Polimorfisms (atkārtojums)

- **Vārdam 'polimorfisms' ir grieķu izcelsme un tā nozīme ir 'tāds, kuram ir vairākas formas'.**
- **Objektorientētajā programmēšanā polimorfisma princips ir 'viens interfeiss, vairākas metodes'.**
- **Valodā C++ polimorfismu visbiežāk lieto attiecībā uz funkcijām un operācijām, kad vienādi nosauktas funkcijas vai operācijas realizē dažādas darbības.**

Operācija <<

```
cout << "Ievadiet paroli: ";
...
int n, flag;
flag = n << 2;
```


Polimorfisms - operāciju definēšana

- Viena operācija **+**, bet darbības ir dažādas:
  - 2 + 2 – veselu skaitļu saskaitīšana
  - 2.01 + 3.14 – racionālu skaitļu saskaitīšana
  - 2i + 3i – iracionālu skaitļu saskaitīšana
  - "abc" + "def" – teksta saķēdēšana (konkatenācija)

- Operāciju pieraksta veidi:
  - Prefiksa

    ```
    * + 2 3 + 5 6
    ```

    vai

    ```
    mul(sum(2, 3), sum(5, 6))
    ```

  - Infiksa

    ```
    (2 + 3) * (5 + 6)
    ```

  - Postfiksa

    ```
    2 3 + 5 6 + *
    ```

Operācijas (angļu val. *operators*)

#	*Category*	*Operator*	*Associativity*
1.	**Highest**	`() [] -> :: .`	¾ ®
2.	**Unary**	`! ~ + - ++ -- & * sizeof new delete`	¬ ¾
3.	**Member access**	`.* ->*`	¾ ®
4.	**Multiplicative**	`* / %`	¾ ®
5.	**Additive**	`+ -`	¾ ®
6	**Shift**	`<< >>`	¾ ®
7.	**Relational**	`< <= > >=`	¾ ®
8.	**Equality**	`== !=`	¾ ®
9.	**Bitwise AND**	`&`	¾ ®
10.	**Bitwise XOR**	`^`	¾ ®
11.	**Bitwise OR**	`|`	¾ ®
12.	**Logical AND**	`&&`	¾ ®
13.	**Logical OR**	`||`	¾ ®
14.	**Conditional**	`?:`	¬ ¾
15.	**Assignment**	`= *= /= %= += -= &= ^= |= <<= >>=`	¬ ¾
16.	**Comma**	`,`	¾ ®

135

Operāciju definēšana

- Valodā C++ var definēt visas esošās operācijas, izņemot `. .* :: ?:`
- Definējot operāciju, saglabājas tās operandu skaits, prioritāte un asociativitātes likumi – tos mainīt nav iespējams.
- Nav iespējams pārdefinēt iebūvēto (standarta) tipu operācijas.
- Nav iespējams definēt jaunas operācijas (piemēram, `**`)

136

Operāciju definēšana (turpinājums)

- Ar simbolu @ apzīmēsim jebkuru definējamo operāciju (tāda operācijas simbola valodā C++ nav!)
- Operāciju @ definē kā funkciju, kuras vārds ir **operator@**
- Operācijas var definēt kā klases funkcijas vai kā ārējas funkcijas
- Ja operācija @ ir bināra, tad izteiksmi

  x @ y

  izpilda kā

  x.operator@(y) – klases funkcijas gadījumā
  operator@(x, y) – ārējas funkcijas gadījumā
- Ja operācija @ ir unāra, tad izteiksmi

  @ x

  izpilda kā

  x.operator@() – klases funkcijas gadījumā
  operator@(x) – ārējas funkcijas gadījumā

137

Operāciju definēšana (turpinājums)

- Bināras operācijas gadījumā, abi operācijas operandi ir ārējās funkcijas parametri.
- Klases locekļa funkcijas gadījumā, pirmais operands vienmēr ir pats klases objekts, kuram izpilda funkciju.
- Ārēja funkcija parasti ir klases draugs (**`friend`**), lai tā varētu piekļūt operandu klašu **`private`** un **`protected`** locekļiem.

138

Operāciju definēšana (piemērs)

```
class Vect{
private:
  int *p;
  int size;
public:
  Vect();
  Vect(int n);
  Vect(const Vect& v);
  Vect(const int a[], int n);
  ~Vect() { delete p; }

  int& operator[](int i);           // overloaded []
  Vect& operator=(const Vect& v);   // overloaded =

...
};
```


Operāciju definēšana (piemēra turp.)

```
Vect::Vect()
{  size = 16;
   p = new int[size];
}

Vect::Vect(int n)
{  if ( n <= 0 ){      cout<<"Illegal Vect size: "<< n <<'\n';
                       exit(1);}
   size = n;
   p = new int[size];
}

Vect::Vect(const Vect& v)
{  size = v.size;
   p = new int[size];
   for( int i = 0; i < size; ++i) p[i] = v.p[i];
}

Vect::Vect(const int a[], int n)
{  if ( n <= 0 ){      cout<<"Illegal Vect size: "<< n <<'\n';
                       exit(1);}
   size = n;
   p = new int[size];
   for( int i = 0; i < size; ++i) p[i] = a[i];
}
```

Operāciju definēšana (piemēra turp.)

```
int& Vect::operator [](int i)
{
  if ( i < 0 || i > size-1 ){
      cout <<  "Illegal Vect index: " << i << '\n';
      exit(2);
  }
  return ( p[i] );
}

Vect& Vect::operator =(const Vect& v)
{
  int s = (size < v.size) ? size : v.size;
  for( int i = 0; i < s; ++i) p[i] = v.p[i];
  return ( *this );
}
```


Operāciju definēšana (piemēra turp.)

```
void main(void)
{  Vect v1;
   Vect v2(7);
   int a[] = { 11, 12, 13, 14, 15, 16, 17, 18, 19 };
   Vect v3( a, 5);
   int k, n;

   v1 = v3;          // v1.operator=(v3);
   k = v1[1];        // k = v1.operator[](1);

   k = v2[n+1] + 1;  // k = v2.operator[](n+1) + 1;

   v2[0] = 100;      // v2.operator[](0) = 100;
   v2[1] = a[1];     // v2.operator[](1) = a[1];

   ...
}
```

Operāciju definēšana (piemēra turp.)

```
class Vect{
private:
   int *p;
   int size;
public:
   ...
   Vect& operator+(int c);  //vesela skaitļa pieskaitīšana vektoram
   ...
};

Vect& Vect::operator+(int c)
{  for( int i = 0; i < size; ++i) p[i] += c;
   return ( *this );
}
```

```
Vect v4;
v4 = v4 + 1000;     // Strādā funkcijas operator+ un operator=
v4 += 1000;         // Kļūda !
v4 = 1000 + v4;     // Kļūda !
```

143

Operāciju definēšana

Ja operāciju definē ar ārēju funkciju, tad izteiksmi

x @ y

izpilda kā

operator@(x, y)

t.i. nodod funkcijai abus operandus kā parametrus

Ārēja funkcija, kas nav klases loceklis, parasti ir klases draugs (*friend*), lai tā varētu piekļūt klases *private* mainīgajiem.

Klases draugs (*friend*) ir funkcija vai klase, kurai ir pilnas piekļūšanas tiesības pie citas klases *private* un *protected* locekļiem.

144

Operāciju definēšana (piemēra turp.)

```
class Vect{
   ...
public:
   ...
   friend Vect& operator+(Vect&, int );  //overloaded Vect + int
   friend Vect& operator+(int , Vect&);  //overloaded int + Vect
};

Vect& operator+( Vect& v, int c)
{  for( int i = 0; i < v.size; ++i) v.p[i] += c;
   return ( v );
}

Vect& operator+(int c, Vect& v)
{  for( int i = 0; i < v.size; ++i) v.p[i] += c;
   return ( v );
}
```


Operāciju definēšana (piemēra turp.)

```
Vect v4, v5;
...

v5 = v4 + 1000;   // izmanto operator+(v4, 1000)

v5 = 1000 + v4;   // izmanto operator+(1000, v4)
```

Vai šīs lekcijas piemēros dotās vektora un vesela skaitļa saskaitīšanas operācijas ir korektas !?

Operāciju definēšana (turpinājums)

```
class Vect
{  int *p, size;
   ...
public:
   Vect& operator++();   //prefix:  ++x
   Vect operator++(int); //postfix: x++
   ...
};

Vect& Vect::operator++()
{
   for (int i = 0; i < size; i++)
      p[i]++;

   return *this;
}

Vect Vect::operator++(int)
{
   Vect res = *this;          //objekta kopija pirms izmaiņām
   for (int i = 0; i < size; i++)
      p[i]++;

   return res;                //atgriež sākotnējā objekta kopiju
}
```

Lai atšķirtu postfiksa operāciju prefiksa operācijas, tās deklarācijā raksta fiktīvu parametru int.

Ja klasē ir definēta tikai prefiksa operācija, bet programmā tiek lietota postfiksa operācija, tad kompilators dod brīdinājumu un izmanto prefiksa operāciju.