

# DataVariablesAndOperations

August 30, 2022

## 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
- most programming languages need to represent and use these data correctly
- 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 represented using a pair of single quotes ( ' ' )
    - \* char literal values: 'A', 'a', '%', '1', etc.
  - 2 or more characters are called string
    - \* represented using a pair of double quotes ( " " )
    - \* string literal values: "CO", "John Doe", "1100", "9.99", etc.

### 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
<b>void</b>	an empty set of values; no type	system dependent: 4 or 8 bytes	NA
<b>bool</b>	true or false	1 byte or 8 bits	true or false 1 or 0
<b>char</b>	one ASCII character	1 byte or 8 bits	$-2^7$ to $2^7 - 1$
<b>unsigned char</b>	one ASCII character	1 byte or 8 bits	0 to $2^8 - 1$
<b>int</b>	+/-ve integers	4 bytes	$-2^{31}$ to $2^{31} - 1$
<b>unsigned int</b>	only positive integers	4 bytes or 32 bits	0 to $2^{32} - 1$
<b>long</b>	+ve and -ve big integers	8 bytes or 64 bits	$-2^{63}$ to $2^{63} - 1$
<b>unsigned long</b>	positive big integers	8 bytes or 64 bits	0 to $2^{64} - 1$
<b>float</b>	single precision floating points	32 bits	7 decimal points
<b>double</b>	double precision floating points	64 bits	15 decimal points

- in C++, there's no fundamental type available to work with string data
- two common ways to store string data:
  - use C-string or array of characters
  - use `basic_string` defined in `<string>` library
    - \* more on `basic_string`: [https://en.cppreference.com/w/cpp/string/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 **Strings** chapter

#### 1.4.1 sizeof operator

- one may want to know the size of memory allocated for the fundamental types
  - some of these types are system dependent (e.g., `long` is 32 bit in x86 and 64 bit in x64)
- **sizeof(type)** operator gives size of fundamental types in bytes
- let's check the size of some fundamental types on my 64-bit MacBook Pro laptop

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

```
[1]: 1
```

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

```
[2]: 1
```

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

```
[3]: 4
```

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

```
[4]: 8
```

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

[5]: 4

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

[6]: 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
- the following table shows the various units of digital data

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 digits
  - 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)
  - $A \rightarrow 65; Z \rightarrow 90; a \rightarrow 97; z \rightarrow 122, * \rightarrow 42$ , 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 in reverse order
    - the first remainder is the last (least significant) digit in binary
- example 1: what is decimal  $(10)_{10}$  in binary  $(?)_2$  ?
  - step 1:
$$\frac{10}{2} = \text{quotient: } 5, \text{ remainder: } 0 \quad \frac{5}{2} = \text{quotient: } 2, \text{ remainder: } 1 \quad \frac{2}{2} = \text{quotient: } 1, \text{ remainder: } 0 \quad \frac{1}{2} = \text{quotient: } 0, \text{ remainder: } 1$$
  - step 2:
    - \* collect remainders from bottom up: 1010
  - so,  $(10)_{10} = (1010)_2$
- example 2: what is decimal  $(13)_{10}$  in  $(?)_2$  ?
  - step 1:
$$\frac{13}{2} = \text{quotient: } 6, \text{ remainder: } 1 \quad \frac{6}{2} = \text{quotient: } 3, \text{ remainder: } 0 \quad \frac{3}{2} = \text{quotient: } 1, \text{ remainder: } 1 \quad \frac{1}{2} = \text{quotient: } 0, \text{ remainder: } 1$$
  - 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: what is binary  $(1010)_2$  in decimal  $(?)_{10}$  ?
  - step 1:
$$\begin{aligned} * 0 \times 2^0 &= 0 \\ * 1 \times 2^1 &= 2 \\ * 0 \times 2^2 &= 0 \\ * 1 \times 2^3 &= 8 \end{aligned}$$
  - step 2:
    - \*  $0 + 2 + 0 + 8 = 10$
  - so,  $(1010)_2 = (10)_{10}$
- example 2: what is binary  $(1101)_2$  in decimal  $(?)_{10}$  ?
  - 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  $010_2 + 110_2 = 1000_2 = 2_{10}^3$
- 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:
    1. flipped each bit; 0 is flipped to 1 and 1 is flipped to 0  
 $010 \rightarrow 101$
    2. add 1 to the flipped binary
 

101	
+1	
-----	
110	

### 1.8.1 Example: What is -3 decimal in 8-bit binary representation?

- convert  $3_{10}$  to an 8-bit binary
  - $3_{10} \rightarrow 00000011_2$
- 1. find Two's complement of 8-bit binary
  - $00000011_2 \rightarrow 11111100_2 + 1 = 11111101_2$
- 2. Sanity check:
  - $00000011_2 + 11111101_2 = 100000000_2 = 2_{10}^8$
- So,  $-3_{10} = 11111101_2$  in an 8-bit representation

## 1.9 Exercise

1. Convert decimal integer 7 into binary with 16 bits.
2. Convert -7 decimal integer into binary with 16 bits.

## 1.10 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 what values may be if they're read from standard input, files, etc.
- variables are named memory location where data can be stored for easy access and manipulation
- one can declare and use as many variables as necessary
- C++ is a static and strongly typed programming language
  - variables are tied to their specific data types that must be explicitly declared when declaring variables

### 1.10.1 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
<b>int</b> StudentID	int ?	0x...10
<b>string</b> firstName	string ?	0x...14
<b>string</b> lastName	string ?	0x...99
<b>char</b> MI	char ?	0x...2a
<b>float</b> GPA	float ?	0x...2b

**Fig. C++ Variables and Memory**

### 1.10.2 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 `_` and `$` (underscore and dollar)
- variable names can contain digits but can't start with a digit
- variable names can start with only alphabets (lower or upper) and `_` symbol

### 1.10.3 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

### 1.10.4 C++ 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 C++ keywords are listed here: <https://en.cppreference.com/w/cpp/keyword>

```
[7]: // examples of variable declaration
bool done;
char middleInitial;
char middleinitial; // hard to read all lowercase name
int temperature;
unsigned int age;
long richest_persons_networth;
float interestRate;
float length;
float width;
double space_shuttle_velocity;
```

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

### 1.10.5 String variables

- declare variables that store string data
  - 1 or more string of characters
- an easy way to use string is by using C++ advanced type defined in `<string>` header file
- must include `<string>` header file or library to use string type
- must also use `std` namespace
- strings are represented using a pair of double quotes ("string")
- more on string type is covered in **Strings** chapter
- the following are some examples of string variables

```
[9]: // 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
```

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

### 1.11 Assignment operator ( = )

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

```
varName = value;
```

- since C++ is a strongly typed language, the type of value must match the type of variable
  - strongly typed languages enforces type safety and matching during the compile time

```
[11]: // 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; // number can end with f to represent as float
space_shuttle_velocity = 950.1234567891234567; // 16 decimal points
```

```
[11]: 950.12346
```

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

```
[13]: // TODO: assign different values to variables defined above
```



### 1.11.1 Variable declaration and initialization

- variables can be declared with an initial value at the time of construction
- if you know what value a variable should start with, this saves you typing
- often times it's 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.

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

```
[15]: // Value/uniform initialization
char some_letter{'U'};
int some_length{100};
float some_float{200.99};
string some_string = {"Hello World!"}; // can also combine the two!
```

### 1.11.2 Variable's value can be changed

- variable's value can vary through out the program
  - hence the name variable
- however, type of the value must be same as the type of the variable at the time of declaration
- C++ is a strongly and static typed programming language!

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

```
[17]: price = "4.99"; // is this valid?
```

input\_line\_34:2:10: **error:** assigning to 'float' from

incompatible type 'const char [5]'

```
price = "4.99"; // is this valid?
```

```
^~~~~~
```

Interpreter Error:

### 1.11.3 auto type

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

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

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

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

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

```
[21]: // use typeid function to find the name of the types
      // typeid is defined in typeid library
      #include <typeid>
```

```
[22]: typeid(full_name1).name()
```

```
[22]: "NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE"
```

```
[23]: // should print "i" -> short for integer
      // Note: may also print invalid memory address in Jupyter notebook!
      typeid(var1).name()
```

```
[23]: 0x7fff67b6373c <invalid memory address>
```

### 1.11.4 Visualize variables and memory with [pythontutor.com](https://pythontutor.com)

## 1.12 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](https://en.cppreference.com/w/cpp/language/operator_precedence)
- arithmetic operators: [https://en.cppreference.com/w/cpp/language/operator\\_arithmetic](https://en.cppreference.com/w/cpp/language/operator_arithmetic)

### 1.12.1 Unary operators

- takes one operand
- operands are values that operators work on
- there are two unary operators for numeric operands

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

### 1.12.2 Binary operators

- binary operators take two operands (left operator right)
- the following table shows the binary operators for numeric operands

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

### 1.12.3 Adding numbers

- + symbol is used to add literal values or variables

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

[24]: 0

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

[25]: 100.00000

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

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

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

[28]: 15

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

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

```
[30]: 6.00000f
```

#### 1.12.4 Subtracting numbers

- - symbol is used to subtract literal numbers or variables

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

```
[31]: 9
```

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

```
[32]: 89.000000
```

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

```
[33]: 5
```

#### 1.12.5 Multiplying numbers

- \* asterick symbol is used to multiply literal numbers and variables

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

```
[34]: 6
```

```
[35]: // multiplying literatl floats
2.5 * 2.0
```

```
[35]: 5.0000000
```

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

```
[36]: 8.75000f
```

#### 1.12.6 Dividing numbers

- / symbol is used to divide literal numbers or variables

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

[37]: 5

```
[38]: 9/2 // integer division; remainder is discarded
```

[38]: 4

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

[39]: 4.5000000

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

[40]: 1.40000f

### 1.12.7 Capturing remainder from a division

- use modulo or remainder ( % ) operator to find the remainder of literal values or variables
- only works on positive integers

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

[41]: 1

```
[42]: 4%2 // testing for even number
```

[42]: 0

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

[43]: 10

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

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

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

```
[46]: 719
```

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

```
[47]: 0
```

### 1.12.8 Exercise

- How many hours and minutes are in 121 minutes?

### 1.12.9 Bitwise operators

- <https://www.learncpp.com/cpp-tutorial/38-bitwise-operators/>
- bitwise operators work on binary numbers (bits)
  - integers are implicitly converted into binary and then bitwise operations are applied
- 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	$x \ll y$	all bits in $x$ shifted left $y$ bits; multiplication by $2^y$
bitwise right shift	»	right angular bracket	$x \gg y$	all bits in $x$ shifted right $y$ bits; division by $2^y$
bitwise NOT	~	tilde	$\sim x$	all bits in $x$ flipped
bitwise AND	&	ampersand	$x \& y$	each bit in $x$ AND each bit in $y$
bitwise OR		pipe	$x   y$	each bit in $x$ OR each bit in $y$

Operator	Symbol	Symbol Name	Syntax	Operation
bitwise XOR	$\wedge$	caret	$x \wedge y$	each bit in x XOR each bit in y

#### 1.12.10 Table for bitwise operations

x	y	$x \& y$
1	1	1
1	0	0
0	1	0
0	0	0

$\&$  - bitwise AND

x	y	$x   y$
1	1	1
1	0	1
0	1	1
0	0	0

$|$  - bitwise OR

x	$\sim x$
1	0
0	1

$\sim$  - bitwise NOT

x	y	$x \wedge y$
1	1	0
1	0	1
0	1	1
0	0	0

$\wedge$  - bitwise XOR

bitwise left shift examples

```
[48]: // convert 1 decimal to binary and shift left by 4 bits
1 << 4 // same as 1*2*2*2*2; result is in decimal
```

[48] : 16

### Explanation

- [illegible]

```
[49]: 3 << 4 // same as 3*2*2*2*2 or 3*2^4
```

[49] : 48

### Explanation

- $3_{10} = 00000000000000000000000000000011_2$
- $3 < 4 = 000000000000000000000000000000110000_2 = 2^5 + 2^4 = 32 + 16 = 48_{10}$

## Bitwise right shift examples

```
[50]: 1024 >> 10 // same as 1024/2/2/2/2/2/2/2/2/2/2
```

[50] : 1

### Explanation

- [illegible]

## Bitwise NOT examples

```
[51]: ~0 // result shown is in decimal!
```

[51] : -1

```
[52]: ~1 // Note: 1 in binary using 32-bit width (31 0s and 1) 00000....1
      // result shown is in decimal
```

[52] : -2

### Explanation

- $0_{10} = 00000000000000000000000000000_2$
- $1_{10} = 000000000000000000000000000001_2$
- $-1_{10} = 111111111111111111111111111111_2$
- $2_{10} = 000000000000000000000000000010_2$



- $-2_{10} = 111111111111111111111111111110_2$
- Note: -ve numbers are stored in Two's complement
  - 2's complement is calculated by flipping each bit and adding 1 to the binary of positive integer

### Bitwise AND examples

[53]: 1 & 1

[53]: 1

[54]: 1 & 0

[54]: 0

[55]: 0 & 1

[55]: 0

[56]: 0 & 0

[56]: 0

### Bitwise OR examples

[57]: 1 | 1

[57]: 1

[58]: 1 | 0

[58]: 1

[59]: 0 | 1

[59]: 1

[60]: 0 | 0

[60]: 0

### Bitwise XOR examples

[61]: 1 ^ 1

[61]: 0

[62]: 1 ^ 0

[62]: 1

```
[63]: 0 ^ 1
```

[63]: 1

```
[64]: 0 ^ 0
```

[64]: 0

### 1.13 Order of operations

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

#### 1.13.1 PEMDAS

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

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

[65]: 6

```
[66]: // same as  
(2+((3*4)/2))-2
```

[66]: 6

```
[67]: (2+3)*4/(2-1) // Note: must use * to multiply after ( )
```

[67]: 20

```
[68]: // typical mistake  
(2+3)4/(2-1) // error
```

```
input_line_129:2:7: error: expected ';' after  
expression  
(2+3)4/(2-1) // error
```

```
~  
;
```

Interpreter Error:

### 1.14 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

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

```
[69]: 98
```

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

```
[70]: 64
```

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

```
[71]: 650
```

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

```
[72]: 6
```

```
[73]: 'A'+'A'
```

```
[73]: 130
```

### 1.15 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

```
[74]: // 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;
```

```
[75]: fullName
```

```
[75]: "John Smith"
```

## 1.16 Constants

- constants are named values that remain unchanged through out the program
- useful for declaring values that are fixed
  - e.g. value of  $\pi$ , 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

```
[76]: 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
```

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

```
[77]: 3.1428571
```

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

```
input_line_148:3:4: error: cannot assign to variable
```

```
'pi' with const-qualified type 'const double'
```

```
pi = 3.141592653589793238;
```

```
~~ ^
```

```
input_line_145:2:15: 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:

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

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

```
[80]: 346.50000
```

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

```
[82]: PI*radius*radius
```

```
[82]: 346.36059
```

### 1.16.1 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 provided above

## 1.17 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!

### 1.17.1 Converting numeric values to string type

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

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

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

```
[84]: str_val
```

```
[84]: "99"
```

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

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

```
[86]: "NSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE"
```

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

```
[88]: str_val1
```

```
[88]: "1234"
```

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

```
[90]: str_num1
```

```
[90]: "129.990005"
```

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

```
[92]: str_val2
```

```
[92]: "65"
```

### 1.17.2 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 c-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

```
[93]: #include <cstdlib> //atoi and atof
```

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

```
[94]: 120
```

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

[95]: 43543

```
[96]: atoi("text 123")
```

[96]: 0

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

[97]: 23.550000

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

[98]: 132.68000

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

[99]: 0.0000000

### 1.17.3 Converting C++ strings into numeric types

- <http://www.cplusplus.com/reference/string/>
- `<string>` library provides many functions to convert `std::string` into numeric types
- `stoi( )` - converts `std::string` type to integer
- `stof( )` - converts `std::string` type to float
- `stol( )` - converts `std::string` type to long int
- `stoul( )` - converts `std::string` to unsigned long integer

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

```
[101]: string int_num = "99";
string float_num = "100.99";
```

```
[102]: // typecast string int and string float to corresponding numeric types
// do + operation on numeric types
float result = stoi(int_num)+stof(float_num);
```

```
[103]: result
```

[103]: 199.990f

### 1.17.4 Type casting 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

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

```
[104]: 10
```

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

```
[105]: 345
```

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

```
[106]: 19.0000f
```

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

```
[107]: 3.3299999
```

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

```
[108]: 3.0000000
```

### 1.17.5 Type casting between char and int

- use **char(intValue)** to convert ASCII int to char
- use **int(charValue)** to convert char to ASCII int

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

```
[109]: 'A'
```

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

```
[110]: 65
```

## 1.18 Labs

### 1. Variables Lab

- write a C++ program that produces the following output on console
- use the partial solution provided in [labs/variables/main.cpp](#)
- observe and note how the special symbols such as single quote, double quotes and black slashes
- run the program as it is using the provided make file in the stdio folder
- complete the rest of the ASCII Art by fixing all the FIXMEs
- write #FIXED next to each FIXME



```

| \ _ / |      *****      ( \ _ / )
/  @  @  \      *      ASCII Art      *      ( = ' . ' = )
(  >  0  <  )    *      Author: <Your Name>    *      ( " ) _ ( " )
>>x<<           *      CS Foundation Course    *
/  0  \          *****

```

### 1.19 Exercises

1. Declare some variables required to store information about a student for a university banner system. Assign some values to those variables.
  - see sample answer here [exercises/variables/exercisel](#)
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 for an inventory management system of a store. 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 or Jupyter Notebook, write your full name in binary.
  - e.g., Ram Basnet in Binary is:
  - 01010010 01100001 01101101 00100000 01000010 01100001 01110011 01101110 01100101 01110100

### 1.20 Summary

- this notebook discussed data and C++ fundamental 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

[ ]: