Stacks: ADT and implementations

- stacks: store a collection of data items
- can add new data items to top/front of collection
- can retrieve content of top item
- can remove top item from the collection
- can query how many items are in the collection
- referred to as LIFO (last in, first out)
- can use implementation similar to linked lists (but always inserting/removing from front/top)
- array based implementations also suitable if we limit size of stack

Stacks: concept

- idea is we pile new data values on top of old ones
- most recent values wind up on top of stack, older values get buried deeper and deeper
- operations referred to as:
  - push: adding new value at top of stack
  - pop: removing top value from the stack
  - top: looking at content of top value, not changing stack
  - size: lookup current number of items in stack

Stack: sample interface, use

- stack of ints (ignoring what the private data portions might be)

```
class stack {
   private:
      // skip this for now
   public:
      stack(); // constructor, empty stack
      ~stack(); // destructor
      void push(int d); // insert d at top
      void pop(); // remove/discard top item
      int top(); // lookup value in top item
      int size(); // lookup current # items in stack
};
```

```
int main()
{
   stack s;
   s.push(10); // 10 only item
   s.push(37); // 37 on top, 10 underneath
   s.push(1); // 1 on top, then 37, 10 on bottom
   cout << s.top() << endl;  // displays 1
   // display and remove items from top down
   while (s.size() > 0) {
        int i = s.top();
        cout << i << endl;
        s.pop();
    }
    // would display in order 1, 37, 10
}
```

Alternate interface: error checking

- have functions return true if ok, false if fail (e.g. if try to use pop/top on empty stack, or if no space left to push)

```
class stack {
  private:
    // skip this for now
  public:
    stack(); // constructor, empty stack
    ~stack(); // destructor
    bool push(int d); // insert d at top
    bool pop(); // remove/discard top item
    bool top(int &val); // lookup value in top item
    int size(); // lookup current # items in stack
};
```

Array based implementation

- have fixed sized array, keep track of # items in stack
- pop/top/push can check the #items and size before proceeding

```
class stack {
  private:
    static const int StackSize = 20; // a const to be shared by all our stacks
    int content[StackSize]; // the actual stack storage
    int current; // number of items currently in stack, init to 0 in constructor
                 // the top item in the stack will always be in content[current-1]
  public:
    stack(); // constructor, empty stack
    ~stack(); // destructor
    bool push(int d); // insert d at top
    bool pop(); // remove/discard top item
    bool top(int &val); // lookup value in top item
    int size(); // lookup current # items in stack
};
```

Array implemenation continued

```
// just start size at 0
stack::stack()
{
   current = 0;
}

// no cleanup needed
stack::~stack()
{
}

// just returns current size
int stack::size()
{
   return current;
}
```

```
// “remove” top item, update size
bool stack::pop() {
    if (current > 0) {
       current--;
       return true;
   }
   return false; // stack was empty
}


// lookup top item, if any
// (top item always in array position current-1)
bool stack::top(int &val) {
    if (current > 0) {
       val = content[current-1];
       return true;
   }
   return false; // stack was empty
}
```

```
// “insert” new top item at next
// available array position
bool stack::push() {
    if (current < StackSize) {
       content[current] = val;
       current++;
       return true;
   }
   return false; // stack was empty
}
```

Linked list style of implementation

- maintain like a linked list of nodes, inserting (pushing) and removing (popping) from front (top)

```
class stack {
  private:
    struct node {
      int value; // value stored in this node
      node* next; // ptr to node “under” this one
    };
    node* content; // pointer to top node, null when empty
    int current; // number of items currently in stack, init to 0 in constructor
  public:
    stack(); // constructor, empty stack
    ~stack(); // destructor
    bool push(int d); // insert d at top
    bool pop(); // remove/discard top item
    bool top(int &val); // lookup value in top item
    int size(); // lookup current # items in stack
};
```

List-style implementation continued

```
stack::stack()
{
   content = NULL;
   current = 0;
}

// delete each node
stack::~stack()
{
   while (content != NULL) {
        node* tmp = content;
        content = content->next;
        delete tmp;
    }
}

int stack::size() {
   return current;
}
```

```
// lookup top value if stack not empty
bool stack::top(int &val)
{
   if (content != NULL) {
      val = content->value;
      return true;
   }
   return false;
}

// remove top item if stack not empty
bool stack::pop()
{
   if (content != NULL) {
      content = content->next;
      return true;
   }
   return false;
}
```

```
// create new node
// insert at front if creates ok
bool stack::push(int val)
{
   node* n = new node;
   if (n != NULL) {
      n->value = val;
      n->next = content;
      content = n;
      return true;
   }
   return false;
}
```