C Programming

Lesson 1: Introduction to C

Lesson Objectives

- **To understand the following topics:**
  - Evolution of C
  - Escape Sequences and Format Specifiers
  - Common Best Practices

Stages in Evolution

- C supports development of all type of applications using one high-level language

Language Developed	Drawbacks
ALGOL 60	- Too general and too abstract
CPL ↓	- Hard to learn and difficult to implement
BCPL ↓	- Less powerful and too specific
B ↓	- Too specific
C ↓	- Final Solution

Character Data Type

- Machine representation:
  - character constant 1 word
  - character variable 1 byte
- Character constants:
  - single character enclosed in single quotes

- For example:

```
'D', '3', '?', '* '
```

- Some special character constants or escape sequences:

```
\n      \v      \r      \"
  \b      \\      \t
```

Character Data Type

Declaration

```
char c;
char response, answer ... ;
char var1, var2, .. , varN ;
```

Character Input / Output

Functions provided - getchar() - putchar(c)

where C_value is of type char/int

char C_value ; or int C_value ;

C_value = getchar() ; or C_value = getchar()

getchar() returns ASCII values from range 0 to 255

Data Types and their Ranges

Data type	Range in environment		Usage
	16 bit	32 bit	
char	-128 to 127	-128 to 127	A single byte capable of holding one character
short int	-215 to 215-1	-231 to 231-1	An integer, short range
int	-215 to 215-1	-231 to 231-1	An integer
long int	-231 to 231-1	-231 to 231-1	An integer, long range
float	-3.4e38 to +3.4e38 (4 bytes)		Single-precision floating point
double	-1.7e308 to +1.7e308 (8 bytes)		Double-precision floating point
unsigned int	0 to 216-1	0 to 232-1	Only positive integers
unsigned char	0 to 255	0 to 255	Only positive byte values

Escape Characters

- Escape characters are:
  - Non-graphic characters including white spaces
  - Non-printing characters and are represented by escape sequences consisting of a backslash (\) followed by a letter

Character	Description
\b	Backspace
\n	New line
\a	Beep
\t	Tab
\”	”
\\	\
\’	’
\r	Carriage return

Format Specifiers

- **Format Specifiers are:**
  - Formatting characters used to accept or display the value in a specific format

Data Type	Conversion Specifier
signed char	%c
unsigned char	%c
short signed int	%d
short unsigned int	%u
long signed int	%ld
long unsigned int	%lu
float	%f
double	%lf
long double	%Lf

Lab

- Lab 1

Indentation-whitespaces

- **Whitespace:**
  - Use vertical and horizontal whitespace generously
  - Indentation and spacing should reflect the block structure of the code
  - A long string of conditional operators should be split onto separate lines

Indentation-whitespaces (contd..)

- For example:

```
if (foo->next==NULL && number < limit  && limit <=SIZE    &&
node_active(this_input)) {...
  might be better as:
```

```
if (foo->next == NULL
 && number < limit && limit <= SIZE
 && node_active(this_input))      {       ...
```

Indentation-Loops

- Loops:
  - Elaborate for loops should be split onto different lines:

```
for (curr = *varp, trail = varp;
    curr != NULL;
    trail = &(curr->next), curr = curr->next )
{
                    ...
}
```

Indentation-Expressions

- Complex expressions, such as those using the ternary ?: operator, are best split on to several lines, too

```
z = (x == y)
? n + f(x)
: f(y) - n;
```

Indentation-Comments

- **Comments:**
  - The comments should describe what is happening, how it is being done, what parameters mean, which globals are used and any restrictions or bugs
  - Avoid unnecessary comments
  - comments are not checked by the compiler, there is no guarantee they are right
  - Too many comments clutter code
  - Nesting of comments produce unpredictable result
    - /* / */ 2 * /* */ 1 is evaluated as 2*1 =2

Indentation-Comments (Contd..)

- Here is a superfluous comment style:
  - i=i+1; /* Add one to i */

- It's pretty clear that the variable i is being incremented by one. And there are worse ways to do it:

```
/**********************************
 *                                *
*       Add one to i             *
*                                *                    **********************************/
```

- i=i+1;

Constants

- Constants:
  - Avoid using direct constants in execution statements
  - Instead, use sizeof, or a #define or an enum for the same
  - Symbolic constants make code easier to read. Numerical constants should generally be avoided
  - Even simple values like 1 or 0 are often better expressed using defines like TRUE and FALSE

Constants

- Bad Example:

```
main()
{
char buf[512];                    /* hard coding */
if((n=read(stdin,buf,512) > 0)/* difficult to modify later */
{…}}
```

- Good Example

```
#define BUFSIZE 512
main()
{
char buf[BUFSIZE];                          /* better */
if((n=read(stdin,buf, BUFSIZE) > 0)/* easy to modify later */
{…}}
```

Constants (contd..)

- Data type of constant will be decided by compiler if its data type is not there in the declaration ,so be careful while using constants in expressions

- For example:
  - x = 20.25 + 10
    - 20.25,10 acting as constants in this expression but we haven't declare the data type for the two. so the compiler will decide the data type of two that is by default it will assign double data type to 20.25 and integer to 10

Constants (contd..)

- **const qualified values in initializes cannot be used for array dimensions, as in:**
  - const const int n = 5;
  - int a[n];
    - The const qualifier really means ‘read-only’; an object so qualified is a run-time object that can no be assigned to. The value of a const qualified object is therefore not a constant expression in the full sense of the term and cannot be used for array dimensions, case labels. When you need a true compile time constant use a preprocessor #define

Variables

- Variable names:
  - When choosing a variable name, length is not important but clarity of expression is
  - A long name can be used for a global variable which is rarely used
  - For an array index used on every line of a loop need not be named any more elaborately than I
  - Using “index” or “elementnumber” instead is not only more to type but also lead more mistakes

Variables (Contd..)

- Consider:

Bad Example

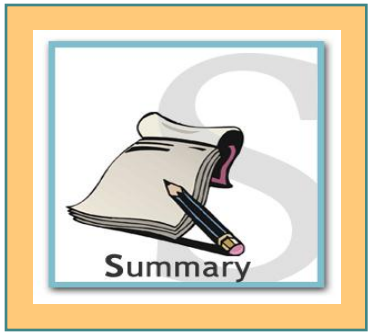
```
for(i=0;i<=100;i++)
        array[i]=0
```

Good Example

```
for(elementnumber=0;
elementnumber<=100;elementnumber++)
array[elementnumber]=0;
```

Summary

- C supports development of all type of applications using one high-level language.
- Variable names are the names (labels) given to the memory location where different constants are stored.
- Expressions can contain constants, variable or function calls.
- Preprocessor directive statements begin with a # symbol.



Review Question

- Question 1: Character occupies _____ bytes.
- Question 2: _____ converts source code to object code.
- Question 3: String constants ends with _____ special character.
- Question 4: Object code is output of _____.

Review Question: Match the Following

1. File inclusion directives	Links the object codes to form a single Executable Code.
2. getchar()	Replace tokens in the current file.
3. Macro definition directives	Embed files within the current file.
4. Link Editor	Returns ASCII value range from 0 to 255