C++ 2

- Klondike card game, example of OO design
- More issues regarding inheritance
  - Abstract classes
  - Multiple inheritance - strengths and weaknesses
- Templates - generics
- Iterators
Klondike

• Klondike solitaire card game:
  – “Set up 7 tableaus/piles; cards dealt from deck onto discard pile; topmost card from the discard pile can be placed on suit pile or tableau piles, according to rules.” (demo a game in class)

• Required, a graphical interface to Klondike:
  – to display piles, and when a card (at the top of a pile) is clicked on, move it automatically according to the rules of the game.
Klondike

• Classes/kinds of objects: nouns, noun-phrases
• Methods/what one needs to do with them: verbs
  – Card class: `getSuit`, `getNumber`, `flip`, `isFaceUp`?
  – Pile class: `display`, `isEmpty?`, `whatsOnTop`, `canTakeOnTop?`, `popCard`, `addCard`, `clickOn`

• Kinds of Pile: Deck, Discard, Suit, Tableau
  – need $7 \times 4 = 28$ methods ???
Klondike

Superclass AbsPile will

- provide signatures for all ops,
- **definitive** implementations for some (e.g., isEmpty?() )
- **default** implementations for others (e.g., display() : print topmost card only)

**Pile : clickOn [just the signature]**

- isEmpty, whatsOnTop, -- **definitive** implementation
- popCard (**default: pop stack**)
- canTakeOnTop? (**default: false**)
- addCard (**default: push**)
- display (**default: show topmost, if face up**),
Klondike

• Superclass (abstract) **AbsPile** will
  – provide signatures for all ops,
  – definitive implementations for some (e.g., ```isEmpty?()```)
  – default implementations for others (e.g., ```display()``` : print topmost card only)

• Superclass **Pile** :
  – ```clickOn``` [just the signature]
  – ```isEmpty, whatsOnTop``` -- definitive implementation
  – ```popCard``` (default: pop stack)
  – ```canTakeOnTop?``` (default: false)
  – ```addCard``` (default: push)
  – ```display``` (default: show topmost, if face up),
Klondike

• Redefine in subclasses:
  • **Suit**: clickOn (beeps), canTakeOnTop
  • **Deck**: clickOn
  • **Discard**: clickOn, addCard (to also flip it)
  • **Tableau**: display, clickOn, canTakeOnTop, popCard (turn over the card underneath)
Parametric Polymorphism in C++

- **Same algorithm, except for type of value.**

- **Classes**
  - **Declaration**

    ```cpp
template <class C> class Stack {
    C elts[20];
    int top;
    public:
    void push(C);
    C peek();
    ...}

    C Stack::peek(){return elts[top];}
    ```

  - **Use**

    ```cpp
    my_stack = new Stack<int>;  // type passed as arg: C=int
    my_stack.push(3);
    ```
Abstract Classes

#include <stdio.h>
#include <stream.h> //example inspired by pohl book

class A {
public: virtual void print_i() =0;
};

class D : public A {
public: void print_i() { cout << 2 << " inside D\n";}
};
class C : public A {
public:
    void print_i() { cout << 3 << " inside C\n";}
};

Cannot create A objects because A is an abstract class; note missing Implementation for A::print_i()
Abstract Classes

main()
{
    C *pc = new C();  D *pd = new D(); A *pA;
    pc -> print_i();  //should print 3 inside C
    pd -> print_i();  //should print 2 inside D
    pA = pd;
    pA -> print_i();  //should print 2 inside D
    pA = pc;
    pA -> print_i();  //should print 3 inside C
}

//40 scherzo!programs> ./a.out
//3 inside C
//2 inside D
//2 inside D
//3 inside C
Inheritance in C++

- Use abstract class to create consistent interfaces for subclasses
  - Promises an implementation for every non-abstract subclass (although this implementation can be inherited)
- Have **subtype polymorphism** if never redefine inherited functions
- Code sharing and reuse
- Automatic propagation of changes to subclasses
- C++ has no equivalent to Java **final** which prevents subclass extension
- Benefits to class designer and users
Inheritance

• As subtyping
  – Inheriting implementation and external specification
  – S is subtype of T if all operations on type T objects are meaningful on S objects; substitutability in behavior

• As code reuse
  – Inheriting only implementation; not necessarily an is-a relation
  – Building new components from old
Assume we have a `Deque` class with `InsertAtRear()`, `removeFrFront()`, `insertAtFront()`, `removeFrRear()`. 

And we want to define `Queue` as a subclass of `Deque` 

```
  Q: private DQ;
```

Private inheritance allows use of `DQ` protected functions within the defn of class `Q`, but does not allow users of `Q` to apply `DQ` functions to `Q` objects.

Contrast with: given a `Queue` class, extend it to a `Deque` subclass by adding `insertAtFront()`, `removeFrRear()`, 

```
  DQ : public Q
```
Example

• Two ways to define queue and dequeue

<table>
<thead>
<tr>
<th>Queue</th>
<th>Dequeue</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertAtRear()</td>
<td>InsertAtFront()</td>
<td>exclude:</td>
</tr>
<tr>
<td>RemoveFrFront()</td>
<td>InsertAtRear()</td>
<td>RemoveFrFront()</td>
</tr>
<tr>
<td></td>
<td>RemoveFrRear()</td>
<td></td>
</tr>
</tbody>
</table>

subtyping - similar behavior

as code reuse
Inheritance

• Multiple versus single
  – Real world is multiple
  – Linearizing lookup
    • Problem: interpretation depends on non-local inheritance structure, not robust in face of changes
  – No problem if no conflicts
Multiple Inheritance

- Needed to describe certain complex *is-a* relationships

Endangered species

- Carnivore
  - Cat
  - Leopard

- Herbivore
  - Bear
  - Cow
  - Panda
  - Polar

Animal
Multiple Inheritance Conflict Resolution

• Problems:
  – Member clash
  – Inheriting more than one copy of same member

• Approaches
  – Linearize hierarchy so only one parent is “closest” (CLOS, Flavors)
  – Throw an exception when same member is applied more than once due to duplicate paths
  – Exclude some members to avoid problem (C++ virtual base classes)
Example - Mixins

virtual class window

bordered_window

draw() { window::_draw(); _draw(); }

window_w_menu

draw() { window::_draw(); bordered_window::_draw(); window_w_menu::_draw(); _draw(); }

draw() calls _draw() and does work for base case

_draw() does what’s specific to its class.
Iterators

- Q: How does one go through a collection of values (e.g., set, list, queue,...) if one may not need to have a look at all of them, and one needs to remain ignorant how the collection is stored?
  - **Example task**: there is a pile of cards, and player is looking through it top-down for a card with a certain property; the pile is not affected by this.

- Need to consider how this is a part of the PL design
  - CLU supported this action as a built-in operation on collections
Soln 1: Use linked list

• **Designer**: defines a type and specification of Stack ADT provides function which returns pointer to linked list of elements in the stack;

typedef class Cell{int data, Cell *link}
  *CellType;
CellType getStkelements(Stack)

• **User**: walks down the linked list in her own code.

```cpp
Stack sx; ...
Cell* vals = getStkElements(sx);
while ( vals ){
    ... //process vals->data here
    vals = vals->link;
}
```
Soln 1: Use linked list

• **Implementer:**
  – if Stack was implemented as linked list, could just return it -- but this breaks encapsulation :-(
  – if Stack is implemented as an array:
    • time and space needed to copy from array to a linked list (it promised a list!)
    • much of it could be wasted since user may stop after finding some value part way through the list
Soln 2: pump enumerator for ADT

- **Designer:** promises new Stack ADT functions
  
  ```
  void startPumpStk(Stack s)
  bool hasMore(Stack s)
  eltType getNext(Stack s)
  ```

- **User:**

  ```
  EltType v;
  Stack sx;
  startPumpStk(sx);
  ... //use sx
  while ( hasMore(sx) ){
    v = getNext(sx);
    ... //process v here
  }
  ```
Soln 2: pump enumerator

- **Implementer:**
  - Adds a field `current` to Stack structure (an `int` if array implementation or a `Cell*` if linked list implementation)
  - **Problem:** can only have one iterator through the stack at one time (i.e., no nested iteration)
Soln 3: Use Separate ADT

**Designer:** creates new ADT type

- **StackIterator**
- **StackIterator**(Stack s) //constructor
- bool hasMore(StackIterator si)
- EltType **getNext**(StackIterator si)

**User:**

```cpp
EltType v_outer, v_inner;
Stack sx;
... //push stuff on sx
StackIterator *it1, *it2;
for (it1 = new StackIterator(sx);
    it1->hasMore();
    v_outer = it1->getNext();)
{ for (it2 = new StackIterator(sx);
    it2->hasMore();
    v_inner = it2->getNext();)
    { ... //process v_inner and v_outer here }
}
```
Soln 3: Use Separate ADT

typedef struct {current, Stack s} * StackIterator;

StackIterator createStackIter(Stack)
allocates space for iterator and initializes current
(e.g. sets it to top)

bool hasMore(StackIterator)
checks if there is more left in the collection
(e.g. checks if current >= 0)

EltType getNext(StackIterator)
returns the current value and advances the cursor
(e.g., return s[current--];)

Shown as C code
C++ Stack Iterator Specification

```cpp
class Stack_iter {
    // used to iterate over a stack from top down

public:
    Stack_iter(Stack &goOver);
    ~Stack_iter();
    bool hasMore();
    EltType getNext();
}
```

class Stack {
    private:
        eltType s[MAX];
        int top;
        static int Empty = -1;
    friend Stack_iter;
}

class Stack_iter{
    Stack& sk; //points to Stack
    int current;
    Stack_iter(Stack &goOver)
        { sk = goOver; current = sk.top; }
    bool Stack_iter::hasMore() {
        return (current!=sk.EMPTY);
    }
    EltType Stack_iter::getNext() {
        if( current!=sk.EMPTY )
            return sk.s[current--];
        else error("...");
    }
    Stack_iter::~Stack_iter(){
}
Using the Iterator

```cpp
int main() {
    stack y;
    stack z(5);
    y.push(2);
    y.push(3);
    y.push(5);
    z.push(10);
    z.push(11);
    z.push(12);
    Stack_iter ity1(z); // constructor invoked implicitly
    stack_iter ity2(y);
    stack_iter ity4(y);

    while (ity4.hasMore()) {
        cout << ity4.getNext() << " \n";
    }
    z.push(13);
    while (ity1.hasMore()) {
        cout << ity1.getNext() << " \n";
    }
}
```

// sample run:> a.out
// 5 3 2
// 12 11 10
// why don't we see 13 in the output of iterator ity1 on z?
Using the iterator

while (ity2.hasMore()) {
    // nested iterators: no problem!
    stack_iter ity3(y);
    while (ity3.hasMore())
        {cout << ity3.getNext() << " " << ity2.getNext() << "\n";
        }
}
// sample run:> a.out
// 5 3 2
// 12 11 10
// 5 2 3 2 2 2
// 5 3 3 3 2 3
// 5 5 3 5 2 5
C Iterator to use with C++

- C Interface has one fewer function, but that one has an extra pass-by-result parameter

```
bool getMore(StackIterator si, EltType & e)
// returns true, if there are more, in which case it sets e to next one
```

- C++ usage of C iterator saves a function call

```
EltType v;
Stack sx;
createStk(sx);
... //use sx
StackIterator *it1 = new StackIterator(sx);
while ( it1.getMore(it1, &v) ){
    ... //process v
}
```
Use C++ STL

• Think of a “cursor” running along the collection:
  – `start()` , `atEnd()` -- positions of the cursor
  – `advance()` -- move cursor forward
  – `getCurrent()` -- get value at current cursor
• Use a notation that resembles for loops:
  ```
  for( now = <start>; now!=<finish>; now++)
  { cout<< *now->...  
    /* access value at now */  }
  ```
• Done through overloading
class Stack {
    private: ...//implemented as an array;
    public:
        StackIterator begin();
        StackIterator end();
    class StackIter{ public:
        void operator ++();
        EltType operator *();
        bool operator ==(StackIter other);
    }
}
Another Approach

class stack { 
    private: elt *s; int top; friend class stack_iter;
    const int EMPTY = -1;
public:
    stack(){s = new elt[100]; top = -1;} ...
}

class stack_iter {//will enumerate stack from bottom to top of stack
    private: elt *st; int n; int t;
    //invariant: elements in st[0..n] have already been returned
    stack_iter(stack &goOver){ // creates copy of stack
        t = goOver.top;
        st = new elt[t+1];
        for (int j=0; j<=t; ++j)
            st[ j]=goOver.s[j];
        n = goOver.EMPTY;}//initializes subscript pointing into copy
    boolean getNext(elt &val){
        if (n < t) {val = st[++n]; return 1;} else return 0;
    }
}
Summary

• Can’t define iterator as subclass of the collection class
  – Because then each iterator could only work with respect to one collection object
• Can’t define iterator as member of the collection class
  – Because member functions have no way to preserve state between calls (class vars are not enough since they are shared by all objects)
Summary

• There is NO natural subtyping relation between iterators and the collections they iterate over!

• Solution - break encapsulation to create an iterator
  – Use *friend* methods which lets iterator see into the private collection instance variables