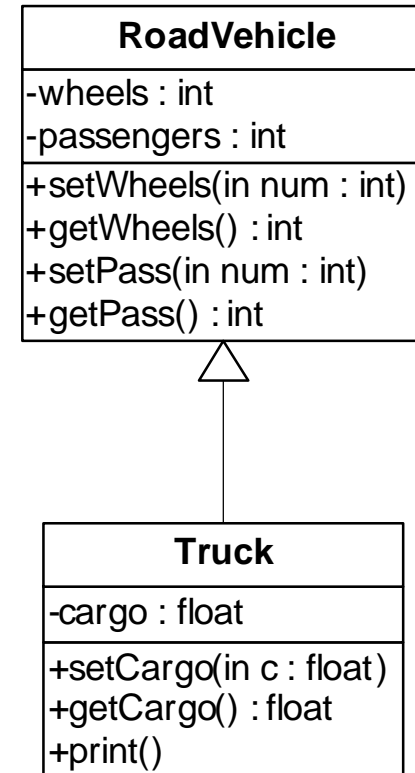


Virtuālās funkcijas

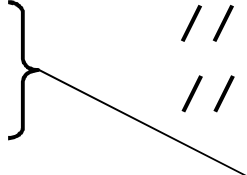
```
class RoadVehicle{
private: int wheels;
        int passengers;
public:
    void setWheels(int num) { wheels = num; }
    int getWheels() { return wheels; }
    void setPass(int num) { passengers = num; }
    int getPass() { return passengers; }
};
```

```
class Truck : public RoadVehicle{
private: float cargo;
public:
    void setCargo(float weight) { cargo = weight; }
    float getCargo() { return cargo; }
    void print();
};
```



Virtuālās funkcijas (turp.)

```
void main()
{ Truck t1;
. . .
  RoadVehicle *p; // bāzes klases rādītājs
  p = &t1;         // norāda uz objektu t1
  p->setPass(1);    // pasažieru skaits
//      p->setCargo(2.5); } // Kļūda!
//      p->print();      } // Kļūda!
. . .
}
```



setCargo() un *print()* nav bāzes klases *RoadVehicle* locekļi!

Lai varētu piekļūt atvasināto klašu funkcijām, izmantojot bāzes klases rādītāju, tām jābūt definētām kā virtuālām funkcijām.

- n Virtuālo funkciju deklarē bāzes klasē un atvasinātajās klasēs.
- n Virtuālo funkciju atgriežamajiem vērtību tiem un parametriem ir jāsakrīt.
- n Ja kādā atvasinātajā klasē virtuālā funkcija nav pārdefinēta, tad tiks izsaukta bāzes klases virtuālā funkcija.

Virtuālās funkcijas (turp.)

```
class RoadVehicle{
private:
    int wheels;
    int passengers;
public:
    . . .
    void setPass(int num) { passengers = num; }
    virtual void print();
};

void RoadVehicle::print()
{
    cout << "wheels " << wheels << '\n';
    cout << "passengers " << passengers << '\n';
}
```

Virtuālās funkcijas (turp.)

```
class Truck : public RoadVehicle{
private:
    float cargo;
public:
    . . .
    void setCargo(float weight) { cargo = weight; }
    virtual void print();        //pārdefinēta metode
};
```

```
void Truck::print()
{
    cout << "Truck :" << '\n';
    cout << "wheels " << getWheels() << '\n';
    cout << "passengers " << getPass() << '\n';
    cout << "load " << getCargo() << " t" << "\n\n";
}
```

Virtuālās funkcijas (turp.)

```
class Car : public RoadVehicle{
private:
    int doors;
public:
    . . .
    void setDoors(int n) { doors = n; }
    virtual void print();          //pārdefinēta metode
};
```

```
void Car::print()
{
    cout << "Car :" << '\n';

    RoadVehicle::print();    // use base class member function

    cout << "doors " << getDoors() << "\n\n";
}
```

Virtuālās funkcijas (turp.)

```
void main()
{
    Truck t;
    Car c;
    RoadVehicle *p;
    . . .

    p = &t;
    p->setPass(1);          // setPass() ir bāzes klases funkcija
    p->print();              // print() ir virtuāla funkcija,
                            // šeit izpildīs klases Truck funkciju print()

    Truck *tp;
    tp = (Truck*)p;         // lietojam atvasinātās klases rādītāju
    tp->setCargo(2.5);       // setCargo() ir klases Truck funkcija

    p = &c;
    p->setPass(4);
    ((Car*)p)->setDoors(4);  // uzmanīgi ar operāciju izpildes secību!
    p->print();              // print() ir virtuāla funkcija
                            // šeit izpildīs klases Car funkciju print()98
}
```

Virtuālās funkcijas (turp.)

Virtuālām funkcijām darbojas šādi noteikumi:

- n Klases konstruktori nevar būt virtuāli, bet destruktori var.
- n Virtuālas funkcijas esamība bāzes klasē neuzliek pienākumu šo funkciju pārdefinēt atvasinātajā klasē (ja vien tā nav tīri virtuāla).
- n Funkcija, kas ir deklarēta kā virtuāla, tāda paliek arī visās atvasinātajās klasēs, neatkarīgi no tā, vai atvasinātajās klasēs tiek rakstīts atslēgvārds **virtual** vai nē.
- n Virtuālās funkcijas argumentiem un atgriežamajam tipam jāsakrīt gan bāzes klasē, gan atvasinātajās klasēs, pretējā gadījumā notiks bāzes klases funkcijas slēpšana un polimorfisms nebūs spēkā.
- n Virtuāla funkcija nevar būt statiska.

Virtuālās funkcijas (turp.)

- n Virtuālās funkcijas lieto, ja bāzes klases funkcijām nepieciešams definēt citu funkcionalitāti atvasinātajā klasē. To sauc arī par klases metožu *pārdefinēšanu*.
- n Lai noteiktu, kura no metodēm ir jāizsauc, valodā C++ tiek pielietots *polimorfisma* principa izpausme - **dinamiskā (vēlā) sasaistīšana** (*late binding*).
Tas nozīmē, ka izsaucamā metode (funkcija) tiek noteikta programmas izpildes laikā, nevis kompilēšanas laikā
- n Ja izsaucamo metodi (funkciju) nosaka jau kompilēšanas laikā, tad to sauc par **statisko (agro) sasaistīšanu** (*early binding*).

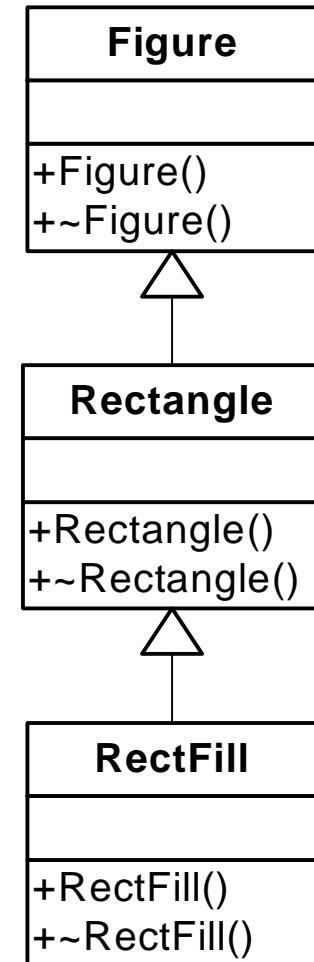
Virtuālie destruktori

```
void main()
{
    RectFill *prf;
    prf = new RectFill;
    // ...

    Figure *pf = prf;
    // ...

    delete pf;
}
```

```
Figure created
Rectangle created
RectFill created
Figure destroyed
Press any key to continue_
```



RectFill un Rectangle destruktorus neizpilda!

Virtuālie destruktori (turp.)

```
class Figure
{
public:
    Figure() { cout << "Figure created\n"; }
    virtual ~Figure() { cout << "Figure destroyed\n"; }
};

class Rectangle : public Figure
{
public:
    Rectangle() { cout << "Rectangle created\n"; }
    virtual ~Rectangle() { cout << "Rectangle destroyed\n"; }
};

class RectFill : public Rectangle
{
public:
    RectFill() { cout << "RectFill created\n"; }
    virtual ~RectFill() { cout << "RectFill destroyed\n"; }
};
```

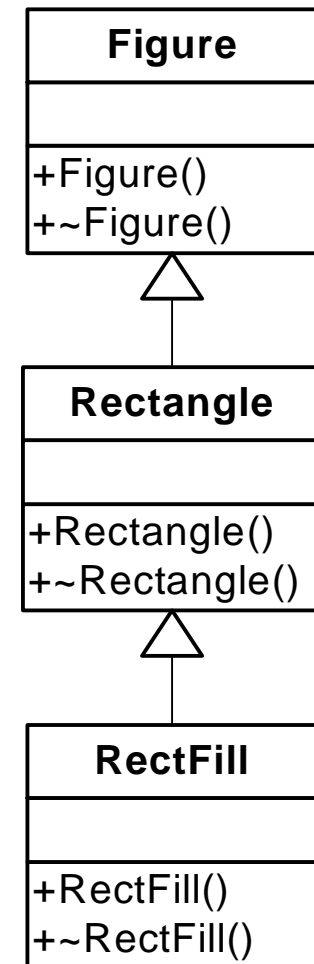
Virtuālie destruktori (turp.)

```
void main()
{
    RectFill *prf;
    prf = new RectFill;
    // ...

    Figure *pf = prf;
    // ...

    delete pf;
}
```

```
Figure created
Rectangle created
RectFill created
RectFill destroyed
Rectangle destroyed
Figure destroyed
Press any key to continue
```



Izpilda visus destruktorus!

Virtuālie destruktori (turp.)

- n Ja nepieciešams iznīcināt objektu, izmantojot rādītāju, kura tips ir kāds no objekta bāzes klases tipiem, ir svarīgi, lai tiktu izsaukti visi objekta un tā bāzes klašu destruktori.
- n Lai to nodrošinātu, nepieciešams bāzes klases destrukturu deklarēt kā *virtuālu*.

Tīrās virtuālās funkcijas un abstraktās klases

- n Tīru virtuālu funkciju (*pure virtual function*) bāzes klasē tikai deklarē.
- n Tīra virtuāla funkcija obligāti jāpārdefinē visās no bāzes klases atvasinātajās klasēs.
- n Tīras virtuālas funkcijas deklarācija:

`virtual tips vārds (parametri) = 0;`

- n Ja klasē ir deklarēta kaut viena tīra virtuāla funkcija, tad šādu klasi sauc par abstraktu.
- n Abstraktai klasei nevar būt instances, t.i. nav iespējams radīt tās objektus.

Tīrās virtuālās funkcijas un abstraktās klases

```
class Figure{
protected:
    double x;
    double y;
public:
    Figure();
    Figure(double, double);

    void setCoord(double x, double y)
    {this->x=x; this->y=y;}

    void getCoord(double *x , double *y)
    {*x=this->x; *y=this->y;}

    virtual double area() = 0;
};
```

Tirās virtuālās funkcijas un abstraktās klases

```
class Rectangle : public Figure {  
private:  
    double w, h;  
public:  
    . . .  
    void resize(double neww, double newh);  
  
    virtual double area();  
};
```

```
void Rectangle::resize(double neww, double newh)  
{ w = neww; h = newh; }
```

```
double Rectangle::area()  
{ return ( w * h ); }
```

Tirās virtuālās funkcijas un abstraktās klases

```
class Circle : public Figure {  
private:  
    double r;  
public:  
    . . .  
    void resize(double newr);  
  
    virtual double area();  
};
```

```
double Circle::area()  
{ return ( 3.14159 * r * r ); }
```

```
void Circle::resize(double newr)  
{ r = newr; }
```


Tirās virtuālās funkcijas un abstraktās klases

```
// Funkcija printArea() izsauc klases Rectangle vai Circle
// virtuālo funkciju area() atkarībā no saņemtā rādītāja p vērtības
    void printArea( Figure *p )
    { cout << p->area() << '\n'; }

void main()
{ Figure *fp1, *fp2;

    fp1 = new Rectangle(2, 3);
    fp1->getCoord( &x, &y );
    ((Rectangle*)fp1)->getSize( &w, &h );
    cout << "Area of rectangle : " << fp1->area() << '\n';

    fp2 = new Circle(1);
    cout << "Area of circle : " << fp2->area() << '\n';
    cout << "Area of circle : ";

    printArea( fp1 ); printArea( fp2 );
}
```

Tīrās virtuālās funkcijas un abstraktās klases

Gan klasei Rectangle, gan klasei Circle ir funkcija `resize()`.

Kāpēc šo funkciju nevaram deklarēt kā tīru virtuālu funkciju bāzes klasē Figure?