Lecture 2: C Basics

CMP1201: COMPUTER PROGRAMMING FUNDAMENTALS

Dr. Andrew Katumba

The Pioneers of UNIX & C

- Dennis Ritchie (UNIX, C programming language)
- Ken Thompson (UNIX)
- Alfred Aho (AWK programming language)
- Brian Kernighan (AWK programming language, many C and UNIX programming books)
- Lorinda Cherry (many document processing tools)

SECOND EDITION

THE C

PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

DON’T PANNIC!

```
#include <stdio.h>

main(t,_,a)
char *a;
{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a ):3,main ( -94, -27+t, a
)&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(_,
t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+\
,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/\
+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# ){n\
l]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

C is a *high-level programming language* developed in 1972 by Dennis Ritchie at Bell Labs.

Originally used to write the *Unix* operating system

- *systems programming language*

But nowadays used for a wide variety of applications

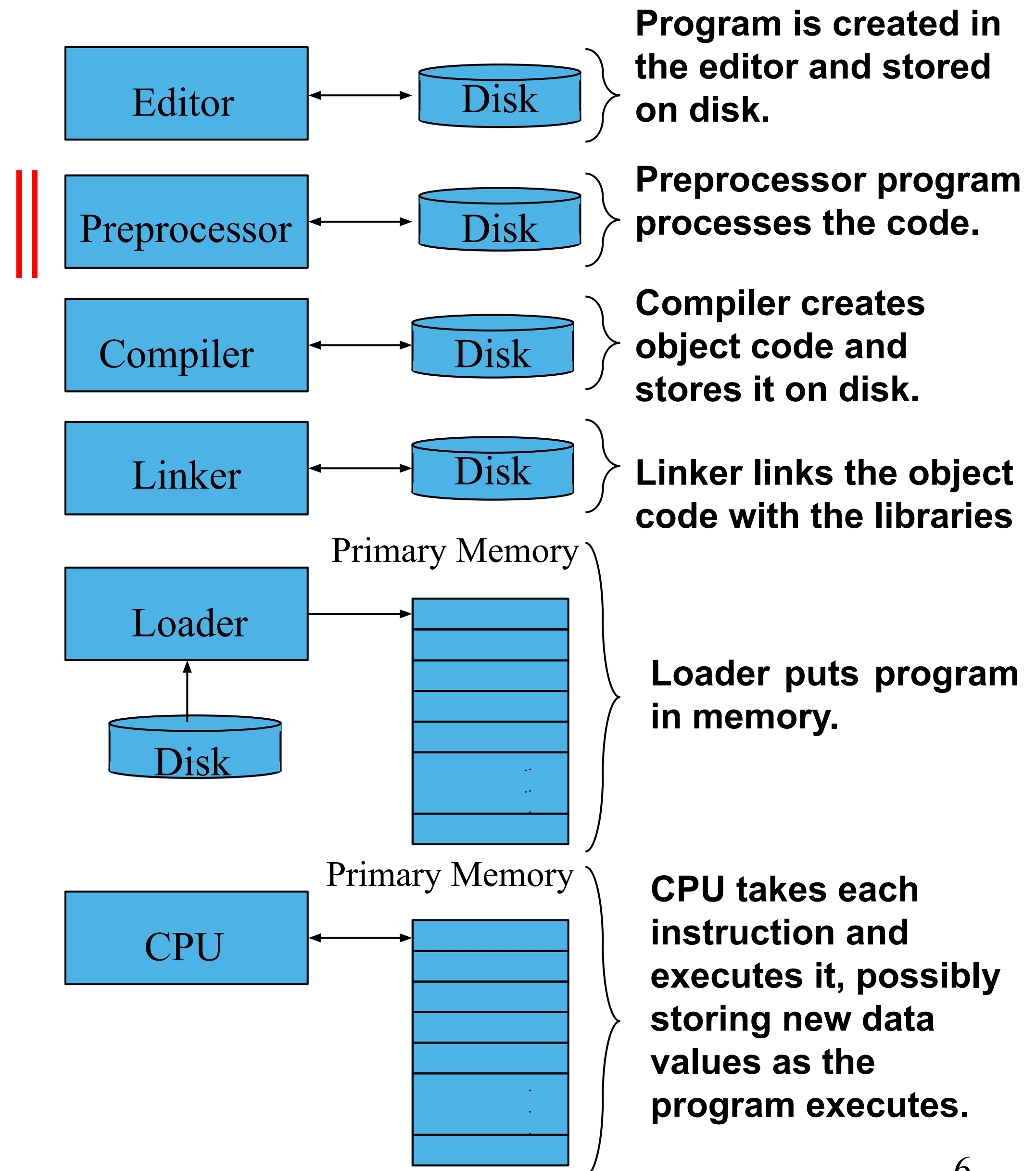
- industry-standard language

Note: High-level language => “resembles” everyday English. Instructions are written in a condensed (pseudo) form of English

Basics of a Typical C Program Development Environment

Phases of C Programs:

1. ***Edit***
2. ***Preprocess***
3. ***Compile***
4. ***Link***
5. ***Load***
6. ***Execute***



Anatomy of a simple C program

```
#include <stdio.h>
int main(void)
{
    printf("Hello World!\n");
    return (0);
}
```

C program comprises two parts:

- Preprocessor directives
- Program code

Preprocessor directives give commands to *C preprocessor* which "modifies" C program text before complilation.

`#include <stdio.h>`

- ***Preprocessor directive***
  - Tells computer to load contents of a certain file
  - Begins with a `#` symbol
- `<stdio.h>` contains standard input/output operations
  - Allows C program to perform I/O from keyboard/screen

```c
int main(void)
{
```

- C programs contain one or more *functions*, exactly one of which must be function ***main***
- Parenthesis *()* used to indicate a function
- ***int*** means that main "returns" an integer value (status code) to operating system when it terminates.
- ***void*** means that the main function receives no data from the operating system when it starts executing.
- **Left brace** *{* indicates start of a *block* (of code)
  - The “bodies” of all functions must be contained in braces - the body of a function is a block of code
  - Here it is the body of the function *main*

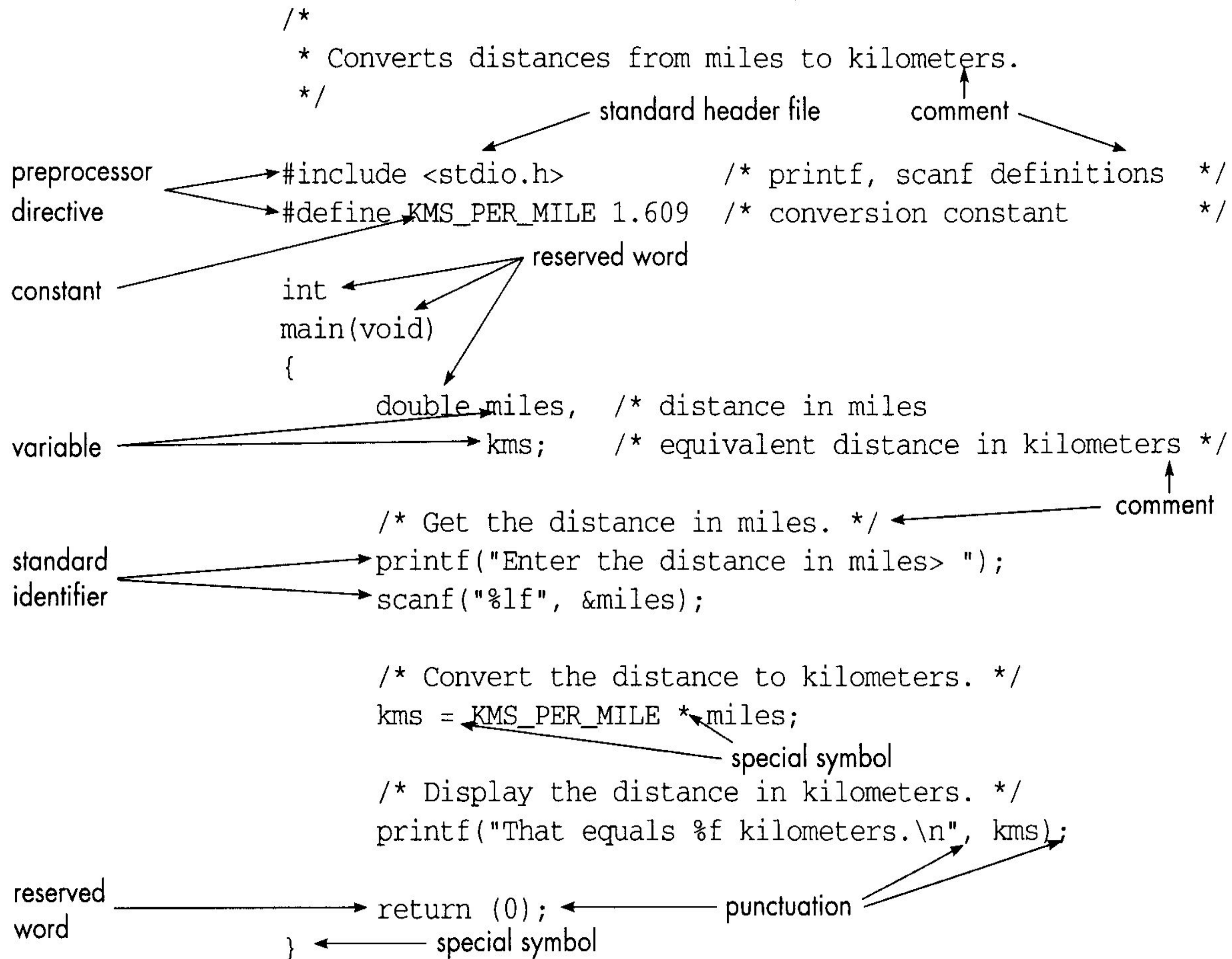
```
printf( "Hello World!\n" );
```

- Instructs computer to perform an action
  - Specifically, print the string of characters within quotes (“ ”)
  - Makes use of standard I/O function, *printf*
- Entire line called a ***statement***
  - All statements must end with a semicolon (;)
- **Escape character** (\)
  - Indicates that *printf* should do something out of the ordinary
  - `\n` is the *newline* character

```
return (0);
}
```

- A way to exit a function
- ***return (0)*** means that the program terminated normally
- **Right brace *}*** indicates end of a *block* (of code)
  - Here indicates end of *main* function has been reached

Consider a C program for the miles-to-kilometers problem.



Different kinds of statements in a C program

Preprocessor directives

```
#include <stdio.h>
```

Include the source code for library file *stdio.h*. Enables C compiler to recognize *printf* and *scanf* from this library.

```
#define KMS_PER_MILE 1.609
```

Defines a *constant*. The value 1.609 is associated with the name KMS_PER_MILE everywhere in the program.

Preprocessor substitutes value 1.609 for the name KMS_PER_MILE wherever it appears in program.

As a result, the code statement

```
kms = KMS_PER_MILE * miles;
```

would be modified by the preprocessor to become

```
kms = 1.609 * miles;
```

#include Directive for Defining Identifiers from Standard Libraries

SYNTAX: `#include` <*standard header file*>

EXAMPLES: `#include <stdio.h>`
`#include <math.h>`

INTERPRETATION: `#include` directives tell the preprocessor where to find the meanings of standard identifiers used in the program. These meanings are collected in files called *standard header files*. The header file `stdio.h` contains information about standard input and output functions such as `scanf` and `printf`. Descriptions of common mathematical functions are found in the header file `math.h`. We will investigate header files associated with other standard libraries in later chapters.

#define Directive for Creating Constant Macros

SYNTAX: `#define` *NAME value*

EXAMPLES: `#define MILES_PER_KM 0.62137`
`#define PI 3.141593`
`#define MAX_LENGTH 100`

INTERPRETATION: The C preprocessor is notified that it is to replace each use of the identifier *NAME* by *value*. C program statements cannot change the value associated with *NAME*.

Function *main*

- C programs consist of functions one of which must be *main*.

- Every C program begins executing at the function *main*.

main Function Definition

SYNTAX:

```
int
main(void)
{
        function body
}
```

EXAMPLE:

```
int
main(void)
{
   printf("Hello world\n");
   return (0);
}
```

INTERPRETATION: Program execution begins with the main function. Braces enclose the main *function body*, which contains declarations and executable statements. The line `int` indicates that the main function returns an integer value (`0`) to the operating system when it finishes normal execution. The symbols (`void`) indicate that the main function receives no data from the operating system before it begins execution.

Comments

Statements that clarify the program - **ignored by compiler** but "read" by humans.

Comments begin with `/*` and end with `*/`.

Programmers insert comments to document programs and improve their readability.

Comments **do not cause the computer to perform any action** when the program is run.

```
/* Get the distance in miles */

double miles;  /* distance in miles */
```

Note two types of *comments*:

- First form appears by itself on a program line.

- Second form appears at end of line following a C statement.

Inside the CPU and Memory

We have talked about what the CPU does:

- Executes instructions one at a time.

A series of instructions is a program

The memory holds the instructions and data for the CPU

Memory is organized as **a set of numbered cells**

- Each holds one unit of information

Everything is Binary!

All of the information in the computer is stored as 1s and 0s:

- Integers - whole numbers
- Floating-point numbers - fractional numbers
- Characters
- Strings (of characters)
- Pictures
- Programs
- …

The binary representation for, say, an integer, is totally different to that of, say, floating-point numbers and characters. Need to specify what memory cell holds.

Variables

If programmers had to do everything in binary … *they would go crazy!*

If they had to remember the memory locations of data … *they would go crazy!*

Fortunately, the programming language helps us out.

It allows us to declare

- *Variables* - which are names for places in memory

- *Data Types* - to specify the kind of data stored in the variable. Need to be specific.

Unfortunately, *programmers still go crazy …*

Declaration of variables in the program

Variables are names of memory cells that will store information - input data and results.

E.g.

```
double miles;
```

Declares a variable *miles*:

- `miles` is the *name* or *identifier* of the variable.
- `double` is the *data type* of the variable. (This particular type is for storing a real number. See later!)

Must declare the *name* and *data* type of all variables used in the program.

Other example declarations:

```
int kids, courses;
```

Declares the variables `kids` and `courses` that can store integer (whole number values).

Note may have more than one variable named in the declaration statement.

```
char initial;
```

Declares the variable `initial` that can store a single character.

Syntax Display for Declarations

SYNTAX: `int` *variable_list*;
`double` *variable_list*;
`char` *variable_list*;

EXAMPLES:

```
int count,
    large;
double x, y, z;
char first_initial;
char ans;
```

INTERPRETATION: A memory cell is allocated for each name in the *variable_list*. The type of data (`double`, `int`, `char`) to be stored in each variable is specified at the beginning of the statement. One statement may extend over multiple lines. A single data type can appear in more than one variable declaration, so the following two declaration sections are equally acceptable ways of declaring the variables `rate`, `time`, and `age`.

```
double rate, time;        |    double rate;
int age;                  |    int    age;
                          |    double time;
```

Identifiers

There are three categories of identifiers in C:

- ***reserved words***: a word that has a special meaning in the C language. E.g. *main*, *void*, *int*, … Reserved words are just that - they cannot be used for any other purpose. Also called *keywords*.

- ***standard identifier***: a word that also has a special meaning in C. E.g. *printf*, *scanf*, … However, their use can be re-defined - but it is not recommended.

- ***user-defined*** *identifier*: a word used by the programmer to name program constants and variables.

Reserved Word	Meaning
`int`	integer; indicates that the main function returns an integer value
`void`	indicates that the main function receives no data from the operating system
`double`	indicates that the memory cells store real numbers
`return`	returns control from the main function to the operating system

User-defined identifiers

Rules:

- identifier must consist only of letters, digits and underscores. Must not begin with a digit.

- limit length of identifier to maximum of 31 symbols.

- do not use a C reserved word as an identifier.

The C compiler is ***case-sensitive*,** i.e. it differentiates between uppercase and lowercase letters.

Thus, the identifiers

rate
Rate
RATE

are considered to be <u>different</u> identifiers.

Be careful! But adopt a **consistent style**.

<u>Note</u>: that all C reserved words and names of standard library functions are in lowercase only.

One common practice is

- to use uppercase for CONSTANTS

- to use lowercase for all other identifiers.

Program Style Hints

A program that "looks good" is easier to read, understand and debug.

1. Always choose meaningful identifier names.

E.g. identifier ***miles*** is better than **m**

2. If identifier consists of two or more words, separate words by an underscore, _.
E.g. ***KMS_PER_MILE*** is better than ***KMSPERMILE***

3. Avoid excessively long names - avoid typing errors.
E.g. ***KMS_PER_MILE*** is better than
KILOMETERS_PER_MILE

4. Do not use names that are similar to each - the compiler does not know that you may have mistyped one name for another.

In particular, **avoid names that differ only in the use of lower case and uppercase, or differ by the presence or absence of an underscore.**

E.g. *Large* and *large*, *x_coord* and *xcoord*

Summary

- All variables in a C program must be declared before they can be used in the program.

- Every variable stored in the computer's memory has a name, a value and a type.

- A variable name in C is any valid identifier. An identifier is a series of characters consisting of letters, digits and underscores (_). Identifiers cannot start with a digit. Identifiers can be any length; however, only the first 31 characters are significant.

- C is case sensitive

Variable Declarations and Data Types

A *data type* specifies **a set of values and a set of operations** on those values.

The (*data*) *type* of every program variable must be specified in the declaration statement.

Types tell the CPU how to interpret the 0s and 1s in the memory cell where any given variable is stored.

Example: the integer 1086324736 is stored as

01000000110000000000000000000000

If we read it as a floating point number we get 6.0!

Types help the computer and the programmer **keep things straight**

Basic Data Types

`int` - integer numbers - whole numbers in the range:
-32767 -- 32767

`char` - <u>single</u> character - character must be enclosed by single quotes <u>within</u> the program.

`double` - numbers with fractional parts - real numbers.
In program, write with decimal point, e.g. 123.0 or scientific notation, e.g. 1.23e2 or 1.23E2

We will see more types later.

Declaring Variables

`int months;`
Can hold integer data, like 6, 12, -17

`double pi;`
Can hold floating-point representations of numbers, like 3.14159, 2.71828

`char initial;`
Can hold a single character, like ‘i’, ‘K’, ‘@’

Note that a declaration is terminated with a semi-colon, ;.

Executable Statements

Executable statements follow the data declarations in a function.

They are the C program statements used to write or code the algorithm.

C compiler translates these statements to machine code.

Then the computer executes these when we run the program.

Assignment statements

Most common *executable statement* is the *assignment statement*.

Used to assign a value to a variable.

Computer first evaluates expression to determine its value

General format:

variable = expression;

The value to be assigned is written on the right hand of the assignment operator =.

The variable getting the value is on the left hand side.

Note that the statement is terminated by a semi-colon, ;

Expression may be a single variable or constant, or some combination, e.g. an arithmetic expression

E.g.

```
kms = 1.609 * miles;
```

Variable ***kms*** is assigned the value of *1.609*miles*
(* means multiply in C). The **current value of *kms* is overwritten by the new value.**

Note: the variable *miles* must be given a value before the assignment statement. This is true for any assignment statement involving variables.

The = sign is rather odd.

In maths, the = means two things are equal

The developers of C were cruel, wicked fiends, who just wanted to confuse poor students.

In C, the = sign means “**takes the value of”.**

It is the assignment operator:

`x = y;` means “*x* takes the value of *y*”

Or, “*x* is assigned the value of *y*”

```
sum = x + y;
```

sum takes the value of *x* + *y*.

```
kids = kids + 1;
```

kids gets the value of the *"current value of kids"* + *1*. If *kids* was 5, its new value will be 6 (strange, but true!)

```
hypotenuse = sqrt(side1 * side1 +
                  side2 * side2);
```

(*sqrt* is a C function for calculating square roots.)

Assignment statements can be quite complex!

Notice that

```
x + y = sum;
```

is invalid. Why?

Assignment Statement

FORM: *variable = expression;*

EXAMPLE: `x = y + z + 2.0;`

Interpretation: The *variable* before the assignment operator is assigned the value of the *expression* after it. The previous value of *variable* is destroyed. The *expression* can be a variable, a constant, or a combination of these connected by appropriate operators (for example, +, -, /, and *).

Initializing Variables

Initialization means giving something a value for the first time.

Any way which changes the value of a variable is a potential way of initializing it:

– assign an initial value in a **declaration**:
E.g.

```
int i = 7;
```

– assign a value by an **assignment statement**:
E.g.

```
count = 0;
```

- assign a value by reading
E.g.

```
scanf("%lf", &miles);
```

Syntax Display for return Statement

SYNTAX: `return` *expression*`;`

EXAMPLE: `return (0);`

INTERPRETATION: The `return` statement transfers control from a function back to the activator of the function. For function `main`, control is transferred back to the operating system. The value of *expression* is returned as the result of the function execution.

Summary

An assignment statement puts a value into a variable

The assignment may specify a simple value, or an expression

Remember = means “takes the value of”

The computer always evaluates what is on the right of the = and stores it in the variable on the left

Review Quiz

Find the assignments, declarations, and initializations:

```
int main (void) {
    double income; /* ??? */
    income = 35500.00; /* ??? */
    printf(“Old income is %f”, income);
      income = 39000.00; /* ??? */
    printf(“After raise: %f”, income);
}
```

Arithmetic expressions

To solve most programming problems, need to write arithmetic expressions that manipulate numeric type data.

Expressions may be combinations of numeric constants (E.g. 105), program constants (E.g. *KMS_PER_MILE*) and/or program variables (E.g. *miles*), together with arithmetic operators

Arithmetic operators are

+,   -,   *,   /

Arithmetic operators

+	Addition	5+9=14
-	Subtraction	5.2 - 3.1 = 2.1
*	Multiplication	5.2 * 2 = 10.4
/	Division	5.0 / 2.0 = 2.5
		5 / 2 = 2
%	Remainder	7 % 6 = 1

Relationship between the operators / and % for an integer dividend of *m* and integer divisor of *n*:

m equals (m/n)*n + (m%n)

Example:

11 equals (11/2)*2 + (11%2)

equals 5 *2 + 1

Data type of an Expression

Data type of each variable must be specified in its declaration.
Data type of an expression depends on the type of its operands.

`var1` ***arithmetic_operator*** **`var2`**

is of type `int` if both `var1` and `var2` are of type `int`; otherwise, it is of type `double`

Mixed-Type Assignment Statement

An expression that has operands of both type `int` and `double` is a **mixed-type expression**

a = b + c * (b – d)

Double = int and /or double

int = int and/or double but only the integral part is saved into a

The expression is first evaluated; then the results is assigned to the variable listed to the left of the assignment operator (=).

Rules for evaluating multiple operators

- **Parenthesis rule**
- **Operator precedence rule**
    - Unary ++, - - , +, -
    - Binary +, -, *, /, %
- **Associativity rule**
    - Unary operators at the same precedence level – right to left evaluation (right associativity)
    - Binary operators at the same precedence level – left to right evaluation (left associativity)
    - Z-(a+b+b/2)+w*-y
    - In complicated expressions use extra parentheses

Writing mathematical formulas

- Always specify multiplication explicitly by using the operator * where needed
- Use parenthesis when required to control the order of operator evaluation
- Two arithmetic operators can be written in succession if the second is a unary operator

I/O Operations and Library Functions

Need to get data into program from keyboard and display information (including results) on screen.

The library functions *scanf* and *printf* do this for us.

- ***printf* is used for output to the screen**

- ***scanf* is used for input from the keyboard**

E.g.

```
printf("Enter distance in miles> ");
```

- write some text

```
scanf("%lf", &miles);
```

- read a value for variable *miles*

```
printf("That equals %f kms.\n", kms);
```

- write some text and value of *miles*

Each call to *printf* and *scanf* begins with a *format string* in double quotes "…".
I.e.

```
printf(format string);
printf(format string, print list);
scanf(format string, input list);
```

The *format string* specifies the format for the input or output.

After the format string comes

- an *input list* (for *scanf*)
- a *print list* (for *printf*). Optional.

The variables named in the *input list* must be preceded by an '&' (e.g. *&miles*).

The *format strings* contain multiple *placeholders*, one for each variable in the list.

Placeholders indicate the data type and position of a variable in a format string.

Use:

`%lf` in *scanf* for type *double* value
`%f` in *printf* for type *double* value
`%d` in *printf* and *scanf* for type *int* value
`%c` in *printf* and *scanf* for type *char* value

E.g.


```
printf("That equals %f kms.\n", kms);
```


When this statement is executed, the current value of *kms* is printed at the place indicated by the placeholder in the format string.

kms is a double data type, so placeholder is `%f`

If kms has the value 25, say, then the what is printed is

That equals 25 kms

Syntax Display for printf Function Call

SYNTAX: `printf(`*format string*`,` *print list*`);`
`printf(`*format string*`);`

EXAMPLES:

```
printf("I am %d years old, and my gpa is %f\n",
       age, gpa);
printf("Enter the object mass in grams> ");
```

INTERPRETATION: The `printf` function displays the value of its *format string* after substituting in left-to-right order the values of the expressions in the *print list* for their placeholders in the *format string* and after replacing escape sequences such as `\n` by their meanings.

Syntax Display for scanf Function Call

SYNTAX: `scanf` (*format string*, *input list*) ;

EXAMPLE: `scanf("%c%d", &first_initial, &age);`

INTERPRETATION: The `scanf` function copies into memory data typed at the keyboard by the program user during program execution. The *format string* is a quoted string of placeholders, one placeholder for each variable in the *input list*. Each `int`, `double`, or `char` variable in the *input list* is preceded by an ampersand (`&`). Commas are used to separate variable names. The order of the placeholders must correspond to the order of the variables in the *input list*.

You must enter data in the same order as the variables in the *input list*. You should insert one or more blank characters or carriage returns between numeric items. If you plan to insert blanks or carriage returns between character data, you must include a blank in the format string before the `%c` placeholder.

Examples

1. Display a *prompt(ing) message* that tells program user what to enter.
E.g.

```
printf("Enter distance in miles> ");
scanf("%lf", &miles);
```

What the user sees is

Enter distance in miles>

The program user then can type in the data value requested which is processed by the *scanf* function.

2. Displaying blank lines

The two-character sequence, `\n`, produces "*newline*" on output - equivalent to typing *Enter*.

E.g.

```
printf(“\n”);
```

E.g.

```
printf("That equals %f kms.\n", kms);
```

Prints a newline after the message

3. Printing two or more variables

printf might have more than one expression in its list.

E.g.

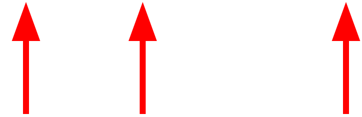
```
int multiplier = 2;
double number, double_number;
number = 3.14;
double_number = multiplier * number;
printf(" %d times %f is %f. \n",
        multiplier , number , double_number );
```

Output: *2 times 3.14000 is 6.28000*

Basic rule: placeholders in format string must match expressions in output list in number, order, and type.

4. Inputting a value into a variable

```
scanf("%lf", &miles);
```



Placeholder Note! Variable

The real number value typed at the keyboard is assigned to the variable, *miles*.

Do NOT forget the &

5. Reading two or more data items with *scanf*

E.g.

```
printf("Enter hours and rate>");
scanf("%lf%lf", &hours, &rate);
```

First number stored in *hours*, second in *rate*. Should be at least one space between numbers. Don't forget &s!

E.g

```
char first, second;

printf("Enter your two initials>");
scanf("%c%c", &first, &second);
```

Basic rule: placeholders in the format must match variables in the input list.

Must match one-for-one in *number*, *order*, and *type*.

```
int studentID ;
double grade ;
scanf ("%d %lf", &studentID , &grade ) ;
```

Be aware that advanced formatting options exist and can be looked up when needed.

Formatting Numbers in Program Output

Field width – the number of columns used to display a value

Formatting values of type *int*

Placeholder format is: %nd, right justified

E.g.

printf (“Results: %3d meters = %4d ft. %2d in.\n”,meters, feet, inches);

Results:

■ ■ 21 ■ meters ■ = ■ ■ ■ 68 ■ ft. ■ 11 ■ in.

(■ - represents blank character)

Formatting Values of type *double*

Placeholder format is: %n.m

n – total field width (incl. decimal point and minus sign)
m – desired number of decimal places

See Table 2.11 and 2.12, Section 2.6

Interactive mode, batch mode and data files

- *Interactive mode*: The user interacts with the program – types in data while it is running.

- *Batch mode*: the program scans the data it needs from a previously prepared file
  - input redirection: program <input_file
  - output redirection: program >output_file
  - program <input_file   >output_file
  - input and output controlled by the program

Batch Version of Miles-to-Kilometers Conversion Program (page 77)

```
/* Converts distances from miles to kilometers. */
#include <stdio.h>      /* printf, scanf definitions */
#define KMS_PER_MILE  1.609 /* conversion constant */
int main(void)
{
	double miles, /* distance in miles */
		kms; /* equivalent distance in kilometers */
	/* Get and echo the distance in miles. */
	scanf("%lf", &miles);
	printf("The distance in miles is %.2f.\n", miles);
	/* Convert the distance to kilometers. */
	kms = KMS_PER_MILE * miles;
	/* Display the distance in kilometers. */
	printf("That equals %.2f kilometers.\n", kms);
	return (0);
}
```

The command line for running this program (`metric`) is:

```
metric < mydata
```

The file `mydata` contains input data (112.00)

```
The distance in miles is 112.00.
That equals 180.21 kilometers.
```

Common Programming errors

- **Syntax Errors**
  - **Caused by violation of grammar rules of C**
- **Run-Time Errors**
  - **Displayed during execution**
    - **Division by 0**
- **Undetected errors**
  - **Usually lead to incorrect results**
- **Logical errors**
  - **Caused by following an incorrect algorithm**
  - **Difficult to detect**

Compiler Listing of a Program with Syntax Errors

```
221 /* Converts distances from miles to kilometers. */
222
223 #include <stdio.h>
266 #define KMS_PER_MILE  1.609
267
268 int
269 main(void)
270 {
271      double kms
272
273      /* Get the distance in miles. */
274      printf ("Enter the distance in miles> ");
***** Semicolon added at the end of the previous
      source line

275      scanf ("%lf", &miles);
***** Identifier "miles" is not declared within this
      scope
***** Invalid operand of address-of operator
```

```
276
277      /* Convert the distance to kilometers. */
278      kms = KMS_PER_MILE * miles;
***** Identifier "miles" is not declared within this scope

279
280      /* Display the distance in kilometers. * /
281      printf ("That equals %f kilometers.\n", kms);
282
283      return (0);
284 }
***** Unexpected end-of-file encountered in a comment
***** "}" inserted before end-of-file
```

A Program with a Run-time Error

```
111 #include <stdio.h>
262
263 int  main (void)
264 {
266       int    first, second;
267       double temp, ans;
268
269       printf ("Enter two integers> ");
270       scanf ("%d%d", &first, &second);
271       temp = second / first; /* temp = 3/14 = 0 */
272       ans = first / temp;
273       printf ("The result is %.3f\n", ans);
274
275       return (0);
276 }
```

```
Enter two integers> 14 3
Arithmetic fault, divide by zero at line 272 of routine main
```

A Program That Produces Incorrect Results Due to & Omission

```
#include <stdio.h>
int main(void)
{
    int    first, second, sum;

    printf ("Enter two integers> ");
    scanf ("%d%d", first, second);     /* ERROR! Should be  &first, &second */

    sum = first + second;

    printf ("%d + %d = %d\n", first, second, sum);

    return (0);
}
```

```
Enter two integers> 14  3
5971289 + 5971297 = 11942586
```

Summary

- Every C program has preprocessor directives and a *main* function
- Variable names must begin with a letter and consist of letters, digits and underscore symbol. A reserved word cannot be used as an identifier
- The data type of each variable must be declared. Three standard data types are *int*, *double* and *char*.
- Assignment statements are used to perform computations and store results in memory.
- Function calls are used to get data (functions *scanf* and *fscanf*) and to display values stored in memory (functions *printf* and *fprintf*)

Reading for Lecture:

PSC: Chapter 2

CPL: Chapter 2

Write code