C++ for Science and Engineering COSC3000/6000

2017 Spring Semester

Part II
C++ Basics

1 A Sample Program

```
//
//  main.cpp
//  sample01
//
//  Created by Hideki Fujioka on 3/12/13.
//  Copyright (c) 2013 Tulane University. All rights reserved.
//
/*
  this is comments
 */
#include <iostream>

int main(int argc, const char * argv[])
{

    // cout is in the namespace 'std'
    std::cout << "Hello, World!\n";
    std::cout << "Hello, World!" << std::endl;

    using namespace std;
    cout << "Hello, World!" << endl;

    return 0;
}
```

1.1 Comments

- // is the symbol for a single line comment
  - Comments are explanatory notes for the programmer
  - All text on the line following // is ignored by the compiler
  - Example: //calculate regular wages gross_pay = rate * hours;
- /* and */ enclose multiple line comments
  - Example: /* This is a comment that spans multiple lines without a comment symbol on the middle line */

1.2 #include Directives

- Include Directives add library files to our programs
- Tells compiler where to find information about items used in the program.
- **iostream** is a library containing definitions of **cin** and **cout**
- Standard C++ Library reference `http://www.cplusplus.com/reference/`
  - To make the definitions of the **cin** and **cout** available to the program:
    ```
    #include <iostream>
    ```
- Using Directives include a collection of defined names
  - To make the names **cin** and **cout** available to our program:
    ```
    using namespace std;
    ```

1.3 using namespace std;

- Tells the compiler to use names in **iostream** in a "standard" way.
  - A **namespace** is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.
  - Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.
  - All identifiers at namespace scope are visible to one another without qualification.
  - Identifiers outside the namespace can access the members by using the fully qualified name for each identifier.

1.4 main Function

To begin the main function of the program

```
int main(int argc, const char * argv[]) {
```

To end the main function

```
        return 0;
}
```

Main function ends with a return statement

2 Variables and assignments

- Variables are like small blackboards
  - We can write a number on them
  - We can change the number
  - We can erase the number
- C++ variables are names for memory locations
  - We can write a value in them
  - We can change the value stored there
  - We cannot erase the memory location
    - Some value is always there

2.1 Identifiers

- Variables names are called identifiers
- Choosing variable names
  - Use meaningful names that represent data to be stored
  - First character must be
    - a letter
    - the underscore character
  - Remaining characters must be
    - letters
    - numbers
    - underscore character, _

2.2 Keywords

- Keywords (also called reserved words)
  - Are used by the C++ language
  - Must be used as they are defined in the programming language
  - Cannot be used as identifiers
  - See `http://en.cppreference.com/w/cpp/keyword`

2.3 Declaring Variables

- Before use, variables must be declared
  - Tells the compiler the type of data to store
    - Examples:

```
int number_of_bars;
double one_weight, total_weight;
```

  - **int** is an abbreviation for integer.
    - could store 3, 102, 3211, -456, etc.
    - number_of_bars is of type integer
  - **double** represents numbers with a fractional component
    - could store 1.34, 4.0, -345.6, etc.

* one_weight and total_weight are both of type **double**

- Two locations for variable declarations

```
#include <iostream>
int main(int argc, const char * argv[]) {
        ...
        int sum;
        sum = score1 + score 2;
        ...
        return 0;
}
```

```
#include <iostream>
int main(int argc, const char * argv[]) {
        int sum;
        ...
        sum = score1 + score 2;
        ...
        return 0;
}
```

- Declaration syntax:

```
Type_name Variable_1 , Variable_2,...;
```

- Declaration Examples:

```
double average, m_score, total_score;
double moon_distance;
int age, num_students;
int cars_waiting;
```

2.4 Assignment Statements

- An assignment statement changes the value of a variable
  - `total_weight = one_weight + number_of_bars;`
    * total_weight is set to the sum one_weight + number_of_bars
  - Assignment statements end with a semi-colon
  - The single variable to be changed is always on the left of the assignment operator '='
  - On the right of the assignment operator can be
    * Constants – `age = 21;`
    * Variables – `my_cost = your_cost;`
    * Expressions – `circumference = diameter * 3.14159;`

2.5 Assignment Statements and Algebra

- The '=' operator in C++ is not an equal sign
  - The following statement cannot be true in algebra
    `number_of_bars = number_of_bars + 3;`
  - In C++ it means the **new** value of number_of_bars is the **previous** value of number_of_bars plus 3

2.6 Initializing Variables

- Declaring a variable does not give it a value
  - Giving a variable its first value is initializing the variable
- Variables are initialized in assignment statements

```
double mpg; // declare the variable
mpg = 26.3; // initialize the variable
```

- Declaration and initialization can be combined using two methods

– Method 1

```
double mpg = 26.3, area = 0.0 , volume;
```

– Method 2

```
double mpg(26.3), area(0.0), volume;
```

3 Input and Output

- A **data stream** is a sequence of data
  - Typically in the form of characters or numbers
- An input stream is data for the program to use
  - Typically originates
    - at the keyboard
    - at a file
- An output stream is the program's output
  - Destination is typically
    - the monitor
    - a file

3.1 Output using "cout"

- **cout** is an output stream sending data to the monitor
- The insertion operator "<<" inserts data into **cout**
- Example: `cout << number_of_bars << " candy bars\n";`
  - This line sends two items to the monitor
    - The value of **number_of_bars**
    - The quoted string of characters **" candy bars\n"**
      - Notice the space before the '**c**' in candy
      - The '**\n**' causes a new line to be started following the '**s**' in bars
    - A new insertion operator is used for each item of output

3.1.1 Examples Using cout

- This produces the same result as the previous sample

```
cout << number_of_bars;
cout << " candy bars\n";
```

- Here arithmetic is performed in the **cout** statement

```
cout << "Total cost is $" << (price + tax);
```

- Quoted strings are enclosed in double quotes ("Walter")
  - Don't use two single quotes (')
- A blank space can also be inserted with

```
cout << " ";
```

  if there are no strings in which a space is desired as in **" candy bars\n"**

3.2 Escape Sequences

- Escape sequences tell the compiler to treat characters in a special way
- '\' is the escape character
  - To create a newline in output use **\n**

    ```
    cout << "\n";
    ```

    or the newer alternative

    ```
    cout << endl;
    ```

  - Other escape sequences:
    - \t – a tab
    - \\ – a backslash character
    - \" – a quote character

3.3 Formatting Real Numbers

- Real numbers (type **double**) produce a variety of outputs

  ```
  double price = 78.5;
  cout << "The price is $" << price << endl;
  ```

- The output could be any of these:
  The price is $78.5
  The price is $78.500000
  The price is $7.850000e01
- The most unlikely output is:
  The price is $78.50

3.3.1 Showing Decimal Places

- **cout** includes tools to specify the output of type double
- To specify fixed point notation
  - **setf**(ios::fixed)
- To specify that the decimal point will always be shown
  - **setf**(ios::showpoint)
- To specify that two decimal places will always be shown
  - **precision**(2)
- Example:

  ```
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout.precision(2);
  cout << "The price is " << price << endl;
  ```

3.4 Input Using "cin"

- cin is an input stream bringing data from the keyboard
- The extraction operator (>>) removes data to be used
- Example:

```
cout << "Enter the number of bars in a package\n";
cout << " and the weight in ounces of one bar.\n";
cin >> number_of_bars; cin >> one_weight;
```

- This code prompts the user to enter data then reads two data items from **cin**
  - The first value read is stored in number_of_bars
  - The second value read is stored in one_weight
  - Data is separated by spaces when entered

3.4.1 Reading Data From cin

- Multiple data items are separated by spaces
- Data is not read until the enter key is pressed
  - Allows user to make corrections
- Example:

```
cin >> v1 >> v2 >> v3;
```

  - Requires three space separated values
  - User might type
    34 45 12 <enter key>

3.5 Designing Input and Output

- Prompt the user for input that is desired
  - **cout** statements provide instructions

```
cout << "Enter your age: ";
cin >> age;
```

    * Notice the absence of a new line before using **cin**
- Echo the input by displaying what was read
  - Gives the user a chance to verify data

```
cout << age << " was entered." << endl;
```

3.5.1 Sample02

```
//
//  sample02
//
//  Created by Hideki Fujioka on 3/12/13.
//  Copyright (c) 2013 Tulane University. All rights reserved.
//

#include <iostream>

using namespace std;

int main(int argc, const char * argv[])
{
    int a, b;

    cout << "input two numbers :";
    cin >> a >> b;

    cout << "a=" << a << " b=" << b << endl;

    return 0;
}
```

4 Data Types and Expressions

- 2 and 2.0 are not the same number
  - A whole number such as 2 is of type **int**
  - A real number such as 2.0 is of type **double**
- Numbers of type **int** are stored as exact values
- Numbers of type **double** may be stored as approximate values due to limitations on number of significant digits that can be represented

4.1 Writing Constants

4.1.1 Writing Integer constants

- Type int does not contain decimal points
  - Examples: 34 45 1 89

4.1.2 Writing Double Constants

- Type double can be written in two ways
  - Simple form must include a decimal point
    - Examples: 34.1 23.0034 1.0 89.9
  - Floating Point Notation (Scientific Notation)
    - Examples:
      - 3.41e1 means 34.1
      - 3.67e17 means 367000000000000000.0
      - 5.89e-6 means 0.00000589
  - Number left of **e** does not require a decimal point
  - Exponent cannot contain a decimal point

4.2 Other Number Types

- Various number types have different memory requirements
  - More precision requires more bytes of memory
  - Very large numbers require more bytes of memory
  - Very small numbers require more bytes of memory

Syntax Type Name	Syntax Memory Used	Syntax Size Range
short (also called *short int*)	2 bytes	–32,767 to 32,767
int	4 bytes	–2,147,483,647 to 2,147,483,647

`http://www.cplusplus.com/doc/tutorial/variables/`

4.3 Integer types

- **long** or **long int** (often 4 bytes)
  - Equivalent forms to declare very large integers
    ```
    long big_total;
    long int big_total;
    ```
- **short** or **short int** (often 2 bytes)
  - Equivalent forms to declare smaller integers
    ```
    short small_total;
    short int small_total;
    ```

4.4 Floating point types

- **long double** (often 16 bytes, depends on system)
  - Declares floating point numbers with up to 34 significant digits
    ```
    long double big_number;
    ```
- **float** (often 4 bytes)
  - Declares floating point numbers with up to 7 significant digits
    ```
    float not_so_big_number;
    ```

4.5 Type char

- Computers process character data too
- **char**
  - Short for character
  - Can be any single character from the keyboard
- To declare a variable of type **char**:
  ```
  char letter;
  ```

4.5.1 char constants

- Character constants are enclosed in single quotes
  ```
  char letter = 'a';
  ```
- Strings of characters, even if only one character is enclosed in double quotes
  - "a" is a string of characters containing one character

– 'a' is a value of type character

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

ASCII Table (American standard code for information interchange)

– This doesn't work. You will get an error!!

```
char letter = "a";
```

because "a" is a string of characters (even just one character). To store a string of characters, you need an array of **char**.

4.5.2 Reading Character Data

- **cin** skips blanks and line breaks looking for data
- The following reads two characters but skips any space that might be between

```
char symbol1, symbol2;
cin >> symbol1 >> symbol2;
```

- User normally separate data items by spaces

  J D

- Results are the same if the data is not separated by spaces

  JD

4.5.3 sample03

```
/*
 * sample03.cpp
 *
 *  Created on: Jan 15, 2016
 *      Author: fuji
 */

#include <iostream>

int main(int argc, const char * argv[]){
    char a, b;
    std::cout << "input two characters :";
    std::cin >> a >> b;

    std::cout << "You input " << a << " and "<< b << std::endl;

    return 0;
}
```

4.6 Type bool

- **bool** is a new addition to C++
  - Short for boolean
  - Boolean values are either **true** or **false**
- To declare a variable of type bool:
  ```
  bool old_enough;
  ```

4.7 Type Compatibilities

- In general store values in variables of the same type
  - This is a type mismatch:
    ```
    int int_variable;
    int_variable = 2.99;
    ```
  - If your compiler allows this, **int_variable** will most likely contain the value **2**, not **2.99**

4.7.1 int ⟷double

- Variables of type **double** should not be assigned to variables of type **int**
  ```
  int int_variable;
  double double_variable;
  double_variable = 2.00;
  int_variable = double_variable;
  ```
- If allowed, **int_variable** contains **2**, not **2.00**
- Integer values can normally be stored in variables of type **double**
  ```
  double double_variable;
  double_variable = 2;
  int_variable = double_variable;
  ```
- **double_variable** will contain **2.0**

4.7.2 char ⟷int

- The following actions are possible but generally not recommended!
- It is possible to store char values in integer variables
```
int value = 'A';
```
value will contain an integer representing 'A'
- It is possible to store int values in char variables
```
char letter = 65;
```

4.7.3 bool ⟷int

- The following actions are possible but generally not recommended!
- Values of type **bool** can be assigned to **int** variables
  - **True** is stored as **1**
  - **False** is stored as **0**
- Values of type **int** can be assigned to **bool** variables
  - Any non-zero integer is stored as **true**
  - Zero is stored as **false**

4.8 Arithmetic

- Arithmetic is performed with operators
  + for addition
  - for subtraction
  * for multiplication
  / for division
- Example: storing a product in the variable **total_weight**
```
total_weight = one_weight * number_of_bars;
```

4.8.1 Results of Operators

- Arithmetic operators can be used with any numeric type
- An operand is a number or variable used by the operator
- Result of an operator depends on the types of operands
  - If both operands are **int**, the result is **int**
  - If one or both operands are **double**, the result is **double**

4.8.2 Division of Doubles

- Division with at least one operator of type **double** produces the expected results.
```
double divisor, dividend, quotient;
divisor = 3;
dividend = 5;
quotient = dividend / divisor;
```
  - quotient = 1.6666...
  - Result is the same if either dividend or divisor is of type **int**

4.8.3 Division of Integers

- Be careful with the division operator!
  - **int** / **int** produces an integer result (true for variables or numeric constants)

```
int dividend, divisor, quotient;
dividend = 5;
divisor = 3;
quotient = dividend / divisor;
```

  - The value of quotient is **1, not 1.666... (not even 2)**
  - Integer division does not round the result, the fractional part is discarded!

4.8.4 Integer Remainders

- % operator gives the remainder from integer division

```
int dividend, divisor, remainder;
dividend = 5;
divisor = 3;
remainder = dividend % divisor;
```

- The value of remainder is 2

4.8.5 sample04

```
/*
 * sample04.cpp
 *
 *  Created on: Jan 15, 2016
 *      Author: fuji
 */
#include <iostream>

int main(int argc, const char * argv[]){
        int dividend, divisor;

        std::cout << "input two integers :\n";
        std::cout << "dividend=";
        std::cin >> dividend;
        std::cout << "divisor=";
        std::cin >> divisor;

        int quotient = dividend / divisor;
        int remainder = dividend % divisor;

        std::cout << dividend << " / " << divisor << " = " << quotient << std::endl;
        std::cout << dividend << " % " << divisor << " = " << remainder << std::endl;
        return 0;
}
```

4.8.6 Type Casting

The problems in 4.8.3. If the variable for quotient declared as **double**,

```
        int dividend, divisor;
        double quotient;
        dividend = 5;
        divisor = 3;
        quotient = dividend / divisor;
```

The value of quotient is still 1, not 1.666... A Type Cast produces a value of one type from another type

- **static_cast<double>(int_variable)**

produces a double representing the integer value of int_variable. For example:

```
quotient = static_cast<double>(dividend) / divisor;
```

The value of quotient is 1.666...
It also works with old C-style type cast:

```
quotient = (double)dividend / divisor;
```

4.8.7 Arithmetic Expressions

- Use spacing to make expressions readable
  - Which is easier to read?
    `x+y*z` or `x + y * z`

- Precedence rules for operators are the same as used in your algebra classes
- Use parentheses to alter the order of operations
  `x + y * z` (y is multiplied by z first)

  `(x + y) * z` (x and y are added first)

4.8.8 Operator Shorthand

- Some expressions occur so often that C++ contains to shorthand operators for them
- All arithmetic operators can be used this way
  `count = count + 2;` becomes `count += 2;`

  `bonus = bonus * 2;` becomes `bonus *= 2;`

  `time = time / rush_factor;` becomes `time /= rush_factor;`

  `remainder = remainder % (cnt1+ cnt2);` becomes `remainder %= (cnt1 + cnt2);`

4.9 Increment / Decrement

For a integer variable,

```
int n = 0;
n++;
```

'n++' increment 1.
this is equivalent with

```
int n = 0;
n += 1;
```

For decrement,

```
int n = 0;
n--;
```

4.9.1 n++ or ++n

Those are same, but programs below behave different.

```
        int n = 0;
        std::cout << n++ << srd::endl;
```

0

```
        int n = 0;
        std::cout << ++n << srd::endl;
```

1

5 Binary Data

5.1 Numeric Limit

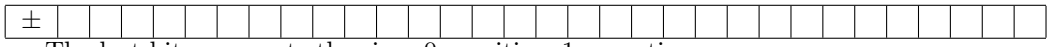
- Various number types have different memory requirements
  - More precision requires more bytes of memory

5.1.1 sizeof

"**sizeof()**" returns the size of data type in byte.

```
std::cout << "sizeof(char)=" << sizeof(char) << std::endl;
std::cout << "sizeof(short)=" << sizeof(short) << std::endl;
std::cout << "sizeof(int)=" << sizeof(int) << std::endl;
std::cout << "sizeof(long)=" << sizeof(long) << std::endl;
std::cout << "sizeof(float)=" << sizeof(float) << std::endl;
std::cout << "sizeof(double)=" << sizeof(double) << std::endl;
std::cout << "sizeof(long double)=" << sizeof(long double) << std::endl;
```

5.1.2 Integer (int) 4 bytes = 8×4=32bits



The last bit represents the sign; 0: positive, 1: negative.
$2^{31} = 2147483648$, therefore, **int** can represent -2147483648 to 2147483647.
$(0)_{10} = 00000000000000000000000000000000$
$(1)_{10} = 00000000000000000000000000000001$
$(-1)_{10} = 10000000000000000000000000000001$
$(2147483647)_{10} = 01111111111111111111111111111111$
$(-2147483648)_{10} = 10000000000000000000000000000000$

5.1.3 IEEE754 (Standard for Floating Point Arithmetic)

Floating Point Representation:

- Scientific Notation -1.23456×10^7 : [sign][significand]x[base=10]^[exponent]

Binary Floating Point Representation

- -1.001010×2^3 : [sign][significand]x[base=2]^[exponent]
- $\pm b_0.b_{-1}b_{-2}b_{-3}b_{-4}b_{-5}b_{-6}\cdots b_{-(p-1)} \times 2^E$

5.1.4 IEEE754 single precision (float)

4 bytes = 32 bits

1bit	8bits	23bits
s	e	f
sign	exponent	fraction

The fraction is digits after decimal point, f = $b_{-1}b_{-2}b_{-3}b_{-4}b_{-5}\cdots b_{-23}$.
$b_0 = 1$ for all numbers except **e=0**. b_0 is the hidden bit, called **normalized**, and not stored.
Base 10 number : $(-1)^s \times \left(1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + b_{-4} \times 2^{-4} + b_{-5} \times 2^{-5} + \cdots + b_{-23} \times 2^{-23}\right) \times 2^{e-127}$
Since the fraction is represented by 23bits, the precision is approximately 7 decimal digits ($\log_{10}\left(2^{23}\right) = 6.92\cdots$)

5.1.5 IEEE754 double precision (double)

8 bytes = 64 bits

1bit	11bits	52bits
s	e	f
sign	exponent	fraction

The fraction is digits after decimal point, $\mathrm{f} = b_{-1}b_{-2}b_{-3}b_{-4}b_{-5}\cdots b_{-52}$.

$b_0 = 1$ for all numbers except **e=0**. b_0 is the hidden bit, called **normalized**, and not stored.

Base 10 number : $(-1)^s \times \left(1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + b_{-4} \times 2^{-4} + b_{-5} \times 2^{-5} + \cdots + b_{-52} \times 2^{-52}\right) \times 2^{e-1023}$

Since the fraction is represented by 52bits, the precision is approximately 15 decimal digits ($\log_{10}\left(2^{52}\right) = 15.65\cdots$)

5.1.6 Special Numbers

- zero = | ± | 00000000 | 00000000000000000000000 | s=0or1, e=0000, f=0000 (all bits are 0 (off))
- $\pm\infty$ infinity = | ± | 11111111 | 00000000000000000000000 | s=0or1, e=11111 (all bits are 1 (on)), f=000000 (all 0)
- NaN, not a number = | ± | 11111111 | non zero | s=0or1, e=11111 (all bits are 1). f=non-zero

5.1.7 Smallest Positive Number

- **Denormalized numbers**

  The number representations described above are called **normalized**, meaning that the implicit leading hidden binary digit is a 1. To reduce the loss of precision when an underflow occurs, IEEE 754 includes the ability to represent fractions smaller than are possible in the normalized representation, by making the hidden leading digit a 0. Such numbers are called **denormal**. A denormal number is represented with a exponent of all 0 bits, which represents an exponent of −126 in single precision (not −127), or −1022 in double precision (not −1023).

- Single Precision

  $N_{min} = 2^{-23} \times 2^{-126} = 2^{-149} \sim 1.4012985{\times}10^{-45}$

- Double Precision

  $N_{min} = 2^{-52} \times 2^{-1022} = 2^{-1074} \sim 4.940656458412465 \times 10^{-324}$

5.1.8 Largest Positive Number

- Single Precision

  $N_{max} = 1.1111...1 \times 2^{emax-127} = (2 - 2^{-23}) \times 2^{127} \sim 3.4 \times 10^{38}$

- Double Precision

  The range of positive number is from

  $N_{max} = 1.1111...1 \times 2^{emax-1023} = (2 - 2^{-52}) \times 2^{1023} \sim 1.8 \times 10^{308}$

5.2 Bits / Binary

The byte is a unit of digital information in computing and telecommunications that most commonly consists of 8 bits. Historically, the byte was the number of bits used to encode a single character of text in a computer and for this reason it is the smallest addressable unit of memory in many computer architectures. 8 bits binary digits can represent decimal values of 0 through 255.

A bit is the basic unit of information in computing and digital communications. A bit can have only one of two values, (yes/no, on/off, true/false). The most common representation of these values are 0 and 1. The term bit is a contraction of binary digit.

Computer use binary arithmetic two values "0" and "1". In the 8-bit ASCII code, a lowercase "a" is represented by the bit string "01100001". Since binary digits are too long, Hexadecimal (base 16) digits are usually used.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

5.2.1 Conversion from binary to decimal (base 10) number,

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

$= b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 + b_3 \times 2^3 + b_4 \times 2^4 + b_5 \times 2^5 + b_6 \times 2^6 + b_7 \times 2^7$

- For example, 255=11111111, which is FF in base 16.

5.2.2 Binary Values after Decimal Point

b_3	b_2	b_1	b_0	b_{-1}	b_{-2}	b_{-3}	b_{-4}

$= b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 + b_3 \times 2^3 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + b_{-4} \times 2^{-4}$

- For example, 14.75=1110.11

Base 10	Base 2
0.5	0.1
0.25	0.01
0.125	0.001
0.0625	0.0001
0.03125	0.00001
0.015625	0.000001
0.0078125	0.0000001
0.00390625	0.00000001
0.001953125	0.000000001
0.0009765625	0.0000000001

It is impossible to represent base 10 numbers with base 2 exactly, for example, 0.1=0.0001100110..... irrational number!! (causing **rounding errors**)

5.2.3 K byte, M byte, T byte, P byte ...

- K byte(Kilo Byte) = 2^{10}= 1024 $\sim 10^3$bytes
- M byte(Mega Byte) = 2^{20}= 1048576 $\sim 10^6$bytes
- G byte(Giga Byte) = 2^{30}= 1073741824 $\sim 10^9$bytes
- T byte(Tera Byte) = 2^{40}= 1099511627776 $\sim 10^{12}$bytes
- P byte(Peta Byte) = 2^{50}= 1125899906842624 $\sim 10^{15}$bytes
- E byte(Exa Byte) = 2^{60}= 1.152921504606847e+18 $\sim 10^{18}$bytes

6 Simple Flow Controls

- Branch
  - Lets program choose between two alternatives
- Flow of control
  - The order in which statements are executed

6.1 'if' statement

'if' statement makes a branch.

```
        int num;
        std::cout << "input number :";
        std::cin >> num;
        if (num % 2 == 0){
                std::cout << "The number is even\n";
        }
```

6.1.1 Boolean Expressions

Boolean expressions are expressions that are either **true** or **false.** Comparison operators such as '>' (greater than) are used to compare variables and/or numbers. Including the parentheses, **(num % 2 == 0)** is the boolean expression. Comparison operators are:

- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to
- != not equal or inequality
- == equal or equivalent (Not '=' !!!!!)

A few of the comparison operators that use two symbols (No spaces allowed between the symbols!)

6.1.2 'if-else' flow control

When the boolean expression is true, only the true statements enclosed in { } are executed . When the boolean expression is false, only the false statements enclosed in { } are executed.

```
        int num;
        std::cout << "input number :";
        std::cin >> num;
        if (num % 2 == 0){
                std::cout << "The number is even\n";
        }else{
                std::cout << "The number is odd\n";
        }
```

6.1.3 AND/OR/NOT

Boolean expressions can be combined into more complex expressions.

&& – The AND operator.

```
        if ((2 < x) && (x < 7)){
                // true;
        }
```

- This goes to the true statement only if x is between 2 and 7.
- Inside parentheses are optional but enhance meaning.
- If the first expression is false, the second expression won't be evaluated.

| | – The OR operator (no space!)

```
if (( x == 1) || ( x == y)){
        // true;
}
```

- This goes to the true statement if x is 1 or x is equal to y, also if both comparisons are true.

! – negates any boolean expression.

```
if (!( x < 1)) {
        // true;
}
```

This goes to the true statement if x is greater than or equal to 1. This can be rewritten as

```
if ( x >= 1) {
        // true;
}
```

! Operator can make expressions difficult to understand. . . use only when appropriate.

6.2 'while' loop

```
while(expression){
        //Loop body : do here while expression is true
}
```

- First, the expression is evaluated.
  - If false, the program skips to the line following the while loop.
  - If true, the body of the loop is executed.
- During execution, some item from the expressions changed.
- After executing the loop body, the expression is checked again repeating the process until the expression becomes false.
- A while loop might not execute at all if the expression is false on the first check.

6.2.1 sample05

```
/*
 CPP sample05.cpp
 */

#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
        int a;

        while (1) {
                cout << "input a number:";
                cin >> a;

                if (a == 0)
                        break;

                if (a % 2) {
                        cout << a << " is odd number" << endl;
                } else {
                        cout << a << " is even number" << endl;
                }
        }
        cout << "bye" << endl;

        return 0;
}
```

6.3 'do-while' loop

```
        do{
                //Loop body
        } while(expression);
```

- A do-while loop is always executed at least once.
- The body of the loop is first executed.
- The boolean expression is checked after the body has been executed.

6.4 'for' loop

```
        for (initialize ; expression ; increment){
                // do here while expression is true
        }
```

- The initialize step is executed first, and only once.
- Next, the expression (condition) is evaluated.
  - If it is true, the body of the loop is executed.
  - If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the increment statement.

- This statement allows you to update any loop control variables.
- The condition is now evaluated again.
- If it is true, the loop executes and the process repeats.

Example:

```
//
// Example
//
#include <iostream>
using namespace std;
int main (int argc, char *argv[]) {
        for (int i = 1 ; i <= 3 ; i++){
                for (int j = 1 ; j <= 3 ; j++){
                        cout << i << " x " << j << " = " << i * j << endl;
                }
        }
        return 0;
}
```

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
```

6.4.1 sample06

```cpp
/*
 CPP sample06.cpp
 */

#include <iostream>
#include <iomanip>

using namespace std;

int main(int argc, char *argv[]) {
        int n;

        cout << "number of terms : ";
        cin >> n;
        cout << "n=" << n << endl;
        ;

        double x = 0.0;
        for (int i = 0; i < n; i += 2) {
                x += 1.0 / (2 * i + 1);
                x -= 1.0 / (2 * i + 3);
        }

        cout << "Pi=" << setprecision(16) << 4.0 * x << endl;

        return 0;
}
```

6.5 break, continue

```cpp
        int a;
        int i = 0;
        while(i < 5){
                std::cout << "input a number:";
                std::cin >> a;
                if (a == 0) break;
                if (a < 0) continue;
                std::cout << "i=" << i << std::endl;
                i++;
        }
        std::cout << "bye" << std::endl;
```

When a=0, get out from the loop. when a<0, go to the end of loop without executing **cout** and **i++**.

7 Program Style

- A program written with attention to style
  - is easier to read
  - easier to correct
  - easier to change

7.1 Indenting

- Items considered a group should look like a group
  - Skip lines between logical groups of statements
  - Indent statements within statements

```
if (x == 0)
        statement;
```

- Braces {} create groups
  - Indent within braces to make the group clear
  - Braces placed on separate lines are easier to locate
  - IDE helps to find the corresponding open and close braces.

```
#include <iostream>

int main(int argc, const char * argv[])
{
    // cout is in the namespace 'std'
    std::cout << "Hello, World!\n";
    std::cout << "Hello, World!" << std::endl;

    { // a scope begins
        using namespace std;
        cout << "Hello, World!" << endl;
    } // a scope ends
    // 'using namespace std' is no longer good

    return 0;
}
```

7.2 Comments

- // is the symbol for a single line comment
  - Comments are explanatory notes for the programmer
  - All text on the line following // is ignored by the compiler
  - Example:

```
//calculate regular wages
gross_pay = rate * hours;
```

- /* and */ enclose multiple line comments
  - Example:

```
/* This is a comment that spans
multiple lines without a comment
symbol on the middle line */
```

7.3 Constants

- Number constants have no mnemonic value
- Number constants used throughout a program are difficult to find and change when needed
- Constants
  - Allow us to name number constants so they have meaning
  - Allow us to change all occurrences simply by changing the value of the constant

7.3.1 const

- const is the keyword to declare a constant
- Example:
```
const int WINDOW_COUNT = 10;
```
declares a constant named WINDOW_COUNT
  - Its value cannot be changed by the program like a variable
  - It is common to name constants with all capitals

7.3.2 macros

- Macro can define constants.
- Example:
```
#define WINDOW_COUNT 10
```
  - Pre-processor replace the string "WINDOW_COUNT" with 10 before compelling it.
    So be careful if you write arithmetic in macro
```
#include <iostream>

#define WINDOW_COUNT 10 + 20
#define WINDOW_PRICE 10.0

int main(int argc, const char * argv[]) {
    std::cout << "Cost = " << WINDOW_PRICE * WINDOW_COUNT << std::endl;
    return 0;
}
```
- Some Macros defined by systems's include file,
- Example: M_PI, M_PI_2, M_PI_4, M_E, etc....
```
#include <cmath>
:
:
        double rad = angle * M_PI / 180;
```
  - Note that M_PI etc. are not in Standard C++, but those are defined because those are taken over from C. For some system(VC++), you have to do:

```
#define _USE_MATH_DEFINES
#include <cmath>
```