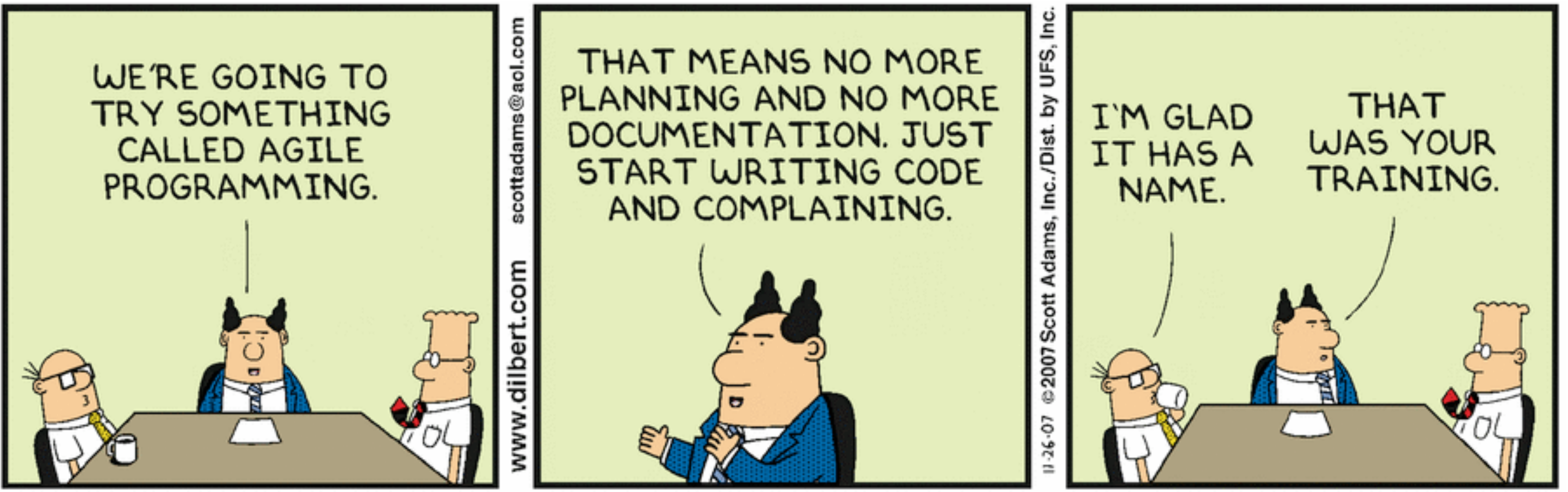
VE280 Programming and Elementary Data Structures

Paul Weng
UM-SJTU Joint Institute

Final Review



Final Exam

- August 4th, 2020, 10:00 am – 11:40 pm
- Via Zoom
- Open book and open notes
- No communication allowed
- Read carefully the instructions and the questions

Final Exam

- Written exam
  - Like previous years
  - A number of questions which only require you to provide a very short answer
  - A few questions which require you to write code

- Abide by the **Honor Code**!

Final Exam Topics

Range: Everything after Midterm

- Subtype and inheritance
- Virtual function
- Abstract base class (interface)
- Representation invariants
- Dynamic memory management and dynamic array
- Constructor taking default arguments and destructor
- Deep copy: copy constructor, overloaded assignment operator
- Linked list
- Template
- Container of pointers: one invariant and three rules
- Operator overloading
- Stack and queue
- STL

Subtypes

Creating

- Subtype: satisfying "substitution principle"

- In an Abstract Data Type, there are three ways to create a subtype from a supertype:

1. Add one or more operations. E.g. IntSet -> MaxIntSet
2. **Strengthen** the **postcondition** of one or more operations. E.g., MaxIntSet -> SafeMaxIntSet
3. **Weaken** the **precondition** of one or more operations. E.g., MaxIntSet -> SafeMaxIntSet

Inheritance

- C++ has a mechanism to enable subtyping, called **inheritance**.

```
class bar : public foo {
    ...
};
```

- `bar` is a **derived class** of `foo`
- Legal to have

```
bar b;
foo &fr = b;
foo *fp = &b;
```

- **Protected** data members
  - Versus private data members

Virtual Functions

```
class IntSet {
    ...
public:
    ...
    virtual void insert(int v);
    ...
};
```

fn. method( )

- This makes it possible to run the function based on the actual type.
- "virtualness" is inherited.

Virtual Functions

```
class foo {
public:
    void f();
    virtual void g();
};
class bar: public foo {
public:
    void f();
    void g();
};
```

non-virtual

virtual

```
bar b;
foo *fp = &b;
fp->f(); //Call foo::f()
fp->g(); //Call bar::g()
```

Abstract Base Classes

- An "interface-only" class, from which an implementation can be **derived**.
- Cannot be instantiated, because there is no implementation.
- Define **pure virtual functions** for abstract base classes.
  ```
  virtual void insert(int v) = 0;
  ```
- Put the implementation in a **derived class**.
  ```
  class IntSetImpl : public IntSet
  ```
- Create instance using pointers/references.
  ```
  IntSet *getIntSet();
  ```

Representation Invariants

- A **representation invariant** applies to the data members of ADT.
- It describes the conditions that must hold on those members for the representation to correctly implement the abstraction.
- Essentially, for each method, you should:
  - Do the work of the method (i.e. insert)
  - Repair the invariants you broke
- Invariants can be coded, to check the sanity of the structure.
  - To check: `assert(repOK());`

Dynamic Memory Allocation

- Dynamic objects, about which the compiler doesn't know
  - **How big it is.**
  - **How long it lives.**
- Dynamic storage management: **`new`** and **`delete`**
- Memory leak problem ✓
- Checking memory leak: **`valgrind`**
- Dynamic Arrays

  ← Computed

```
int *ia = new int[5];
delete[] ia;
```

  - <u>Note</u>: difference between **`delete`** and **`delete []`**

IntSet with Dynamic Array

- Overloaded Constructor
  - **`IntSet();`**
  - **`IntSet(int size);`**

- Calling constructor

default

```
IntSet is1;
IntSet is2(200);
```

IntSet with Dynamic Array

- Function with Default Argument

```
IntSet(int size = MAXELTS);

int f(int a, int b = 3, int c = 4);
f(2, 5);  a = 2, b = 5, c = 4
```

- There could be multiple default arguments in a function, but they must be the last arguments.

```
int add(int a, int b = 0, int c = 1) // OK
int add(in a, int b = 1, int c) // Error
```

- Destructor
  - `~IntSet();`
  - Automatically called

Deep Copy

- Shallow Copy versus Deep Copy
  - We need to copy the dynamic array, not just the array pointer.
- Copy Constructor

```
IntSet(const IntSet &is);
```

- Assignment Operator

```
IntSet &operator=(const IntSet &is);
```

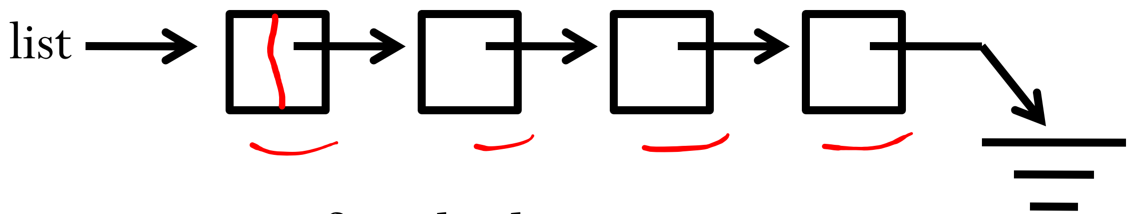
  - Assignment returns a **reference** to the left-hand-side object.
  - Can handle self-assignment correctly by first checking

    ```
    if(this != &is)
    ```

    x = x;
  - `return *this;`
- The Rule of the Big Three
  - destructor, copy constructor, and assignment operator

Linked List

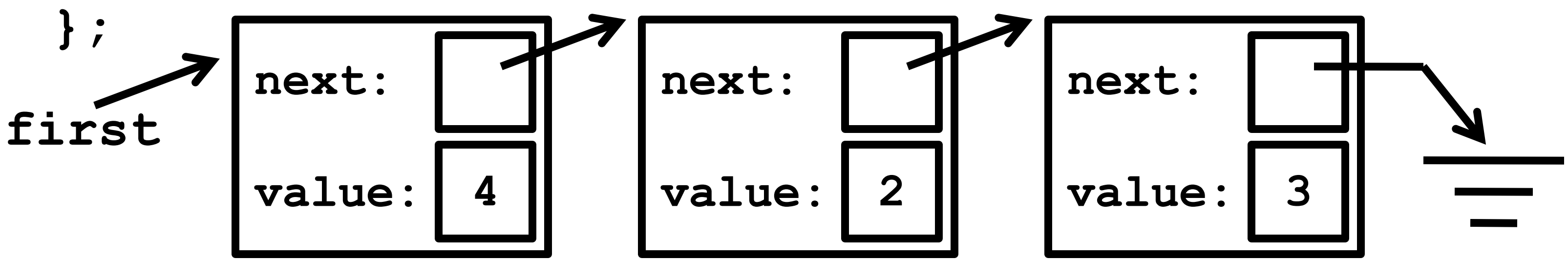
- A linked list is one with a series of zero or more data containers, connected by pointers from one to another, like:



- Implementation of Linked List

```
class IntList {
  node *first;
  public:
  ...
};
```

```
struct node {
  node *next;
  int   value;
};
```



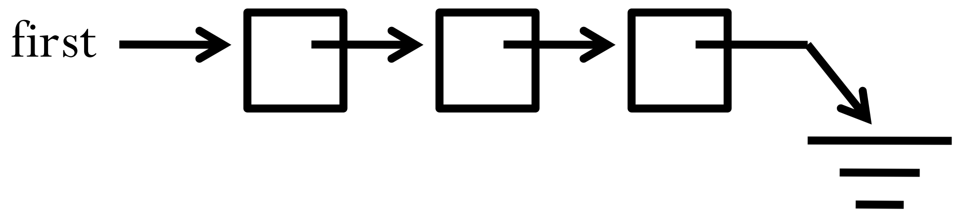
Linked Lists

```
class IntList {
  node *first;    dynamic object
public:
  bool isEmpty();
  void insert(int v);
  int remove();
  IntList();                      // default ctor
  IntList(const IntList& l);      // copy ctor
  ~IntList();                     // dtor
  // assignment
  IntList &operator=(const IntList &l);
};
```

- Variations of linked lists.

Linked List Traversal

- With the “first” pointer, we can traverse the linked list.




```
int IntList::getSize() {
  int count = 0;
  node *current = first;
  while(current){
    count++;  // Some operation
    current = current->next;
  }
  return count;
}
```

for(node * current = first;
current ; current = current
->next)
Count ++

Traverse
through the list.

Polymorphism and Templates

- Things like `IntSet` and `IntList` are often called **containers** or **container classes**.
  - We can also define `CharList`.
- Reusing code for different types is called **polymorphism**.

type

```
template <class T>
class List {
   ...
};
```

Templates

- Each **method** must also be declared as a "**templatized**" method.

```
template <class T>
void List<T>::isEmpty() {
  return (first == NULL);
}
```

- To use templates, you specify the type T when creating the container object.

```
List<int> li;
```

Container of Pointers

- Instead of copying large types by value, we usually insert and remove them **by reference**.

```
void  insert(BigThing *v);
BigThing *remove();
```

- At-most-once invariant.
- Existence, ownership, and conservation rules.

Containers

Destructor

```
template <class T>
List<T>::~List() {
   while (!isEmpty()) {
      T *op = remove();
      delete op;
   }
}
```

Container of Pointers

Copy

```
template <class T>
void List<T>::copyList(node *list) {
  if (!list) return;
  copyList(list->next);
  T *o = new T(*list->value);
  insert(o);
}
```

Operator Overloading

- C++ lets us **redefine** the meaning of the operators when applied to objects of **class type**.

- Most overloaded operators may be defined as ordinary **nonmember** functions or as class **member** functions.

```
A operator+(const A &l, const A &r);
// returns l "+" r
A A::operator+(const A &r);
// returns *this "+" r
```

Friend

- A mechanism to make a function/class as a "**friend**" of another class, so the function/class can directly visit the private members of the other class

```
class foo {
   friend class bar;
   friend void baz();
   int f;
};
class bar { ... };
void baz() { ... }
```

Friendship of both class and function.

Stack ADT

- A "pile" of objects where new object is put on **top** of the pile and the top object is removed first. LIFO
- Five operations
  - size(), isEmpty(), push(), pop(), top()

- Can be implemented using either array or linked list

- Applications
  - Web browser's "back" feature
  - Parentheses matching

Queue ADT

- A "line" of items in which the **first** item inserted is the **first** one out.
  - Insert to the back and remove from the front

FIFO

- Six operations
  - size(), isEmpty(), enqueue(), dequeue(), front(), rear()
- Can be implemented using either linked list or array
  - What kind of linked list? double-ended singly-linked list
  - What kind of array? circular array
- Application: wire routing in electronic design automation

Standard Template Library

- Sequential container: store and retrieve elements by position
  - vector, deque, list
- Associative container: store and retrieve elements based on their **keys**. We focus on two associative containers where the order depends on the keys of the elements:
  - map, set
- Iterators: companion type of a container
  - Iterators are more general than subscripts: All of the library containers define iterator types, but only a few of them support subscripting.
  - Operations: ++iter, --iter, iter1 == iter2, iter1 != iter2, *iter
  - const_iterator: cannot change the element referred to

Sequential Container: vector

- Contructor
  - `vector<T> v1; vector<T> v2(v1);`
  - `vector<T> v3(n, t);`
  - `vector<T> v4(b, e);`
    - Iterator range. Can even use another container type / built-in array to initialize
- Random access through subscripting: `d[k]`
- size(), empty(), push_back(), pop_back(), front(), back(), begin(), end(), clear()
- Supports iterator arithmetic (iter+3) and relational operations on iterator (iter1 </<=/>/>= iter2)

Differences between vector, deque, list

- deque and list support push_front() and pop_front(); vector does not support

- **list** does not support subscripting

```
list<string> li(10, "hi");
li[1] = "hello"; // Error!
```

- No iterator arithmetic for **list**

```
list<int>::iterator it;
it+3; // Error!
```

- No relational operation <, <=, >, >= on iterator of **list**

```
list<int>::iterator it1, it2;
it1 < it2; // Error!
```

Which Sequential Container to Use?

- `vector` and `deque` are fast for random access, but are not efficient for inserting/removing at the middle
- `list` is efficient for inserting/removing at the middle, but not efficient for random access
- Choose based on the required operations and their frequencies
  - Use `vector`, unless you have a good reason to prefer another container.

Associative Container: map

- It stores (key, value) pair
- `map<string,int> word_count;`
- We can use subscripting to add elements to a map

  `word_count["Anna"] = 1;`
  - If key exists, subscripting return the value
  - If not existing, adds an element with that index to the map
- How can we determine if a key is present without causing it to be inserted?
  - `m.find(k)`
- Iterator for map elements
  - `iter->first; iter->second;`

Good Luck to Everyone!

Questions?

Sample Question 1

- Consider the following class Foo:

```
class Foo{
      IntSet set;
public:
      Foo(){
            set = IntSet();
      }
      // other members are omitted
};
```

initialization syntax

- How many times is the constructor of IntSet called when the default constructor of Foo is called?

2

Sample Question 2

- For the following code, there is **at most one** major problem. If there is one, write down the problem. You don't need to tell us how to fix it. If there is none, write "None".

```
vector<string> ds;
ds.push_back("toto");
ds.push_back("tata");
stack<string> s(ds);
s.pop();
```

deque<string>

stack by default is based on deque.

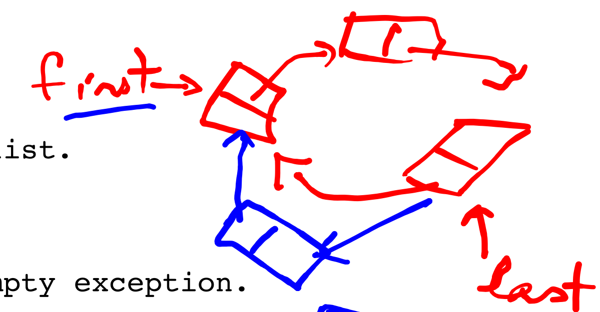
It can't be constructed from vector

Sample Question 3

- A circular list can be defined as follows:

```
template<class T> class CircularList {  // OVERVIEW: a circular singly-linked list
    struct Node{
        Node *next; // next node, NULL if empty
        T val; // value contained by this node
    };
    node *last; // pointer to the last node of the list
public:  /* Other member functions are omitted */
    void insert(T v);
    // MODIFIES: this
    // EFFECTS: Add the value v to the front of the list.
    T remove();
    // MODIFIES: this
    // EFFECTS: If the list is empty, throws a ListEmpty exception.
    //              Otherwise, remove the first node of the list and
    //              return its value
    unsigned int size() const;
    // EFFECTS: Return the number of elements in the list.
};
```

```
class ListEmpty {
     // OVERVIEW: an exception class
};
```



- Why do we keep a pointer to the last node (instead of the first)?
- Give the representation invariant
- Implement the insert and remove methods