Constructors & Destructors

- Constructing – Destructing and Classes in C++
- USE CLASSES TO REPRESENT CONCEPTS

Destructors

- Free resources
  - Automatically called
  - Variable goes out of scope
  - Delete object in free store
  - constructor/destructors

```
class Name {const char *s};

class Table {
    Name* p;
    size_t sz;
public:
    Table(size_t s = 15)
   {p = new Name[sz=s]; }
   ~Table() {delete [] p; }
   Name* lookup(const char *);
   bool insert(Name *);
}
```

Construction and Destruction

- Constructor of local variable executed each time the thread of control passes through the declaration
- Destructor executed each time the local variable's block is exited
- Destructors for local variables are executed in reverse order of their construction

Constructor/Destructors

- Class objects are constructed from the bottom up: first the base, then the members, and then the derived class itself.
- They are destroyed in the opposite order: first the derived class, then the members, then the base
- Members/bases constructed in order of declaration – destructed in reverse

Copy and assignment constructors

Rule of thumb: everytime you have pointer members you need one

```
Thing(const Thing&);                  // copy constructor
Thing& operator=(const Thing&);   // assignment constructor
```

copy constructor needs only to allocate necessary resources
assignment operators must first cleanup then allocate resources

Hint: if you want to prohibit assignment/copying make these functio
private

Assignment operator

```
String& String::operator=(const String& s)
{
    if (&s != this)  {                  // IMPORTANT FREQUENTLY OMMITTED
        delete [] data;
        data = new char[strlen(s.data)+1];
        strcpy(data,s.data);
    }
    return *this;
}
```

Class hierachies

- protected, private, public
- virtual (can be overriden by derived classes)
- virtual ..... = 0; (just interface)
- A form of polymorphism
- To get polymorphic behavior member function must be virtual and objects manipulated through pointers or references

A classic example

```
class Employee {
   string name; ...
public:
   Employee(const string& name, int dept);
   virtual void print() const;          // provides "default" behavior
                                        // which derived classes can
                                        // can modify
}

class Manager: public Employee {
    set<Employee*> group;
public:
    Manager(...);
    void print() const;                 // can define derived specific print functio
}
```

Derived Classes

- Important to provide virtual destructor for any class that can be a base class
- When anyone ever executes a delete expression on any object of type B* that actually points to an object of type D

Surrogate classes

```
class Vehicle {
    public:
        virtual double weight() const = 0;
        virtual void start() = 0;
};

class RoadVehicle: public Vehicle {....}
class Automobile   : public RoadVehicle
```

We want to have a container of Vehicles of different kinds :

```
Vehicle parking_lot[1000];  // this doesn't work
```

Why it doesn't work

```
Automobile x = /* ..... */
parking_lot[num_vehicles++] = x;
```

This converts x to a Vehicle by slicing it (removing all members not found in the Vehicle class). The truncated object is copied which is not what we want.

In fact we have said that parking_lot is a collection of Vehicles, not a collection of objects derived from Vehicle

The classic solution

```
Vehicle*parking_lot[1000];   // array of pointers

Automobile x;
parkin_lot[num_vehicles++] = &x;
```

Works but has two disadvantages:

1) If pointer is to a local variable danger of loosing that memory

Solution: parking_log[num_vehicles++] = new Automobile(x)
copy objects, free objects pointed two
Disadvantage: burden of dynamic memory management

More problems

Even with explicit copying and freeing this works only when we know the static type of objects we wish to install in parking lot.

For example:

```
if (p != q)
  {delete parking_lot[p];
    parking_lot[p] = parking_lot[q];
   }  // will not work they will point to the same object

if (p != q) {
     delete parking_lot[p];
     parking_lot[p] = new Vehicle(parking_lot[q]);
} // doesn't work creates slicing truncated object
```

Clone (virtual copy) function

```
class Vehicle {
    public:
        virtual double weight() const = 0;
        virtual void start() = 0;
        virtual Vehicle* clone() const = 0;
        virtual ~Vehicle() {};
}

 Vehicle* Truck::clone() const
 {
     return new Truck(*this);
 }
```

Goal: way to create copies of objects whose type we do not know at compile time

The way to do anything in C++ with objects of unknow type is to use a virtual function

Surrogate class

- Something that acts like a Vehicle, but pottentially represents an objects of any class derived from Vehicle
- USE CLASSES TO REPRESENT CONCEPTS
- Each surrogate will stand for an object. The object will persist exactly as long as the surrogate is associated with it.

Implementation

```
class VehicleSurrogate {
public:
    VehicleSurrogate();
    VehicleSurrogate(const Vehicle&);
    ~VehicleSurrogate()
    VehicleSurrogate(const VehicleSurrogate&);
    VehicleSurrogate&operator=(const VehicleSurrogate &);
private:
    Vehicle *vp;
};

VehicleSurrogate::VehicleSurrogate(): vp(0) {}
```

Implementation (cntd)

```
VehicleSurrogate::VehicleSurrogate(const Vehicle& v):
    vp(v.clone()) {}

VehicleSurrogate::~VehicleSurrogate()  {delete vp; }
VehicleSurrogate::VehicleSurrogate(const VehicleSurrogate& v) :
    vp(v.vp ? v.vp->clone() : 0 {}
VehicleSurrogate&
VehicleSurrogate::operator=(const VehicleSurrogate& v)  {
    if (this != &v) {
        delete vp;
        vp = (v.vp ? v.vp->clone() :0);
    }
    return *this;
}
```

Some more details

```
double VehicleSurrogate::weight() const
{
    if (vp == 0)
        throw “empty VehicleSurrogate.weight()”;
    return vp->weight();
}
```

Functions of the base Vehicle class must be redirected that way

What have we achieved ?

```
VehicleSurrogate parking_lot[1000];
Automobile x;
parking_lot[num_vehicles++] = x;
```

same thing as:

```
parking_lot[num_vehicles++] = VehicleSurrogate(x);
```

When parking lot deleted all copies are deleted.

Some problems: copies can be costly for large objects (use counts are a possible solution)