C++ Programming for Engineers

Petar Jervic
Francisco Luro
Andrew Moss (course-resp)

May 2, 2012

Encapsulation

- Hiding details
- Syntax for member functions
- Scoping rules
- Specifying interfaces
- Linked List Example

Managing Complexity

As humans we can only remember 5 ± 2 things at once

- Working memory size limits the complexity of tasks
- Engineers need to tackle larger problems than just seven pieces
- Learning how to manage complexity is a question of focus

As humans we have millions or billions of memories

- Focus dictates which 5 ± 2 we use at one time
- The rest are hidden details
- Grouping, clustering, chunking . . .
- We need to learn to apply these techniques to programming

Details

All details are important

- Not all details are important all of the time

The first LP of the course focused on teaching you the details

- Now we will focus on learning when they can be ignored
- Ignoring details in this way is called information hiding
- Writing programs using information hiding is called encapsulation

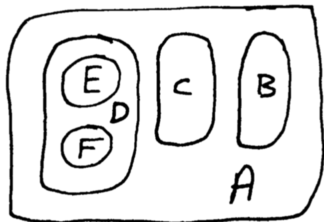
Encapsulation

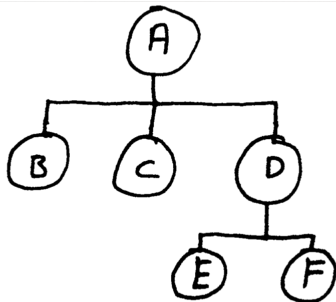
Enclosing within a solid shell, such as a capsule

Hierarchy

A hierarchy is an organisation of components ranked under one other

- Many such human organisations: governments, companies etc ...
- In Computer Science we draw hierarchies as trees
  - Normally upside down, root at the top, leaves at the bottom
- In Mathematics they are partially ordered sets
  - Normally think of them as inclusion relations

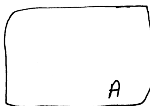




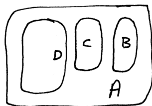
Hierarchy is repeated encapsulation

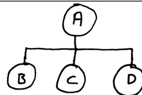
The goal is to hide details through encapsulation

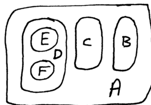
- This means putting one thing inside another
- If we repeat this many times then we form a tree
- We can forget about the innermost things (the details)
  - focus on the outermost things (an overview)

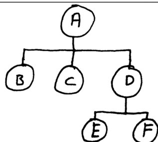


A









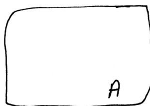
Interfaces

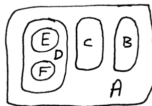
When we replace all of the details with a simplified view

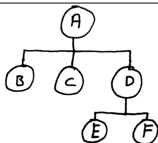
- Is it a good approximation of what the full version does?
- Does it model the full behaviour well?
- Do we lose anything important about how the full version works?

To answer these questions we need to consider the interface

- A description of how the component works with those around
- What kind of data is exchanged?
- Which parts of the code interact with each other?







Functions

We've already seen this process before

- When we put code inside a function we can forget the details
- We just have to think about what the function does
- This lets us write more complex code by combining the functions

```
int **leftSide = makeArray(10,10);
int **rightSide = makeArray(20,20);
```

Using the function we defined last week

- Details are the memory management, the loops, filling the pointers
- The overview is that we make two arrays of known sizes
- We see the inputs (size parameters), the outputs (2D arrays)
- The name of the function is clue to us about the functionality
- The interface in this case is the prototype

Call-graphs

When one function (X) calls another (Y)

- The functionality / behaviour of Y is used in X
- The functionality / behaviour of Y builds upon X

These relations form a hierarchy of functions, called the call-graph

- It implies that these functions are related / similar to each other
- There are different ways to organise this similarity
  - Today we will look at classes
  - In a few weeks we will look at libraries
- Both of these techniques are ways of organising programs

Classes

Functions are a technique for organising related pieces of code

- But sometimes are concerned with a particular piece of functionality
- This can be more than just code
- When the functionality requires state we need to store data as well

Classes are a way of combining related functions and data

- The implementation details are hidden within the class
- The external interface is enough to use the class
- We have seen this with provided classes already: strings, vectors etc . . .
- It is also possible to define custom classes

Containers

Containers are classes that store data in a certain way

- In the lecture we will look at a linked list
- In the exercises for this week you will implement a vector
  - The extension in the exercises is a hashtable

In each case the questions to ask are the same

- What is the functionality? What should the class do ...
- How do we use it? What does the interface look like...
- What data should the class store?
- How / When is the data manipulated?

Break

What is a linked list?

Container for storing data in a sequence (order)

- Generally lists store any type of data
- For simplicity we will start with a linked list of integers
- Unlike an array a list grows and shrinks during use
- Arrays allow random access (every element addressed by an index)
- Lists only allow sequential access (always start at the head)

When we implement a list we need to think about how it will be used

- Making new (empty) lists
- Adding elements to the list will be useful
- Removing elements
- Some way of reading the list. . .

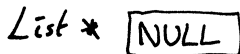
How do we build a linked list?

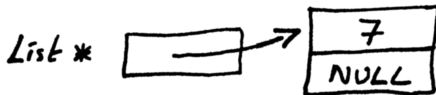
Previous description of *“what”* is not detailed enough

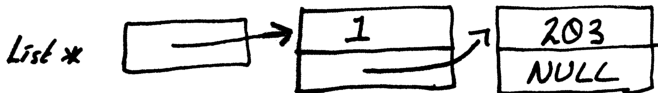
- Now we sketch what some simple lists will look like
- Give us ideas on how they need to manipulated
- Which in turn tells us how we will use / access / build them

Empty list is the most simple, then $[7]$, finally $[1, 203]$

- They can grow and shrink because they are separate pieces
- Linked together by pointers







What does the interface look like?

This always depends on the target usage, lets consider possibilities

Creation

- All empty lists look the same — no information required
- Must look like `List create();`

Adding a new value to a list

- Where does it end up within the sequence?
- Could always make the same choice (front, end, etc. . .)
- Give the choice to the programmer (caller)
  - Insert a number of places into the list
    `void insert(List this, int data, int place);`
  - Insert relative to a value
    `void insert(List this, int data, int after);`
  - Insert relative to an item
    `void insert(List this, int data, List *after);`

What does the interface look like (cont)?

Reading an item

- Similar issue to adding — which item to read
- Static choice, always first / last / other . . .
- Dynamic choices
  - number of places
  - relative to value
  - relative to an item

Removing an item

- Which one?

Other calls

After we have the parts that we need, what else could be useful?

- Counting items in the list / number of items with a value . . .

What data is stored?

Each item in the list stores exactly one value

- That value is an int
- Each item needs a pointer to the rest of the list

```
class List
{
    int value;
    List *next;
};
```

Nothing unusual here, combining datatypes using a class

- This is as you've seen since the beginning of the course

How / When is the data manipulated?

Function prototypes can also be listed inside a class

- The constructor function is special — no return type
- Don't pass which list is being used explicitly
- Every function has a hidden `List *this` argument

```
class List
{
    int value;
    List *next;
public
    List();
    void insert(int, int);
    void insert(int, List*);
    void remove(int);
    void count(int);
};
```

What is a Vector?

A random access container that can change size

- Data values are accessed by index (similar to an array)
- Unlike an array the vector can grow to fit new data

When we allocate an array the size is fixed

- The size can change in different executions of the program
- Once the size is set we cannot change it
- It is not possible to push other data out of the way

A Vector is a more complicated container than an array

- There are some details hidden (encapsulated)
- The programmer using the Vector should not see the memory management

What does the interface look like?

A container is used to store data

- We need to write values into elements using an index
- We need to read values from an element addressed by an index
- It is useful to know how many elements are in the vector

It is possible to make the read and write operations look like an array

- This is called operator overloading
- Allows the `vec[x] = vec[y];` syntax to redirect to functions
- We will ignore this at first for simplicity

Reading data values

We need to specify which element by index, and retrieve a value.

- `int read(int);`

Vector Interface (cont)

Writing data values

We need to specify what value, and where to store it.

- `void write(int,int);`

Finding the size

Doesn't require any further information, returns a number

- `unsigned int size();`

Tricky issues

- What should happen if we read an element that has not been written?
- If we grow the storage when we write a value, what should we fill with?

What data is stored?

We need some actual storage, this will be an array.

```
class Vector
{
    int *storage;
    int ssize;
public
    Vector();
    void write(int,int);
    int read(int);
    int size();
};
```

Questions

- What should the constructor do?
- When is the data in a stable (correct) state?
- When does the data need to change?

This Week

You need to make some decisions first

- How to answer the previous questions
- What the functions inside the class need to do
- How your Vector class will behave

Exercises this week are about implementing this design

- But this design is not yet complete
- We want to see how you finish off the design
- Sketches? Psuedo code? Written descriptions?
  - Pick a way that makes sense for you

Once you finish off the holes in the design you will understand how your Vector works

- Then write the code

REMINDER: No lecture next week, straight onto week 10 exercises.