# 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

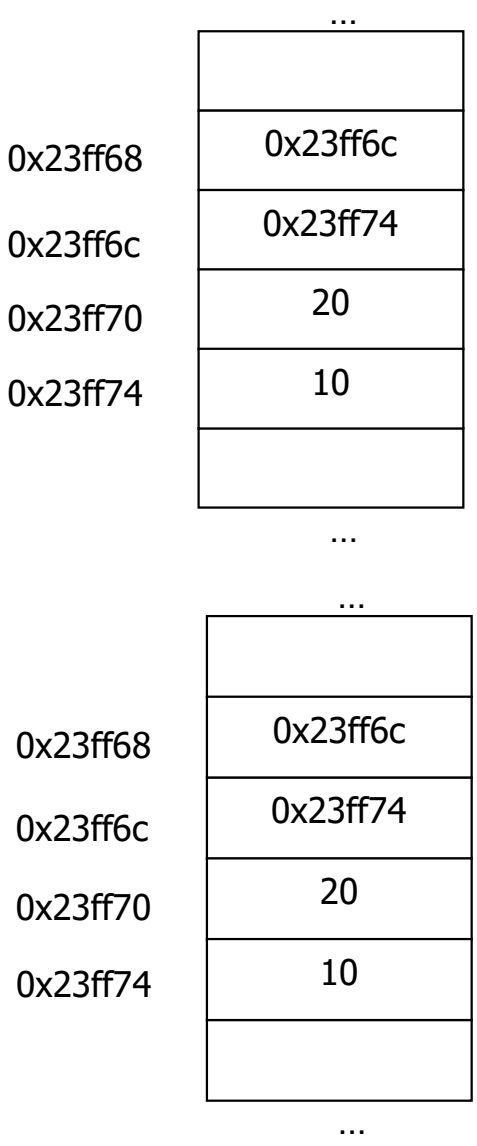
- V programskem jeziku C++ je razlika med kazalcem in referenco še manjša. Referenca je samo konstanten kazalec (vedno kaže na isto lokacijo), pri kateri se derefenciranje izvede implicitno.
- Prav tako v jeziku C++ reference niso vrednosti prvega razreda, saj ne moremo ustvariti polja referenc, kazalce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

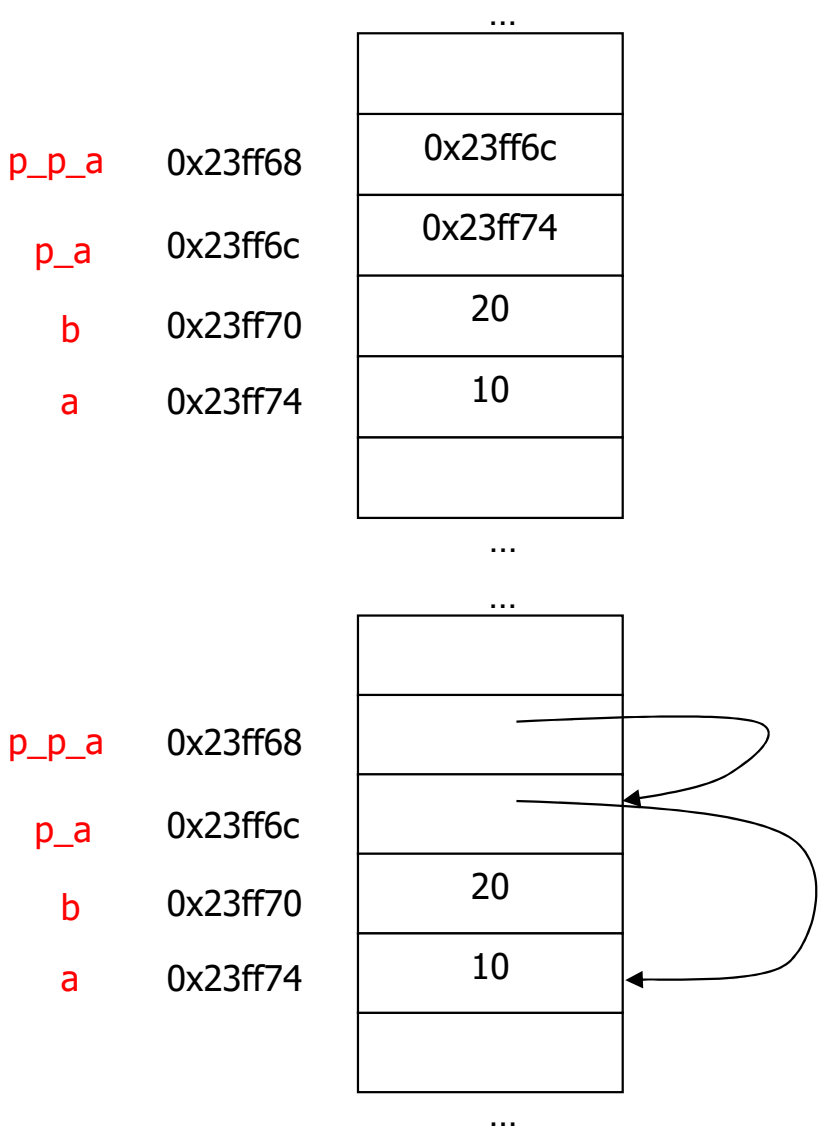


# Kazalci - ponovitev

// Example13

```
#include <iostream>

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

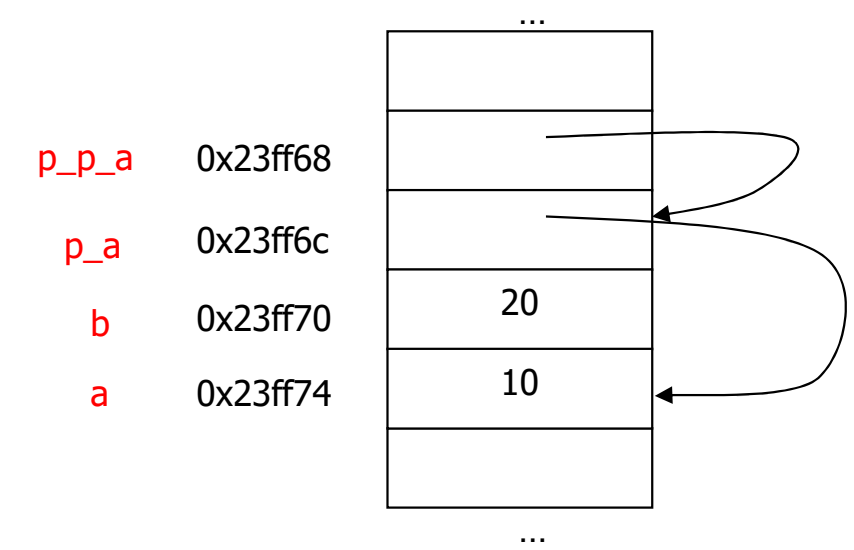


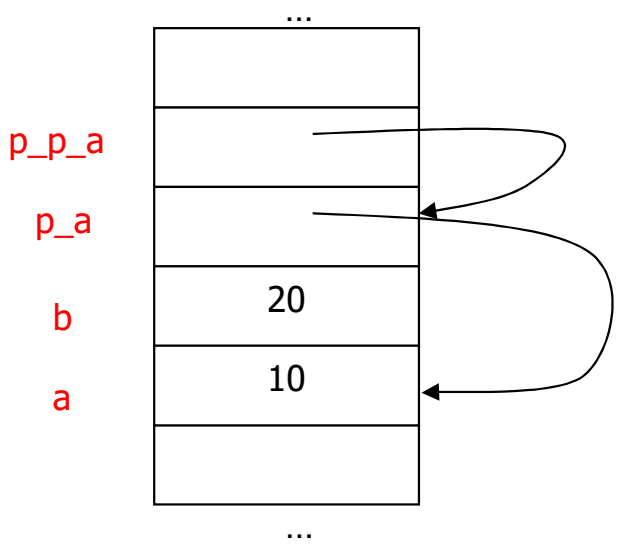
# Kazalci - ponovitev

// Example13

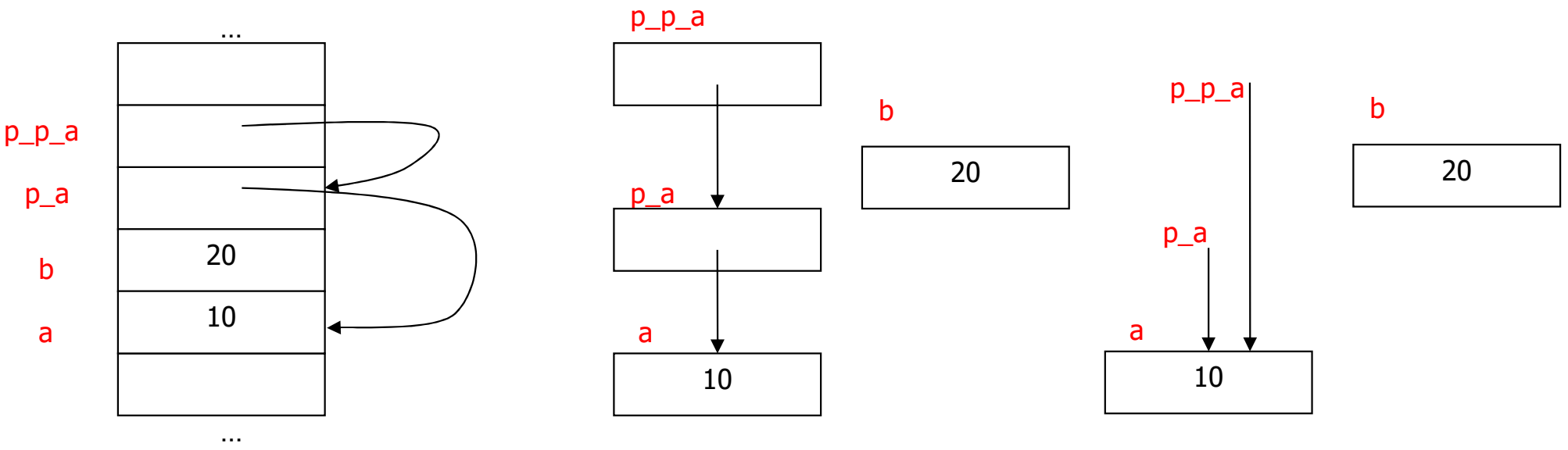
```
#include <iostream>

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



# Kazalci - ponovitev



# Kazalci - ponovitev

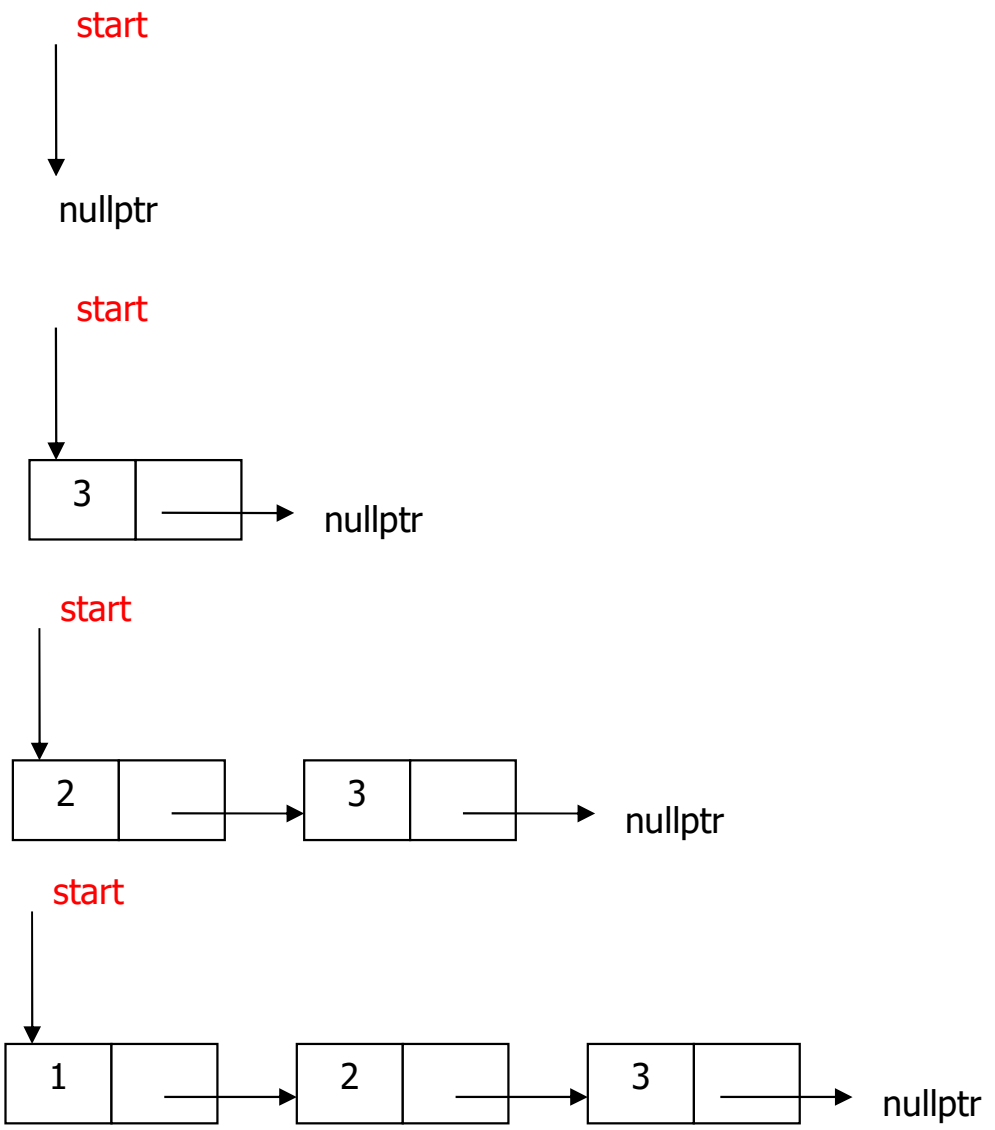
// Example13

```
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);
```



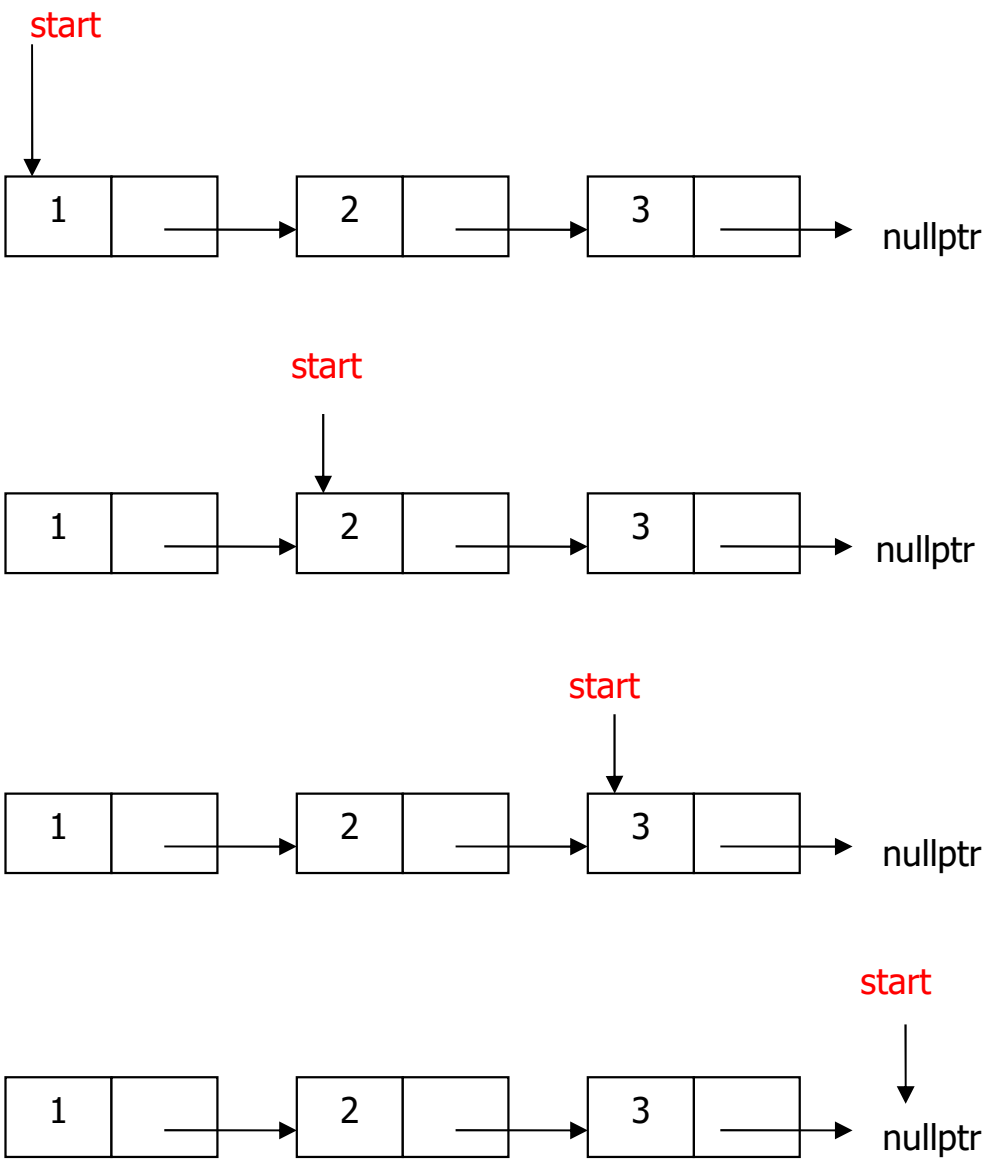
# Kazalci - ponovitev

// 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);
```



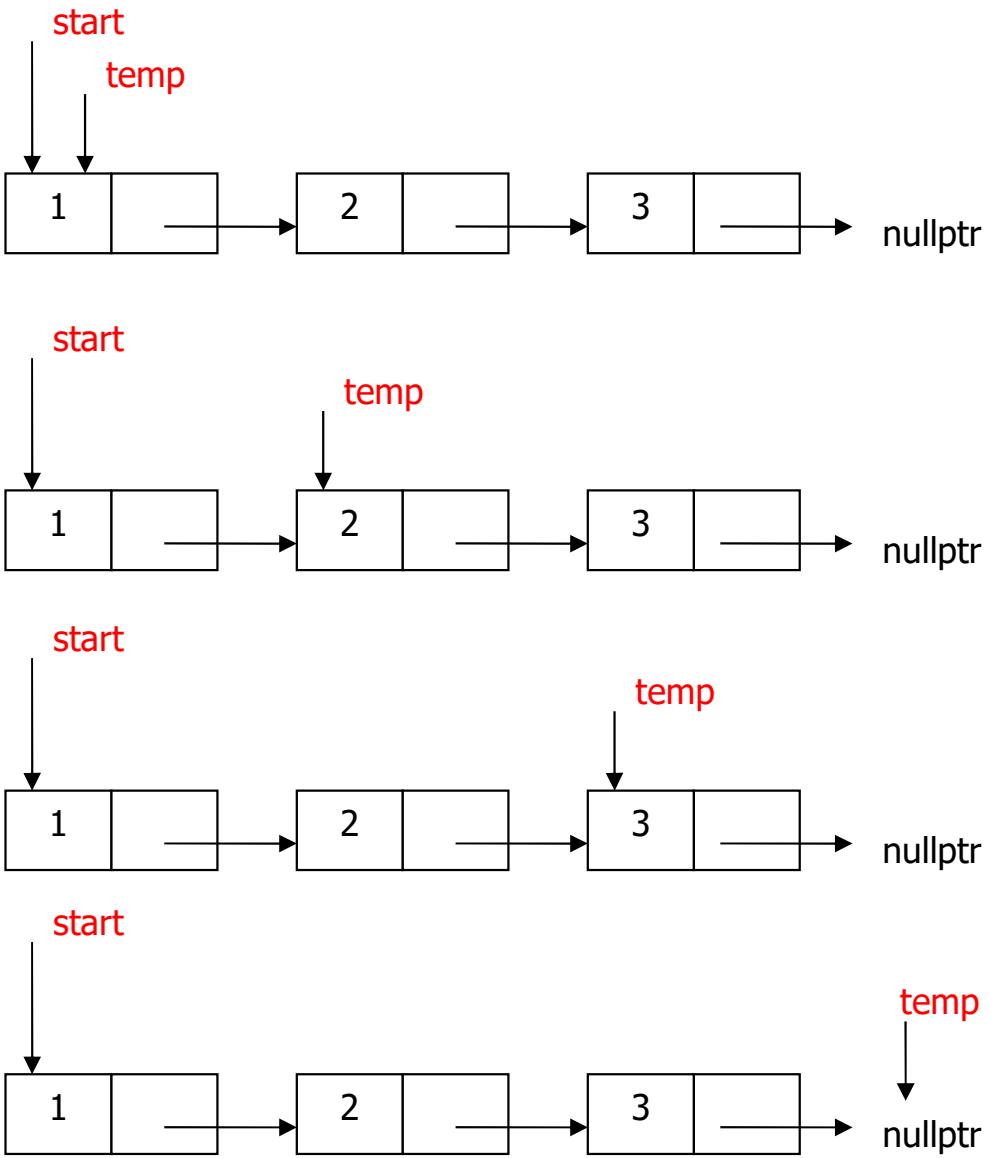
# Kazalci - ponovitev

// 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

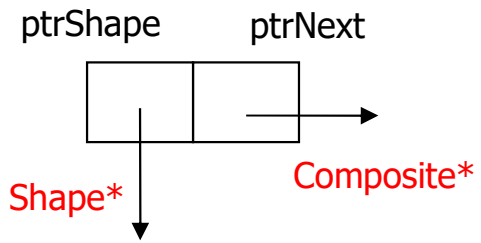
```
#include <iostream>

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();
}
```

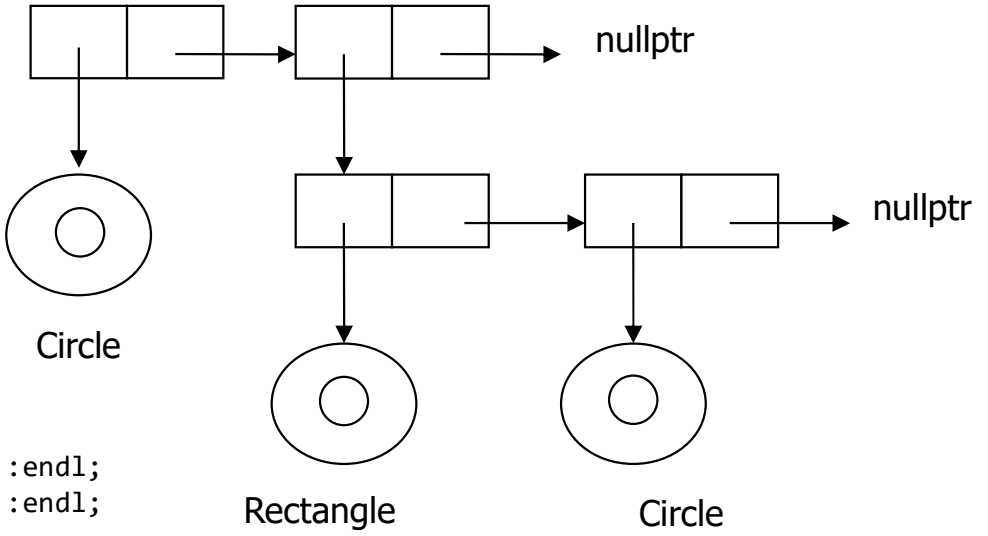
# Primer 14

//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