

Klases

n Klase reprezentē objektu kopu un operācijas, kas ļauj manipulēt ar šiem objektiem, radīt un likvidēt tos. Atvasinātās klases manto bāzes klases locekļus.

n Vienkāršota klases deklarācijas sintakse:

```
class vārds [: bāzes_klase]
{
    klases_locekļu_saraksts
};
```

n Piemērs:

```
class Alpha
{
    private: int a, b;
    public: void set(int, int);
};
```

```
Alpha al, m[10], *pa = &m[3];
int k, n[100], *p;
```

42

Klases (turpinājums)

Klases locekļu sarakstā var būt:

- datu deklarācijas;
- funkciju deklarācijas un definīcijas;

Funkcijām var būt vienādi vārdi, ja ir atšķirīgi parametru saraksti – notiek t. s. funkcijas pārlāde (*overloading*).

```
class Beta
{
    private:    int temp;
               float delta;
    public:    void set(int, float);
               void set(int);
};
```

Nav iespējams atšķirt parametru pēc vērtības un pēc atsaucēs!

```
void setVal( int x );
void setVal( int& x );
```

//setVal(z); - Kļūda! Nevar noteikt, kura funkcija jāizsauc.

43

Klases (turpinājums)

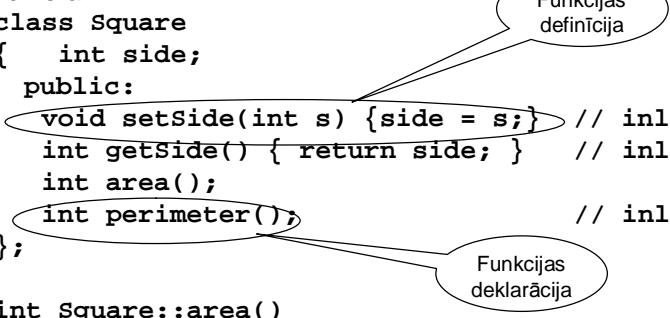
- n Ja funkcija ir definēta klases locekļu sarakstā, tad tā ir iebūvēta (*inline*) funkcija, kas kompilatoram iesaka, ka funkcijas izsaukums jāaizvieto ar tās definīciju.

Piemēram:

```
class Square
{
    int side;
    public:
        void setSide(int s) {side = s;} // inline
        int getSide() { return side; } // inline
        int area();
        int perimeter(); // inline
};

int Square::area()
{
    return side * side;
}

inline int Square::perimeter()
{
    return side * 4;
}
```



44

Klases locekļu pieejamība – iekapsulēšanas princips

- n Klases locekļiem (mainīgajiem un funkcijām) iespējami trīs piekļuves veidi:

public – publisks

private – privāts (pēc noklusēšanas)

protected – aizsargāts

- n Klases deklarācijas sintakse:

```
class klases_vārds [: bāzes_klase(s)]
{
    public:
        publisko_klases_locekļu_saraksts
    private:
        privāto_klases_locekļu_saraksts
    protected:
        aizsargāto_klases_locekļu_saraksts
};
```

45

Piemērs

```
class Triangle {
    private:
        int a, b, c;
    public:
        int setSides(int x, int y, int z);
        void getSides(int *x, int *y, int *z);
        float area();
        int perimeter() { return (a + b + c); }
};
```

46

Piemērs (turpinājums)

```
#include <math.h>

float Triangle::area()
{ float p, s;
  p = perimeter() / 2.0;
  s = ( p*(p-a)*(p-b)*(p-c) );
  if ( s >= 0 ) return( sqrt( s ) );
  else return -1;
}

int Triangle::setSides(int x, int y, int z)
{ a = x; b = y; c = z;
  if ( a + b < c || b + c < a || a + c < b ) return 0;
  return 1;
}

void Triangle::getSides(int *x, int *y, int *z)
{ *x = a;
  *y = b;
  *z = c;
}
```

47

Nepilnā klases deklarācija

```
class klases_vārds;
```

- n Ļauj atsaukties uz klases rādītāju, pirms klases pilnās deklarācijas. Piemēram:

```
class Document; //nepilnā klases Document deklarācija

...

Document *d;

Document* docSearch(int key);
```

48

Klases konstruktori

- n Konstruktors ir speciāla klases funkcija, kas tiek automātiski izsaukta, kad tiek izveidots klases objekts
- § Lokālajam objektam (auto) konstruktors tiek izsaukts, kad blokā izveido mainīgo
 - § Dinamiskiem objektiem, konstruktors tiek izsaukts, kad izpilda new.
 - § Globāliem un statiskiem mainīgajiem, konstruktors tiek izsaukts pirms funkcijas mai n() izsaukuma.

```
#include "triang.h" // Triangle deklarācijas iekļaušana

Triangle t1; // konstruktoru izsauc pirms funkcijas main()

void main()
{
    Triangle t2;          // konstruktora izsaukums
    Triangle* pt3;
    pt3 = new Triangle; // konstruktora izsaukums
    ...
    delete pt3;
}
```

49

Klases konstruktori (turpinājums)

- n Konstruktors ir klases funkcija, kuras vārds sakrīt ar klases vārdu
- n Konstruktoram nav atgriežamās vērtības tipa

```
class X
{
    int a;
    char b;

public:
    X();
    X(int);
    X(int, char);
    //X(X); // KĻŪDA!
    X(X&); // kopijas konstruktors

};
```

- n Kopijas konstruktoram vienīgais parametrs ir atsauce uz šīs klases objektu
- n Kopijas konstruktors rada dotā objekta kopiju

50

Klases konstruktori (turpinājums)

- n Konstruktori parasti tiek izmantoti, lai inicializētu klases mainīgos (atribūtus) un sagatavotu klases objektu

```
X::X()
{
    a = 0;
    b = '\0';
}

X::X(int n)
{
    a = n;
    b = '\0';
}

X::X(int a, char b)
{
    this->a = a;
    this->b = b;
}

X::X(X& x)
{
    a = x.a; //atļauta piekļuve klases privātajiem atribūtiem!
    b = x.b; // -- " --
}
```

51

Klases konstruktori (turpinājums)

- n Ja klasei nav deklarēts neviens konstruktors, tad kompilators automātiski ģenerē konstruktoru bez parametriem
- n Ja klasei nepieciešams kopijas konstruktors, bet tas nav deklarēts, tad kompilators tādu ģenerē pats

```
class Y
{
    int alpha;
public:
    Y();
    Y(int);
    Y(Y&);
};

class Z
{
    int beta;
    char gamma;
};

void main()
{
    Y a;          // Y()
    Y b(10);      // Y(int)
    Y c(a);       // Y(Y&)
    Y d = a;      // Y(Y&)
    Y e = 15;     // Y(int)
    Z k;          // Z()
    Z n = k;      // Z(Z&)
    Z m = 15;     // KĻŪDA!
}
```

52

Klases destruktori

- n Destruktors ir speciāla klases funkcija, kas tiek automātiski izsaukta, kad objekts tiek likvidēts.
 - § Klasei var būt tikai viens destruktors
 - § Destruktoram nevar būt parametri
 - § Destruktoram nav atgriežamās vērtības tipa
 - § Destruktora vārds sakrīt ar klases vārdu un tildes (~) simbolu tā sākumā.
- n Ja klasei nav deklarēts destruktors, tad kompilators to ģenerē pats.

```
class R
{
    int nn;
public:
    R(int); // konstruktors
    ~R();   // destruktors
};
```

Cik klasei R ir konstruktoru?

53

Klases destruktori (turpinājums)

n Destruktors tiek automātiski izsaukts:

- § Lokālajiem objektiem (auto) – kad programmas izpilde iziet no bloka, kurā objekti aprakstīti
- § Dinamiskiem objektiem – kad izpilda **delete**
- § Globāliem un statiskiem mainīgajiem – pēc funkcijas main() beigām

54

Konstruktori un destruktori funkcijās

```
#include <iostream.h>
class A
{
private: int a;
public:
    A()
    { a = 0;
      cout << "Default constructor";
    }
    A(int v)
    { a = v;
      cout << "INT constructor";
    }
    A(const A& aa)
    { a = aa.a;
      cout << "Copy constructor";
    }
    ~A()
    { cout << "Destructor";
    }

    int getA() { return a; }
};
```

```
void printA(A o) //parametrs ir objekts
{
    cout << o.getA() << endl;
}
```

```
void main()
{
    cout << "main begin" << endl;
    A f(4);
    printA(f);
    cout << "main end" << endl;
}
```

Programmas darbības rezultāts:

```
main begin
INT constructor
Copy constructor
4
Destructor
main end
```

55

Konstruktori un destruktori funkcijās

```
#include <iostream.h>
class A
{
private: int a;
public:
    A()
    { a = 0;
      cout << "Default constructor";
    }
    A(int v)
    { a = v;
      cout << "INT constructor";
    }
    A(const A& aa)
    { a = aa.a;
      cout << "Copy constructor";
    }
    ~A()
    { cout << "Destructor";
    }

    int getA() { return a; }
};
```

```
void printA(A& o) //parametrs ir atsauce
{
    cout << o.getA() << endl;
}
```

```
void main()
{
    cout << "main begin" << endl;
    A f(4);
    printA(f);
    cout << "main end" << endl;
}
```

Programmas darbības rezultāts:

```
main begin
INT constructor
4
main end
```

56

Vienkārša simbolu virknes klase

```
class MyString{
private:
    char s[256];
    int len;
public:
    void assign(char *str);
    int length() {return len;};
    void print();
};

// Assign string value
void MyString::assign(char *str)
{
    strcpy(s, str);           //copy string
    len = strlen(str); //assign length
}

// Output of string value with newline
void MyString::print()
{
    cout << s << "\n";
}
```

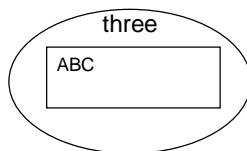
57

Vienkārša simbolu virknes klase

```
MyString one, two; // Izsauc konstruktoru MyString() 2 reizes
one.assign("ABC");
two.assign("Neliels teksta fragments");
```



```
MyString three = one; // Kopijas konstruktors
```

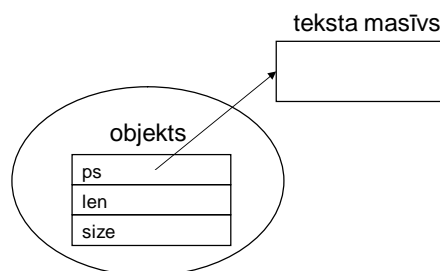


sizeof(three) vērtība ir 260

58

Simbolu virknes klase ar konstruktoriem un destrukturu

```
class MyString{
private:
    char *ps;
    int size;
    int len;
public:
    MyString();
    MyString(int maxLength);
    MyString(char *str);
    ~MyString();
    void assign(char *str);
    int length() {return len;};
    void print();
};
```



59

Simbolu virknes klase ar konstruktorem un destruktorem

```
MyString::MyString()
{ ps = new char[256];
  size = 256;
  len = 0;
}
MyString::MyString( int maxLength )
{ if (maxLength < 1) {
    cout << "Illegal string size : " << maxLength;
    exit(0);}
  ps = new char[maxLength + 1];
  size = maxLength + 1;
  len = 0;
}
MyString::MyString( char *str )
{ len = strlen( str );
  ps = new char[len+1];
  strcpy( ps, str );
  size = len+1;
}
MyString::~MyString()
{ delete ps; }
```

60

Simbolu virknes klase ar konstruktorem un destruktorem

```
// Test MyString class
void main()
{
  char text[] = "Neliels teksta fragments";
  MyString a, b(10), c("ABC");

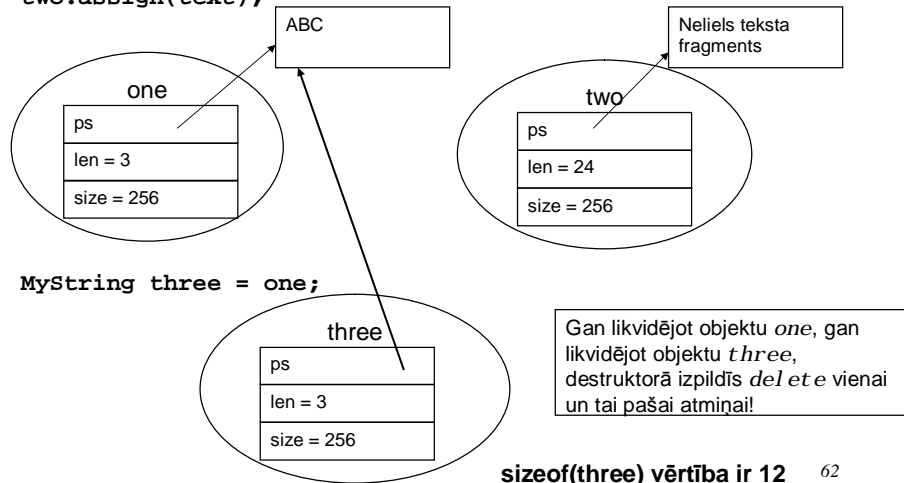
  a.assign("GAMMA");
  b.assign(text);
  a.print();
  b.print();
  c.print();

  MyString *q;
  q = new MyString(25);
  q->assign("DELTA");
  q->print();
}
```

61

Simbolu virknes klase ar konstruktoriem un destruktoru - PROBLĒMAS

```
MyString one, two;
one.assign("ABC");
two.assign(text);
```



Simbolu virknes klase ar konstruktoriem un destruktoru

n Risinājums – jāraksta savs kopijas konstruktors:

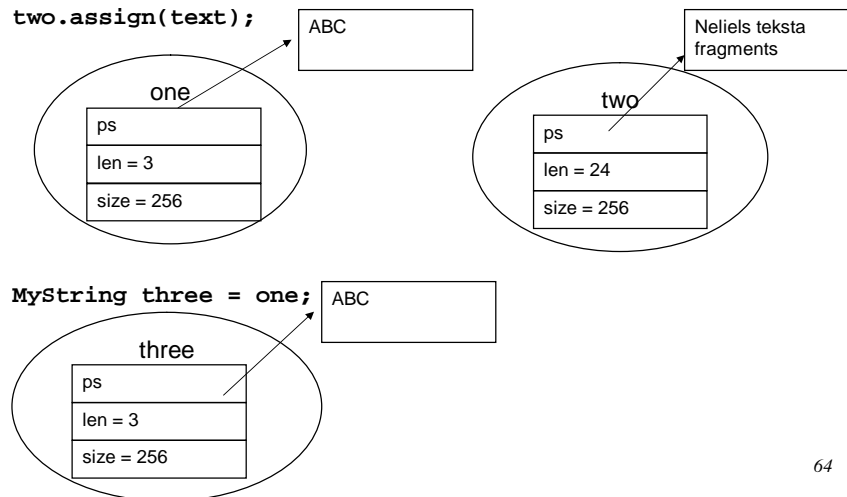
```
class MyString {
private:
    char* ps;
    int len;
public:
    ...
    MyString(const MyString&);
    ...
};

MyString::MyString(const MyString& s)
{
    len = s.len;
    size = s.size;
    ps = new char [size];
    for (int i=0; i<=len; ++i) ps[i] = s.ps[i];
}
```

63

Simbolu virknes klase ar konstruktoriem un destrukturu

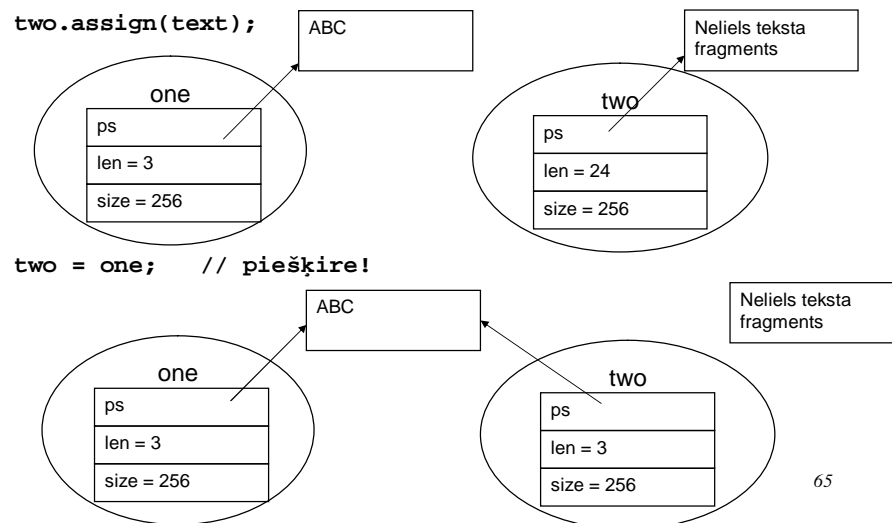
```
MyString one, two;  
one.assign("ABC");  
two.assign(text);
```



64

Simbolu virknes klase ar konstruktoriem un destrukturu

```
MyString one, two;  
one.assign("ABC");  
two.assign(text);
```



65

Simbolu virknes klase ar konstruktoriem un destruktoru

n Risinājums – jāraksta sava piešķires operācija:

```
class MyString {
private:
    char* ps;
    int len;
public:
    ...
    MyString(const MyString&);
    MyString& operator=(const MyString&);
    ...
};

MyString& MyString::operator=(const MyString& s)
{
    len = s.len; size = len+1;
    if (ps) delete[] ps;
    ps = new char [size];
    for (int i=0; i<=len; ++i) ps[i] = s.ps[i];
    return *this;
}
```

66

Konstruktori un destruktori

n Secinājums

Ja starp klases atribūtiem ir rādītāji uz dinamiski iedalītiem atmiņas apgabaliem, tad jāprogrammē savs:

- § kopijas konstruktors
- § piešķires operators
- § destruktors

67

Aprakstu darbības apgabali

n Identifikatora darbības apgabals (scope) ir, programmas daļa, kurā šo identifikatoru var lietot.

n iespējamie darbības apgabali:

- § Bloks
- § Funkcija un tās prototips
- § Klase
- § Fails

68

Aprakstu darbības apgabali – bloks

n Bloka, jeb lokālais darbības apgabals ir spēkā, ja identifikators ir definēts blokā { ... }

```
void Foo()
{
    a = 0; // KĻŪDA! a vēl nav definēts
    int a; // sākas a darbības apgabals
    a = 5; // OK

    {
        int i; // sākas i darbības apgabals
        for (i = 0; i < 10; i++) // sākas b darbības apgabals
        {
            a = a + b; // KĻŪDA! b nav pieejams
        }
    }
} // beidzas a un i darbības apgabali
```

69

Aprakstu darbības apgabali – funkcija un tās prototips (deklarācija)

- n Funkcija ir bloka darbības apgabala speciāls gadījums – funkcijas definīcijas formālo parametru darbības apgabals sakrīt ar funkcijas bloku.

```
class A
{
public:
    float F(int aaa, float bbb); // sākas un beidzas
                                // aaa un bbb darbības apgabali
};

float A::F(int a, float b) // sākas a un b darbības apgabali
{
    float c; // sākas c darbības apgabals
    c = a * b;
    return c;
} // beidzas a, b un c darbības apgabali
```

70

Aprakstu darbības apgabali – klase

- n Klase ir bloka darbības apgabala speciāls gadījums – klases locekļu definīciju darbības apgabals sakrīt ar klases bloku.

```
class A {
public:
    int r;
    float F(int, float);
    void set_r(int rr) { this->r = rr; }
    int get_r() { return r; }
};

void main()
{
    A aa;
    //r = 1;                // KĻŪDA !
    aa.r = 5;               // OK
    float m;
    //m = F(5, 3.14);       // KĻŪDA !
    m = aa.F(5, 3.14);      // OK
    m = aa.A::F(5, 2.17);   // OK
}
```

71

Aprakstu darbības apgabali – fails

n Ja identifikators ir definēts ārpus bloka, tad tā darbības apgabals ir fails, kurā tas ir definēts.

```
float x = 3.14;
void main()
{
    cout << x << endl;    // izvada - 3.14
    ...
    char x[] = "pieci";    // 'paslēpj' float x identifikatoru
    cout << x << endl;    // izvada - pieci
    cout << ::x << endl;  // izvada - 3.14
    ...
}

// beidzas float x darbības apgabals
```

72

Atmiņas klases

n Katram objektam ir tips un atmiņas klase. Atmiņas klase nosaka objekta dzīves ilgumu un tā novietojumu atmiņā.

\$ **static**
\$ **auto**
\$ **register**
\$ **extern**

- Šīs klases lokālie mainīgie netiek iznīcināti, kad notiek izeja ārpus bloka, kurā tie ir definēti.
- Programmas izpildes laikā tie tiek izveidoti tikai vienu reizi, vai arī vispār netiek izveidoti, ja programmas izpilde neieiet blokā, kurā tie ir definēti.
- Šīs klases globālajiem mainīgiem un funkcijām, kas nepieder klasēm, ir faila darbības apgabals.

```
void demoStat()
{
    static int a = 1;
    int b = 1;
    cout << a << " " << b << endl;
    a++;
    b++;
}
```

```
void main()
{
    // ...
    demoStat();
    demoStat();
    demoStat();
    // ...
}
```

73

Atmiņas klases

n Katram objektam ir tips un atmiņas klase. Atmiņas klase nosaka objekta dzīves ilgumu un tā novietojumu atmiņā.

\$ **static**
\$ **auto**
\$ **register**
\$ **extern**

- Automātiskie mainīgie tiek izveidoti programmas stekā brīdī, kad tie ir definēti blokā, un izdzēsti no steka, kad notiek izeja no šī bloka.

- Šī atmiņas klase ir pēc noklusēšanas lokālajiem mainīgajiem, tāpēc tiešā veidā to raksta reti.

```
void MyFun(int i, double d, Triangle* t)
{
    auto int j;
    ...
}
```

74

Atmiņas klases

n Katram objektam ir tips un atmiņas klase. Atmiņas klase nosaka objekta dzīves ilgumu un tā novietojumu atmiņā.

\$ **static**
\$ **auto**
\$ **register**
\$ **extern**

- Reģistra atmiņas klase iesaka kompilatoram objektu glabāt procesora reģistrā nevis stekā.

- Šīs atmiņas klases mainīgajiem nav iespējams iegūt adresi (&).

```
register int i, j;
for ( i=0; i<=bigMAX; ++i)
    for ( j=0; i<=bigMIN; ++j)
    ...
```

75

Atmiņas klases

- n Katram objektam ir tips un atmiņas klase. Atmiņas klase nosaka objekta dzīves ilgumu un tā novietojumu atmiņā.

\$ static

\$ auto

\$ register

\$ **extern**

- Globāliem mainīgajiem, konstantēm vai funkcijām, kas nav klases metodes, šī atmiņas klase norāda, ka mainīgais, konstante vai funkcija ir definēti citā failā.
- Funkcijas, kas nav klases metodes, pieder šai atmiņas klasei pēc noklusēšanas.

math.cpp

```
const double pi = 3.14159;
bool gFlag = false;
double Sin(double p)
{
    ...
}
```

tool.cpp

```
extern const double pi;
extern bool gFlag;
extern double Sin(double p);
```

76

Atsauces

- n Atsauces tiek izmantotas kā mainīgo alternatīvie nosaukumi.
- n Definējot atsauci, tās nosaukuma sākumā liek ampersanda (&) zīmi.
- n Objektam, uz kuru norāda atsauce, ir jāeksistē. Tāpēc atsauces definīcijā tā obligāti ir arī jāinicializē. Inicializētu atsauci nevar mainīt, bet var mainīt objektu, uz kuru atsauce norāda.

```
int i = 20;
int &r = i; // atsauce uz vesela skaitļa mainīgo i
r++;       // mainīgā i tiks izmainīta vērtība uz 21
```

- n Atsauces visbiežāk izmanto kā funkciju parametrus, tādā veidā ļaujot funkcijai mainīt nodotos parametrus un izmaiņas atdot izsaucošajai funkcijai.
- n Atsauces bieži lieto funkcijās, kas kā savu vērtību atgriež objektu – tā vietā tiek atgriezta objekta atsauce:

```
Triangle& getTriangle(int index);
...
cout << getTriangle(1).perimeter() << endl;
```

77

Kvalifikators - const

n Kvalifikators `const` aizliedz veikt izmaiņas vērtībā, kuram identifikators ir piesaistīts.

§ Globālie un lokālie mainīgie ar šo kvalifikatoru obligāti jāinicializē. Pēc tam tā vērtību vairs nav iespējams mainīt.

```
const int wheels = 4;
...
//wheels = 3; //KĻŪDA !
```

§ Funkcijas argumentus ar šo kvalifikatoru nav iespējams mainīt.

78

Kvalifikators – const rādītājiem

n Kvalifikatoru `const` var lietot rādītāju definīcijās, bet atkarībā no šī kvalifikatora norādīšanas vietas, rādītāja darbība ir dažāda.

```
int i = 100;
int j = 200;
```

```
const int * p1; // rādītājs uz vesela skaitļa konstanti
p1 = &i;
// *p1 = j; // KĻŪDA !
p1 = &j;
```

```
int * const p2 = &i; // konstants rādītājs uz veselu skaitli
*p2 = j;
// p2 = &j; // KĻŪDA !
```

```
const int * const p3 = &i; // konstants rādītājs uz vesela skaitļa
// konstanti
// p3 = &j; // KĻŪDA !
// *p3 = j; // KĻŪDA !
```

79