i++ => increase after the expression is evaluated.

++i => increase before the expression is evaluate. (both are same in the for-loop)

a = b = c; / a = b = 3; => assign the right-most value ot all the LHS values.

M_PI => get the π. This can be converted to different type automatically in the assignment & directly output.

sin(x); / cos(x); => return the trig value, given x is a radian.

pow(2, 10); => return an INT value of 2 to the power of 10.

abs(x) (flexible return) => x can be int/long/float/double

fabs(x) (return float only) => x can be int/long/float/double.

floor(x) (return the floor int) (if x < 0, return the smaller closest INT) => x can be float/double

log(x) (log of e) => x can be float/double.

In a statement, the conversion will not be proceed withouit an explicit conversion (e.g. float y = 5/2 => y = 2 [in float].

file input: #include <fstream>

ifstream file;

file.open('…'); = ifstream file("…");

STL vectors (dynamically-sized, 1-D array) #include <vector>

define: std::vector<int> scores;

>start empty unless specified => by contrast: int[] will contain garbage when unspecified defined.

func:

vector_name.size()

vector_name.push_back('a') => push 'a' to the last position of the vector

constructions:

std::vector<int> a => empty vector

std::vector<double> b(100, 3.14) => create 100 of 3.14 in the vector.

std::vector<int> c(100*100) => create 100,000 of empties in the vector.

std::vector<double> d(b) => copy vector b to vector d. (cause error when different type)

STL sort: #include <algorithm>

usage: std::sort(vector_name.begin(), vector)name.end())

default: sorts from least to greatest.

begin/end can also be the *q pointer, direct to the array in the heap.

switch statement:

```
switch (<variable>) {
        case [variable-value]:
                ... ;
                break; <= without the break, it will go through
                all rest of the statements, until a break or the
                switch statement ends.
        default:
                ...;}
```

c++ class: (default: private)

define:

```
class class_name{
  public:
      NEED: constructor...
      default constructor: class();
      constructor with argument(s): class(1, 2)
  private:
      (only access/modified within the class)
};
```

> Only const class can call const class.

in .h file:

#ifndef <class_name_in_capital>_H

#define <class_name_in_capital>_H

…

#endif

in .cpp file:

constructor:

Class_Name::Class_Name(<variables){};

Accessor:

<return_type>& Class_Name::get() const{};

Modifier:

void Class_Name::set(<variable>){return …};

Non-member function:

write a brief one in .h; call it in .cpp just like a normal function

customise sort function:

sort(vector.begin(), vector.end(), self-define-rule);

self-define-rule should be bool function;

comparison function can add "const" and "&" in the parameters

comparison:

first > second: sort from greatest to smallest

first < second: sort from smallest to greatest

Default constructor:

every parameters have to be initialized. Including STL library.

e.g: string = ""

define the operator in the class:

in .h file:

bool operator< (const class_name& first);

in class.cpp file:

```
bool operator< (const class_name& first, cons class_name&
second){
   operating rule;
}
```

"&" should be included in all the "get" function. (attach just after the return type) int: 1 bit for sign, 31 bits for value => –(2^{31}-1) … 2^{31}+1

unsigned int: 32 bits for value => -(2^{32}-1) … 2^{31}+1

array: always fix-size

string: an array of char.

std::string string_name; => create an empty string.

std::string string_name(string2) => copy string 2 to string_name.

std::string my_string(10, ‘0’) => create a string with 10 * ‘0’

function of string: string.size() => get the size of the string (type: unsigned int)

C-style string: char h[] = “HELLO”;

STD string: std::string s1;

conversion:

C-style to STD: std::string s2(h);

STD to C-style: char h[] = s1.str();

STD string are mutable, python string is immutable.

----

multi-line comment: start: /*; stop: */; [No matter how many /*, comments end in the first */] [this type of comment can also be added during the statement]

run in the terminal:

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

Dynamic memory:

- created by "new"
- accessed by pointers
- removed by "delete"

int *p = new int; => initialize pointer p;

```
a.k.a. => int *p;
        p = new int;
```

if a defined pointer assigned a new heap "new int", this will disconnect the previous

arrow and pointer to the newly-allocated memory address.

When define a new array, the memory allocated in the heap will be continuous.

This is why we can (p = a; p < a + n; ++p) where p is a pointer (define as int * p;)

double *a = new double[n] => allocate an array in heap to the pointer a (points to the start)

delete [] a => delete the whole array in the heap. CROSSING the whole array

#include<iostream> #include<vector> #include<string> #include<fstream> #inlcude<algorithm> #include <iomanip> #include “header.cpp”

NAME: YANZHEN LU

SESSION: 02

Professor: Jasmine A.Plum

"return; ": can terminate the "void return" functions.

pointer:

  p = &x: &x memory address of x, assigning to p

  when change the value of *p, which points to x, it will also change the value of x;

when p and q are both a pointer:

  when without a "*":

    p and q are both a memory address, instead of the pointing value.

```
int *c = b;
  // b => y, c => b
  // therefore c => y
```

Array:

The name of an array is the start pointer of the array;

e.g.: a is an array;

  double* p;

  "p = a" is equivalent as "p = a[0]"

increment of the pointer in an array:

  adding the size of the datatype.

  e.g.: ++p = adding 8 bytes to the address, since p is a double, and a double worth 8 bytes

size of the array in terms of pointer;

```
  n = 10;
  double a[n];
  double* p;
  for (p = a, p < a + n; ++p){
  }
```

  "a + n" = 80 since "a" is an double array, and each size worth 8.

*: change the direction of arrow;

without *: change the actual value in the destination of the arrow.

When there are two pointers points to a same element, if the first pointer change its arrow,

the second pointer will NOT change its arrow as the first changed, but continue pointing to its original element.

delete [] a: delete the array in the heap which a direct.

delete b: delete the pointer.

2-D array:

```
double** a = new double*[rows]; => an array of pointers
 for (int i = 0; i < rows; i++) {
  a[i] = new double[cols];    => create an array with each pointer
  for (int j = 0; j < cols; j++) {
   a[i][j] = double(i+1) / double (j+1);
  }
}
```

delete process: delete the "column" first, then delete the "row"

```
for (int k=0; k < rows; k++){
  delete [] a[k];
}
```