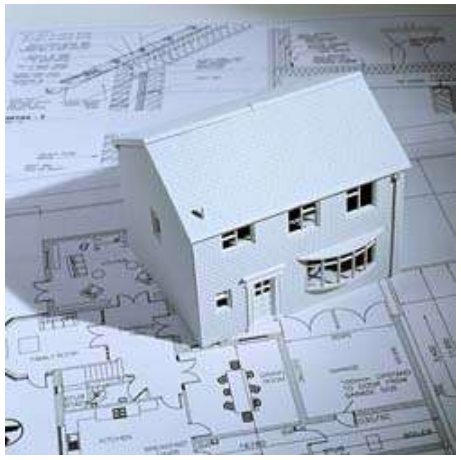
PROGRAMIRANJE II

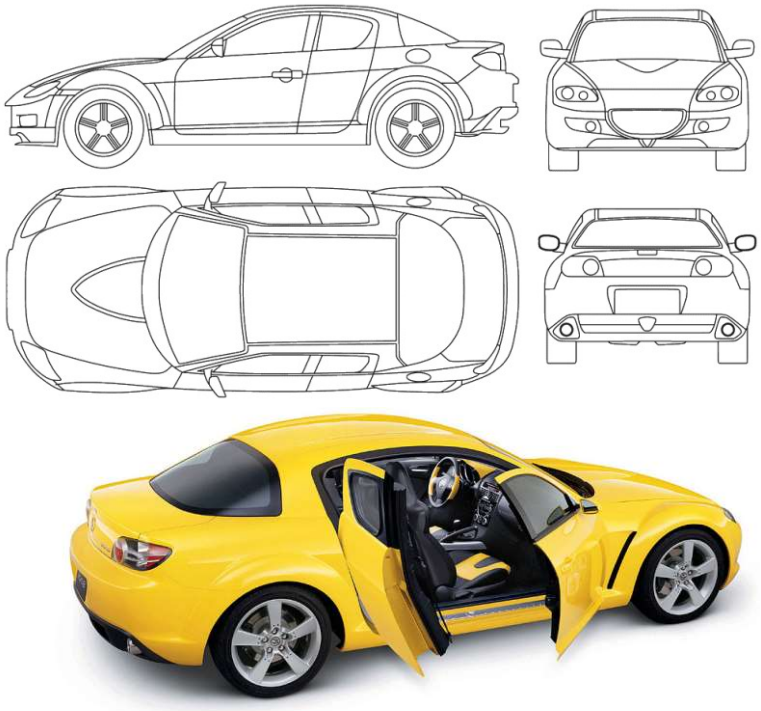
Programiranje: znanost ali umetnost?

- "If computer programming is to become an important part of computer research and development, a transition of programming from an art to a disciplined science must be effected." (CACM 1959)

Objekti in razredi

- Objekt – ograjuje stanje in obnašanje
- Razred – šablona za ustvarjanje objektov





Objekti in razredi

- Stanje oz. strukturo objektov opišemo s spremenljivkami, ki jim pravimo spremenljivke objekta ali tudi instančne spremenljivke (*instance variables*), obnašanje pa s sporočili oz. z metodami (*methods*).
- Pravimo tudi, da je objekt množica metod, ki si delijo stanje.

Definicija razreda

```
class X {
  private:
    // podatki (instančne spremenljivke)
  public:
    // metode
};
```

Definicija razreda

- Definicija razreda – vključitvena datoteka
- Implementacija razreda – glavna datoteka
- Z razredom definiramo nov tip (ADT), ki se obnaša podobno kot vgrajeni tip.
- Kapsuliranje oz. ograjevanje

Ustvarjanje objekta

- Kreiramo lahko spremenljivke takšnega razreda (tipa)
- Spremenljivko imenujemo objektna spremenljivka ali objekt

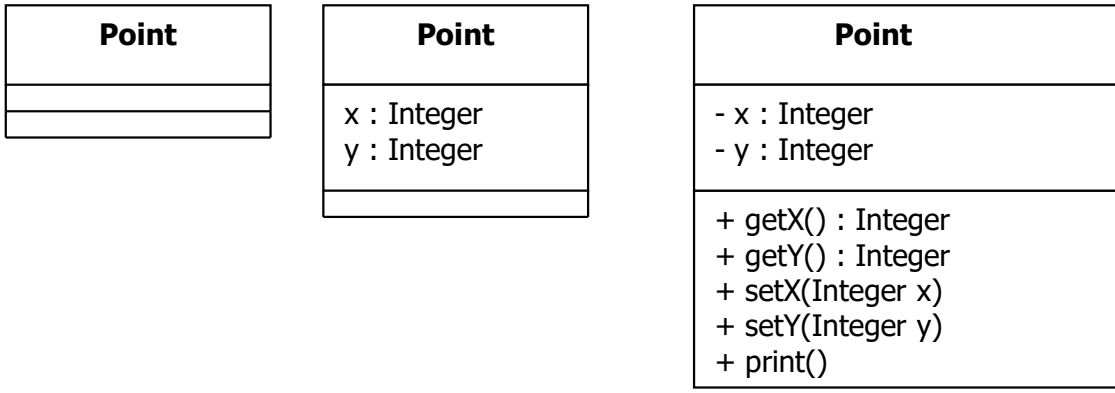
```
// example.cpp

int a;
Stack my_stack1;
X o;
```

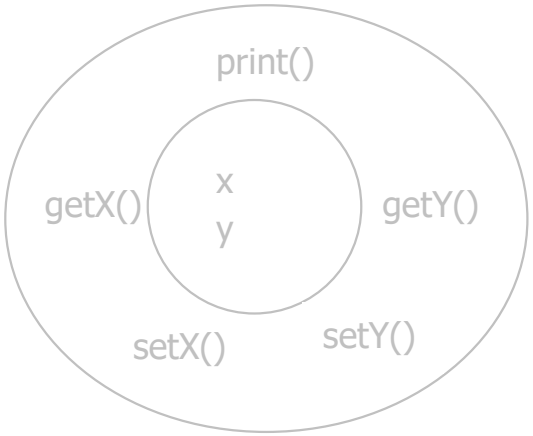
Primer 1 – definicija razreda

// Point.h

```
class Point {
private:
    int x, y;
public:
    int getX();
    int getY();
    void setX(int x);
    void setY(int y);
    void print();
};
```

UML notacija





Primer 1 – implementacija razreda

// Point.h

```
class Point {
private:
    int x, y;
public:
    int getX();
    int getY();
    void setX(int x);
    void setY(int y);
    void print();
};
```

// Point.cpp

```
#include <iostream>
#include "Point.h"

int Point::getX() {
    return x;
}
int Point::getY() {
    return y;
}
void Point::setX(int x) {
    this->x=x;
}
void Point::setY(int y) {
    this->y=y;
}
void Point::print() {
    std::cout << "(" << x << "," << y << ")" << std::endl;
}
```

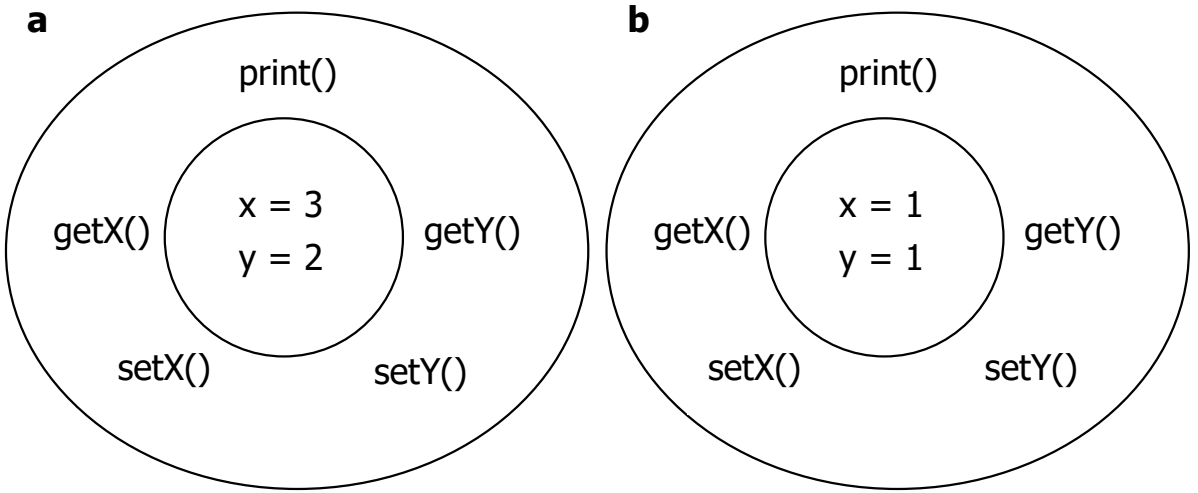
Primer 1 – ustvarjanje objektov

// Example01.cpp

```
#include <iostream>
#include "Point.h"

int main() {
    Point a;
    Point b;
    a.setX(3);
    a.setY(2);
    b.setX(1);
    b.setY(1);
    std::cout << a.getX() << std::endl;
    std::cout << a.getY() << std::endl;
    //b.x=10;
    a.print();
    b.print();
    return 0;
}
```

UML notacija

a:Point	b:Point
x : Integer = 3 y : Integer = 2	x : Integer = 1 y : Integer = 1



Dobro je vedeti

- Skrivanje komponente (data hiding) razreda dosežemo z določilom **private**.
- Komponento izvozimo z določilom **public**.
- Vse komponente, ki so skrite lahko načrtovalec razreda poljubno spreminja ne da bi to vplivalo na programsko kodo, ki uporablja ta razred.
- Ograjevanje (kapsuliranje) je eno izmed temeljev OOP, saj omogoča enostavnejšo spreminjanje in vzdrževanje programov.

Dobro je vedeti

- Kratke metode lahko zapišemo kar v definicijo razreda (vključitvena datoteka)
- Prevajalnik jih bo obravnaval kot vrinjene (*inline*) metode.

// Point.h

```
class Point {
private:
    int x, y;
public:
    int getX() {return x;}
    int getY() {return y;}
    void setX(int x) {this->x=x;}
    void setY(int y) {this->y=y;}
    void print();
};
```

Dobro je vedeti

- Razred lahko definiramo tudi s ključno besedo **struct**.
- Če razred definiramo s **struct** in ne podamo določil (**private**) so vse komponente javne.
- Če razred definiramo s **class** in ne podamo določil (**public**) so vse komponente skrite.

// Point.h

```
struct Point {
    int getX();
    int getY();
    void setX(int x);
    void setY(int y);
    void print();
private:
    int x, y;
};
```

Konstruktorji in destruktorji

- Potrebujemo mehanizem za inicializacijo in brisanje objekta.
- Konstruktor je metoda, ki se pokliče ob kreiranju objekta in rezervira pomnilniški prostor ter inicializira podatke.
- Destruktor je metoda, ki se pokliče ob brisanju objekta in sprosti pomnilniški prostor.
- Življenjska doba objekta!

// Point.h

```
class Point {
private:
    int x, y;
public:
    Point();                  // default constructor
    Point(const Point& t);    // copy constructor
    Point(int xy);            // conversion constructor
    Point(int x, int y);      // other constructor
    ~Point();                 // destructor
    // methods
    int getX();
    int getY();
    void print();
    double distance(Point t);
};
```

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Konstruktorji in destruktorji

- Konstruktor in destruktor imata isto ime, kot je ime razreda.
- Destruktor ima pred imenom še znak ~ (tilda).
- Konstruktorji in destruktorji ne vračajo vrednosti in ne smejo imeti definiranega tipa metode (niti tipa **void**).
- Dinamično rezervirani pomnilniški prostor (**new**) moramo sprostiti sami (**delete**), ostali se sprosti avtomatsko.
- Po principu prekrivanja funkcij (*overloading*) lahko definiramo več konstruktorjev, a le en destruktor.

Konstruktorji in destruktorji

- Privzeti konstruktor je konstruktor brez argumentov.
- Kopirni konstruktor tvori objekt iz že obstoječega. Njegov argument je referenca na že obstoječ objekt tega razreda.
- Pretvorbeni konstruktor tvori nov objekt iz drugega podatkovnega tipa (razreda).
- Ostali konstruktorji imajo drugačne argumente in nimajo posebnega imena.
- Prevajalnik priskrbi privzeti* in kopirni konstruktor ter destruktor, če ga ne zapiše programer.

// Point.h

```
class Point {
private:
    int x, y;
public:
    Point();                // default constructor
    Point(const Point& t);  // copy constructor
    Point(int xy);          // conversion constructor
    Point(int x, int y);    // other constructor
    ~Point();               // destructor
    // methods
    int getX();
    int getY();
    void print();
    double distance(Point t);
};
```

Primer 2 – implementacija konstruktorjev

// Point.h

```
class Point {
private:
    int x, y;
public:
    Point();                // default constructor
    Point(const Point& t);  // copy constructor
    Point(int xy);          // conversion constructor
    Point(int x, int y);    // other constructor
    ~Point();               // destructor
    // methods
    int getX();
    int getY();
    void print();
    double distance(Point t);
};
```

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

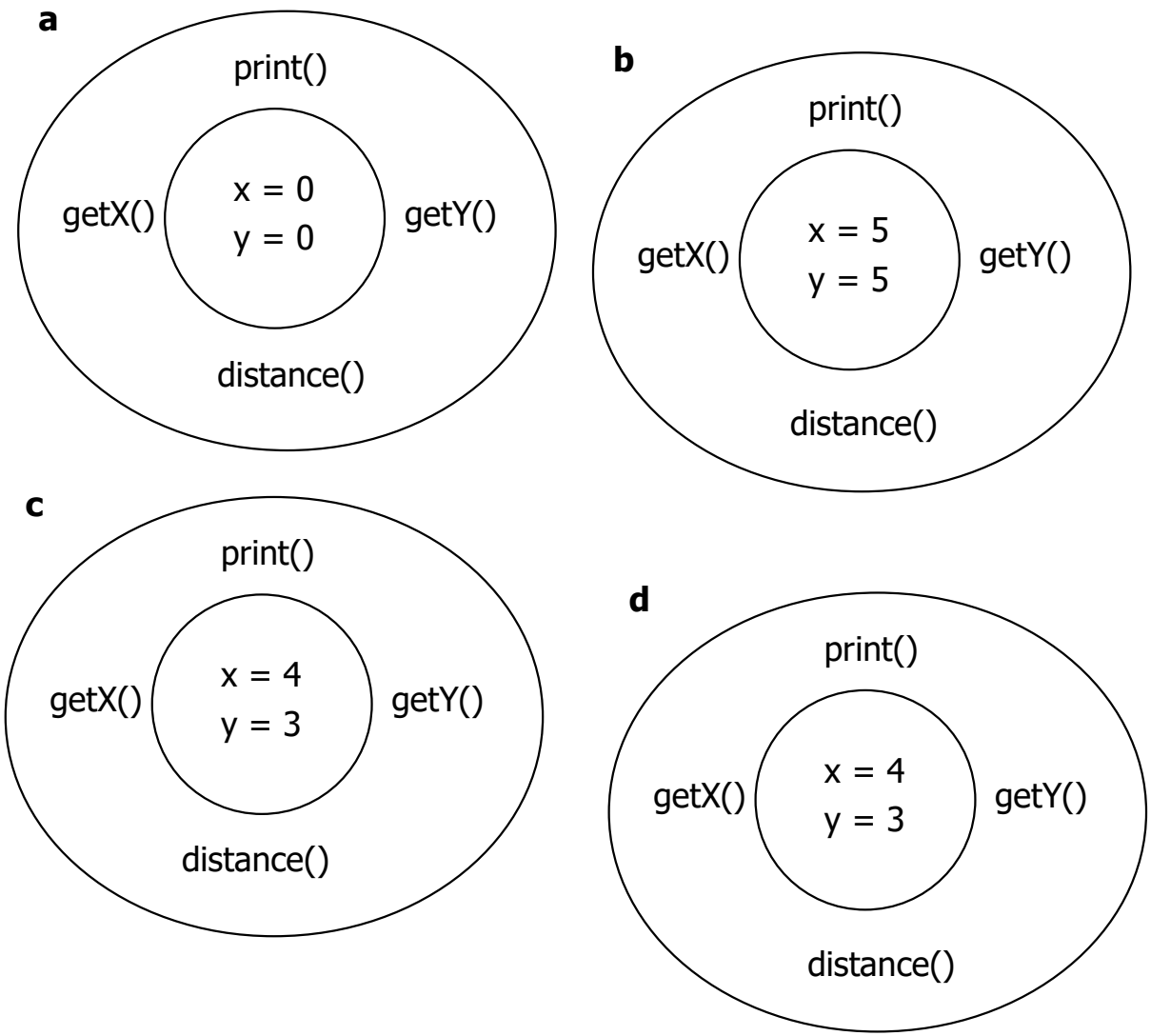
// Point.cpp

```
#include <iostream>
#include <cmath>
#include "Point.h"

Point::Point() : x(0), y(0) {
}
Point::Point(const Point& t) : x(t.x),  y(t.y) {
}
Point::Point(int xy) : x(xy), y(xy) {
}
Point::Point(int x, int y)  {
    this->x=x;
    this->y=y;
}
Point::~Point() {
}
int Point::getX() {
    return x;
}
int Point::getY() {
    return y;
}
void Point::print() {
    std::cout << "(" << x << ", " << y << ") " << std::endl;
}
double Point::distance(Point t) {
    return std::sqrt((double)(x - t.x)*(x - t.x)+(y - t.y)*(y -
t.y));
}
```

Primer 2 – implementacija konstruktorjev

// Example02.cpp

```
#include <iostream>
#include "Point.h"

int main() {
    Point a;
    Point b(5);
    Point c(4,3);
    Point d(c);
    a.print();
    b.print();
    c.print();
    d.print();
    std::cout << a.distance(c) << std::endl;
    //std::cout << a.distance(5) << std::endl;
    return 0;
}
```



Primer 2 – implementacija konstruktorjev

- Konstruktorji z inicializacijskim seznamom (*initialization list*)
- Za glavo metode zapišemo dvopičje in seznam instančnih spremenljivk, ki dobijo vrednosti, zapisane v oklepaju.
- Način zapisa z inicializacijskim seznamom je implementacijsko učinkovitejši!

// Point.cpp

…

```
Point::Point() : x(0), y(0) {
}
Point::Point(const Point& t) : x(t.x),  y(t.y) {
}
Point::Point(int xy) : x(xy), y(xy) {
}
Point::Point(int x, int y) : x(x), y(y) {
}
```

…

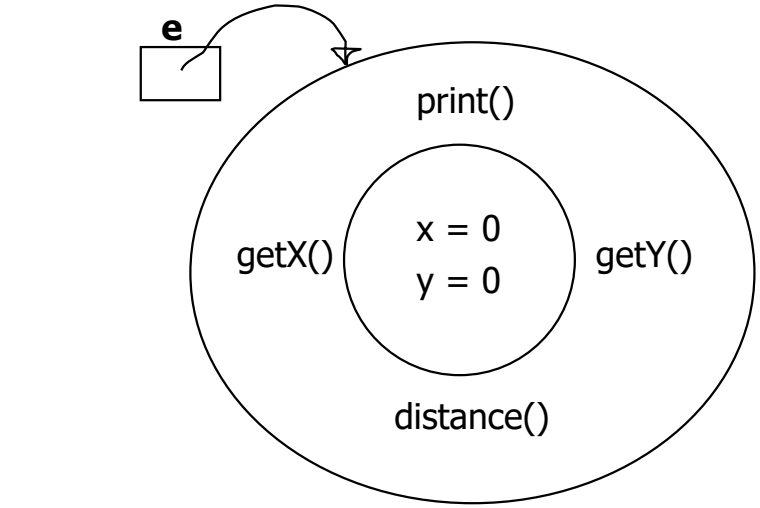
Primer 2 – implementacija konstruktorjev

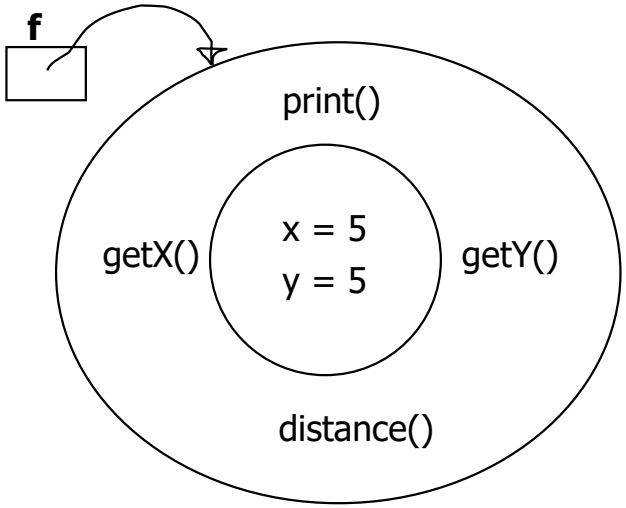
- Alokacije:
  - statična,
  - avtomatična,
  - dinamična.
- Dinamična alokacija objektov:
  - konstruktor se pokliče ob operatorju **new**
  - destruktor se pokliče ob operatorju **delete**

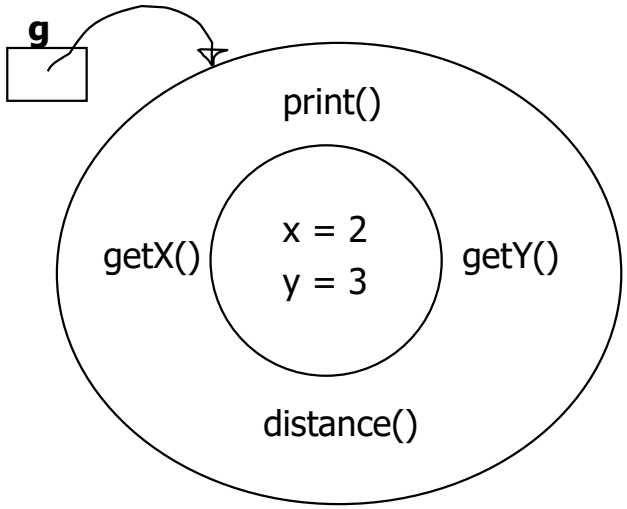
// Example02.cpp

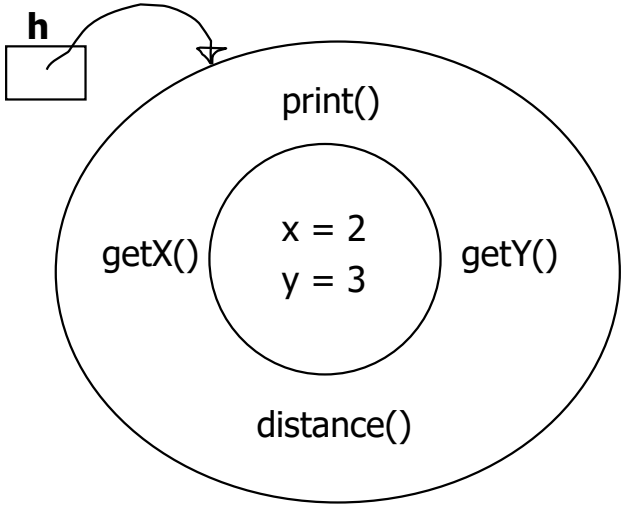
```
...

std::cout << "Dynamic allocation" << std::endl;
Point* e = new Point();
Point* f = new Point(5);
Point* g = new Point(2,3);
Point* h = new Point(*g);
e->print();
f->print();
g->print();
h->print();
delete e;
delete f;
delete g;
delete h;


...
```

Primer 2 – implementacija konstruktorjev









Pravila dobrega programiranja

- Dovolj kratke metode zapišemo znotraj definicije razreda.
- Za vidnost komponent vedno navedimo določili **public** in **private**.
- Zaradi večje preglednosti v razredu navedimo določili **public** in **private** le enkrat.
- **Če kopirni konstruktor in destruktor ne opravljata nobenega dodatnega dela ga naj priskrbi prevajalnik! JIH NE PIŠEMO SAMI**

Pogoste napake programerja

- Razred ali struktura se ne zaključi s podpičjem.
- Definicija izhodnega tipa za konstruktor ali destruktor.
- Vračanje vrednosti iz konstruktorja ali destruktorja s stavkom return.
- Ne definiramo privzetega konstruktorja, če smo definirali druge konstruktorje.
- Definicija destruktorja z argumenti.
- Doseganje privatnih komponent razreda.

Vprašanja

PROGRAMIRANJE II

Univerza v Mariboru

FERI
Fakulteta za elektrotehniko, računalništvo in informatiko

Primer 3 – Kompleksna števila

- Vmesnik razreda (*class interface*) predstavljajo vse javne komponente razreda.
- Uporabnik nekega razreda mora poznati le njegov vmesnik, podrobnosti o implementaciji pa zanj niso pomembne.
- Konstruktorji lahko imajo tudi privzete vrednosti.

// Complex.h

```
class Complex {
private:
    double real, imag;
public:
    Complex();  // default constructor
    // conversion constructor with default arguments
    Complex(double r, double i = 0);

    void print();
    Complex plus(Complex& c);
};
```

Primer 3 – Kompleksna števila

// Complex.cpp

```cpp
#include <iostream>
#include "Complex.h"

Complex::Complex() : real(0), imag(0) {
}

Complex::Complex(double r, double i): real(r), imag(i) {
}

void Complex::print() {
    std::cout << "(" << real << ", " << imag << "i)" << std::endl;
}

Complex Complex::plus(Complex& c) {
    Complex temp(real+c.real, imag+c.imag);
    return temp;
}
```

Primer 3 – Kompleksna števila

// Example03

```
#include <iostream>
#include "Complex.h"

int main() {
    Complex c1(1,1), c2(1), i(0,1);
    c1.print();
    c2.print();
    i.print();
    //c1.plus(5).print();
    c1.plus(i).print();

    std::cout << "Dynamic allocation" << std::endl;
    Complex* p_c = new Complex(4,2);
    p_c->print();
    c1.plus(*p_c).print();
    Complex c3(c1.plus(*p_c));
    c3.print();
    Complex* p_c1 = new Complex(c1.plus(i));
    p_c1->print();
    delete p_c;
    delete p_c1;

    return 0;
}
```

Konstantni objekti

- Konstanta spremenljivka (konstanta) const int i=1;
- Konstantni objekt – njegovega stanja ne moremo spreminjati!
- Konstantni objekt definiramo z določilom **const** pred definicijo objektne spremenljivke.
- Konstanti objekt lahko kliče samo konstantne metode. Takšne metode ne spreminjajo stanja objekta.
- Konstante metode definiramo, tako da pri definiciji metode za argumenti navedemo določilo **const**.

Konstantni objekti – primer 4

// Complex.h

```
class Complex {
private:
    double real, imag;
public:
    Complex();
    Complex(double r, double i = 0);

    void print() const;                    // constant method
    // argument is a constant object and method is constant
    Complex plus(const Complex& c) const;
    void add(double d);                // non-constant method
};
```

Konstantni objekti – primer 4

// Complex.cpp

```
#include <iostream>
#include "Complex.h"

Complex::Complex() : real(0), imag(0) {
}

Complex::Complex(double r, double i): real(r), imag(i) {
}

void Complex::print() const {
    std::cout << "(" << real << ", " << imag << "i)" << std::endl;
}

Complex Complex::plus(const Complex& c) const {
    Complex temp(real+c.real, imag+c.imag);
    return temp;
}

void Complex::add(double d) {
    real+=d;
}
```

Konstantni objekti – primer 4

// Example04

```
#include <iostream>
#include "Complex.h"

int main() {
    Complex c1(1,1);
    const Complex i(0,1);
    c1.add(10);
    //i.add(1);
    i.print();
    c1.print();
    std::cout << "---------------" << std::endl;
    i.plus(c1).print();
    c1.plus(i).print();
    return 0;
}
```

Razredne spremenljivke in metode

- Vsak objekt ima shranjene svojstvene, njemu lastne, podatke v instančnih spremenljivkah.
- Metode se vedno izvajajo nad podatki objekta, kateremu je bilo poslano sporočilo.
- Občasno je zaželjeno, da imamo podatek, ki je skupen vsem objektom nekega razreda.
- Imeti kopijo takšnega podatka v vsakem objektu ni dobra rešitev. Zakaj?

Razredne spremenljivke in metode

- Podatek, ki je skupen vsem objektom imenujemo statični podatek ali **razredna spremenljivka**.
- Metode, ki operirajo nad statičnimi podatki imenujemo statične oz. **razredne metode**.
- Razredne spremenljivke (statični podatki) so znani še preden ustvarimo objekte tega razreda.
- Razredne spremenljivke in metode določimo z določilom **static**.

Primer 5

// Complex.h

```
class Complex {
private:
    double real, imag;
    static int counter;  // class variable
public:
    Complex();  // default constructor
    Complex(double r, double i = 0);  // conversion constructor
    Complex(const Complex& c);  // copy constructor
    ~Complex();   // destructor
    void print() const;
    Complex plus(const Complex& c) const;
    static int getCounter() {   // class method
        return counter;
    }
};
```

Primer 5

// Complex.cpp

```
#include <iostream>
#include "Complex.h"

int Complex::counter=0; // class variables can't be initialized in class definition

Complex::Complex(): real(0), imag(0) {
    counter++;
}

Complex::Complex(double r, double i): real(r), imag(i) {
    counter++;
}

Complex::Complex(const Complex& c): real(c.real), imag(c.imag) {
    counter++;
}

Complex::~Complex() {
    counter--;
}

void Complex::print() const {
    std::cout << "(" << real << ", " << imag << "i)" << std::endl;
}

Complex Complex::plus(const Complex& c) const {
    Complex temp(real+c.real, imag+c.imag);
    return temp;
}
```

Primer 5

// Example05

```
#include <iostream>
#include "Complex.h"

int main() {
    std::cout << "Current number of objects: " << Complex::getCounter() << std::endl;
    Complex c1(1,1);
    const Complex i(0,1);     // constant object
    c1.print();
    i.print();
    std::cout << "Current number of objects: " << c1.getCounter() << std::endl;
    std::cout << "Current number of objects: " << i.getCounter() << std::endl;

    Complex* p_c1 = new Complex(c1.plus(i));
    Complex* p_c2 = new Complex();
    p_c1->print();
    p_c2->print();
    std::cout << "Current number of objects: " << p_c1->getCounter() << std::endl;
    delete p_c1;
    delete p_c2;
    std::cout << "Current number of objects: " << c1.getCounter() << std::endl;

    return 0;
}
```

Kazalec this

- Razlika med funkcijo in metodo
  - plus(c1, i) vs. c1.plus(i)
- Metoda ima implicitni parameter, to je kazalec na objekt, ki je prejel sporočilo.
- Ta kazalec v programu dosegamo s rezervirano besedo **this**, ki je konstantni kazalec.

Kazalec this

- Primer

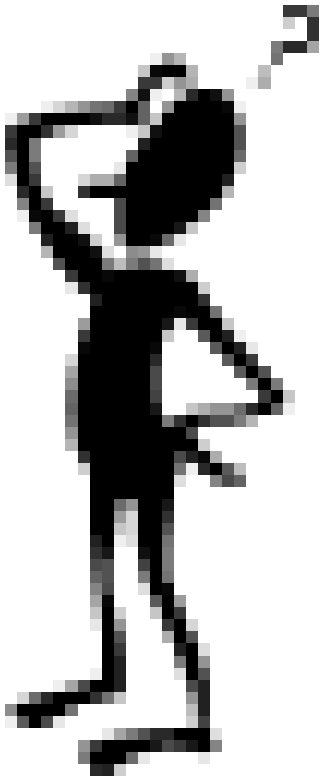
```
void Complex::print() const {
    std::cout << "Object address: " << this << "=(" << this->real << ", " << this->imag << "i)" <<
       this->counter << std::endl;
}
```

Razlika med konstantnim kazalcem (int* const) in kazalcem na konstanto (const int*).

Kazalec this

- Kazalec **this** je konstantni kazalec in ga lahko uporabljamo samo znotraj nerazrednih (nestatičnih) metod razreda.
- Zunaj metod in v razrednih (statičnih) metodah kazalec **this** ne obstaja.

Vprašanja



PROGRAMIRANJE II

Objekti kot elementi razredov

- Podatki (instančne spremenljivke) so lahko poljubnega tipa, torej tudi razreda.
- V tem primeru govorimo o vsebovanju oz. agregaciji/kompoziciji (*aggregation*, *composition*).

Primer 6

// Point.h

```
class Point {
private:
    int x, y;
public:
    Point();  // default constructor
    Point (int x, int y);  // constructor

    int getX() const;
    int getY() const;
    void print() const;
    double distance(const Point& p) const;
};
```

// Point.cpp

```
#include <iostream>
#include <cmath>
#include "Point.h"

Point::Point() : x(0), y(0) {
}

Point::Point (int x, int y) : x(x), y(y) {
}

int Point::getX() const {
    return x;
}

int Point::getY() const {
    return y;
}

void Point::print() const {
    std::cout << "(" << x << ", " << y << ") " << std::endl;
}

double Point::distance(const Point& p) const {
    return std::sqrt((double)(x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
}
```

Primer 6

// CPoint.h

```
#include "Point.h"

class CPoint {
private:
    Point p;    // composition
    int color;
public:
    CPoint();
    CPoint(int x, int y, int c);
    void print() const;
};
```

// CPoint.cpp

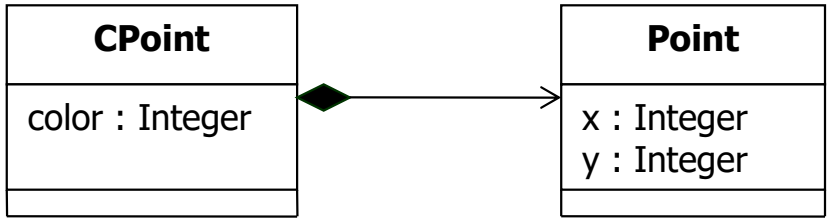
```
#include <iostream>
#include "CPoint.h"

CPoint::CPoint() : p(), color(0) {
}

CPoint::CPoint(int x, int y, int c) : p(x, y), color(c) {
}
void CPoint::print() const {
    std::cout  << "("  << p.getX() << ", " << p.getY() <<
                  ", color= " << color << ")" << std::endl;
}
```

Primer 6

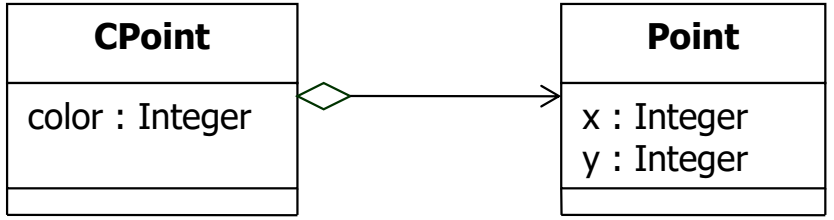
// Example06

```
int main() {
    Point p1(3,4);
    Point p2;
    CPoint c1;
    CPoint c2(1,2,3);
    CPoint c3(c2);

    p1.print();
    c1.print();
    c2.print();
    c3.print();
    std::cout << p1.distance(p2) << std::endl;
    //std::cout << p1.distance(c2) << std::endl;
    //std::cout << c2.distance(p1) << std::endl;
    return 0;
}
```

UML notacija (kompozicija)



UML notacija (agregacija)



Dedovanje

- Najpomembnejši koncept OOP je dedovanje.
- Dedovanje omogoča inkrementalni razvoj programov, kjer pri razvoju programske opreme iz nadrazredov podedujemo lastnosti, tj. strukturo in obnašanje.
- Programer zapiše le specifične lastnosti, ki se razlikujejo od tistih v nadrazredih.
- Dedovanje tako med razrede uvede tranzitivno relacijo in jih uredi v hierarhijo glede na njihove splošne in specifične lastnosti.

Dedovanje

- Dedovanje uvedemo z namenom enostavnejše konceptualne specializacije.
- Konceptualno modeliranje je proces organiziranja znanja o neki aplikacijski domeni v hierarhični red z namenom, da dobimo natančnejšo sliko o problemu oziroma da problem bolje razumemo.
- Dedovanje lahko uporabimo tudi za ponovno uporabo kode*.

Dedovanje - pojmi

- **class** B : **public** A { … };
- Nadrazred (*superclass*) ali bazni razred (*base class*)
- Podrazred (*subclass*) ali izpeljani razred (*derived class*)
- Izpeljava (*derivation*)
- Enkratno (*single*) in večkratno dedovanje (*multiple inheritance*)
- Posplošitev ali generalizacija (*generalization*)
- Specializacija (*specialization*)
- Relacija "is-a" (dedovanje) in "has-a" (vsebovanje: agregacija, kompozicija)

Dedovanje – Osnovno načelo

- Kjerkoli pričakujemo objekt nadrazreda lahko varno uporabimo objekt podrazreda. Saj ima takšen objekt vse lastnosti nadrazreda in dodatne specifične lastnosti.
- Primeri nadrazredov in podrazredov:

Nadrazred	Podrazred
Oseba	Student
Student	PodiplomskiStudent
Lik	Trikotnik
Štirikotnik	Pravokotnik

Primer 7

//Point.h

```
class Point {
private:
    int x, y;
public:
    Point();
    Point (int x1, int y1);

    int getX() const;
    int getY() const;
    void print() const;
    double distance(const Point& p) const;
};
```

// Point.cpp

```
#include <iostream>
#include <cmath>
#include "Point.h"

Point::Point() : x(0), y(0) {
}

Point::Point (int x, int y) : x(x), y(y) {
}

int Point::getX() const {
    return x;
}

int Point::getY() const {
    return y;
}

void Point::print() const {
    std::cout << "(" << x << ", " << y << ") " << std::endl;
}

double Point::distance(const Point& p) const {
    return std::sqrt((double)(x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
}
```

Primer 7

//CPoint.h

```
#include "Point.h"

class CPoint : public Point {
private:
    int color;
public:
    CPoint();
    CPoint(int x, int y, int c);

    void print() const;
};
```

// CPoint.cpp

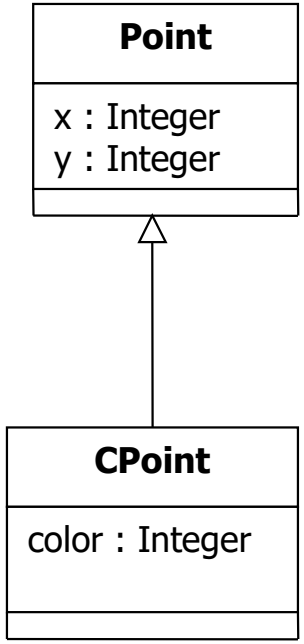
```
#include <iostream>
#include "CPoint.h"

CPoint::CPoint() : Point(), color(0) {
}

CPoint::CPoint(int x, int y, int c) : Point(x, y), color(c) {
}

void CPoint::print() const {
    std::cout << "(" << getX() << ", " << getY() <<
                 ") color=" << color << std::endl;
}
```

Primer 7

// Example07

```
#include <iostream>
#include "CPoint.h"

int main() {
    Point p1(3,4);
    Point p2;
    CPoint c1;
    CPoint c2(1,2,3);
    CPoint c3(c2);

    p1.print();
    c1.print();
    c2.print();
    c3.print();

    std::cout << "-------------" << std::endl;
    std::cout << p1.distance(p2) << std::endl;
    std::cout << p1.distance(c2) << std::endl;
    std::cout << c1.distance(p1) << std::endl;
    std::cout << c2.distance(c1) << std::endl;
    return 0;
}
```

UML notacija



Določilo protected

- Ograjevanje oz. skrivanje elementov razreda (instančnih spremenljivk in metod) implementiramo z določili:
  - private (privatni)
  - public (javni)
  - protected (zaščiteni)
- Zaščiteni elementi (protected) posameznega razreda so dosegljivi tudi v metodah izpeljanih razredov tega razreda.

Virtualne metode - Primer 8

//Point.h

```
class Point {
protected:
    int x, y;
public:
    Point();
    Point(int x, int y);
    virtual ~Point();  // virtual destructor

    int getX() const;
    int getY() const;
    virtual void print() const;
    double distance(const Point& p) const;
};
```

//Point.cpp

```
#include <iostream>
#include <cmath>
#include "Point.h"

Point::Point() : x(0), y(0) {
}

Point::Point (int x, int y) : x(x), y(y) {
}

Point::~Point() {
     std::cout << "destructor Point" << std::endl;
}

int Point::getX() const {
    return x;
}

int Point::getY() const {
    return y;
}

void Point::print() const {
    std::cout << "(" << x << ", " << y << ") " << std::endl;
}

double Point::distance(const Point& p) const {
    return std::sqrt((double)(x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
}
```

Virtualne metode - Primer 8

//CPoint.h

```
#include "Point.h"

class CPoint : public Point {
protected:
    int color;
public:
    CPoint();
    CPoint(int x, int y, int c);
    ~CPoint();
    void print() const;
};
```

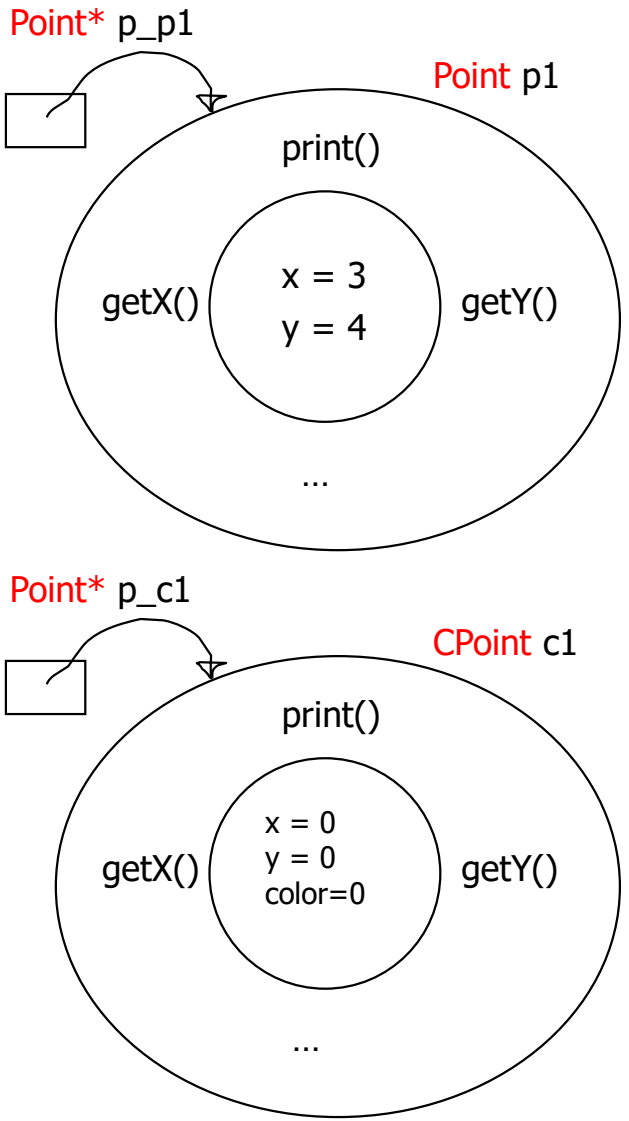
//CPoint.cpp

```
#include <iostream>
#include "CPoint.h"

CPoint::CPoint() : Point(), color(0) {
}

CPoint::CPoint(int x, int y, int c) : Point(x, y), color(c) {
}

CPoint::~CPoint() {
    std::cout << "Destructor CPoint" << std::endl;
}

void CPoint::print() const {
    std::cout <<  "(" << x << ", " << y << ") color= " << color << std::endl;

}
```

Virtualne metode

//Example08

```
#include <iostream>
#include "CPoint.h"

int main() {
    Point p1(3,4);
    CPoint c1;
    p1.print();
    c1.print();
    Point* p_p1 = &p1;
    Point* p_c1 = &c1;
    p_p1->print();
    p_c1->print();
    delete p_p1;
    delete p_c1;
    return 0;
}
```



Rezultat brez virtualnih metod:
(3, 4)
(0, 0) color= 0
(3, 4)
(0, 0) Zakaj?

Virtualne metode

- Tip p_c1 je Point* (statični tip)
- Tip c1 je CPoint (dinamični tip)
- Pri klicu metode je merodajen statični tip!
- Pri virtualnih metodah (določilo **virtual**) je merodajen dinamični tip!
- Virtualne metode imenujemo tudi polimorfne metode, za katere velja, da se na isto sporočilo objekti odzovejo na njim lasten način.

Virtualne metode

- Kadar je kakšna metoda v hierarhiji definirana kot virtualna, so vse metode, ki so ponovno definirane v podrazredih (polimorfna redefinicija), prav tako virtualne.
- Pri polimorfni redefiniciji se redefinirana metoda mora ujemati z virtualno metodo v signaturi (ime metode, število in tipi argumentov) ter tip rezultata metode*.
- Virtualne metode v podrazredih ni potrebno definirati ponovno. Če podrazred nima definirane virtualne metode, se kliče ustrezna metoda nadrazreda.

Pravila dobrega programiranja

- Vse metode, ki ne spreminjajo stanja objekta, definirajmo kot konstantne metode.
- V razredu, ki ima kako virtualno metodo, definirajmo tudi virtualni destruktor.

Pogoste napake programerja

- Klicanje metode, ki ni konstantna, s konstantnim objektom.
- Spreminjanje podatkov objekta v konstantni metodi.
- Uporaba kazalca **this** in nestatičnih podatkov v razrednih (statičnih) metodah razreda.

Pogoste napake programerja

- Obravnava objektov nadrazreda, kot da so objekti podrazreda.
- Pri redefiniciji metode nadrazreda v podrazredu je v navadi, da v njej pokličemo metodo nadrazreda in nato opravimo še nekaj dodatnega dela. Napačno je, da metoda v podrazredu kliče samo sebe, če tega ne želimo.

Pogoste napake programerja

- Ponovno definirana virtualna metoda v izpeljanem razredu nima istega izhodnega tipa in istih argumentov kot v nadrazredu.

Vprašanja

PROGRAMIRANJE II

Abstraktni razredi

- Pomembna aktivnost pri načrtovanju programa je poiskati potrebne razrede za dani problem. Ko razrede identificiramo jih poskušamo uvrstiti v hierarhijo (relacija nadrazred – podrazred)
- Nadaljnji korak je poiskati še neidentificirane razrede, ki so posplošitev (generalizacija) danih razredov.

Abstraktni razredi

- Postopek generalizacije nas lahko pripelje do takšnih razredov za katere vemo, da njihovih objektov ne bomo kreirali.
- Razred želimo imeti v hierarhiji samo zato, ker so posplošitev že znanih razredov in zaradi uporabe polimorfnih (virtualnih) metod.
- Vsem objektom bomo poslali isto sporočilo, objekti pa se bodo odzvali na njim svojstven način.

Abstraktni razredi

- Takšnim razredom pravimo **abstraktni razredi** (*abstract classes*)
- Ostalim razredom pravimo tudi **konkretni razredi** (*concrete classes*)
- Primeri abstraktni razredov:
  - Žival – Pes, Muca, …
  - Lik – Krog, Kvadrat, …
  - Vozilo – Letalo, Vlak, Kolo, …

Abstraktni razredi

- Mnogo metod v abstraktnih razredih ne znamo implementirati (npr. kako definirati ploščino Lika).
- Takšne metode imenujemo abstraktne metode ali tudi čisto virtualne (*pure virtual*).

Abstraktni razredi

- Abstraktne metode zato nimajo implementacije telesa metode.
- To zapišemo v C++ kot

```
tip ImeFun(Args) = 0;
```

- Prav tako ne moremo ustvarjati objektov abstraktnih razredov, saj se takšni objekti ne bi znali odzivati na sporočila.

Primer 9

// Animal.h

```
class Animal {   // abstract class
public:
    virtual ~Animal() {}
    virtual void voice() const = 0;   // abstract method
};
```

// Cat.h

```
#include <iostream>

class Cat : public Animal {
public:
    void voice() const {
        std::cout << "meow" << std::endl;
    }
};
```

// Dog.h

```
#include <iostream>

class Dog : public Animal {
public:
    void voice() const {
        std::cout << "bark" << std::endl;
    }
};
```

// Cow.h

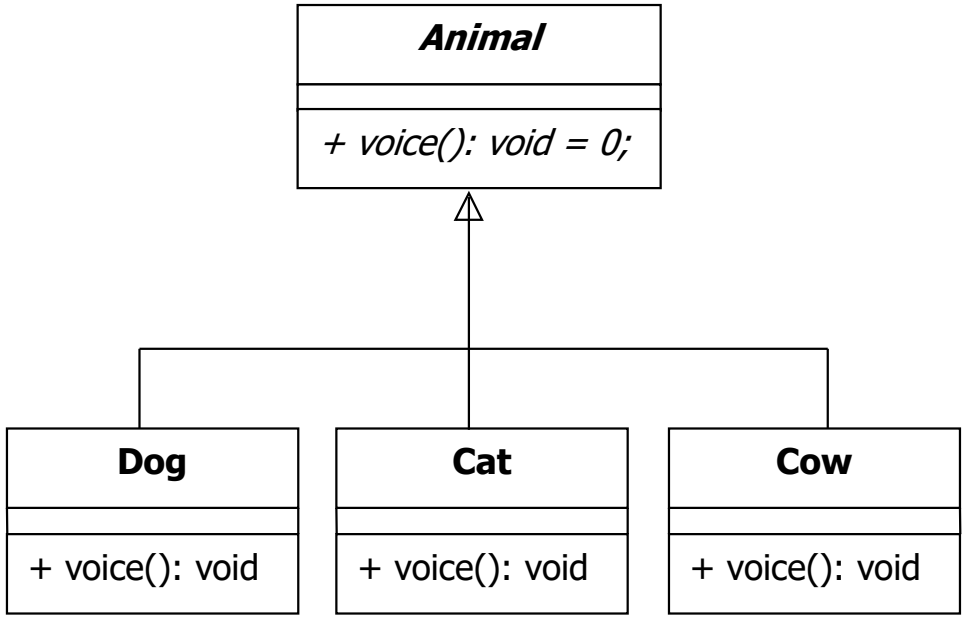
```
#include <iostream>

class Cow : public Animal {
public:
    void voice() const {
        std::cout << "moo" << std::endl;
    }
};
```

Primer 9

// Example09.cpp

```
#include <iostream>
#include "Animal.h"
#include "Dog.h"
#include "Cat.h"
#include "Cow.h"

int main() {
    Animal* zoo[4];

    //zoo[0] = new Animal;
    zoo[0] = new Dog;
    zoo[1] = new Cat;
    zoo[2] = new Dog;
    zoo[3] = new Cow;
    for (int i=0; i < 4; i++)
      zoo[i]->voice();

    for (int i=0; i<4; i++)
        delete zoo[i];
   return 0;
}
```

UML notacija



Primer 10 (agregacija)

//Employee.h

```
#include <iostream>

class Employee {
private:
    std::string name;
public:
    Employee(std::string name) : name(name) {
        std::cout << "Employee::constructor" << std::endl;
    }
    virtual ~Employee() {
        std::cout << "Employee::destructor" << std::endl;
    }
    virtual const std::string& toString() const {
        return name;
    };
};
```

Primer 10 (agregacija)

//Company.h

```
#include <iostream>
#include "Employee.h"

class Company {
private:
    std::string name;
    Employee* ptrEmployee;  // aggregation
public:
    Company(std::string cname, Employee* ename) : name(cname), ptrEmployee(ename) {
        std::cout << "Company::constructor" << std::endl;
    }
    virtual ~Company() {
        std::cout << "Company::destructor" << std::endl;
    }
    virtual const std::string& toString() const {
        return name;
    };
    virtual void employed() const {
        std::cout << ptrEmployee->toString();
    }
};
```

Primer 10 (agregacija)

//Example10.cpp

```
#include <iostream>
#include "Employee.h"
#include "Company.h"

int main() {
    std::cout << "Aggregation example" << std::endl;
    {
        std::cout << "----------- nested block 1 begin -----------" << std::endl;
        Employee* ptrEmp = new Employee("John");
        {
            std::cout << "----------- nested block 2 begin -----------" << std::endl;
            Company c("SmartCo", ptrEmp);
            std::cout << "Employee ";
            c.employed();
            std::cout << " works for company " << c.toString() << std::endl;
            std::cout << "----------- nested block 2 end -----------" << std::endl;
        }
        std::cout << "Company doesn't exist anymore" << std::endl;
        std::cout << "But, employee " << ptrEmp->toString();
        std::cout << " still exists!" << std::endl;
        std::cout << "----------- nested block 1 end -----------" << std::endl;
        delete ptrEmp;
    }
    return 0;
}
```

Primer 11 (kompozicija)

//Room.h

```
#include <iostream>

class Room {
private:
    std::string name;
public:
    Room(std::string name) : name(name) {
        std::cout << "Room::constructor" << std::endl;
    }
    virtual ~Room() {
        std::cout << "Room::destructor" << std::endl;
    }
    virtual void print() const {
        std::cout << "Room " << name << std::endl;
    }
};
```

Primer 11 (kompozicija)

//House.h

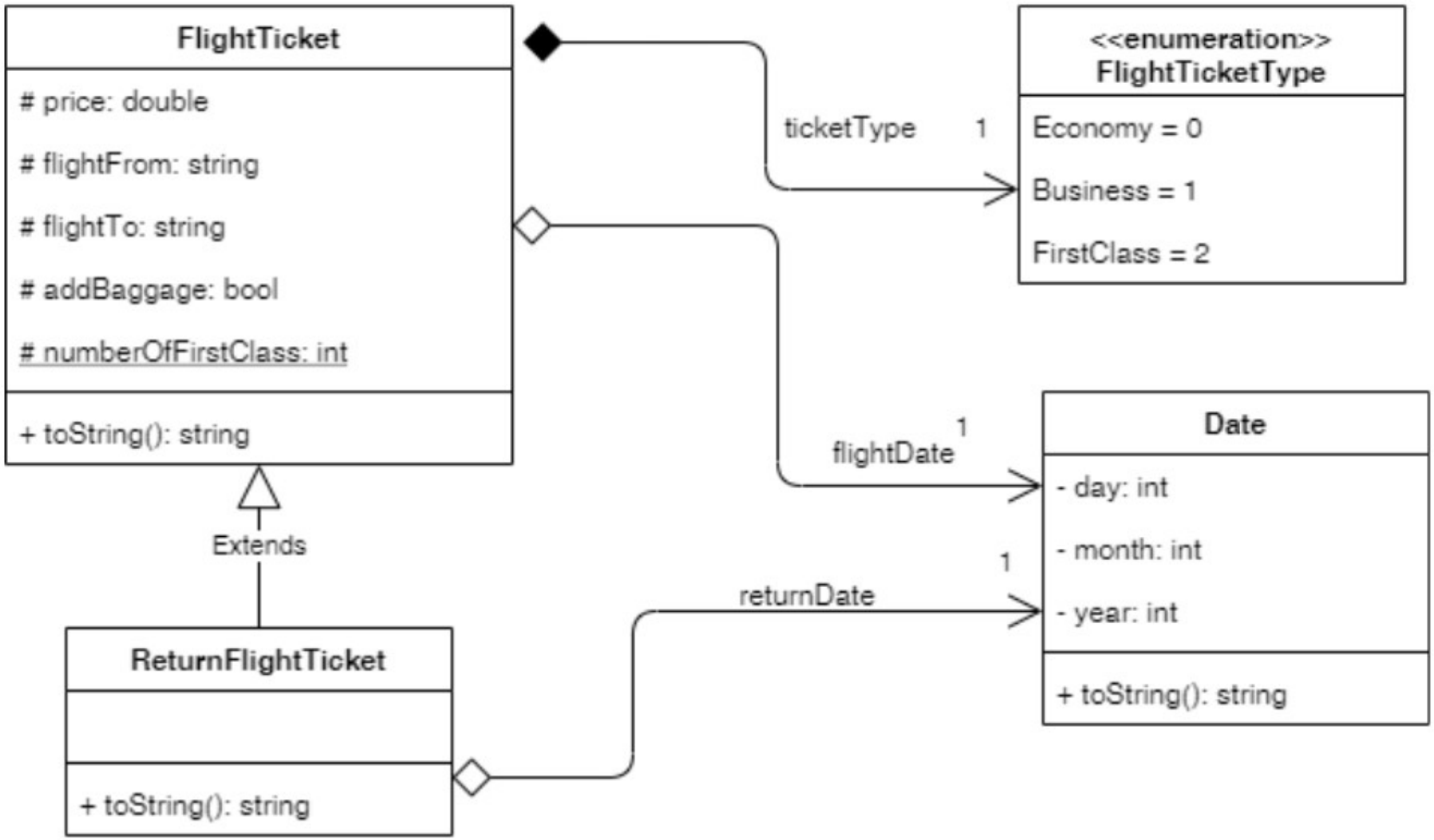
```
#include <iostream>
#include "Room.h"

class House {
private:
    std::string name;
    Room kitchen, livingRoom, bedroom;
public:
    House(std::string name) : name(name), kitchen("Kitchen1"), livingRoom("living room 1"),
                              bedroom("TV bedroom") {
        std::cout << "House::constructor" << std::endl;
    }
    virtual ~House() {
        std::cout << "House::destructor" << std::endl;
    }
    virtual void print() const {
        std::cout << "House name: " << name << std::endl;
        kitchen.print();
        livingRoom.print();
        bedroom.print();
    }
};
```

Primer 11 (kompozicija)

//Example11.cpp

```
int main() {
    std::cout << "Composition example" << std::endl;
    {
        std::cout << "----------- nested block begin -----------" << std::endl;
        House h1("My home");
        h1.print();
        std::cout << "----------- nested block end -----------" << std::endl;
    }
    std::cout<<"House and rooms don't exist anymore!" << std::endl;
    return 0;
}
```

Primer 12 (dedovanje in vsebovanje)



Primer 12 (dedovanje in vsebovanje)

Naštevni tip enum in enum class

Nekateri programski jeziki omogočajo uporabnikom definirati popolnoma nov osnovni tip z naštevanjem vrednosti. Takšen osnovni tip imenujemo naštevni tip ("enumeration type") in posamezno vrednost enumerand ("enumerand").

V C++ definiramo naštevni tip z naštevanjem identifikatorjev, ki predstavljajo njihove vrednosti.

```
enum BarvaKarte {srce, karo, pik, kriz};
```

Enumerandom pa lahko priredimo celoštevilčne vrednosti:

```
enum StevEnum { ena = 1, dva = 2, tri = 3, stiri = 4, pet = 5,
                sest = 6, sedem = 7, osem = 8, devet = 9, deset = 10 };
```

Primer 12 (dedovanje in vsebovanje)

```
struct Karta {
        BarvaKarte barva;
        StevEnum stev;
};
Karta mojaKarta;
mojaKarta.barva = karo;
mojaKarta.stev = dva;
cout << (mojaKarta.barva+1 == mojaKarta.stev) << endl;
```

C++11 uvede **enum class**, kjer

- ni dovoljena implicitna pretvorba v int
- ni možna primerjava med različnimi naštevnimi tipi

Primer 12 (dedovanje in vsebovanje)

//Example12.cpp

```
#include <iostream>
#include "FlightTicket.h"
#include "ReturnFlightTicket.h"

int main() {
    Date* ptrDate1 = new Date(17, 3, 2020);
    Date* ptrDate2 = new Date(18, 3, 2020);
    Date* ptrDate3 = new Date(19, 3, 2020);
    Date* ptrDate4 = new Date(29, 3, 2020);
    Date* ptrDate5 = new Date(31, 3, 2020);
    FlightTicket* ptrFlights[3];
    ptrFlights[0] = new FlightTicket(400, "Moskva", "Ljubljana", false, FlightTicketType::Economy, ptrDate1);
    ptrFlights[1] = new ReturnFlightTicket(1000, "Hong Kong", "Washington", true,
                                           FlightTicketType::FirstClass, ptrDate2, ptrDate4);
    ptrFlights[2] = new ReturnFlightTicket(800, "Vienna", "Chicago",
                                           false, FlightTicketType::Economy, ptrDate3, ptrDate5);
    for (int i=0; i<3; i++)
        std::cout << ptrFlights[i]->toString() << std::endl;

    delete ptrDate1;
    delete ptrDate2;
    delete ptrDate3;
    delete ptrDate4;
    delete ptrDate5;
    for (int i=0; i<3; i++)
        delete ptrFlights[i];
    return 0;
}
```

Vprašanja

PROGRAMIRANJE II

Kazalci - ponovitev

- Kazalci oz. reference so abstrakcija pomnilniških naslovov. Kazalec oz. referenca je pomnilniška lokacija, v kateri je zapisan naslov pomnilniške lokacije, kjer je shranjena kaka vrednost. Vrednost kazalca (reference) je torej pomnilniški naslov.
- Na naslovu, kamor kaže kazalec (referenca), pa je lahko vrednost poljubnega tipa, tako osnovnega kot sestavljenega. Tako imamo opravka tudi s kazalci na polja, kazalci na strukture, kazalci na objekte, kazalci na funkcije in tudi s kazalci na kazalce.

Kazalci - ponovitev

- Programski jeziki običajno definirajo posebno vrednost (NULL, nullptr), ki pove, da kazalec ne kaže na nobeno vrednost.

  NULL – vrednost 0, ki je tipa int

  nullptr – rezervirana beseda, ki predstavlja naslov 0

- Najpomembnejša operacija nad kazalci je operacija dostopa do vrednosti, na katero kazalec kaže oz. dereferenciranje (“dereferencing”).

Kazalci - ponovitev

- Razlika med kazalcem in referenco je majhna, vendar zelo pomembna. S pomočjo kazalcev dejansko upravljamo s pomnilniškimi lokacijami in pri tem delu moramo biti zelo pazljivi. Zato določenih operacij kot npr. kazalčna aritmetika, ki jih lahko izvajamo nad kazalcem, nad referenco ne dovolimo. Nadalje, določene operacije kot npr. dereferenciranje se nad referenco izvedejo vedno (implicitno), medtem kot nad kazalcem ne. Zato tudi pravimo, da so reference varni kazalci ("safe pointers"), saj ne omogočajo kazalčne aritmetike.
- Tipičen programski jezik, ki pozna reference, je programski jezik java, ki kazalcev sploh ne omogoča.

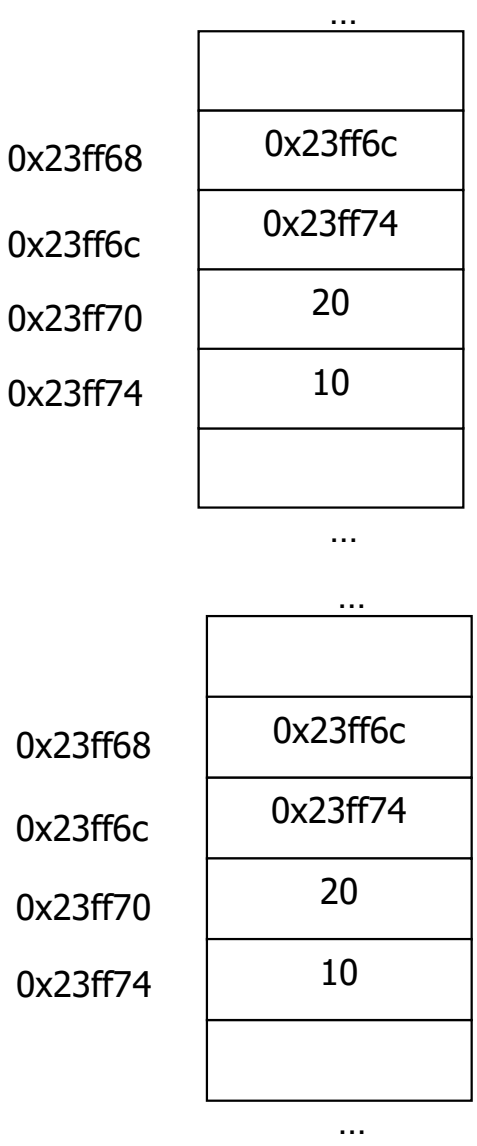
Kazalci - ponovitev

- V programskem jeziku C++ je razlika med kazalcem in referenco še manjša. Referenca je samo konstanten kazalec (vedno kaže na isto lokacijo), pri kateri se derefenciranje izvede implicitno.
- Prav tako v jeziku C++ reference niso vrednosti prvega razreda, saj ne moremo ustvariti polja referenc, kazalce na reference in reference na tip **void** (**void**&).

Kazalci - ponovitev

// Example13

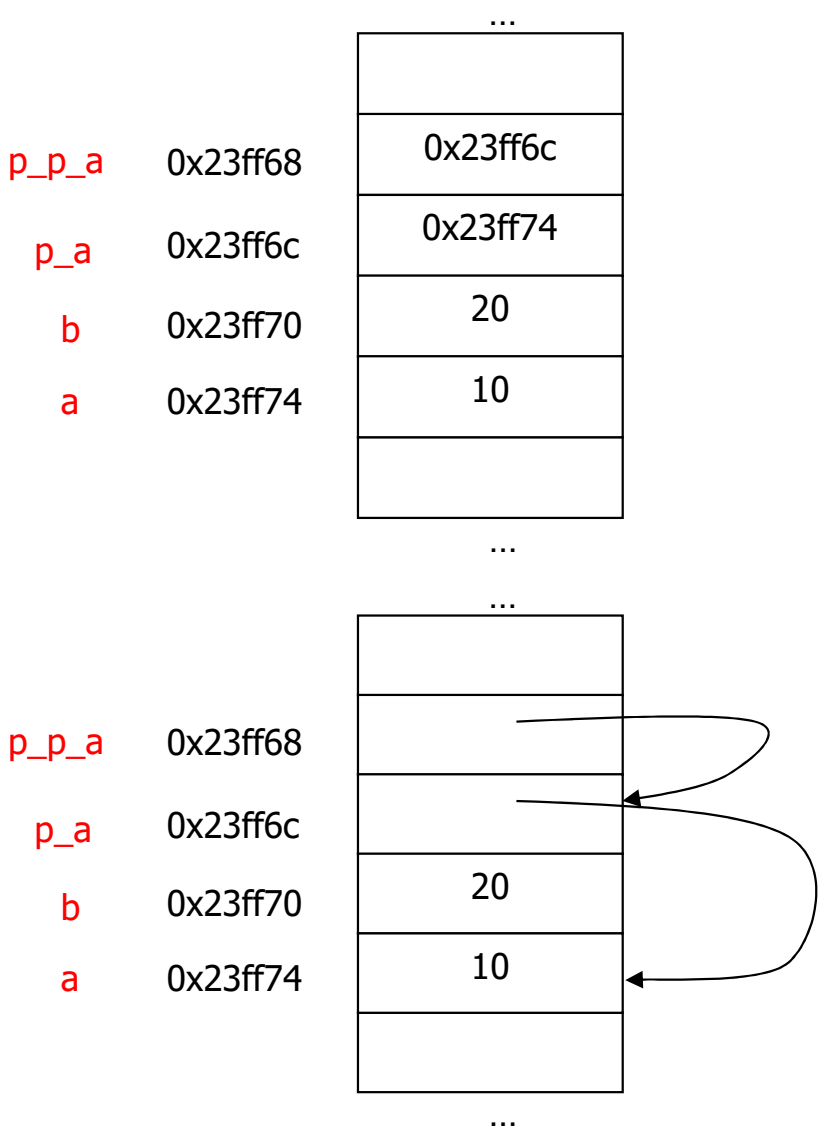
```
#include <iostream>

int main() {
    int a = 10;
    int b = 20;
    int* p_a = &a;
    int** p_p_a = &p_a;
    std::cout << "a " << &a << " " << a << std::endl;
    std::cout << "b " << &b << " " << b << std::endl;
    std::cout << "p_a " << &p_a << " " << p_a << " " <<
                           *p_a << std::endl;
    std::cout << "p_p_a " << &p_p_a << " " << p_p_a <<
                  *p_p_a  << " " << **p_p_a << std::endl;
    return 0;
}
```



Kazalci - ponovitev

// Example13

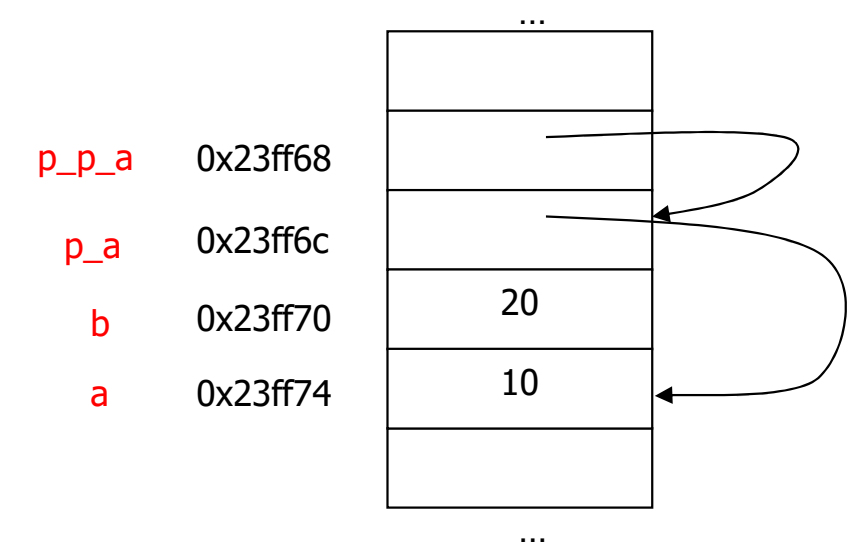
```
#include <iostream>

int main() {
    int a = 10;
    int b = 20;
    int* p_a = &a;
    int** p_p_a = &p_a;
      …
    return 0;
}
```

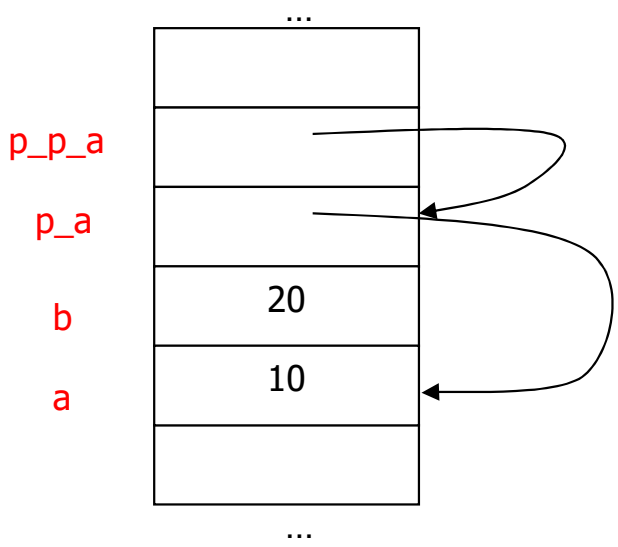


Kazalci - ponovitev

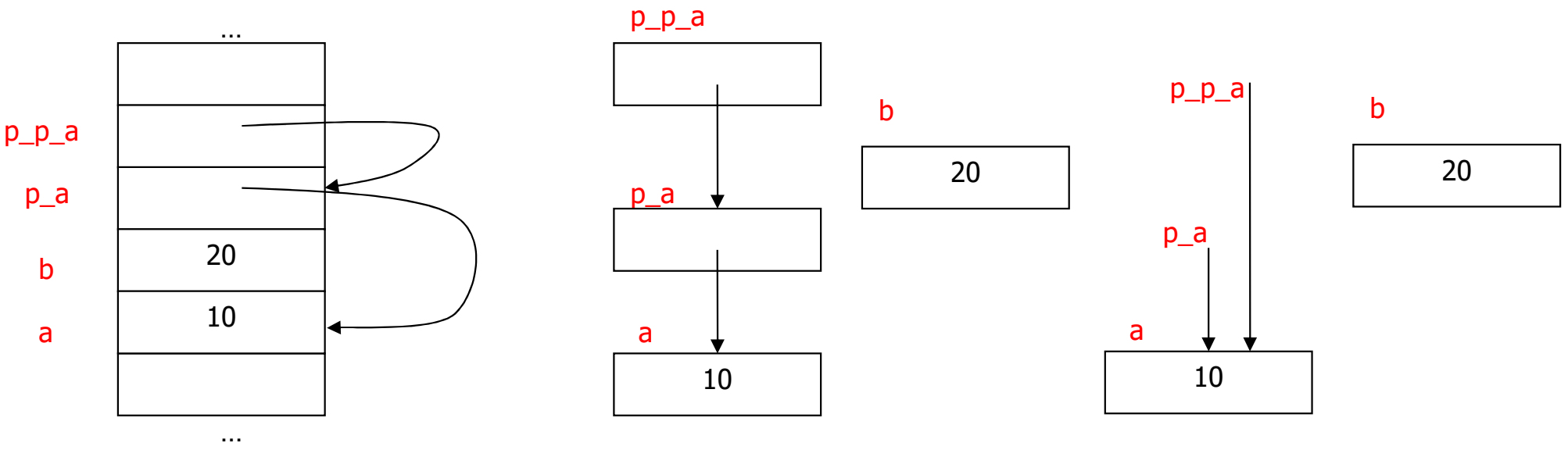
// Example13

```
#include <iostream>

int main() {
    int a = 10;
    int b = 20;
    int* p_a = &a;
    int** p_p_a = &p_a;
      …
    return 0;
}
```





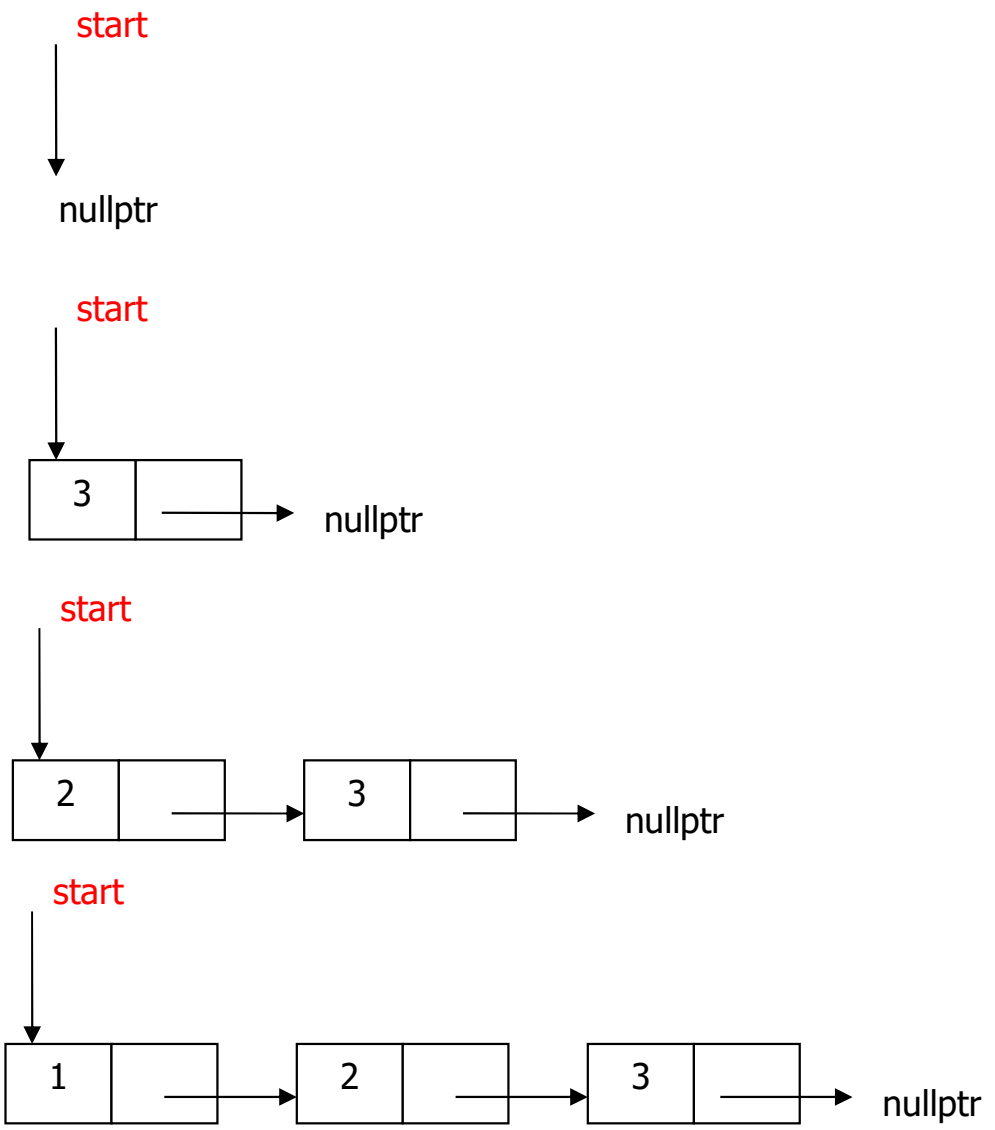
Kazalci - ponovitev



Kazalci - ponovitev

// Example13

```
struct Node {
    int el;
    Node* ptrNext;
};

//void insertAtBeginning(Node* start, int n) {
void insertAtBeginning(Node*& start, int n) {
    Node* ptrTemp = new Node;
    ptrTemp->el = n;
    ptrTemp->ptrNext = start;
    start=ptrTemp;
}

...

//The nullptr keyword represents a null pointer value.
Node* ptrStart = nullptr;
insertAtBeginning(ptrStart, 3);
insertAtBeginning(ptrStart, 2);
insertAtBeginning(ptrStart, 1);
```



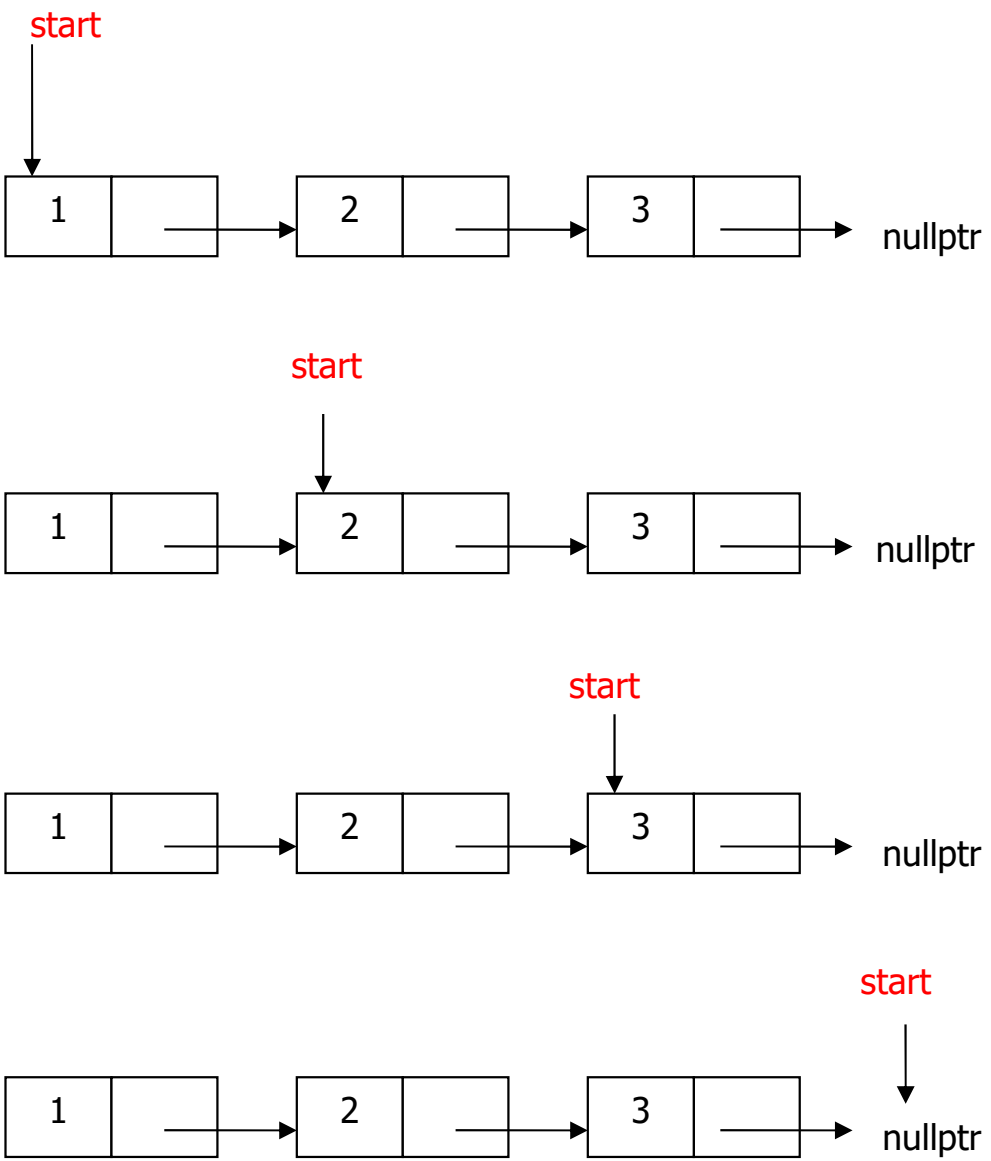
Kazalci - ponovitev

// Example13

```
//void print(Node*& start) {
void print(Node* start) {
    while (start) {
        std::cout << start-> el << " ";
        start=start->ptrNext;
    }
    std::cout << std::endl;
}
```

...

```
Node* ptrStart = nullptr;
insertAtBeginning(ptrStart, 3);
insertAtBeginning(ptrStart, 2);
insertAtBeginning(ptrStart, 1);
print(ptrStart);
print(ptrStart);
```



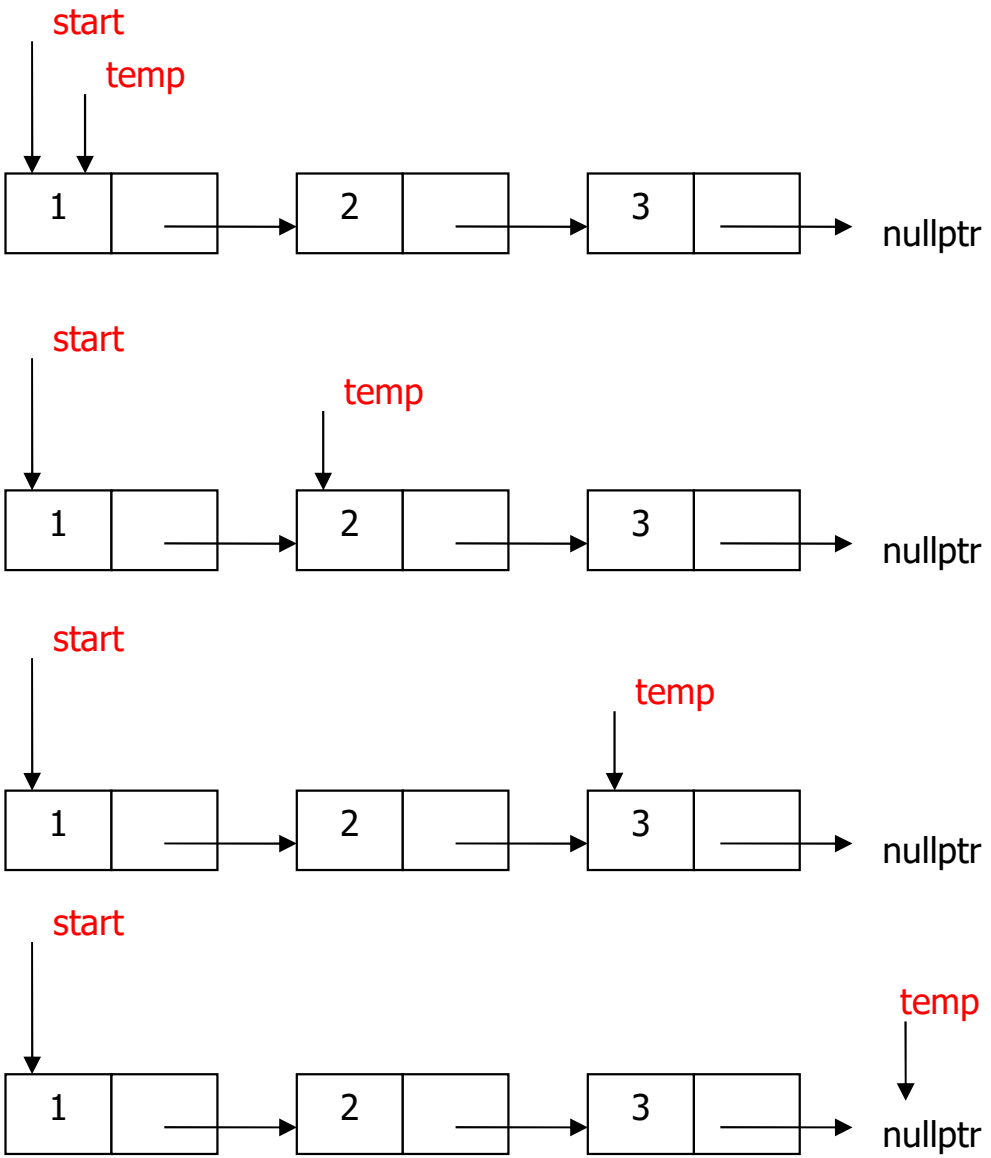
Kazalci - ponovitev

// Example13

```
//void print(Node*& start) {
void print(Node* start) {
    Node* temp = start;
    while (temp) {
        std::cout << temp-> el << " ";
        temp=temp->ptrNext;
    }
    std::cout << std::endl;
}
```

...

```
Node* ptrStart = nullptr;
insertAtBeginning(ptrStart, 3);
insertAtBeginning(ptrStart, 2);
insertAtBeginning(ptrStart, 1);
print(ptrStart);
print(ptrStart);
```



Primer 14

//Shape.h

```
#include <iostream>

class Shape {  // abstract class
protected:
    int x,y;
public:
    Shape() : x(0), y(0) {}
    Shape(int x, int y) : x(x), y(y) {}
    virtual double area() const = 0;   //abstract constant method
    virtual void print() const  = 0;   //abstract constant method
    virtual void relMove(int dx, int dy) {
        x+=dx;
        y+=dy;
    }
};
```

Primer 14

//Circle.h

```
#include <iostream>

class Circle : public Shape {
private:
    int r;
public:
    Circle() : Shape(), r(0) {
    }
    Circle(int x, int y, int r) : Shape(x, y) , r(r) {
    }
    double area() const {
        return 3.14*r*r;
    }
    void print() const {
        std::cout << "Circle(" << x << ", " << y << ", " << r << ")" << std::endl;
    }
};
```

Primer 14

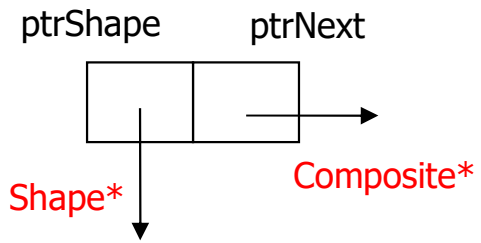
//Rectangle.h

```
#include <iostream>

class Rectangle : public Shape {
private:
    int w, h;
public:
    Rectangle() : Shape(), w(0), h(0) {
    }
    Rectangle(int x, int y, int w, int h) : Shape(x, y), w(w), h(h) {
    }
    double area() const {
        return w*h;
    }
    void print() const {
        std::cout << "Rectangle(" << x << ", " << y << ", " <<
                                     w << ", " << h << ")" << std::endl;}
};
```

Primer 14

//Composite.h

```
class Composite : public Shape {
private:
    Shape* ptrShape;
    Composite* ptrNext;
public:
    Composite(Shape* s = nullptr);
    void add(Shape* s);
    double area() const;
    void print() const;
    void relMove(int x1, int y1);
    void deleteRec();
};
```



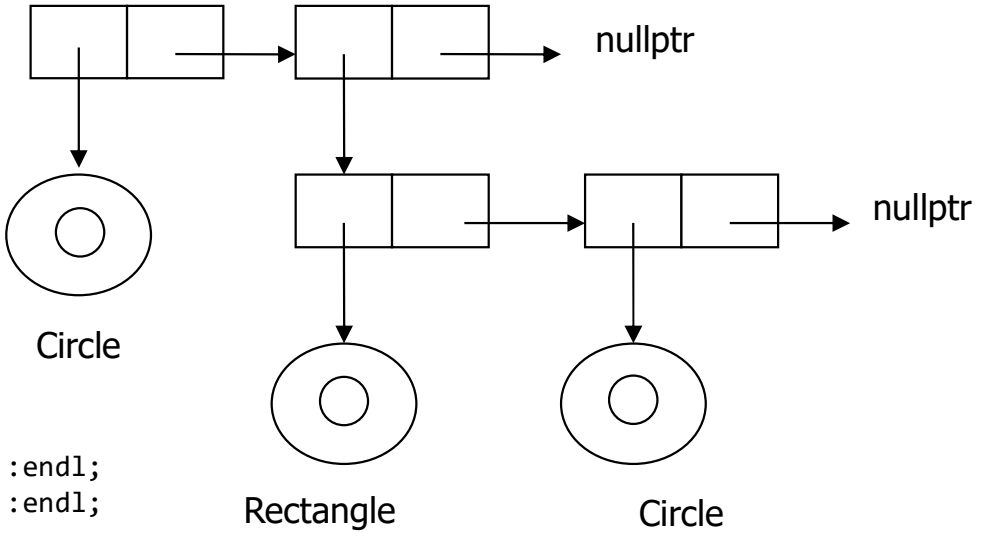
//Composite.cpp

```
Composite::Composite(Shape* s) : Shape(), ptrShape(s), ptrNext(nullptr) {
}

void Composite::add(Shape* s) {
    if (!ptrShape) ptrShape = s;
    else {
        if (!ptrNext) ptrNext = new Composite(s);
        else ptrNext->add(s);
    }
}

double Composite::area() const {
    if (!ptrNext) return ptrShape->area();
    else return ptrShape->area() + ptrNext->area();
}

void Composite::print() const {
    if (ptrShape) ptrShape->print();
    if (ptrNext) ptrNext->print();
}

void Composite::relMove(int x1, int y1) {
    if (ptrShape) ptrShape->relMove(x1, y1);
    if (ptrNext) ptrNext->relMove(x1, y1);
}

void Composite::deleteRec() {
    if (ptrShape) delete ptrShape;
    if (ptrNext) ptrNext->deleteRec();
}
```

Primer 14

//Example14.cpp

```
#include <iostream>
#include "Shape.h"
#include "Circle.h"
#include "Rectangle.h"
#include "Composite.h"

int main() {
     Rectangle r1(0,0,10,10);
     Circle c1(0, 0, 10);
     r1.print();
     c1.print();
     std::cout << "area of a circle    = " << c1.area() << std::endl;
     std::cout << "area of a rectangle = " << r1.area() << std::endl;

     std::cout << "---------------" << std::endl;
     Composite c;
     c.add(new Rectangle(1,1,1,10));
     c.add(new Circle(1,1,1));
     c.print();
     std::cout << "---------------" << std::endl;
     Composite cc;
     cc.add(new Circle(2,2,2));
     cc.add(&c);
     cc.print();
     std::cout << "area of a composite = " << cc.area() << std::endl;  //3.14*2*2 + 1*10 + 3.14*1*1
     std::cout << "---------------" << std::endl;
     cc.relMove(10,10);
     cc.print();
     c.deleteRec();
     cc.deleteRec();
    return 0;
}
```



Vprašanja