

Ch02-DataVariablesAndOperations

September 2, 2020

1 Data, Variables and Operations

1.1 Topics

- data and values
- C++ fundamental data types
- digital units and number systems
- variables and data assignment
- keywords and operators
- order of operations
- operators for numbers and strings
- constants
- type casting

1.2 Data and values

- data and values are the fundamentals to any computer language and program
- a value is one of the fundamental things – like a letter or a number – that a program manipulates
- almost all computer programs use and manipulate some data values

1.3 Literal values and representations

- at a high level, we deal with two types of data values: Numbers and Texts
- numbers can be further divided into two types:
 - Whole number literal values: 109, -234, etc.
 - Floating point literal values: 123.456, -0.3555, etc.
- text is a collection of 1 or more characters (symbols, digits or alphabets)
 - single character is represent using single quote (')
 - * char literal values: 'A', 'a', '%', '1', etc.
 - 2 or more characters are called string
 - * represented using double quotes (")
 - * string literal values: "CO", "John Doe", "1100", etc.
- programming languages need to represent and use these data correctly

1.4 C++ Fundamental types

- there are many fundamental types based on the size of the data program needs to store
 - most fundamental types are numeric types

- see here for all the supported types: <https://en.cppreference.com/w/cpp/language/types>
- the most common types we use are:

Type	Description	Storage size	Value range
void	an empty set of values; no type	system dependent: 4 or 8 bytes	NA
bool	true or false values	1 byte or 8 bits	true or false or 1 or 0
char	represents one ASCII character; inside single quote	1 byte or 8 bits	-2^7 to $2^7 - 1$ or -128 to 127
unsigned char	represents one ASCII character inside a single quote	1 byte or 8 bits	0 to $2^8 - 1$ or 0 to 255
int	+/-ve integers or whole numbers	4 bytes	-2^{31} to $2^{31} - 1$ or -2,147,483,648 to 2,147,483,647
signed int	same as int; signed (+ve and -ve) integers	4 bytes or 32 bits	-2^{31} to $2^{31} - 1$ or -2,147,483,648 to 2,147,483,647
unsigned int	unsigned (only positive) representation	4 bytes or 32 bits	0 to $2^{32} - 1$ or 0 to 4,294,967,295
long	+ve and -ve big integers	8 bytes or 64 bits	-2^{63} to $2^{63} - 1$ or -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long	positive big integers	8 bytes or 64 bits	0 to $2^{64} - 1$ or 0 to 18,446,744,073,709,551,615
float	single precision floating points	32 bits	7 decimal digits precision
double	double precision floating points	64 bits	15 decimal digits precision

- in C++, there's no fundamental type available to work with string data
- use `basic_string` defined in `<string>` library
 - more on `basic_string`: https://en.cppreference.com/w/cpp/string/basic_string
 - must include `<string>` library and `std` namespace
 - we'll dive into string more in depth in string chapter
- `sizeof(type)` operator gives size of fundamental types in bytes

```
[5]: sizeof(bool)
```

```
[5]: 1
```

```
[15]: sizeof(char)
```

```
[15]: 1
```

```
[2]: sizeof(int)
```

```
[2]: 4
```

```
[17]: sizeof(long)
```

```
[17]: 8
```

```
[18]: sizeof(float)
```

```
[18]: 4
```

```
[19]: sizeof(double)
```

```
[19]: 8
```

1.5 Units of digital data

- digital computers use binary number system consisting of two digits (0 and 1)
- every data and code is represented using binary values
 - hence the name binary or byte code for executable programs
 - letter A is encoded as 1000001 (7 binary digits)
- humans use decimal number system with 10 digits (0 to 9)
 - we have ways to represent texts using alphabets for English language e.g.: Hello Bond 707!
 - texts must be encoded into numbers, if we lived in the world that only understood numbers

Unit	Equivalent
1 bit (b)	0 or 1
1 byte (B)	8 bits (b)
1 kilobyte (KB)	1,024 B
1 megabyte (MB)	1,024 KB
1 gigabyte (GB)	1,024 MB
1 terabyte (TB)	1,024 GB
1 petabyte (PB)	1,024 TB
...	...

1.6 Number systems

- there are several number systems based on the base
 - base is number of unique digits number system uses to represent numbers
- binary (base 2), octal (base 8), decimal (base 10), hexadecimal (base 16), etc.

1.6.1 Decimal number system

- also called Hindu-Arabic number system
- most commonly used number system that uses base 10
 - has 10 digits or numerals to represent numbers: 0..9
 - e.g. 1, 79, 1024, 12345, etc.
- numerals representing numbers have different place values depending on position:
 - ones (10^0), tens(10^1), hundreds(10^2), thousands(10^3), ten thousands(10^4), etc.
 - e.g. $543.21 = (5 \times 10^2) + (4 \times 10^1) + (3 \times 10^0) + (2 \times 10^{-1}) + (1 \times 10^{-2})$

1.7 Number system conversion

- since computers understand only binary, everything (data, code) must be converted into binary
- all characters (alphabets and symbols) are given decimal codes for electronic communication
 - these codes are called ASCII (American Standard Code for Information Interchange)
 - A -> 65; Z -> 90; a -> 97; z -> 122, * -> 42, etc.
 - see ASCII chart: <https://en.cppreference.com/w/c/language/ascii>

1.7.1 Converting decimal to binary number

- algorithm steps:
 1. repeatedly divide the decimal number by base 2 until the quotient becomes 0
 - note remainder for each division
 2. collect all the remainders
 - the first remainder is the last (least significant) digit in binary
- example 1: convert $(10)_{10}$ to $(?)_2$
 - step 1:
 - 10 / 2 : quotient: 5, remainder: 0
 - 5 / 2 : quotient 2, remainder: 1
 - 2 / 2 : quotient: 1, remainder: 0
 - 1 / 2 : quotient: 0, remainder: 1
 - step 2:
 - * collect remainders from bottom up: 1010
 - so, $(10)_{10} = (1010)_2$
- example 2: convert $(13)_{10}$ to $(?)_2$
 - step 1:
 - 13 / 2 : quotient: 6, remainder: 1
 - 6 / 2 : quotient 3, remainder: 0
 - 3 / 2 : quotient: 1, remainder: 1
 - 1 / 2 : quotient: 0, remainder: 1
 - step 2:
 - * collect remainders from bottom up: 1101
 - so, $(13)_{10} = (1101)_2$

1.7.2 Converting binary to decimal number

- once the computer does the computation in binary, it needs to convert the results back to decimal number system for humans to understand

- algorithm steps:
 1. multiply each binary digit by its place value in binary
 2. sum all the products
- example 1: convert $(1010)_2$ to $(?)_{10}$
 - step 1:
 - * $0 \times 2^0 = 0$
 - * $1 \times 2^1 = 2$
 - * $0 \times 2^2 = 0$
 - * $1 \times 2^3 = 8$
 - step 2:
 - * $0 + 2 + 0 + 8 = 10$
 - so, $(1010)_2 = (10)_{10}$
- example 2: convert $(1101)_2$ to $(?)_{10}$
 - step 1:
 - * $1 \times 2^0 = 1$
 - * $0 \times 2^1 = 0$
 - * $1 \times 2^2 = 4$
 - * $1 \times 2^3 = 8$
 - step 2:
 - * $1 + 0 + 4 + 8 = 13$
 - so, $(1101)_2 = (13)_{10}$
- we got the same decimal vales we started from in previous examples
- food for thought: think how you'd go about writing a program to convert any positive decimal number into binary and vice versa!

1.8 Negative (signed) integers - Two's complement

- most common method of storing negative numbers on computers is a mathematical operation called Two's complement
- Two's complement of an N-bit number is defined as its complement with respect to 2^N
 - the sum of a number and its two's complement is 2^N
- e.g.: for the 3-bit binary number 010_2 , the two's complement is 110_2 , because $010_2 + 110_2 = 1000_2 = 2^3_{10} = 8_{10}$
- Two's complement of N-bit number can be found by flipping each bit and adding one to it
- e.g. find two's complement of 010
- Algorithm steps:
 - flipped each bit; 0 is flipped to 1 and vice-versa
 - add 1 to the flipped binary

$$\begin{array}{r}
 101 \\
 +1 \\
 \hline
 110
 \end{array}$$

1.9 Variables

- programs must load data values into memory to manipulate them

- data may be large and used many times during the program
 - typing the data values literally all the time is not efficient and fun
 - most importantly error prone due to typos
 - you may not even know that values may be if they're read from standard input, files, etc.
- variables are named memory location where data can be stashed for easy access and manipulation
- one can declared and use as many variables as necessary
- C++ is statically and strongly typed programming language
 - variables are tied to their specific data types that must be explicitly declared when declaring variables

1.9.1 variable declaration

- statements that create variables/identifiers to store some data values
- as the name says, value of variables can vary/change over time
- syntax:

```
type varName;
type varName1, varName2, ...; //declare several variables all of the same type
```

Variable name	Memory Space	Memory address
int StudentID	int ?	0x...10
string firstName	string ?	0x...14
string lastName	string ?	0x...99
char MI	char ?	0x...2a
float GPA	float ?	0x...2b

Fig. C++ Variables and Memory

1.9.2 rules for creating variables

- variable names are case sensitive
- must declare variables before they can be used
- can't define variable with the same name more than once
- can't use keywords as variable names

- data stored must match the type of variable
- variable names can't contain symbols (white spaces, #, &, etc.) except for (_ underscore)
- variable names can contain digits but can't start with a digit
- variable names can start with only alphabets (lower or upper) and _ symbol

1.9.3 best practices

- use descriptive and meaningful but concise name
 - one should know quickly what data you're storing
- use lowercase; camelCase or (_ underscore) to combine multiple words

1.9.4 keywords

- keywords are reserved names and words that have specific purpose in C++
 - they can only be used what they're intended for
- e.g., char, int, unsigned, signed, float, double, bool, if, for, while, return, struct, class, operator, try, etc.
- all the keywords are listed here: <https://en.cppreference.com/w/cpp/keyword>

```
[1]: // examples of variable declaration
bool done;
char middleInitial;
char middleinitial;
int temperature;
unsigned int age;
long richest_persons_networth;
float interestRate;
float length;
float width;
double space_shuttle_velocity;
```

```
[2]: // TODO:
// Declare 10 variables of atleast 5 different types
```

1.9.5 string variables

- declare variables that store string data
 - 1 or more string of characters
- in C++ string is an advanced type
- must include <string> header file or library to use string type
- must use **std** namespace
- strings are represented using a pair of double quotes ("string")
- more on string is covered in later chapter

```
[2]: // string variables
#include <string>

using namespace std;
```

```
string fullName;
string firstName;
string address1;
string country;
string state_name;
std::string state_code; // :: name resolution operator
```

```
[5]: // TODO:
      // Declare 5 string variables
```

1.10 Assignment

- once variables are declared, data can be stored using assignment operator (=)
- **assignment statements** have the following syntax

```
varName = value;
```

```
[3]: // assignment examples
done = false;
middleInitial = 'J'; // character is represent using single quote
middleinitial = 'Q';
temperature = 73;
age = 45;
richest_persons_networth = 120000000000; // 120 billion
interestRate = 4.5;
length = 10.5;
width = 99.99f; // can end with f for representing floating point number
space_shuttle_velocity = 950.1234567891234567; // 16 decimal points
```

```
[3]: 950.123
```

```
[4]: // string assignment examples
fullName = "John Doe";
firstName = "John";
address1 = "1100 North Avenue"; // number as string
country = "USA";
state_name = "Colorado";
state_code = "CO";
```

```
[8]: // TODO: assign some values to variables defined above
```

1.10.1 variable declartion and initialization

- variables can be declared with initial value at the time of construction
- if you know what value a variable should start with; this saves you typing
- often times its the best practice to initialize variable with default value
- several ways to initialize variables: <https://en.cppreference.com/w/cpp/language/initialization>

- two common ways:
 1. Copy initialization (using = operator)
 2. Value initialization (using {} curly braces)
 - also called uniform initialization
 - useful in initializing advanced types such as arrays, objects, etc.

```
[9]: // Copy initialization
float price = 2.99f;
char MI = 'B'; //middle initial
string school_name = "Grand Junction High";
```

```
[10]: // Value/uniform initialization
char some_letter{'U'};
int some_length{100};
float some_float{200.99};
string some_string{"Hello World!"};
```

1.10.2 variables' values can be changed

- however, type of the values must be same as the type of the variables
- C++ is strongly and statically typed programming language!

```
[11]: price = 3.99;
price = 1.99;
MI = 'Q';
school_name = "Fruita Monument High";
some_string = "Goodbye, World!";
```

1.10.3 auto type

- if variable is declared and initialize, you can use **auto** keyword to let compiler determine type of variable based on the value it's initialized with

```
[5]: auto var1 = 10; // integer
auto var2 = 19.99f; // float
auto var3 = 99.245; // double
auto var4 = '@'; // char
```

```
[3]: // char * (pointer) type and not string type
auto full_name = "John Doe";
```

```
[8]: // can automatically declare string type
#include <string>
using namespace std;

auto full_name1 = string("Jake Smith"); // string type!
```

1.10.4 Visualize variables and memory with pythontutor.com

1.11 Operators

- special symbols used to represent simple computations
 - like addition, multiplication, modulo, etc.
- C++ has operators for numbers, characters, and strings
- operators and precedence rule: https://en.cppreference.com/w/cpp/language/operator_precedence
- arithmetic operators: https://en.cppreference.com/w/cpp/language/operator_arithmetic

1.11.1 unary operators

- takes one operand (value)

Operator	Symbol	Syntax	Operation
positive	+	+100	positive 100 (default)
negative	-	-23.45	negative 23.45

1.11.2 binary operators

- binary operators take two operands (left operator right)
- operands are values that operators work on

Operator	Symbol	Name	Syntax	Operation
add	+	plus	x + y	add the value of y with the value of x
subtract	-	hyphen	x - y	subtract y from x
multiply	*	asterick	x * y	product of x and y
divide	/	slash	x / y	divide x by y (int division if x and y are both ints)
modulo	%	percent	x % y	remainder when x is divided by y

1.11.3 adding numbers

- + can be used to add literal values or variables

```
[2]: // adding literal integer values
+1 + (-1)
```

```
[2]: 0
```

```
[2]: // adding literal floating points
99.9 + 0.1
```

```
[2]: 100.00000
```

```
[3]: // adding int variables
int num1, num2, sum;
```

```
[4]: num1 = 10;  
      num2 = 5;  
      sum = num1 + num2;
```

```
[5]: // let's see the value of sum  
      sum
```

[5]: 15

```
[6]: // adding float variables  
      float n1 = 3.5;  
      float n2 = 2.5;  
      float total = n1+n2;
```

```
[7]: // see total values  
      total
```

[7]: 6.00000f

1.11.4 subtracting numbers

```
[9]: // subtracting literal integers  
      10-1
```

[9]: 9

```
[10]: // subtracting literal floating points  
      99.99 - 10.99
```

[10]: 89.000000

```
[11]: // subtracting variables  
      num1-num2
```

[11]: 5

1.11.5 multiplying numbers

```
[12]: // multiplying literal integers  
      2*3
```

[12]: 6

```
[13]: // multiplying literal floats  
      2.5 * 2.0
```

[13]: 5.0000000

```
[14]: // multiplying numeric variables  
n1*n2
```

[14]: 8.75000f

1.11.6 dividing numbers

```
[15]: // dividing literal integers  
10/2
```

[15]: 5

```
[16]: 9/2 // integer division; remainder is discarded
```

[16]: 4

```
[17]: // dividing literal floats  
// if one of the operands is floating point number, C++ performs float division  
9.0/2
```

[17]: 4.5000000

```
[18]: // dividing numeric variables  
n1/n2
```

[18]: 1.40000f

1.11.7 capturing remainder from a division

- use modulo or remainder (%) operator
- only works on integers

```
[19]: // modulo or remainder operator  
5%2 // testing for odd number
```

[19]: 1

```
[20]: 4%2 // testing for even number
```

[20]: 0

```
[21]: // can't divide 10 by 11  
10%11
```

[21]: 10

```
[23]: // expressions with variables and literals
// declare some variables
int hour, minute;
```

```
[24]: // assign some values
hour = 11;
minute = 59;
```

```
[27]: // Number of minutes since midnight
hour * 60 + minute
```

```
[27]: 719
```

```
[28]: // Fraction of the hour that has passed
minute/60
```

```
[28]: 0
```

1.11.8 bitwise operators

- <https://www.learncpp.com/cpp-tutorial/38-bitwise-operators/>
- bitwise operators work on binary numbers (bits)
- bitwise operations are used in lower-level programming such as device drivers, low-level graphics, communications protocol packet assembly, encoding and decoding data, encryption technologies, etc.
- a lot of integer arithmetic computations can be carried out much more efficiently using bitwise operations

Operator	Symbol	Symbol Name	Syntax	Operation
bitwise left shift	<<	left angular bracket	x << y	all bits in x shifted left y bits; multiplication by 2
bitwise right shift	>>	right angular bracket	x >> y	all bits in x shifted right y bits; division by 2
bitwise NOT	~	tilde	~x	all bits in x flipped
bitwise AND	&	ampersand	x & y	each bit in x AND each bit in y

– 2's complement is calculated by flipping each bit and adding 1

[18] : ~0

[18] : -1

[4] : 1 & 1

[4] : 1

[5] : 1 & 0

[5] : 0

[21] : 0 & 1

[21] : 0

[22] : 0 & 0

[22] : 0

[23] : 1 | 1

[23] : 1

[24] : 1 | 0

[24] : 1

[25] : 0 | 1

[25] : 1

[26] : 0 | 0

[26] : 0

[27] : 1 ^ 1

[27] : 0

[28] : 1 ^ 0

[28] : 1

[29] : 0 ^ 1

[29]: 1

```
[30]: 0 ^ 0
```

[30]: 0

1.12 Order of operations

- expressions may have more than one operators
 - the order of evaluation depends on the rules of precedence

1.12.1 PEMDAS

- acronym for order of operations from highest to lowest
 - P** : Parenthesis
 - E** : Exponentiation
 - M** : Multiplication
 - D** : Division
 - A** : Addition
 - S** : Subtraction
- when in doubt, use parenthesis!

```
[18]: // computation is similar to what we know from Elementary Math
      2+3*4/2-2
```

[18]: 6

```
[29]: (2+3)*4/(2-1)
```

[29]: 20

1.13 Operators for characters

- mathematical operators also work on characters
- characters' ASCII values are used in computations
- C++, when safe, converts from one type to another; called type **coercion**
 - characters are converted into their corresponding integer ASCII values
 - coercion** is safe when data is not lost, e.g. converting int to float

```
[30]: 'a'+1 // a -> 97
```

[30]: 98

```
[31]: 'A'-1 // A -> 65
```

[31]: 64

```
[24]: 'A'*10
```


[24]: 650

```
[33]: 'A'/10
```

[33]: 6

```
[34]: 'A'+'A'
```

[34]: 130

1.14 Operators for strings

- certain operators are defined or overloaded for string types
 - more on user defined advanced types and operator overloading later
- + : concatenates or joins two strings giving a new longer string

```
[36]: // variables can be declared and initialized at the same time  
#include <iostream>  
#include <string>  
using namespace std;  
  
string fName = "John";  
string lName = "Smith";  
string space = " ";  
string fullName = fName + space + lName;
```

```
[38]: fullName
```

[38]: "John Smith"

1.15 Constants

- constants are named values that remain unchanged through out the program
- useful for declaring values that are fixed
 - e.g. value of π , earth's gravity, unit conversions, etc.
- two ways to define constants in C++
 1. use **const** keyword in front of an identifier
 - syntax:
`const type identifier = value;`
 2. use **#define** preprocessor directive
 - syntax:
`#define identifier value`
 - after an identifier has been defined with a value, preprocessor replaces each occurrences of PI with value

```
[40]: const double pi = 22/7.0; // evaluate 22/7.0 and use it as the const value for  
      ↪ pi  
const float earth_gravity = 9.8; // m/s^2 unit
```

```
[41]: // let's see the value of constant pi
      pi
```

[41]: 3.1428571

```
[42]: // try to assign different value to constant pi
      pi = 3.141592653589793238;
```

```
input_line_83:3:4: error: cannot assign to variable
      'pi' with const-qualified type 'const double'
pi = 3.141592653589793238;
~~ ^

input_line_80:2:15: note: variable 'pi' declared const
here
    const double pi = 22/7.0; // evaluate 22/7.0 and use it as the const value for
pi
~~~~~^~~~~~
```

Interpreter Error:

```
[43]: // let's use constants
      double radius = 10.5;
      double area_of_circle = pi*radius*radius;
```

```
[44]: // value of area of circle
      area_of_circle
```

[44]: 346.50000

```
[45]: // preprocessor directive to declare named constant
      #define PI 3.141592653589793238
```

```
[46]: PI*radius*radius
```

[46]: 346.36059

1.15.1 floating point operation accuracy

- floating point calculations may not be always 100% accurate
- you have to choose the accuracy upto certain decimal points to accept the results as correct
- [google area of circle](#)
 - use same radius 10.5 and compare the results

1.16 Type casting

- data values need to be converted from one type to another to get correct results
- explicitly converting one type into another is called **type casting**
- implicit conversion is called **coercion**
- not all values can be converted from one type to another!

1.16.1 converting numeric values to string type

- use `to_string(value)` to convert value to string
- must include `<string>` library and `std` namespace

```
[1]: #include <string>
using namespace std;

string str_val = to_string(99); // 99 is casted "99" and the value is assigned
    ↪ to str_val
```

```
[2]: str_val
```

```
[2]: "99"
```

```
[3]: // typeid library can be used to know the name of data types
#include <typeid>
```

```
[5]: // typeid operator is defined in typeid library
typeid(str_val).name()
```

```
[5]: "NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE"
```

```
[7]: int whole_num = 1234;
string str_val1 = to_string(whole_num);
```

```
[8]: str_val1
```

```
[8]: "1234"
```

```
[9]: float float_num = 129.99;
string str_num1 = to_string(float_num);
```

```
[10]: str_num1
```

```
[10]: "129.990005"
```

```
[12]: string str_val2 = to_string('A'); // uses ASCII value
```

```
[13]: str_val2
```

[13]: "65"

1.16.2 converting string values to numeric types

- certain values can be converted into numeric types such as int, float, double, etc.
- `<cstdlib>` provides some functions for us to convert string to numeric data
- more on `<cstdlib>`: <http://www.cplusplus.com/reference/cstdlib>
- `atoi("value")` converts string value to integer
 - converts all leading consecutive digits as integer
- `atof("value")` converts string value to double
- must include `<cstdlib>` library to use its functions
 - * converts all leading consecutive digits and period as floating point number

```
[28]: #include <cstdlib> //stoi and stof
```

```
[22]: // converting string to integers
atoi("120")
```

[22]: 120

```
[23]: atoi("43543 alphabets")
```

[23]: 43543

```
[16]: atoi("")
```

[16]: 0x7fff6778073c <invalid memory address>

```
[24]: atof("23.55")
```

[24]: 23.550000

```
[25]: atof("132.68 text")
```

[25]: 132.68000

```
[27]: atof("text 4546.454")
```

[27]: 0.0000000

1.16.3 typcasting among numeric types

- at times, you may need to convert integers to floating points and vice versa
- use `int(value)` to convert float to int
- use `float(value)` to convert int or double to float
- use `double(value)` to convert int or float to double
- don't need to include any library to use these built-in functions

```
[2]: int(10.99) // convert double to int; discard decimal points or round down
```

```
[2]: 10
```

```
[6]: int(345.567f) // discard decimal points or round down
```

```
[6]: 345
```

```
[3]: float(19)
```

```
[3]: 19.0000f
```

```
[7]: double(3.33f) // convert float to double
```

```
[7]: 3.3299999
```

```
[5]: double(3)
```

```
[5]: 3.0000000
```

1.16.4 typecasting between char and int

- use `char(intValue)` to convert to char
- use `int(charValue)` to convert to int

```
[8]: char(65) // ASCII code to char
```

```
[8]: 'A'
```

```
[9]: int('A') // char to ASCII code
```

```
[9]: 65
```

1.17 Exercises

1. Declare some variables required to store information about a student at a university for an a banner system. Assign some values to those variables.

```
[39]: // solution to Exercise 1
#include <string>
using namespace std;

int main() {
    long st_id; // student id
    string st_first_name; // first name
    string st_last_name;
    string st_address; // complete address
    string emg_contact_name; // emergency contact's full name
```

```

float GPA;
// courses enrollment info?

st_id = 700123456;
st_first_name = "Jane";
st_last_name = "Smith";
st_address = "123 Awesome Street";
emg_contact_name = "Joe Smith";
GPA = 4.0;

return 0;
}

```

2. Declare some variables required to store information about an employee at a university. Assign some values to those variables.
3. Declare some variables required to store information about a merchandise in a store for inventory management system. Assign some values to those variables.
4. Declare some variables required to store information about a rectangular shape. Calculate area and perimeter of a rectangle. Assign some values to those variables.
5. Declare variables required to store information about a circle to calculate its area and perimeter. Assign some values to those variables. Calculate area and perimeter.
6. Declare some variables required to store information about a hotel room for booking management system.
7. Declare some variables required to store length of sides of a triangle. Calculate area using Herons' formula.
 - Search for Heron's formula, if you're not sure what it is.
8. Using pencil and paper, write your full name in binary.
 - E.g. Ram Basnet in Binary is:
 - 01010010 01100001 01101101 00100000 01000010 01100001 01110011 01101110 01100101 01110100

1.18 Summary

- this notebook discussed data and C++ standard data types
- variables are named memory location that store data values
- C++ variables are static and strongly typed
- looked into C++ operators for various data types
- learned about order of operations, PEMDAS
- learned that constants are used to store values that should not be changed in program
- exercises and sample solutions

[]: