Basics and Random

Classes → { Data + functions }

#include <~~stdio~~ iostream>

* using namespace std; → cin, cout, endl
  (is container of names)

(<<) cout → output } can use multiple in the same line
(>>) cin → input }

⇒ but we can't use both in the same line right?

'\n' ⇒ endl

return 0; → helps exit main function
↳ program

if not include *
write std::cin / std::cout / std::endl;
std::vector<bool>

Data Type

int 32 bits — short 16 bits — long >= 32 bits — long long much more

float 32 bits — double 64 bits — char 8 bits — bool 8 bits

why can't I store bool in 1 bit?
I only need 1 or 0 right?
→ because CPU can't address anything smaller than a byte
→ this is just a design choice

Arrays:

store multiple data elements of the same type

double grades [6];
double grades [6] = { };
double grades [6] = {4, 2, 3, 4};
double grades [] = {1, 2, 3, 4};
double grades [] {1, 2, 3, 4};

Multidimensional arrays

double grades [1][2][3][4]; → grades [1][2][3] = grades [6]

~~Arrays as~~

string.h functions size_t strlen (~~const~~ const char *str) ← unsigned int

C++

↳ <string>

declaration string name;

comparison name == kalash; name != kalash;

concatenate name + "kalash";

Loops

decision making : if-else

repetition : for, do-while, while

operations ① arithmetic + - ++ -- * / *= /= -\= -=

② logical bool A,B;
A||B A&&B, !A

③ relational == >= <= != > <

Functions

ret_type name (input arg) { function header
// body prototype, signature
}

Pass by value
→ value passed → copy of value passed → ends when function ends

Pass by reference
~~→ don't have a separate memory location~~
→ name / alias of the same entity passed

```
int x = 1;        int &rx = x;   // rx: another name for x; reference to integer
```

needs to be assigned and declared at the same time

Pointer → variable that holds address of another variable

```
int *p;     int x;
p = &x;
*p = x;
```

dereference operator
* pointer to
& address of
↳ reference operator

not to be confused with pointer declaration

[*(&x)] = x

int *p

* Pass by reference [function initialized before calling → passed by reference

```
void swap (int&rx, int&ry);   // didn't initialize reference variable/alias
int main(){
    int x=7, y=13;
    swap(x,y);   // call in the same way
    return 0;
}
```

changes x, y in main

reinitializing changes the value of actual variable → can't be reassigned

Multiple file program

run/execute
main.cpp → compile → g++ main.cpp → a.out → -o main (name) → main

#include "print.h" #include "input.h"

main.cpp
if needed #include <iostream>, using namespace std;

input.cpp → include → input.h → function headers go here
if needed #include <iostream>, using namespace std;

print.cpp → include → print.h → function headers go here
if needed #include <iostream>, using namespace std;

① Source files
② PREPROCESSING → removes #include <x.h> and inserts the file there
↳ .h dies here
③ COMPILATION g++ main.cpp input.cpp print.cpp -o main
source files → gives a.out → binary executable file
↳ g++ -c main.cpp g++ -c print.cpp g++ -c input.cpp
④ object files
⑤ LINKING main.o print.o input.o
g++ print.o input.o main.o -o main
⑥ executable file → main

editing 1 file? (.cpp)
compile 1 file ⟷ editing 1 file? (.h)
link all files compile all files where .h is used
run -

Header Guards

reference

```
#ifndef MACRO
#define MACRO
  .
  .
  .
#endif
```

avoid multiple function def. in preprocessing

CLASSES AND OBJECTS

user defined data type (object) → instance of class ↳* independent
class: template of object (just like structs)
~~or blueprint~~

definition of class → inside header file
but also make a constructor in .cpp (the default constructor)
↳ always called when we make object

abstraction
↳ hide info? private: public:
↳ inside a class ↳ outside a class

outside class:
dclass :: print() { ... } for eg.
defining and creating object or including namespace student

```
class A {
    int B = 2;        getB();        A();  constructor
};
```

```
int main() {
    A x;
    A y;
    x.B; → 2
unless if B is private
    x.getB();
```

Constructor

→ default constructor (automatically called at time of instantiation)

→ must be public

→ same name as class

→ doesn't return anything

:: scope operator

Student :: Student() {...} → default constructor

↳ class ↳ constructor

Student(int x, string n); → extra [function overloading]

↳ I can call Student(2) as well [not enough arguments] ✓ ↳ same name, diff parameters

Student x; constructor called

student y[10]; constructor called 10 times

array of objects

student *z; constructor NOT called

no obj created, stores only address

Dynamically Allocated Memory within an object

Heap	dma (malloc-free) need to free
Stack	local var automatically deleted
Data	const or global variables
Code	instructions: 0/1

memory leak: didn't free it, can't access it

pointer: `p = new int;` — can do `*p = 11;` ✓

`delete p;` the address returned to the OS

good practice? → `p = NULL;` → because p is still storing the same address

* BUT what happens when I dma in object? → destructors

array
```
int *arr = new int[10];
   :
delete [] arr;
arr = NULL;
```

integer
```
int *p = new int;
   :
delete p;
p = NULL;
```

Destructors

→ automatically given by compiler
→ called when object goes out of scope or is destroyed
→ empty if not defined

~Student();

→ only 1 destructor (for all constructors of the class)
→ ~~take~~ no parameters, no returns
→ should be public

```
Student :: ~Student() {
    if (grades != NULL)
        delete [] grades;
}
```

nothing happens if I try delete NULL but to avoid any unnecessary trouble delete if not NULL

When is a destructor called?
→ obj goes out of scope
→ obj destroyed on heap
→ called recursively when object is a pointer to dma

```
Student x = new Student;
delete x;
x = nullptr;
```

I/O

forcing write into terminal/file from buffer

	objects	class	
<iostream>	cout	ostream	.flush()
	cin	istream	
<fstream>	inFile	ifstream	.close();
	outFile	ofstream	.close(); .flush()
	↳ overwrite		↳ flush out
	↳ ios::app appends (don't overwrite)		↳ privacy

ofstream outfile ("file.txt", ios::app);

you ~~don't~~ can also write path of file and it can overwrite it, doesn't need to be a file in the same directory

CASE:

invalid input 13.7 (int x, int y)

cin>>x >>y; all cins will silently fail

↓ 13 ↓ ~~uninitialized~~

FAILURE:

1. file DNE
2. incorrect input
3. file empty

cin.fail() → bool → true if failed / infile.fail() (object)

→ cin.clear() → clears failure flag for next use

infile.eof(); cin.eof();

don't change the order (clear→ignore)

• → cin.ignore (int n, char c); eg. cin.ignore(3, '\n')

remove/discard ~~upto~~ n characters

upto n characters

String Streams

```
#include <sstream>
#include <string>
{ string line;
  stringstream ss(line);   → string to stringstream
  ss.str();   → stringstream to ss
```

input:

```
getline(cin, line);
          ↳ the string goes here
```

Operator Overloading

\+ − * / % = != == >> <<

Complex
↑ return type
Complex :: operator + (Complex &rhs) {
    Complex temp;
    rhs.real=0; # compile-time error!
    temp.real = rhs.real + real;
    temp.img = rhs.img + img;
    return temp;
}

operator +

x.operator+(y);
function member of class of x

protects y ← const Complex & rhs) → protects x

```
Complex Complex :: operator+(const Complex & rhs){

    return Complex(real+rhs.real, img+rhs.img);
}
```

Lec 15 (watch video)

Copy Constructor

Automatically called when
→ student a(b); } obj of student
→ student a = b; (not when student a; a=b;)
→ passing obj by value
→ returning obj by value

```
class student {
    student (const student &other) {
        name = other.name;
        id = other.id;
    }
};
```

* must pass by reference or copy constructor would be called infinitely

→ shallow copy

```
Mystring:: Mystring (const Mystring & other) {
    buf = new char[100];
    strcpy(buf, src.buf);
    len = src.len;
}
```

→ deep copy

Class requires a destructor, copy constructor, assignment operator=

Deep Destructor:

```
Mystring::~Mystring() {
    if (buf != nullptr) {
        delete [] buf;
        buf = nullptr;
    }
}
```

~~Deep Assignment:~~
~~Mystring& Mystring::ope~~

Deep Assignment

```
Mystring& Mystring::operator=(const Mystring &src){
    if (this == &src) return *this;

    delete []buf;
    buf = new char[src.len+1];
    strcpy(buf, src.buf);
    len = src.len;
    return *this;
}
```

Rule of three

if it requires any one, it requires all 3

- destructor
- copy constructor
- operator=

class
Abstract function {

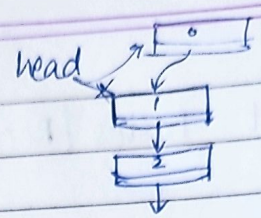
has one (at least) virtual function
↓
* Base class
Compiles derived class function

}

purely virtual function → never called (virtual void f() = 0)
(void virtual f() = 0)

→ it is slower than calling a non-virtual function

Stacks



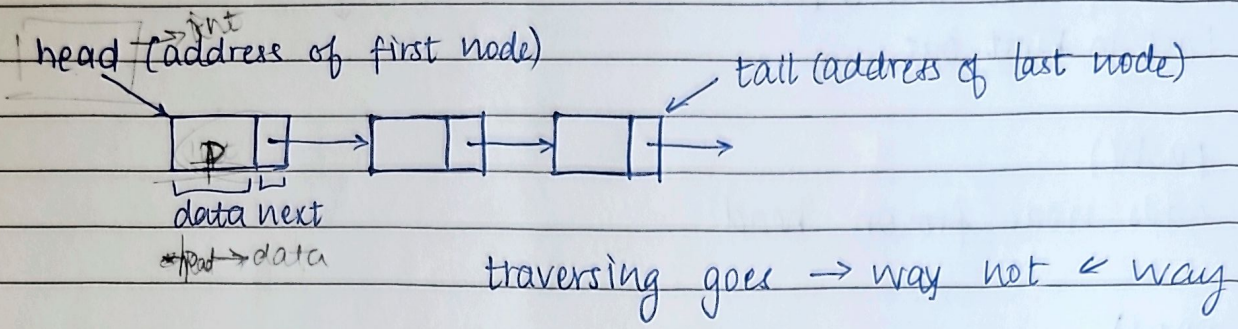
kind of like linked list
Last in First out

push()
adds node ~~to~~ on head

pop()
removes node from head

```
class    stack {
        private:
          Node*   head;
        public:
          stack() {head = NULL;}
          ~stack() { delete head ; }
                    ↳ takes care of deleting next nodes

          void  push(int d) {
        Node* p = new Node (d, head);
        head = p;
        }
          int  pop() {
        if  (head == NULL)  return -1;
        Node *p   =  head;
        int d  =  p  → getData();
        head  =  p  → get Next();
        p → setNext (NULL);   *** prevent deleting next node
        delete p;
        return d;
}
```

Linked List Basics / Recap



traversing goes → way not ← way

Node, as defined in all linked lists

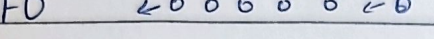
```
class Node {
private:
    int data;
    Node* next;
public:
    Node() { data = 0;  next = NULL; }
    Node(int d) { data = d;  next = NULL; }
    Node(int d, Node* n) { data = d; next = n; }
    ~Node() { delete next; }

    int getData() {  return data; }
    Node* getNext() { return next; }
    void setData(int d) { data = d; }
    void setNext(Node* n) { next = n; }
};
```

Queues

~~class Node {~~

enqueue(): add node to tail

dequeue(): remove a node from head

FIFO 

```
class Queue {
    private:
        Node * head;
        Node* tail;
    public:
        Queue() {head = NULL;  tail = NULL;}
        ~Queue() {delete head;}

        void enqueue (int d){
            Node * p = new Node (d, NULL);
            if (tail != NULL) {
                tail -> setNext(p);
            }
            tail=p;
            if (head == NULL) {head = p; }
        }
        int dequeue () {
            if (head == NULL) return -1;   if(tail==head) {tail==NULL;}
            Node * p = head;
            head = p -> getNext();
            if (head == NULL) {tail = NULL; }
            int d = p -> getData();
            p -> setNext (NULL); ***
            delete p;
            return d;
        }
};
```

Ordered Linked Lists

```
#include "Node.h"

class list {
    private:
        Node* head;
    public:
        List () {head = NULL;}        constructor
        ~List () {delete head}        destructor

        bool dataExists (int d) {
            Node *p = head;
            while (p!=NULL && p->getData()<=d) {
                if (p->getData() == d)
                    return true;
                else
                    p = p->getNext();
            }
        return false;
        }

        bool list :: insertData (int d){
            Node *n = new Node (d);
            Node *p = head, *prev =NULL;
            if (p== NULL)  head =n;
            while (p!=NULL && p->getData() < d ){
                    prev = p;
                    p = p->getNext();
            }
        }n->setNext (p);
```

```
if (prev == NULL) {
        head = n;
} else {
        prev -> setNext (n);
}
```

special cases to remember in lists
→ list = empty, head == nullptr
→ List has one node, head->getNext() = nullptr
→ if insert at end/tail node->getNext()=nullptr
→ if insert at head, node->getNext() = head

```
bool list :: deleteData (int d) {
        Node *p = head,  *prev = NULL;

        while (p != NULL && p->getData() != d) {
                if (p->getData() > d)  return false;
                prev = p;
                p = p -> getNext();
        }

        if (p == NULL)  return false;
        if (prev == NULL) head = p->getNext (); -> delete at head
        else  prev->setNext(p->getNext()); -> delete mid/tail

        p->setNext(NULL)
        delete p;
}
```

why prev, why not next
→ can use next ✓

Why can't we do a shallow copy?

→ because shallow copy will only copy addresses (pointers) but not the actual list

Do a deep copy of the list, all nodes one by one

```
List:: List (const List & original) {
        Node *p = original head;
        Node * np = NULL;

        head = NULL;
        while (p != NULL) {
                Node*n = new Node(p -> getData(), NULL);
                if (np == NULL) {
                        head = n;
                } else {
                        np -> setNext (n);
                }
                p = p -> getNext();
                np = n;
        }
}
```

~~I don't understand this~~

Grilled Cheese
Obama sandwich

operator =

```
List& List::operator=(const list &original) {
        if (&original == this) {
                return *this;
        if (head != NULL) {
                delete head;
                head = NULL;
        }
```

now same as copy constructor body:
(but why?, and what's LHS, why can't it be empty)

```
        Node *p = original.head;
        Node *np = NULL;
        while (p != NULL) {
                Node *n = new Node(p->getData(), NULL);
                if (np == NULL) { head = n; }
                else { np->setNext(n); }
                p = p->getNext();
                np = n;
        }
    return *this;
}
```

Recursion

print nodes in linked list recursively

```
class Node {
    public:
        int data;
        Node* next;
};

void rprint (Node *p) {
    if (p == NULL) {return;}
    else {
        cout << p->data;
        rprint (p->next);
    }
}
```

or p->getNext();?
no because it is already public

Backtracking

```
bool solveMaze(int row, int col) {

    if (row < 0 || row >= height ||
        col < 0 || col >= width) {
        return false;
    }

    if (maze[row][col] == 'E') {return true;}
    if (maze[row][col] != ' ') {return false;}

    maze[row][col] = '*';

    if (solveMaze(row+1, col)) return true;
    if (solveMaze(row-1, col)) return true;
    if (solveMaze(row, col+1)) return true;
    if (solveMaze(row, col-1)) return true;

    maze[row][col] = ' ';
    return false;
}
```

out of bounds function

reached exit

reached wall/ path again

already on path

none of the 4 directions are valid steps → stuck.

Why make it a whitespace, isn't it already one?
→ ohh we defined it as * before

Binary Trees

Binary Tree:

→ root → top, leaves → bottom
→ each node stores some data
→ max 2 children nodes, 1 parent (except root got none)
→ 0 children → leaf node

Binary Search Tree:

Node
left < ↓ < Right

min value: leftmost of node
max value: rightmost of node

Preorder : CLR ↙
Inorder : LCR ↑
Postorder: LRC ↘

```
class BSTNode {
  private:
    int value;
    BSTNode * left, *right;
  public;
    BSTNode (int v) { value = v; left = right = NULL; }
    ~BSTNode () { delete left; delete right; }

    int getValue () { return value; }
    BSTNode* getRight () { return right; }
    BSTNode* getLeft () { return left ; }
    void setRight (BSTNode *r) { right = r; }
    void setRight (BSTNode * l) { left = l; }
};
```

```
class BSTree {
private:
    BSTNode* root;
    bool searchNode(int V, BSTNode *n) {
        if (n == NULL) return false;
        else if (n->getValue() == V) return true;
        else if (n->getValue() > V)
            return search(V, n->getLeft());
        else
            return search(V, n->getRight());
    }
    void insertHelper(int V, BSTNode* n);
    void printInOrderHelper(BSTNode *n);
public:
    BSTree() { root = NULL; }
    ~BSTree() { delete root; }
    BSTNode* getRoot() { return root; }

    bool search(int V) {
        return searchNode(V, root);
    }
    void insert(int V);
    void printInOrder();
};
```

```
void BSTree :: insertHelper (int v, BSTNode * n) {
    if (n -> getValue() == v) return;   no two nodes can have the same value
    else if (v < n -> getValue()) {
        if (n -> getLeft() == NULL)
            n -> setLeft(new BSTNode (v));
        else
            insertHelper (v, n -> getLeft());
    }
    else {
        if (n -> getRight() == NULL) {
            n -> setRight (new BSTNode(v));
        else
            insertHelper (v, n -> getRight());
}

Void BSTree :: insert (int v)
    if (root == NULL)
        root = new BSTNode (v);
    else
        insertHelper (v, root);
```

```
void BSTree:: PrintInOrderHelper (BSTNode * n)

    if (n != NULL) {
        printInOrderHelper (n -> getLeft());
        cout << n -> getValue() << " ";
        printInOrderHelper (n -> getRight());
    }

void BSTree:: PrintInOrder() {
    return printInOrderHelper (root);
}
```

why can't we directly call this and define all lines from above in this?

```
void BSTree:: printPreOrderHelper (BSTNode * n)

    if (n != NULL) {
        cout << n -> value << " ";
        printPreOrder (n -> getLeft());
        printPreOrder (n -> getRight());
    }

void BSTree :: printPostOrderHelper (BSTNode *n)

    if (n != NULL) {
        printPostOrder (n -> getLeft());
        printPostOrder (n -> getRight());
        cout << n -> value << " ";
    }
```

```
BSTNode*    BSTree :: minValueNodeHelper(BSTNode* p) {
        if (p != NULL && p->getLeft() != NULL) {
            return minValueNodeHelper(p->getLeft());
        else
            return p;
}

BSTNode*  BSTree :: minValueNode() {
        return minValueNodeHelper(root);
}

BSTNode*  BSTree :: maxValueNode() {
        return maxValueNodeHelper(root);
}

BSTNode*  BSTree :: maxValueNodeHelper(BSTNode *p) {
        if (p != NULL && p->getRight() != NULL)
            return maxValueNodeHelper(p->getRight());
        else
            return p;
}
```

Delete Node

Cases:

- no children
- one child
- two children
- occupied parent

```
BSTNode* BSTree :: deleteNode (int v, BSTNode * node)
    if (node == NULL) {return NULL;}
    if (v < node -> getValue())
        node -> setLeft( deleteNode (v, node ->getLeft()));
    else if (v > node -> getValue())
        node -> setRight (deleteNode (v, node -> getRight()));
    else
    { if (node -> getLeft() == NULL) {          one or no children
        BSTNode* temp = node -> getRight();
        node -> setRight(NULL);
        delete node;
        return temp;                            one child on left
      } else if (node -> getRight() == NULL) {
        BSTNode * temp = node -> getLeft();
        node -> setLeft(NULL);
        delete node;
        return temp;
      } else {
        if (node -> getLeft() == NULL) {
            BSTNode * temp = node -> getRight();
        BSTNode * temp = minValue (node -> getRight());
        node -> setValue (temp -> getValue());
        node -> setRight (deleteNode (temp -> getValue()
                                      node -> getRight());
        return node;
    }
```

Inheritance

Parent

Child
Parent: Public ✓
Parent: Protected ✓

Friend
Parent: Public ✓
Parent: Protected ✓
Parent: Private ✓

Class acquires properties of parent class.

Person.h (base class)

```
class Person {
    private:
        string name;
        int age;
    public:
        Person() { name = ""; age = 0; }
        Person(string n, int a) { name = n; age = a; }

        void setName(string n) { name = n; }
        void print() {
            cout << "Name:" << name << endl;
            cout << "Age:" << age << endl;
        }
};
```

Student.h (derived class)

```
#include "person.h"
```

→ makes everything within a class public unless specified

```
class Student: public Person {
    private: int ID;
    public: Student() { ID = 0; }
            void setNameID(string n, int d) {
                Person::setName(n);
                ID = d; }
            void Print() { → overrides print() from Person.h
                Person::print();
                cout << "ID:" << ID << endl;
            }
};
```

can't access name and age here (private)

do they → show?

in main.cpp file

```
#include "person.h"
#include "student.h"
using namespace std;

int main() {
    Person P("Joe", 23); // Person();
    Student S;           // Person() & Student();

    P.setName("Joseph");
    P.print();           //from Person

    S.setName("Ryan");   // function inherited & used
    S.print();           //from Student

    S.setNameID("Marina", 125);
    S.print();

    return 0;
}
```

just person ←

student and person ←

Then What's Membership?

```
class Student {
    private:
        Person P;
        ...
};
```

Want to call different Person constructor when I create a Student object

```
Student (string n, int a, int d): Person (n, a) {
                                  initializer's list
    id = d;
}

Student s("Kalash", 20, 354);
```

Data protection

public protected private

protected
↳ inherited
↳ accessible to derived classes (only)

* We do not inherit
1) constructors ← you can call them
2) copy constructors } should create new
3) operator=
4) destructors

watch lec 31 video

Pointers to dynamic memory in derived class

make operator= and copy constructor

Polymorphism

rectangle : rectangle & polygon p=r ✓
polygon : only polygon r=p ✗

Virtual Functions (run time polymorphism)
dynamic binding

~~Compile Time~~
Non-Virtual Functions (Compile time polymorphism)
static binding

make destructors virtual

Pure Virtual Functions abstract class

Hash Tables

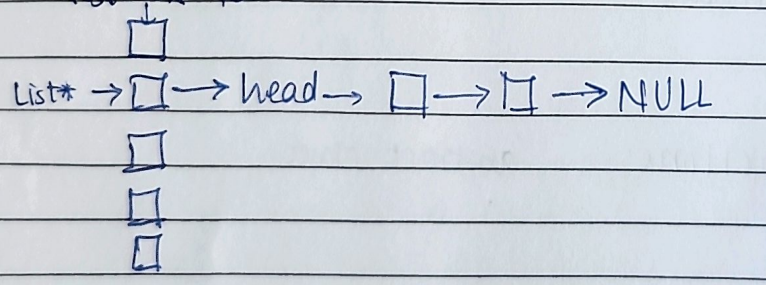
Very large Data sets
Stores randomly/ doesn't store empty boxes
making search O(1) and storage O(k)

division operation: $h(k) = k \% (m+1)$

We need to avoid collision (good hashing table)
Chaining
each box is a pointer to linked list keys stored at the same entry

multiplication operation $h(k) = |k * 31| \% m;$
large prime

list ** table



linear probing → find next free slot
avoid collision by inserting value at next available space in table
if (collides) try (next)
h(k) (h(k)+1) % sizeoftable

quadratic probing:
(hash(v) + i*i) % size

double hashing:
(hash1(v) + i hash2(v)) % size

Rehash example