("That equals %f kilometers.\n", kms);
reserved word
return(0);
punctuation
}
special symbol

Variable Declarations and Data Types

- Variable Declaration
- Data Types

Variables Declarations

- Variables - a name associated with memory cells (`miles`, `kilometers`) that store a programs input data. The value of this memory cell can change.
- Variable declarations - statements that communicate to the compiler that names of variables in the program and the kind of information stored in each variable.
  - Example: `double miles, kms;`
  - Each declaration begins with an identifier to indicate the type of data
  - Every variable used must be declared

Data Types

- Data Types: a set of values and a set of operations that can be used on those values. In other words, it is a classification of a particular type of information.
  - `int`: integers from -32767 to 32767
  - `double`: uses a decimal point, 3.14159 or 1.23e5 = 123000.0
  - `char`: an individual character values with single quotes around it – a letter, a digit, or a special symbol (can do arithmetic operations on them, but be careful) (i.e. 'a' + 'a' would equal $\widehat{\text{A}}$, 194, or 2.189514 depending on if we are outputting a `char`, `int`, or `double`)
- The idea is to give semantic meaning to 0's and 1's.

Executable Statements

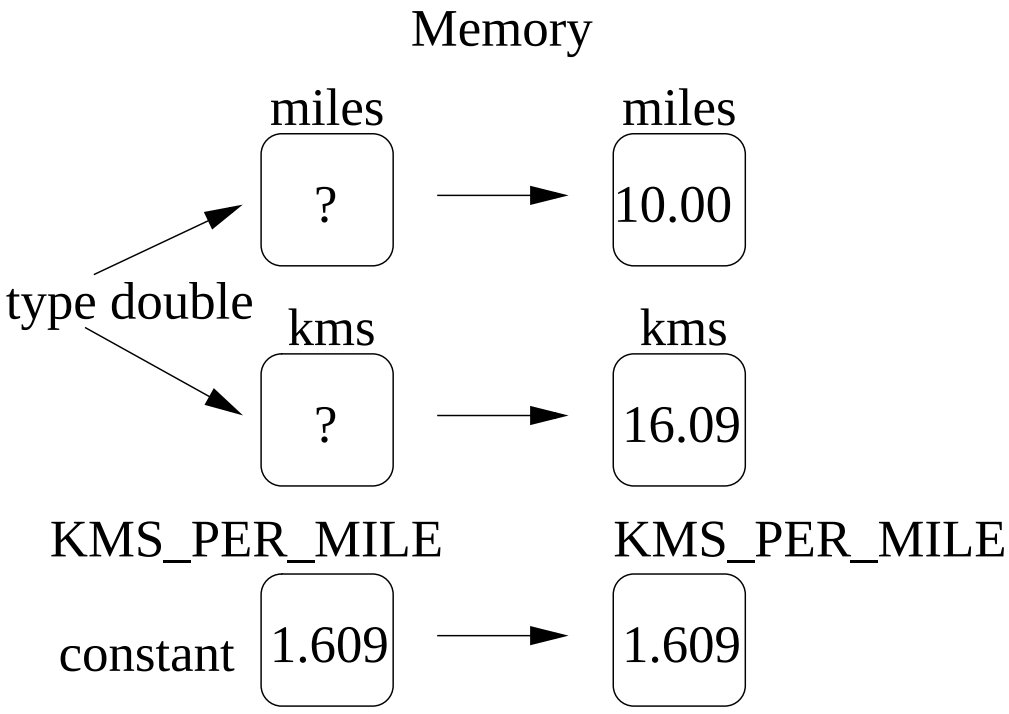
- Assignment Statements
- Input/Output Operations and Functions
- printf Function
- scanf Function
- return Statement

Assignment Statements

- Assignment statements - stores a value or a computational result in a variable,
- Used to perform most arithmetic operations in a program.
- Form: `variable = expression;`
  - `kms = KMS_PER_MILE * miles;`

The assignment statement above assigns a value to the variable kms. The value assigned is the result of the multiplication of the constant macro KMS_PER_MILE (1.609) by the variable miles.

Memory of Program



Assignments Cont.

- In C, the symbol = is the assignment operator.
  - Read as "becomes", "gets", or "takes the value of" rather than "equals"
  - In C, == tests equality.
- In C you can write assignment statements of the form

$$sum = sum + item;$$

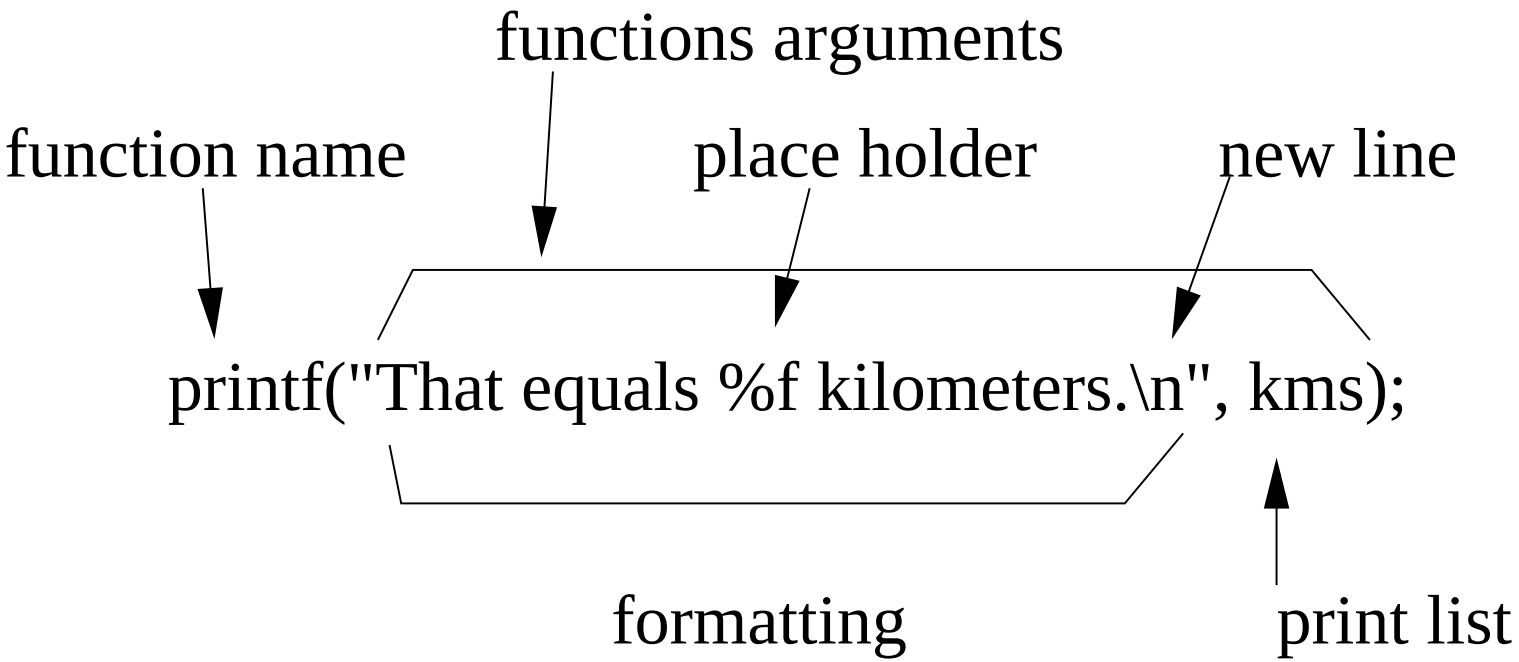
where the variable sum appears on both sides of the assignment operator. This is obviously not an algebraic equation, but it illustrates a common programming practice. This statement instructs the computer to add the current value of sum to the value of item; the result is then stored back into sum.

Input/Output Operations and Functions

- input operation - data transfer from the outside world into memory.
- output operation - An instruction that displays program results to the program user.
- input/output functions - special program units that do all input/output operations. Common I/O functions found in <stdio.h>
- Function call - in C, a function call is used to call or activate a function.
    - Analogous ordering food from a restaurant. You (the calling routine) do not know all of the ingredients and procedures for the food, but the called routine (the restaurant) provides all of this for you.

printf Function

Included in the <stdio.h> library.



Place Holder

A placeholder always begins with the symbol %. Also the newline escape sequence \n. Format strings can have multiple placeholders.

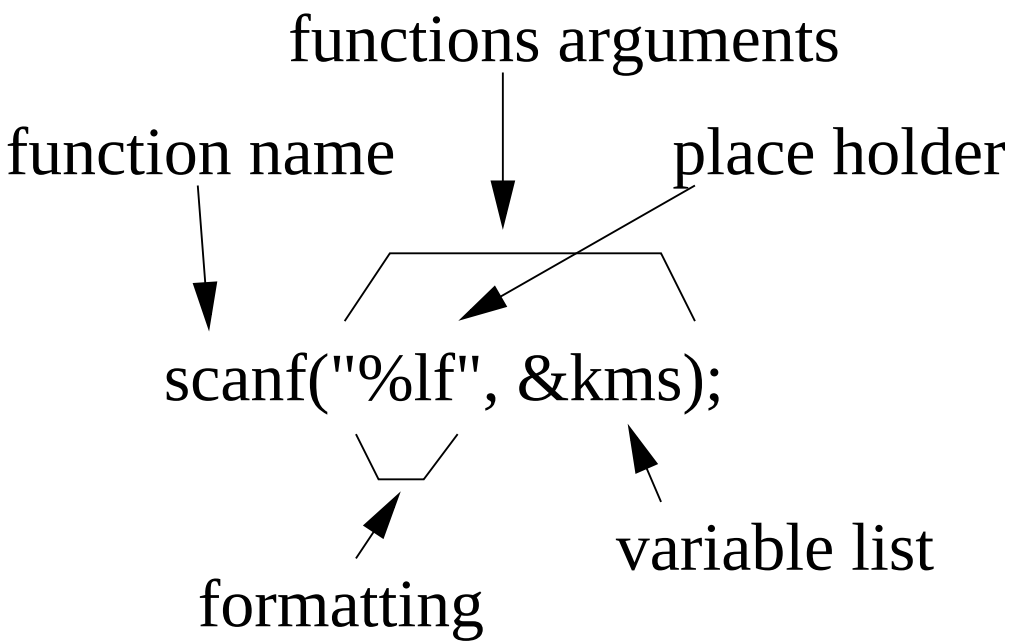
Placeholder	Variable Type	Function Use
%c	`char`	printf/scanf
%d	`int`	printf/scanf
%f	`double`	printf
%lf	`double`	scanf

Displaying Prompts

When input data is needed in an interactive program, you should use the `printf` function to display a **prompting message**, or **prompt**, that tells the program user what data to enter.

```
printf("Do you have any questions?  ");
```

scanf


functions arguments
function name
place holder
scanf("%lf", &kms);
formatting
variable list

Return Statement

```
return 0;
```

This statement returns a 0 to the Operating System to signify that the program ended in a correct position. It does not mean the program did what it was suppose to do. It only means there was no syntax errors. There still may have been logical errors.

Program Example

```
#include <stdio.h>

int main(void) {
    char first, last;

    printf("Enter you first and last initials");
    scanf("%c %c", &first, &last);

    printf("Hello %c. %c.  How are you?\n", first, last);

    return 0;
}
```

General Form of a C Program

- Program Style
- Comments in Programs

General Form of a C Program

- Programs begin with preprocessor directives that provide information about functions from standard libraries and definitions of necessary program constants.
  - `#include` and `#define`
- Next is the main function.
  - Inside the main function is the declarations and executable statements.

General Form of a C Program

```
proprocessor directives

main function heading {

    declarations

    executable statements
}
```

Program Style - Spaces in Programs

The complier ignores extra blanks between words and symbols, but you may insert space to improve the readability and style of a program.

- You should always leave a blank space after a comma and before and after operators such as $*$, $-$, and $=$.
- Indent the body of the main function, as well as between any other curly brackets.

```
int main(void) {
    {
        {
            { }
        }  // End Level 2
    }  /* End Level 1 */
    return 0;
} // end main
```

Comments in Programs

Use comments to do **Program Documentation**, so it help others read and understand the program.

- The start of the program should consist of a comment that includes programmer's name, date of current version, and brief description of what the program does.
- Include comments for each variable and each major step in the program.
- For any function, make comments to briefly describe the input to the function, the output of the function, and the use of the function.

Comments in Programs

Style:

```
/*
 * Multiple line comments are good
 * for describing functions.
 */

/* This /* is NOT */ ok.  */

/* // ok. */
```

```
/*
 * Calculate and display the difference of two input vals
 *)
#include <stdio.h>
int
main(void) {int X, /* first input value */ x, /* second
  input value */
sum; /* sum of inputs */
scanf("%i%i"; X; x); X+x =sum;
printf("%d + %d = %d\n"; X; x; sum); return(0);}
```

Arithmetic Expressions

- Operators / and % (Read `mod` or `remainder`)
- Data Type of Expression
- Mixed-Type Assignment Statement
- Type Conversion through Cast
- Expressions with Multiple Operators
- Writing Mathematical Formulas in C

To solve most programming problems, you will need to write arithmetic expressions that manipulate type `int` and `double` data. The next slide shows all the arithmetic operators. Each operator manipulates two operands, which may be constants, variables, or other arithmetic expressions.

+, –, *, / can be used with integers or double, whereas % can be used only with integers to find the remainder.

Arithmetic Operator	Meaning	Examples
+	addition	5 + 2 is 7
		5.0 + 2.0 is 7.0
-	subtraction	5 - 2 is 3
		5.0 - 2.0 is 3.0
*	multiplication	5 * 2 is 10
		5.0 * 2.0 is 10.0
/	division	5 / 2 is 2
		5.0 / 2.0 is 2.5
%	remainder	5 % 2 is 1

/

- When applied to two positive integers
  - The division operator (/) computes the integral part of the result of dividing its first operand by its second.
  - For example, 7 / 2 = 3.
  - The reason for this is that C allows the answer to only have the same accuracy as the operands. Thus if both operands are integers, the result will be an integer.
- If one or both operands are double, the answer will be a double.
- According to the book different C implementations differ on dividing by a negative number.
- / is undefined when the second operand is 0. 4 / 0 = ??

%

The remainder operator (%) returns the integer remainder of the result of dividing its first operand by its second.

- Similar to integer division, except instead of outputting integral portion, outputs remainder.
- According to the book % can give different answers when the second operand is negative.
- As with division, % is undefined when the second operand is 0.

Data Type of an Expression

The data type of each variable must be specified in its declaration, but how does C determine the data type of an expression?

- The data type of an expression depends on the type(s) of its operand. If both are of type `int`, then solution is of type `int`. If either one or both is of type `double`, then solution is of type `double`.
- An expressions that has operands of both `int` and `double` is a mixed-type expression, and will be typed as `double`.

For a mixed-type assignment, be aware that the expression is evaluated first, and then the *result* is converted to the correct type.

Type Conversion through Casts

C is flexible enough to allow the programmer to convert the type of an expression by placing the desired type in parentheses before the expression, an operation called a *type cast*.

For example, `(double)5 / 2` is 2.5, not 2 as seen previously.

Expressions with Multiple Operators

- In expressions, we often have multiple operators
  - Equations may not evaluate as we wish them to:
  - Is $x/y * z$ the same as $(x/y) * z$ or $x/(y * z)$?
- **Unary operators** take only one operand, i.e. -5, +3, -3.1415, etc.
- **Binary operators** take two operands, i.e. 3 + 4, 7 / 5, 2 * 6, 4

Rules for Evaluating Expressions

a. *Parenthese rule*: All expressions in parentheses must be evaluated separately. Nested parenthesized expressions must be evaluated from the inside out, with the innermost expression evaluated first.

b. *Operator precedence rule*: Operators in the same expression are evaluated in the following order:
unary +,- first
*,/,& next
binary +,- last

c. *Associativity rule*: Unary operators in the same subexpression and at the same precedence level are evaluated right to left. Binary operators in the same subexpression and at the same precedence level are evaluated left to right.

$$x = -5 * 4/2 * 3 + -1 * 2$$

$$x = (((-5 * 4)/2) * 3) + (-1 * 2)$$

$$x = -32$$

Writing Mathematical Formulas in C

You may encounter two problems in writing a mathematical formula in C.

- Multiplication often can be implied in a formula by writing two letters to be multiplied next to each other such as (2a). The equivalent instruction in C is $2 * a$.
- When dealing with division, we often have $\frac{a+b}{c+d}$, which should be written $(a + b)/(c + d)$.

Formatting Numbers in Program Output

- Formatting Values of Type `int`
- Formatting Values of Type `double`
- Program Style

C displays all numbers in its default notation unless you instruct it to do otherwise.

Formatting Values of Type ***int***

Specifying the format of an integer value displayed by a C program is fairly easy. You simply add a number between the % and d of the %d placeholder in the `printf` format string. This number specifies the field width - the number of columns to use for the display of the value.

For example

```
int  x = 2345;
printf("6 spaces: %6d, ",x);
printf("4 spaces: %4d, ",x);
printf("2 spaces: %2d, ",x);
```

```
  2345, 2345, 2345
```

If the field width is smaller than the number being printed, the field width is ignored.

Formatting Values of Type double

Printing for `double`'s is similar for `int`'s, but now we also need to specify the number of digits after the decimal that are printed. The format is: %*n*.*m*f, where *n* is the field width (need to be large enough to accomodate all digits, both before and after the decimal point), and *m* is the number of digits after the decimal to be printed.

Thus if we have 3.141592 and we had a %.2f we would get 3.14, there is nothing in front of the . so the field width defaults to the size of the number, the 2 after the . means to print 2 digits after the decimal (14).

To get rid of the leading blanks, delete the number between % and d, and delete the number between % and .*m*f.

Interactive Mode, Batch Mode, and Data Files

- Input Redirection
- Program Style
- Output Redirection
- Program-Controlled Input and Output Files

Definitions

- active mode - the program user interacts with the program and types in data while the program is running.
- batch mode - the program scans its data from a data file prepared beforehand instead of interacting with its user.

Input Redirection

In the next frame we will see the miles-to-kilometers conversion program rewritten as a batch program. We assume here that the standard input device is associated with a batch data file instead of with the keyboard. In most systems, this association can be accomplished relatively easily through *input/output redirection* using operating system commands.
Instead of calling the program such as:

```
c:> conversion
```

We would call it as:

```
c:> conversion < mydata
```

```c
/* Converts distance from miles to kilometers */

#include <stdio.h>
#define KMS_PER_MILES 1.609

int main(void) {
    double miles, kms;

    scanf("%lf", &miles);
    printf("The distance in miles is %.2f.\n", miles);

    kms = KMS_PER_MILES * miles;

    printf("That equals %.2f kilometers.\n", miles);

    return 0;
}
```

Echo Prints vs. Prompts

In the above program, `scanf` gets a value for *miles* from the first (and only) line of the data file. Because the program input comes from a data file, there is no need to precede this statement with a prompting message. Instead, we follow the call to `scanf` with the statement

```
printf("The distance in miles is %.2f.\n",miles);
```

This statement *echo prints* or displays the value just stored in `miles` and provides a record of the data manipulated by the program. Without it, we would have no easy way of knowing what value `scanf` obtained for `miles`. Whenever you convert an interactive program to a batch program, make sure you replace each prompt with an echo print after the `scanf`.

Output Redirection

You can also redirect the output of the program to a file instead of the screen. Then you can send the output file to the printer to obtain a hard copy of the program output.
The command line:

```
c:> conversion > outfile
```

sends the output of the program conversion to the `outfile`. Now we can do both input and output redirection by using:

```
c:> conversion < input_file > output_file
```

Program-Controlled Input and Output Files

As an alternative to input/output redirection, C allows a program to explicitly name a file from which the program will take input and a file to which the program will send output. The steps needed to do this are:

1. Include `stdio.h`
2. Declare a variable of type `FILE *`.
3. Open the file for reading, writing or both.
4. Read/write to/from the file.
5. Close the file.

```
#include <stdio.h>
#define KMS_PER_MILE 1.609

int main(void) {
 double kms, miles;
 FILE *inp, *outp;

 inp = fopen("distance.dat","r");
 outp = fopen("distance.out","w");
 fscanf(inp, "%lf", &miles);
 fprintf(outp, "The distance in miles is %.2f.\n", miles);

 kms = KMS_PER_MILES * miles;
 fprintf(outp, "That equals %.2f kilometers.\n", miles);
 fclose(inp);
 fclose(outp);
 return 0;
}
```

Common Programming Errors

- Syntax Errors
- Run-Time Errors
- Undetected Errors
- Logic Errors

Errors

- Bugs - Errors in a programs code.
- Debugging - Finding and removing errors in the program.
- When the compiler detects an error, it will output an error message.
  - Difficult to interpret
  - Often misleading
- Three types of errors
  - Syntax error
  - Run-time error
  - Undetected error
  - Logic error

Syntax Errors

A syntax error occurs when your code violates one or more grammar rules of C and is detected by the compiler at it attempts to translate your program. If a statement has a syntax error, it cannot be translated and your program will not be executed.

Common syntax errors:

- Missing semicolon
- Undeclared variable
- Last comment is not closed
- A grouping character not closed ('(', '{', '[')

Run-Time Errors

Run-time errors are detected and displayed by the computer during the execution of a program. A run-time error occurs when the program directs the computer to perform an illegal operation, such as dividing a number by zero or opening a nonexistant file. When a run-time errors occurs, the computer will stop executing your program and will display a diagnostic message that indicates the line where the error was detected.

Undetected Errors

- Many execution errors may not prevent a C program from running to completion, but they may simply lead to incorrect results. Therefore it is essential tat you predict the results your program should produce and verify that the actual output is correct.
- A very common source of incorrect results in C programs is the input of a mixture of character and numeric data. Errors can be avoided if the programmer always keeps in mind the placeholders.

Logic Errors

Logic errors occur when a program follows a faulty algorithm. Because logic errors usually do not cause run-time errors and do not display error messages, they are difficult to detect. The only sign of a logic error may be incorrect program output. You can detect logic errors by testing the program thoroughly, comparing its output to calculated results.

Pre-planning your algorithm with pseudocode or flow-charts will also help you avoid logic errors.

End of Chapter 2, are there any questions?