Ve 280

Programming and Introductory Data Structures

Final Review

Reminder

- Make-up lecture this Friday, 12:10 pm – 1:50 pm
  - Same classroom
  - On associative container

Final Exam

- 4:00 pm – 5:40 pm, August 11^{th}, 2016
- Find your seat on Sakai
- Closed book and closed notes
- No electronic devices are allowed
  - These include laptops and cell phones
  - We will show a clock on the screen

Final Exam

- Written exam
  - Like midterm
  - A number of questions which only require you to provide a very short answer
  - A few questions which require you to write code on the paper

- Abide by the **Honor Code**!

Final Exam Topics

Lectures 14~26 (including up to sequential container; **no** container adapter and associative container)

- Subtype and inheritance
- Virtual function
- Abstract base class (interface)
- 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
- Synthesized maintenance methods and derived class maintenance Methods
- STL: sequential container

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);
    ...
};
```

- 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();
```

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

```
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

```
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);
```

$\boxed{\texttt{a = 2, b = 5, c = 4}}$

- 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

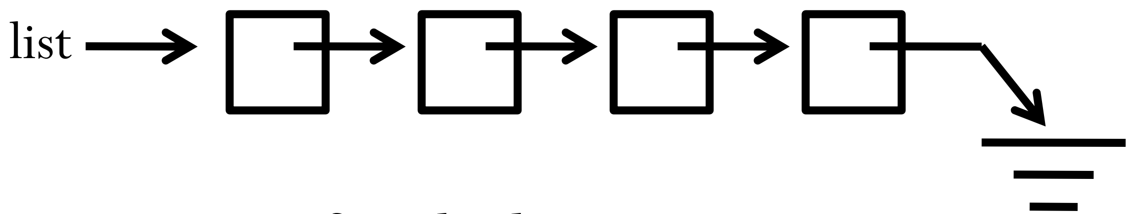
  `IntSet(const IntSet &is);`
- Assignment Operator

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

Linked List

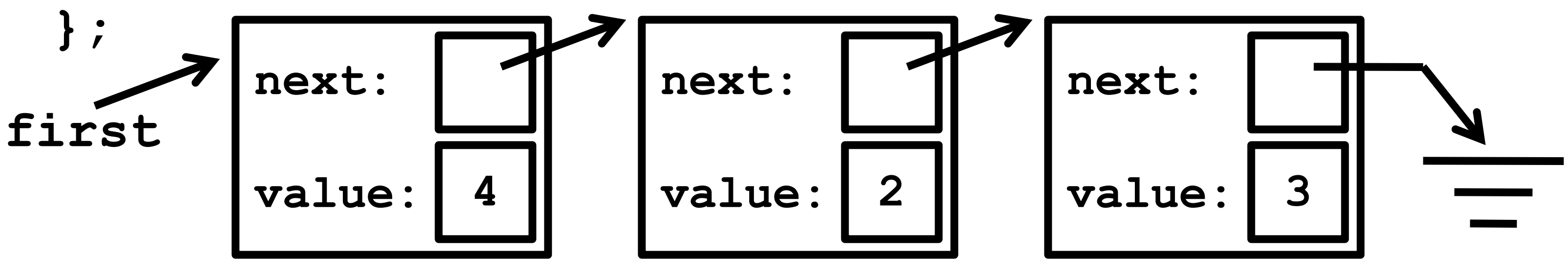
- A linked list is one with a series of zero or more data containers, connected by pointers from one to another, like:



- Implementation of Linked List

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

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



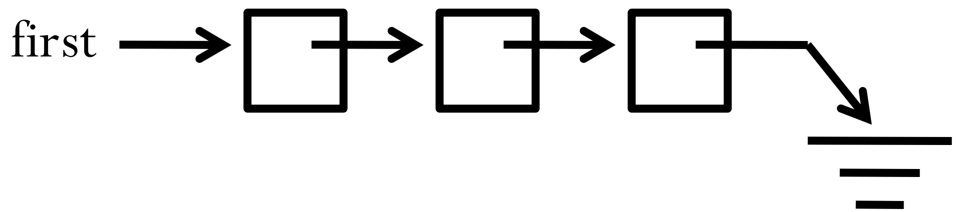
Linked Lists

```
class IntList {
  node *first;
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++;
    current = current->next;
  }
  return count;
}
```

Traverse through the list.

Operations Should Be in Right Order

```
int IntList::remove() {
  node *victim = first;
  int result;
  if (isEmpty()) {
    listIsEmpty e;
    throw e;
  }
  first = victim->next;
  result = victim->value;
  delete victim;
  return result;
}
```

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**.

```
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
```

Special Operators

- Subscript operator [].
- Two versions:

```
const int &IntSet::operator[](int i) const {
    if(i >= 0 && i < numElts) return elts[i];
    else throw BoundsError();
}
```

const version returning a const reference to int

```
int &IntSet::operator[](int i) {
    if(i >= 0 && i < numElts) return elts[i];
    else throw BoundsError();
}
```

nonconst version returning a reference to int

Special Operators

- Output operator <<

```
ostream &operator<<(ostream &os, const foo &o){
  ...
  return os;
}
```

- Input operator >>

```
istream &operator>>(istream &is, foo &o){
  ...
  return is;
}
```

Friend

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

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

Friendship of both class and function.

Stack

- A "pile" of objects where new object is put on **top** of the pile and the top object is removed first.
- Five operations
  - size(), isEmpty(), push(), pop(), top()
- Can be implemented using either array or linked list
- Application
  - Web browser's "back" feature
  - Parentheses matching

Queue

- A “line” of items in which the **first** item inserted is the **first** one out.
- Six operations
  - size(), isEmpty(), enqueue(), dequeue(), front(), rear()
- Can be implemented using either linked list or array
  - What kind of linked list?
  - What kind of array?
- Application: wire routing in electronic design automation

Synthesized Maintenance Method

- If we do not explicitly define any constructors, what will happen?
  - Answer: The compiler will automatically synthesize a default constructor, i.e., a constructor that takes no argument
  - Members of built-in types (int, double, etc.) are uninitialized
  - Members of class types are initialized using their default constructor
- If a class defines at least one constructor, then the compiler will not generate the default constructor
- There is always a synthesized destructor even if you explicitly write one

Derived Maintenance Method

- The way to define constructor that can initialize base part

```
class Base {
  int i;
public:
  Base(int _i = 0):
    i(_i) { }
};
```

```
class Derived: public Base {
  double d;
public:
  Derived(int _i =0, double _d = 0):
    Base(_i), d(_d) { }
};
```

Define Derived-Class Copy Constructor

- If we define the copy constructor, it usually should explicitly use the base-class copy constructor to initialize the base part of the object

```
class Base {
  int i;
public:
  Base(const Base &b): i(b.i) {}
};
```

```
class Der::public Base {
  double d;
public:
  Der(const Der &dr): Base(dr), d(dr.d) {}
};
```

- Similar for assignment operator: call the base-class assignment operator

Virtual Destructors

```
class Base {
public:
   virtual ~Base() { ... }
};
```

- Why need?
  - When deleting a pointer that points to a **dynamically allocated** object, it may actually point to a derived class object
  - We need to ensure the proper destructor is run
- Good practice:
  - The root class of an inheritance hierarchy should define a virtual destructor even if the destructor has no work to do

Standard Template Library

- Sequential container: store and retrieve elements by position
  - vector, deque, list
- 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 / bulit-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)

Difference 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.

Good Luck to Everyone!

Questions?