Unit 2

Console I/O and Preprocessor

The input and output functions

- One of the essential operations performed in a C language programs is to
  - ✓ Provide input values to the program
  - ✓ Output the data produced by the program to a standard output device.
- One method for input is to use the function scanf which can be used to read data from a key board.
- For outputting results we use extensively the function printf which sends results out to a terminal.
- There exist several functions in 'C' language that can carry out input output operations.
- These functions are collectively known as standard Input/Output Library.
- Each program that uses standard input / output function must contain the statement.
  - ✓ #include<stdio.h>
- at the beginning.

Reading and writing a character

- The basic operation done in input output is to read characters from the standard input device such as the keyboard and to output or writing it to the output unit usually the screen.
- The getchar function can be used to read a character from the standard input device.
- The scanf can also be used to achieve the function.
- The getchar is a macro that gets a character from the standard input device.
- It returns a character read after converting it to an int without sign extension.
- On error it returns EOF.
- The getchar has the following form.
  - variable name = getchar();
  - variable name is a valid 'C' variable that has been declared.
- When this statement is encountered the computer waits until a key has been pressed and then assigns this character as a value to the getchar function.
- The getchar function can also be called successively to read characters contained in a line of text.
- Example
  - char c;

  c = ' ';

  while(c!='\n')   c = getchar();
- The putchar is a macro that outputs a character on to the standard output device.
- On success it returns the characters given by the variable.

- On error it returns an EOF.
- The general form is
  - putchar (variable name);
  - where variable is a valid C type variable that has already been declared
- It displays the character contained in the variable name at the terminal.
- Example
  - puctchar('n');
  - putchar('\n');
    - Causes the cursor on the screen to move to the beginning of the next line.

Problems with getchar()

- It buffers until ENTER is pressed.
  - ✓ It is called as 'line-buffered' input.
- It inputs only one character each time it is called and line buffering may leave one or more characters waiting in the input queue.

Alternatives

- Alternatives to getchar() is
  - ✓ getch()
  - ✓ getche()
- Cannot be used in gcc since it requires the header file 'conio.h'

String input and output

- The gets function relieves the string from standard input device while puts outputs the string to the standard output device.
- A string is an array or set of characters.
- The function gets
  - accepts the name of the string as a parameter
  - fills the string with characters that are input from the keyboard till newline character is encountered i.e till we press the enter key.
  - appends a null terminator as must be done to any string and returns.
  - Stores the string at the address pointed to it by its argument.
  - ✓ It performs no boundary checks on the array that is receiving the input.
    - ɥ User can enter more characters than that can be stored in the array.
- The standard form of the gets function is
  - gets (str)

- The puts function displays the contents stored in its parameter on the standard screen followed by a newline character.
- It recognizes the
- The standard form for the puts character is
  - puts (str)
  - Where str is a string variable.

Formatted input

- It refers to an input data that has been arranged in a particular format.
- scanf is a general purpose console input-output routine.
- It can read all built-in data types and automatically convert numbers into proper internal format.
  - int scanf(const char *control string, ….);
    - Returns the number of data items successfully assigned a value .
    - If an error occurs it returns EOF.
- The control string determines how values are read into the variables pointed to in the argument list.
- They contain
  - Format specifiers
  - White-space characters
  - Non-white-space characters
  - The argument arg_1, arg_2,……arg_n specify the address of locations where the data is stored.
  - Control string and arguments are separated by commas.
  - Control string contain field specifications which direct the interpretation of input data.
- In scanf they have a form of
  - %[width] type_char
- Each format specifier begins with the percent character (%).
- **width**
  - It is optional
  - It specifies the maximum number of characters to read
- **type_char**
  - It is required
  - They allow us to define the type in which we want the values to be read.
- **Scanf type_char**
  - **a** - float-point value (C99)

  - **c** – single character
  - **d -** Decimal integer
  - **e** –Floating point number
  - **f -** Floating point number
  - **g-** Floating point number
  - **i -** Decimal, octal or hexadecimal integer
  - **n** – receives an integer value equal to the number of characters read so far.
  - ✓ **o -** Octal integer
  - ✓ **p** - pointer
  - ✓ **s -** character string
  - ✓ **u -** unsigned decimal integer
  - ✓ **x -** Hexadecimal integer
  - ✓ **[]** – scans for a set of characters
  - ✓ **%** - percent sign
- Input data items must be separated by spaces, tabs or newlines.
- Punctuation marks do not act as separators.
- It bypasses any white space character.

Inputting Integer Numbers

- The field specification is %xd.
  - ✓ Here percent sign (%) denotes that a specifier for conversion follows and x is an integer number which specifies the width of the field of the number that is being read.
  - ✓ The data type character d indicates that the number should be read in integer mode.
- scanf (“%3d %4d”, &sum1, &sum2);
  - ✓ If the values input are 175 and 1342 here value 175 is assigned to sum1 and 1342 to sum 2.
  - ✓ Suppose the input data was 1342 and 175.
  - ✓ The number 134 will be assigned to sum1 and sum2 has the value 2 because of %3d the number 1342 will be cut to 134 and the remaining part is assigned to second variable sum2.
  - ✓ If floating point numbers are assigned then the decimal or fractional part is skipped by the computer.
- Input field can be skipped by giving an * in place of the width field.
- Example
  - ✓ scanf(“%d%*d%d”,&i,&j);

printf(“i=%d,j=%d”,i,j);

- ✓ 12 34 56
- ✓ i =12, j=56

Inputting Real Numbers

- Field specifications are not to be used while representing a real number therefore real numbers are specified in a straight forward manner using %f specifier.
- The general format of specifying a real number input is
  - ✓ scanf (“% f “, &variable);
- Example
- scanf (“%f %f %f”, &a, &b, &c);
  - ✓ With the input data
  - ✓ 321.76, 4.321, 678
  - ✓ The values 321.76 is assigned to a , 4.321 to b & 678 to C.
- If the number input is a double data type then the format specifier should be % lf instead of %f.

Inputting character strings

- Single character or strings can be input by using the character specifiers.
- The general format is
  - ✓ % xc or %xs
  - ✓ Where c and s represents character and string respectively and x represents the field width.
  - ✓ The address operator need not be specified while we input strings.
  - ✓ Example
    - պ scanf (“%C %15C”, &ch, name);
- Some versions of scanf support
  - ✓ %[characters]
  - ✓ %[^characters]
  - ✓ %[characters]
    - only characters specified within the brackets are allowed in the input string.
  - ✓ If the input contains any other character, the string will be terminated at the first occurrence of such a character.
  - ✓ %[^characters] –characters specified within the brackets are not allowed in the input string.
  - ✓ If the input contains any these character, the string will be terminated at the first occurrence of such a character.

Inputting mixed data types

- When one scanf statement is used to input mixed mode data, care should be taken to ensure that the input data items match the control specifications in order and type.

Rules for scanf

- Each variable to be read must have a field specification.
- For each field specification, there must be a variable address of proper type.
- Any non-whitespace character used in the format string must have a matching character in the user input.
- Never end the format string with whitespace.
- The scanf reads until
    - A whitespace character is found in a numeric specification
    - The maximum number of characters have been read
    - An error is detected
    - The end of file is reached.

Formatted Output

- The prototype of the printf() is
    - ✓ int printf(const char * control string, ….);
        - Returns the number of characters written
        - Returns a negative value if an error occurs
- Control string consists of three types of items
    - ✓ Characters that will be printed on the screen as they appear.
    - ✓ Format specifications that define the output format for display of each item.
    - ✓ Escape sequence characters.
- The control string indicates how many arguments follow and what are their types.
- The arguments should match in number, order and type with the format specifiers.
- In printf they have a form of
    - ✓ %[flag][width][.prec] type_char
- Each format specifier begins with the percent character (%).
- **flag**
    - ✓ It is optional
    - ✓ It is used for output justification, numeric signs.
    - ✓ - and 0 are flags used for left justification and 0 padding.
- **width**

  - ✓ It is optional
  - ✓ It specifies the minimum number of characters to print
- **.prec**
  - ✓ It is optional
  - ✓ It specifies the maximum number of characters to print
- **type_char**
  - ✓ It is required
  - ✓ They allow us to define the type in which we want the statements to be printed.
- **a -** Hexadecimal output in the form 0xh.hhhhp+d (C99)
- **A -** Hexadecimal output in the form 0Xh.hhhhP+d (C99)
- **c -** character
- **d -** Signed decimal integer
- **i -** Signed decimal Integer
- **e –** Scientific notation (lower case e)
- **E –** Scientific notation (upper case e)
- **f –** Decimal floating point
- **g –** Uses %e or %f, whichever is shorter
- **G –** Uses %E or %F, whichever is shorter
- **o –** Unsigned Octal
- **s -** Prints characters until a null-terminator is pressed or precision is reached
- u – unsigned decimal integers
- x – unsigned hexadecimal (lowercase letters)
- X – unsigned hexadecimal (uppercase letters)
- p – Pointer
- n - associated argument must be a pointer to an integer.
  - ✓ causes the number of characters written (upto the point at which the %n is encountered) to be stored in the integer
- % - prints a % sign
- By default all output is right justified.
- Force the output to be left justified by placing a '-' sign immediately after the '%'.

Output of Integers

- The format specification for printing an integer is
  - ✓ %[w]d

  - w specifies the minimum field for the output.
  - If the number is greater than the specified output field width it will be printed in full.

- The number is written right-justified in the given field width.
- printf("%d",9945);

9	9	4	5

- printf("%6d",9945);

		9	9	4	5

- printf("%2d",9945);

9	9	4	5

- printf("%-6d",9945);

9	9	4	5

  - ✓ Force the printing to be left-justified
- printf("%06d",9945);

0	0	9	9	4	5

  - ✓ Pad with zeroes the leading blanks by placing a 0 before the field specifier.

Output of Real numbers

- The syntax is
  - ✓ %[w.p] f
  - ✓ w indicates the minimum number of positions that are used for the display of the value
  - ✓ p indicates the number of digits to be displayed after the decimal point.
  - ✓ The value when displayed is rounded to p decimal places and print right-justified in the field of w columns.
- The default precision is 6 decimal places.
- The negative numbers will be printed with the minus sign.
- It takes the form
  - ✓ [-]mmm.nnn
- To display a real number in exponential notation
  - ✓ %[w.p] e
- It takes the form

- ✓ [-]m.nnnnnne[±]xx
- ✓ The length of the string of n's is specified by p.
- ✓ Default precision is 6.
- ✓ The field width w should satisfy the condition
  - ▪ w ≥ p+7
- ✓ The value will be rounded off and printed right justified in the field of w columns.

- printf("%7.4f",y);

9	9	.	5	4	6	7

- printf("%7.2f",y);

		9	9	.	5	5

- printf("%f",y);

9	9	.	5	4	6	7

- printf("%-7.2f",y);

9	9	.	5	5		

- printf("%10.2e",y);

		9	.	9	5	e	+	0	1

- printf("%11.4e",-y);

-	9	.	5	4	6	7	e	+	0	1

- printf("%e",y);

9	.	9	5	4	6	7	0	e	+	0	1

Printing a single character

- A single character can be displayed using
  - ✓ %[w]c
    - ɥ The character will be displayed right-justified in the field of w columns.
    - ɥ The display can be made left justified by placing a minus sign before the integer w.
    - ɥ The default value for w is 1.

Printing a string

- It takes the form
  - ✓ %[w.p]s
    - ɥ w specifies the field width for display.

- ɥ p instructs that only the first p characters of the string are to be displayed.
- ɥ The display is right justified.

Example – BANGALORE 560040

Specification	**Output**																			
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
%s	B	A	N	G	A	L	O	R	E		5	6	0	0	4	0				
%20s					B	A	N	G	A	L	O	R	E		5	6	0	0	4	0
%20.10s											B	A	N	G	A	L	O	R	E	
%.5s	B	A	N	G	A															
%-20.10s	B	A	N	G	A	L	O	R	E											
%5s	B	A	N	G	A	L	O	R	E		5	6	0	0	4	0				

Formatted output – other data types

- Format specifiers can be applied on 'd', 'i', 'o', 'u' to display short and long integers.
- l – long data type
- h – short data type
- It can also be applied to a 'n'
- If the compiler allows 'l' can also be used with 'c' to show wide character.

L – indicates long double and works with 'e', 'f', 'g'.

- ll and hh (C99)
  - ✓ Applied to 'd', 'i', 'o', 'u', 'x' or 'n'
  - ✓ hh – signed or unsigned
  - ✓ ll – signed or unsigned long long int

Modifiers - *, #

- #
  - ✓ Applied on 'g', 'G', 'f', 'e', 'E'
    - ɥ Will display decimal point even if no decimal deigits are there.

- ✓ Applied on 'x' or 'X'
  - ʯ The hexadecimal number will be printed with 0x prefix.
- ✓ Applied on 'o'
  - ʯ The number will be printed with leading zeroes.
- ✓ Applied on 'a' (C99)
  - ʯ Ensures that a decimal point will be displayed

▪ Some systems allow the user to define the field size at run time.

- ✓ printf("%*.*f",width, precision, number);
  - ʯ Both width and precision are given as arguments.
- ✓ Example
  - ʯ printf("%*.*f",7,2,num);

Mixed data output

▪ Mixed data types can be printed in one printf statement.

▪ printf uses the control string to decide how many variables are to be printed and what their types are.

Questions

▪ Briefly describe the syntax of printf and scanf functions using examples.

▪ If a=13.37, b= 0.00875 find the output of the following

- ✓ printf("%f\n",a); printf("%f\n",b);
- ✓ printf("%e\n",a); printf("%e\n",b);
- ✓ printf("%2f\n",a); printf("%3f\n",b);
- ✓ printf("%4e\n",a); printf("%3e\n",b);

Preprocessor

Introduction

▪ A unique feature of c language is the preprocessor.

▪ A program can use the tools provided by preprocessor to make the program easy to read, modify, portable and more efficient.

▪ Preprocessor is a program that processes the code before it passes through the compiler.

▪ It operates under the control of preprocessor command lines and directives.

▪ Preprocessor directives are placed in the source program before the main line.

▪ Before the source code passes through the compiler it is examined by the preprocessor for any preprocessor directives.

▪ If there is any command, appropriate actions are taken.

- The source program is then handed over to the compiler.
- Preprocessor directives follow the special syntax rules and begin with the symbol # in column1 and do not require any semicolon at the end.

Preprocessor directives – example

- #define - Defines a macro substitution
- #undef - Undefines a macro
- #include - Specifies a file to be included
- #ifdef - Tests for macro definition
- #endif - Specifies the end of #if
- #ifndef - Tests whether the macro is not def
- #if - Tests a compile time condition
- #else - Specifies alternatives when # if test fails

Categories

- The directives can be divided into three categories
  - ✓ Macro Substitution directives
  - ✓ File Inclusion directives
  - ✓ Compiler control directives

Macro substitution

- Macro substitution is a process where an identifier in a program is replaced by a pre defined string composed of one or more tokens.
- Use the #define directive for the task.
- It has the following form

  #define identifier token_string

- The preprocessor replaces every occurrence of the identifier in the source code by the token_string.
- All subsequent instances of the identifier in the source text will be replaced by the text defined by token_string.
- There are different forms of macro substitution.
  - ✓ Simple macro substitution
  - ✓ Argument macro substitution
  - ✓ Nested macro substitution

Simple Macro Substitution

- Simple string replacement is commonly used to define constants.
- Example

  #define PI 3.1415926

- Writing macro definition in capitals is a convention and not a rule.
- A macro definition can include more than a simple constant value it can include expressions as well.
  - ✓ #define AREA 7*12.36
  - ✓ #define SIZE sizeof(int) * 10
- When expressions are used, unexpected order of evaluation should be prevented.
- Example
  - ✓ #define D 45-23
  - ✓ #define A 89+90
  - ✓ ratio = D/A
  - ✓ ratio – 45-23/89+90
- The statement is completely different from the statement ratio = (45-23)/(89+90)
- The macro definition should be changed to
  - ✓ #define D (45-23)
  - ✓ #define A (89+90)
- The preprocessor performs a literal text substitution.
- No text substitution occurs if the identifier is within a quoted string.

Argument Macro Substitution

- The preprocessor permits us to define more complex and more useful form of replacements.
- It takes the following form.

  # define identifier(f1,f2,f3…..fn) string

- Notice that there is no space between identifier and left parentheses and the identifier f1,f2,f3 …. fn is analogous to formal arguments in a function definition.
- A simple example of a macro with arguments is

  # define CUBE (x) (x*x*x)

- If the following statements appears later in the program,

  volume=CUBE(side);

- The preprocessor would expand the statement to volume =(side*side*side)

- It increases the execution speed of the code
    - ✓ There is no function call overhead.
- If the macro is very large, there will be an increase in the size of the program due to duplicated code.

Nested Macro Substitution

- Use one macro in the definition of another macro.
- That is macro definitions may be nested.
- Example
    - ✓ #define N 25
    - ✓ #define M N+20
    - ✓ #define SQUARE(a) ((a)*(a))
    - ✓ #define CUBE(a) (SQUARE(a)*(a))
    - ✓ #define SIX(a) (CUBE(a)*CUBE(a))
- It is substituted as
    - ✓ ((SQUARE(a)*(a))*((SQUARE(a)*(a))
- Will be substituted as
    - ✓ (((a)*(a))*(a))*(((a)*(a))*(a)))
- Macros can be nested the same way as function calls.
- Example
    - ✓ #define HALF(a) ((a)/2)
    - ✓ #define Y HALF(HALF(a))

Undefining a Macro

- A macro can be undefined using
    - ✓ #undef macro_name
- It undefines a symbol specified by macro_name which was previously defined with a #define directive.
- It removes a previously defined definition of the macro name that follows it.

File inclusion

- The preprocessor directive

  #include "file name"

can be used to include any file in to your program.

- If the functions or macro definitions are present in an external file they can be included in your file.

- In the directive the filename is the name of the file containing the required definitions or functions.
- Alternatively this directive can take the form

  #include<filename>

- The directive treats the file specified by filename as if it appeared in the current file.
- #include"filename"
  - ✓ Searches the search path (current directory) first then the include path (standard directories).
- #include<filename>
  - ✓ Searches only the include path.
- Include files can have #include directives in them.
  - ✓ This is referred as nested includes.
- C89 allows at least 8 levels of nested inclusions.
- C99 allows at least 15 levels of nested inclusions.

Compiler control directives

- A preprocessor conditional compilation directive causes the preprocessor to conditionally suppress the compilation of portions of source code.
- These directives
  - ✓ test a constant expression or an identifier to determine
    - ʯ which token the preprocessor should pass on to the compiler
    - ʯ which tokens should be bypassed during preprocessing.
- It spans several lines
  - ✓ The condition specification line(beginning with #if, #ifdef, #ifndef)
  - ✓ Lines containing code that the preprocessor passes on the compiler if the condition evaluates to a nonzero value(optional)
  - ✓ The #elif line (optional).
  - ✓ Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional).
  - ✓ The **#else** line (optional)
  - ✓ Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to zero (optional)
  - ✓ The preprocessor **#endif** directive
- #if <constant_expression> <token_string>

  #else <token_string>

  #endif

- The compiler only compiles the <token_string> that follow the #if directive when <constant_expression> evaluates to non-zero.
- Otherwise the compiler skips the <token_string> that follow until it encounters the matching #else or #endif.
- If the expression evaluates to false and there is a matching #else, the <token_string> between the #else and #endif are compiled.
- #if directive can be nested but matching #else and #endif directives must be in the sam efile as the #if.
- C89 states that #ifs and #elifs may be nested at least 8 levels.
- C99 allows at least 63 levels of nesting.
- #ifdef <identifier>

  Controlled text

  #endif

  - This is called a conditional group.
- Controlled text will be included in the output of the preprocessor if and only if the <identifier> is defined.
- #ifdef <identifier>
  - Evaluates to 1 if the symbol specified by <identifier> has been previously defined with a #define directive.
- #ifndef<identifier>
  - Evaluates to 1 if the symbol specified by <identifier> has not been defined.
- C89 states that #ifdef's and #ifndef's may be nested at least 8 levels.
- C99 allows at least 63 levels of nesting.
- #elif – provides alternative test facility.
  - #elif <constant_expression> <token_string>
- If the constant expression evaluates to a nonzero value, the lines of code that immediately follow the condition are passed on to the compiler.
- If the expression evaluates to zero and the conditional compilation directive contains a preprocessor **#elif** directive,
  - the source text located between the **#elif** and the next **#elif** or preprocessor **#else** directive is selected by the preprocessor to be passed on to the compiler.
- The **#elif** directive cannot appear after the preprocessor **#else** directive.

-
  ```
  #if TEST >= 1
          printf("i = %d\n", i);
          printf("array[i] = %d\n", array[i]);
  #elif TEST < 0
  ```

```
        printf("array subscript out of bounds \n");
    #endif
```

Defined

- To determine whether a macro name is defined or not
    - ✓ Use the #if directive with 'defined' compile-time operator.
- defined macro_name
- If macro_name is currently defined the expression is true else it is false.

#pragma

- #pragma directive is used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole.
- A #pragma directive is an instruction to the compiler and is ignored during preprocessing.
- #pragma *pragma-string*
    - *pragma-string* can be one of the following instructions to the compiler with any required parameters.
        - COPYRIGHT -- Specify a copyright string.
        - COPYRIGHT_DATE -- Specify a copyright date for the copyright string.
        - hdr_stop -- When using header caching, specify the end of the prefix header region. In a given source file, this header cannot be reset.
        - HP_SHLIB_VERSION -- Create versions of a shared library routine.
        - LOCALITY -- Name a code subspace.
        - OPTIMIZE -- Turn optimization on or off.
        - OPT_LEVEL -- Set an optimization level.
        - pack -- Allows maximum alignment of class fields having non-class types.
        - VERSIONID -- Specify a version string.

#error directive

- A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.
- A **#error** directive has the form
    - ✓ #error error_message
- The error_message should not be written within the double quotes.
- The **#error** directive is often used in the **#else** portion of a **#if**-**#elif**-**#else** construct, as a safety check during compilation.
- For example, **#error** directives in the source file can prevent code generation if a section of the program is reached that should be bypassed.
- #define BUFFER_SIZE 255

```
#if BUFFER_SIZE < 255

#error "BUFFER_SIZE is too small."

#endif
```

- generates the error message
- BUFFER_SIZE is too small.

#line

- It changes the contents of __LINE__ and __FILE__ that are predefined identifiers in the compiler.
- The __LINE__ identifier contains the line number of the currently compiled line of code.
- The __FILE__ identifier is a string that contains the name of the source file being compiled.
- #line number “filename”
  - Number is any positive integer that becomes the new value of __LINE__
  - Filename is any valid file identifier that become sthe new value of __FILE__.
    - It is optional.
- #line is used for debugging and special applications.

Stringizing Operator

- The number-sign or "stringizing" operator (**#**) converts macro parameters to string literals without expanding the parameter definition.
- It is used only with macros that take arguments.

Token Pasting Operator

- ## enables us to combine two tokens within a macro definition to form a single token.

Questions

- What is a preprocessor? Explain the three different types of preprocessors available in C.
- What are preprocessor directives? Mention and explain the categories of directives with suitable examples.