The cin Object

- Short for **console input** . It is used to read data typed at the keyboard.
- Must include the iostream library.
- When this instruction is executed, it **waits** for the user to type, it reads the characters until space or enter (newline) is typed, then it stores the value in the variable.
- **>>** the stream extraction operator use it to read data from cin (entered via the keyboard)
- right-hand operand MUST be a variable.

Example :

```
int length, width, area;

cout << "This program calculates the area of a rectangle.\n ";
cout << "What is the length of the rectangle? ";
cin  >>  length;
cout << "What is the width of the rectangle? ";
cin  >>  width;
area = length * width;
cout << "The area of the rectangle is " << area << ".\n";
```

Entering Multiple Values

```
cout << "Enter  the length and the width of  the rectangle? ";
cin  >> length >> width;

area = length * width;
cout << "The area of the rectangle is " << area << ".\n";
```

Reading a Character

```
int number;
char letter;

cout << "Enter an integer, and a character: ";
cin  >> number  >>   letter;
```

Mathematical Expressions

- An **expression** is a program component that evaluates to a value.
- An expression can be :
  - a literal,.
  - a variable, or
  - a combination of these using operators and parentheses.

Examples:

```
x + 5          x * y / z
num            'A'
4              -15e10
8 * x * x - 16 * x + 3
```

Each expression has a type, which is the **type** of the **result value**.

Where can expressions occur?

Expressions can occur in the **rhs** of an **assignment** statement.

Examples :-

```
x = y * 10 / 3;
y = 8;
num = num + 1;
aLetter = 'W';
x = y;
```

Expressions can also occur in the **rhs** of a **stream insertion operator (<<) (cout) :**

```
cout << "The pay for the week is  "<< hours * rate << endl;
cout << num;
cout << 25 / y;
```

Arithmetic Operators

- An operator is a symbol that tells the computer to perform specific mathematical or logical manipulations
- An operand is a value used in an operation.
- C++ has **Unary** operators : unary operators have one operand / argument:
    - -5
    - +9
    - -x
- minus sign (unary op) is called the negation operator - (negation)

    e.g. -5

- C++ also has **Binary** operators : binary operators have two operands:

```
+    addition          x + y
-    subtraction       index - 1
*    multiplication    hours * rate
/    division          total / count
%    modulus           count % 3
```

- `/ forms an **integer** division.`
- `% requires **integers** for both operands`

```
cout << 13 % 5.0 ; // ERROR
```

- `There is **no** operator for exponentiation in C++`
- `There IS a **library function** called "**pow**".`
- `The expression is a call to the pow function with arguments x and 3.0.`
- `Arguments **should** have type double and the result is a double.`
- `If x is 2.0, the result is 8.0.`

Note : #include <cmath> is required to use pow.

Example :

```
cout<< "\n\nThe power of 3 is  " <<  pow(3,2.0);
cout << "\n\nThe square root of 64 is  "  <<  sqrt(64);
cout << "\n\nThe round of 2.6 is  "  <<   round(2.6);
cout << "\n\nThe Natural log  of 2 is  " <<log(4);
cout << "\n\nThe log base 2 of 64 is  "  <<  log2(64) ;
```

Precedence rules (order of operations)

- **Which operation gets done first ???**

- ( ) parentheses/
- unary minus (an operator that has only one operand) e.g. -5 evaluate right to left.
- * / % (binary operators have two operands) evaluate left to right.
- + - (binary operators) evaluate left to right.
- If the expression has multiple operators from the same level, they associate left to right or right to left.
- You can use parentheses to **override** the precedence or **associatively** rules

Examples :

```
cout<< 2.5 + 3.0 / 1.5 <<endl;
cout << 6 - 3 * 5 / 2 - 1 << endl ;
cout<< 1- (2 + 1) % 2 * 4 << endl;
cout<< 20%(3*(4/2))-2*((3+1)+3) << endl;
```

- If both operands are integers, / (division) operator always performs integer division. **The fractional part is lost!!**

- If either operand is floating point, the result is floating point.

```
21 / 5          //value is: 4  ( integer division )
21.0 / 5        //value is: 4.2  ( float )
21 / 5.0        //value is: 4.2  ( float )
21 % 5          //value is: 1
```

Computational Shortcuts

```
int x = 10;
x++;       // same as x = x + 1     --->      x has value 11
x -= 10;   // same as x = x - 10    --->      x has value 1;
x *= 5;    // same as x = x * 5     ---> x has value 5;
```

Note :

A more elaborate statements may be expressed with the combined assignment operators.

```
result *= a + 5;
```

In this example , result is multiplied by the sum of a + 5.

The above statement is equivalent to

```
result = result * ( a + 5 ) ;
```

which is different from

```
result = result * a + 5;
```

Type Conversion Rules

Binary operations convert lower ranking value to the type of the other expression / value.

The **rhs** of assignment operator is **converted to** the type of the variable on the **lhs**.

Example :

```
int years;
float interestRate;
int result = years * interestRate;

// years is converted to float before being multiplied


int x, y = 4;
float z = 2.7;
x = y * z;

// y is converted to float, 10.8 is converted to int (10)
```

Integer Division

When an integer is divided by an integer the result is an integer.

The remainder/fractional part is discarded, NO ROUNDING.

Example :

```
double result;
result = 15 / 6;      // 2.5 ==> 2
result = 15.0 / 6;    // 6 ==> 6.0, result is 2.5
```

How About This

```
cout<<15/6<<endl;
cout<<15.0/6<<endl;

int   result1;
result1 = 15 / 6;
cout<<result1<<endl;
result1 = 15.0 / 6;
cout<<result1<<endl;
```

Type Casting

Type casting is an explicit or manual type conversion.

static_cast<datatype>(expr)

mainly used to force floating-point division

Example :

```
int hits, atBats ;
float battingAvg;

cin >> hits >> atBats;
battingAvg = static_cast<float>(hits) / atBats;    //  converts hits to float
cout<<"battingAvg  =  "<<battingAvg<<endl;
```

Why Not:

```
static_cast<float>(hits / atBats)
```

More Example :

```
cout<<static_cast<int>(3.4+5.3)<<endl;
cout<<static_cast<int>(3.4)+5.3<<endl;
cout<<static_cast<char>(65)<<endl;
cout<<static_cast<int>('A')<<endl;
cout<<static_cast<double>(5.2/2)<<endl;
cout<<static_cast<float>(5)/3<<endl;
```

Overflow/Underflow.

When the value assigned to a variable is too large or small for its type.

integers tend to wrap around, **without** warning:

```
short testVar = 32767;
cout << testVar << endl;    // 32767, max
value
testVar = testVar + 1;
cout << testVar << endl;   //-32768, min
value
```

floating point value overflow/underflow:

- may or may not get a warning
- result may be 0 or random value

Write a Valid C++ Expression for the following algebraic Expressions :

$$y = 32 + \left(X * \frac{180.0}{100.0} \right) \qquad y = \frac{x + x^3}{y^4 - 1} * l$$

What is the Output of the following Program

```
double x = 0.0;
 cout << "Please enter a number: ";
 cin >> x;
 cout << "You entered " << x

      << ", whose square is " << x*x << endl;
```

Multiple Assignment

- You can assign the same value to several variables in one statement:

  a = b = c = 12;

- is equivalent to:

  a = 12;
  b = 12;
  c = 12;

Combined Assignment

Assignment statements often have this form:

```
number = number + 1;    //add 1 to number
total = total + x;      //add x to total
y = y / 2;          //divide y by 2

int number = 10;
number = number + 1;
cout << number << endl;
```

C/C++ offers **shorthand** for these:

```
number  += 1;       // short for number = number+1;
total  -= x;  // short for total = total-x;
y  /= 2;       // short for y = y / 2;
```

Increment and decrement shorthand operators

- **number = number + 1;**
  - ++ number ; **pre-increment**
  - number ++ ; **post-increment**

- **number _down = number _down – 1 ;**
  - -- number _down ; **pre-decrement**
  - number _down -- ; **post-decrement**

Example

```
int num1 = 5;
int num2 = 3;
int num3 = 2;

num1 = num2++;      // Post increment operator.
cout << num1 << " "<< num2 << endl

num2 = --num3;      // Pre decrement operator.
cout << num1 << endl
     << num2 << endl
     << num3 << endl << endl;
```

Prefix vs Postfix

• ++ and -- operators can be used in expressions

• In prefix mode (++val, --val) the operator increments or decrements, **then** returns the value of the variable

• In postfix mode (val++, val--) the operator returns the value of the variable, **then** increments or decrements

More Mathematical Library Functions

- These take **`double`** as input, return a **`double`**

```
pow y = pow(x,d); // returns x raised to the power d
abs y = abs(x);  // returns absolute value of x
sqrt y = sqrt(x);     // returns square root of x
sin y = sin(x);   // returns the sine of x (in radians)

randomNumber  = rand() % 100;
                        // number in the range 0 to 99
```

****** Must include the math library when using these functions ******

```
srand (time (NULL) );
int randomNumer = rand() % 100 ;
```

Must include the ctime or (time.h)

& cstdlib or (stdlib.h)

libraries when using these functions ***

Example

```
.....
#include <cmath>
.
.
.

            double  result;
            result = pow(3,2.0);
            result = abs(-150);
            result = sqrt(16);
            result = sin(.510);

      ......................
```

Example : Creating random numbers between (5 and 99)

```
#include <iostream>
#include <time.h>
#include <cstdlib>

using namespace std;

int main()
{
     int min = 5 ;
     srand(time(NULL));
     int randNumber;

       randNumber = rand()% 94 + min  ;
       cout << randNumber <<"  " ;

 return 0;

}
```

Hand Tracing a Program

You be the computer. Track the values of the variables as the program executes.

Example

```
double num1, num2, num3, avg;

 cout << "Enter first number";
 cin >> num1;
 cout << "Enter second number";
 cin >> num2;
 cout << "Enter third number";
 cin >> num3;
 avg = num1 + num2 + num3 / 3;
cout << "The average is " << avg  << endl;
```

What is the Output of the above code fragment ???

Formatting Output

- Formatting: the way a value is printed:
  - spacing
  - decimal points, fractional values
  - scientific notation

- cout has a standard way of formatting values of each data type
- cout has "stream manipulators" to override the default formatting.
- Must the iomanip library by using

#include <iomanip>

Unformatted Output

```
cout << 2897 << " " << 5 << "  " << 837 << endl;
cout << 34 << "  " << 7 << "  " << 1623 << endl;

2897 5 837
34 7 1623
```

To line up the output, we can specify the (minimum) width for each number

Formatting Output : setw

- **setw** is a "stream manipulator", like endl
- **setw(n)** specifies the minimum width for the **next** item to be output

```
cout << "(" << setw(6) << 209 << ")"<< endl;

(      209)
```

- The value is **right** justified and padded with spaces.

Formatting Output : setw

```
cout << setw(6) << 7<< endl;
cout << setw(6) << 1623 << endl;
```

- What happens If the value is too big to fit it's printed in full ????

```
cout << "(" << setw(2) << 2096 << ")"<< endl;
```

Formatting Output : setprecision

1. **setprecision(n)** specifies the number of significant digits to be output for floating point values.

2. ***It remains in effect until it is changed***

3. The default seems to be 6

```
cout << 123.45678 << endl;
          // default is 6 >>   123.457 No SetPre.

cout << setprecision(4) << 1.3 << endl;     //  1.3

cout << 123.45678 << endl;         //  123.5 rule 2

cout << setprecision(12) << 34.123456789 << endl;
                                   //  34.123456789

cout << setprecision(9) << 34.1234567891 <<endl;
                                   //  34.1234568

cout << setprecision(2) << 34.21;          // 34
```

Formatting Output : fixed

a) **fixed forces floating point values to be output in decimal format, and not scientific notation.**

b) **When used with setprecision**, the value of setprecision is used to determine the number of digits after the decimal

```
cout << 12345678901.23 << endl;   //   1.23457e+10
cout << fixed << 12345678901.23 << endl;
                                  //   12345678901.23
cout << setprecision(2) << 123.45678 << endl;

                                  //  123.46  rule b
```

Is This Possible

```
cout << fixed <<setprecision(3)
     <<512345678901.23 << endl;
```

Formatting Output: right and left

- `left` causes all **subsequent** output to be **left** justified in its field
- `right` causes all **subsequent** output to be **right** justified in its field.
- `The default is` **`right`** `justified .`

```
cout << left << setw(10) << 1623 << endl;
```

Input : strings

Reading Strings

The cin object can read a string as input and store it in memory as C-string. C-string are commonly stored in ***character arrays***. For example

```
char companyName[12];
```

the number inside the brackets indicates the size of the array. The name of the array is company and it is large enough to hold 12 characters. Remember that the C-string have the null terminator at the end '\0'. So , the array is large enough to hold a C-string that is 11 character long

```
cin >> companyName;
```

- an example definition of an array variable:

```
char lastName[15];
```

- Input / Output with character arrays (**don't type spaces in the input string**):

```
char lastName[15];
cout << "Enter your last name: ";
cin >> lastName;
cout << "Your last name is: " << lastName;
```

What Happens if you enter more than 15 characters ?????

Formatted Input : setw

- Specifies the maximum width for the next item to be input
- Used to prevent putting too many characters into an array.

```
char lastName[15];
cout << "Enter your last name: ";
cin >> setw(5)>> lastName;
cout << "Your last name is: " << lastName;
```

Using string instead of array of characters

```
string lastName;
cout << "Enter your last name: ";
cin >> lastName;
cout << "Your last name is: " << lastName;
```

Problems

- It skips over any leading whitespace chars (spaces, tabs, or line breaks)
- It stops reading strings when it encounters the next whitespace character!

Using *getline* to input strings

- To work around this problem, you can use a C++ function named **getline.**

- **getline(cin,var);** reads in an entire line, including all the spaces, and stores it in a string variable.

```
string dName;
cout << "Enter your department  name: ";
getline (cin,dName);
cout << "You are in : " << dName;
```

Sample Output

```
Enter your department   name: Computer Science Department
You are in  : Computer Science Department
```

Reading a Line of input

- cin.getline(<array>,<size>)
- getline reads <size> - 1 characters from the screen into the char array <array> (and adds '\0' at the end)
- getline reads spaces, doesn't need setw

```
char sentence[60];
cout << "Enter a sentence: ";
cin.getline(sentence, 60);
cout << "You entered " << sentence << endl;
```

Sample Output

```
Enter a sentence: Life is a box of chocolates.
You entered Life is a box of chocolates.
```

Using cin.ignore & cin.clear

- **cin.ignore(10,'\n')** skips the next 10 characters, or until ’\n’ is encountered.

- Use it before a getline to consume the newline so it will start reading characters from the following line.

cin.clear() : clears the error flag on cin (so that future I/O operations will work correctly)

```
string name;
cin.clear()
cin.ignore(10,'\n');
        // skip the newline or 10 characters
cout << "Enter a name: ";
getline(cin,name);   // Read a string
cout << "Name " << name << endl;
```

Sample Output

```
Texas State University
Enter a name: Name e University
```

Example

```
/* This Program is written By Husain Gholoom
 * CS1428
 * This is A Sample Program
 *
 * Find all Syntax Errors in This Program

 *\


using namespace std,

intt main();

[

long double  X# = 0.254;

int X = 0;

int y = 1234567891

cout<<x<<endl;

cout<<y<endl;

cout<<20%(3*(4/2))-2\((3+1)+3)<<endl;

cout<< (4/2))-2*((3+1)+3)<<endl;

cout<< ((4/2))-2//((3+1)+3)<<endl;

retrn 0;

 }
```

Example

```
// This program calculates hourly wages, including overtime.

#include <iostream>
using namespace std;

int main()
{

   double regularWages,             // To hold regular wages
          basePayRate = 18.25,      // Base pay rate
          regularHours = 40.0,      // Hours worked less overtime
          overtimeWages,            // To hold overtime wages
          overtimePayRate = 27.78,  // Overtime pay rate
          overtimeHours = 10,       // Overtime hours worked
          totalWages;               // To hold total wages


   // Calculate the regular wages.

        regularWages = basePayRate * regularHours;

   // Calculate the overtime wages.

        overtimeWages = overtimePayRate * overtimeHours;


   // Calculate the total wages.

        totalWages = regularWages + overtimeWages;


   // Display the total wages.

   cout << "Wages for this week are $" << totalWages << endl;

   return 0;

}
```

Sample Output

```
Wages for this week are $1007.8
```