A TYPICAL CLASS – ARRAY BASED STACK

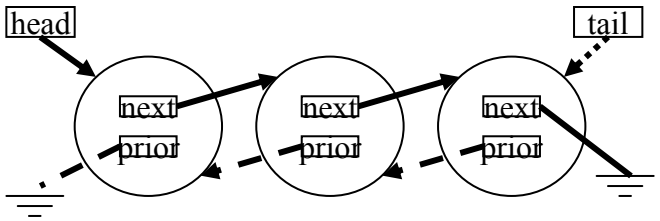
HEADER

```
class IntArrayStack {
private:

  int *data, // pointer to dynamically allocated array
      capacity; // capacity of stack, size of array
  bool elastic; // true if capacity of stack (i.e. the size of
                //  the array) can be changed as required.
  int count; // number of values on stack

public:

  // creates an "elastic" stack
  IntArrayStack ();
  // creates a stack with a fixed capacity
  IntArrayStack (int capacity);
  IntArrayStack (const IntArrayStack &otherStack);
  ~IntArrayStack ();
  // quits if the stack
  //  has a fixed capacity and is full.
  void push (int value);
  // quits if stack empty
  int pop ();
  bool isEmpty() const { return count == 0; }
  int getcount() const { return count; }
  // copies contents only (capacity attributes are not
  //  copied).  quits if the
  // destination stack has inadequate capacity
  IntArrayStack& operator=
                        (const IntArrayStack &otherStack);
};
```

IMPLEMENTATION

```
static const int initialCapacity = 10,
                 capacityIncrement = 5;

IntArrayStack::IntArrayStack () {
   data = new int [initialCapacity];
   capacity = initialCapacity; elastic = true;
   count = 0;
}

IntArrayStack::IntArrayStack (int capacity) {
  if (capacity <= 0) {
     *this=IntArrayStack();
     quit(
       "IntArrayStack::IntArrayStack - invalid capacity\n");
  }
  data = new int [capacity];
  this -> capacity = capacity; elastic = false;
  count = 0;
}

IntArrayStack::IntArrayStack
          (const IntArrayStack &otherStack) {
  data = new int [otherStack.capacity];
  for (int i = 0; i < count; i++) {
    data[i] = otherStack.data[i];
  }
  capacity = otherStack.capacity;
  elastic = otherStack.elastic;
  count = otherStack.count;
}

IntArrayStack::~IntArrayStack () {
  delete [] data;
}

void IntArrayStack::push (int value) {
  int *temp, i;
  if (elastic && (count == capacity)) {
    // time to increase the capacity
    capacity += capacityIncrement;
    temp = new int [capacity];
    for (i = 0; i < count; i++) temp[i] = data[i];
    delete [] data;
    data = temp;
  } else if (count == capacity) {
     quit("IntArrayStack::push - stack overflow\n");
  }
  data[count++] = value;
}

int IntArrayStack::pop () {
  if (count == 0) {
    quit("IntArrayStack::pop - stack underflow\n");
  }
  return data[--count];
}

IntArrayStack& IntArrayStack::operator=
          (const IntArrayStack &otherStack) {
  if (capacity < otherStack.count) {
    if (!elastic) {
      quit("IntArrayStack - copy overflow\n");
    }
    delete [] data;
    data = new int[otherStack.count];
    capacity = otherStack.count;
  }
  for (int i = 0; i < otherStack.count; i++) {
    data[i] = otherStack.data[i];
  }
  count = otherStack.count;
  return *this;
}
```

LINKED LISTS:



Doubly linked lists only ····▶ Only if tail pointer used

```
// outline of typical list-based class
class Typical {
public:

  // public methods go here

private:

  class Node {
    int data; // could be anything
    // prior pointers are optional
    Node *next, *prior;
  }

  // tail pointer is optional
  Node *head, *tail;
  int size;

};
```

```
// typical list traversal
Node *p, *c;
p = null; c = head;

while ( (c != NULL) &&
          (c -> data != data) ) {
  p = c; c = c-> next;
}

if (c == NULL) {
  // not found
}
```

Partial Date Class:

```
class Date {
private:
  long dayNumber;  // 1 = Jan 1, 1900 and so on

  // constructs a date containing the specified day number.
  Date (long dayNumber);

public:
  // This class ensures that dates are always valid and always
  // lie between Jan 1, 1900 and Dec 31, 2099.  Invalid
  // constructor arguments, read errors, etc. result the program quitting.

  // constructs a date containing Jan 1, 1900
  Date ();

  // constructs a date containing the specified date.
  Date (int day, int month, int year);

  // reads a date (in dd mm yyyy format) from the specfied input stream.
  void read (istream &is);

  // writes a date to the specified output stream (in dd/mm/yyyy format)
  void write (ostream &os) const;

  // moves a date the specified number of days forward (positive day
  // values) or backwards (negative day values).
  void move (int days);
};
```

String2002 Class:

```
String2002 a, b;
cout << "Type two words separated by a space: "
cin >> a >> b;
// String Comparisons:
// case sensitive
if (a < b ) {…}
elseif ( a==b ) {…}
elseif (a > b ) {…}
// case insensitive
if (a.isEqualCaseInsensitive("STOP")) == 0) {
   cout << "First word is STOP (ignoring case).\n";
} else {
   cout << "First word is not STOP.\n";
}
```