AGGREGATION WITH SMART POINTERS

Automating heap memory management

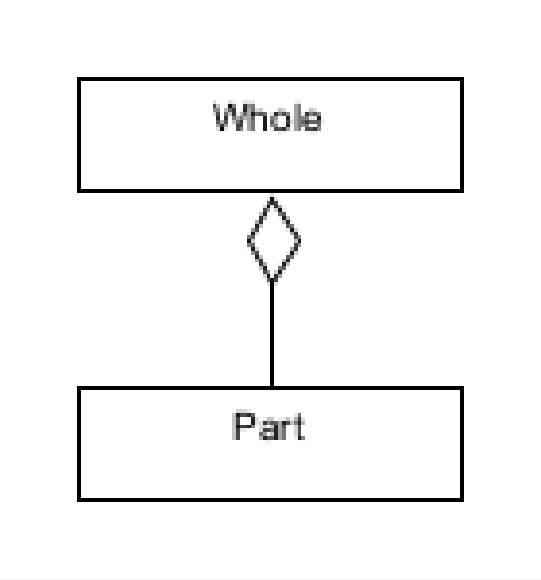
Delroy A. Brinkerhoff

AGGREGATION WITH RAW POINTERS

```
class Whole
{
    private:
        Part* part = nullptr;
};

Whole() {}
Whole(Part* p) : part(p) {}
```

```
~Whole()
{
    if (part != nullptr)
        delete part;
}

void set_part(Part* p)
{
    if (part != nullptr)
        delete part;
    part = p;
}
```

SIMPLE AGGREGATION WITH SMART POINTERS

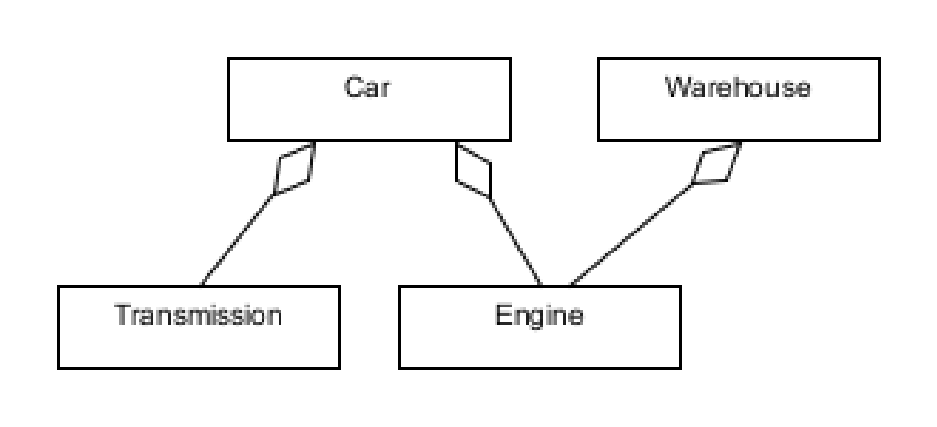


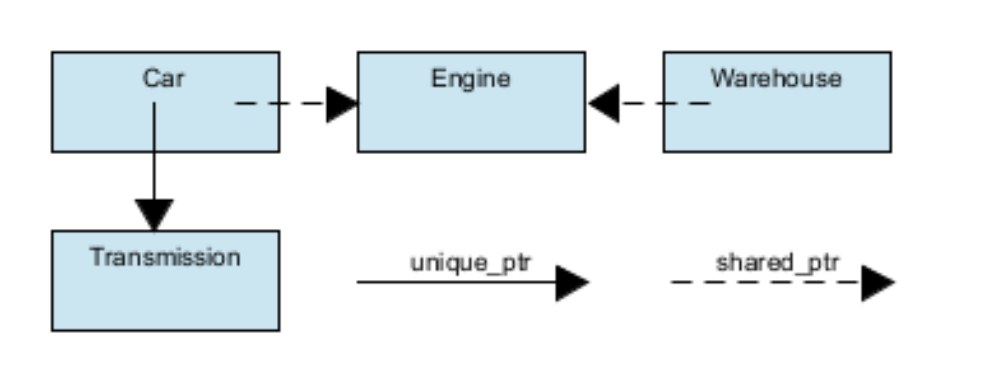

```
int main()
{
    Whole whole("Widget");
    whole.display();
    whole.set_part(new Part("Bolt"));
    whole.display();

    return 0;
}
```

THE WHOLE AND PART CLASSES

```
class Part
{
    private:
        string name;
    public:
        Part(string n) : name(n) {}
        ~Part() { cout << "Part dtor: “ << name << endl; }
        void display() { cout << name << endl; }
};

class Whole
{
    private:
        shared_ptr<Part> part;
    public:
        Whole(string n) { part = make_shared<Part>(n); }
        ~Whole() { cout << "Whole dtor\n"; }
        void set_part(Part* n) { part.reset(n); }
        void display() { cout << "Whole: "; part->display(); }
};
```

SHARED AGGREGATION WITH SMART POINTERS





THE Car CLASS

```
class Car
{
    private:
        unique_ptr<Transmission>  trans;
        shared_ptr<Engine>        engine;

    public:
        Car(string t) : trans(make_unique<Transmission>(t)) {}
        ~Car() { cout << "Car dtor" << endl; }
        void set_engine(shared_ptr<Engine> e) { engine = e; }
        friend ostream& operator<<(ostream& out, Car& me)
        {
            out << *me.engine << " " << *me.trans.get();
            return out;
        }
};
```

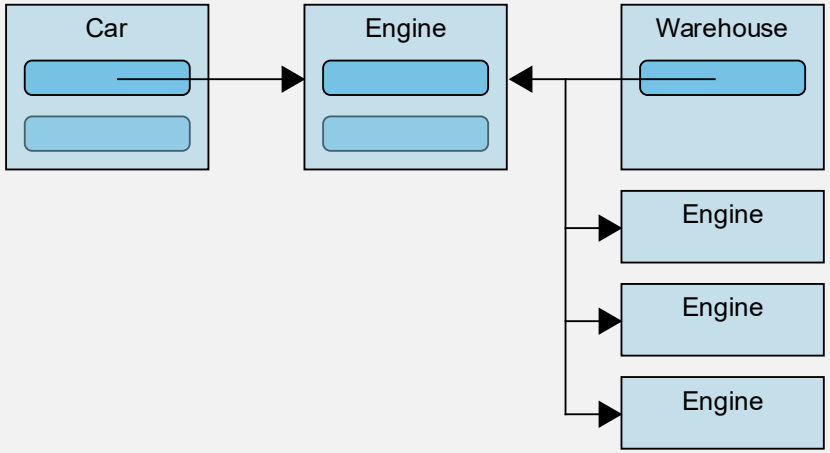
THE Warehouse CLASS

```
class Warehouse
{
    private:
        shared_ptr<Engine>  engine;

    public:
        ~Warehouse() { cout << "Warehouse dtor" << endl; }
        void set_engine(shared_ptr<Engine> e) { engine = e; }
        friend ostream& operator<<(ostream& out, Warehouse& me)
        {
            out << *me.engine;
            return out;
        }
};
```

THE DRIVER

```
int main()
{
    Car                         c("Automatic");
    Warehouse                   w;
    shared_ptr<Engine>          e = make_shared<Engine>(440);

    c.set_engine(e);
    w.set_engine(e);

    cout << "(1) Engine: " << *e << endl;
    cout << "(2) Car: " << c << endl;
    cout << "(3) Warehouse: " << w << endl << endl;

    e = make_shared<Engine>(380);
    //e.reset(new Engine(380));     // alternative
    c.set_engine(e);
    w.set_engine(e);
}
```

MULTIPLE SHARED POINTERS



- The Warehouse "owns" and manages the Engines
- Previously implemented with an array of pointers:
  - `Engine* spares[10];`
  - Limits the number of Engines
- Replace the array with an STL `vector`

THE UPDATED Warehouse CLASS

```
class Warehouse
{
    private:
        vector<shared_ptr<Engine>> engines;

    public:
        ~Warehouse() { cout << "Warehouse dtor" << endl; }

        void add_engine(shared_ptr<Engine> e) { engines.push_back(e); }
        shared_ptr<Engine> get_engine(int index) { return engines[index]; }

        void display(int index) { engines[index].get()->display(); }
        friend ostream& operator<<(ostream& out, Warehouse& me)
        {
            vector<shared_ptr<Engine>>::iterator i = me.engines.begin();
            while (i != me.engines.end())
                out << "\t" << **i++ << endl;
            return out;
        }
};
```

THE UPDATED DRIVER

```
int main()
{
    Car          c("Automatic");
    Warehouse    w;

    w.add_engine(make_shared<Engine>(454));
    w.add_engine(make_shared<Engine>(440));
    w.add_engine(make_shared<Engine>(429));
                . . . . .

    c.set_engine(w.get_engine(1));

    cout << "(1) Car: " << c << endl;
    cout << "(2) Warehouse:\n" << w << endl;

    c.set_engine(w.get_engine(5));

    cout << "(3) Car: " << c << endl;
    cout << "(4) Warehouse:\n" << w << endl;
}
```

SMART POINTER SUMMARY

- Smart pointers automatically deallocate objects allocated on the heap
  - Eliminate memory leaks
  - Eliminate destructors whose sole task is destroying dynamic objects
  - In the case of shared objects, they eliminate ownership protocols