CS32 Discussion
Week 2

Muhao Chen
muhaochen@ucla.edu
http://yellowstone.cs.ucla.edu/~muhao/
About me and CS32

Muha Chen
• Ph.D. Candidate, working with Prof. Carlo Zaniolo and Prof. Kai-Wei Chang
• Research interest: natural language processing, machine learning, and spatiotemporal reasoning
• Teaching Fellow
• Homepage: http://yellowstone.cs.ucla.edu/~muhao/ (slides will be posted here)

CS32
• Mostly about data structures, and some object-oriented design.
• Basic knowledge of computational complexity analysis.
My Recent Research

• Representation learning of knowledge graphs
  • How do we obtain quantified representations for commonsense knowledge and incorporate them into machine learning?

• Semi-supervised learning
  • Matching concepts across multiple human languages with limitedly known cross-lingual alignment

• Machine reading comprehension
  • Predict what happens next in short stories

• I am working with some undergraduate researchers
  • Pei Zhou (15 Math Comp.): leading author of an ICWSM’18 paper on computational modeling of linguistic variation
  • Haiying Huang (16EE), Ryan Liu (16 Math Comp.), Xian Kai Ng (16CS): Machine reading comprehension
Outline

• Dynamic memory allocation revisit
• Copy Constructor
• Assignment Overloading
• Linked Lists
Dynamic Memory Allocation
Static memory allocation

• If we want to type in a paragraph and save it into a C-string.
  • #define MAXLENGTH 10000
  • char s[MAXLENGTH+1]; cin.getline(s);

• What if the paragraph is extremely long?
  • out-of-bound

• What if the paragraph has only five words?
  • Over-allocated memory
Dynamic allocation

• What if we want to fit the paragraph into a C-string with right the sufficient size?

• Dynamic allocation of an array
  • `<type> *<name> = new <type>[<#elements>];`
  • `char *article = new char[length];`

```cpp
int length;
cout << how many characters are at most in your article? << endl;
cin >> length;
char *article;
if (length >0)
    article = new char[length + 1];
```

Unsigned int variable
new

- **new** will dynamically allocate the sequential memory space for the requested data type and size.
- **new** will always return the starting address of the allocated memory space.
- `int array = new int[size];` ✗
- `int *array = new int[size];` ✓

- Anything allocated with **new** will remain in the memory unless we manually **delete** it.
What if we want to dynamically allocate a 2-D array

```c
int rows = 5; int cols = 20;

int **array = new int*[rows];
for (int i=0; i<rows; ++i)
    array[i] = new int[cols];

//array is now array[5][20]
```
Memory Leak

```c
int *p;
p = new int[2000000];
p = new int[1000000];
```

- We allocate 2M blocks of int and point `p` to it.
- Then we allocate another 1M and point `p` to it. `p` no longer points to the first 1M blocks.
- The first 2M blocks of int become the “memory ghost”. We no longer can access and release it.
- This situation is called **Memory Leak**.
Delete

Dynamic allocation has to be deleted once we no longer use it.

delete[] p;

• Delete the dynamic array pointed by $p$. 
Copy Constructor
Copy Constructors - Motivation

class School
{
public:
    School(const string &name);
    string getName() const; // accessor
    void setName(const string &name); // modifier
    void addStudent(const Student &student); // modifier
    Student *getStudent(const string &name) const; // accessor
    bool removeStudent(const string &name); // modifier
    int getNumStudents() const; // accessor
private:
    string m_name; // Name of the school.
    Student *m_students; // Dynamic array of students.
    int m_numStudents; // Number of students.
};
Copy Constructors - Motivation

Student st1("Brian");
Student st2("John");
School s1("UCLA");
s1.addStudent(st1);
s1.addStudent(st2);
Student *p = s1.getStudent("John");

We want to create a new School called s2, with exactly the same content as s1. In other words, we want to clone s1.
Copy Constructors - Motivation

• Candidate 1: Use an assignment.

```java
School s2("" );
s2 = s1;
```

– What are the issues with this method?
Copy Constructors - Motivation

- **Candidate 1**

  School s2(""");
  s2 = s1;

  - **Correctness**: Every member variable gets copied – even the pointers (but not the pointees).
  - **Efficiency**: It will first call the default constructor of s2, initialize members with default values, and then copy the values.
The Problem Here: Shallow Copy

• When we copy C-strings:

```
char a[100] = “2016 Corvette Stingray”;
char b[100];
b = a;
a[2] = ‘0’;
cout << b;
```

2006 Corvette Stingray

• Deep copy: grab every character from a to b:

```
for (int i = 0; i <= strlen(a); ++i)
    b[i] = a[i];
```
Copy Constructors - Motivation

- Candidate 2: Just grab values out of s1 and manually copy them into s2.

```java
School s2("");  
s2.setName(s1.getName());
...
```

- What are the limits to this approach?
Copy Constructors - Motivation

• Candidate 2

School s2();
s2.setName(s1.getName());
// how do I get students out of s1?

- We may not have accessors and modifiers to all member variables!
- It is often not desirable to have the user (of a class) know all the internals.
- Too long to write!
Copy Constructors

```cpp
public:
    School(const School &aSchool);

• A constructor to define the behavior of copying from one instance to another.
```
School::School(const School &aSchool) {
    m_name = aSchool.m_name;
    m_numStudents = aSchool.m_numStudents;
    m_students = new Students[m_numStudents];
    for (int i = 0; i < m_numStudents; ++i)
        m_students[i] = aSchool.m_students[i];
}
School Copy Constructor

- With the copy constructor defined, you can now use:

```cpp
School s2(s1);
```

or equivalently,

```cpp
School s2 = s1;
```
But how about assignments?

\[ s2 = s1; \]
Assignment Operator Overloading

\[ s2 = s1; \]

- Overload the operator (in this case, we overload the assignment operator).

public:

    School& operator=(const School &aSchool)
Assignment Operator Overloading

School& School::operator=(const School& aSchool)
{
    m_name = aSchool.m_name;
    m_numStudents = aSchool.m_numStudents;
    m_students = new Student[m_numStudents];
    for (int i = 0; i < m_numStudents; i++)
        m_students[i] = aSchool.m_students[i];

    return *this;  // don't forget this!
}

I assume we have = operator properly defined in Student class.
Assignment Operator Overloading

School& School::operator=(const School &aSchool)
{
    m_name = aSchool.m_name;
    m_numStudents = aSchool.m_numStudents;
    delete[] m_students;
    m_students = new Student[m_numStudents];
    for (int i = 0; i < m_numStudents; i++)
        m_students[i] = aSchool.m_students[i];

    return *this;       // don’t forget this!
}

I assume we have = operator properly defined in Student class.
Assignment Operator Overloading

School& School::operator=(const School &aSchool)
{
  if (this != &aSchool)
  {
    m_name = aSchool.m_name;
    m_numStudents = aSchool.m_numStudents;
    delete[] m_students;
    m_students = new Students[m_numStudents];
    for (int i = 0; i < m_numStudents; i++)
      m_students[i] = aSchool.m_students[i];
  }
  return *this; // don’t forget this!
}

I assume we have = operator properly defined in Student class.
Before we talk about linked lists...

• CS32 is all about organizing data. We call an organization scheme a **data structure**. For every data structure, we must define:
  – rules for organizing data items (e.g., array with integers stored in a nondecreasing order),
  – a method to **add** a new data item without breaking any of the rules,
  – a method to **remove** a data item without breaking any of the rules, and
  – most importantly, how to **search** for an item

• We will examine various data structures and algorithms, pros and cons of each, as well as their efficiency.
Linked Lists

• A key component of a linked list is a node, which is a single unit of data.

• The first box carries a value, and the second is a pointer to another node.

• Here is an example node definition in the form of a C++ struct:

```cpp
typedef int ItemType;
struct Node
{
    ItemType value;
    Node *next;
};
```
Linked Lists

• Linked list is a series of nodes, each pointing to the next one.

• The last node’s next pointer is NULL.

• What is the information you need to complete the picture?
Head Pointer!

- Obviously, you need to know where it begins.

- We keep a pointer that points to the first item and call it the **head pointer**.

- e.g.
  ```
  Node *head;
  ```
Linked Lists (Min. Requirements)

- You need a description of a node, which must contain a next pointer.
- You need a head pointer that points to the first node.
- The list must be loop-free (unless it is a circularly linked list, in which case one (and only one) loop must exist).
Linked Lists (Insertion)

- Adding a new value to the list.
Linked Lists (Insertion)

1. Create a new node. Call the pointer to it $p$. 
Linked Lists (Insertion)

2. Make its *next* pointer point to the first item.
   
   \[ p \rightarrow next = \text{head}; \]

\[ \text{p} \rightarrow \text{new} \rightarrow \text{head} \rightarrow \text{other nodes} \]
3. Make the head pointer point to the new node.
   head = p;
Linked Lists (Insertion)

• What about insertion in the middle of the list?
• At the end of the list?

End (q): q->next = p; p -> next = NULL;
Middle (after q): p -> next = q -> next; q -> next = p;
Node* Search(int key, Node* head){
    Node *q = head;
    while(q != NULL) {
        if(q -> value != key) q = q -> next; //move to next
        else return q;
    }
    return NULL;
}
Linked Lists (Removal)

- Suppose there is an item that you want to remove, and it is pointed by a pointer, say p.
- Can I just do “delete p;”?  

```
  q <---> p
```

- We need to set the previous node’s (q) next pointer to point to the next node of p!
Linked Lists (Removal)

- When looking up $p$, keep the pointer to the previous node ($q$).

- Then...
  
  ```
  q->next = p->next;
  delete p;
  ```
Linked Lists (Removal)

- **Sanity Checks**
  - Does it work if `p == head`?
  - Does it work if `p` points to the last one?

```c
q->next = p->next;
delete p;
```
Linked Lists (Removal)

• If \( p == \text{head} \), there is no “previous” node to \( p \).
• Make an exception for this.
  – We need to reset the head pointer.

\[
\text{head} = p->\text{next};
\text{delete } p;
\]
void remove(int valToremove, Node* head) {
    Node *p = head, *q = NULL;
    while (p != NULL) {
        if (p->value == valToRemove)
            break;
        q = p; //q keeps track the previous of p
        p = p->next;
    }
    if (p == NULL)
        return;
    if (p == head) //node to remove is head
        head = p->next;
    else
        q->next = p->next;
    delete p;
}
What’s nice about linked lists

• Very efficient insertion
• Flexible memory allocation
  – Think about what you should do if you have to grow/shrink a dynamically allocated array.
  – And yes, there is a little overhead, but that’s the price we pay.
• Simple to implement
What’s not so nice about linked lists

- Slow search (i.e. accessing a certain element, e.g. “get the 4237th item”)
  - Usually, search is the operation that matters more than insertion or removal.
Variations

• Sorted Linked Lists
  – Make changes to the insertion method.

• Doubly Linked Lists
  – Each node has *prev* and *next* pointers.
  – A *tail* pointer is kept to point to the last node.
  – Why do you think this is useful?

• Circularly Linked Lists
  – The last node’s next pointer points to the first one.
  – Essentially, there is no “first” node.
A Classic Problem about Circularly Linked List

• Given a head node of a linked list, how to verify whether there is a loop in the linked list or not?
Bugs in your software are actually special features :)