C Program,
Variables, Data types, sizes and constants

O b j e c t i v e s

- To learn and appreciate
  - General Structure of C program
  - C Tokens
  - Variables
  - Declarations
  - Data Types and Sizes

Session outcome

At the end of session student will be able to learn and understand

- General structure of C program
- C Tokens
- Variables
- Declarations
- Data Types and Sizes

General Structure of C program

Documentation section

Link section

Definition section

Global declaration section

main () Function section

{

Declaration part
Executable part

}

Subprogram section

Function 1
Function 2
....
....
Function n

(User defined functions)

C program for reading a number and display it on the screen

```
//Program to read and display a number
#include<stdio.h>
int main()
{
 int num;
 printf("\nEnter the number: ");
 scanf("%d", &num);
 printf("The number read is: %d", num);
return(0);
}
```

Adding two integers

```
#include <stdio.h>
int main( void )
{/* start of function main */
    int sum; /* variable in which sum will be stored */
    int integer1; /* first number to be input by user */
    int integer2; /* second number to be input by user */
    printf( "Enter first integer\n" );
    scanf( "%d", &integer1 ); /* read an integer */
    printf( "Enter second integer\n" );
    scanf( "%d", &integer2 ); /* read an integer */
    sum = integer1 + integer2; /* assign total to sum */
    printf( "Sum is %d\n", sum ); /* print sum */
    return 0; /* indicate that program ended successfully */
} /* end of function main */
```

C Character set

- Character set is a set of valid characters that a language can recognize.
- C character set consists of letters, digits, special characters, white spaces.

(i) **Letters** → 'a', 'b', 'c',……….'z' Or 'A', 'B', 'C',……….'Z'

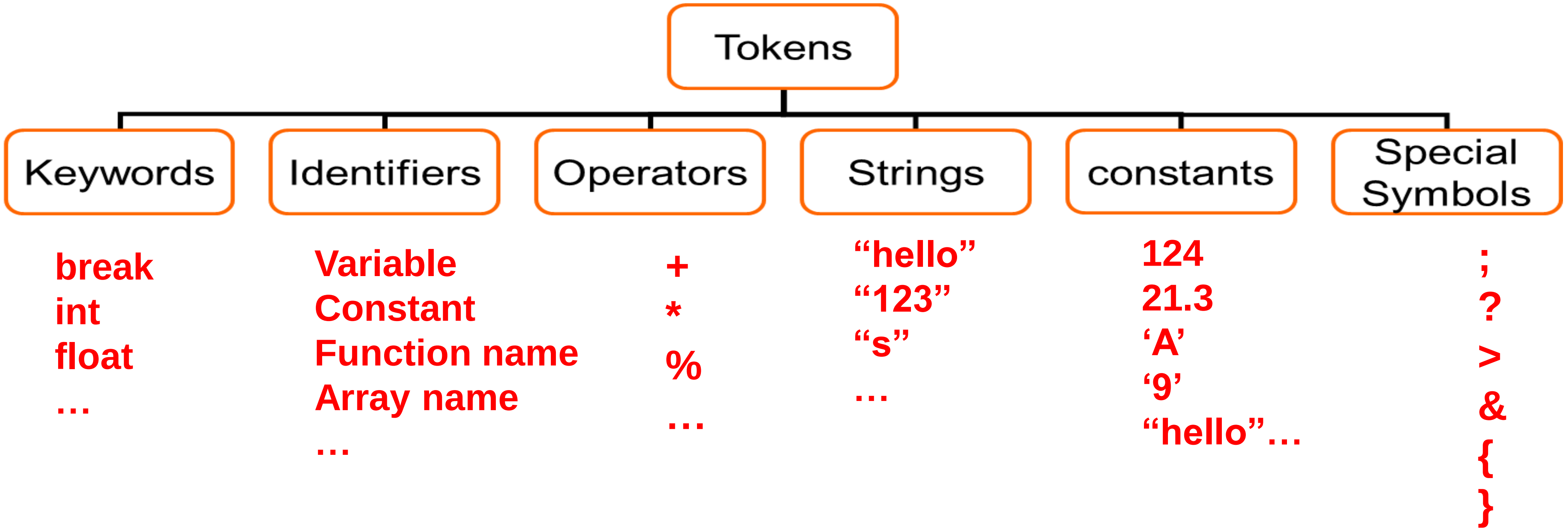
(ii) **Digits** → 0, 1, 2,……………………9

(iii) **Special characters** → ;, ?, >, <, &,{, }, [,]……

(iv) **White spaces** → New line (\n), Tab(\t), Vertical Tab(\v) etc

C Tokens

- ✓ A token is a group of characters that logically belong together.
- ✓ The programmer writes a program by using tokens.
- ✓ C uses the following types of tokens.



Keywords

- **These are some reserved words in C which have predefined meaning to compiler called keywords.**
- **Keywords are not to be used as variable and constant names.**
- **All keywords have fixed meanings and these meanings cannot be changed.**

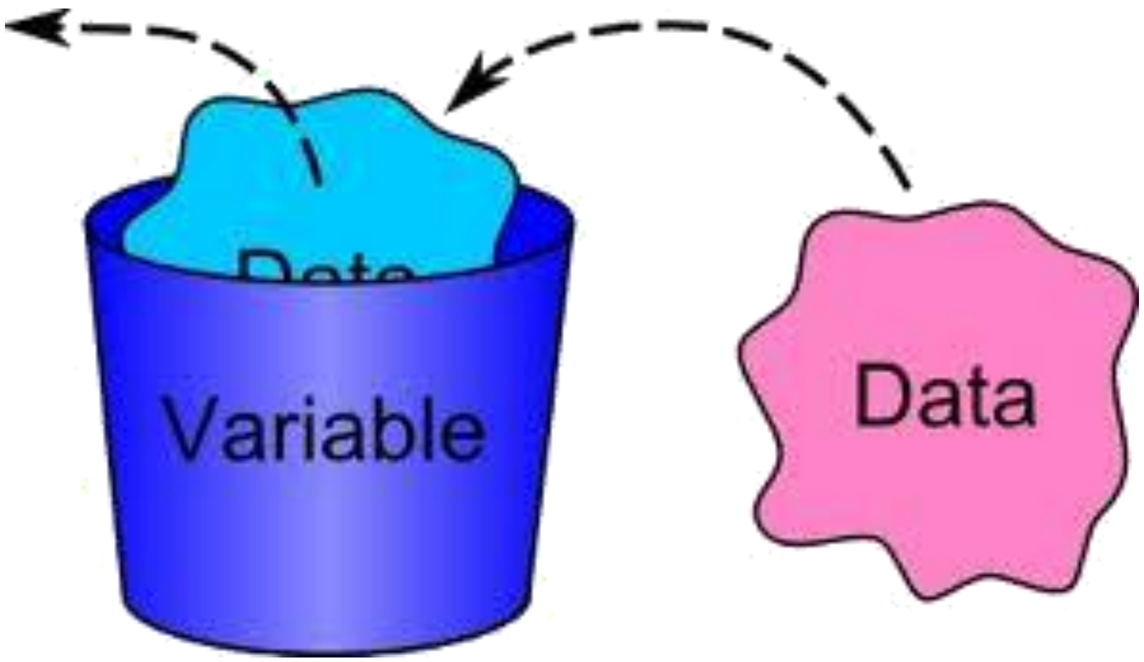
Compiler specific keywords

Some commonly used keywords are given below:

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

Variables

Variables are data storage locations in the computer's memory.



Variables

- **Variables are the symbolic names for storing computational data.**

- **Variable: a *symbolic name* for a memory location**

- **In C variables have to be *declared* before they are used**

  Ex: int x

- **A variable may take different values at different times during execution.**

- ***Declarations* reserve storage for the variable.**

- **Value is assigned to the variable by *initialization* or *assignment***

Variable declarations

Data type Variable name

Which data types are possible in C ?

Which variable names are allowed in C ?

Variable Names- Identifiers

- Symbolic names can be used in C for various data items used by a programmer.
- A symbolic name is generally known as an identifier. An identifier is a name for a variable, constant, function, etc.
- The identifier is a sequence of characters taken from C character set.

Variable names

Rules for valid variable names (*identifiers*) :

- Name must begin with a letter or underscore (_) and can be followed by any combination of letters, underscores, or digits.
- Key words cannot be used as a variable name.
- C is case-sensitive: sum, Sum, and SUM each refer to a different variable.
- Variable names can be as long as you want, although only the first 63 (or 31) characters might be significant.
- Choice of meaningful variable names can increase the readability of a program

Variable names

- Examples of *valid* variable names:
  1) Sum
  2) _difference
  3) a
  4) J5x7
  5) Number_of_moves

- Examples of *invalid* variable names:
  1) sum$value
  2) 3val
  3) int

Declaring variables

- **C imposes to declare variables before their usage.**
- **Advantages of variable declarations:**
  - **Putting all the variables in one place makes it easier for a reader to understand the program.**
  - **Thinking about which variables to declare encourages the programmer to do some planning before writing a program.**
  - **The obligation to declare all variables helps prevent bugs of misspelled variable names.**
  - **Compiler knows the amount of memory needed for storing the variable.**
  - **Compiler can verify that operations done on a variable are allowed by its type.**

Primary (built-in or Basic)Data types

INTEGER

SIGNED TYPE
INT
SHORT INT
LONG INT

UNSIGNED TYPE
UNSIGNED INT
UNSIGNED SHORT INT
UNSIGNED LONG INT

CHARACTER

SIGNED CHARACTER
UNSIGNED CHARACTER

FLOATING POINT TYPE

FLOATING POINT TYPE
FLOAT
DOUBLE
LONG DOUBLE

VOID

VOID

Data types

Basic data types: int, float, double, char, and void.

✓**int:** can be used to store integer numbers (values with no decimal places).

✓**float:** can be used for storing floating-point numbers (values containing decimal places).

✓**double:** the same as type float, and roughly twice the size of float.

✓**char:** can be used to store a single character, such as the letter *a*, the digit character *6*, or a semicolon.

✓ **void:** is used to denote nothing or empty.

Integer Types

- The basic integer type is **int**
  - The size of an **int** depends on the machine and on PCs it is normally 16 or 32 or 64 bits.
- **modifiers (type specifiers)**
  - **short**: typically uses less bits
  - **long**: typically uses more bits
  - **Signed:** both negative and positive numbers
  - **Unsigned:** only positive numbers

SIZE AND RANGE OF VALUES FOR 16-BIT MACHINE (INTEGER TYPE)

	Type	Size	Range
short	short int or signed short int	8	-128 to 127
	unsigned int	8	0 to 255
Integer	int or signed int	16	-32,768 to 32,767
	unsigned int	16	0 to 65,535
Long	long int or signed long int	32	-2,147,483,648 to 2,147,483,647
	unsigned long int	32	0 to 4,294,967,295

The `char` type

- A `char` variable can be used to store a single character.
- A character constant is formed by enclosing the character within a pair of single quotation marks. Valid examples: 'a' .
- Character zero ('0') is not the same as the number (integer constant) 0.
- The character constant '\n'—the newline character—is a valid character constant. It is called as an escape character.
- There are other *escape sequences* like, \t for tab, \v for vertical tab, \n for new line etc.

Character Types

- Character type **char** is related to the integer type.
- Modifiers(type specifiers) ***unsigned*** and ***signed*** can be used
  - **char →1 byte (-128 to 127)**
  - **signed char →1 byte (-128 to 127)**
  - **unsigned char →1 byte (0 to 255)**
- ASCII (American Standard Code for Information Interchange) is the dominant encoding scheme for characters.
  - Examples
    - ✓ **' '** encoded as 32 **'+'** encoded as 43
    - ✓ **'A'** encoded as 65 **……………………'Z'** encoded as 90
    - ✓ **'a'** encoded as 97 **…………………….** **'z'** encoded as 122
    - ✓ **'0'** encoded as 48 **……………………..'9'** encoded as 57

Assigning values to char

```
char letter;      /* declare  variable letter of type char */

letter = 'A';       /* OK */
letter = A;        /* NO! Compiler thinks A is a variable */
letter = "A";     /* NO! Compiler thinks "A" is a string */
letter = 65;      /* ok because characters are internally stored as
                     numeric values (ASCII code) */
```

Floating-Point Types

- Floating-point types represent real numbers
  - Integer part
  - Fractional part
- The number 108.1517 breaks down into the following parts
  - 108 - integer part
  - 1517 - fractional part
- Floating-point constants can also be expressed in *scientific notation*. The value

  1.7e4 represents the value 1.7×10^4.

  The value before the letter e is known as the ***mantissa***, whereas the value that follows e is called the ***exponent***.
- There are three floating-point type specifiers
  - float
  - double
  - long double

SIZE AND RANGE OF VALUES FOR 16-BIT MACHINE (FLOATING POINT TYPE)

	Type	Size
Single Precision	Float	32 bits 4 bytes
Double Precision	double	64 bits 8 bytes
Long Double Precision	long double	80 bits 10 bytes

void

- **2 uses of void are**
  - **To specify the return type of a function when it is not returning any value.**
  - **To indicate an empty argument list to a function.**

S u m m a r y

- **We have learnt about**
  - General Structure of C program
  - C Tokens
  - Variables
  - Declarations
  - Data Types and Sizes

O b j e c t i v e s

- To learn and appreciate
  - Arithmetic Operators
  - Relational and Logical Operators
  - Type conversions
  - Increment and Decrement Operators
  - Bitwise Operators
  - Assignment Operators and Conditional Expressions
  - Precedence and Order of Evaluation

Session outcome

At the end of session student will be able to learn and understand

- Arithmetic Operators
- Relational and Logical Operators
- Type conversions
- Increment and Decrement Operators
- Bitwise Operators
- Assignment Operators and Conditional Expressions
- Precedence and Order of Evaluation

Operators

- The different operators are:
    - Arithmetic
    - Relational
    - Logical
    - Increment and Decrement
    - Bitwise
    - Assignment
    - Conditional

Arithmetic Operators

- The binary arithmetic operators are +, -, *, / and the modulus operator %.
- The / operator when used with integers truncates any fractional part i.e. E.g. 5/2 = 2 and not 2.5
- Therefore % operator produces the remainder when 5 is divided by 2 i.e. 1
- The % operator cannot be applied to float or double
- E.g. x % y wherein % is the operator and x, y are operands

The unary minus operator

```
#include <stdio.h>
int main ()
{
  int a = 25;
  int b = -2;
  printf("%d\n",-a);
  printf("%d\n",-b);
  return 0;
}
```

Working with arithmetic expressions

- Basic arithmetic operators: `+, -, *, /, %`
- **Precedence**: One operator can have a higher priority, or *precedence*, over another operator. The operators within C are grouped hierarchically according to their ***precedence*** (i.e., order of evaluation)
  - ➢ Operations with a higher precedence are carried out before operations having a lower precedence.

    High priority operators   * / %

    Low priority operators   + -

  - Example: * has a higher precedence than +

    a + b * c → a+(b*c)

  - ➢ If necessary, you can always use parentheses in an expression to force the terms to be evaluated in any desired order.
- **Associativity**: Expressions containing operators of the same precedence are evaluated either from left to right or from right to left, depending on the operator. This is known as the ***associative*** property of an operator.
  - Example: + has a *left to right* associativity

  For both the precedence group described above, ***associativity is "left to right"***.

Working with arithmetic expressions

```c
#include <stdio.h>
int main ()
{
   int a = 100;
   int b = 2;
   int c = 25;
   int d = 4;
   int result;
   result = a * b + c * d;
   printf(“ Result1: %d\n”,result);
   result = a * (b + c * d);
   printf(“ Result2: %d\n”,result);
   return 0;
}
```

```
Result1: 300
Result2: 10200
```

Relational operators

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

The relational operators have lower precedence than all arithmetic operators:
`a < b + c` **is evaluated as** `a < (b + c)`

ATTENTION !
the “is equal to” operator == and the “assignment” operator =

Relational operators

- An expression such as a < b containing a relational operator is called a *relational expression*.

- The value of a relational expression is one, if the specified relation is true and zero if the relation is false.

  E.g.:

  10 < 20 is TRUE
  20 < 10 is FALSE

- A simple relational expression contains only one relational operator and takes the following form.

  ae1 relational operator ae2

  ae1 & ae2 are arithmetic expressions, which may be simple constants, variables or combinations of them.

Relational operators

The arithmetic expressions will be evaluated first & then the results will be compared. That is, arithmetic operators have a higher priority over relational operators. > >= < <= all have the same precedence and below them are the next precedence equality operators i.e. == and !=

Suppose that i, j and k are integer variables whose values are 1, 2 and 3 respectively.

Expression	Interpretation	Value
`i<j`	`true`	`1`
`(i+j)>=k`	`true`	`1`
`(j+k)>(i+5)`	`false`	`0`
`k!=3`	`false`	`0`
`j==2`	`true`	`1`

Logical operators

Truth Table

op-1	op-2	value of 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

Operator	Symbol	Example
AND	&&	expression1 && expression2
OR	||	expression1 || expression2
NOT	!	!expression1

The result of logical operators is always either 0 (FALSE) or 1 (TRUE)

Logical operators

Expressions	Evaluates As
`(5 == 5)&&(6 != 2)`	True (1) because both operands are true
`(5 > 1) || (6 < 1)`	True (1) because one operand is true
`(2 == 1)&&(5 ==5)`	False (0) because one operand is false
`! (5 == 4)`	True (1) because the operand is false
`!(FALSE) = TRUE` `!(TRUE) = FALSE`	

Increment and Decrement operators (++ and --)

- The operator **`++`** adds 1 to the operand.
- The operator **`--`** subtracts 1 from the operand.
- Both are **`unary operators`**.
- Ex: **`++i`** or **`i++`** is equivalent to **`i=i+1`**
- **They behave differently when they are used in expressions on the R.H.S of an assignment statement.**

Increment and Decrement operators

Ex:

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

Prefix Mode

In this case, the value of y and m would be 6.

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

Postfix Mode

Here y continues to be 5. Only m changes to 6.

Prefix operator ++ appears before the variable.

Postfix operator ++ appears after the variable.

Increment and Decrement operators

Don’ts:

Attempting to use the increment or decrement operator on an expression other than a modifiable variable name or reference.

Example:

`++(5)` ***is a syntax error***

`++(x + 1)` ***is a syntax error***

Bitwise Operators

- Bitwise Logical Operators
- Bitwise Shift Operators
- Ones Complement operator

Bitwise Logical operators

- **`&(AND),|(OR),^(EXOR)`**
- These are *binary operators* and require two integer operands.
- These work on their operands bit by bit starting from LSB (rightmost bit).

op 1	op 2	&	|	^
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Example

Suppose x = 10, y = 15

z = x **&** y sets z=10 like this

0000000000001010 ← x

0000000000001111 ← y

0000000000001010 ← z = x & y

Same way **|** , **^** according to the table are computed.

Bitwise Shift operators

- `<<`, `>>`

- These are used to move bit patterns either to the left or right.

- They are used in the following form

- **`op<<n`** or **`op>>n`** here op is the operand to be shifted and n is number of positions to shift.

Bitwise Shift operator: <<

- << causes all the bits in the operand op to be shifted to the left by n positions.

- The *leftmost* n bits in the original bit pattern will be lost and the *rightmost* n bits that are vacated are filled with 0’s

Bitwise Shift operator: >>

- >> causes all the bits in operand op to be shifted to the right by n positions.

- The *rightmost* n bits will be lost and the left most vacated bits are filled with 0’s if number is unsigned integer

Examples

- Suppose X is an unsigned integer whose bit pattern is 0000 0000 0000 1011

✓ x<<1 0000 0000 0001 0110 ⬅ **Add ZEROS**

✓ x>>1 **Add ZEROS** ➡ 0000 0000 0000 0101

Examples

- Suppose X is an unsigned integer whose bit pattern is 0000 0000 0000 1011 whose equivalent value in decimal number system is 11.

✓x<<3 0000 0000 0101 1000 = 88 Add ZEROS

✓x>>2 Add Z EROS 0000 0000 0000 0010 = 2

Note:

✓**x=y<<1;** same as x=y*2 (Multiplication)

✓**x=y>>1;** same as x=y/2 (Division)

Bitwise Shift operators

- Op and n can be constants or variables.

- There are 2 restrictions on the value of n
  - ✓ n cannot be –ve
  - ✓ n should not be greater than number of bits used to represent Op.(E.g.: suppose op is ***int*** and size is 2 bytes then $\boldsymbol{n}$ cannot be greater than 16).

Bitwise complement operator

- The complement operator(**~**) is an *unary operator* and inverts all the bits represented by its operand.

- Suppose x=1001100010001111

  ~x=0110011101110000 (**`complement`**)

- Also called as 1’s complement operator.

Type Conversions in Expressions

- C permits mixing of constants and variables of different types in an expression

- C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance.

- This automatic conversion is known as **implicit type conversion**

- The table in the next slide gives the implicit type conversions

Type Conversions in Expressions

The following are the sequence of rules that are applied while evaluating expressions

Lower Type Operands	Higher Type Operands
Short or Char	int
One operand is Long double, the other will be converted to long double	Result is also long double
One operand is double, the other will be converted to double	Result is also double
One operand is float, the other will be converted to float	Result is also float
One operand is unsigned Long int, the other will be converted to unsigned long int	Result is also unsigned long int

Type Conversions in Expressions

Lower Type Operands	Higher Type Operands
One operand 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 b) Else both operands will be converted to unsigned long int	a) Result will be long int b) Result will be unsigned long int
One operand is Long int, the other will be converted to long int	Result is also long int

Type Conversions in Expressions

- The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it
- However the following changes are introduced during the final assignment
  - Float to int causes truncation of the fractional part
  - Double to float caused rounding of digits
  - Long int to int causes dropping of the excess higher order bits

Type Conversions in Expressions

Explicit type conversion or type casting

- There are instances when we want to force a type conversion in a way that is different from the automatic conversion

  E.g. `ratio=57/67`

- Since 57 and 67 are integers in the program , the decimal part of the result of the division would be lost and ratio would represent a wrong figure

- This problem can be solved by converting locally as one of the variables to the floating point as shown below:

```
ratio= (float) 57/67
```

Type Conversions in Expressions

- The operator (float) consider 57 for computation to floating point then using the rule of automatic conversion
- The division is performed in floating point mode, thus retaining the fractional part of result
- The process of such a **local conversion** is known as **`explicit conversion`** or **`casting a value`**

The Type Cast Operator

The general form of a type casting is

```
(type-name) expression
```

```
int a =150;
float f;   f = (float) a / 100; // type cast operator
```

- The type cast operator has the effect of converting the value of the variable 'a' to type float for the purpose of evaluation of the expression.
- This operator does NOT permanently affect the value of the variable 'a';
- The type cast operator has a higher precedence than all the arithmetic operators except the unary minus and unary plus.
- Examples of the use of type cast operator:

  (int) 29.55 + (int) 21.99 results in **29 + 21**

  (float) 6 / (float) 4 results in **1.5**

  (float) 6 / 4 results in **1.5**

Type Conversions in Expressions

Example	Action
`x=(int) 7.5`	7.5 is converted to integer by truncation
`a=(int) 21.3/(int)4.5`	Evaluated as 21/4 and the result would be 5
`b=(double)sum/n`	Division is done in floating point mode
`y=(int)(a+b)`	The result of a+b is converted to integer
`z=(int)a+b`	a is converted to integer and then added to b
`p= cos((double)x)`	Converts x to double before using it

Integer and Floating-Point Conversions

- Assign an integer value to a floating variable: does not cause any change in the value of the number; the value is simply converted by the system and stored in the floating format.
- Assign a floating-point value to an integer variable: the decimal portion of the number gets truncated.
- Integer arithmetic (division):
  - `int divided  by int` => result is integer division
  - `int divided by float or float divided by int` => result is real division (floating-point)

Integer and Floating-Point Conversions

```
#include <stdio.h>
int main ()
{           float f1 = 123.125, f2;
            int i1, i2 = -150;
            i1 = f1;  // float to integer conversion
            printf ("float assigned to int produces");
            printf("%d\n",i1);
            f2 = i2;  // integer to float conversion
            printf("integer assigned to float produces");
            print("%.2f\n",f2);
            i1 = i2 / 100;  // integer divided by integer
            printf("integer divided  by integer produces");
            printf("%d\n",i1);
            f1 = i2 / 100.0;  // integer divided by a float
            printf("integer divided  by float produces");
            printf("%.2f\n",f1);
            return 0;
}
```

123

-150.00

-1

-1.50

The assignment operators

- **The C language permits you to join the arithmetic operators with the assignment operator using the following general format: op=, where op is an arithmetic operator, including +, –, *, /, and %.**

- **Example:**

```
count += 10;
```

Equivalent to:

```
count=count+10;
```

- **Example: precedence of op=:**

```
a /= b + c
```

- **Equivalent to:**

```
a = a / (b + c)
```

The conditional operator (? :)

```
condition ? expression1 : expression2
```

- *condition* is an expression that is evaluated first.
- If the result of the evaluation of *condition* is TRUE (nonzero), then *expression1* is evaluated and the result of the evaluation becomes the result of the operation.
- If *condition* is FALSE (zero), then *expression2* is evaluated and its result becomes the result of the operation.

```
maxValue = ( a > b ) ? a : b;
```

Equivalent to:

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

Comma (,) operator

- The coma operator is used basically to separate expressions.

    i = 0 , j = 10; // in initialization [l → r]

- The meaning of the comma operator in the general expression `e1 , e2` is

"evaluate the sub expression *e1*, then evaluate *e2*; the value of the expression is the value of *e2* ".

Operator precedence & Associativity

Operator Category	Operators	Associativity
Unary operators	+ – ++ – – ~ !	R→L
Arithmetic operators	* / %	L→R
Arithmetic operators	+ –	L→R
Bitwise shift left Bitwise shift right	<< >>	L→R
Relational operators	< <= > >=	L→R
Equality operators	== !=	L→R
Bitwise AND, XOR, OR	& ^ |	L→R
Logical and	&&	L→R
Logical or	||	L→R
Assignment operator	= += – = *= /= %=	R→L

Summary of Operators – detailed precedence

Precedence	Operator	Description	Associativity
1 highest	::	Scope resolution	None
2	++	Suffix increment	Left-to-right
	--	Suffix decrement	
	()	Parentheses (Function call)	
	[]	Brackets (Array subscripting)	
	.	Element selection by reference	
	->	Element selection through pointer	
3	++	Prefix increment	Right-to-left
	--	Prefix decrement	
	+	Unary plus	
	-	Unary minus	
	!	Logical NOT	
	~	Bitwise NOT (One's Complement)	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
4	.*	Pointer to member	Left-to-right
	->*	Pointer to member	
5	*	Multiplication	Left-to-right
	/	Division	
	%	Modulo (remainder)	
6	+	Addition	Left-to-right
	-	Subtraction	

Precedence	Operator	Description	Associativity
7	<<	Bitwise left shift	Left-to-right
	>>	Bitwise right shift	
8	<	Less than	Left-to-right
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
9	==	Equal to	Left-to-right
	!=	Not equal to	
10	&	Bitwise AND	Left-to-right
11	^	Bitwise XOR (exclusive or)	Left-to-right
12	|	Bitwise OR (inclusive or)	Left-to-right
13	&&	Logical AND	Left-to-right
14	||	Logical OR	Left-to-right
15	?:	Ternary conditional	Right-to-left
16	=	Direct assignment	Right-to-left
	+=	Assignment by sum	
	-=	Assignment by difference	
	*=	Assignment by product	
	/=	Assignment by quotient	
	%=	Assignment by remainder	
	<<=	Assignment by bitwise left shift	
	>>=	Assignment by bitwise right shift	
	&=	Assignment by bitwise AND	
	^=	Assignment by bitwise XOR	
	|=	Assignment by bitwise OR	
17 lowest	,	Comma	Left-to-right

Example:

Show all the steps how the following expression is evaluated. Consider the initial values of i=8, j=5.

```
2*((i/5)+(4*(j-3))%(i+j-2))
```

Example solution:

```
2*((i/5)+(4*(j-3))%(i+j-2))    i→8, j→5
```

```
2*((8/5)+(4*(5-3))%(8+5-2))
2*(1+(4*2)%11)
2*(1+8%11)
2*(1+8)
2*9
18
```

Operator precedence & Associativity

Ex: `(x==10 + 15 && y < 10)`

Assume x=20 and y=5

Evaluation:

`+`	`(x==25 && y< 10)`
`<`	`(x==25 && true)`
`==`	`(False && true)`
`&&`	`(False)`

Tutorial Problems

- Suppose that a=2, b=3 and c=6, What is the answer for the following: (a==5)
  (a * b > =c)
  (b+4 > a *c)
  ((b=2)==a)
- Evaluate the following:
  1. ((5 == 5) && (3 > 6))
  2. ((5 == 5) || (3 > 6))
  3. 7==5 ? 4 : 3
  4. 7==5+2 ? 4 : 3
  5. 5>3 ? a : b
  6. K = (num > 5 ? (num <= 10 ? 100 : 200) : 500); where num =30
- In b=6.6/a+(2*a+(3*c)/a*d)/(2/n); which operation will be performed first.
- If a is an integer variable, a=5/2; will return a value
- The expression, a=7/22*(3.14+2)*3/5; evaluates to
- If a is an Integer, the expression a = 30 * 1000 + 2768; evaluates to

Tutorial Problems

1. Suppose that a=2, b=3 , c=6 and d=5 What is the answer for the following:
   i. (a==5) → 0
   ii. (a * b > =c) → 1
   iii. (b+4 > a *c) → 0
   iv. ((b=2)==a) → 1
2. Evaluate the following:
   i. ((5 == 5) && (3 > 6)) → 0
   ii. ((5 == 5) || (3 > 6)) → 1
   iii. 7==5 ? 4 : 3 → 3
   iv. 7==5+2 ? 4 : 3 → 4
   v. 5>3 ? a : b → 2
   vi. K = (num > 5 ? (num <= 10 ? 100 : 200) : 500); where num =30 → 200
3. In b=6.6/a+(2*a+(3*c)/a*d)/(2/n); which operation will be performed first.
4. If a is an integer variable, a=5/2; will return a value → 2
5. The expression, a=7/22*(3.14+2)*3/5; evaluates to → 0
6. If a is an Integer, the expression a = 30 * 1000 + 2768; evaluates to → 32768

S u m m a r y

We have learnt about

- **Arithmetic Operators**
- **Relational and Logical Operators**
- **Type conversions**
- **Increment and Decrement Operators**
- **Bitwise Operators**
- **Assignment Operators and Conditional Expressions**
- **Precedence and Order of Evaluation**