

Overview of C

SDC OSW CSE 3541

**Department of Computer Science & Engineering
ITER, Siksha 'O' Anusandhan Deemed To Be University
Jagamohan Nagar, Jagamara, Bhubaneswar, Odisha - 751030**

Book(s)

Text Book(s)



Jeri R Hanly, & Elliot B. Koffman

Problem Solving & Program Design in C

Seventh Edition

Pearson Education



Kay A. Robbins, & Steve Robbins

UnixTM Systems Programming **Communications, concurrency, and Treads**

Pearson Education

Reference Book(s)



Brain W. Kernighan, & Rob Pike

The Unix Programming Environment

PHI

Talk Flow

- 1 Introduction
- 2 C Language Elements
- 3 Variable Declarations and Data Types
- 4 Executable Statements
- 5 Arithmetic Expression
- 6 Operator and its precedence Chart
- 7 Some Necessary information of C

Introduction

- 1 This chapter introduces C (a high-level) programming language developed in 1972 by Dennis Ritchie at AT&T Bell Laboratories. Because C was designed as a language in which to write the UNIX® operating system, it was originally used primarily for systems programming. Over the years, however, the power and flexibility of C, together with the availability of high-quality C compilers for computers of all sizes, have made it a popular language in industry for a wide variety of applications.
- 2 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.

C Language Elements

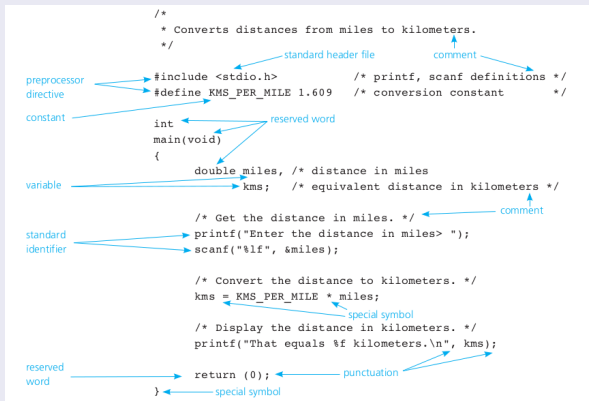


Figure 1: C Language Elements in Miles-to-Kilometers Conversion Program

Elements in C Program

- ① '#' Pre-processor Directives
 - #include
 - #define
- ② Main Function
- ③ Reserved Words
- ④ Identifiers
 - Standard Identifiers
 - User Defined Identifiers
- ⑤ Comment lines
- ⑥ Indentation

‘#’ Pre-processor Directive

```
#include <stdio.h>           /*printf, scanf definitions */  
#define KMS_PER_MILE 1.609    /*conversion Constant */
```

- The #include directive gives a program access to a library. This directive causes the pre-processor to insert definitions from a standard header file into a program before compilation. The directive notifies the pre-processor that some names used in the program (such as scanf and printf) are found in the standard header file <stdio.h>.
- The second directive associates the constant macro KMS_PER_MILE with the meaning 1.609 . This directive instructs the pre-processor to replace each occurrence of KMS_PER_MILE in the text of the C program by 1.609 before compilation begins.

‘#’ Pre-processor Directive (Contd...)

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

would read

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

- Only data values that never change (or change very rarely) should be given names using a `#define`, because an executing C program cannot change the value of a name defined as a constant macro.
- Using the constant macro `KMS_PER_MILE` in the text of a program for the value 1.609 makes it easier to understand and maintain the program.

Note : The text on the right of each pre-processor directive, starting with `/*` and ending with `*/`, is a comment.

Syntax Displays for #include Pre-processor Directives

SYNTAX: `#include <standard header file>`

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

Interpretation

`#include` directives tell the pre-processor 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`.

Syntax Displays for #define Pre-processor Directives

SYNTAX: `#define NAME value`

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

Interpretation

The C pre-processor 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

```
int main(void)
```

The above line marks the beginning of the main function where program execution begins. Every C program has a main function. The remaining lines of the program form the body (code block) of the function which is enclosed in braces , .

A function body has two parts: declarations and executable statements. The **declarations** tell the compiler what memory cells are needed in the function (for example, miles and kms. To create this part of the function, the programmer uses the problem data requirements identified during problem analysis. The textbfexecutable statements (derived from the algorithm) are translated into machine language and later executed.

Function Main (Cont...)

main Function syntax

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.

Reserved Words

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

All the reserved words (in total 32) appear in lowercase; they have special meaning in C and cannot be used for other purposes.

Reserved Words (Contd...)

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

Standard Identifiers

Like reserved words, standard identifiers have special meaning in C. In Fig. 1, the standard identifiers `printf` and `scanf` are names of operations defined in the standard input/output library. Unlike reserved words, standard identifiers can be redefined and used by the programmer for other purposes however, we don't recommend this practice. If you redefine a standard identifier, C will no longer be able to use it for its original purpose.

User Defined Identifiers

We choose our own identifiers (called user-defined identifiers) to name memory cells that will hold data and program results and to name operations that we define. The first user-defined identifier in Fig. 1, `KMS_PER_MILE`, is the name of a constant macro. You have some freedom in selecting identifiers.

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

NOte: In general, there is no limit on length of identifier, but some ANSI C compilers consider only first 31 characters as significant.

Valid or Invalid?

1Letter, good, double, int, auto, TWO*FOUR, joe's, sqrt, __var,
hello_good_morning

Example at a glance

Reserved Words	Standard Identifiers	User-Defined Identifiers
<code>int, void,</code> <code>double,</code> <code>return</code>	<code>printf, scanf</code>	<code>KMS_PER_MILE, main,</code> <code>miles, kms</code>

Figure 2: Reserved Words and Identifiers in Fig. 1

Invalid Identifier	Reason Invalid
<code>1Letter</code>	begins with a letter
<code>double</code>	reserved word
<code>int</code>	reserved word
<code>TWO*FOUR</code>	character <code>*</code> not allowed
<code>joe's</code>	character <code>'</code> not allowed

Figure 3: Invalid Identifiers

User Defined Identifiers (Contd...)

Some other rules for user defined identifier

- Case Sensitive
- Choosing Identifier Names

Self Check

1. Which of the following identifiers are (a) C reserved words, (b) standard identifiers, (c) conventionally used as constant macro names, (d) other valid identifiers, and (e) invalid identifiers?

```
void      MAX_ENTRIES    double   time      G      Sue's  
return    printf         xyz123    part#2    "char"   #insert  
this_is_a_long_one
```

2. Why should `E (2.7182818)` be defined as a constant macro?
3. What part of a C implementation changes the text of a C program just before it is compiled? Name two directives that give instructions about these changes.
4. Why shouldn't you use a standard identifier as the name of a memory cell in a program? Can you use a reserved word instead?

Comment lines

Programmers can make a program easier to understand by using comments to describe the purpose of the program, the use of identifiers, and the purpose of each program step. Comments are part of the program documentation because they help others read and understand the program. The compiler, however, ignores comments and they are not translated into machine language.

Types of comment line

- Single line :

```
double miles; // input - distance in miles.
```

- Multiple Lines

```
double kms; /* output - distance in kilometers  
any other lines to comment goes here  
and here  
ad here too */
```

Indentation

The consistent and careful use of blank spaces can improve the style of a program. A blank space is required between consecutive words in a program line. The compiler 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 `=`. You should indent the body of the main function and insert blank lines between sections of the program.

Although stylistic issues have no effect whatever on the meaning of the program as far as the computer is concerned, they can make it easier for people to read and understand the program. Take care, however, not to insert blank spaces where they do not belong. For example, there cannot be a space between the characters that surround a comment (`/*` and `*/`). Also, you cannot write the identifier `MAX_ITEMS` as `MAX ITEMS`.

Example of Indentation

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

int main(){
int a[10],i;

printf("Enter Elements : \n");
for(i=0; i<10; i++){
printf("Enter %d number : ",i+1);
scanf("%d",&a[i]);
}

int sum = 0;
for(i=0; i<10; i++){
sum+=a[i];

printf("Sum of all values %d\n",sum);

return 0;
}
```

Without
Indentation



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

int main(){
    int a[10],i;

    printf("Enter Elements : \n");
    for(i=0; i<10; i++){
        printf("Enter %d number : ",i+1);
        scanf("%d",&a[i]);
    }

    int sum = 0;
    for(i=0; i<10; i++){
        sum+=a[i];

    printf("Sum of all values %d\n",sum);

    return 0;
}
```

With
Indentation



Variable Declarations and Data Types

```
double miles; /* input - distance in miles.*/  
double kms; /* output - distance in kilometers */
```

The memory cells used for storing a program's input data and its computational results are called variables because the values stored in variables can change (and usually do) as the program executes. The variable declarations in a C program communicate to the C compiler the names of all variables used in a program. They also tell the compiler what kind of information will be stored in each variable and how that information will be represented in memory.

Variable Declarations and Data Types (Contd...)

```
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`.

Data Type

A data type is a set of values and a set of operations on those values. Knowledge of the data type of an item (a variable or value) enables the C compiler to correctly specify operations on that item. A standard data type in C is a data type that is predefined, such as `char`, `double`, and `int`. We use the standard data types `double` and `int` as abstractions for the real numbers and integers (in the mathematical sense).

- **char**: The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
- **int**: As the name suggests, an `int` variable is used to store an integer.
- **float**: It is used to store decimal numbers (numbers with floating point value) with single precision.
- **double**: It is used to store decimal numbers (numbers with floating point value) with double precision.

Valid double Constants	Invalid double Constants
3.14159	150 (no decimal point)
0.005	.12345e (missing exponent)
12345.0	15e-0.3 (0.3 is invalid exponent)
15.0e-04 (value is 0.0015)	
2.345e2 (value is 234.5)	12.5e.3 (.3 is invalid exponent)
1.15e-3 (value is 0.00115)	34,500.99 (comma is not allowed)
12e+5 (value is 1200000.0)	

Figure 4: Type double Constants (real numbers)

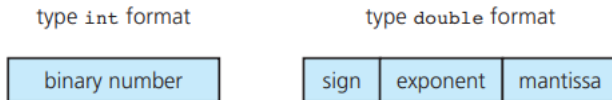


Figure 5: Internal Formats of Type `int` and Type `double`

Data Type	Range	Bytes	Format
signed char	-128 to +127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-2147483648 to +2147483647	4	%d
unsigned int	0 to 4294967295	4	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf
Note: The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 32-bit compiler.			

Figure 6: Range, Bytes Format specifier for each data type in c)

Character Data Type ASCII Code

Character	ASCII Code
' '	32
'*'	42
'A'	65
'B'	66
'Z'	90
'a'	97
'b'	98
'z'	122
'0'	48
'9'	57

Figure 7: ASCII Codes for Characters

Self Check

1. a. Write the following numbers in normal decimal notation:

103e-4 1.2345e+6 123.45e+3

- b. Write the following numbers in C scientific notation:

1300 123.45 0.00426

2. Indicate which of the following are valid type `int`, `double`, or `char` constants in C and which are not. Identify the data type of each valid constant.

'PQR' 15E-2 35 'h' -37.491 .912 4,719 'true' "T"
& 4.5e3 '\$'

3. What would be the best variable type for the area of a circle in square inches?
Which type for the number of cars passing through an intersection in an hour?
The first letter of your last name?

Solve this Program

1. Write the `#define` preprocessor directive and declarations for a program that has a constant macro for `PI` (3.14159) and variables `radius`, `area`, and `circumf` declared as `double`, variable `num_circ` as an `int`, and variable `circ_name` as a `char`.

Types of executable statements

- ① Assignment Statements
- ② Input/Output Operations and other Functions
 - printf()
 - scanf()
 - sumOfTwoNumber(int, int)
 - bubbleSort(int [],int)
 - findMax(int [],int)
 - ... any 'system defined' or 'user defined' functions
- ③ The return Statement

Arithmetic Statements

An assignment statement stores a value or a computational result in a variable, and is used to perform most arithmetic operations in a program. The assignment statement

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

assigns a value to the variable kms.

Syntax

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 *).

Arithmetic Statements Contd...

Before assignment

KMS_PER_MILE

miles

kms

1.609

10.00

?

*

After assignment

KMS_PER_MILE

miles

kms

1.609

10.00

16.090

16.090

The printf function

A diagram showing the components of the `printf` function call. The code `printf("That equals %f kilometers.\n", kms);` is shown. Above the code, "function name" has an arrow pointing to `printf`. "function arguments" has an arrow pointing to the opening parenthesis and a bracket above the closing parenthesis. Below the code, "format string" has an arrow pointing to the opening quote of the string, and "print list" has an arrow pointing to the variable `kms`.

function name

function arguments

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

format string

print list

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

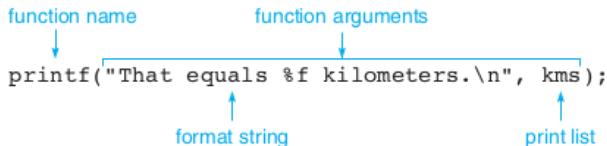
The `printf` function displays the value of its format string after substituting in left-to-right order the values of the expressions in the print list for their placeholders in the format string and after replacing escape sequences such as `\n` by their meanings.

The scanf function

function name function arguments

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

format string print list

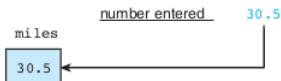
A diagram showing the components of the printf function call. The text 'printf("That equals %f kilometers.\n", kms);' is shown. Above 'printf' is the label 'function name' with a blue arrow pointing down to it. Above the opening parenthesis is the label 'function arguments' with a blue arrow pointing down to it. Below the opening parenthesis is the label 'format string' with a blue arrow pointing up to it. Below the closing parenthesis is the label 'print list' with a blue arrow pointing up to it. A blue bracket is drawn above the arguments, spanning from the opening parenthesis to the closing parenthesis.

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

number entered 30.5

miles

30.5

A diagram showing the execution of 'scanf("%lf", &miles);'. The text 'number entered' is underlined in blue, followed by '30.5' in blue. A blue arrow points from '30.5' to a light blue box labeled 'miles' which contains the value '30.5'.

Scan only 1 integer data

```
scanf("%c%c%c", &letter_1, &letter_2, &letter_3);
```

letters entered Bob

letter_1

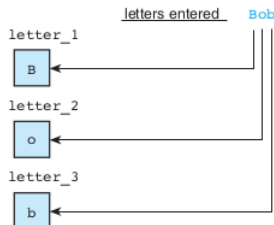
B

letter_2

o

letter_3

b

A diagram showing the execution of 'scanf("%c%c%c", &letter_1, &letter_2, &letter_3);'. The text 'letters entered' is underlined in blue, followed by 'Bob' in blue. Three blue arrows point from 'B', 'o', and 'b' to three light blue boxes labeled 'letter_1', 'letter_2', and 'letter_3' respectively. The boxes contain the characters 'B', 'o', and 'b'.

scan 3 character data

The scanf function Contd...

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 place-holders, 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.

Arithmetic Expression

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	5.0 / 2.0 is 2.5 5 / 2 is 2
%	remainder	5 % 2 is 1

$$3 / 15 = 0$$

$$18 / 3 = 6$$

$$15 / 3 = 5$$

$$16 / -3 \text{ varies}$$

$$16 / 3 = 5$$

$$0 / 4 = 0$$

$$17 / 3 = 5$$

$$4 / 0 \text{ is undefined}$$

$$3 \% 5 = 3$$

$$5 \% 3 = 2$$

$$4 \% 5 = 4$$

$$5 \% 4 = 1$$

$$5 \% 5 = 0$$

$$15 \% 5 = 0$$

$$6 \% 5 = 1$$

$$15 \% 6 = 3$$

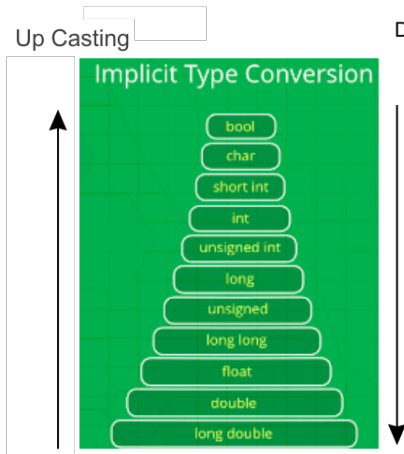
$$7 \% 5 = 2$$

$$15 \% -7 \text{ varies}$$

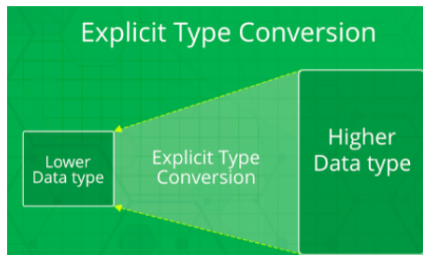
$$8 \% 5 = 3$$

$$15 \% 0 \text{ is undefined}$$

Type casting



Down Casting



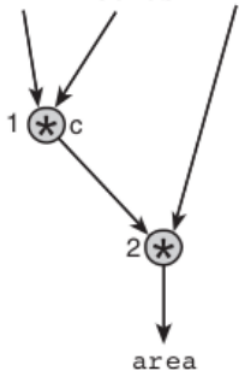
The syntax in C:

```
(type) expression
```

Mathematical Expression in C programming

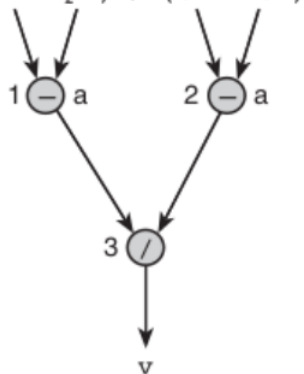
$$a = \pi r^2$$

area = PI * radius * radius



$$v = \frac{p_2 - p_1}{t_2 - t_1}$$

v = (p2 - p1) / (t2 - t1)



Mathematical Expression in C programming (Contd...)

Mathematical Formula	C Expression
1. $b^2 - 4ac$	<code>b * b - 4 * a * c</code>
2. $a + b - c$	<code>a + b - c</code>
3. $\frac{a + b}{c + d}$	<code>(a + b) / (c + d)</code>
4. $\frac{1}{1 + x^2}$	<code>1 / (1 + x * x)</code>
5. $a \times -(b + c)$	<code>a * -(b + c)</code>

Operator and its precedence Chart

- Unary

+, -

- Binary

+, -, *, /, %, ^

- Ternary

?:

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){list}	Compound literal	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement	
3	*, /, %	Multiplication, division, and remainder	Left-to-right
4	+	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and <= respectively	
	> >=	For relational operators > and >= respectively	
7	== !=	For relational = and != respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13	?:	Ternary conditional	Right-to-left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
15	&= ^= =	Assignment by bitwise AND, XOR, and OR	
	,	Comma	Left-to-right

Which type of language is 'C'?

- Machine language
- Assembly language
- High level language

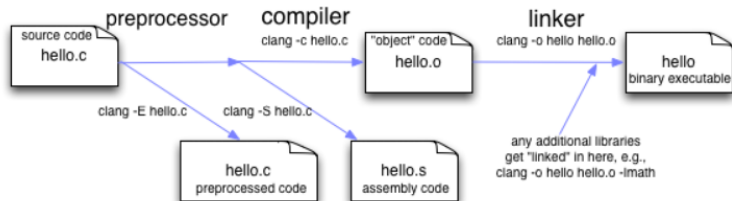
Structure of a C-Program (hello.c)

```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

Compiling a C-Program



Self Check

1. Change the following comments so they are syntactically correct.

```
/* This is a comment? *\n/* This one /* seems like a comment */ doesn't it */
```

2. Correct the syntax errors in the following program, and rewrite the program so that it follows our style conventions. What does each statement of your corrected program do? What output does it display?

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

You can do

1. Write a program that stores the values 'x' and 76.1 in separate memory cells. Your program should get the values as data items and display them again for the user when done.

Thank you

ANY QUESTIONS?