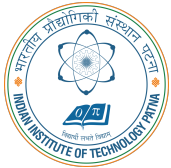
# CS1101: Foundations of Programming

## Data types, operators, expression




**Dept. of Computer Science & Engineering**

**Indian Institute of Technology Patna**

# Data types

- Variables and constants are the basic data objects manipulated in program
- There are only four basic data types in C
  - `char` – character data, typically occupies 1 byte (8 bits)
  - `int` – integer data, typically occupies 4 bytes (32 bits)
  - `float` – single precision floating point (real) numbers
    - typically needs 4 bytes, stores 7 decimal points
  - `double` – double precision floating point (real) numbers,
    - typically needs 8 bytes, stores 15 decimal points

# Other qualifiers for data types

- There can be other qualifiers for datatype
  - `short, long, signed, unsigned`
  - **Example:** `short int, unsigned int, long int`
  - `short` **and** `long` **provide different lengths of integers**
    - `short` **needs 2 bytes**
    - `long` **is at least 4 bytes, depends on machine specification**
  - `signed` **and** `unsigned` **are applicable for** `int` **and** `char`
  - `unsigned` **— always positive or zero**

# Constants

- **Integer constant consists of digits without any other character in between, can have +/- in the beginning**
  - **Valid:** `1234, -4321, 123456789L,` **Invalid:** `1,234,534`
  - `l` **or** `L` **to denote long constant,** `u` **or** `U` **for unsigned**
  - `UL` **to denote** `unsigned long`
- **Floating point constants - two different notations**
  - **Decimal notation:** `12.34, -4.321, 0.0004`
  - **Exponential notation:** `1e-2, 0.12e-3, 4.56e12`
  - `l` **or** `L` **to denote long double,** `f` **or** `F` **for float, default type is** `double` **unless suffixed**

# Constants

- **Single character constant is an integer written as one character within single quotes**
  - **Example:** `'x', 'a', 'Z', '1', '+'` **etc.**
  - **There are special backslash characters**
    - `'\n'` **new line**
    - `'\t'` **horizontal tab**
    - `'\''` **single quote**
    - `'\"'` **double quote**
    - `'\\'` **backslash**
    - `'\0'` **null**
- **String constant - sequence of characters (letters, numbers, special characters, blank spaces) enclosed in double quotes**
  - **Example:** `"good", "IIT Patna", "9+16", "T"`

# Constants

- **What is the difference between `'T'` and `"T"` or equivalent?**

# Constants

- What is the difference between `'T'` and `"T"` or equivalent?
  - `'T'` is equal to some integer while `"T"` is not
  - They are not equivalent

# Constants

- What is the difference between `'T'` and `"T"` or equivalent?
  - `'T'` is equal to some integer while `"T"` is not
  - They are not equivalent
- In C, each character is represented by 1 byte an integer
  - Example: `'0'` has value 48, ...`9` has 57
  - Example: `'A'` has value 65, ...`'Z'` has 90
  - Example: `'a'` has value 97, ...`'z'` has 122
- Example

```
char var='A'
printf("%c %d", var, var);
```

# Constants

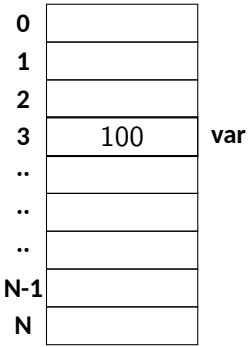
- What is the difference between `'T'` and `"T"` or equivalent?
  - `'T'` is equal to some integer while `"T"` is not
  - They are not equivalent
- In C, each character is represented by 1 byte an integer
  - Example: `'0'` has value 48, ...9 has 57
  - Example: `'A'` has value 65, ...`'Z'` has 90
  - Example: `'a'` has value 97, ...`'z'` has 122
- Example

```
char var='A'
printf("%c %d", var, var);
```

Same values will be printed once as a character and second time as integer

# Variable value and variable address

- **Consider** `int var=100; var = var + 2;`
  - **In an expression** `var` **refers to the content of the memory location where it is stored**
  - `&var` **refers to the memory address**
  - `var` **refers to 100**
  - `&var` **refers to 3**
- **Example**
  - `scanf("%d", &var);`
  - `printf("%d", var);`



# Assignment statement

- Used to assign values to variable using assignment operator (=)
  - syntax: `var_name = expression`
  - Left of = is known as l-value, it must be a modifiable variable
  - Right of = is known as r-value, it can be any expression
- Example

```
var = 100;
z = 40 * 3 * 20;
V = n * R * T / P;
d = u * t + 0.5 * f * t ;
a = b = c = 10;
int a = 10;
```

# Types of l-value and r-value

- Usually type of these two should be the same
- If not, the type of r-value will be converted to the type of l-value internally and then assigned to LHS
- Examples:
  - `double var; var = 5 * 7;`
  - Type r-value is int and value is 35
  - Type of l-value is double, so it stores 35.0
- Example
  - `int var; var = 5.5 * 7;`
  - Type r-value is double and value is 38.5
  - Type of l-value is int, so it stores 38

# Operators

- The operators can broadly be classified as
  - Arithmetic operators — deal with numerical operands
  - Relational operators — outcome is either true or false
  - Logical operators — primarily takes boolean inputs
  - Assignment operator — assigning values to variables
  - Unary operator — negation of a variable, etc.
  - Conditional operators — similar to `if-else` statement

# Arithmetic operators

- List of arithmetic operators
  - `+` – Addition
  - `-` – Subtraction
  - `*` – Multiplication
  - `/` – Division
  - `%` – Remainder after integer division
- Example

```
w = f * d;
y = m * x + c;
y = a * x * x - b * x - c;
i = v / r;
rem = dividend % 2;
```

# Arithmetic operators

- **List of arithmetic operators**
  - **+ – Addition**
  - **− – Subtraction**
  - **∗ – Multiplication**
  - **/ – Division**
  - **% – Remainder after integer division**
- **Example**

```
w = f * d;
y = m * x + c;
y = a * x * x - b * x - c;
i = v / r;
rem = dividend % 2;
```

Suppose:

```
int x = 23, y = 5;
```

| | |
|---|---|
| `x + y` | 28 |
| `x - y` | 18 |
| `x * y` | 115 |
| `x / y` | 4 |
| `x % y` | 3 |

**All operators except % can be used with int, float, double, char operands. % can be used with int only**

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

```
a+b*c-d/e
```

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

`a+b*c-d/e` $\rightarrow$ `a+(b*c)-(d/e)`

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

```
a+b*c-d/e      →      a+(b*c)-(d/e)
a*-b+d%e-f
```

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

`a+b*c-d/e` $\rightarrow$ `a+(b*c)-(d/e)`

`a*-b+d%e-f` $\rightarrow$ `a*(-b)+(d%e)-f`

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

```
a+b*c-d/e      →     a+(b*c)-(d/e)
a*-b+d%e-f     →      a*(-b)+(d%e)-f
a-b+c-d
```

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

```
a+b*c-d/e      →     a+(b*c)-(d/e)
a*-b+d%e-f     →     a*(-b)+(d%e)-f
a-b+c-d        →     (((a-b)+c)-d)
```

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

```
a+b*c-d/e      →     a+(b*c)-(d/e)
a*-b+d%e-f     →     a*(-b)+(d%e)-f
a-b+c-d        →     (((a-b)+c)-d)
x*y*z
```

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

```
a+b*c-d/e     →     a+(b*c)-(d/e)
a*-b+d%e-f    →     a*(-b)+(d%e)-f
a-b+c-d       →     (((a-b)+c)-d)
x*y*z         →     ((x*y)*z)
```

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

```
a+b*c-d/e      →     a+(b*c)-(d/e)
a*-b+d%e-f     →     a*(-b)+(d%e)-f
a-b+c-d        →     (((a-b)+c)-d)
x*y*z          →     ((x*y)*z)
a+b+c*d*e
```

# Arithmetic operator precedence

- Decreasing order of priority
  - Parentheses :: `()`
  - Unary minus :: `-3`
  - Multiplication, division, modulus
  - Addition and subtraction
- For operators of same priority, evaluation is from left to right as they appear
- Example

```
a+b*c-d/e     →     a+(b*c)-(d/e)
a*-b+d%e-f    →     a*(-b)+(d%e)-f
a-b+c-d       →     (((a-b)+c)-d)
x*y*z         →     ((x*y)*z)
a+b+c*d*e     →     (a+b)+((c*d)*e)
```

# Unary operator

- Operators that act on a single operand to produce new value
- Usually, such operators precede their single operand (for some operators it can be written after their operand)
- Unary minus: `-743`, `-3*(x+y)`, `-(x+y)`, etc.
- We will see other unary operators later

# Integer, real, mixed mode arithmetic

- **Integer arithmetic - expression involved with integers only and produces integer**
  - **Example:** `23 / 5` $\rightarrow$ **4**
- **Real arithmetic - expression involved with real numbers only**
  - **Example:** `1.0 / 6.0 * 6.0` $\rightarrow$ **0.9999**
  - **Floating point values rounded to the the number of significant digits permissible**
- **Mixed-mode arithmetic - expression involved with both real and integer numbers**
  - **Example:** `23 / 5` $\rightarrow$ **4**
  - **Example:** `23.0 / 5.0` $\rightarrow$ **4.6**

# Implicit type conversion

- When an operator has operand of different types, they are converted to a common type
- Automatic conversion convert narrower operand to wider one without losing information
- Converting an integer to floating point in an expression `f + i`
- Information can be lost if longer integer type is assigned to a shorter one

# Similar code but different results

```
int a=32, b=5, c;
float z;
c = a / b;
z = a / b;
```

# Similar code but different results

```
int a=32, b=5, c;
float z;
c = a / b;
z = a / b;
```

**Output:**
**Value of c will be 6**
**Value of z will be 6.0**
**We want 6.4 to be stored in z**

# Explicit type conversion: Typecasting

```
int a=32, b=5, c;
float z;
c = a / b;
z = ((float) a) / b;
```

# Explicit type conversion: Typecasting

```
int a=32, b=5, c;
float z;
c = a / b;
z = ((float) a) / b;
```

- **Typecast:** `(type name) expression`
- `expression` **is converted to the named type**
- **Unary operator**

- **Output:**
- **Now** `z` **will store 6.4**
- **Type of** `a` **will be float**
- **Mixed mode arithmetic will be performed**

# Restriction on typecasting

- Not everything can be typecast to anything
- Use typecast carefully, many times compiler issue no error
- `float / double` should not be typecast to `int`
- `int` should not be typecast to `char`

# Average of 2 integers

```
int a, b;
float avg;
scanf("%d%d",&a, &b);
avg = (a + b) / 2;
printf("%f",avg);
```

# Average of 2 integers

```
int a, b;
float avg;
scanf("%d%d",&a, &b);
avg = (a + b) / 2;
printf("%f",avg);
```

```
int a, b;
float avg;
scanf("%d%d",&a, &b);
avg =((float) (a + b)) / 2;
printf("%f",avg);
```

# Average of 2 integers

```
int a, b;
float avg;
scanf("%d%d",&a, &b);
avg = (a + b) / 2;
printf("%f",avg);
```

```
int a, b;
float avg;
scanf("%d%d",&a, &b);
avg =((float) (a + b)) / 2;
printf("%f",avg);
```

```
int a, b;
float avg;
scanf("%d%d",&a, &b);
avg = (a + b) / 2.0;
printf("%f",avg);
```

# Average of 2 integers

**Incorrect**

```
int a, b;
float avg;
scanf("%d%d",&a, &b);
avg = (a + b) / 2;
printf("%f",avg);
```

**Correct**

```
int a, b;
float avg;
scanf("%d%d",&a, &b);
avg =((float) (a + b)) / 2;
printf("%f",avg);
```

**Correct**

```
int a, b;
float avg;
scanf("%d%d",&a, &b);
avg = (a + b) / 2.0;
printf("%f",avg);
```

# Other assignment operator

- `+=, -=, *=, /=, %=`
- **These are shorthand notation,** `a += b` $\rightarrow$ `a = a + b`
- **Similarly for the other operators**
- **Suppose** `a` **and** `b` **are two integer variables having values 12 and 3 respectively**
  - `a += b` $\rightarrow$ **15 will be stored in** `a`, `a = a + b`
  - `a -= b` $\rightarrow$ **9 will be stored in** `a`, `a = a - b`
  - `a *= b` $\rightarrow$ **36 will be stored in** `a`, `a = a * b`
  - `a %= b` $\rightarrow$ **0 will be stored in** `a`, `a = a % b`

# Increment (++) & Decrement (−−) operators

- ++ adds 1 to its operand, −− subtracts 1
- Unusual aspect is that they can be used as prefix / suffix operator
  - Example: `++n, n++, --n, n--`
  - `n++, ++n` — `n` will be incremented by 1, `++n` increments `n` before using it, `n++` increments `n` after using it
- Example: Let `a = 13, b = 7`
  - `x = 5 + ++a`

# Increment (++) & Decrement (−−) operators

- ++ adds 1 to its operand, −− subtracts 1
- Unusual aspect is that they can be used as prefix / suffix operator
  - Example: `++n, n++, --n, n--`
  - `n++, ++n` — `n` will be incremented by 1, `++n` increments `n` before using it, `n++` increments `n` after using it
- Example: Let `a = 13, b = 7`
  - `x = 5 + ++a` $\rightarrow$ `x = 19, a = 14`

# Increment (++) & Decrement (−−) operators

- ++ adds 1 to its operand, −− subtracts 1
- Unusual aspect is that they can be used as prefix / suffix operator
  - Example: `++n, n++, --n, n--`
  - `n++, ++n` — `n` will be incremented by 1, `++n` increments `n` before using it, `n++` increments `n` after using it
- Example: Let `a = 13, b = 7`
  - `x = 5 + ++a` $\rightarrow$ `x = 19, a = 14`
  - `x = 5 + a++`

# Increment (++) & Decrement (−−) operators

- ++ adds 1 to its operand, −− subtracts 1
- Unusual aspect is that they can be used as prefix / suffix operator
  - Example: `++n, n++, --n, n--`
  - `n++, ++n` — `n` will be incremented by 1, `++n` increments `n` before using it, `n++` increments `n` after using it
- Example: Let `a = 13, b = 7`
  - `x = 5 + ++a` → `x = 19, a = 14`
  - `x = 5 + a++` → `x = 18, a = 14`

# Increment (++) & Decrement (−−) operators

- ++ adds 1 to its operand, −− subtracts 1
- Unusual aspect is that they can be used as prefix / suffix operator
  - Example: `++n, n++, --n, n--`
  - `n++, ++n` — `n` will be incremented by 1, `++n` increments `n` before using it, `n++` increments `n` after using it
- Example: Let `a = 13, b = 7`
  - `x = 5 + ++a` → `x = 19, a = 14`
  - `x = 5 + a++` → `x = 18, a = 14`
  - `x = a++ + --b`

# Increment (++) & Decrement (−−) operators

- ++ adds 1 to its operand, −− subtracts 1
- Unusual aspect is that they can be used as prefix / suffix operator
  - Example: `++n, n++, --n, n--`
  - `n++, ++n` — `n` will be incremented by 1, `++n` increments `n` before using it, `n++` increments `n` after using it
- Example: Let `a = 13, b = 7`
  - `x = 5 + ++a` → `x = 19, a = 14`
  - `x = 5 + a++` → `x = 18, a = 14`
  - `x = a++ + --b` → `x = 19, a = 14, b = 6`

# Increment (++) & Decrement (−−) operators

- ++ adds 1 to its operand, −− subtracts 1
- Unusual aspect is that they can be used as prefix / suffix operator
  - Example: `++n, n++, --n, n--`
  - `n++, ++n` — `n` will be incremented by 1, `++n` increments `n` before using it, `n++` increments `n` after using it
- Example: Let `a = 13, b = 7`
  - `x = 5 + ++a` → `x = 19, a = 14`
  - `x = 5 + a++` → `x = 18, a = 14`
  - `x = a++ + --b` → `x = 19, a = 14, b = 6`
  - `x = a++ - ++a`

# Increment (++) & Decrement (−−) operators

- ++ adds 1 to its operand, −− subtracts 1
- Unusual aspect is that they can be used as prefix / suffix operator
  - Example: `++n, n++, --n, n--`
  - `n++, ++n` — `n` will be incremented by 1, `++n` increments `n` before using it, `n++` increments `n` after using it
- Example: Let `a = 13, b = 7`
  - `x = 5 + ++a` → `x = 19, a = 14`
  - `x = 5 + a++` → `x = 18, a = 14`
  - `x = a++ + --b` → `x = 19, a = 14, b = 6`
  - `x = a++ - ++a` → ?? — called side effects while calculating some values something else get changed. Avoid such situation

# Relational operators

- Used for comparison
  - < :: Less than
  - > :: Greater than
  - <= :: Less than or equal to
  - >= :: Greater than or equal to
  - == :: Equal to
  - != :: Not equal to

# Relational operators

- **Used for comparison**
  - **< :: Less than**
  - **> :: Greater than**
  - **<= :: Less than or equal to**
  - **>= :: Greater than or equal to**
  - **== :: Equal to**
  - **!= :: Not equal to**

```
10 > 20 → false, value is 0
34 < 45 → true, value is non-zero
34 == (30 + 5) → false, value is 0
34 != (30 + 5) → true, value is non-zero
```

# Relational operators

- Used for comparison
  - < :: Less than
  - > :: Greater than
  - <= :: Less than or equal to
  - >= :: Greater than or equal to
  - == :: Equal to
  - != :: Not equal to

```
10 > 20 → false, value is 0
34 < 45 → true, value is non-zero
34 == (30 + 5) → false, value is 0
34 != (30 + 5) → true, value is non-zero
```

- Value corresponding to true is any non-zero value not necessarily 1, false is 0 always
- When arithmetic expression are used on either side of a relation operator, expression will be evaluated first, then the results will be compared

# Logical operators

- There are 3 logical operators / connectives
  - `!` :: unary negation (NOT)
  - `&&` :: logical AND
  - `||` :: logical OR
- These operators act upon operands that are themselves logical expressions
- The final outcome is either true or false
- Unary negation (`!`)
  - Single operand
  - Value is 0 if operand is non-zero, 1 if operand is 0
  - Example: `!(34 == x), !(val == 'Y')`

# Logical operators

- Result of logical AND operation will be true if both operands are true
- Result of logical OR operation will be true if at least one operands is true

# Logical operators

- Result of logical AND operation will be true if both operands are true
- Result of logical OR operation will be true if at least one operands is true

| x | y | x && y | x\|\|y |
|---|---|---|---|
| false | false | false | false |
| false | true | false | true |
| true | false | false | true |
| true | true | true | true |

**Example: assume** `i=7, f=5.5, c='w'`

`(i >= 6) && (c == 'w')` → **true**

`(i >= 6) || (c == 119)` → **true**

`(f < 11) && (i >= 100)` → **false**

`(c != 'p') || i + f <= 100` → **true**

# Logical operators

- Result of logical AND operation will be true if both operands are true
- Result of logical OR operation will be true if at least one operands is true

| x | y | x && y | x\|\|y |
|---|---|---|---|
| false | false | false | false |
| false | true | false | true |
| true | false | false | true |
| true | true | true | true |

**Example: assume** `i=7, f=5.5, c='w'`

`(i >= 6) && (c == 'w')` → **true**

`(i >= 6) || (c == 119)` → **true**

`(f < 11) && (i >= 100)` → **false**

`(c != 'p') || i + f <= 100` → **true**

**Suppose we wish to express that** `a` **should not have the value of 2 or 3. Does the following expression capture this requirement?**

`((a != 2) || (a != 3))`

# Example of logical operators

```
#include<stdio.h>
 int i, j;
 scanf("%d%d",&i,&j);
 printf("%d AND %d = %d, %d OR %d=%d\n",i,j,i&&j,i,j,i||j);
}
```

# Example of logical operators

```
#include<stdio.h>
 int i, j;
 scanf("%d%d",&i,&j);
 printf("%d AND %d = %d, %d OR %d=%d\n",i,j,i&&j,i,j,i||j);
}
```

**Output:**

```
3 0
3 AND 0 = 0, 3 OR 0 = 1
```

# Precedence for operators

| Operator class | Operators | Associativity |
|---|---|---|
| Unary | postfix ++, −− | Left to Right |
| Unary | prefix ++, −− , −, !, & | Right to Left |
| Binary | *, /, % | Left to Right |
| Binary | +, – | Left to Right |
| Binary | <, <=, >, >= | Left to Right |
| Binary | == != | Left to Right |
| Binary | && | Left to Right |
| Binary | \|\| | Left to Right |
| Assignment | =, +=, -=, *=, /=, %= | Right to Left |

# Assignment expression (contd.)

- An assignment expression evaluates to a value same as any other expression
- Value of an assignment expression is the value assigned to the l-value
- Example: value of
  - `a = 3` is 3
  - `b = 2*4 - 6` is 2
  - `n = 2*u + 3*v - w` is whatever the arithmetic expression 2*u + 3*v - w evaluates to given the current values stored in variables u, v, w
- Several variables can be assigned the same value using multiple assignment operators

```
a = b = c = 5;
flag1 = flag2 = 'y';
speed = flow = 0.0;
```

# Assignment expression (contd.)

- Easy to understand if you remember that
  - The assignment expression has a value
  - Multiple assignment operators are right-to-left associative
- Consider a = b = c = 5
  - Three assignment operators
  - Rightmost assignment expression is c=5, evaluates to value 5
  - Now you have a = b = 5
  - Rightmost assignment expression is b=5, evaluates to value 5
  - Now you have a = 5
  - Evaluates to value 5
  - So all three variables store 5, the final value the assignment expression evaluates to is 5

# Assignment expression (contd.)

- **A non trivial example:** `a = 3 && (b = 4)`
  - **(b=4) is an assignment, evaluates to 4**
  - **&& has higher precedence than =**
  - **3 && (b=4) evaluates to true**
  - `a = 3 && (b = 4)` **is an assignment expression evaluates to 1, (true)**

# Statements and Blocks

- **An expression followed by a semicolon is a statement**

```
a = 3;
j = i++;
scanf("%d",&x);
```

- **Braces are used to group declarations and statements together into a compound block**

```
{
   a++;
   sum = sum + a;
   printf("%d", sum);
}
```

# Library functions

- C language is accompanied by a number of library functions
- One of the popular library is math.h that provides many common mathematical utilities
- Two step process to use
  - Need to include header file `#include<math.h>`
  - Tell compiler to link math library: `gcc <prog_name> -lm`
- Example:

```
printf("%f %f", sqrt(43.0), cos(2*PI));
```

- Return values of math functions are double
- Arguments can be constant, variable, expressions

# Math library functions

- `double acos(double x)` – compute arc cosine of x
- `double asin(double x)` – compute arc sine of x
- `double atan(double x)` – compute arc tangent of x
- `double atan(double x, double y)` – compute arc tangent of y/x
- `double cos(double x)` – compute cosine of angle x in radians
- `double cosh(double x)` – compute hyperbolic cosine of x
- `double sin(double x)` – compute sine of angle x in radians
- `double sinh(double x)` – compute hyperbolic sine of x
- `double tan(double x)` – compute tangent of angle x in radians
- `double tanh(double x)` – compute hyperbolic tangent of x
- `double ceil(double x)` – get smallest integral value that exceeds x
- `double floor(double x)` – get largest integral value less than x

# Math library functions (contd.)

- `double exp(double x)` – **compute exponential of x**
- `double fabs(double x)` – **compute absolute value of x**
- `double log(double x)` – **compute log of x base e**
- `double log10(double x)` – **compute log of x base 10**
- `double pow(double x, double y)` – **compute** $x^y$
- `double sqrt(double x)` – **compute square root of x**

- **There are other library functions too. We will explore later in this course**

# Example: Triangle area given length of 3 sides

```
#include<stdio.h>
#include<math.h>
int main(){
 double a, b, c, s, area;
 printf("Enter the length of 3 sides");
 scanf("%lf%lf%lf", &a, &b, &c);
 s = (a + b + c)/2;
 area = sqrt(s * (s - a) * (s - b) * (s - c));
 printf("Area is %lf\n", area);
}
```

# Practice problems

- Read in 3 integers and print their sum and average
- Read in 3 real numbers — u, initial velocity; f, acceleration; t, time; — of a vehicle, determine the distance traveled by the vehicle at the given time. [$s = ut + \frac{1}{2}ft^2$]
  - (a) Do not use math library function
  - (b) Use math library function
- Read in the coordinates (real numbers) of 3 points in 2-d plane. Print the area of the triangle formed by these points
- Read in the coefficient a, b, c of the expression $ax^2 + bx + c = 0$. Print the roots of the equation, assume no imaginary roots.