C-Programming

Computer Science

&

Information Technology (CS)

20 Rank under AIR 100

Postal Correspondence

- ✓ Examination Oriented Theory, Practice Set
- ✓ Key concepts, Analysis & Summary

ENGINEERS INSTITUTE OF INDIA
India's Best Institute for IES, GATE & PSUs
A Team of IES & GATE Toppers

CONTENT

CHAPTER-1

INTRODUCTION TO C-PROGRAMMING

Introduction to C- Programming

→ C is structured programming language used for system programming as well as application programming.

→ C follow procedure oriented modular approach. It means problems are divided into module and each module is individual entity.

Sample program in C

```
#include < stdio.h>
    main ( )              /*..... Line2 ..... */
    {                     /*..... Line3 ..... */
/*.... display .... */    /*..... Line4 ..... */
   printf ("hello");      /*..... Line5 ..... */
/* .... display ends .... /*..... Line6 ..... */
    }                     /*..... Line7 ..... */
```

Figure : A program to print "hello"

In the above figure first line informs the system that the standard input and output library is included in program.

When the above program executed, it will print $\boxed{\text{hello}}$ on to the display screen.

(1) In the above figure, first line is used for including input/output library function. < stdio.h> is standard input output library. However, this is not necessary for the functions printf and scanf.

→ In the above figure, second line informs the system that the name of function is main ( ).

Note: In C programming, the execution starts at main ( ) function. Every C program should have exactly one main ( ) function.

→ Here, main ( ) function have empty pair of parenthesis. It means in this program, it will not take any argument.

→ At the end of first line we have /* ... line 1 ... */. This is comment. These are used in a program to enhance its readability and understanding.

Note: The lines beginning with /* and ending with */ knows as comment lines. Comments lines are not executable statement and therefore anything between /* and */ is ignored by the compiler.

* ***We cannot have comments inside comment. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The below comment line time is not valid.***

/ * ...... / * ...... * / ...... * /

Therefore, result and error.

∗ Comment do not affect the execution speed of programs.

→ The opening brace “{” in the second line marks the beginning of the main ( ) function.

→ At third line we have comment that tells to reader of program that next code is used for displaying of data.

→ At fourth line, we have printf ( ) function that is used to display the string or value of variable. Here, It is used to display the $\boxed{\text{hello}}$ string.

→ At fifth line, we have closing brace of main function.

Note: C does make a distinction between uppercase and lowercase letter.

For example: printf and PRINTF are not the same.

Main ( ) function

– Every program in C must have exactly one main function.

– There main ( ) function is used to tell compiler where the program starts.

– There are many forms of main are there:

- main ( )
- int main ( )
- void main ( )
- main (void)
- void main (void)
- int main (void)
- main (int argc, char * argv [ ])

Note: The last form of main is used to take command line argument. Here argc specified number of argument and argu is array of argument.

# define directive

⇒ A # define is a pre-processor directive

⇒ # define lines should not end with a semicolon.

⇒ A # define instruction defines value to a symbolic constant for use in the program.

⇒ Symbolic constant always written in upper case letter. We can’t change the value of symbolic constant during the execution of programs.

A symbolic constant can defined following manner

  # define symbolic-name value of constant.

Example: Write a program in C that calculate the area of circle.

/* ... calculating area of circle.... */

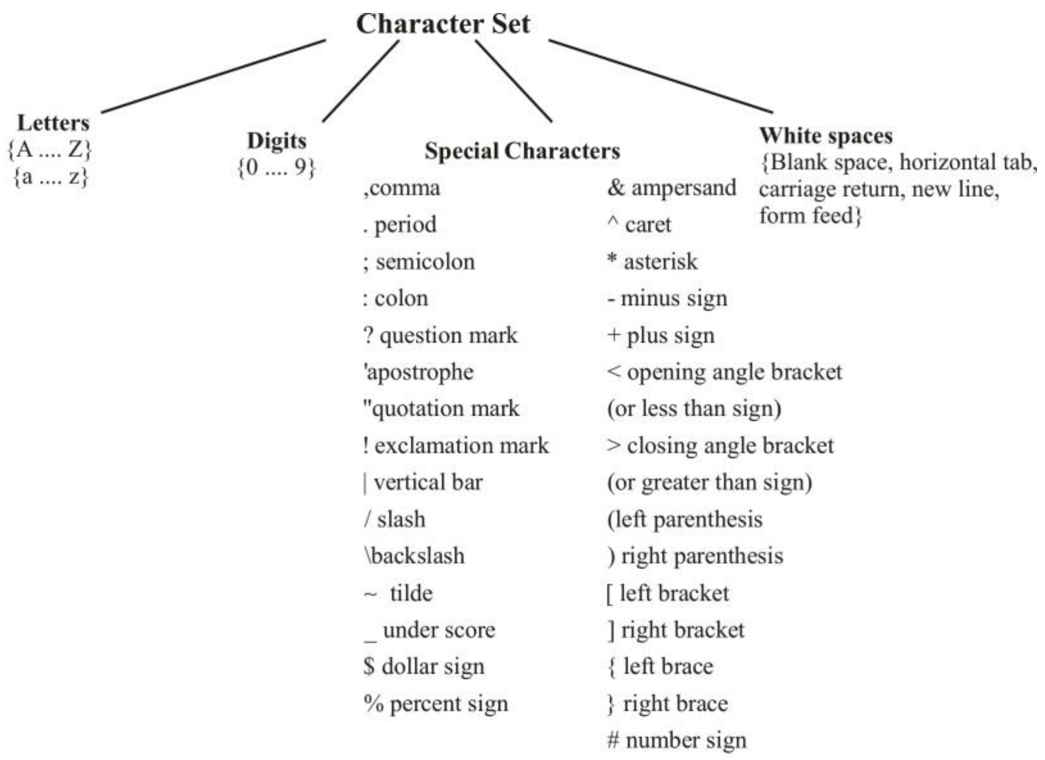
# define PI3.14 / * symbolic constant declaration */

```
main ( )
  {
    int radius;
    float area;
 radius = 12;
 area = PI * radius * radius;
  }
```

→ At first line we declare the symbolic constant PI to 3.14. We can't change the value in between program.

Character set in C

– The characters are used in words, variable name, expression. The character in C grouped into following categories.



Note: The compiler ignores white space unless they are a part of a string constant. White spaces may be used to separate words, but are prohibited between characters of keywords and identifiers.

Trigraph Characters

– Many keyboards don't support all the characters. So trigraph characters are used to enter certain characters that are not available in some keyboards.

– Each trigraph sequence consists of three characters. Two question mark followed by another characters.

Example:

If keyboard doesn't support # sign then we can use.

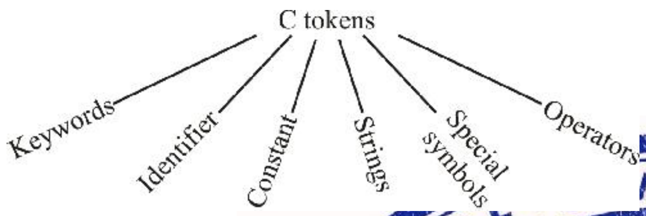
? ? = to denote # sign

– The following table shows the other trigraph sequence that represents the corresponding characters:

Trigraph sequence	Translation
?? =	# number sign
?? (	[Left bracket
??)	] right bracket
?? <	{ Left brace
?? >	} right bracket
?? !	| vertical bar
?? /	\ back slash
?? -	~ tlide

Tokens in C

– In a C program the smallest individual units are known as C tokens. C has six types of tokens. C programs are written using these tokens and the syntax of the language.



Keywords

→ Every C word is either a keyword or an identifier.

→ Meaning of keywords are fixed and can’t be changed.

→ All keywords must written in lower case.

→ These are the following keywords.

break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	size of	volatile
do	if	static	while

Identifiers

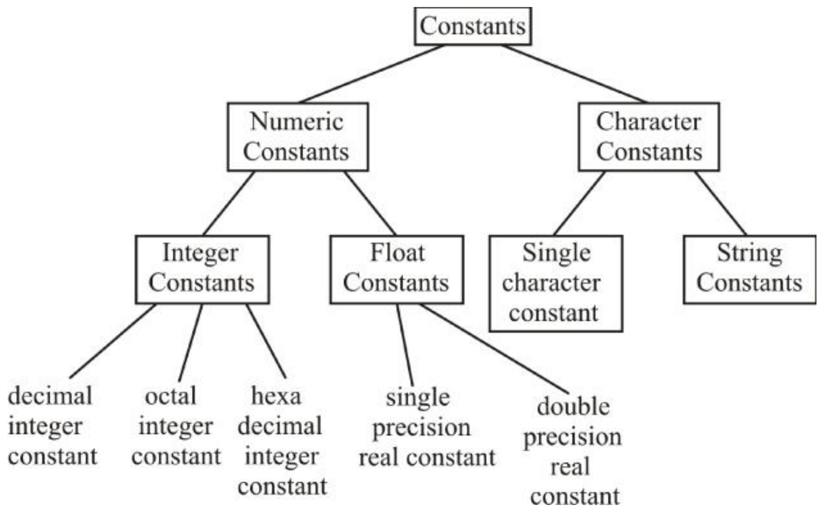
– Identifiers refers to the names of variables, functions and arrays. These are user defined names and consists of a sequence of letters and digits, rules for identifiers:

1. First character must be an alphabet (or underscore)
2. It must consists of only letters, digits or underscore.
3. Only first 31 characters are significant
4. We can’t use keyword name as identifiers
5. It must not contain white space.

Constants

Constants are fixed values that do not change during the execution of programs.

Type of constant



Variables

- A variables is a data name that may be used to store a data value.
- A variable may take different values at different times during execution.
- Variable names may consist of letters, digits and the underscore (_) character,

Rules for naming variable:

- They must begin with letter or underscore.
- First 31 character are significant
- Uppercase and lowercase are different. That is, the variable Radius is not same as Radius or RADIUS
- It should not be a keyword
- White space is not allowed.

Example:

area (valid) _area (valid) 0-area (invalid)

area (valid) $ area (invalid) are a (invalid)
↓
white space

Data types

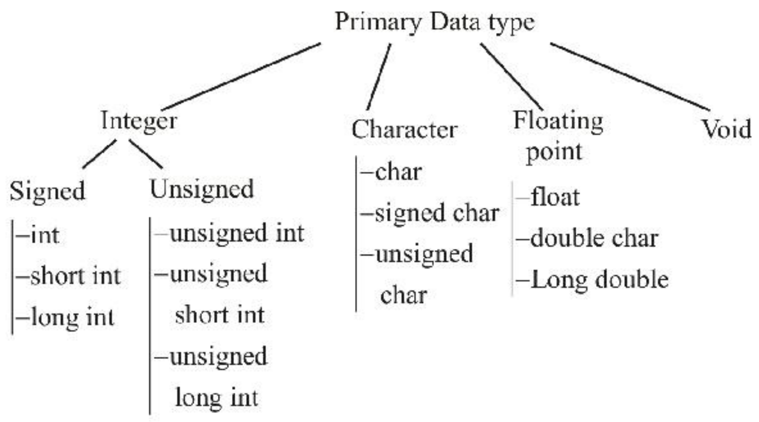
There are three categories of data types are available in C:

(i) Primary (or fundamental) data types

(ii) Derived data types

(iii) User – defined data types

Primary data type



Type	Size (bits)	Range
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int or signed int	16	-32,768 to 32,767
unsigned int	16	0 to 65535
short in or		
signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or		
signed long int	32	-2,147, 483, 648 to 2, 147, 483, 647
unsigned long int	32	0 to 4, 294, 967, 295
float	32	3.4e – 38 to 3.4e + 38
double	64	1.7e – 308 to 1.7e + 308
long double	80	3.4e – 4932 to 1.1e + 4932

Declaration of variables

– Declaration of variable have two significance,

(a) It tells the compiler what the variable name is

(b) It specifies what type of data the variable will hold.

– The declaration of variables must be done before they are used in the program declaration of variable have following form:

$\boxed{\text{data-type } V_1 \ V_2, \ldots, V_n;}$

For example:

int count;
float radius, area;
double ratio;

* User – defined type declaration

(1) “type definition” allows user to define an identifier that would represent an existing data tyre. It takes the following form:

$\boxed{\text{typedef. type identifier,}}$

Example:

typedef int unit;

Here units symbolizes int. It can later used to declare variable like:

Units count; (It will declare count of integer)

Enumerated data type

→ Enumerated data type is used to declare variables that can have one of the values enclosed within the braces. The values are known as enumeration constants.

  It takes the following form:

  Enum identifier {value 1, value 2, ... , value 3};

→ Here identifier is a user defined enumerated data type have one of the enclosed valves.

→ After this declaration we can declare variable to be of this type as enum identifier V_1, V_2 ...., V_n.

Example:

enum day {monday, tuesday, ..., sunday};
enum day week_st, week_end;
week_st = monday;
week_end = sunday;

Storage classes

There are two attributes associated with every C variable

- *data type* (e.g. **int**, **double**, etc.)
- *storage class*

The ***storage class*** of a variable tells the compiler *how* the variable is to be stored

It also specifies *how long* the variable remains in existence, i.e. its ***lifetime***

Thirdly, it specifies the variable's *visibility*, i.e. its ***scope***

The *scope* of a variable determines where, in the program, the variable can be accessed

There are four storage classes in C

i. ***automatic***
ii. ***external***
iii. ***static***
iv. ***register***

The corresponding keywords in C to specify such are

(i) **auto**
(ii) **extern**
(iii) **static**
(iv) **register**

Also, variables have a *default storage class* if we do not specify otherwise.

Scope Rules

- The basic rule of scoping:
  Identifiers are accessible only with the block in which they are declared
  They are unknown outside the boundaries of that block
  Blocks may be defined inside of other blocks
  A variable may be redefined inside an inner block
  The innermost block's variable declarations take precedence over the outer block

Note

- The *scope* of an object determines who can see what
- The *lifetime* of an object determines when it is activated and when it is deactivated
- Notice that inner blocks can see the outer block's environment
- However, the outer block *cannot* see inside the inner block

Example: Consider the following code fragment:

```
{
        int m = 5;                    /*outer block*/
        printf ("m=%d", m);           /*5 is printed*/
        {
                int m = 10;           /*inner block*/
                printf ("m=%d, m); /*10 is printed*/
        }
        printf ("m=%d", m);           /*5 is printed*/
}
```

An equivalent code fragment is:

```
{
        int m_outer = 5;
        printf ("m=%d", m_outer);          /*5 is printed*/
        {
                int m_inner = 10;
                printf ("m=%d, m_inner); /*10 is printed*/
        }
        printf ("m=%d", m_outer);          /*5 is printed*/
}
```

- Conceptually, a new **m** is born in the *innermost* block
- It lives throughout that inner block, hiding any external **m**
- It dies when we leave the block and the outer **m** then takes effect

The storage class: auto

- Variables declared within function bodies are by default automatic

- If a compound statement starts with variable declarations, then these variables can be acted on within the *scope* of the compound statement
- A compound statement with declarations will be called a *block* to distinguish it from those which do not begin with declarations

- Declarations of variables within a block are *implicitly* of storage class *automatic*

typical code fragment	Equivalent code fragment

{
    int m, n;
    double o;
    ...
}
	{
	    auto int m, n;
	    auto double o;
	    ...
	}

- When the block is *entered*, the system *allocates memory* for the automatic variables
- This is usually performed by *incrementing* the **stack frame pointer**
- Automatic variables defined within such a block are considered **local** to that block
- When the block is *exited*, the system *deallocates the memory* it had allocated for the automatic variables.This is usually performed by *decrementing* the stack frame pointer
- The initial values of automatic variables, if not specified, is *undefined*
- In the following example, identify which variables are active and visible within each block
- Also draw the activation frame for each block

The storage class: register

- The **register** specifier may be used to declare heavily used variables
- Hint to compiler to try to optimize use of variable
- This does *not* guarantee that a register will actually be used

example

```
register int x;
register char c;
```

- Can only be applied to *automatic* variables, i.e. function or function arguments:

```
void f (register unsigned m, register long n)
{
 register int i;
 ...
}
```

- The register declaration is taken as *advice* to the compiler
- If a register variable cannot be stored in a register, it defaults to *automatic*

- Cannot take the *address* of a register variable (more on this later)

```
register float radius;

printf ("Enter radius: ");
scanf ("%f", &radius); /*illegal*/
```

- The compiler forbids us from taking the address of **radius** because it may be stored in a hardware register, not memory

Compilers with good optimizers can do the job without register allocations. However, it may be useful in “guaranteeing” optimal register allocation in critical code.