No.

Date

OH. T 9~12 a.m. 220 FGH Basis of C++

① ; at the end of .cta .h file

② ClassName :: function(){}

③ C++ is a collection of functions

④ bool: false: zero true = not zero

⑤ char: char* / char[] strings

string → objects, methods

#include <string>

⑥ variable can be initialized when declared

int num=0; int value(5);

⑦ use constant const modifier to declare constants

const double SALES_TAX(0.0525);

★⑧ in Java, primitive type → value variable

class type → reference variable

in C++, all variables are value variable

C++: int i; → i

i=5; → 5

Java: Point P; → P

P = new Point();

Point P; → Point Obj.

created when declared

→ ⊕ Console i/o:

#include <iostream>;

>> extration operator

<< insertion operator

→ break: allows us to end a loop

continue: allows us to go to next iteration

→ ! Warning:

assignment stat returns a value, which is the value that got assigned.

if(x=y){ → assignment value.

}

no error in C++, error in Java

- fundamental data type:

Character

Numerical integer { signed / unsigned

Floating-point

Boolean

void

NULL

- 16-bit unsigned integer

[0, 65535]

16-bit signed integer

[-32768, 32767]

[-2^{15}, $2^{15}-1$]

character sequences

- *** the null character, '\0'

char myword[] = {'H','i','\0'}

" char myword[] = 'Hi';

Assume mystext is a char[]

mystext = 'Hi';

or mystext = {'H','i','\0'};

Is not valid

or mystext[] = 'Hi';

X i.e. assigning values after declaration

1~10 automatically appended

→ cin and cout support null-terminated sequence as valid containers... so they can be used directly to extra strings of characters from cin or to insert them into cout

e.g.

char yourName[80]; //have to specify the size

cin >> yourName;

cout << yourName;

Input: Xue

Output: Xue

No.

Date

* const method: tells the compiler that it should not change any member variables in the object
// (and can't call ~~const~~ regular // method on a const objects)

if (x==y){ → equality operator
...
}

→ typedef
allows us to declare a new data type
typedef oldtype newtype;
e.g. typedef int MyInt;
MyInt num;

→ sequences of char → string objects
e.g. string mystring;
char myntcs[] = "some";
mystring = myntcs;

→ function:
- Java contains collection of classes.
C++ contains collection of functions.
- can be methods of a class
- free function
△ main class is always free
- declaration, definition (.h or .cpp)
- * class methods can be const methods ?

→ Parameter Passing 3 ways
- pass by value (default) /call
- called function receive a copy of the value being passed
* Modify copy won't affect original
- pass-by-reference /call
- called function receives a reference of the parameter
- any modifications would be seen by caller
- e.g. void change (int pv, int & pr)
- call/pass-by-const-reference
- passing large objects by value is expensive
- but passing objects by reference allows others to change
- compromise
- pass a reference while compiler ensures it is not changed

No.
Date

No.
Date

-e.g. void type (const) Graph (&) gr)
const modifier type ptr name

→ C++ variable

-no initialization to default value
-using uninitialized value is allowed ? (LOCAL)** Indeterminate → undefined Behavior
- gives a warning still generate codes
-can have global const BAD *

* (every function has access to these, hard to figure out which functions actually read and write these variable (and make the application hard to understand)

** static variable (file scope and function static) are initialized to zero:
Non-static variables (local variables) are indeterminate.

→ Assertion

- when we make assumptions, it's good to verify them

```
#include <cassert>
// assume j is nonzero
assert(j>=0);
// if j<0, (exception is thrown) terminate the program
```

→ C++ classes

-class declaration goes in .h file
-definition goes in .cpp file
-e.g.

```
#ifndef CLASSNAME_H   /* if the class isn't defined, process following codes */
#define CLASSNAME_H
class ClassName{
  private:
  public:
};
#endif                process code H to here
```

- the .h file provides interface to the class
-users can't access staff

No.

Date

→ ☆ class ctor. Basic Member Initialization

- initialize data members when they are created as opposed to using assignment in body of ctor
- e.g. Point::Point(): x(10), y(0){
  //nothing to write
  }
  Point::Point(int x, int y): x(x) y(y){}

- used when parameter are objects

```
Person::Person(): name("Irene"),
                  age(18),
                  birthday(1,1,2000){
    //other work initialize here ?
}
```

- NOTES
  - for data members of primitive type, initialization vs assignment is not a big deal
  - for data of class type
  - always use BMI
  - important to order BMI in some order that fields are declared in class because that is the order they are executed

→ Arrays

- store a sequence of some type elements with direct access to any one element by index
- index is zero-based
- Type name[size]; ☆ const value non-dynamic memory whose size must be determined before execution.
- Access reference: name[index]
- primitive type, x class
  - no method - ~~no ins~~ (no instance variabe)
- stored in contiguous memory locations
  ☆ array name stores a reference to 1st element

also int a[]={3,4,5}
//a would be assumed size 3

e.g. int test[3]={1,2,3};
std::cout << *test; //print 1

No.

Date

- if you write score[5]=0 it still change the ...
  → Buffer Overflow Problem ★ *
  → not caught by run system (x out of range error)

(leftover from primitive array in C)
★ do not cause out-of-range problem, but can cause run-time error
Reason: ptrs ?
our reasbonsibility to...

- passing array to function, function header must specify that it is receiving an array parameter
  void process(int arrayForm[])
  → the size is not inside []
  → ★ the size, if needed, must be passed as an additional parameter
  void process (int array[], size_t size)
  → arrays are passed by reference, the array name ★ contains a reference to array in memory

* an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations

→ Top-concern in CS: correctness
- many ways to correctly solve a problem. Then other concerns: - efficiency
  - maintainability
  - resources used
  - time to implement
- e.g. find the missing one in range [0, n-1], n unique elements
- Solution 1: scan range for one ... two, need n^2 times, $O(n^2)$
- Solution 2: sort array
  scan it for missing array
  $O(n \lg n)$?
  *Mergesort $O(n \lg n)$
- Solution 3: create boolean array of size n and initialize to "false"
  - do linear scan of set A[] and set B(A[i]) to be "true"
  - Scan B to find "false" → return A
  → linear O(n) + size n extra memory

No.

Date

-solution 4: - sum all elements in range
-substract from expected value $\frac{n(n-1)}{2}$
-linear (n) solution

- Selection Sort $O(n^2)$
-find smallest & swap it with 1st element
-repeat

```
typedef int ItemType;
void SelectionSort (ItemType a[], size_t size){
    for (size_t i=0; i<size-1; i++){
        size_t indexMin= findSmallest(a,i,size);
        std::swap (a[i], a[indexMin]);
    }
}

size_t findSmallest ( ItemType a[],size_t start, size_t size){
    ItemType min (a[start]);
    size_t indexMin (start);
    for (size_t i=start+1; i<size ;i++){
        if (a[i] < min){
            min= a[i];
            indexMin=i;
        }
    }
    return indexMin;
}
```

- operations: main loop executes :size-1" times
- each iteration :"find smallest" and "swap"
↳ "size-(start+1) times 3 assign-ments

- 1st call: size-(0+1) → size-1 iterations
2rd : size-(1+1) → size-2
3rd size-3
last size-(size+1) ⇒ 1

TOTAL: (size-1+1)(size-2)/2

- the "findSmallest" performs $\frac{size(size-1)}{2}$ iterations
  ↓
  1 test + 2 assignments

- TOTAL numbers of operations for selectionSort
  $\frac{size(size-1)}{2} + 3(size-1) = \frac{size^2}{2} + \frac{5size}{2} - 3$ Big-Oh
  ↓
  since $size^2$ grows faster than other terms, we call it $O(n^2)$
  - O gives us a convenient way to discuss growth rate of an algorithm in terms of the size of problem it is to solve.

→ common Big-Ons
- constant time $O(1)$
- $\log_2$ time $O(\lg n)$
- linear time $O(n)$ combination $O(n \lg n)$
- quadratic time $O(n^2)$
- cubic time $O(n^3)$
- exponential time $O(2^n)$

→ Bubble Sort $O(n^2)$
- works by comparing adjacent elements in array, and swapping if out of order
- ☆ After one pass, the largest value has bubbled up to last location of array
- multiple passes are required, start each pass at beginning of array but don't have to go all the way to the end
  - don't check the last
- Must we perform all passes? No. If any pass doesn't perform a swap; then we know array is sorted.

```
void bubbleSort(ItemType a[], size_t size)
{
    bool sorted = false;
    for(size_t pass = 1; pass<size && !sorted; pass++){
        sorted=true;
        for(size_t i=0; i<size-pass; i++){
```

```
            if (a[i] > a[i+1]) {
                std::swap(a[i], a[i+1]);
                sorted = false;
            }
        }
    }
}
```

- operations of bubbleSort : size-1 passes
- How many operation on pass 1 ?

on pass 1 size-1 comparison and up to size-1 swaps

pass 2 size-2 comparison and up to size-2 swaps

pass i size-i comparison and up to size-i swaps

TOTAL operation: $(size-1)+\dots+1 = \frac{size(size-1)}{2}$ major passes

WORST case ↗ → $4\times\frac{size(size-1)}{2}$ (+3 data moves)

best case size-1 → O(n)

Typically never look at the best case analysis

Searching:

Sequential search: look for an item in array, and return its index or -1 if not found

```
int seqSearch (ItemType a[], size_t s,
               ItemType target){
    for (size_t i=0; i<size; i++){
        if (a[i] == target){
            return (int) i;
        }
    }
    return -1;
}
```

- Each iteration, perform one comparison → only need to see number of iterations,

Best: O(1)

worst: size: O(n) Average: size over two → O(n)

No.

Date

- Do better: binary search

if array is sorted

look at middle element and see if is the target, if not continue search on the half array

→ O(nlgn)

→ used when have lots of searches, Sort array and then search

→ int binSearch (ItemType a[], size_t s, ItemType t){

```
//pre: array sorted
//post: return...
int mid, first(0), last (int)(size-1)
while (first <= last){
    mid= (first +last)/2;   *
    if(a[mid]= target){ return mid;
    }else if (target < a[mid]){
            last=mid-1;
    }else {
            first=mid+1;
    }
return -1;
}
```

* if last too big, integer overflow

better: $mid = first + \frac{last-first}{2}$

→ Analysis:

worst case (unexist)

1st pass: cut half $\frac{1}{2}$

2 $\frac{1}{4}$

3 $\frac{1}{8}$

i $\frac{1}{2^i}$

stop when we get to a single element => $\frac{1}{n}$ array

$\frac{1}{2^i} = \frac{1}{n}$ ∴ $i = \lg n$, perform lgn iteration

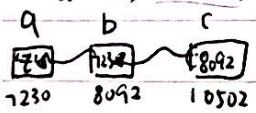
∴ O(lgn) algorithm;

- a bite = 8 bits (a memory cell)
- do not confuse pointer declaration with dereference operator (different things with the same sign) (functions)

- int *p1, p2 ≠ int *p1, *p2
∵ precedence relationship

→ const char * terry = "hello";
initialize the content at which the pointer points with constants at the same moment the pointer is created
*(terry+4), terry[4] → '0'

- Pointer to pointers
add an * for each level
```
char *a;
char *b;
char **c;
a='z'; b=&a; c=&b;
```



const int ptr;
declares ptr a pointer to const int type

int*
const ptr;
declares ptr a conster pointer to int type;

→ Pointers: (Memory Address)

→ Any variable is pointer which points to allocated array

→ Pointer variables:

→ a variable must be declared to have a pointer type (pointer) and must declare what (type) of value it will point at

→ use * to declare pointer. (reference) eg. int *p1;

→ use (&) to get the address of a variable which can be assigned to a pointer variable

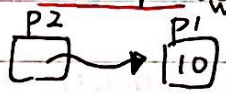
A variable referenced with && can be dereferenced with *

→ to access the memory to which a pointer variable points, we must dereference the pointer → use "*" to dereference "value pointed by"

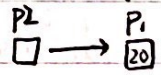
→ NOTE: you can take address of any variable, but can only deref pointer variable

eg. int p1(10);

* int *p2(&p1); = int p1(10); int *p2; *p2=&p1; not *p2=&p1;



→ std::cout << p2 << std::endl;
//point out p2's address
std::cout << *p2 << std::endl;
//point out p1's value 10. (dereferenced)
*p1 = 20;



std::cout << *p2 << std::endl; //20

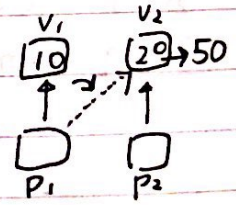
*p2 = 50

std::cout << *p1 << std::endl; //50

// keep track of what's going on

→ Consider:

```
int V1, V2 = *p1, *p2;
V1=10; V2=20;
p1=&V1; p2=&V2;
cout << *p1 + *p2 << //30
p1=p2;
cout << *p1 + *p2 //40
*p1=50;
cout << *p1 + *p2 //100
```

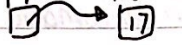


No.

Date

→ Dynamic variable: ☆ New & delete

- Pointers give us ability to point at variables even if that variable don't have name;
- such variable come into existence at run time ☆ (eg. user input)
  as opposed to compile time



cin >> *p1;
*p1 = 10; *p1 + 10; //not *p1 += 20;
can initialized these variables when created

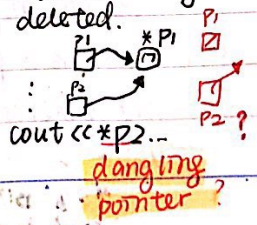
→ NULL Ptr

- special constant pointer value that should be assigned to any parameter that is currently not used.
- All pointer variables should hold nullptr when not in use
- Always use nullptr than literal zero or older NULL. ☆
- sidenote:
  when a new operation fails, it throws an exception (desired)
  older compiler don't throw exception but rather return a null pointer
- when done using a dynamic variable, it's important to return the memory to runtime system so that it can be reused.
  → use: delete operation on pointer

eg. int *p1;
p1 = new int[17];
⋮
delete p1; //not *p1
delete what p1 points at),
p1 = nullptr;

→ must be careful, never to ~~over~~ a dynamic variable when it has been deleted.

eg. int *p1;
p1 = new int[17];
p2 = p1;
delete p1;
p1 = nullptr;



→ pointer = new type
pointer = new type [number_of_elements]

→ int *bobby;
bobby = new int [5];

bobby // dynamically assigns space

OLDER:
C++ check if the allocation was successful:
{ handling exceptions, when bad_alloc fails
nothrow (returned by new is a null pointer)

→ delete pointer;
delete[] pointer;

Date

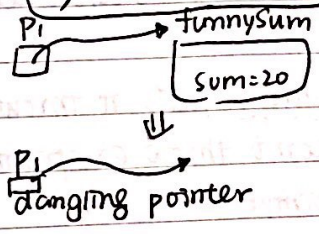
→never attempt to delete a dynamic variable twice;

```
delete p1;
p1=nullptr;
delete p2;
```
→ X

NOTE: It's ok to call delete on a pointer holding nullptr;
→ just does nothing

- ordinary variables (local variables in function)
→ their new & delete are done automatically by compiler.
→ never to use the address of an automatic variable after it goes away

eg.
```
p1= funnySum(10,20);
*p1= *p1+10;
std::cout<<*p1<<std::endl;
int funnySum(int v1, int v2){
    int sum;
    sum= v1+v2;
    return &sum;
}
```



→common to use typedef in declaring pointer types

eg. typedef int* IntPtr;

IntPtr p1, p2;

{ *int* p1, p2 => * p1 is ptr, p2 is int
  &int *p1, *p2 => p1, p2 are ptrs

ps: maxSize
this→maxsize
pointer: we
reference: compiler

→this

difference between pointer and reference:
- a pointer can be re-assigned any number of time while a reference cannot be re-seated after binding

Java: within methods of a class, if you want to refer to the obj on which method called.

C++: "this" is a pointer to obj (not a reference?)
- must dereference this ptr to access object, use * operator (*this)

No.

Date

- to access a field in object, use dot notation

Problem: dot has higher preferences than *

write: (*this).x

or

* this -> x ← always use this

~~(x must be ptr~~

-> used in compare two ptr's function)

- optional to use "this" in function

char c='10'; equal c=b;

Arrays and pointers ★

- arrays are implemented by pointer, one can treat array variables as pointer and vise versa.

e.g.

```
int score[5];
int *p;
p=score; //same.
for(size_t=0;i<5;i++){
    p[i]=0;
}
```

whereas //numbers will always point to 1st element "const pointer".

- pointer arthemetics

*p → access to score[0] (also a dereference operator offset operator)

*(p+1) → access to score[1]

*(p+i) → access to score[i] //reason why index start with 0 part of

*p++ (++ higher precedence) the expression is evaluated with the value it had before being increased

increase the value of p, (so it now points to the next element), but ++: postfix, the whole expression is evaluated as the value pointed by the original reference

Different! (*p)++

Dynamic Array

- since array and pointer are associated, we can create dynamic array by using pointer

- 4 STEPS:

1) declare pointer variable
2) use new and specify a size
3) use ptr as an array
4) DELETE~

step 2 and 4 require us to use [] "to"

e.g.

1) typedef int* IntA;
   IntA *p;
2) p = new int[arraysize];
3) for (size_t i = 0; i < arraysize; i++) {
       p[i] = ...;
   }
4) delete [] p;
   p = nullptr;

- using array as in a class:

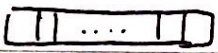
① e.g. (large number),

```
const size_t SIZE=50;   can be changed
class BigNum {
  private:
    int myD[SIZE];
  public:
    ...
};
```

Q: How do we lay out digits of our number on the array?

A: store ones digit at index 0

10^{index} power to the ones digit



myD → can be seen as a large number

② want to give the ability to allow users to specify size at runtime (use a default size)

eg. const size_t DEFAULT=50;

```
class BigNum{
  private:
    size_t maxSize;
    int *myD.  // order is important
  public:
    :
}
//default ctor
BigNum::BigNum():
        maxSize(DEFAULT)
        myD(new int[maxSize]){
    for(...) //set array element to zero
}
//Alt ctor - specify by user
```

No.

Date

pass by reference

```
BigNum::BigNum(const std::string & int):        -overflow
        maxSize(int.length()), myD(new int[maxSize]){   underflow
    for(size_t i=0; i<maxSize; ++i){
        char temp = int[maxSize-1-i];
        myD[i]= temp-'0';   ★ (ASCII)
    }}
```

→ <stdexcept> < logic errors / Runtime errors

```
throw ExceptionClass(stringArguments);
try{
    <statement(s) that might throw an exception>
}catch(ExceptionClass Identifier){
    <statement(s) that react to an exception of type
    ExceptionClass>
}
```

eg: .h file:

```
if(!inRange(index))
    throw std::out_of_range("Index out of Range");
```

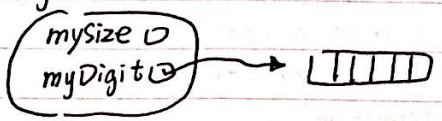
test file:

```
try{
    std::cout<<"Expect an\"out_of_rang\" exception."
            <<std::endl;
    std::cout<<"1..."<<std::endl;
    dna_1.set('A',40);
    std::cout<<"EXCEPTION ERROR -- YOU SHOUID NOT
            SEE THIS MASSAGE"<<std::endl;
}catch(std::out_of_range &excpt){
    std::cout<<"Exception was properly thrown
            and caught::"<
    std::cout<<excpt.what()<<std::endl;
}catch(...){
    std::cout<<"EXCEPTION ERROR - set()
            threw the wrong exception."
            <<std::endl;
```

由 扫描全能王 扫描创建

Q: What does a BigNum object look like?

BigNum



∴ NOTE: Dynamic array is not inside objects

· has implications on our design of a c++ class

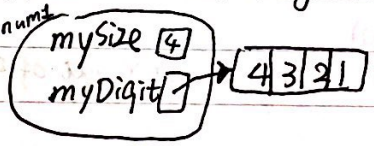
→ Whenever we design a class that uses dynamic memory allocation, we must address "the law of Big 3" (old)

© consider the following:

```
void foo(){
    BigNum("1234");    //uses Num 1
}          ^num1
```

What happens when foo() ~~exists~~ exits?

All local variable goo away.



When this object goes away, the compiler cleans up what is allocated. → things on the bubble

But dynamic array is left in memory.

⇒ MEMORY LEAK - BAD

→ ↔ C++ allows "destructor" which gets executed whenever an object is destoyed

→ if you don't define one, the compiler gives you one

↔ only clean up the bubble.

① DTOR

→ .h file

→

```
Class BigNum{
  public:
    ~BigNum(); //no return type, no parameter
};
```

No.

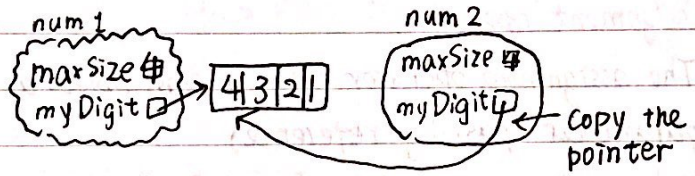
Date

→ .cpp file //dtor

```
BigNum::~BigNum(){
    delete [] myDigit; //Dynamic array ★
    myDigit = nullptr; //GOOD Practice
    //mySize = 0; optional
}
```

Q: What happens when we create a BigNum object and then make a copy of it?
(pass-value in a function: same thing, copy)

eg.
```
BigNum num1("1234");
BigNum num2(num1);
```



By default, the fields of original object are copied into fields of the new object [same is true for pass-by-value parameter]

- PROBLEM: The two objects share the same dynamic array and thus changes made to array by one will seen by the other.
- BIGGER Problem:
  If num2 goes away, it deletes the dynamic array num1 is left by a dangling pointer. ②
- To access this, we must define a "copy constructor" in the class. It is called anytime we need to make a copy of an object. //const-reference, ✗value

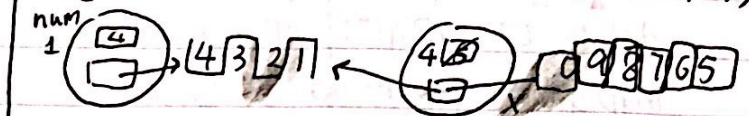
```
BigNum::BigNum(const BigNum &rhs):
    maxSize(rhs.maxSize),
    myDigit(new int[maxSize]){
    for(size_t i=0; i<maxSize; i++){
        myDigit[i] = rhs.myDigit[i]; //copy contents
    }
}
```

//right hand side - rhs
//must pass-by-reference.: we are defining a copy constructor

★ Any BigNum method could be accessed by any BigNum obj. (in BigNum::) (private) data

Q: What happens when we do?

```
BigNum num1("1234"); num2("56789");
```




```
num2 = num1;
```

- By default, the compiler assigns each field of the rhs obj to corresponding field of lhs obj.
- Problem: we end up with a shared dynamic array
  Other problem: memory leak since we lost the pointer to the original array.
- To fix this, we must define our own

(3) Assignment operator,

- The assignment operator takes a class object as a parameter (pass-by-reference)
- and we then assign it to the ~~original~~ object on which the method is called
- Assignment operators are supposed to return a value s.t. they can be chained.
  (e.g. a=b=c=d;)

past: void

To do this, we will return a const reference to the 'this' object

- Hearder

```
const BigNum& BigNum::operator=(const BigNum& rhs)
```

(warning)

const BigNum&: return type
BigNum::: member of BigNum class
operator=: assignment operator
(const BigNum& rhs): param

- Basically we need to
  ① free up the dynamic array of the ~~left~~ lhs object
  ② copy rhs object
  → seems like we only need to copy-n-paste dtor code and copy ctor code
- But more detail

num1 = num1 ?

---

int v1 = 10;
- int& type = v1; // assigned once
reference type (alias for v1)
cout << type // 10

* only when a=b (in the operator method), the reference (and the objects) won't be changed

No.

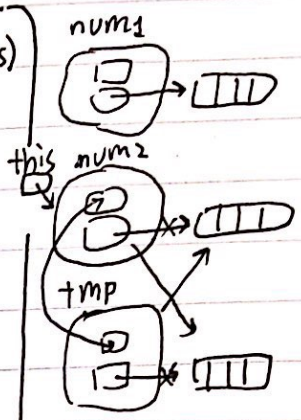
Date

→ If we deleted array on lhs object, there is no rhs object to copy

★ So, all assignment ops always start by a check for self assignment.

→ could copy-n-paste dtor & copy ctor code BUT a better way

```
const BigNum & BigNum :: operator= (const BigNum & rhs)
{
    if(this==&rhs){  //note: address comparison, not object comparison
        return *this;
    }
    BigNum tmp(rhs);
    std:: swap(mgx size, tmp.maxSize);
    std:swap(myDigit, tmp.myDigit);  //swap all the field
    return *this;
} //tmp goes way, destor calls on it
```

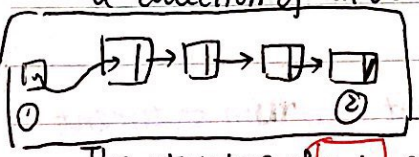


Read

Page. 133~148
150~154
272~281

Linked List

- allow us to grow and shrink as needed
- a collection of structures / nodes linked together



The structure of node consist of two parts:

- a data to be stored
- a pointer to next node

- Two other issues:
  1. Need to know where the list starts
  2. — use a head pointer (not a node only a pointer to 1st node)
  3. Need to know where list ends
     - use a nullptr in last node

(In a ~~struct~~ class: all variable by defaut are ~~private~~ public)

- How do we define a node?
  - → it has multiple parts so we can use a class or struct
  - → by default, everything in a class is private, whereas it is public in a struct

```
struct Node
{
    Itemtype item;
    Node* next;
};
```

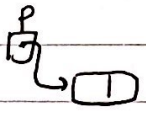
Each node needs to hold 2 pieces of information
1) use typedef
2) a pointer to the next node

Given this, we can declare pointers to nodes as following

```
typedef Node * NodePtr;
NodePtr p;
```

Now we can dynamically create node w/ "new"

```
p = new Node;
```



we can then later free that node with statement

```
delete p;
```

In between, we can access the fields of that node using arrow notation.

`p→item` to access item field

// (same (*p).item) BAD style

`p→next` to access next field

Given a linked list, how would we display its contents?

→ must traverse/walk the list and visit each node

No.

Date

How do we traverse a list? → use a temp pointer

How do know where we start? → use special head pointer

How do we get next node? → use the next field on node

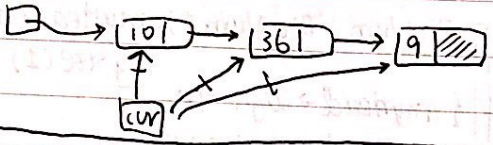
How do know when to stop? → when we get to nullptr

last pointer?

//code

```
for (NodePtr cur = headPtr; cur != nullptr; cur = cur->next) {
    std::cout << cur->item << std::endl;
}
```

need to change



Let's develop BigNum to use a linked list rather than array

→ still performs dynamic memory allocation

thus we need to do big three.

BigNum.h

```
struct Node {
    int digit;
    Node* next;
};
typedef Node* NodePtr;
class BigNum
{   private:
        NodePtr myHead;
        size_t mySize; // # of digits (optional)
    public:
        BigNum(); // default constructor
        BigNum(const std::string& mNum); // alt ctor
        BigNum(const BigNum &rhs) // copy ctor
        ~BigNum(); // dtor
        const BigNum & operator=(const DigitNum &rhs);
```

```
//other methods
std::string toString() const;
void add(const BigNum& val);
    ⋮
};
```

How do we want to store digits in the linked list?

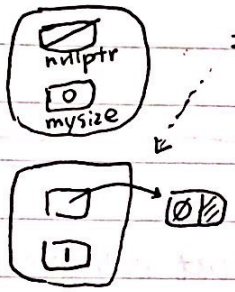
→ store ones digit in 1st node

→ tens digit in 2nd node

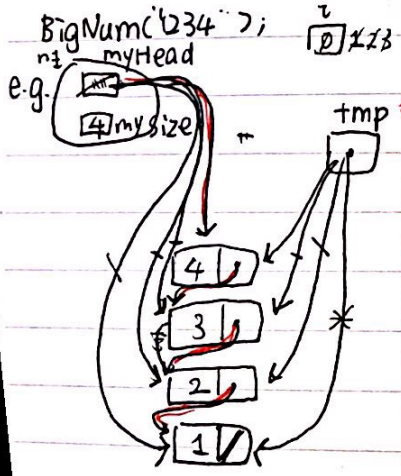
→

Consider default ctr. → represent # 0 (HOW? ① empty list ② one node "0" bigNum.cpp)

1)
```
BigNum::BigNum(): myHead(nullptr), mySize(0)
{
// nothing to do
}
```



2)
```
BigNum::BigNum(): myHead(new Node)
                  mySize(1)
{ //myHead->digit=0;
  myHead->next=nullptr;
}
```


```
// Alt ctor that takes a string
BigNum::BigNum(const std::string & num):
        myHead(nullptr), mySize(num.length()){
    //create a node for each digit in num
    NodePtr tmp;
                          //easier than each time call the function length()
    for(size_t i=0;i<mySize;i++){
        temp=new Node;
        temp->digit=num[i]-'0'; //give us a number
        tmp->next=myHead;
    }   myHead=temp;
}
```

Edge Condition

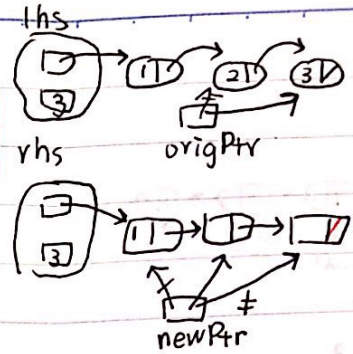
Empty String ✓

No.

Date

copy ctor

```
BigNum::BigNum(const BigNum &rhs):
        myHead(nullptr), mySize(rhs.mySize){
    if(rhs.myHead != nullptr){
        //copy a node
        myHead = new Node;
        myHead->digit = rhs.myHead->digit;
        //copy rest of that
        NodePtr newPtr = myHead;
        for(NodePtr origPtr = rhs.myHead->next;
            origPtr != nullptr;
            origPtr = origPtr->next){
            newPtr->next = new Node;
            newPtr = newPtr->next;
            newPtr.digit = origPtr->digit;
        } //end for-loop
        NewPtr->next = nullptr;
    }
}
```



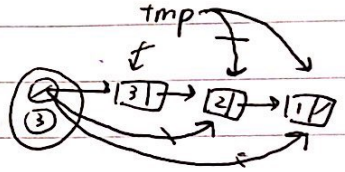
Edge Condition:

1) empty list ✓

2) list of only 1 node ✓

Dtor

```
BigNum::~BigNum(){
    NodePtr tmp;
    while(myHead != nullptr){
        tmp = myHead;
        myHead = myHead->next;
        delete tmp;
    }
    mySize = 0; //optional
}
```



Assignment Operator

```
const BigNum & BigNum::B1operator=(const BigNum & rhs)
    if(this != &rhs){    //*self-check
        BigNum tmp(rhs);
        std::swap(myHead, tmp.myHead);
        std::swap(mySize, tmp.mySize);
    }
    return *this;
}
```

Programming of linked list

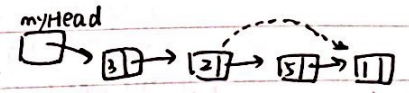
- often need to process of the data in a list - do a traversal

e.g. BigNum中:: toString()

```
std::string BigNum::toString() const {
    if (myHead == nullptr){
        return "0";
    std::string tmpStr("");
    for (NodePtr cur = myHead;
         cur != nullptr;
         cur = cur->next){
        tmpStr = std::to_string(cur->digit) + tmpStr;
    }                                   //prepend
    return tmpStr;
}
```

tmpStr
123

Delete nodes from a list



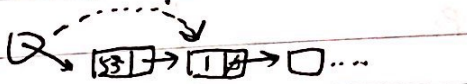
want to delete the '5' node

need the previous node to link around node to be deleted

→ need a cur pointer to point to node to be deleted

→ need a prev pointer to point to predecessor

But what if node to be deleted is 1st node in list?



Edge Condition

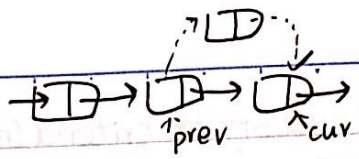
1) first node ✓
2) last node ✓
3) 1 node ✓

*update prev cur at the same time */

```
//given pointer cur to node to be deleted
if(myHead == cur){
    myHead = cur->next;
}else{
    NodePtr prev;
    for (prev = myHead; prev->next != cur; prev = prev->next){
        *assert(pre->next != nullptr); //optional
    }
    prev->next = cur->next;
}
* delete cur;
cur = nullptr; mysize--;
```

No.

Date



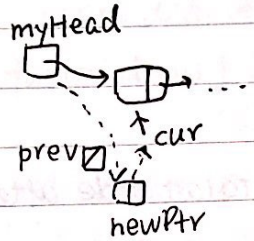
Inserting a node

Let's assume we know the position, and it's between prev and cur

- need prev point to new node
- need the new node point to cur

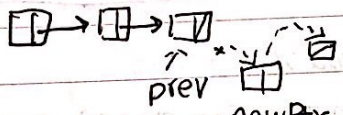
- Edge Condition

① Inserting at head of the list



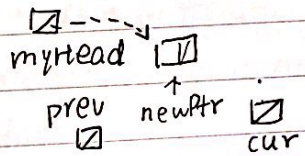
→need to update myHead

② Inserting at the end of the list



→note: newPtr→next still points to where cur is pointing

③ Inserting into an empty list



- since prev is nullptr, update myHead

How do we set prev & cur?

Initially we start our traversal

prev=nullptr

cur=myHead;

Then we walk list to insertion point

→ some criteria to determine point of insertion or walk off the end

```
while(cur!=nullptr && criteria(newPtr, cur)){
    prev=cur;
    cur=cur->next;
}
```

for a sorted list, the criteria:

newPtr->item > cur->item

Note: order of the test is important (first test if cur is nullptr, or cur->item would throw exception)

Does this work for

① empty list ✓

② inserting at front of list? ✓

③ inserting at the end? (prev point to last node after the traversal) ✓ ie.

Read Page424

→ can all be written in a for loop

NewPtr created some place else

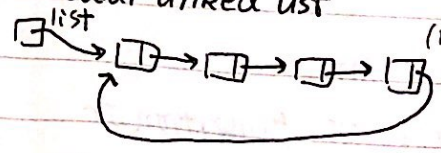
```
NodePtr prev, cur;
for(prev=nullptr, cur=myHead;
    cur!=nullptr && criteria_test(newPtr, cur); prev=cur
    cur=cur->next)
{ } //nothing to do
//now link node into the list
if(prev==nullptr){
    myHead=newPtr;
}else{
    prev->next=newPtr;
}
newPtr->next=cur;
mySize++;
```

nctio

er

Variations of linked list

① circular linked list



(not null-pointer terminated)
(no headptr)

very useful when your data has cycle

(or listptr change)

process: ->next != list (traversal)

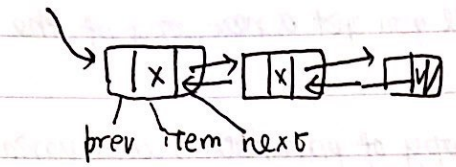
No.

Date

② Doubly linked list

Each node has a ptr to next node & previous node

myHead



Read P47~89

Recursion

→ Stand ard defn of factorial?

$$\text{factorial}(n) \begin{cases} (n)\times(n-1)\times(n-2)\times\cdots\times 2\times 1, & \text{if } n>0 \\ 1 & \text{if } n=0 \end{cases}$$

Lets write it up

```
size_t fact(size_t n){
    size_t product (1);
    for(size_t i=1; i<=n; i++){
        product *= i;
    }
    return product;
}
```

This solution uses a loop

→ call it an iterative solution

A mathemetician define factorial as:

$$\text{fact}(n) \begin{cases} 1 & \text{if } n=0 \\ n\times \text{fact}(n-1) & \text{if } n>0 \end{cases}$$

why do they do this?

→ easy to prove correct by an induction proof

→ requires a base case

recursive solution

```
size_t fact(size_t n){
    if(n==0){
        return 1;
    }else{
        return n*fact(n-1)
    }
}
```

→ solution depends upon calling itself on a smaller problem

→ requires definition of base case

On each recursive call you get a new copy of the function to execute

→ get a new copy of parameters, local variables

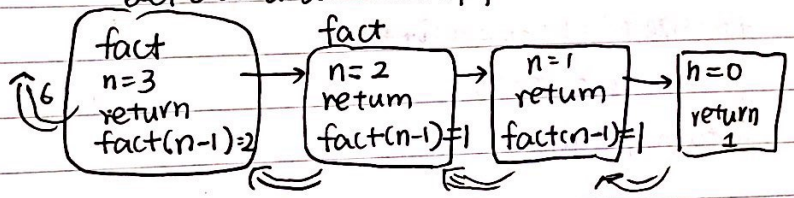
→ whenever you change a variable, you only the local copy.

→ One way to keep track at things is w/ a box trace

- Each recursive call creates a new box that contains the local environment

(parameters, local vars, return value, ...)

Let's do a box trace of fact(3)



△Think Recursively

If someone else could "magically" solve the smaller identical problem for you, how would you use that solution to solve your larger problem

let's write a function to print positive integers backward.

```
// first, let's do it iteratively
void printBackwards(size_t i) //pre: i>0
{
    while (i>0) {
        std::cout << i%10;
        i=i/10;
    }
}
```

Recursive:

If someone can print an n-1 digit backwords for us... how can we use them to print a n-digit # backwards

2 options:

① print Backwards (i minus 1st digit)
write 1st digit

② write last digit
printBackwards (i minus last digit)

removing the last is easy, use that option

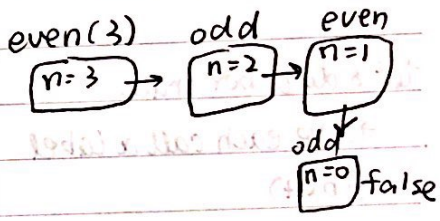
```
void printBackward (size_t i)
//pre: i >0
{ //base case: 1 digit left or no digit left
    if (i==0) { //base case of no-digits
      // do nothing
    }else{
      std::cout<< i%10;
      printBackwards (i/10);
    } }
```

→ Test a non-negative number for even or odd
→ do w/o mod division
→ only use subtraction

```
bool even(size_t n)
{ if(n==0) //base case{
    return true;
  }else {
    return odd(n-1);
  }
}
```

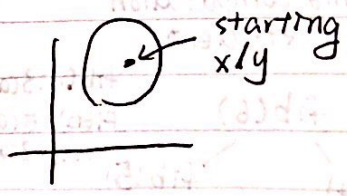
```
bool odd(size_t n)
{ if(n==0) //base case{
    return false;
  }else{
    return even(n-1);
  }
}
```

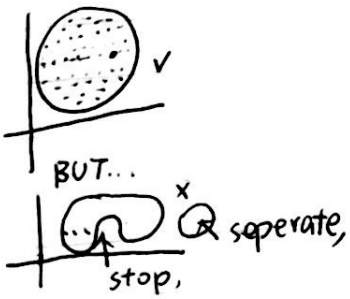
even(3)

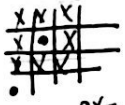


Mutual recursion.

Consider this problem:
A doctor is looking at an X-ray, and sees a cancer blob (group of black pixels) → Dr wants to know size of blob → Dr clicks on blob to provide a starting (x,y) coord, it is our job to compute size of blob (size == pixels)







keep spreading out until we off the grid/ blob

can you solve this iteratively?

~~Yes~~.

A recursive solution is easy

```
computeSize(size_t x, size_ty){
  if (x & y are off grid){
      return zero;
  }else if {pixel x/y is white}{
      return zero;
  }else{
      recolor-pixel at x,y to be white
      return 1+(computeSize(1→ for each
                of eight neighboring pixels)
```

compute $pow(x, n) = x^n$

where n is non-negative

- could use iterative loop

for loop that does n multiplication

$\Rightarrow O(n)$

Can we do better?

consider this define that uses $x^n = (x^2)^{n/2}$

$$pow(x,n): \begin{cases} 1, & \text{if } n=0 \\ pow(x*x, n/2), & \text{if } n \text{ is even} \\ x * pow(x, n-1) & \text{if } n \text{ is odd} \end{cases}$$

How many multiplication are performed?

$\Rightarrow O(\lg n)$ ($2 \lg n$)

exponential solution $O(2^n)$

fibonacci sequence

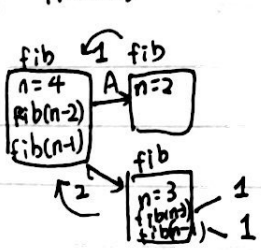
want to write a function to return the n^{th} fib #

```
int fib(int n)   //task: compute nth fib#
{                //pre: n is positive
  assert(n>0);
```

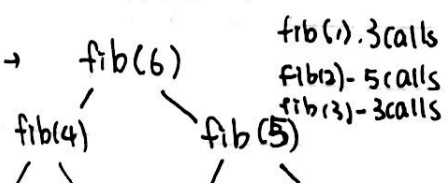
-let's do a box trace

→give each call a label

fib(4)



Is this computation efficient? →



Consider an iterate solution:

→need to keep of 2 values on each iteration

→Here's one way

```
int fib(int n)
{
  assert(n>0)
  int cur=1
  int prev=1;
  while(n>2){
    cur=cur+prev;
    prev=cur-prev; //old cur
    n--;
  }
return cur;
```

O(n) solution

The Towers of Hanoi

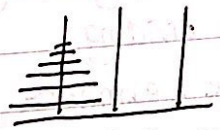
Given n disks of different sizes and 3 poles on which to stack disks

2 rules:

1) only move 1 disk at a time.

2) only place smaller disks on top of larger disk

Goal: move a pole of n disks from a source pole to destination pole using 3rd pole as a space.



src dest space

We're supposed to move n disks

→if someone else can magically move n-1 disk for us, How would we use them?

→move top n-1 disks to space pole

→move n^{th} disk to destination

→move n-1 disks to destination

No.

Date

```
void Towers(int n, char src, char dest, char spare)
//pre: n is positive
{ if (n==1) { //base case
    //move the top disk from src to dist
    std::cout <<"move top disk" << src << "to pole" << dest
    << std::endl;
  }else{
    Towers(n-1, src, spare, dest);
    Towers( 1 , src, dest, spare);
    Towers(n-1, spare, dest, scr);
  } //exponential solution
```

(1 disk more, twice...) 10 disk 1023

By and conquer

Recursion is well suited for solving problems

by a divide-conquer solution

⇒ divide-an-conquer has 3 steps

1) Divide Problem into smaller identical subproblems
2) Conquer the subproblems via recursion
   → request a base case
3) combine solution of subproblems into final solution

→ Problem: find largest element in a nonempty array

~~3 steps~~ Given an array, can solve this by dividing array in half, then compare those 2 elements

maxArray (array)

find largest in left half: maxArray (leftHalf)

find largest in right half: maxArray (right half)