Chapter 21

Introduction to C Programming Language

Computer Fundamentals - Pradeep K. Sinha & Priti Sinha
Learning Objectives

In this chapter you will learn about:

- Features of C
- Various constructs and their syntax
- Data types and operators in C
- Control and Loop Structures in C
- Functions in C
- Writing programs in C
Features

- Reliable, simple, and easy to use
- Has virtues of high-level programming language with efficiency of assembly language
- Supports user-defined data types
- Supports modular and structured programming concepts
- Supports a rich library of functions
- Supports pointers with pointer operations
- Supports low-level memory and device access
- Small and concise language
- Standardized by several international standards body
# C Character Set

<table>
<thead>
<tr>
<th>Category</th>
<th>Valid Characters</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uppercase alphabets</td>
<td>A, B, C, ..., Z</td>
<td>26</td>
</tr>
<tr>
<td>Lowercase alphabets</td>
<td>a, b, c, ..., z</td>
<td>26</td>
</tr>
<tr>
<td>Digits</td>
<td>0, 1, 2, ..., 9</td>
<td>10</td>
</tr>
<tr>
<td>Special characters</td>
<td>~`!@#$%^&amp;*()_+|{}[]:;&quot;'&lt;&gt;,.? /</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>93</td>
</tr>
</tbody>
</table>
Constants

- Constant is a value that never changes
- Three primitive types of constants supported in C are:
  - Integer
  - Real
  - Character
Rules for Constructing Integer Constants

- Must have at least one digit
- + or – sign is optional
- No special characters (other than + and – sign) are allowed
- Allowable range is:
  - -32768 to 32767 for integer and short integer constants (16 bits storage)
  - -2147483648 to 2147483647 for long integer constants (32 bits storage)
- Examples are: 8, +17, -6
Rules for Constructing Real Constants in Exponential Form

- Has two parts – mantissa and exponent - separated by ‘e’ or ‘E’
- Mantissa part is constructed by the rules for constructing real constants in fractional form
- Exponent part is constructed by the rules for constructing integer constants
- Allowable range is -3.4e38 to 3.4e38
- Examples are: 8.6e5, +4.3E-8, -0.1e+4
Rules for Constructing Character Constants

- Single character from C character set
- Enclosed within single inverted comma (also called single quote) punctuation mark
- Examples are: ‘A’ ‘a’ ‘8’ ‘%’
Variables

- Entity whose value may vary during program execution
- Has a name and type associated with it
- Variable name specifies programmer given name to the memory area allocated to a variable
- Variable type specifies the type of values a variable can contain
- Example: In $i = i + 5$, $i$ is a variable
Rules for Constructing Variables

- Can have 1 to 31 characters
- Only alphabets, digits, and underscore (as in last_name) characters are allowed
- Names are case sensitive (nNum and nNUM are different)
- First character must be an alphabet
- Underscore is the only special character allowed
- Keywords cannot be used as variable names
- Examples are: I saving_2007 ArrSum
# Data Types Used for Variable Type Declaration

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Minimum Storage Allocated</th>
<th>Used for Variables that can contain</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>2 bytes (16 bits)</td>
<td>integer constants in the range -32768 to 32767</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes (16 bits)</td>
<td>integer constants in the range -32768 to 32767</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes (32 bits)</td>
<td>integer constants in the range -2147483648 to 2147483647</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes (32 bits)</td>
<td>real constants with minimum 6 decimal digits precision</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes (64 bits)</td>
<td>real constants with minimum 10 decimal digits precision</td>
</tr>
<tr>
<td>char</td>
<td>1 byte (8 bits)</td>
<td>character constants</td>
</tr>
<tr>
<td>enum</td>
<td>2 bytes (16 bits)</td>
<td>Values in the range -32768 to 32767</td>
</tr>
<tr>
<td>void</td>
<td>No storage allocated</td>
<td>No value assigned</td>
</tr>
</tbody>
</table>
Variable Type Declaration Examples

```c
int count;
short index;
long principle;
float area;
double radius;
char c;
```
## Standard Qualifiers in C

<table>
<thead>
<tr>
<th>Category</th>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifetime</td>
<td>auto</td>
<td>Temporary variable</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>Attempt to store in processor register, fast access</td>
</tr>
<tr>
<td></td>
<td>static</td>
<td>Permanent, initialized</td>
</tr>
<tr>
<td></td>
<td>extern</td>
<td>Permanent, initialized but declaration elsewhere</td>
</tr>
<tr>
<td></td>
<td>const</td>
<td>Cannot be modified once created</td>
</tr>
<tr>
<td></td>
<td>volatile</td>
<td>May be modified by factors outside program</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign</td>
<td>signed</td>
<td>+ or –</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
<td>+ only</td>
</tr>
<tr>
<td>Size</td>
<td>short</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

Ref. Page 399  Chapter 21: Introduction to C Programming Language  Slide 13/65
Lifetime and Visibility Scopes of Variables

- Lifetime of all variables (except those declared as static) is same as that of function or statement block it is declared in.
- Lifetime of variables declared in global scope and static is same as that of the program.
- Variable is visible and accessible in the function or statement block it is declared in.
- Global variables are accessible from anywhere in program.
- Variable name must be unique in its visibility scope.
- Local variable has access precedence over global variable of same name.
Keywords (or reserved words) are predefined words whose meanings are known to C compiler.

- C has 32 keywords
- Keywords cannot be used as variable names

```
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
```
Comments

- Comments are enclosed within `/*` and `*/`.
- Comments are ignored by the compiler.
- Comment can also split over multiple lines.
- Example: `/* This is a comment statement */`
Operators

- Operators in C are categorized into data access, arithmetic, logical, bitwise, and miscellaneous.
- **Associativity** defines the order of evaluation when operators of same precedence appear in an expression.
  - `a = b = c = 15`, `=` has R→L associativity.
  - First `c = 15`, then `b = c`, then `a = b` is evaluated.
- **Precedence** defines the order in which calculations involving two or more operators is performed.
  - `x + y * z`, `*` is performed before `+`.
### Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning with Example</th>
<th>Associativity</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Addition; x + y</td>
<td>L → R</td>
<td>4</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction; x - y</td>
<td>L → R</td>
<td>4</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication; x * y</td>
<td>L → R</td>
<td>3</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division; x / y</td>
<td>L → R</td>
<td>3</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Remainder (or Modulus); x % y</td>
<td>L → R</td>
<td>3</td>
</tr>
<tr>
<td><code>++</code></td>
<td>Increment; x++ means post-increment (increment the value of x by 1 after using its value);;</td>
<td>L → R</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>++x means pre-increment (increment the value of x by 1 before using its value)</td>
<td>R → L</td>
<td>2</td>
</tr>
</tbody>
</table>
## Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning with Example</th>
<th>Associativity</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>Decrement;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x--</td>
<td>x-- means post-decrement (decrement the value of x by 1 after using its value);</td>
<td>L → R</td>
<td>1</td>
</tr>
<tr>
<td>--x</td>
<td>--x means pre-decrement (decrement the value of x by 1 before using its value)</td>
<td>R → L</td>
<td>2</td>
</tr>
<tr>
<td>=</td>
<td>x = y means assign the value of y to x</td>
<td>R → L</td>
<td>14</td>
</tr>
<tr>
<td>+=</td>
<td>x += 5 means x = x + 5</td>
<td>R → L</td>
<td>14</td>
</tr>
<tr>
<td>-=</td>
<td>x -= 5 means x = x - 5</td>
<td>R → L</td>
<td>14</td>
</tr>
<tr>
<td>*=</td>
<td>x * = 5 means x = x * 5</td>
<td>R → L</td>
<td>14</td>
</tr>
<tr>
<td>/=</td>
<td>x /= 5 means x = x / 5</td>
<td>R → L</td>
<td>14</td>
</tr>
<tr>
<td>%=</td>
<td>x %= 5 means x = x % 5</td>
<td>R → L</td>
<td>14</td>
</tr>
</tbody>
</table>
## Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning with Example</th>
<th>Associativity</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Reverse the logical value of a single variable; !x means if the value of x is non-zero, make it zero; and if it is zero, make it one</td>
<td>R → L</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than; x &gt; y</td>
<td>L → R</td>
<td>6</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than; x &lt; y</td>
<td>L → R</td>
<td>6</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to; x &gt;= y</td>
<td>L → R</td>
<td>6</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to; x &lt;= y</td>
<td>L → R</td>
<td>6</td>
</tr>
<tr>
<td>==</td>
<td>Equal to; x == y</td>
<td>L → R</td>
<td>7</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to; x != y</td>
<td>L → R</td>
<td>7</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND; x &amp;&amp; y means both x and y should be true (non-zero) for result to be true</td>
<td>L → R</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR; x</td>
</tr>
<tr>
<td>z?x:y</td>
<td>If z is true (non-zero), then the value returned is x, otherwise the value returned is y</td>
<td>R → L</td>
<td>13</td>
</tr>
</tbody>
</table>
## Bitwise Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning with Example</th>
<th>Associativity</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Complement; ( \sim x ) means All 1s are changed to 0s and 0s to 1s</td>
<td>( R \rightarrow L )</td>
<td>2</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND; ( x &amp; y ) means ( x ) AND ( y )</td>
<td>( L \rightarrow R )</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L → R</td>
<td>10</td>
</tr>
<tr>
<td>^</td>
<td>Exclusive OR; ( x \uparrow y ) means ( x \uparrow y )</td>
<td>( L \rightarrow R )</td>
<td>9</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left shift; ( x \ll 4 ) means shift all bits in ( x ) four places to the left</td>
<td>( L \rightarrow R )</td>
<td>5</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift; ( x \ll 3 ) means shift all bits in ( x ) three places to the right</td>
<td>( L \rightarrow R )</td>
<td>5</td>
</tr>
<tr>
<td>&amp;=</td>
<td>( x &amp;= y ) means ( x = x &amp; y )</td>
<td>( R \rightarrow L )</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>( x</td>
<td>= y ) means ( x = x</td>
</tr>
<tr>
<td>^=</td>
<td>( x ^{= y} ) means ( x = x ^{= y} )</td>
<td>( R \rightarrow L )</td>
<td>14</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>( x \ll 4 ) means shift all bits in ( x ) four places to the left and assign the result to ( x )</td>
<td>( R \rightarrow L )</td>
<td>14</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>( x \ll 3 ) means shift all bits in ( x ) three places to the right and assign the result to ( x )</td>
<td>( R \rightarrow L )</td>
<td>14</td>
</tr>
</tbody>
</table>
# Data Access Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning with Example</th>
<th>Associativity</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[y]</td>
<td>Access $y^{th}$ element of array $x$; $y$ starts from zero and increases monotonically up to one less than declared size of array</td>
<td>L → R</td>
<td>1</td>
</tr>
<tr>
<td>x.y</td>
<td>Access the member variable $y$ of structure $x$</td>
<td>L → R</td>
<td>1</td>
</tr>
<tr>
<td>x -&gt; y</td>
<td>Access the member variable $y$ of structure $x$</td>
<td>L → R</td>
<td>1</td>
</tr>
<tr>
<td>&amp;x</td>
<td>Access the address of variable $x$</td>
<td>R → L</td>
<td>2</td>
</tr>
<tr>
<td>*x</td>
<td>Access the value stored in the storage location (address) pointed to by pointer variable $x$</td>
<td>R → L</td>
<td>2</td>
</tr>
<tr>
<td>Operator</td>
<td>Meaning with Example</td>
<td>Associativity</td>
<td>Precedence</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
<td>---------------</td>
<td>------------</td>
</tr>
<tr>
<td>x(y)</td>
<td>Evaluates function x with argument y</td>
<td>L → R</td>
<td>1</td>
</tr>
<tr>
<td>sizeof (x)</td>
<td>Evaluate the size of variable x in bytes</td>
<td>R → L</td>
<td>2</td>
</tr>
<tr>
<td>sizeof (type)</td>
<td>Evaluate the size of data type “type” in bytes</td>
<td>R → L</td>
<td>2</td>
</tr>
<tr>
<td>(type) x</td>
<td>Return the value of x after converting it from declared data type of variable x to the new data type “type”</td>
<td>R → L</td>
<td>2</td>
</tr>
<tr>
<td>x,y</td>
<td>Sequential operator (x then y)</td>
<td>L → R</td>
<td>15</td>
</tr>
</tbody>
</table>
Statements

- C program is a combination of statements written between { and } braces.
- Each statement performs a set of operations.
- Null statement, represented by ";" or empty {} braces, does not perform any operation.
- A simple statement is terminated by a semicolon ";".
- Compound statements, called statement block, perform complex operations combining null, simple, and other block statements.
Examples of Statements

ß a = (x + y) * 10; /* simple statement */

ß if (sell > cost) /* compound statement
follows */
{
    profit = sell – cost;
    printf ("profit is %d", profit);
}
else /* null statement follows */
{
  
}
Simple I/O Operations

- C has no keywords for I/O operations
- Provides standard library functions for performing all I/O operations
## Basic Library Functions for I/O Operations

<table>
<thead>
<tr>
<th>I/O Library Functions</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>getch()</strong></td>
<td>Inputs a single character (most recently typed) from standard input (usually console).</td>
</tr>
<tr>
<td><strong>getche()</strong></td>
<td>Inputs a single character from console and echoes (displays) it.</td>
</tr>
<tr>
<td><strong>getchar()</strong></td>
<td>Inputs a single character from console and echoes it, but requires <em>Enter</em> key to be typed after the character.</td>
</tr>
<tr>
<td><strong>putchar() or putch()</strong></td>
<td>Outputs a single character on console (screen).</td>
</tr>
<tr>
<td><strong>scanf()</strong></td>
<td>Enables input of formatted data from console (keyboard). Formatted input data means we can specify the data type expected as input. Format specifiers for different data types are given in Figure 21.6.</td>
</tr>
<tr>
<td><strong>printf()</strong></td>
<td>Enables obtaining an output in a form specified by programmer (formatted output). Format specifiers are given in Figure 21.6. Newline character “\n” is used in <em>printf()</em> to get the output split over separate lines.</td>
</tr>
<tr>
<td><strong>gets()</strong></td>
<td>Enables input of a string from keyboard. Spaces are accepted as part of the input string, and the input string is terminated when <em>Enter</em> key is hit. Note that although <em>scanf()</em> enables input of a string of characters, it does not accept multi-word strings (spaces in-between).</td>
</tr>
<tr>
<td><strong>puts()</strong></td>
<td>Enables output of a multi-word string</td>
</tr>
</tbody>
</table>
### Basic Format Specifiers for `scanf()` and `printf()`

<table>
<thead>
<tr>
<th>Format Specifiers</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%d</code></td>
<td>integer (short signed)</td>
</tr>
<tr>
<td><code>%u</code></td>
<td>integer (short unsigned)</td>
</tr>
<tr>
<td><code>%ld</code></td>
<td>integer (long signed)</td>
</tr>
<tr>
<td><code>%lu</code></td>
<td>integer (long unsigned)</td>
</tr>
<tr>
<td><code>%f</code></td>
<td>real (float)</td>
</tr>
<tr>
<td><code>%lf</code></td>
<td>real (double)</td>
</tr>
<tr>
<td><code>%c</code></td>
<td>character</td>
</tr>
<tr>
<td><code>%s</code></td>
<td>string</td>
</tr>
</tbody>
</table>
/ * A portion of C program to illustrate formatted input and output */

int maths, science, english, total;
float percent;

clrscr(); /* A C library function to make the screen clear */
printf ("Maths marks = "); /* Displays “Maths marks = ” */
scanf ("%d", &maths); /* Accepts entered value and stores in variable “maths” */
printf ("\n Science marks = "); /* Displays “Science marks = ” on next line because of \n */
scanf ("%d", &science); /* Accepts entered value and stores in variable “science” */
printf ("\n English marks = "); /* Displays “English marks = ” on next line because of \n */
scanf ("%d", &english); /* Accepts entered value and stores in variable “english” */

total = maths + science + english;
percent = total/3; /* Calculates percentage and stores in variable “percent” */

printf ("\n Percentage marks obtained = %f", percent); /* Displays “Percentage marks obtained = 85.66” on next line because of \n */

(Continued on next slide)
Output:
Maths marks = 92
Science marks = 87
English marks = 78
Percentage marks obtained = 85.66
Preprocessor Directives

- **Preprocessor** is a program that prepares a program for the C compiler.

- Three common preprocessor directives in C are:
  - `#include` – Used to look for a file and place its contents at the location where this preprocessor directives is used.
  - `#define` – Used for macro expansion.
  - `#ifdef..#endif` – Used for conditional compilation of segments of a program.
Examples of Preprocessor Directives

#include <stdio.h>
#define PI 3.1415
#define AND &&
#define ADMIT printf("The candidate can be admitted");

#ifdef WINDOWS
   .
   .
   .
   Code specific to windows operating system
   .
   .
#else
   .
   .
   .
   Code specific to Linux operating system
   .
   .
#endif
   .
   .
   .
   Code common to both operating systems
### Standard Preprocessor Directives in C

<table>
<thead>
<tr>
<th>Preprocessor Directive</th>
<th>Meaning</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#</code></td>
<td>Null directive</td>
<td></td>
</tr>
<tr>
<td><code>#error message</code></td>
<td>Prints <em>message</em> when processed</td>
<td></td>
</tr>
<tr>
<td><code>#line linenum filename</code></td>
<td>Used to update code line number and filename</td>
<td></td>
</tr>
<tr>
<td><code>#pragma name</code></td>
<td>Compiler specific settings</td>
<td></td>
</tr>
<tr>
<td><code>#include filename</code></td>
<td>Includes content of another file</td>
<td>File</td>
</tr>
<tr>
<td><code>#define macro/string</code></td>
<td>Define a macro or string substitution</td>
<td></td>
</tr>
<tr>
<td><code>#undef macro</code></td>
<td>Removes a macro definition</td>
<td>Macro</td>
</tr>
<tr>
<td><code>#if expr</code></td>
<td>Includes following lines if <em>expr</em> is true</td>
<td>Conditional</td>
</tr>
<tr>
<td><code>#elif expr</code></td>
<td>Includes following lines if <em>expr</em> is true</td>
<td></td>
</tr>
<tr>
<td><code>#else</code></td>
<td>Handles otherwise conditions of <code>#if</code></td>
<td></td>
</tr>
<tr>
<td><code>#endif</code></td>
<td>Closes <code>#if</code> or <code>#elif</code> block</td>
<td></td>
</tr>
<tr>
<td><code>#ifdef macro</code></td>
<td>Includes following lines if macro is defined</td>
<td></td>
</tr>
<tr>
<td><code>#ifndef imacro</code></td>
<td>Includes following lines if macro is not defined</td>
<td></td>
</tr>
<tr>
<td><code>#</code></td>
<td>String forming operator</td>
<td>Operators</td>
</tr>
<tr>
<td><code>##</code></td>
<td>Token pasting operator</td>
<td></td>
</tr>
<tr>
<td><code>defined</code></td>
<td>same as <code>#ifdef</code></td>
<td></td>
</tr>
</tbody>
</table>
Pointers

- C pointers allow programmers to directly access memory addresses where variables are stored.

- Pointer variable is declared by adding a ‘*’ symbol before the variable name while declaring it.

- If \( p \) is a pointer to a variable (e.g. `int i, *p = i;`)
  - Using \( p \) means address of the storage location of the pointed variable.
  - Using \( *p \) means value stored in the storage location of the pointed variable.

- Operator ‘&’ is used with a variable to mean variable’s address, e.g. `&i` gives address of variable `i`.
Illustrating Pointers Concept

Address of i = 1000
Value of i = 62

int i = 62;
int *p;
int j;
p = &i; /* p becomes 1000 */
j = *p; /* j becomes 62 */
j = 0; /* j becomes zero */
j = *(&i) /* j becomes 62 */
Array

- Collection of fixed number of elements in which all elements are of the same data type
- Homogeneous, linear, and contiguous memory structure
- Elements can be referred to by using their subscript or index position that is monotonic in nature
- First element is always denoted by subscript value of 0 (zero), increasing monotonically up to one less than declared size of array
- Before using an array, its type and dimension must be declared
- Can also be declared as multi-dimensional such as Matrix2D[10][10]
### Illustrating Arrays Concept

<table>
<thead>
<tr>
<th>int marks[6];</th>
<th>float price[4];</th>
<th>char city[6];</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each element being an int occupies 2 bytes</td>
<td>Each element being a float occupies 4 bytes</td>
<td>Each element being a char occupies 1 byte</td>
</tr>
<tr>
<td>marks[0] = 45</td>
<td>price[0] = 82.75</td>
<td>city[0] = ‘B’</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>(a) An array of integers having 6 elements</td>
<td>(b) An array of real numbers having 4 elements</td>
<td>(c) An array of characters having 6 elements</td>
</tr>
</tbody>
</table>
String

- One-dimensional array of characters terminated by a null character (‘\0’)
- Initialized at declaration as
  - `char name[] = "PRADEEP";`
- Individual elements can be accessed in the same way as we access array elements such as `name[3] = 'D'`
- Strings are used for text processing
- C provides a rich set of string handling library functions
### Library Functions for String Handling

<table>
<thead>
<tr>
<th>Library Function</th>
<th>Used To</th>
</tr>
</thead>
<tbody>
<tr>
<td>strlen</td>
<td>Obtain the length of a string</td>
</tr>
<tr>
<td>strlwr</td>
<td>Convert all characters of a string to lowercase</td>
</tr>
<tr>
<td>strupr</td>
<td>Convert all characters of a string to uppercase</td>
</tr>
<tr>
<td>strcat</td>
<td>Concatenate (append) one string at the end of another</td>
</tr>
<tr>
<td>strncat</td>
<td>Concatenate only first n characters of a string at the end of another</td>
</tr>
<tr>
<td>strncpy</td>
<td>Copy only the first n characters of a string into another</td>
</tr>
<tr>
<td>strncpy</td>
<td>Copy a string into another</td>
</tr>
<tr>
<td>strcpyp</td>
<td>Compare two strings</td>
</tr>
<tr>
<td>strncmp</td>
<td>Compare only first n characters of two strings</td>
</tr>
<tr>
<td>stricmp</td>
<td>Compare two strings without regard to case</td>
</tr>
<tr>
<td>strnicmp</td>
<td>Compare only first n characters of two strings without regard to case</td>
</tr>
<tr>
<td>strdup</td>
<td>Duplicate a string</td>
</tr>
<tr>
<td>strchr</td>
<td>Find first occurrence of a given character in a string</td>
</tr>
<tr>
<td>strrchr</td>
<td>Find last occurrence of a given character in a string</td>
</tr>
<tr>
<td>strstr</td>
<td>Find first occurrence of a given string in another string</td>
</tr>
<tr>
<td>strset</td>
<td>Set all characters of a string to a given character</td>
</tr>
<tr>
<td>strnset</td>
<td>Set first n characters of a string to a given character</td>
</tr>
<tr>
<td>strrev</td>
<td>Reverse a string</td>
</tr>
</tbody>
</table>
User Defined Data Types (UDTs)

- UDT is composite data type whose composition is not include in language specification
- Programmer declares them in a program where they are used
- Two types of UDTs are:
  - Structure
  - Union
Structure

- UDT containing a number of data types grouped together
- Constituents data types may or may not be of different types
- Has continuous memory allocation and its minimum size is the sum of sizes of its constituent data types
- All elements (member variable) of a structure are publicly accessible
- Each member variable can be accessed using “.” (dot) operator or pointer (*EmpRecord.EmpID or EmpRecord → EmpID*)
- Can have a pointer member variable of its own type, which is useful in creating linked list and similar data structures
struct Employee
{
    int EmpID;
    char EmpName[20];
};

Struct Employee EmpRecord;
Struct Employee *pEmpRecord = &EmpRecord

struct Employee
{
    int EmpID;
    char EmpName[20];
} EmpRecord;
Union

- UDT referring to same memory location using several data types
- Mathematical union of all constituent data types
- Each data member begins at the same memory location
- Minimum size of a union variable is the size of its largest constituent data types
- Each member variable can be accessed using “,” (dot) operator
- Section of memory can be treated as a variable of one type on one occasion, and of another type on another occasion
unionNum
{
  int intNum;
  unsigned unsNum'
};
union Num Number;
Difference Between Structure and Union

- Both group a number of data types together
- Structure allocates different memory space contiguously to different data types in the group
- Union allocates the same memory space to different data types in the group
Control Structures

- *Control structures* (branch statements) are decision points that control the flow of program execution based on:
  - Some condition test (conditional branch)
  - Without condition test (unconditional branch)
- Ensure execution of other statement/block or cause skipping of some statement/block
Conditional Branch Statements

- **if** is used to implement simple one-way test. It can be in one of the following forms:
  - if..stmt
  - if..stmt1..else..stmt2
  - if..stmt1..else..if..stmtn

- **switch** facilitates multi-way condition test and is very similar to the third *if* construct when primary test object remains same across all condition tests.
Examples of “if” Construct

- if (i <= 0)  
  i++;  
- if (i <= 0)  
  i++;  
  else  
  j++;  
- if (i <= 0)  
  i++;  
  else if (i >= 0)  
  j++;  
  else  
  k++;
Example of “switch” Construct

switch(ch)
{
    case 'A':
    case 'B':
    case 'C':
        printf("Capital");
        break;
    case 'a':
    case 'b':
    case 'c':
        printf("Small");
        break;
    default:
        printf("Not cap or small");
}

Same thing can be written also using if construct as:

    if (ch == 'A' || ch == 'B' || ch == 'C')
        printf("Capital");
    else if (ch == 'a' || ch == 'b' || ch == 'c')
        printf("Small");
    else
        printf("Not cap or small");
Unconditional Branch Statements

- **Break**: Causes unconditional exit from `for`, `while`, `do`, or `switch` constructs. Control is transferred to the statement immediately outside the block in which `break` appears.

- **Continue**: Causes unconditional transfer to next iteration in a `for`, `while`, or `do` construct. Control is transferred to the statement beginning the block in which `continue` appears.

- **Goto label**: Causes unconditional transfer to statement marked with the label within the function.

(Continued on next slide)
Return [value/variable]: Causes immediate termination of function in which it appears and transfers control to the statement that called the function. Optionally, it provides a value compatible to the function’s return data type.
Loop Structures

- Loop statements are used to repeat the execution of statement or blocks
- Two types of loop structures are:
  - Pretest: Condition is tested before each iteration to check if loop should occur
  - Posttest: Condition is tested after each iteration to check if loop should continue (at least, a single iteration occurs)
Pretest Loop Structures

**for:** It has three parts:
- **Initializer** is executed at start of loop
- **Loop condition** is tested before iteration to decide whether to continue or terminate the loop
- **Incrementor** is executed at the end of each iteration

**While:** It has a loop condition only that is tested before each iteration to decide whether to continue to terminate the loop
Examples of “for” and “while” Constructs

- for (i=0; i < 10; i++)
  printf("i = %d", i);

- while (i < 10)
  {
    printf("i = %d", i);
    i++;
  }
Posttest Loop Construct
“do...while”

- It has a loop condition only that is tested after each iteration to decide whether to continue with next iteration or terminate the loop.

- Example of do...while is:

```c
    do {
        printf("i = %d", i);
        i++;
    } while (i < 10);
```
Functions

- Functions (or subprograms) are building blocks of a program
- All functions must be declared and defined before use
- Function declaration requires `functionname, argument list, and return type`
- Function definition requires coding the body or logic of function
- Every C program must have a `main` function. It is the entry point of the program
Example of a Function

```c
int myfunc ( int Val, int ModVal )
{
    unsigned temp;
    temp = Val % ModVal;
    return temp;
}
```

This function can be called from any other place using simple statement:

```c
int n = myfunc(4, 2);
```
Sample C Program (Program-1)

/* Program to accept an integer from console and to display whether the number is even or odd */

#include <stdio.h>

void main()
{
    int number, remainder;
    clrscr(); /* clears the console screen */
    printf("Enter an integer: ");
    scanf("%d", &number);
    remainder = number % 2;
    if (remainder == 0)
        printf("\n   The given number is even");
    else
        printf("\n   The given number is odd");

    getch();
}
/* Program to accept an integer in the range 1 to 7 (both inclusive) from console and to display the corresponding day (Monday for 1, Tuesday for 2, Wednesday for 3, and so on). If the entered number is out of range, the program displays a message saying that */

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

#define MON printf("\n Entered number is 1 hence day is MONDAY");
#define TUE printf("\n Entered number is 2 hence day is TUESDAY");
#define WED printf("\n Entered number is 3 hence day is WEDNESDAY");
#define THU printf("\n Entered number is 4 hence day is THURSDAY");
#define FRI printf("\n Entered number is 5 hence day is FRIDAY");
#define SAT printf("\n Entered number is 6 hence day is SATURDAY");
#define SUN printf("\n Entered number is 7 hence day is SUNDAY");
#define OTH printf("\n Entered number is out of range");

void main()
{
    int day;
    clrscr();
    printf("Enter an integer in the range 1 to 7");
    scanf("%d", &day);
    switch(day)
    {
      case 1: MON; break;
      case 2: TUE; break;
      case 3: WED; break;
      case 4: THU; break;
      case 5: FRI; break;
      case 6: SAT; break;
      case 7: SUN; break;
      default: OTH; break;
    }
}
Sample C Program (Program-2)

(Continued from previous slide..)

```c
{
    Case 1:
        MON;
        break;
    Case 2:
        TUE;
        break;
    Case 3:
        WED;
        break;
    Case 4:
        THU;
        break;
    Case 5:
        FRI;
        break;
    Case 6:
        SAT;
        break;
    Case 7:
        SUN;
        break;
    default:
        OTH;
}
getch();
```
Sample C Program (Program-3)

/* Program to accept the radius of a circle from console and to calculate and display its area and circumference */

#include <stdio.h>
#include <conio.h>
define PI 3.1415

void main()
{
    float radius, area, circum;
    clrscr();
    printf("Enter the radius of the circle: ");
    scanf("%f", &radius);
    area = PI * radius * radius;
    circum = 2 * PI * radius;
    printf("\n Area and circumference of the circle are %f and %f respectively", area, circum);
    getch();
}
Sample C Program (Program-4)

/* Program to accept a string from console and to display the number of vowels in the string */

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

void main()
{
    char input_string[50]; /* maximum 50 characters */
    int len;
    int i = 0, cnt = 0;
    clrscr();
    printf("Enter a string of less than 50 characters: \n");
    gets(input_string);
    len = strlen(input_string);
    for (i = 0; i < len; i++)
    {
        switch (input_string[i])
        {
        
        
        
        
        
        
        
        
        (Continued on next slide)
Sample C Program (Program-4)

```c
{ 
    case 'a': 
    case 'e': 
    case 'i': 
    case 'o': 
    case 'u': 
    case 'A': 
    case 'E': 
    case 'I': 
    case 'O': 
    case 'U':
        cnt++
    }
}
printf("\n Number of vowels in the string are: %d", cnt);
getch();
}
/* Program to illustrate use of a user defined function. The program initializes an array of \( n \) elements from 0 to \( n-1 \) and then calculates and prints the sum of the array elements. In this example \( n = 10 \). */

#include <stdio.h>
#define SIZE 10

int ArrSum(int *p, int n);
{
    int s, tot = 0;
    for(s = 0; s < n; s++)
    {
        tot += *p;
        p++;
    }
    return tot;
}

int main()
{
    int i = 0, sum = 0;
    int nArr[SIZE] = {0};
    while(i < SIZE)
    {
        nArr[i] = i;
        i++
    }
    sum = ArrSum(nArr, SIZE);
    printf("Sum of 0 to 9 = %d\n", sum);
    return 0;
}
## Key Words/Phrases

- Arithmetic operators
- Arrays
- Assignment operators
- Bit-level manipulation
- Bitwise operators
- Branch statement
- Character set
- Comment statement
- Compound statement
- Conditional branch
- Conditional compilation
- Constants
- Control structures
- Format specifiers
- Formatted I/O
- Function
- Keywords
- Library functions
- Logical operators
- Loop structures
- Macro expansion
- Main function
- Member element
- Null statement
- Operator associativity
- Operator precedence
- Pointer
- Posttest loop
- Preprocessor directives
- Pretest loop
- Primitive data types
- Reserved words
- Simple statement
- Statement block
- Strings
- Structure data type
- Unconditional branch
- Union data type
- User-defined data types
- Variable name
- Variable type declaration
- Variables