C++ Programming

Classes, Constructors, Operator overloading (Continued)

M1 Math
Michail Lampis
michail.lampis@dauphine.fr

Classes

- These (and the previous) slides demonstrate a number of C++ features related to classes:
  - Data encapsulation
  - Operator overloading
  - Constructors/Destructors and memory management
- In this set of slides we continue on the same set of topics with another example: a vector class.

A vector class

- In mathematics, a vector is just an ordered collection of numbers
  - A vector in R^2 is a point on the plane
  - A vector in R^3 is a point in space
  - …
  - We can also consider vectors in R^n
- The closest thing to vectors in C/C++ is arrays

A vector class

- Let's try to design a vector class that works similar to arrays but with some improved features:
  - Bounds-checking
  - Resizing
  - Assignment (v1 = v2)
  - Mathematical operations
  - ...

Prototype

```
#include <iostream>
using namespace std;
class Vector {
        int size;
        double *data;
public:
        Vector(int); //must supply a size
        ~Vector();
        int get_size();
        void set_size(int);
        double read_element(int);
        void write_element(int,double);
        friend ostream& operator<<(ostream &, Vector &);
};
```

About the prototype

- How are we planning to implement Vector?
  - Essentially, with an array (allocated dynamically)
  - For this, we need a pointer to the data
- This choice should be **irrelevant** to a programmer using Vector
  - We could change our minds later
  - In fact, we will!
- Therefore, we need to provide an interface for the data
  - read_element( ), write_element( ), ...

Constructor

- In this class a constructor method becomes very important
  - Our plan is to allocate some memory when a Vector object is created, in order to store the data
  - The constructor should know the size of the Vector being created
  - In this case, we make no default constructor (why?)
  - As we will see, it's generally better to have one...

Constructor

```
Vector::Vector(int size)
{
        this -> size = size;
        this -> data = new double[size];
}
```

Using the constructor

```
//in main program...
int main( ){
    Vector v1(5);
    Vector v2; //This does not compile!!
}
```

- Why not add a default (parameter-less) constructor?
- Something else is missing here!
  - Every new must be followed by a delete somewhere...

Default parameters

- One way to make a default constructor: write another function
- Better way: use default parameters on the constructor we have

  (in vector.h)

  ```
  public:
      Vector(int size=0);
  ```

- In case no parameter supplied: size=0
- A default constructor is **important**! We may need to say

  ```
  new Vector[5]; //create 5 Vectors using the default constructor!
  ```

Destructor

Back to the other thing that was missing...

- Vector::~Vector()

```
{
        /* For debugging
        cout << "Destructor called" << endl;
        cout << *this; */
        delete [ ] data;
}
```

Why do I need a destructor?

- Every time we call the constructor (e.g. by declaring a Vector object) a **new** is executed.
- When the object goes out of scope, its memory is released (good!)
  - But not the memory allocated inside the constructor
  - It is our responsibility to clean this up!!
- To help us, C++ allows us to define a destructor function that will be automatically called.

When is the destructor called?

- When a local object goes out of scope:

```
void f( )
{
    Vector v1(5);
    …
    return; //this will call v1.~Vector() before returning
}
```

- Note: You cannot call ~Vector() directly...

Destructors and pointers

- Consider the following:

  ```
  Vector *vp = new Vector (5);

  …

  delete vp;
  ```

- The last line will call the destructor on *vp and **afterwards** delete vp
- What does the last line do if we don't define a destructor?
  - Destructor is called only if it exists...

Destructors and pointers

- Consider the following:

  ```
  Vector *vp = new Vector [5];

  …

  delete vp;
  ```

- What is the difference with previous slide?
- What is wrong?

Destructors and pointers

- Consider the following:

  ```
  Vector *vp = new Vector [5];
  …
  delete [ ] vp;
  ```

- The first line allocates memory for 5 Vectors
  - The default constructor is called for each one
  - This will not compile without a default constructor!
- The last line calls the destructor for each of the 5 Vectors
  - Destructors are called in reverse order! (vp[4], then vp[3],...)

Summary

- Memory management is a non-trivial task that must be handled in the constructor/destructor
  - Be careful! This can easily be done wrong!
- Usually, constructor allocates needed memory (new) and destructor frees all used memory (delete)
- We are not done! These tasks appear in many other places
  - Copy constructor, operator=, ...

Accessing elements

- Let's write a couple of easy methods:

```
double Vector::read_element(int index)
{
        return data[index];
}

void Vector::write_element(int index, double e)
{
        data[index] = e;
}
```

- Can we add bounds checking to these? (Check if index<size)

Printing a Vector

- Let's overload the << operator

```
ostream& operator<<(ostream & o, Vector &v)
{
        for(int i=0; i<v.size; i++)
                o << v.read_element(i) << ", ";
        o << endl;
}
```

- Recall: this function is a **friend**
- We are passing v as a ref, so the copy constructor is not needed
  - This is a good idea...
- Note: we are using v.read_element(i), not v.data[i]. Why?

A simple program

```
int main()
{
        Vector v1(5);
        v1.write_element(0,1.1);
        v1.write_element(1,2.1);
        cout << v1;
}
```

- Output?

Resizing

- Time to write a non-trivial method for our Vectors (Exercise!)

  void set_size(int newsize)

- This function should **resize** the vector
  - If the new size is larger, we keep all the data
  - Otherwise some data is thrown out
- We need to allocate appropriate new space and free the old data space

Resizing

```
void Vector::set_size(int newsize)
{
        double *newdata = new double[newsize];
        for(int i=0; i<size && i<newsize ; i++)
                newdata[i] = data[i];
        delete [ ] data; //Clean up!!
        data = newdata;
        size = newsize;
        return;
}
```

Functions that use vectors

- Suppose that I want to find a function that calculates the sum of all elements of a vector

  double find_sum(Vector v)

- 1-minute exercise: write this function
  - Note: I don't need to pass the vector size as a parameter (unlike arrays)

Find Sum function

```
double find_sum(Vector v)
{
        int i;
        double sum=0.0;
        for(i=0; i<v.get_size(); i++)
                sum += v.read_element(i);
        return sum;
}
```

- Everyone happy?

A nasty bug

- The previous function contains a serious problem

```
Vector v(3);
v.write_element(0,1);
v.write_element(1,2);
v.write_element(2,3);
cout << find_sum(v) << endl;
cout << find_sum(v) << endl;
```

- The previous program **crashes! Why??**
- Careful: a copy constructor is implicitly called

Another nasty bug

- Consider the following program:

```
int main()
{
        Vector v1(5);
        Vector v2(5);
        v2 = v1;
}
```

- This program also crashes with the same error (**double free**)
- **Why??**

Explain the bugs

- What does v2 = v1 mean?
  - Why does it even compile?
- Reminder: = means copy everything field-by-field
  - The data field is copied
  - This copies the **pointer**, not the data
- v2 and v1 are **pointing** to the same data
  - Which is then freed twice at the end of scope!
- Added problem: memory leak on v2.data
- Same problem with parameter passing...

Overloading =

- Add the following to the class definition (vector.h)

  Vector & operator=(Vector &);

- Explanation: we overload the = operator, so that it does something more sensible
- Parameter is **ref** to Vector (so that we don't have the same problem as in previous slides)

Overloading =

```
Vector & Vector::operator=(Vector &v2)
{
        int i;
        delete [ ] data;
        size = v2.size;
        data = new double[size];
        for(i=0; i<size; i++)
                data[i] = v2.data[i];
}
```

Copy constructors

- One way to get around the problem with using Vector parameters is to use Vector & params.
- Another (less efficient) is to define a copy constructor:
- Add to vector.h

  Vector(Vector &);
- This method is automatically called when a vector is passed as a parameter

Copy constructor

```
Vector::Vector(Vector &v2)
{
        size = v2.size;
        data = new double[size];
        int i;
        for(i=0; i<size; i++)
                data[i] = v2.data[i];
}
```

- Note: **this** is initially an empty object → no delete

Using vectors more naturally

- One problem with read_element, write_element
  - Too much typing!
- How about the following:

```
Vector v1(5);
v1[2] = 3.2;
v1[3] = 4.5;
```

- Unfortunately, this does not compile!
  - But we can fix it! Overload the [ ] operator!

Overloading [ ]

- What is the right prototype?
  - What should we return?
  - v[i] should be a double
  - It should be able to appear on left-hand-side of =
  - → double &
- Add to vector.h

  double &operator[ ](int);

Overloading [ ]

- The implementation is very easy!

```
double & Vector::operator[ ](int index)
{
        return data[index];
}
```

An alternative implementation

- Why should we implement vector with an array?
- Let's do the same exercise again but with a linked list!
  - Why? Practice.
  - Also, this can be more efficient for some things (and less efficient for others)

A new class definition

```
struct Node {
        int index;
        double data;
        struct Node * next;
};

class Vector {
        int size;
        Node *head;
public:
        Vector(int size=0);
        Vector(Vector &);
        ~Vector();
        int get_size();
        void set_size(int);
        double read_element(int);
        void write_element(int,double);
        double &operator[](int);
        Vector & operator=(Vector &);
        friend ostream& operator<<(ostream &, Vector &);
};
```

Outline

- Basic idea: store the data in a linked list
  - Each node of the list contains an index,data pair
  - Meaning: if the user tries to read an element not in the list, we return 0.0
  - → save space for sparse vectors (with many 0s)
- Note: rest of interface is unchanged!
  - User should be able to use this class in the same programs (ideally) without change
- Note2: size field is now not so important...

Constructor

- The constructor now doesn't need to do much
  - … but other methods will be more complicated

```
Vector::Vector(int size)
{
        this -> size = size;
        this -> head = 0; //No memory allocated
}
```

Destructor

- The data is stored in a linked list (pointed to by head)
- When a vector is deleted, we need to delete the list
- Exercise: write a destructor that does this

Destructor:

```
Vector::~Vector()
{
        Node *tmp = head;
        while(tmp){
                head = tmp->next;
                delete(tmp);
                tmp = head;
        }
}
```

Copy constructor

- We want to create a new object with the same data as the old one
- → We must copy the elements of the linked list one-by-one
- To make things easier, we don't care about the order of the list, just that all elements are copied
- Exercise: program the copy constructor

Copy constructor

```
Vector::Vector(Vector &v2)
{
        size = v2.size;
        head = 0;
        Node *tmp = v2.head;
        Node *tmp2;
        while(tmp){ //Data added in opposite order
                tmp2 = new Node;
                tmp2->index = tmp->index;
                tmp2->data = tmp->data;
                tmp2->next = head;
                head = tmp2;
                tmp = tmp->next;
        }
}
```

Read/Write Element

- We want two methods that allow us to find a node of the linked list with a specific index (if it exists)
  - The we read the data
  - Or we change it
- If the node does not exist
  - We return 0
  - Or we create it

Find Element

- To make the previous functions easier, write a helper function

  Node *find_element(int index, Node *head)

- Meaning: given a linked list (head) return the first node with the requested index. If none exists, return NULL (0)
- Exercise: program this

Find Element

```
Node *find_element(int index, Node *head)
{
        while(head){
                if(head->index == index) return head;
                head = head->next;
        }
        return 0;
}
```

Read Element

- Now read_element is easy

```
double Vector::read_element(int index)
{
        struct Node* tmp = find_element(index,head);
        if(tmp) return tmp->data;
        return 0;
}
```

Write Element

- For write_element we have to add non-existing elements

```
void Vector::write_element(int index, double e)
{
        struct Node* tmp = find_element(index,head);
        if(tmp) tmp->data = e;
        else{
                tmp = new Node;
                tmp -> index = index;
                tmp -> data = e;
                tmp -> next = head;
                head = tmp;
        }
}
```

Easy parts

- The << operator still works!
  - Because we wrote it using read_element()
  - Could be optimized to run faster
- The = operator is almost identical to copy constructor!

The [ ] operator

- A design decision must be made for the [ ] operator
  - What does it do if we try to access a non-existing index?
  - Does it add it to the list?
  - Does it simply return 0?
    - But then, how can it return a double &?
    - How will we be able to write on the returned element?

The [ ] operator

This implementation adds to the list non-existing requested indices

```
double & Vector::operator[ ](int index)
{
        Node *tmp = find_element(index,head);
        if(tmp) return tmp->data;
        tmp = new Node;
        tmp -> index = index;
        tmp -> data = 0;
        tmp -> next = head;
        head=tmp;
        return tmp->data;
}
```