Lecture 05
More Header Classes
February 19th, 2016

Announcements: Busy Week
• Reading Assignment
  – Numbered Quiz 4b, covers Sections 4.5 - end
  – Note: all quizzes are always due before class
• Recitations: optional
  – Attend any session you want
  – No new material; GTAs will answer questions
  – GTAs will NOT answer questions about midterm code
• Midterm is Wednesday
  – Code handed out Monday
    • Minus main.cpp
  – Exam is in class on Wednesday
• PA4 handed out electronically on Wednesday

Header Class Example: intvector
class intvector {
  public:
    intvector(int sz);
    ~intvector();
    int at(int index) const;
    int& at(int index);
    void push_back(int value);
  protected:
    int* data;
    int size;
};

Implementing push_back
void intvector::push_back(int value) {
  int* temp = data;
  data = new int[size+1];
  for(int i=0; i< size; i++) {
    data[i] = temp[i];
  }
  data[size] = value;
  size++;
  delete [] temp; // discuss
}

Problem: why does this crash?
void Foo(intvector a) {
  ...
}
void Quagga::Bar() {
  intvector iv(0);
  iv.push_back(37);
  Foo(iv);
}

Copy Constructors
• When an object is passed by value, its copy constructor is called.
• Copy constructors take a constant reference to same-type object as their arguments
  Quagga::Quagga(const Quagga& src)
• The default copy constructor copies the fields of the source instance
  • including pointers
• Fortunately, you can redefine the copy constructor
Fixing intvector

class intvector {
public:
    intvector(int sz);
    intvector(const intvector& src);
    ~intvector();
    int at(int index) const;
    int& at(int index);
    void push_back(int value);
protected:
    int* data;
    int size;
};

Implementing the Copy Constructor

intvector::intvector(const intvector& src) {
    size = src.size;
    data = new int[size];
    for(int i=0; i < size; i++) {
        data[i] = src.data[i];
    }
}

• Are these legal? Yes. Size and data are protected, which means only intvectors can access them.
• It would be legal if they were private, too.

One more thing...

• What happens in the following case?

    intvector a(5);
    intvector b(7);

    a = b;

• We need to redefine assignment to
  – Delete old memory
  – Allocate new memory
  – Copy data

Defining Assignment

• For the moment, just trust me on the syntax

    intvector& operator = (const intvector& src) {
        if (size != src.size){
            size = src.size;
            delete [] data;
            data = new int[size];
        }
        for(int i=0; i < size; i++) {
            data[i] = src.data[i];
        }
    }

Header Class Summary

• Header classes manage heap data
• Goal: treat heap data like stack data
• Every header class needs three things:
  – “Deep” constructors that allocates memory via new and copies data
    • Including a copy constructor
  – “Deep” destructor that deletes memory
  – An assignment operator that does both
    • Deletes old memory
    • Allocates new memory and copies data

What about efficiency?

• Intvector acts lack a dynamic variable
• But it copies the whole vector with every push_back.
  – Pushing back 10 items does 45 copies
    • 45 copy constructors
    • 10 user-specified constructors
    • 45 destructors
  – Pushing back N items does \( \sum_{i=1}^{N} (i - 1) \) copies
• Is there a better way?
Efficient intvector

- Allocate arrays that are bigger than needed
  - Separate size (number of elements)
  - From capacity (amount of memory allocated)
- When there is extra capacity, push_back doesn't have to copy or allocate
- When there isn't, push_back doubles the capacity of the array
  - Why double it?

Changes to intvector

class intvector {
public:
  intvector();
  intvector(int sz);
  intvector(const intvector& src);
  ~intvector();
  void reserve(int sz);
  int at(int index) const;
  int& at(int index);
  void push_back(int value);
protected:
    int* data;
    int size;
    int capacity;
};

New implementations

intvector::intvector() : size(0), capacity(10)
{
    // harder if data is not primitive
    data = new int[capacity];
}
intvector::intvector(int sz) : size(sz), capacity(sz)
{
    data = new int[size];
}
intvector::~intvector()
{
    delete [] data;
    data = NULL;
    size = 0;
    capacity = 0;
}

void intvector::reserve(int sz)
{
    if (sz > capacity) {
        int* temp = data;
        data = new int[sz];
        for(int i=0; i < size; i++) {
            data[i] = temp[i];
        }
        delete [] temp;
        capacity = sz;
    }
}

void intvector::push_back(int value)
{
    if (size == capacity) {
        int* temp = data;
        capacity *= 2;
        data = new int[capacity];
        for(int i=0; i < size; i++) {
            data[i] = temp[i];
        }
        delete [] temp;
        data[size] = value;
        size++;
    }
}

Vectors of Objects

- Vectors of objects are a little different
  - Remember the new does two things:
    - Allocate memory (malloc, or something similar)
    - Invoke the constructor
  - Delete also does two things
    - Call the destructor
    - Deallocate memory (free, or something similar)
  - Vectors separate memory (de)allocation from object creation/destruction
  - The assignment in push_back invokes an assignment operator.
  - The vector destructor has to explicitly destruct the components of the vector.
Quaggavector destructor

quaggavector::~quaggavector()
{
    for(int i=0; i < size; i++) {
        data[i].~Quagga();
    }
    delete [] data;
    data = NULL;
    size = 0;
    capacity = 0;
}