Chapter 2

Tokens & Data Types

Dr. Niroj Kumar Pani

nirojpani@gmail.com

Department of Computer Science Engineering & Applications

Indira Gandhi Institute of Technology

Sarang, Odisha

Chapter Outline…

- Tokens: An Overview
- Character Set
- Keywords
- Identifiers
- Data Types
- Variables
- Constants
- Promotion, Truncation, and Type Casting
- Assignments

Tokens: An Overview

- The smallest individual units in a program are known as tokens / program constructs.
- C has following tokens:
  - Character Set
  - Keywords
  - Identifiers
  - Variables
  - Constants
  - Operators
- All C programs are written using these tokens and the syntax of the language.

In this chapter, we will study all these tokens (except operators, which is discussed in the next chapter) and the data types in C.

Character Set

- **The C character set consists of the following:**

Alphabets (Total 52)	A B C ... Z a b c ... z
Digits (Total 10)	0 1 2 3 4 5 6 7 8 9
Special Characters (Total 29)	, . ; : ? ' " ! | / \ ~ _ $ % & ^ * - + < > ( ) [ ] { } #
White Space Characters (Total 6)	blank space new line carriage return form feed horizontal tab vertical tab

A total of 97 characters.

Keywords

- **[Definition]: Key words are pre-defined words having special meaning and special purpose.**
- **There are 32 keywords in C** (ANSI C keywords in dictionary order are shown below):

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 keywords are in lower case.**

Identifiers

- Every word in C is either a keyword or an identifier.
- [Definition]: Identifiers refers to the names (user-defined names) given to program elements like variables, arrays, pointers, functions etc.
- Rules for Identifiers (giving names):
  1. An identifier may consist of any combination of the following:
     - Uppercase characters: `A B ... Z`
     - Lowercase characters: `a b ... z`
     - Digits: `0 1 2 3 4 5 6 7 8 9`
     - Underscore: `_`
  2. It should either start with an alphabet or an underscore.
  3. It should not be a keyword.
  4. Upper case and lower-case characters are different.
  5. Only first 31 characters are significant (in ANSI C).

- **Examples:**

Identifier ?	Remark
`Name`	A valid identifier
`group.`	An invalid identifier ; contains the special character .
`A14`	A valid identifier
`14A`	An invalid identifier ; starts with a digit
`#MEAN`	An invalid identifier ; contains the special character #
`heLLO`	A valid identifier
`First_salary`	A valid identifier
`_team`	A valid identifier
`basic-hra`	An invalid identifier ; contains the special character -
`over time`	An invalid identifier ; contains a white space
`void`	An invalid identifier ; a keyword
`Void`	A valid identifier

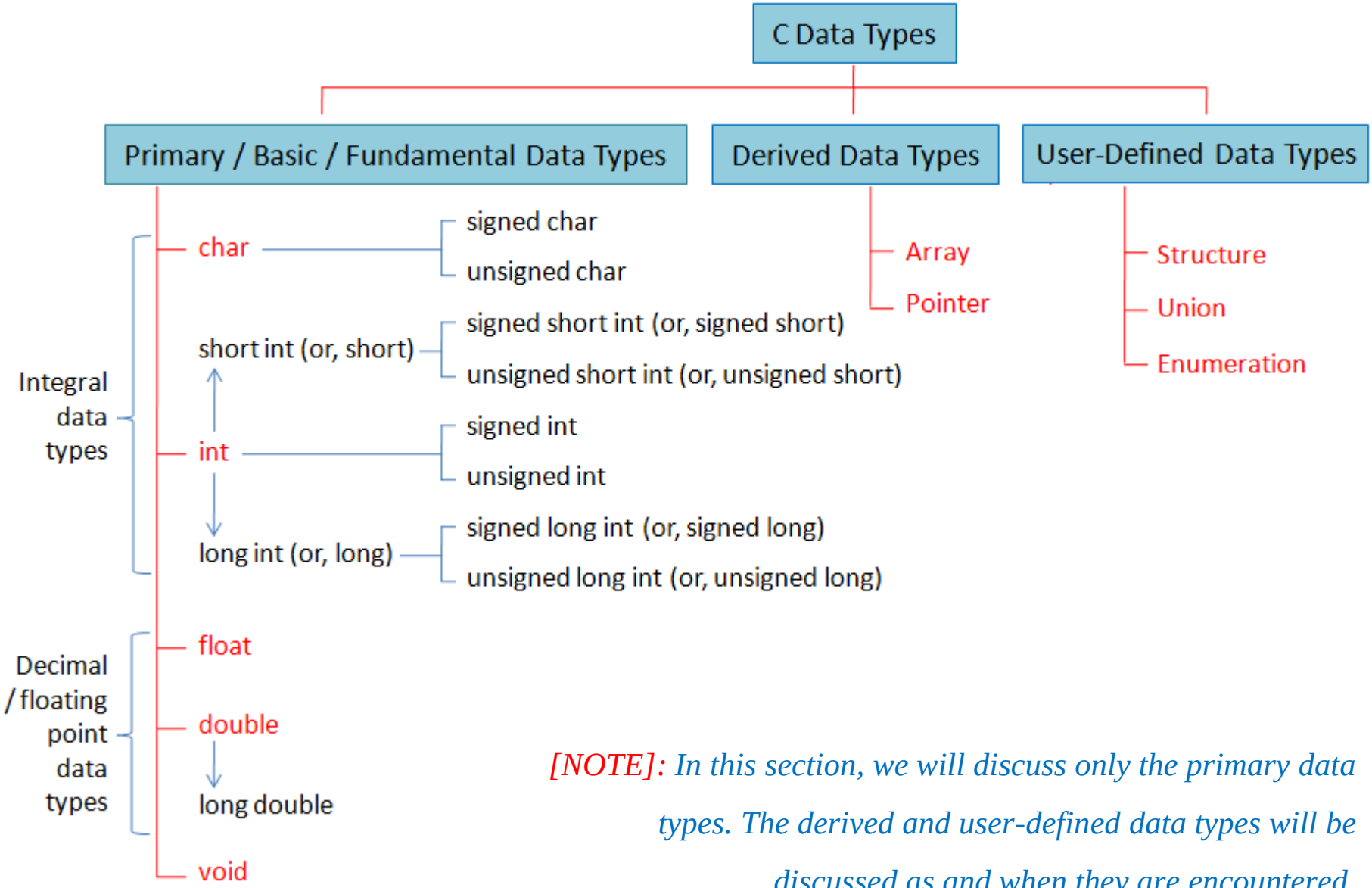
- [NOTES]: (1) Identifier names should be meaningful.

  (2) Avoid using underscore as the 1st character of an identifier, because many of the identifiers in the system library starts with an underscore.

Data Types

[NOTE]: Before discussing other tokens let's 1st discuss the data types in C.

- **[Definition - Data Type]: Data type (of a system) refers to**
  - **A set of values (that the system understands), and**
  - **A set of operations that can be performed on these values.**
- **A detailed classification of C data types is given in the** next slide.



[NOTE]: In this section, we will discuss only the primary data types. The derived and user-defined data types will be discussed as and when they are encountered.

Primary Data Types

- **C mainly defines five primary data types:**
  - **`char:` Refers to a single C character** (Characters are stored in ASCII format).
  - **`int:` Refers to an integer (a whole number).**
  - **`float:` Refers to a floating-point number (a number containing a decimal point and/or an exponent)**
  - **`double:` Refers to a *double* precision floating-point number.**

    [NOTE]: The term "precision" refers to the number of significant digits after the decimal point. For `double` it is *two times* as that of `float`. In most of the compilers, the number of significant digits after the decimal point for `float` is 6. So, for `double` it is 12 (however, it is compiler dependent).
  - **`void:` Refers to no value** (`void` is normally used to specify the return type of a function and to declare generic types).
- `char` **and** `int` **collectively are called integral data types. float and double collectively are called decimal / floating-point data types.**

- **The data types `char`, `int`, and `double` can be further augmented by applying data type qualifiers / modifiers.**
  - **Size modifiers (that alters the size): short & long.**
  - **Sign modifiers (that alters the sign): signed & unsigned.**
- **Different data types supports different modifiers.**
  - **On `char`: Only sign modifier are applicable ;**
    - **`char` can be `signed` or `unsigned`.**
  - **On `int`: Both size and sign modifier are applicable.**
    - **We can have `shot int` (or simply `short`) & `long int` (or simply `long`) in addition to `int`, each of which can be `signed` or `unsigned`.**
  - **On `double`: Only size modifier is applicable.**
    - **We can have `long double` in addition to `double`.**

- `char` **and all** `int` **types are by default signed. i.e.,**
  - `char` **&** `signed char` **are same.**
  - `short int (short)` **&** `signed short int (signed short)` **are same.**
  - `int` **&** `signed int` **are same.**
  - `long int (long)` **&** `signed long int (signed long)` **are same.**

The list of all primary data types with their size & range are given in the next slide.

[List of ALL Primary Data Types with their Size & Range]

Primary Data Type	Size (Bytes)	Range
char / signed char	1	-128 to +127 (Characters are stored in ASCII format)
unsigned char	1	0 to 255
short int / signed short int	1	-128 to +127
unsigned short int	1	0 to 255
int / signed int	2	-32768 to +32767
unsigned int	2	0 to 65535
long int / signed long int	4	-2, 147,438,648 to + 2, 147,438,647
unsigned long int	4	0 to 4,294,967,295
float	4	3.4E -38 to 3.4E +38
double	8	1.7E -308 to 1.7E +308
long double	10	3.4E -4392 to 1.1E +4392

[NOTE]: The actual size of a data type is compiler dependent. The sizes given here are for most of the compilers. However, to know the exact size of a data type in a particular compiler one can use the `sizeof ()` *operator.*

[NOTE]: `typedef` Statement

- **The `typedef` statement is used to give new names to existing data types.**
- **Syntax:**

```
typedef existing_data_type_name new_name_given_by_the_user;
```

- **Programming Example:**

```
/* PR2_1.c: Use of typedef statement */

#include <stdio.h>
#include <conio.h>

void main()
{
    typedef unsigned long int ulong;   /* declares ulong as a new type,
                                          equivalent to unsigned long.
                                          After this declaration ulong can be used
                                          instead of unsigned long int */
    ulong i = 66666666; /* declares the variable i to be of type ulong
                           (i.e., unsigned long)*/
    printf("\n i = %ld ", i); /* ld is the format specifier for unsigned long int */
    getch();
}
```

Output

```
i = 66666666
```

Variables

- **[Definition]:** A variable is a named memory location that is used to store a value (a constant) [constants are discussed next].
  - Unlike constants, that remain unchanged during program execution a variable may take different values at different time.
- **Variable Names:**
  - A variable name is an identifier.
  - So, the rules that are applicable for making identifiers are also applicable to variable names.
- **Variable Declaration:**
  - In C, every variable is associated with a data type. It specifies 2 things
    - The type of data that can be stored within the variable (the memory location).
    - The size (in bytes) allocated for the variable.
  - The data type of a variable is specified during its declaration.

- **Declaration Syntaxes** *(Demonstrated through the following examples):*

```
int i;
int j;
float k;
```

```
int i, j; /* Variables of one type can be declared in one
             statement, separated by commas*/
float k;
```

- **Every variable must be declared before it is used. In C ALL variable must be declared before the 1st executable statement.**

Variable Initialization:

- **Syntaxes** *(Demonstrated through the following examples):*

```
int i; /* Declaration*/
i = 5; /* Initialization*/
```

```
int i = 5; /* Initialization at the place of
              declaration*/
```

```
int i = 5, j = 10;
```

```
int i = 5, j = 10;
int k = j/i;
```

Constants / Literals

- **[Definition]: A constant (also called, literal) is a value that can be stored in a memory location.**
  - **A constant doesn't change** (For example, the number 5 is always 5).
- **Depending upon the basic data types in C, we can have following constants:**
  - **Character constant**
  - **Backslash character constant**
  - **String constant**

  } Corresponds to character data type
  - **Integer constants**

  } Corresponds to integer data type
  - **Floating-point (or, real) constants**

  } Corresponds to floating-point data type

Character Constants

- A character constant is a single C character enclosed within a single quote.
- **Examples:** `'A', 'a', '$', '5', ',', ' '`
- **[NOTE 1]:** `'5'` and `5` are different.
  - `'5'` is a character whereas `5` is an integer.
- **[NOTE 2]:** As pointed out earlier, in C, characters are stored in ASCII format.

  i.e., if we declare `char ch1 = 'A';` then actually 65 is stored within `ch1`.

  Hence, in C, characters can be treated as integers, and vice-versa
  - Integers within the range -128 to +127 can be considered as *signed characters* and integers within the range 0 to 255 can be considered as *unsigned characters*.
  - When an integer beyond these permissible ranges is assigned to a character variable, it will automatically be wind-up within this range.

[See the programming example next slide]

Programming Example:

```
/*PR2_2.c: Representation of characters in C*/

#include <stdio.h>
#include <conio.h>

void main()
{
     char ch1 = 'A';  /* character variable initialized to a character constant*/
     char ch2 = 65;  /* character variable initialized to an integer*/
     char ch3 = 321;  /* 321 will wind-up to 65 (321-256 = 65)*/
     int i1 = 65;  /* integer variable initialized to a integer constant*/
     int i2 = 'A';  /* integer variable initialized to a character constant*/

     printf ("\n ch1 = %c, %d", ch1, ch1);
     printf ("\n ch2 = %c, %d", ch2, ch2);
     printf ("\n ch3 = %c, %d", ch3, ch3);
     printf ("\n i1 = %c, %d", i1, i1);
     printf ("\n i2 = %c, %d", i2, i2);

     getch();
}
```

Output

```
ch1 = A, 65
ch2 = A, 65
ch3 = A, 65
i1 = A, 65
i2 = A, 65
```

Backslash (Escape) Character Constants / Escape Sequences

- **C supports some non-printing characters and some special characters that are used in output functions, to be represented as backslash character constants** (also known as, escape character constant, or Escape Sequence)

- **[NOTE]: Each escape sequence represents one character, although they actually consist of two characters.**

Escape Sequence	Meaning
\a	Alert (audible beep)
\b	Back space
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quote
\"	Double quote
\?	Question mark
\0	Null

[List of escape sequences in C]

String Constants

- **A string constant is a sequence of characters enclosed within double quotes.**
- **Examples:** `"A", "5", "hello", " ", "MCA Students."`
- **[NOTE]:** `5, '5'` **and** `"5"` **are all different.**
  - `5` **is an integer. It occupies** 2 bytes **of memory and is represented in the memory as** 00000000 00000101 (Binary equivalent of 5 is 101)
  - `'5'` **is character. It occupies** 1 byte **of memory and is represented in the memory as** 00100101 (ASCII value of 5 is 37).
  - `"5"` **is string. It occupies** 2 bytes **of memory, one for the character** 5 **and other for the** null character (\0). **Every string terminates with a** null character (\0). **Hence** `"5"` **is represented in the memory as** 00100101 00000000 (ASCII value of 5 is 37, and ASCII value of \0 is 0)

Integer Constants

- **An integer constant refers to a sequence of digits.**
- **An integer constant can be expressed in any one of the following forms:**
  - **Decimal (Base 10):** [This form is most common]
    - **A sequence of digits from** 0 to 9.
    - **Optionally preceded by an** + **or** - **sign (Default sign is** +**).**
    - **Should not precede with a** 0 (zero).
  - **Octal (Base 8):**
    - **A sequence of digits from** 0 to 7.
    - **Optionally preceded by an** + **or** - **sign (Default sign is** +**)**
    - **Indicated by preceding** 0 (zero) **after the** + **or** - **sign (if it is there)**
  - **Hexadecimal (Base 16):**
    - **A sequence of digits from** 0 to 9 and A to F.
    - **Optionally preceded by an** + **or** - **sign (Default sign is** +**).**
    - **Indicated by preceding** 0X (zero and capital X) **or** 0x (zero and small x) **after the** + **or** - **sign (if it is there).**

- **By default, an integer constant is assumed to be `signed int` (or `int`). If we want an integer constant to be explicitly specified as other integer types, then we have to suffix appropriate qualifiers:**
  - `s` **or** `S`**: short**
  - `l` **or** `L`**: long**
  - `u` **or** `U`**: unsigned**
  - `us` **or** `US`**: unsigned short**
  - `ul` **or** `UL`**: unsigned long**

[NOTE 1]: Some compilers may not support some or even all of these qualifiers.

[NOTE 2]: Normally, these suffixes are not needed. The compiler *automatically assumes* the data type of an integer constant to be same as that of the variable to which the constant is assigned. These suffixes are only useful in situations where the value may fit in an `int` but your want to treat it as `long`.

- **Examples: The following examples will clarify the concepts**

Integer constant?	Remarks
75	Signed integer constant in decimal form
0113	Signed integer constant in octal form
-0x32	Signed integer constant in hexadecimal form
078	Illegal: 8 is not an octal digit
379u	Unsigned integer constant in decimal form
075ul	Unsigned long integer constant in octal form
666l	Signed long integer constant in decimal form
0x322UU	Illegal: wrong suffix (two U)
0X 543	Illegal: Space is not permitted.

- **[NOTE]: When an integer constant beyond the permissible range is assigned to an integer variable, it will automatically wind-up within the permissible range.**

Floating-Point (Real) Constants

- **A floating-point constant is a number containing a decimal point and/or an exponent.**
- **According to the ANSI/ISO standard for C, a floating-point constant can only be expressed in decimal (base 10) form** (unlike integers, which can be decimal, octal, or hexadecimal). **Two notations are used to represent them (decimal floating-point constants):**
  - **Decimal Point Notation:**
    - **In this notation, the constant is represented as a whole number (in base 10) followed by a decimal point and a fractional part (in base 10).**
    - **Optionally preceded by an + or - sign (Default sign is +).**
    - **It is possible to omit digits before or after the decimal point.**
    - **Examples:** `0.56, -3.75, 241., .976, -.71, +.5`

- **Floating-Point / Exponential / Scientific Notation:**
  - Exponential notation is useful in representing real numbers that are very large or very small.
  - The exponential notation consists of two parts:
    - **Mantissa:** It is either a real number expressed in decimal notation or an integer.
    - **Exponent:** It is an integer that represents the position of the decimal point.

  The letter E or e separates the mantissa and the exponent.

  The number is represented as: mantissa E exponent

  Which is equivalent to: $\text{mantissa} \times 10^{\text{exponent}}$

  - **Examples:** `0.56e4, 12e-2, 1.e+5, 3.18E3, -1.2E-1, -9E7`

- **[NOTE - Normalized Floating-Point Form]:** A floating point number is said to be normalized if there is a single non-zero digit before the decimal point.

  Examples: 61.378e +3 (not normalized), 6.137e+4 (normalized)

- **By default, all floating-point constants are assumed to be `double`. If we want a floating-point constant to be explicitly specified as other floating-point types (`float` or `long double`), then we have to suffix appropriate qualifiers:**
  - **`f` or `F`: float**
  - **`l` or `L`: long double**

Examples:

Floating-point constant?	Remarks
0.56	double
0.56f	float
0.56L	long double
-0.98e+3	double
-0.98e+3F	float
-0.98e+3L	long double
-98e+3	double
+65.9Lf	Illegal: wrong suffix (Lf)

- **[NOTE]: C99 allows floating-point constants to be expressed in hexadecimal form, in addition to the decimal form.**
  - **Hexadecimal floating-point constants are represented by using the only the exponential notation (there is no decimal point notation for them).**
  - **Example (Hexadecimal floating-point constants):**

```
-0x0.98p+3
```

The hexadecimal floating-point constant is different from a decimal floating-point constant (for example, `-0.98e+3`), in three ways:

- **A hexadecimal floating-point constant is suffixed by 0x.**
- **In hexadecimal floating-point constant we use the letter P or p instead of the letter E or e.**
- **The decimal floating-point constant**

$$-0.98e+3 = (-0.98\times 10^{+3})_{10}$$

where as, the hexadecimal floating-point constant

$$-0x0.98p+3 = (-0.98\times 2^{+3})_{10}$$

[NOTE]: User-Defined Constants

- The constants that we have discussed so far are pre-defined constants (they are defined by the system).
- In addition to these pre-defined constants, C also permits the user to define his/her own constants (user-defined constants), by using:
  - The `#define` preprocessor directive (symbolic constants)
  - The `const` keyword

■ Using `#define` Directive (Symbolic Constant):

- **Syntax:** `#define symbolic_name constant_value`

- **Programming Example:**

```
/* PR2_3.c: Use of #define. This program calculates the area of a circle */

#include <stdio.h>
#include <conio.h>
#define PI 3.141    /* Use of #define. It defines PI as a symbolic constant
                       having  value 3.141. By convention, the symbolic constants
                       are written in UPPERCASE. All instance of  PI will be
                       replaced by 3.141 by the preprocessor. */
void main()
{
   float radius;
   float area;

   printf ("\nEnter the radius of the circle: ");
   scanf ("%f", &radius);
   area = PI*radius*radius;
   printf ("\nThe area of the circle is: %f", area);

   getch();
}
```

Output

```
Enter the radius of the circle: 5.3
The area of the circle is: 88.230698
```

Using the Keyword `const`:

- **Syntax:**

```
const data_type variable_name = constant;
```

- **Programming Example:**

```
/* PR2_4.c: Use of const. This program also calculates the area of a circle */

#include <stdio.h>
#include <conio.h>

void main()
{
   const float PI = 3.141; /* Use of const. It tells the compiler that the
   float radius;           value of the variable PI can't be changed
   float area;             throughout the program. It is fixed to 3.141 */

   printf ("\nEnter the radius of the circle: ");
   scanf ("%f", &radius);
   area = PI*radius*radius;
   printf ("\nThe area of the circle is: %f", area);

   getch();
}
```

Output

```
Enter the radius of the circle: 5.3
The area of the circle is: 88.230698
```

- **Both `#define` and `const` solve the same purpose. Then what is the difference?**

  The key difference is,

  - `const` **is handled by the *compiler.*** `const` **tells the compiler that the value of** PI **is fixed to** 3.141 **throughout the program. It can't be changed.**
  - **But** `#define` **is handled by the *preprocessor* that replaces all instances of** PI **by** 3.141.

[NOTE]: Declaring a Variable as Volatile

- **ANSI C standard defines a qualifier `volatile` that is used to tell the compiler that a variable's value may be changed at any time by some external sources (from outside the program) in addition to its own program.**

  - **Syntax:**

    ```
    volatile int mark;
    ```

    here the value of `mark` can be altered/set by its own program as well as some external factors.

- **If we exclusively want that the value of a variable should ONLY be altered by some external processes, but NOT by it own program we must declare the variable as both `volatile` and `const` as shown below:**

```
volatile const int mark = 100;
```

Promotion, Truncation, and Type Casting

Promotion

- When a variable of lower type (which holds lower range of values) is converted to a higher type (which holds higher range of values), it is called promotion.
- Example:

```
int i = 5;
float f;
f = i;
```

- [NOTE]: Promotion can take place only between compatible data types. i.e., `char` to `int`, `int` to `float`.

Truncation

- When a variable of higher type (which holds higher range of values) is converted to a lower type (which holds lower range of values), it is called truncation.
- **Example 1:**

```
float f = 5.3;
int i;
i = f;
```

- **Example 2:**

```
float f = 5.3; /* 5.3 is by default "double". Here it is truncated to float*/
```

- **[NOTE]: Truncation can take place only between compatible data types. i.e., `int` to `char`, `float` to `int`.**

Type Casting

- Both in promotion and truncation the conversion from one type to other takes place automatically.
- In addition to this automatic conversion, we can also forcibly convert a variable of one type to another type (there are situations in which we have to do this) by using type casting.
- Syntax:

```
x = (data_type)y;
```

where `x` and `y` are variable of compatible data types.

- Example: The following codes show the need of type casting and how it is done:

```
int i = 12;
int j = 5;
float f;
f = i/j;  /* This will output 2.000000, which is not the correct result*/
f = (float)i/j;  /* This will output the correct result 2.4*/
```

Assignments

Complete the experiments given in "Lab Manual - Section 2".

End of Chapter 2