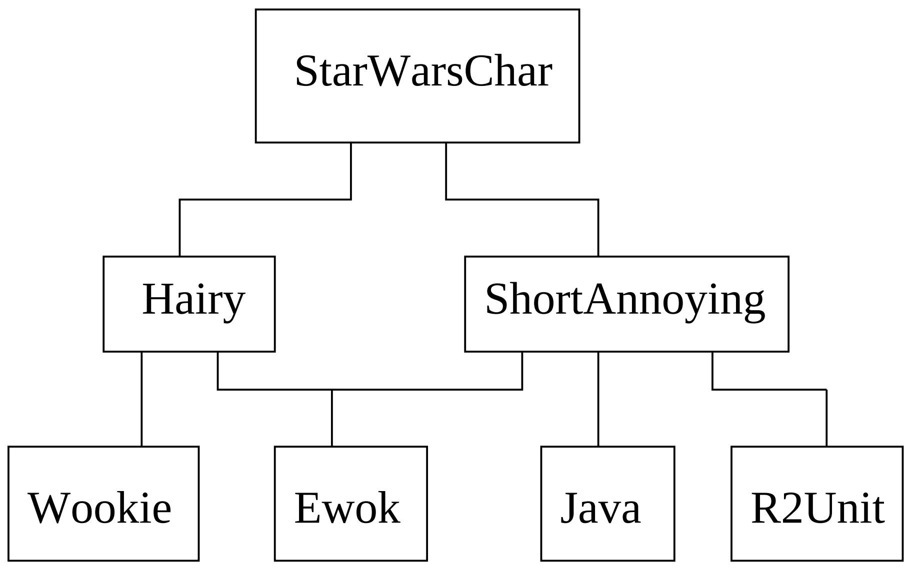
```
class StarWarsMovie {
// private default for class, public default for struct
  int episode_no;
  int year;
  std::string director;
  std::string title;
public:
  StarWarsMovie();
  StarWarsMovie(std::string t,
                int y,
                int e,
                std::string d = "Lucas"); // default arg.
  ~StarWarsMovie();
// your friends are allowed to look at your private parts
  friend std::ostream &operator<<(const std::ostream &out,
                                  const StarWarsMovie &m);
};
```




```
class StarWarsChar{};
class Hairy : public StarWarsChar { public: Hairy(int h);};
class ShortAnnoying : public StarWarsChar {};
class Ewok : public Hairy, public ShortAnnoying {
public:
  Ewok();
  Ewok(int h);
};
```

Tre steg i ett objekts tillblivande:

1. Minnesallokering, automatiskt eller dynamiskt.

2. Initiering av medlemmar, via initieringslistan eller med defaultvärden. Extra viktigt för const och referensmedlemmar. Initieras i den ordning de kommer i klassdeklarationen.

3. Exekvering av koden i konstruktorn.

```
StarWarsMovie::StarWarsMovie() : episode_no(2), year(2002),
director("Lucas"), title("Clones") {
 // title = "AOTC"; // better to use :-list
}
Ewok::Ewok(int no_of_hairs) : Hairy(no_of_hairs) {}
```

```
class EwokGang {
  Ewok *ewoks;
public:
  const unsigned int no_of_ewoks;
  EwokGang(const unsigned int no) : no_of_ewoks(no) {
    if(no)
      ewoks = new Ewok[no];
  }
  ~EwokGang();
};

EwokGang g1(10);
EwokGang g2 = 1; // oops, should be g2 = g1
g2.no_of_ewoks; // 1, make constructor explicit instead

EwokGang::~EwokGang() {
  delete [] ewoks;
}
```

En del saker skapas automatiskt.

```
struct StarWarsLine {
  std::string str;
};
int main() {
  StarWarsLine l; // StarWarsLine()
  l.str = "Luke, I am your father.";
  StarWarsLine again(l); // StarWarsLine(const StarWarsLine &)
  StarWarsLine and_again;
  and_again = l; // StarWarsLine::operator=(const StarWarsLine &)

  return 0; // ~StarWarsLine()
}
```

```
int main() {
  Ewok e;
// runs StarWarsChar(), Hairy(), StarWarsChar(),
// ShortAnnoying(), Ewok()
  return 0;
// runs ~Ewok(), ~ShortAnnoying(), ~StarWarsChar(),
// ~Hairy(), ~StarWarsChar()
}
```

```
struct BadAnimatedChar {
  void speak() { std::cout << "Hello\n";}
};

struct Gungan : BadAnimatedChar {
  void speak() { std::cout << "Meesa speaka Gungan\n";}
};

int main() {
  BadAnimatedChar *chars[2];
  chars[0] = new Gungan();
  chars[0]->speak(); // "Hello"
  return 0;
}
```

```
struct AnimatedChar {
  virtual void speak() { std::cout << "Hello\n";};
};

struct GreenMidget : AnimatedChar {
  void speak() { std::cout << "Speak weirdly, I do\n";}
};

int main() {
  AnimatedChar *chars[2];
  chars[0] = new GreenMidget();
  chars[0]->speak(); // "Speak weirdly, I do"
  return 0;
}
```

Göra destruktorn virtual i basklassen viktigt om man ärver och använder pekare/referenser till basklasser.

Virtuella funktioner avgörs vid runtime baserat på det påpekade objektets faktiska typ, andra funktioner väljs vid compile time baserat på pekartypen. Vill man ha polymorfism ska man alltså använda virtual.

Strikt virtuella (pure virtual) funktioner måste överskrivas (override), annars kan inte objekt instansieras.

```cpp
class DeathStarBase {
  virtual void protect_not_so_hidden_weak_spot() = 0;//must override!
};

class DeathStar : DeathStarBase {
  void protect_not_so_hidden_weak_spot() {
    std::cout << "Fix later\n";
  }
};

int main() {
  DeathStarBase db; // does not compile
  DeathStar db2; // ok
  DeathStarBase *pointer = &db2; // ok
  return 0;
}
```

```
struct DroidArmy {
  int droids, droidekas;
  DroidArmy &operator=(const DroidArmy &da);
  DroidArmy &operator++();
  DroidArmy &operator--();
  DroidArmy &operator+=(const DroidArmy &da);
  DroidArmy &operator-=(const DroidArmy &da);
  DroidArmy &operator*=(const DroidArmy &da);
  DroidArmy &operator/=(const DroidArmy &da);
  bool operator>(const DroidArmy &da);
  bool operator<(const DroidArmy &da);
  bool operator>=(const DroidArmy &da);
  bool operator<=(const DroidArmy &da);
  bool operator==(const DroidArmy &da);
  bool operator!=(const DroidArmy &da);
};
```

```
DroidArmy operator+(const DroidArmy &d1, const DroidArmy &d2);
DroidArmy operator-(const DroidArmy &d1, const DroidArmy &d2);
DroidArmy operator*(const DroidArmy &d1, const DroidArmy &d2);
DroidArmy operator/(const DroidArmy &d1, const DroidArmy &d2);

std::ostream &operator<<(std::ostream &out, const DroidArmy &da);
```

Varför överlagring? Kort skrivsätt (++ istället för add(1)), lätt att skriva, lätt att läsa. Templates.

Vissa operatorer returnerar referens till objektet själv. Det använder man this-pekaren till.

Tre sorters const.

```
class LightSabre {
  enum Colour{Red, Green, Purple};
  static int counter;

  Colour colour;
  int no;
public:
  LightSabre(Colour c) : colour(c), no(counter++) {}
  LightSabre(const LightSabre &s) : no(counter++) {
    colour = s.colour;
  }
  Colour &get_colour() { return colour; }
  const Colour &get_colour() const { return colour; }
};
int LightSabre::counter = 0;
```