Declaration of storage Class

- Variables in C can have not only ***data type*** but also *storage class* to provide information about their location and visibility.
- Storage class decides portion of program within which the variables are recognized.

```
/* Example of storage classes */
int  m;
main()
{
   int i;
   float balance;
   ....
   ....
  function1();
}
function1()
{
   int i;
   float sum;
   ....
   ....
   }
```

- Variable ***m*** is ***global variable***, used in all functions, and known as **external variables**.
- Variable ***i, balance, and sum*** are ***local variables***, declared inside function, are visible and meaningful only under the scope of function, and not known to other functions.
- ***Local variable*** value in a function is not affected by changes made outside function.

Declaration of storage Class contd.

- C provide variety of storage class specifiers used to declare explicitly the scope and lifetime of variables.
- Variable ***scope*** and ***lifetime*** important only in multifunction and multiple file programs.
- ***Four storage classes:***

1. ***auto***
2. ***register***
3. ***static***
4. ***extern***

- Storage another qualifier (like **long** or **unsigned**) with variable declaration: ***auto int*** count; ***register char*** ch; ***static int*** x; ***extern long*** total
- ***static*** and ***extern*** variables automatically initialized to zero.
- ***auto*** contain ***undefined*** or ***garbage*** values unless explicitly initialized.

Storage class	*Meaning*
auto	Local variable known only to the function in which it is declared. *Default is auto.*
static	Local variable which exists and retains its value even after the control is transferred to the calling function.
extern	Global variable known to all functions in the file.
register	Local variable which is stored in the register.

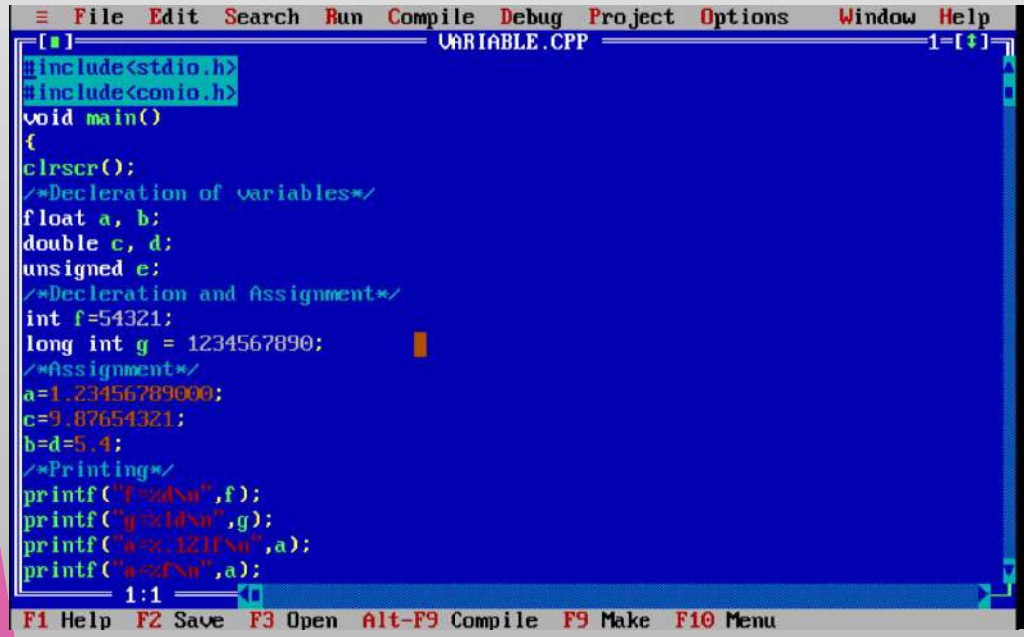
Declaration of storage Class contd.

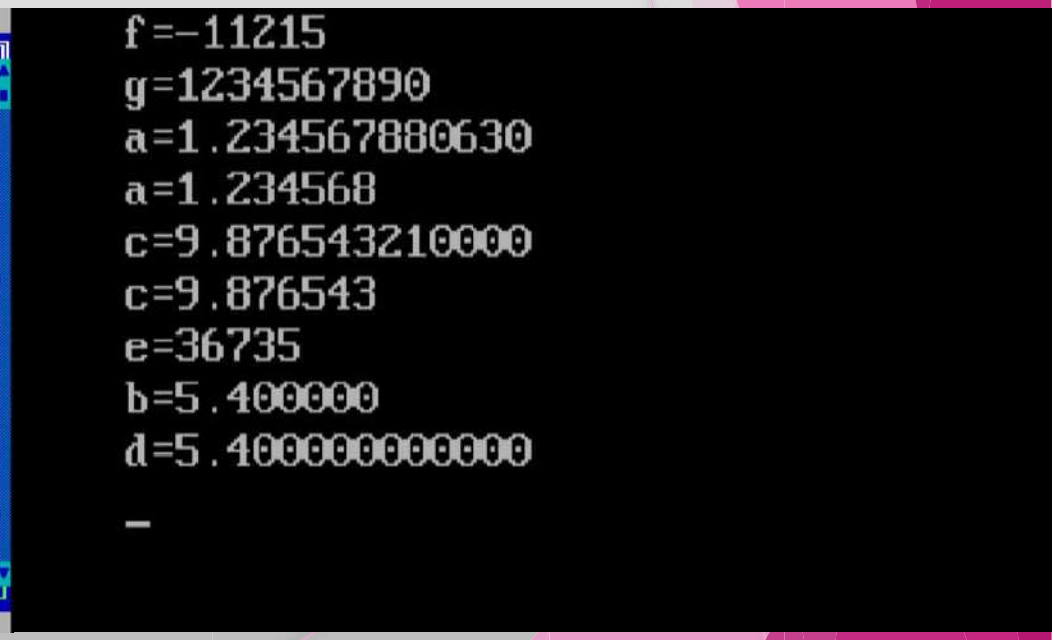
Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

Variable value assigning

- Assignment using assignment operator denoted by '='. e.g. **variable_name=constant**
- Multiple assignments can be done, such as, **initial_value=0; final_value=20;**
- ***Initialization:*** the process of giving initial values to variables.
- ***External and Static*** variables are initialized to ***"zero"***.
- ***Variable*** not initialized ***assigned garbage value***.

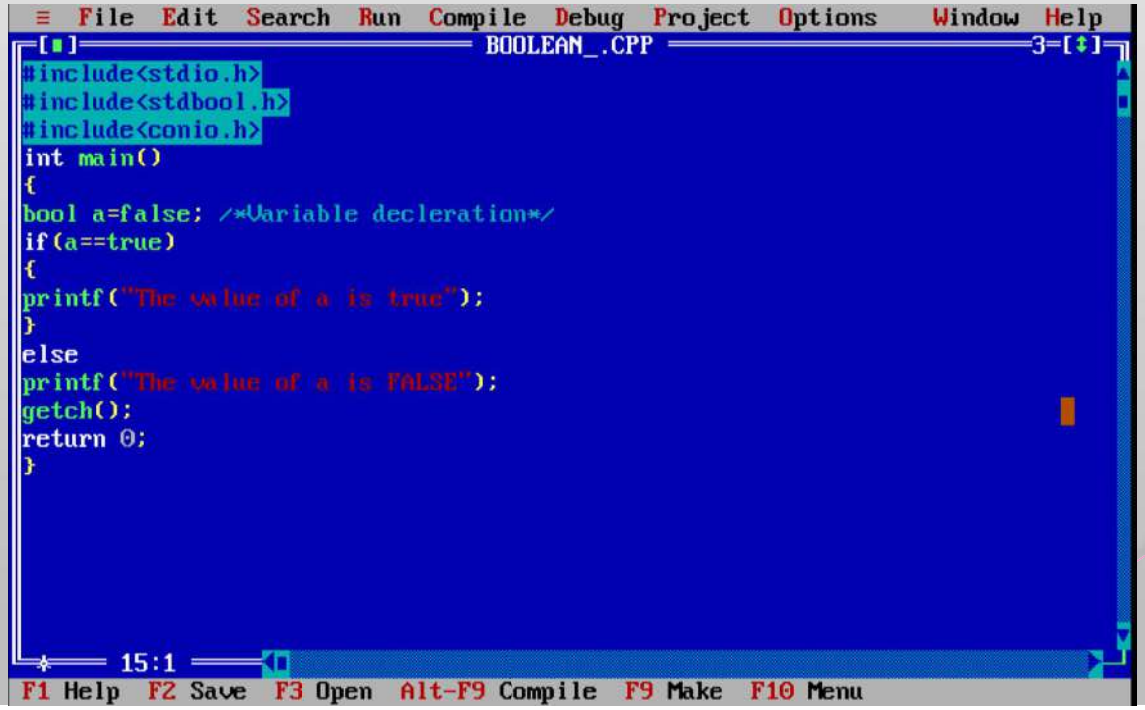




C Boolean Variable value assigning

- Boolean is a data type that contains two types of values, i.e., 0 and 1.
- ***bool*** type value represents two types of behavior, either true or false.
- ***'0'*** represents ***false***, while ***'1'*** represents ***true*** value.
- ***Syntax:***

bool variable_name;



Symbolic Constants

- ***Constants*** play unique role in program.
- Syntax: ***#define Pi 3.14159***
- ***# define*** a preprocessor directive
- ***Example: pi*** with value of 3.142
- ***Two problems*** encountered during symbolic constant definition:
  - ***Modifiability***
  - ***Understandability***
- ***Symbolic*** constants also known as ***constant identifiers*** and don't appear in declaration part
- Rules for defining ***#define*** statement:

1. Symbolic names have the same form as variable names. (Written in CAPITALS for distinguishing)
2. No blank space between ***'#'*** and ***'define'***.
3. Blank space between ***#define*** and ***symbolic name***
4. ***#define*** must not end with semicolon.
5. Symbolic names are NOT declared for data types. Its data type depends on the type of constant.
6. #define may appear anywhere (usually placed at the beginning).

Various Invalid #define statements

Statement	*Validity*	*Remark*
#define X = 2.5	Invalid	'=' sign is not allowed
# define MAX 10	Invalid	No white space between # and define
#define N 25;	Invalid	No semicolon at the end
#define N 5, M 10	Invalid	A statement can define only one name.
#Define ARRAY 11	Invalid	define should be in lowercase letters
#define PRICE$ 100	Invalid	$ symbol is not permitted in name

Variable as Constants and Volatile (Overflow and Underflow Problem)

- ***const*** qualifier utilized to fix variable values throughout program. e.g.: ***const int*** class_size =40;
- ***volatile*** qualifier used to explicitly inform compiler that a variable's value may be changed by external sources. e.g.

volatile int date;

- Compiler examine volatile variable regularly for changes in the examination.
- ***Note: Volatile type qualifier is particularly relevant for multi-threaded program and parallel processes***
- Overflow problem arises when variable data is too big or too small for the data type to hold.
- Variable Largest value dependent on machine.
- ***Overflow*** and ***Underflow*** arises when floating-point values are rounded off to the number of significant digits allowed
- ***Overflow*** results in ***largest possible real value*** and ***Underflow*** results in ***zero***.
- C compiler doesn't provide warning or indication of integer overflow.

Operators and Expressions

- C supports set of built-in operators. e.g. **=, +, -, *, &, *and* <.**
- ***Operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.***
- Used to manipulate data and variables.
- Operators classified into categories such as:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and Decrement operators
6. Conditional operators
7. Bitwise operators
8. Comma operators
9. Other operators
10. ***sizeof*** operators

Arithmetic Operators

- Used for numeric calculations.
- Two types:
  - Unary: Require only one operand
  - Binary: Require two operands
- Five binary arithmetic operators
- Integer division truncates any fractional part
- Unary minus operator, multiples its single operand by -1
- Number preceded by minus sign changes its sign
- ***Note: % (Modulo division) operation cannot be used on floating point data.***

Operator	*Meaning*
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Integer Arithmetic

- Both operands in arithmetic expression such as ***a+b*** are integers then expression ***integer expression*** and operation is called ***integer arithmetic***.
- ***Integer arithmetic*** yields integer values. e.g. if ***a*** and ***b*** are integers then

a+b=18

a-b=10

a*b=56

a/b=3(decimal part truncated)

a%b=2(remainder of division process)

- During division if both operands are of the same sign, ***the result is truncated towards zero***.
- ***During Modulo division, sign of the result is always the sign of the first operand***.

Integer Arithmetic

```
Program
   main ()
   {
      int months, days ;

      printf("Enter days\n") ;
      scanf("%d", &days) ;

      months = days / 30 ;
      days = days % 30 ;
      printf("Months = %d Days = %d", months, days) ;
   }
```

Real Arithmetic

- An arithmetic operation involving only real operands is called ***real arithmetic***.
- A real operand may assume values in decimal or exponential notations.
- ***Floating point values rounded to the number of significant digits permissible***.
- ***Final value*** an approximation of correct result.
- ***Operator % can't be used with real operands***.

Mixed-Mode Arithmetic

- When one of the operands is real and the other is integer.
- ***Expression is called mixed-mode arithmetic.***
- If either operand is real then only the real operation is performed and result is a real number.

If a = 12, b = 2.5

Expression	Result
a+b	14.5
a-b	9.5
a*b	30.0
a / b	4.8

Relational Operators

- Compare two quantities and depending on their relation we make decisions.
- We may compare age of two persons, or price of two items.
- Comparisons done with ***relational operator***.
- We use symbol ***'<'***, meaning ***'less than'***.
- ***An expression such as a<b is termed as relational expression.***
- ***Relational expression value is either 1 (TRUE) or 0 (FALSE).***

Operator	*Meaning*
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Relational Operators Complements

- Among the six relational operators, each one is a complement of another operator.

>	is complement of	<=
<	is complement of	>=
==	is complement of	!=

- We can simplify an expression involving the not and the less than operators using the complements

Actual one	*Simplified one*
!(x < y)	x >= y
!(x > y)	x <= y
!(x != y)	x == y
!(x <= y)	x > y
!(x >= y)	x < y
!(x == y)	x != y

TIP: It is advisable not to use floating point operands with relational operators

Logical Operators

- C has ***three logical operators***.

&&	meaning logical	AND
||	meaning logical	OR
!	meaning logical	NOT

- Logical operators ***&&*** and ***||*** are used when we want to ***test more than one condition and make decisions***. e.g. ***a>b && x==10***
- ***Def: An expression of this kind, which combines two or more relational expressions, is termed as a logical expression or a compound relational expression.***
- Logical expression also yields a value of one or zero, according to the truth table

op-1	*op-2*	*Value of the expression*	
		op-1 && op-2	*op-1 || op-2*
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Logical Operators contd.

- *Some examples of the usage of logical expressions are:*

1. *if(age>55&&salary<1000)*
2. *if(number<0||number>100)*

Note *Relative precedence of the relational and logical operators is as follows:*

Highest	!
	> >= < <=
	== !=
	&&
Lowest	||

It is important to remember this when we use these operators in compound expressions.

Increment and Decrement Operators

- Two very powerful operators ***not found in other languages***.
- ***Increment and Decrement operators: ++ and --***
- ***Operator ++ adds 1 to operand, while -- subtracts 1***. Both are ***unary operators***.
- ***Syntax: ++m; m++; --m; m--;***
- ***Used extensively in iteration statements***
- While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement

```
m = 5;
y = ++m;
```

- In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

```
m = 5;
y = m++;
```

then, the value of y would be 5 and m would be 6.

- ***Prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left.***
- ***Postfix operator first assigns value to the variable on the left and then increments the operand.***

Rules for Increment and Decrement Operators

Rules for ++ and –– Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or ––) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++(or ––) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associatively of ++ and –– operators are the same as those of unary + and unary –.

Conditional Operators

A ternary operator pair "? :" is available in C to construct conditional expressions of the form

exp1 ? exp2 : exp3

where *exp1*, *exp2*, and *exp3* are expressions.

The operator ? : works as follows: *exp1* is evaluated first. If it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the expression. If *exp1* is false, *exp3* is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either *exp2* or *exp3*) is evaluated. For example, consider the following statements.

```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the **if..else** statements as follows:

```
if (a > b)
    x = a;
else
    x = b;
```

Bitwise Operators

- Manipulation of data at bit level.
- ***Operators are used for testing, or shifting of bits.***
- ***Bitwise operator not applied to float*** or ***double.***

Operator	*Meaning*
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

Special Operators

- Operators like ***comma, sizeof, pointer operators (& and *), and member selection operators.***
- Comma operator used to link relational expressions together.
- A comma-linked list of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression. e.g. ***value = (x = 10, y = 5, x+y);***
- ***Comma operator has lowest precedence.***
- The ***sizeof*** is a compile time operator and, when used with an operand, it returns the number of bytes the operand (***a variable, a constant, or a data type qualifier***) occupies
- ***Sizeof*** used to determine array and structures lengths
- ***Allocate memory space dynamically***

```
#include<stdio.h>
main( )
{
    int var;
    printf ("Size of int = %d",sizeof(int));
    printf("Size of float = %d",sizeof(float));
    printf("Size of var = %d",sizeof(var));
    printf("Size of an integer constant = %d",sizeof(45));
}
```

Output:

```
Size of int = 2
Size of float = 4
Size of var = 2
Size of an integer constant = 2
```

Arithmetic Expressions

- Arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language.
- ***Remember C does not have an operator for exponentiation.***

Algebraic expression	*C expression*
a x b - c	a * b - c
(m+n) (x+y)	(m+n) * (x+y)
$\left(\frac{ab}{c}\right)$	a * b/c
$3x^2 + 2x + 1$	3 * x * x 2 * x + 1
$\left(\frac{x}{y}\right) + c$	x/y+c

- ***Expression evaluation is done through assignment like, variable = expression;***

Precedence of Arithmetic Operators

- An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators.

Highest priority * / %

Low priority + -

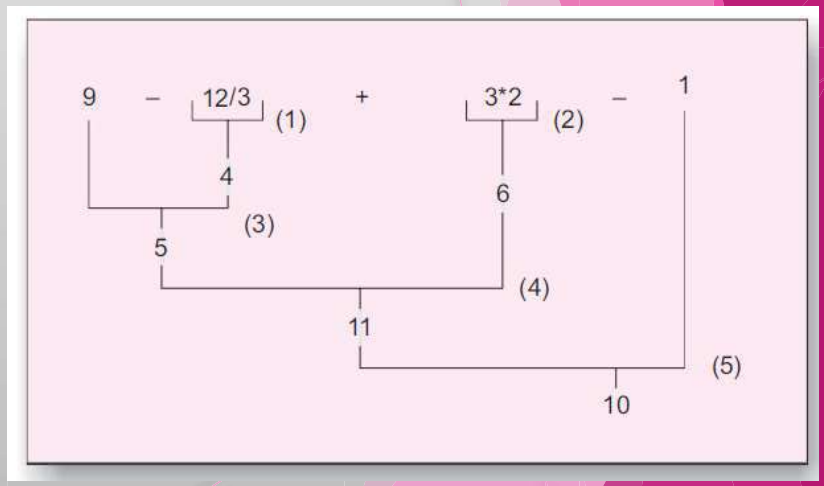
- The basic evaluation procedure includes 'two' left-to-right passes through the expression. Consider the following evaluation statement

x = a-b/3 + c*2-1

- When a = 9, b = 12, and c = 3, the statement becomes

x = 9-12/3 + 3*2-1

- and is evaluated as
- ***Step 1: x=9-4+3*2-1***
- ***Step 1: x=9-4+6-1***
- ***Step 1: x=5+6-1***
- ***Step 1: x=11-1***
- ***Step 1: x=10***



Rules for Evaluation of Expressions

- First, parenthesized sub expression from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

Type Conversion

- C permits mixing of constants and variables of different types in an expression.
- Automatic conversion to proper type without loosing any significance
- ***Implicit Type Conversion***
- ***Explicit Type Conversion or Casting a value***
- Rules that are applied while evaluating expressions:

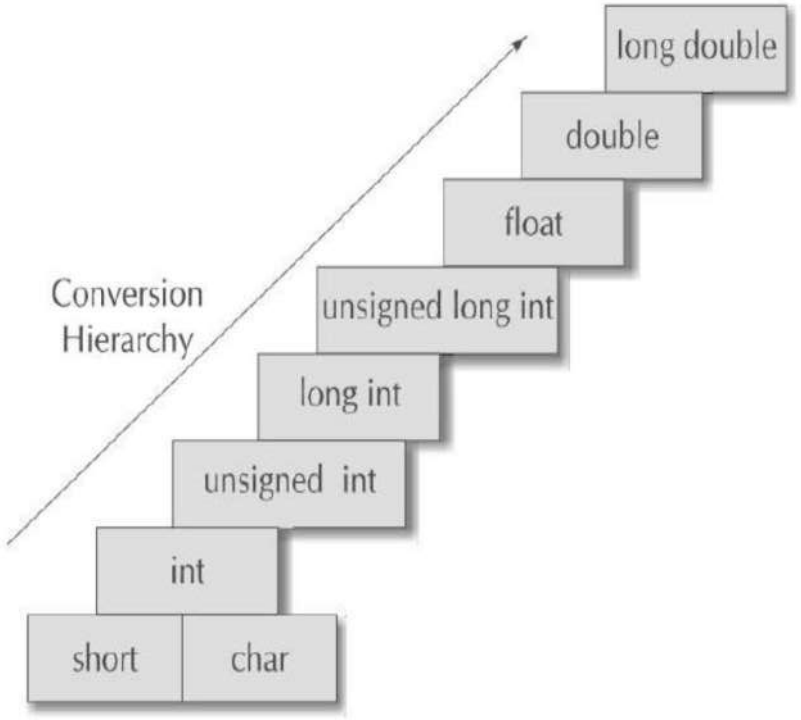
1. All ***short*** and ***char*** are automatically converted to ***int***
2. if one of the operands is ***long double*** the other will be converted to ***long double*** and the result will be ***long double***;
3. else, if one of the operands is ***float*** the other will be converted to ***float*** and the result will be ***float***;
4. else, if one of the operands is ***unsigned long int***, the other will be converted to ***unsigned long int*** and the result will be ***unsigned long int***
5. else, if one of the operands is ***long int*** and the other is ***unsigned int***, then

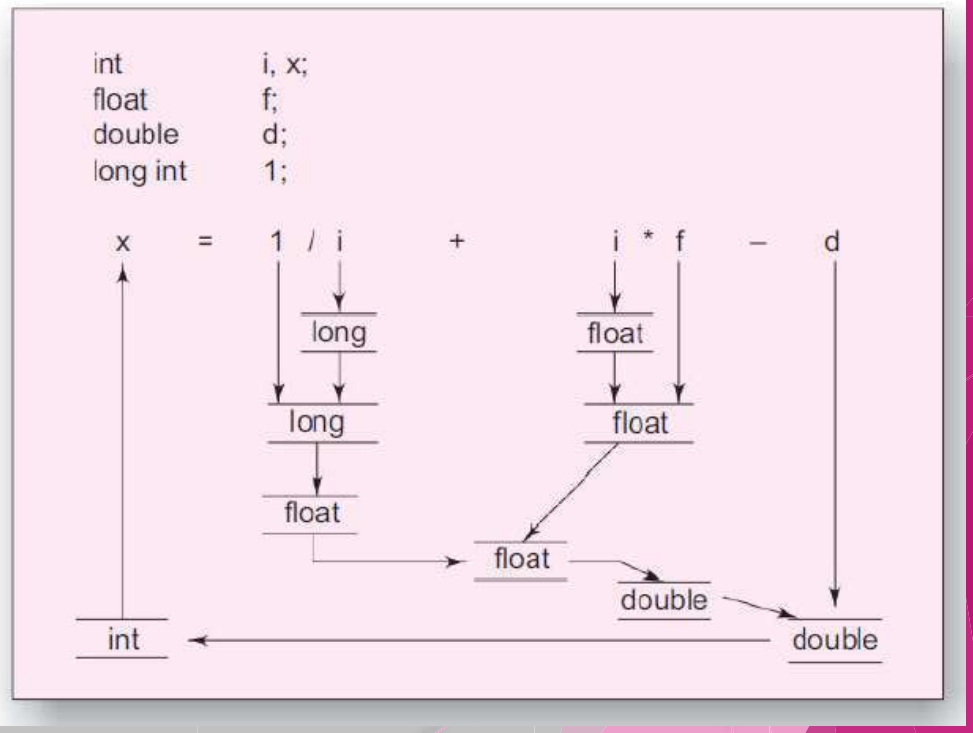
(a) if ***unsigned int*** can be converted to ***long int***, the ***unsigned int*** operand will be converted as such and the result will be ***long int***

(b) else, both operands will be converted to ***unsigned long int*** and the result will be ***unsigned long int***

6. else, if one of the operands is ***long int***, the other will be converted to ***long int*** and the result will be ***long int***
7. else, if one of the operands is ***unsigned int***, the other will be converted to ***unsigned int*** and the result will be ***unsigned int***.

Type Conversion contd.





Operator Precedence

- C has a precedence associated with it used to determine how an expression involving more than one operator is evaluated.
- Distinct levels of operator precedence. Operators with higher precedence level are first evaluated.
- ***Associativity Property: Operators*** with ***same level of precedence*** are evaluated either ***from 'left to right'*** or ***from 'right to left' depending upon the level.***
- Very important to note carefully, the order of precedence and associativity of operators. e.g.

  if (x == 10 + 15 && y < 10)

- rules say that the + has a higher priority than ***&& and == or <<. So + is executed first, then == and << are executed second, and finally && is executed.***

Operator	*Description*	*Associativity*	*Rank*
( ) [ ]	Function call Aray element reference	Left to right	1
+ – ++ –– ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer reference (indirection) Address Size of an object Type cast (conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+ –	Addition Subtraction	Left to right	4
<< >>	Left shift Right shift	Left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right	6
== |=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
= * = /= %= += –= &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

Math Functions

- Points to remember while using math functions

1. x and y should be declared as ***double***
2. In trigonometric and hyperbolic functions, x and y are in radians
3. All the functions return a ***double***.
4. C99 has added ***float*** and ***long double*** versions of these functions.
5. C99 has added many more mathematical functions.
6. ***Always include the respective math header file. #include<math.h>***

Function	*Meaning*
Trigonometric	
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan 2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
Hyperbolic	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
Other functions	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the x power (e^x)
fabs(x)	Absolute value of x.
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Remainder of x/y
log(x)	Natural log of x, x > 0
log10(x)	Base 10 log of x, x > 0
pow(x,y)	x to the power y (x^y)
sqrt(x)	Square root of x, x > = 0

Managing Input and Output Operations

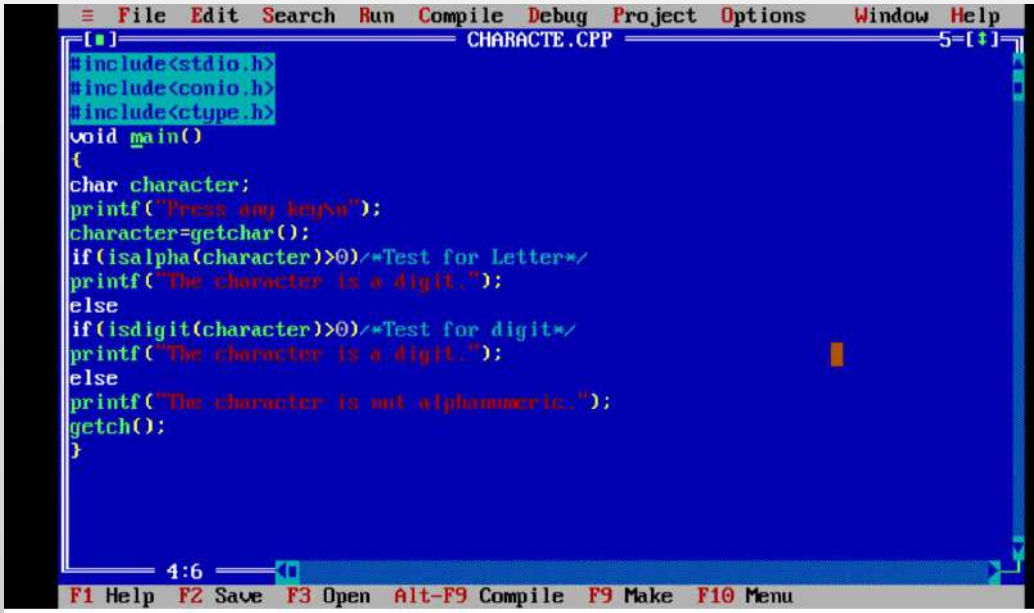
- Reading, processing, and writing of data are the three essential functions of a computer program.
- Programs take some data as input and display the processed data, often known as information or results, on a suitable medium.
- C does ***not have any built-in input/output statements as part of its syntax***.
- ***Input/output*** operations are carried out through function calls such as ***printf*** and ***scanf***.
- For input function ***scanf*** which can read data from a keyboard.
- For outputting results, the function ***printf*** which sends results out to a terminal.
- ***#include <math.h>*** is included to execute math functions.
- ***#include <stdio.h>*** is included to use a standard input/output function.
- ***Not necessary for the functions printf and scanf***
- stdio.h is an abbreviation for standard input-output header file.
- #include <stdio.h> tell compiler to search for file named stdio.h and place its contents at this point in the program.

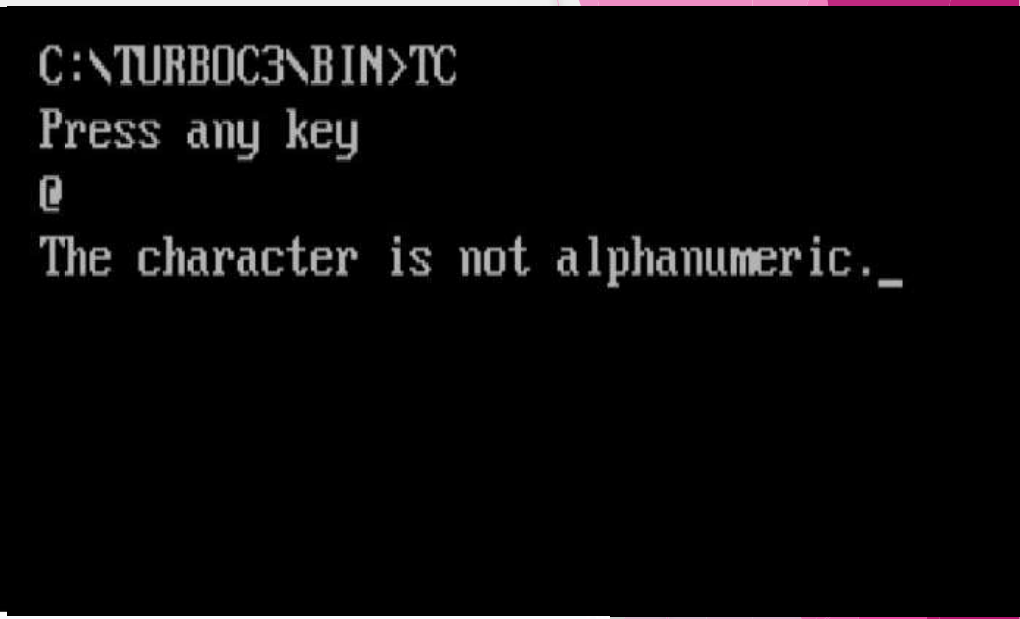
Reading Character

- simplest of all input/output operations is reading a character from the ‘standard input’ unit (usually the keyboard) and writing it to the ‘standard output’ unit (usually the screen).
- Reading a single character can be done by using the function ***getchar()*** (also done with the help of the scanf function)
- ***getchar*** takes the following form: ***variable_name = getchar( );***
- variable_name is a valid C name that has been declared as char type.
- Computer waits until a key is pressed and then assigns this character as a value to ***getchar*** function.
- ***getchar*** is used on the ***right-hand side of an assignment statement***, the ***character value of getchar is in turn assigned to the variable name on the left***.
- Program displays a question of YES/NO type to the user and reads the user’s response in a singlecharacter (Y or N). If the response is Y or y, it outputs the message “My name is BUSY BEE” otherwise, outputs “You are good for nothing”

```
#include <stdio.h>
main()
{
   char answer;
   printf("Would you like to know my name?\n");
   printf("Type Y for YES and N for NO: ");
   answer = getchar(); /* .... Reading a character...*/
   if(answer == 'Y' || answer == 'y')
      printf("\n\nMy name is BUSY BEE\n");
   else
   printf("\n\nYou are good for nothing\n");
}
```

Reading Character contd.





Function	*Test*
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c lower case letter?
isprint(c)	Is c a printable character?
ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a white space character?
isupper(c)	Is c an upper case letter?

Writing Character

- Like ***getchar,*** there is an analogous function ***putchar*** for writing characters one at a time to the terminal.
- ***putchar (variable_name);*** statement displays the character contained in the variable_name at the terminal. For example, the statements answer = 'Y'; putchar (answer);
- ***putchar ('\n');*** would cause the cursor on the screen to move to the beginning of the next line.

```
#include <stdio.h>
#include <ctype.h>
main()
{
   char alphabet;
   printf("Enter an alphabet");
   putchar('\n'); /* move to next line */
   alphabet = getchar();
   if (islower(alphabet))
   putchar(toupper(alphabet));/* Reverse and display */
else
   putchar(tolower(alphabet)); /* Reverse and display */
}
```

```
Enter an alphabet
a
A
Enter an alphabet
Q
q
Enter an alphabet
z
Z
```

- program use: ***islower, toupper, and tolower***. Function ***islower*** conditional function and takes the value ***TRUE*** if argument is lowercase alphabet; otherwise ***FALSE***. The function ***toupper*** converts the ***lowercase argument into an uppercase alphabet*** while the function ***tolower does the reverse***.

Formatted Input

- Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data: ***15.75 123 John***
- ***15.75*** stored in ***float***, ***123*** into ***int***, and ***John*** into ***char***. Possible in C using ***scanf*** function (scanf means scan formatted) as ***scanf ("control string", arg1, arg2, ... argn);***
- Control string (also known as format string) specifies the field format in which the data is to be entered and the arguments arg1, arg2, ..., argn specify the address of locations where the data is stored.
- Control string and arguments are separated by commas.

Inputting Integer Numbers

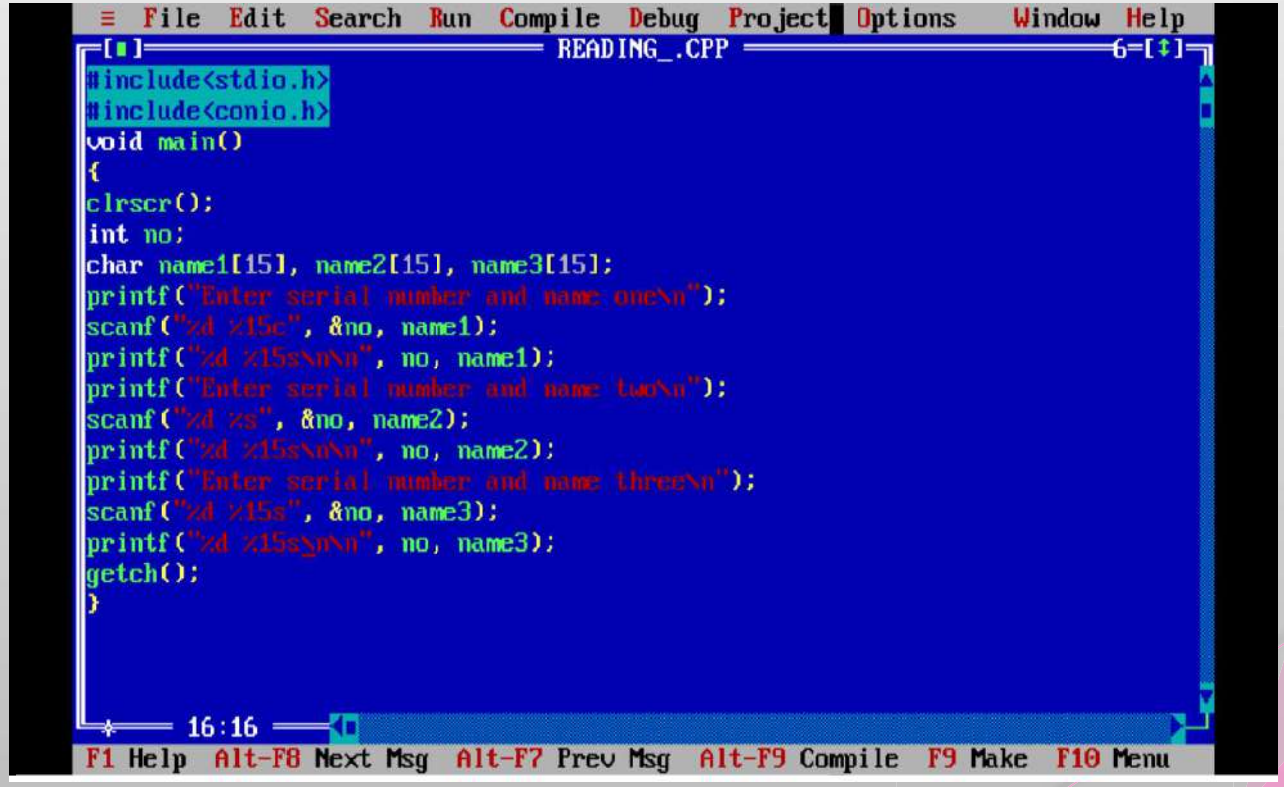
- ***Field specification for integer number reading is %wsd.***
- *%* indicates conversion specific that follows. ***w*** is an integer number that specifies filed width of the number to be read and d, known as data type character, indicates that the number to be read is in integer mode.
- E.g.: ***scanf ("%2d %5d", &num1, &num2); Data line: 50 31426***
- The value 50 assigned to ***num1*** and ***31426*** assigned to num2.
- Suppose new input data is as follows: 31426 50
- num1 takes 31 and num2 will be assigned 426
- 50 will be assigned to next scanf.
- ***What will happen if we enter floating point number.***
- ***What will happen in scanf("%d %*d %d", &a, &b). * is used to skip the contents***

```
main()
{
   int a,b,c,x,y,z;
   int p,q,r;
   printf("Enter three integer numbers\n");
   scanf("%d %*d %d",&a,&b,&c);
   printf("%d %d %d \n\n",a,b,c);
   printf("Enter two 4-digit numbers\n");
   scanf("%2d %4d",&x,&y);
   printf("%d %d\n\n", x,y);
   printf("Enter two integers\n");
   scanf("%d %d", &a,&x);
   printf("%d %d \n\n",a,x);
   printf("Enter a nine digit number\n");
   scanf("%3d %4d %3d",&p,&q,&r);
   printf("%d %d %d \n\n",p,q,r);
   printf("Enter two three digit numbers\n");
   scanf("%d %d",&x,&y);
   printf("%d %d",x,y);
}
```

```
Enter three integer numbers
1 2 3
1 3 -3577
Enter two 4-digit numbers
6789 4321
67 89
Enter two integers
44 66
4321 44
Enter a nine-digit number
123456789
66 1234 567
Enter two three-digit numbers
123 456
89 123
```

Inputting Character String

- ***scanf*** function can input strings containing more than one character as ***%ws*** or ***%wc***.
- Reading Mixed Data Types: ***scanf ("%d %c %f %s", &count, &code, &ratio, name);***



Points to Remember for Scanf

- General points to keep in mind while writing a scanf statement.

1. All function arguments, except the control string, must be pointers to variables.
2. Format Specifications contained in the control string should match the arguments in order
3. Input data items must be separated by spaces and must match the variables receiving the input in the same order.
4. The reading will be terminated, when scanf encounters a 'mismatch' of data or a character that is not valid for the value being read.
5. When searching for a value, ***scanf*** ignores line boundaries and simply looks for the next appropriate character.
6. Any unread data items in a line will be considered as part of the data input line to the next scanf call.
7. When the field width specifier ***w*** is used, it should be large enough to contain the input data size.

Code	*Meaning*
%c	read a single character
%d	read a decimal integer
%e	read a floating point value
%f	read a floating point value
%g	read a floating point value
%h	read a short integer
%i	read a decimal, hexadecimal or octal integer
%o	read an octal integer
%s	read a string
%u	read an unsigned decimal integer
%x	read a hexadecimal integer
%[..]	read a string of word(s)

Formatted Output

- ***printf*** function for printing captions and numerical results such that they are understandable and are in an easy-to-use form.
- ***printf*** statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals.
- General form of printf statemen is: printf("control string", arg1, arg2, ..., argn);
- Control string will consist of
  - 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 such as \n, \t, and \b.
- Format specification form: ***%w.p type-specifier***
- ***w is integer number specifying total number of columns for the output value***
- ***p is integer number specifying number of digits to the right of the decimal point or the number of characters to be printed from a string***

Printing of a Single Character and Strings

- Single character can be displayed in a desired position using the format: ***%wc***
- Character will be displayed ***right-justified in the field of w columns***.
- Display ***left-justified*** by placing a minus sign before the integer ***w***.
- ***default value for w is 1.***
- String can be displayed in a desired position using the format: ***%w.ps***
- ***w specifies field width for display***
- ***p instructs that only first p characters of the string to be displayed***. Display is right-justified

Format for Printf

Code	*Meaning*
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form
%f	print a floating point value without exponent
%g	print a floating point value either e-type or f-type depending on
%i	print a signed decimal integer
%o	print an octal integer, without leading zero
%s	print a string
%u	print an unsigned decimal integer
%x	print a hexadecimal integer, without leading Ox