An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 13

Arrays (ctd.)

Queues and Stacks

Based on material developed by Prof. Abhiram G. Ranade

Recap

- (C-style) Arrays
  - A few examples: Counters, storing text elements for use later (palindrome), sorting
  - Restrictions of C-style arrays: Need to fix the size at compile time, cannot pass by value or return from functions, somewhat unusual syntax for the type declaration

Today

- Using arrays as *Data Structures*
  - To keep data in ways convenient for various tasks/algorithms
  - Queues and Stacks
    - Today: Implementing queues and stacks. Using them in examples.

Queue

- When people form a queue:
  - People join and leave the queue
  - You have to join the queue at the "back"
  - You leave the queue only after you reach the "front"
- Items leave the queue in the order in which they join
  - First In First Out (FIFO)
- Why are queues useful in programs?
  - Storing items (e.g. key presses, mouse clicks, network packets) until they are processed one-by-one
  - Systematic searches (e.g., for a short path in a maze) often use a queue

Example: Taxi Stand

- Taxis arrive and stand in a queue
- A customer, on arriving, gets into the taxi at the front of the queue
- If a customer arrives when the taxi stand is empty, they leave
- If a taxi arrives when the taxi stand is full, it leaves

- A program that tracks the status of the taxi stand:
  - Accept inputs of the form "taxi" (includes, say, the vehicle number) and of the form "customer" (includes, say, the destination)
  - Print out (taxi,customer) pairings, or error messages (customer not accommodated, or taxi not accommodated)

Example: Taxi Stand

```
struct taxi { string id; void read(); };
struct customer { string dest; void read(); };
struct taxiQueue {
  /* members to be filled in */
  void clear(); bool enqueue (const taxi&); bool dequeue (taxi&);
};
void assign(const taxi&, const customer&);
```

```
taxiQueue tq; tq.clear();

for(...) {
  ...
  if (input == 'T') {
    taxi t; t.read();
    bool ok = tq.enqueue(t); // returns true if succeeded
    if (!ok) cerr << "Could not add taxi " << t.id << endl;
  } else if (input == 'C') {
    taxi t; customer c; c.read();
    bool ok = tq.dequeue(t); // returns true if succeeded
    if(ok) assign(t,c);
    else cerr << "Could not find a taxi for " << c.dest << endl;
  }
}
```

Example: Taxi Stand

```
const int qCapacity = 6;
struct taxiQueue {
     taxi Q[qCapacity]; int begin, end; // members
     void clear(); bool enqueue(const taxi&); bool dequeue(taxi&);
};
void taxiQueue::clear () { begin = end = 0; }
bool taxiQueue::enqueue (const taxi& t) {

     /* fill in at Q[end], and increment end */

}
bool taxiQueue::dequeue (taxi& t) {

     /* remove from Q[begin], and increment begin */

}
```


```
enqueue
enqueue
enqueue
        dequeue
enqueue
        dequeue
        dequeue
        dequeue
```

Our queue's capacity is reducing with each dequeue!

Move data so that begin is always 0? (That is what real taxis will do!)

Moving data can be slow…

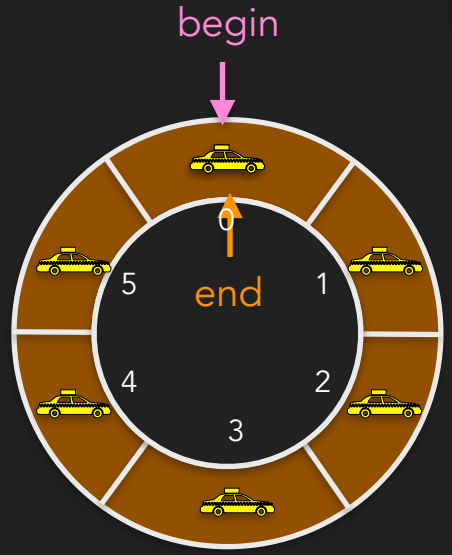
Invariant:
Queue elements stored in Q[begin],…,Q[end-1].
Q[begin] is the first occupied slot, Q[end] is the first empty slot.
Unless queue is empty: Then begin == end.

Example: Taxi Stand

```
const int qCapacity = 6;
struct taxiQueue {
     taxi Q[qCapacity]; int begin, end; bool full; // members
     void clear(); bool enqueue(const taxi&); bool dequeue(taxi&);
};
void taxiQueue::clear () { begin = end = 0; full = false;}
bool taxiQueue::enqueue (const taxi& t) {
     if(full) return false;
     Q[end] = t; // copy to the end of the queue
     end =  (end+1) % qCapacity; // advance end clockwise
     if (end == begin) full = true;
     return true;
}
bool taxiQueue::dequeue (taxi& t) {
     if (end == begin && !full) return false;
     t = Q[begin]; // copy the front element into t
     begin = (begin+1) % qCapacity;  // advance begin clockwise
     full = false; // never full after dequeuing
     return true;
}
```

begin==end can mean empty or full

Demo



Stack

- Recall what a stack is:
  - Objects (say books) are kept one on top of the other
  - An item can be added only to the top of the stack
  - Can remove only the top-most item (the most recently added)
  - Last-In First-Out (LIFO)
- Why are stacks useful in programs?
  - Already saw: Memory allocated for function calls naturally use a stack (last function invoked is the first to return)
  - Systematic searches often use a stack too
  - To support "undo", a stack can used to store actions so far
  - …

A Simple Stack Implementation

```
const int stCap = 1000;  // capacity
struct charStack {
    char St[stCap];
    int top; // top = -1 for empty stack
    bool pop(char& i) { // if stack is empty, return false
        if (top == -1) return false;
        i = St[top--];
        return true;
    }
    bool push(char i) { // if stack is full, return false
        if (top == stCap-1) return false;
        St[++top] = i;
        return true;
    }
    void clear() { top = -1; }
};
```

Example: Palindrome using a Stack

- Needed the array only as a stack

```
int main() {
  charStack st; st.clear();
  char x, y;
  int n; cin >> n;
  for (int i=0; i<n/2; i++)
    if(cin >> x, !st.push(x)) {cerr << "Stack overflow!\n"; return -1; }
  if (n%2) cin >> x; // if n odd, ignore the middle character
  for (int i=n/2-1; i>=0; i--) { // compare stored characters with rest
    if (cin >> x, st.pop(y), x != y) {
      cout << "Not a palindrome!" << endl;  return 1;
    }
  }
  cout << "Palindrome!" << endl;
}
```

Example: Undo Stack

```
bool docommand(char c, charStack& history); // pushes command into history
bool undo(charStack& history);
int main() {
  turtleSim();
  cout << "Enter f/r/l for forward/right/left, u to undo, q to quit: ";
  charStack history; history.clear();
  char input;
  for(cin >> input; !cin.fail(); cin >> input) {
    if(input == 'q')
      break;
    else if (input == 'u')
      if(!undo(history)) // pop history stack and reverse the popped comand
        cerr << "Undo stack empty!\n";
    else if (!docommand(input,history))
      cerr << "Ignoring unrecognized command" << endl;
  }
}
```

Example: Undo Stack

Demo

```
bool docommand(char c, charStack& history) {
  if (c == 'f' || c == 'r' || c == 'l') {
    history.push(c) || (history.clear(), history.push(c));
    if (c == 'f') forward(10);
    else if (c == 'r') right(90);
    else /* (c == 'l') */ left(90);
    return true;
  }
  return false;
}
```

If push failed, clear history and push again.

```
bool undo(charStack& history) {
  char c;
  if(!history.pop(c)) {
    return false;
  } else {
    if (c == 'f') forward(-10);
    else if (c == 'r') right(-90);
    else /* (c == 'l') */ left(-90);
    return true;
  }
}
```

Exercises

- Use two stacks to allow undo and redo.

  Note that the redo stack gets cleared whenever any valid command other than undo/redo is executed

- Implement a stack structure which forgets the oldest item when an item is pushed into a full stack. Use it as the undo stack.

  Hint: Try a "circular stack"