CS 246 - Object Oriented Programming **Fall 2016**

Lecture 6: October 13, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

6.1 Arrays of Objects

Example 6.1 *Example of Incorrect implementation of object arrays*

```
Struct Vec {
    int x, y;
    Vec (int x, int y);
};

Vec *vp = new Vec [10];
Vec moreVecs [10];
```

- *This example will fail, as structures must be initialized. This happens because there is default constructor. So,*
  - *Option 1 : Provide a default constructor*
  - *Option 2 : For stack arrays , you can do this* `Vec moreVecs[3] = { {0,0} , {0,0}, {0,0}}`
  - *Option 3 : For heap arrays*

```
Vec **vp = new Vec*[2];
vp[0] = {0,0};
vp[1] = {0,0};

...

for (int i = 0; i < 5; ++i){
    delete vp[i];
}
delete [] vp;
```

6.2 Constant Objects

`int f (const Vec & v) { ... }`

- Constant objects arise often, especially as function parameters.
- Constant objects are objects where fields can not be modified

Can we call methods on a constant object?
Yes, we can call methods on constant objects that promise not to modify any fields. This can be achieved by including `const` after the function declaration.

Example 6.2 *Function that promises not change fields*

```
struct Student {
    ...
    float grade() const; // This method doesn't modify fields
};
```

- *Note : only constant methods can be called on constant objects*

Now Consider, you want to collect usage statistics on student objects

```
struct Student{
    ...
    int numMethodCalls = 0;
    float grade () const {
        ++numMethodCalls;
        return ... ;
    }
};
```

In the current case, numMethodCalls can't be changed because its a constant object. But , mutating numMethodCalls affects only the Physical Constantness if student objects, not the Logical Constantness.

To ensure the field can be mutated, you can declare the field to be `mutable` . Mutable fields can be changed even if the object is constant

```
struct Student{
    ...
    mutable int numMethodCalls = 0;
    float grade () const {
        ++numMethodCalls;
        return ... ;
    }
};
```

6.3 Static Fields and Methods

`numMethodCalls` tracked the number of times a method was called on a particular object. If we wanted to implement tracking on all objects. You can use static members.

6.3.1 Static Members

Static Members are associated with the class itself and not with any specific instance or object. So one member is created for the class in general instead of one version for each individual object.

Example 6.3 *Example usage of static integer*

```
struct Student{
    ...
    static int numInstances;
    Student (int assns, int mt, int final): assns{assns} .... {
        ++numInstances;
    }
};
```

- *The static members must be declarations, so for each static member you must define the variable in the .cc file*
    - *In .cc :* `int Student::numInstances = 0;`
- *In essence, static fields, must be defined external to the class.*

6.3.2 Static Member Functions

Static member functions are functions whose operations don't depend on the specific instance. So references can't be made to individual objects, as a result they don't have `this`. Given this, static member functions can only call other static functions and other static fields.

Example 6.4 *Example usage of static member functions.*

```
struct Student {
    ...
    static int numInstance;
    ...
    static void printNumInstances(){
        cout << numInstances << endl;
    }
};

Student billy {60, 70, 80};
Student jane {70, 80, 90};
Student::printNumInstances(); // 2
```

6.4 Invariants and Encapsulation

Consider:

```
struct Node {
    int data;
    Node *next
    Node (int  data, Node * next) : data{data}, next{next} {}
    ...
    ~Node () { delete next; }
};
Node n1{1, new Node{n , nullptr}};
Node n2 {3, null ptr};
Node n3 {4, &n2};
```

When n2 and n3 go out of scope, n3's constructor will attempt to delete n2, but n2 is stack allocated. This will result in undefined behaviour.

The Class Node relies on an assumption for its proper operation. Next is supposed to be either a nullptr or was allocated by new. This is an **Invariant**, which is a statement that holds true. Despite this, we can't trust the user to use node properly.

Right now we can't enforce any invariants because the user can interfere with out data

Example 6.5 *A invariant in a stack is the fact that last item pushed is the first thing popped. This invarient doesn't hold if the client can rearrange the underlying data.*

It is hard to reason about programs, if we can't rely on invariants, so to enforce them we can introduce **Encapsulation**.

- Encapsulation forces clients to treat object as black boxes
  - Implementation details sealed away
  - Can only interact via provided functions
  - Creates an abstraction and re-establishes control over objects.
- To implement this you can use the keywords `private` and `public`

Example 6.6 *Using Remove to encapsulate data.*

```
struct vec {
    Vec (int x, int y);
    private :
        int x, y;
    public :
        Vec operator+(const Vec &other);
        ...
};
```

In general : Fields should be private and only methods should be public.
Better : Its Better to have a visibility set to private by default, but this would break C programs. To resolve this we introduce a second construction called `class`

Example 6.7 *Example usage of* `class`

```
class Vec {
        int x,y;
    public:
        Vec (int x, int y);
        Vec Operator+(const Vec &other);
};
```

- *The only difference between* `struct` *and* `class` *is that default visibility is set to private.*

Example 6.8 *Fixing the linked list class to enforce the invariant*

list.h

```
class List{
        struct Node; //Private Nested class and only accessible from within class
        Node * theList = nullptr;
    public:
        void addToFront(int n);
        int ith (int i) const;
        ~List();
        ...
};
```

list.cc

```
#include "list.h"

struct List::Node { //Nested class
    int data;
    Node *next;
    Node (int data, Node *next) : ... {}
    ~Node() {delete next;}
    ...
};

void List::addToFront (int n) {
    theList = new Node {n, theList};
}

int List::ith (int i) const {
    Node *cur = theList;
    for (int n = 0; n < i && cur; ++n, cur = cur ->next);
    return cur->data
}
```

- *Only list can create/manipulate Node Objects. Therefore we can guarantee the invariant that next is always either nullptr or allocated by new.*
- *But :*
  - *Now we can't traverse the list from node to node, as we would a linked list*
  - *Repeatedly calling ith to access the whole list is not ideal, as its a $O(n^2)$ traversal.*
  - *We can't expose the nodes or we loose encapsulation*

Software Engineering Topic : Design Patterns

Certain programming problems arise frequently, so we keep track of good solutions to these problems. Then we use them to solve similar problems.
Design Pattern : If you have this situation, this technique might solve it.

The solution to the problem that arose in Example 6.8, can be solved using the iterator pattern. It works by creating a class that manages access to nodes. This allows abstraction of a pointer and list traversal without exposing the actual pointers

Example 6.9 *Iterator Pattern solution to example 6.8 (E)*

```
class List{
        struct Node;
        Node * theList::nullptr
    public :
        class Iterator {
            explicit Iterator (Node *p):p{p}{}
            int &operator *() {return p->data}
            Iterator &operator ++() {p = p->next ; return *this}
            bool operator== (const Iterator &other) {return p=other.p;}
            bool operator!= (const Iteratir &other) {return !(p ==other);}
        };
        Iterator begin() {return Iterator {theList};}
        Iterator end(){return iterator {nullptr};}
        ...
};
```

```

//Client

int main () {
    List l;
    l.addToFront(1);
    l.addToFront(2);
    l.addToFront(3);
    for (List::Iterator it = l.begin(); it != l.end(); ++it){
        cout << *it << endl;
    }
}
```

Short Cut - Automatic Type Definition : the `auto` keyword assigns the same type of one variable to another and initializes the variable to match the second variable.

```
auto x = y;
```

Shorter Short Cut: Range Based For Loop

```
for (auto n : l){
    cout << n << endl;
}
```

- Class must have methods begin and end that produce iterator
- Class iterator must support !=, *, prefix++

N is a copy of an item, to be able to modify the item or to simply save on copying, you can pass by reference.

```
for (auto &n : l){
    ++n;
}
```

List client can create iterators directly : `auto it = List::Iterator {nullptr};`
This violates encapsulation, as client should be using begin/end. To resolve this, we could make Iterators constructor private, then client can't call `List::Iterator{ ... }`. But then neither can list.

To resolve this we can give list privileged access to Iterator by using the keyword `friend`

```
class {
    ...
    public:
        class Iterator {
                Node *p;
                explicit Iterator (Node *p);
            public :
                ...
                friend class List; //List has access to everything
        };
        ...
};
```

Now List can still create iterators, but client can only create iterators by calling begin/end.

Some Useful Advice

- Give classes as few friends as possible, as it weakens encapsulation.
- Once again, keep your fields private
    - If you wish to provide access, use accessors methods which obtain the value of a fields

6.4.1 Friend Functions

As we established earlier we can have friend classes, but we can also use the keyword `friend` to establish a friend function.

Example 6.10 *Friend Function Example*

```
class Vec {
    ...
    public :
    ...
    friend std::ostream &operator<< (std::ostream &out, const Vec &v);
};

//.cc

ostream &operator<< (ostream &our, const Vec & v){
    return out << v.x << ' ' << v.y;
}
```

6.5 Tools Topic : make

Currently we have to individually compile each c++ file and then we link each individual object file to form a binary. We do this to avoid having recompile sections that haven't change.

In order for us to apply this principle to larger programs without having to memorize files change, we can let Linux help us with `make`. To utilize a make, we must create a Makefile which lists files that depend on which other files.

Example 6.11 *Basic Make File Example*

```
myprogram : main.o lst.o node.o iter.o
    g++-5 -std=c++14 main .o list.o node.o iter.o myprogram
list.o : list.cc list.h node.h
      g++-5 -std=c++14 -c node.cc
..
```

Once you have a make file, you can then type `make` from command line to build the whole project. Every subsequent time you run make, it will only compile files that have modified and relinks the program with the new compiled portion. Make achieves this by creating a dependency graph which allows make to recursively loop through all the dependencies and rebuild the files that have a newer modified date compared to the corresponding object file.

In addition, you can rebuild certain targets by indicating the target file. : `make node.o`
Another common practice is to include a clean target at the end of a Makefile

Example 6.12 *Basic Make File Example*

```
myprogram : main.o lst.o node.o iter.o
.PHONY : clean

clean :
    rm *.o myprogram
```

- *To do a full rebuild you can do*
  - *make clean*
  - *make*