ME 171
Computer Programming Language

Partha Kumar Das
Lecturer,
Department of Mechanical Engineering, BUET

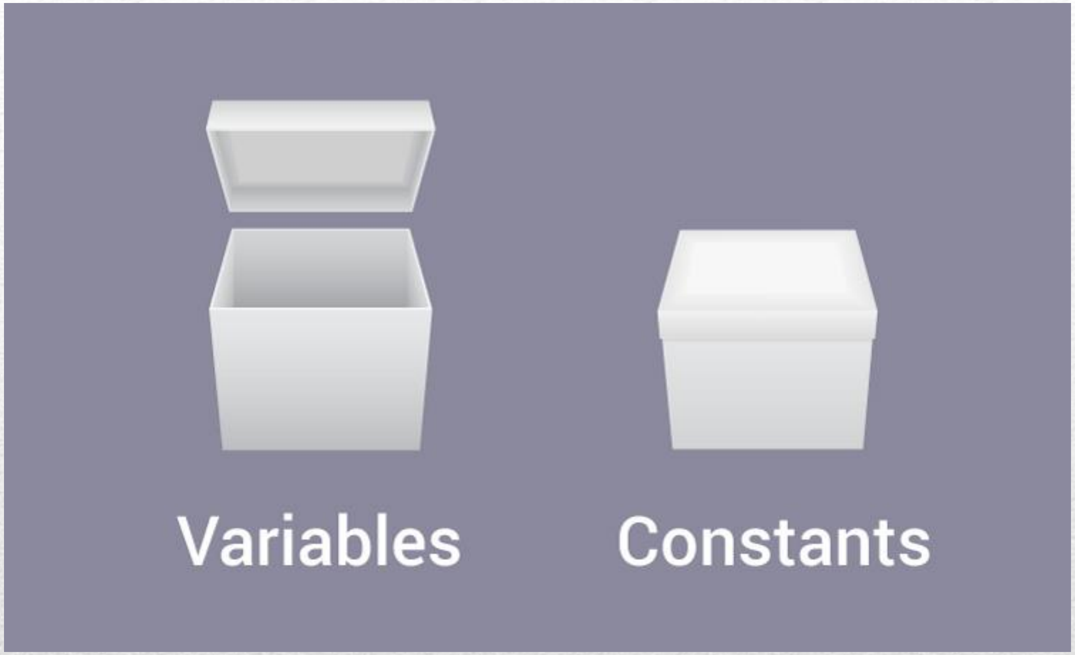
Lecture 3

Variables, Data Types, I/O

1. Variables and Constants
2. Data Types
3. Program Input and Output

Program Variables

- In programming, a variable is a container (storage area) to hold data.
- In real world you have used various type containers for specific purpose.
- For example you have used suitcase to store clothes, match box to store match sticks etc.
- In the same way variables of different data type is used to store different types of data.
- For example integer variables are used to store integers, char variables are used to store characters etc.

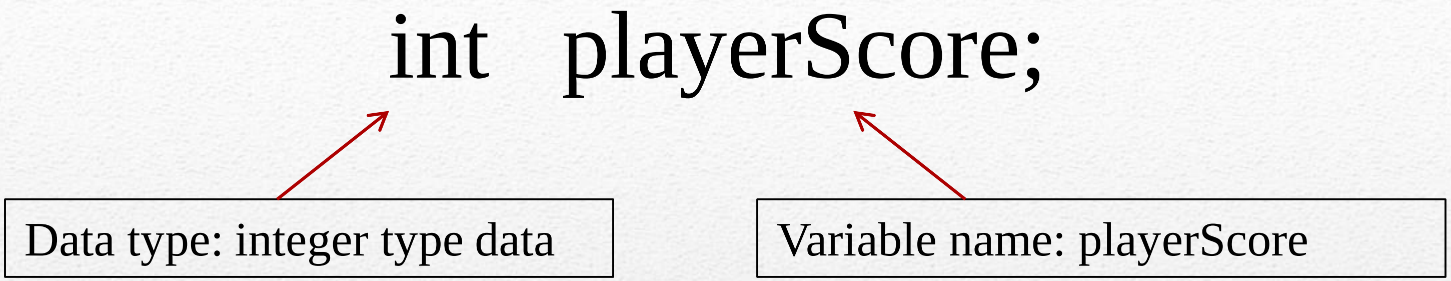


➢ So in C, every variable has two most fundamental attributes:

1. **Data Type:** Which types of data is to be used (int, char, float, double, etc).
2. **Variable Name**: Which name (identifier) is to be used to address and identify the variable in the code.

data_type *(space)*variable_name

Program Variables



int playerScore;

Data type: integer type data

Variable name: playerScore

Naming of Variables

- As variable name is an identifier, it follows the same rules of naming an identifier.
- Most important property of a variables name is its uniqueness. Not two variables in C can have the same name with same visibility. For example:

```
#include<stdio.h>
int main(){
   int a=5;   //Visibility is within main block
   int a=10; //Visibility is within main block
/* Two variables of same name */
   printf("%d",a);
   return 0;}
```

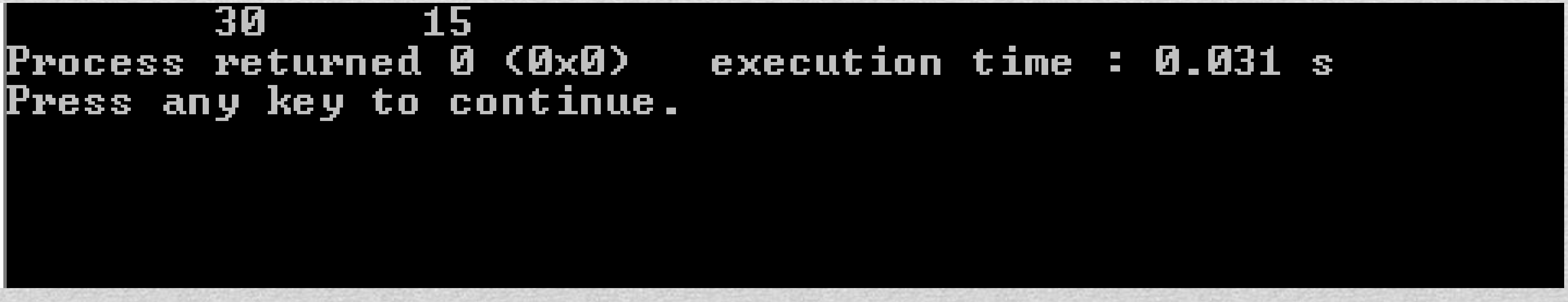
Output: Build Error (redefinition of ‘a’)

```
#include<stdio.h>
int a=5; //Visibility is within the whole
            program
int main(){
   int a=10; //Visibility within main block
   printf("%d",a);
   return 0;}
```

Output: 10

Naming of Variables

```
#include<stdio.h>

int main(){
    int a=10;                           //Visibility within main block.
    {
        a+=5;                           //Accessing outer local variable a.
        int a=20;                       //Visibility within inner block.
        a+=10;                          //Accessing inner local variable a.
        printf("\t%d\t",a);             //Accessing inner local variable a.
    }
    printf("%d",a);                     //Accessing outer local variable a.
    return 0;
}
```



```
        30      15
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Variable Declaration and Definition

- Declaration of variables means to acknowledge the compiler only about variable name and its data type with its modifiers **but compiler doesn’t reserve any memory for the variables.**
- In c we can declare any variable with the help of **extern** keyword while it has not initialized. Example of declaration:

```
extern int a;    // Declaration of variable a
```

- C statement in which a variable gets a memory is known as definition of variable.

```
int a;              //Definition of variable a
static int a;       //Definition of variable a
register int a;     //Definition of variable a
extern int a=5;     //Definition (Declaration plus Initialization) of variable a
```

- In the above c statement all variables has been declared and defined at the same time.
- If any variable has not been declared then it declaration occurs at the time of definition.

Self Study: Surf the internet for static int, register int, auto int, ……(Storage class)

Variable Declaration and Definition

- Since declaration variable doesn’t get any memory space so we cannot assign any value to variable. For example:

```
#include<stdio.h>
extern int a;
int main(){
    a=100;
    printf("%d",a);
    return 0;}
```

Output: Build Error

- We can declare any variable either globally or locally.
- A same variable can be declared many times.

```
#include<stdio.h>
extern int a; //Declaration of variable a
extern int a; //Again declaration of variable a
int a=5;    //Definition of variable a (global variable)
int main(){
    printf("%d",a);
    return 0;}
```

Output: 5

Variable Declaration and Definition

- **Variable declaration or definition must appear first in the main function.**

```
#include<stdio.h>
int main(){
    int a=100;
    printf(“%d”,a);
    int b;
    b=a++;
    printf(“%d”,a)
    return 0;
}
```

WRONG

```
#include<stdio.h>
int main(){
    int a=100,b;
    printf(“%d”,a);
    b=a++;
    printf(“%d”,a)
    return 0;
}
```

CORRECT

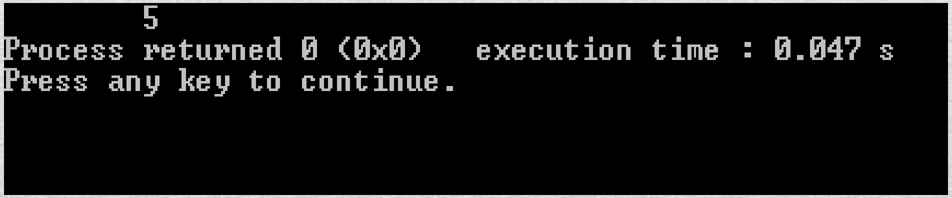
N.B.: Early C compiler needs all local variable definitions before the actual code of function starts (to generate right stack pointer calculation). This was the only way of declaring local variables in early C language, both pre-standard (K&R) and first C standard, C90, published at 1989-1990 (ANSI X3.159-1989, ISO/IEC 9899:1990).

But C99 - the 1999 year ISO standard (ISO/IEC 9899:1999) of C allows declarations in the middle of function.

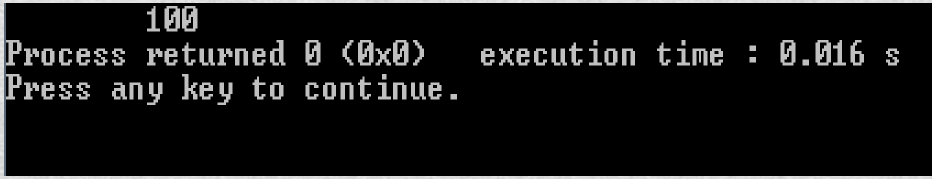
Variable Declaration and Definition

```
#include<stdio.h>
extern int a=5;
int main()
{
    printf("\t%d",a);
    return 0;
}
```

```
#include<stdio.h>
extern int a=5;
int main()
{
    a=100;
    printf("\t%d",a);
    return 0;
}
```



```
        5
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```



```
        100
Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

- Assigning a value against a variable for the first time is known as variable initialization.

```
a=5;   // variable initialization
```

Program Constants

- The value of a constant can not be changed in entire code.
- Two methods of defining a constant in C:

  1. using # define
  2. using const keyword

1. Using #define

- Should be defined before main function.
- No need for specification of data type of the constant.
- Its not a statement, so no semicolon (;) should be put.

NOTE: Any line in C that ends with a semicolon is called a statement.

2. Using const keyword

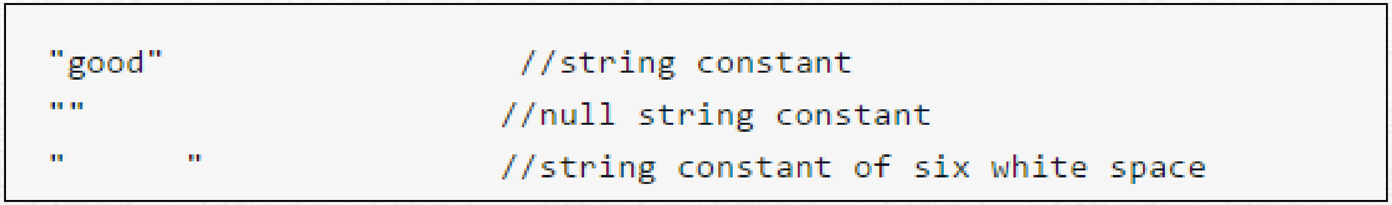
- Can be defined both globally or locally..
- Definition is similar to variable definition with const keyword before the data type.

```
#define TRUE   1
#define FALSE   0
#define Grade   'A'   //Character Constant
#define PI   3.1416
#define CONST   "String Constant"   // String Constant
```

const int id=44;
const float PI = 3.1416;
~~const int grade;~~   // no value is assigned

Program Constants

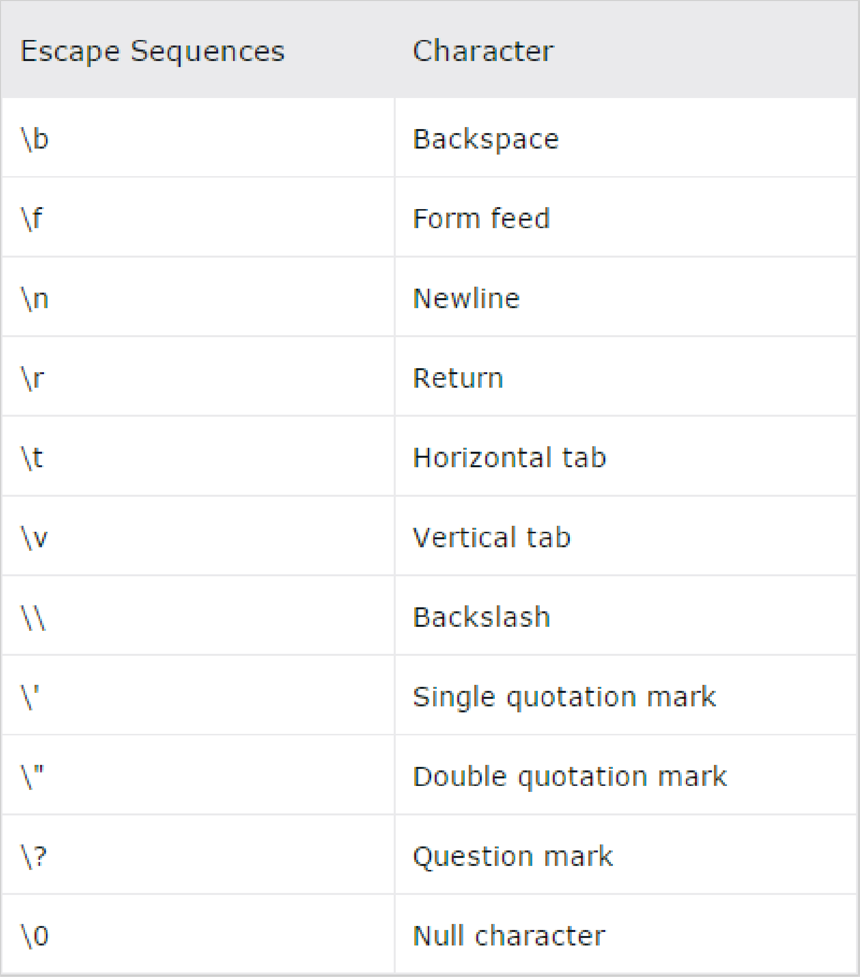
String Constants



```
"good"                  //string constant
""                      //null string constant
"      "                //string constant of six white space
```

Escape Sequence

- Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming.
- For example: newline(enter), tab, question mark etc.
- In order to use these characters, escape sequence is used.
- For example: \n is used for newline.
- The backslash (\ ) causes "escape" from the normal way the characters are interpreted by the compiler

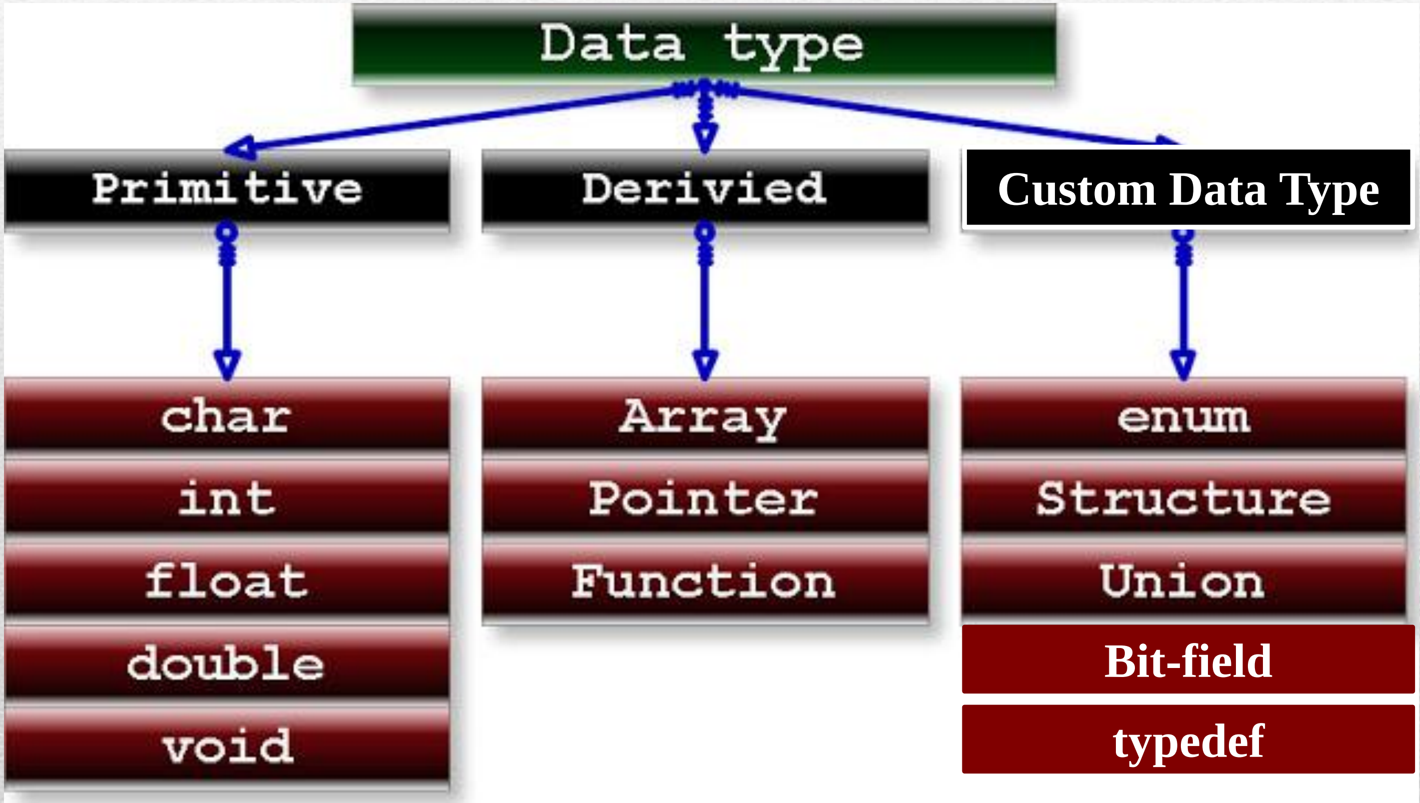


Escape Sequences	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null character

1. Variables and Constants
2. Data Types
3. Program Input and Output

Data Types

- Data types in c refer to an extensive system used for declaring variables or functions of different types.
- **The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.**



Storage Size of Data Types

- The char data type is usually 1 byte, it is so called because they are commonly used to store single characters.
- The size of the other types is dependent on the hardware of your computer.
- On "32-bit" machines the int data type takes up 4 bytes (2^{32}).
- The short is usually smaller, the long can be larger or the same size as an int and finally the long long is for handling very large numbers.

sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)

Type	Storage size	Value range
char	1 byte	-128 to 1 27 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long int	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

NOTE: **signed, unsigned, long, short** are known as modifiers.

Storage Size of Data Types

```
int main()
{
  printf("sizeof(char) == %d\n", sizeof(char));
  printf("sizeof(short) == %d\n", sizeof(short));
  printf("sizeof(int) == %d\n", sizeof(int));
  printf("sizeof(long) == %d\n", sizeof(long));
  printf("sizeof(long long) == %d\n",
sizeof(long long));


  return 0;
}
```

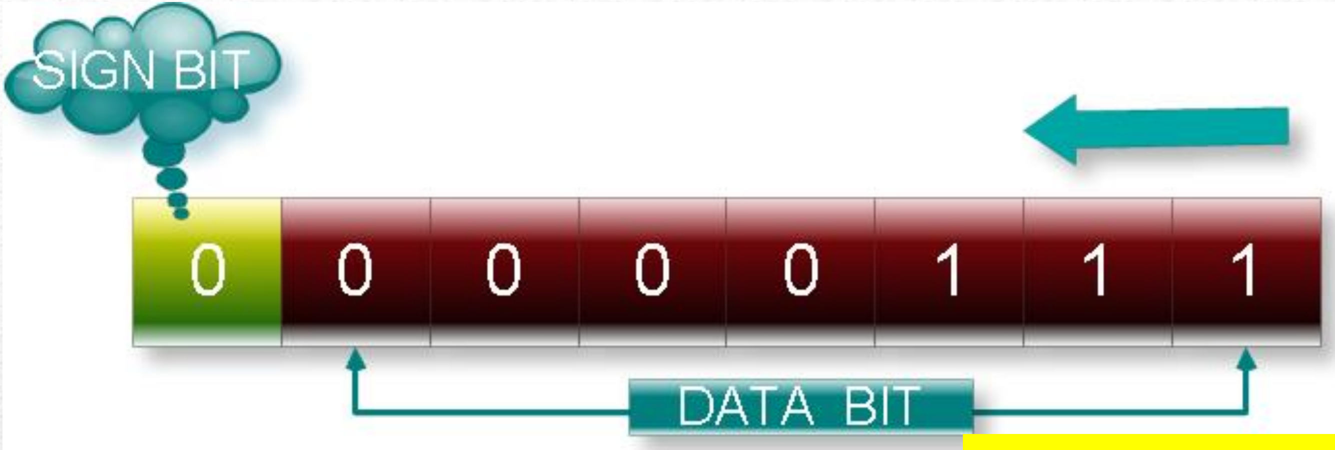
Type	Storage size	Value range
char	1 byte	-128 to 1 27 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long int	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

- ❖ What is the output if you are using 64 bit compiler??

Pass your Leisure

Google “**Data Size Neutrality and 64-bit Support”.**

Why Range of signed char is -128 to 127 not -127 to 128??



127= 0 1 1 1 1 1 1 1

1= 0 0 0 0 0 0 0 1

(Addition) 128= **1 0 0 0 0 0 0 0**

Sign bit Data bit

So in Machine, 10000000 = -128

- This method is known as **2's complement**.
- Almost all modern computers use this representation .

- **How is -127 read in machine?**

  -128 + 1 = -127

  1 0 0 0 0 0 0 0 (-128)

  0 0 0 0 0 0 0 1 (1)

  1 0 0 0 0 0 0 1 (-127)

- 2's Complement

  0 1 1 1 1 1 1 1

  //8-bit binary for absolute value of -127

  1 0 0 0 0 0 0 0

  // all bits flipped

  1 0 0 0 0 0 0 1

  // 1 added to the complement

- So, -127 = 1 0 0 0 0 0 0 1

Storage Size of Data Types

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte (or 12 byte)	3.4E-4932 to 1.1E+4932	19 decimal places

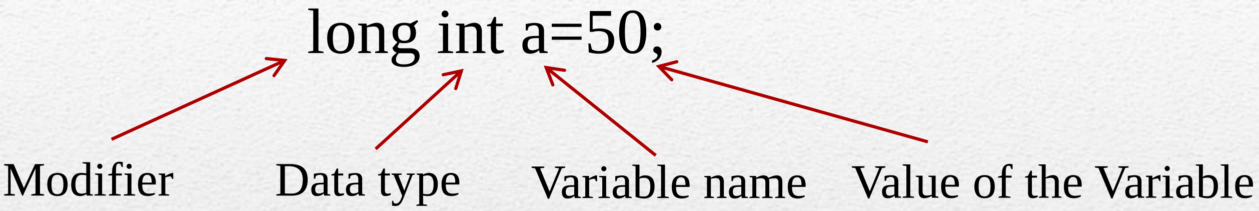
```
#include <stdio.h>
#include <float.h>

int main() {

  printf("Storage size for float : %d \n", sizeof(float));
  printf("Minimum float positive value: %E\n", FLT_MIN );
  printf("Maximum float positive value: %E\n", FLT_MAX );
  printf("Precision value: %d\n", FLT_DIG );

  return 0;
}
```

Do Try At Home

Question: Explain the memory consumption, storage size and value range of char, int and float type data.

Modifiers

- Modifiers are used to modify the storage size and data range of a variable.
- Should be written before or after the data type of a variable.



long int a=50;

Modifier

Data type

Variable name

Value of the Variable

- Modifier without a data type assumes integer type data.

long a=25;
It is equivalent to: **long int** a=25;
signed a=25;
It is equivalent to: **signed int** a=25;

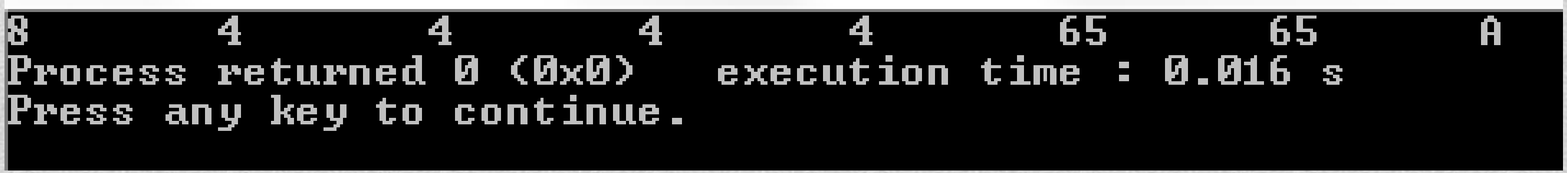
- We cannot use two modifiers of **same type of modification** in any particular data type of c.

~~**short long int** i;~~
~~**static auto char** c;~~
~~**signed unsigned int** array[5];~~

✓ **signed long int** i;

Review

```
#include<stdio.h>
int main(){
    char a='A';float b=6.0;
    printf("%d\t",sizeof(6.0));          =8
    printf("%d\t",sizeof(b));            =4
    printf("%d",sizeof(90000));          =4
    printf("\t%d\t",sizeof(int));        =4
    printf("%d\t",sizeof(long));         =4
    printf("%d\t",'A');                  =65
    printf("%d\t",a);                    =65
    printf("%c",a);                      =A  // this can also be found by
    return 0;                                          printf("%c", 'A');
}
```



```
8       4       4       4       4       65      65      A
Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

CHECK   printf("%d\t",sizeof(long long));

1. Variables and Constants
2. Data Types
3. Program Input and Output

Input and Output in C

- Interactions with environment is not so easy for a programming language.
- There are several library functions for taking inputs and showing outputs from user.
- Input and output can be provided as simple letter, text, or in the form of file.
- The simplest input mechanism is to read a character at a time.
- Some common input and output functions are:
  getch(), getche(), getchar(), putchar()
  gets(), puts()
  scanf(), printf()
  getc(),putc(), fgets(), fputs(), fscanf(), fprintf(). // for data file input and output

- C programming treats all the devices as files.
- So devices such as the display are addressed in the same way as files
- The following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

- The file pointers are the means to access the file for reading and writing purpose.

getchar() and putchar()

int getchar(void)

- Reads the next available **single character at a time** from the standard input (keyboard) and returns it as an integer.

int putchar(int)

- Shows the passed **single character at a time** on the standard output (screen) and returns the same character (on success, otherwise, returns EOF and sets the *error indicator* (ferror).)

```
#include <stdio.h>
int main( ) {
   int c;
   printf( "Enter a value :");
   c = getchar( );      / / try using getch(), getche()

   printf( "\nYou entered: ");
   putchar( c );

   return 0;}
```

gets() and puts()

char *gets(char *s)

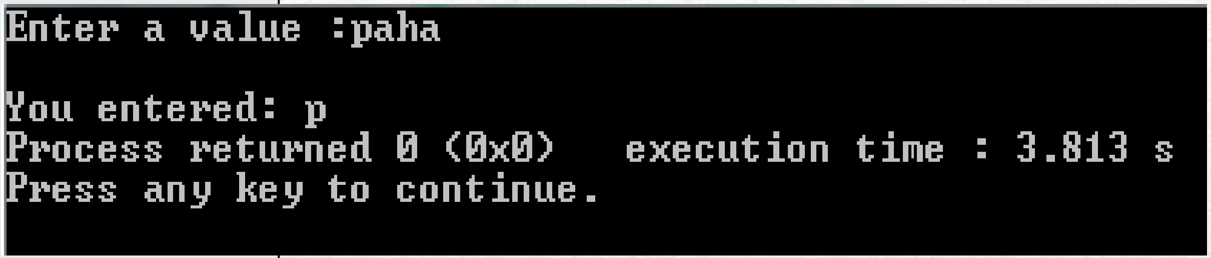
- Reads a line from the standard input (keyboard) as a string until either a terminating newline or EOF (End of File)
- EOF has a value -1 by default.

int puts(const char *s)

- Shows or writes the passed line or string on the standard output (screen) until it reaches the terminating null character ('\0'). This terminating null-character is not copied to the stream.
- On success, a non-negative value is returned.
  On error, the function returns EOF and sets the *error indicator* (ferror)

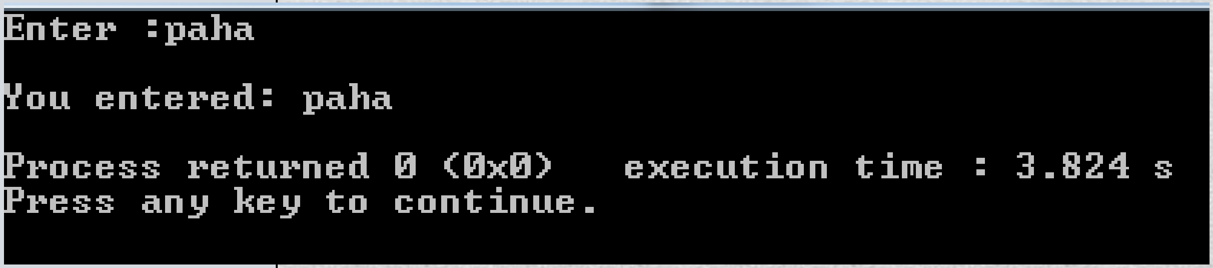
gets() and puts()

```
#include <stdio.h>
int main( ) {
   int c;
   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;}
```



```
Enter a value :paha

You entered: p
Process returned 0 (0x0)   execution time : 3.813 s
Press any key to continue.
```

```
#include <stdio.h>
int main( ) {
   char c[10];
   printf( "Enter :");
   gets(c);

   printf( "\nYou entered: ");
   puts( c );

   return 0;}
```



```
Enter :paha

You entered: paha

Process returned 0 (0x0)   execution time : 3.824 s
Press any key to continue.
```

scanf() and printf()

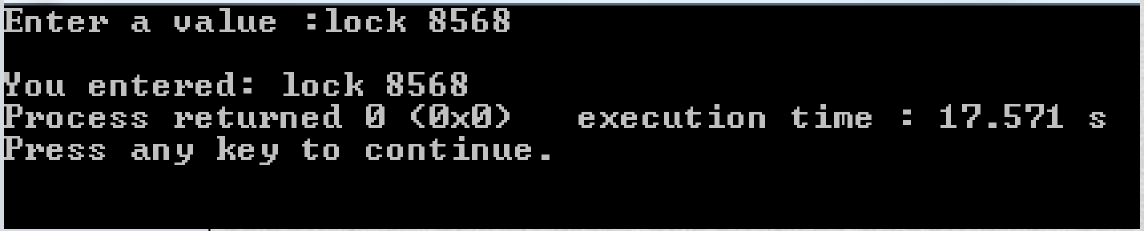
int scanf(const char *format, ...)

- reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.

int printf(const char *format, ...)

- writes the output to the standard output stream **stdout** and produces the output according to the **format** provided.
- The **format** can be a simple constant string, but one can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively.

```
#include <stdio.h>
int main( ) {
  char str[100];
  int i;
  printf( "Enter a value :");
  scanf("%s %d", str, &i);
  printf( "\nYou entered: %s %d ", str, i);
  return 0;
}
```



```
Enter a value :lock 8568

You entered: lock 8568
Process returned 0 (0x0)   execution time : 17.571 s
Press any key to continue.
```

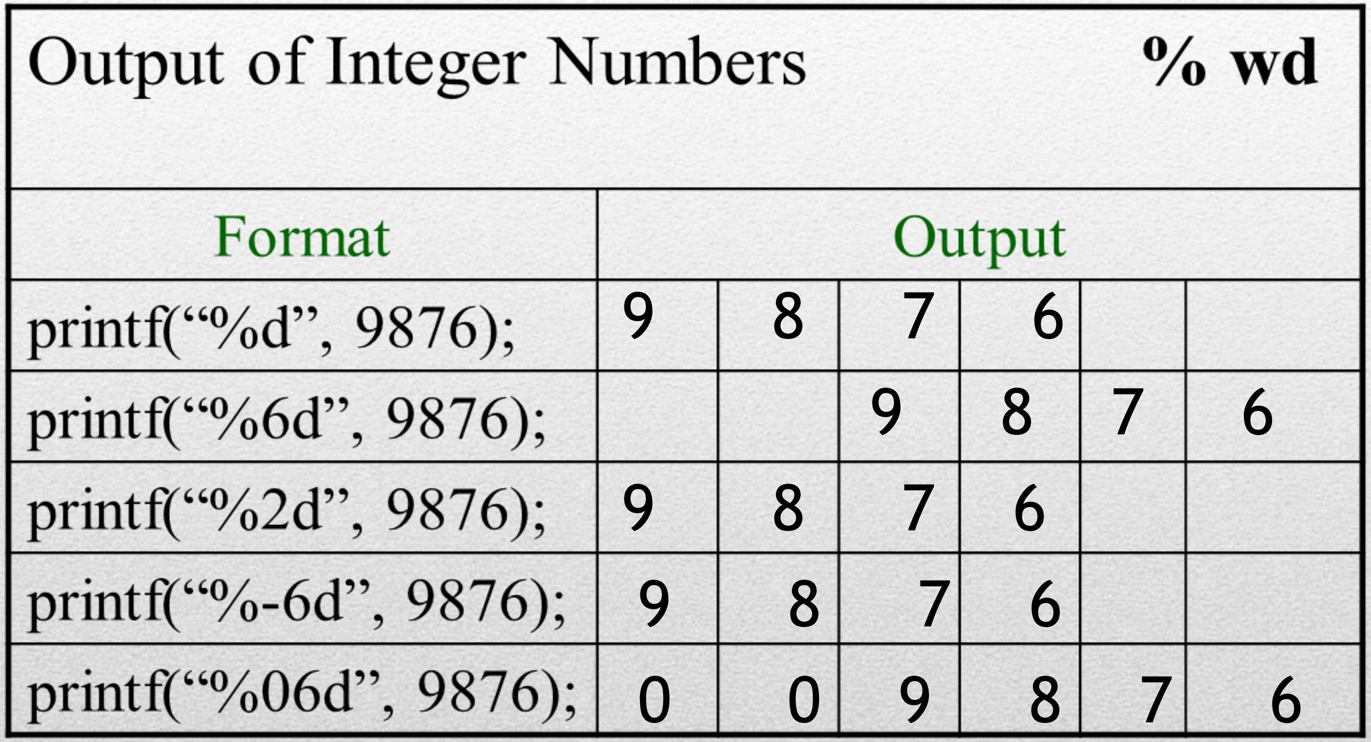
format of scanf()

- ***Whitespace character:*** The scanf() function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters). A single whitespace in the *format* string validates any quantity of whitespace characters extracted from the *stream* (including none).
- ***Non-whitespace character, except format specifier (%):*** Any character that is not either a whitespace character (blank, newline or tab) or part of a ***format specifier*** (which begin with a % character) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of *format*. If the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.
- ***Format specifiers:*** A sequence formed by an initial percentage sign (%) indicates a format specifier, **which is used to specify the type and format of the data to be retrieved from the *stream* and stored into the locations pointed by the additional arguments.**

Format Specifier

%d	**Integer**	**Signed decimal integer**
%i	**Integer**	**Signed decimal integer**
%o	**Integer**	**Unsigned octal integer**
%u	**Integer**	**Unsigned decimal integer**
%x	**Integer**	**Unsigned hexadecimal int (with a, b, c, d, e, f)**
%X	**Integer**	**Unsigned hexadecimal int (with A, B, C, D, E, F)**
%f	**Floating point**	**Signed Floating point value**
%e	**Floating point**	**Signed Floating point value of the exponential form**
%g	Floating point	Signed value in either e or f form, based on given value and precision. Trailing zeros and the decimal point are printed if necessary.
%E	Floating point	Same as e; with E for exponent.
%G	Floating point	Same as g; with E for exponent if e format used
%c	**Character**	**Single character**
%s	**String pointer**	**Prints characters until a null-terminator is pressed or precision is reached**
%%	**None**	**Prints the % character**

Format Modifier



Output of Integer Numbers						% wd
Format	Output					
printf(“%d”, 9876);	9	8	7	6		
printf(“%6d”, 9876);			9	8	7	6
printf(“%2d”, 9876);	9	8	7	6		
printf(“%-6d”, 9876);	9	8	7	6		
printf(“%06d”, 9876);	0	0	9	8	7	6

Format Modifier

Output of Real Numbers				% w.p f			% w.p e					
Format (y = 98.7654)	Output											
printf("%7.4f", y);	9	8	.	7	6	5	4					
printf("%7.2f", y);			9	8	.	7	7					
printf("%-7.2f", y);	9	8	.	7	7							
printf("%f", y);	9	8	.	7	6	5	4					
printf("%10.2e", y);			9	.	8	8	e	+	0	1		
printf("%11.4e", -y);	-	9	.	8	7	6	5	e	+	0	1	
printf("%-10.2e", y);	9	.	8	8	e	+	0	1				
printf("%e", y);	9	.	8	7	6	5	4	0	e	+	0	1

… (additional arguments) of scanf()

- **Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type.**
- **There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the function.**
- **These arguments are expected to be pointers: to store the result of a scanf operation on a regular variable, its name should be preceded by the *reference operator* (&) .**

DATA FILE INPUT/OUTPUT

getc(), putc(), fgets(), fputs(), fscanf(), fprintf().

DATA FILE I/O

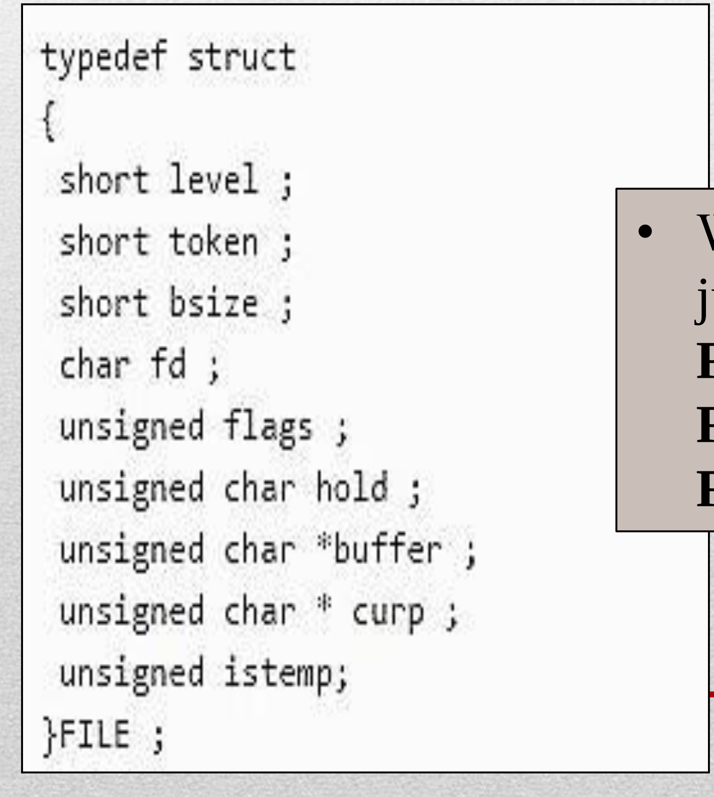
- Until now we have take input of some data and read corresponding output.
- All this input and output data are temporarily stored in RAM and become unavailable after closing the program.
- For some practical purpose some data need to be stored in Hard Disk as files and take input from the files.
- Data may be saved in a file using C and the file may be used in a program to get access of the saved data.
- Data file that is saved in a computer drive/disk may be opened for **reading, writing or appending.**
- The most commonly used functions for these purposes are
  fopen* and *fclose
  fprintf* and *fscanf
  fputs* and *fgets
- This operation reduces the headache of a programmer to entry a huge amount of data every time he run the program.

DATA FILE FORMAT IN C

Format:

FILE *file_pointer_name;

- FILE is a built in structure which is written in stdio.h header file and its members are



```
typedef struct
{
 short level ;
 short token ;
 short bsize ;
 char fd ;
 unsigned flags ;
 unsigned char hold ;
 unsigned char *buffer ;
 unsigned char * curp ;
 unsigned istemp;
}FILE ;
```

- We don’t need to write the whole structure, just write
  FILE *fa;
  FILE *fp;
  FILE *test1, *test2;

fprintf and fscanf

- fprintf is used to write data in an opened file
- fscanf is used to read data from an opened file
  - fprintf(filepointer,“formatspecifier”,arguments);
  - fscanf(filepointer,“formatspecifier”,&arguments);

fputs and fgets

- fputs is used to write string in an opened file
- fgets is used to read string from an opened file
  - fputs(“string”,filepointer);
  - fgets(stringvar,**number_of_character+1**,filepointer);

fprintf and fscanf

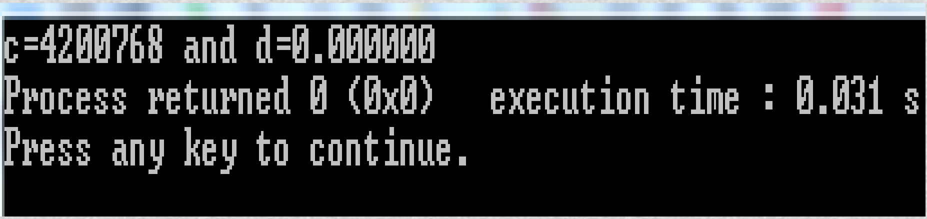
```
#include<stdio.h>
int main (void){
        FILE  *fa;   /* file pointer*/
        int a = 10,c; float b = 15.9,d;
        fa = fopen("D:\\filename.txt","w");
        fprintf(fa,"a = %d, b = %f", a, b);
        fscanf(fa,"%d %f",&c,&d);
        printf("a = c = %d, \t b = d = %f", a, b);
        fclose(fa);
        return 0;
}
```

The code will create a txt file named as filename that contains a = 10, b = 15.9

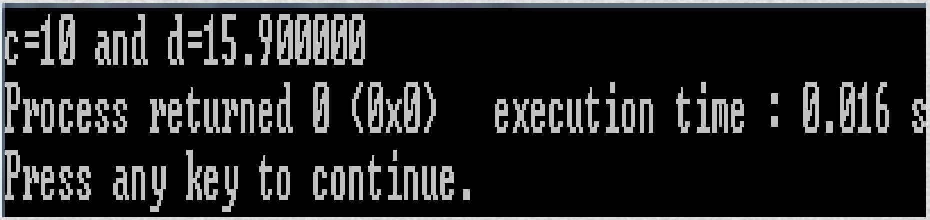
fprintf and fscanf

```
#include<stdio.h>
void main (void){
            FILE  *fa;
            int a = 10,c; float b = 15.9,d;
 fa = fopen("D:\\filename.txt","w");
 fprintf(fa,“ a = %d, b = %f”, a, b);
fclose(fa);
fa = fopen("D:\\filename.txt","r");
fscanf(fa,"%d %f",&c,&d);
printf("c=%d and d=%f", c, d);
fclose(fa);
}
```

```
#include<stdio.h>
void main (void){
            FILE  *fa;
            int a = 10,c; float b = 15.9,d;
            fa = fopen("D:\\filename.txt","w");
            fprintf(fa,"%d %f", a, b);
            fclose(fa);
            fa = fopen("D:\\filename.txt","r");
            fscanf(fa,"%d %f",&c,&d);
            printf("c=%d and d=%f", c, d);
            fclose(fa);
}
```



```
c=4200768 and d=0.000000
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```



```
c=10 and d=15.900000
Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

fputs() and fgets()

```
#include<stdio.h>
main(){
 char getline[60];
 FILE *fp;
 fp=fopen("D:\\test.txt","w");
 fputs("I have to be a good programmer",fp);
 fclose(fp);
 fp=fopen("D:\\test.txt","r");
 fgets(getline,31,fp);   //30 characters + 1
 puts(getline);
 fclose(fp);
}
```

- fgets() reads characters from stream and stores them as a C string into str until **(character number-1)** characters have been read or either a newline or the end-of-file is reached, whichever happens first.
- A terminating null character is automatically appended after the characters copied to str.

What if one writes

fputs("I have to be a good programmer\n I am not a good programmer",fp);