C++ programavimo kalba

Santrauka

doc.dr. Dalius Mažeika
Dalius.Mazeika@fm.vtu.lt
VGTU SC L318

Programavimo kalbos

- Procedūrinės (Assembler, ankstyvos, Fortran versijos)
  - Pagrindą sudaro procedūros. Kiekvienas operatorius, lyg procedūra nurodo kompiuteriui, ką daryti (programa – tai instrukcijų rinkinys).
    Programos kodo ilgis iki kelių tūkstančių eilučių.
- Struktūrinės (Pascal, C, Fortran, Basic, COBOL)
  - Pagrindą sudaro blokai (funkcijos, paprogramės, bibliotekos) ir duomenys, priskirti tiems blokams. Bendrai naudojami duomenys apibrėžiami kaip globalūs. Programos kodo ilgis iki 50 tūkstančių eilučių.
- OOP (C++, C#, Java, Visual Basic)
  - Naudojami objektai bei jų savybės: inkapsuliacija, paveldimumas, polimorfizmas.

OOP

OOP pagrindinė idėja: duomenys ir funkcijos, kurios operuoja tais duomenimis apjungti į vieną vienetą, vadinamą **objektu**.

Objekto funkcijos, kurių dažniausia būna ne viena, vadinamos **metodais**.

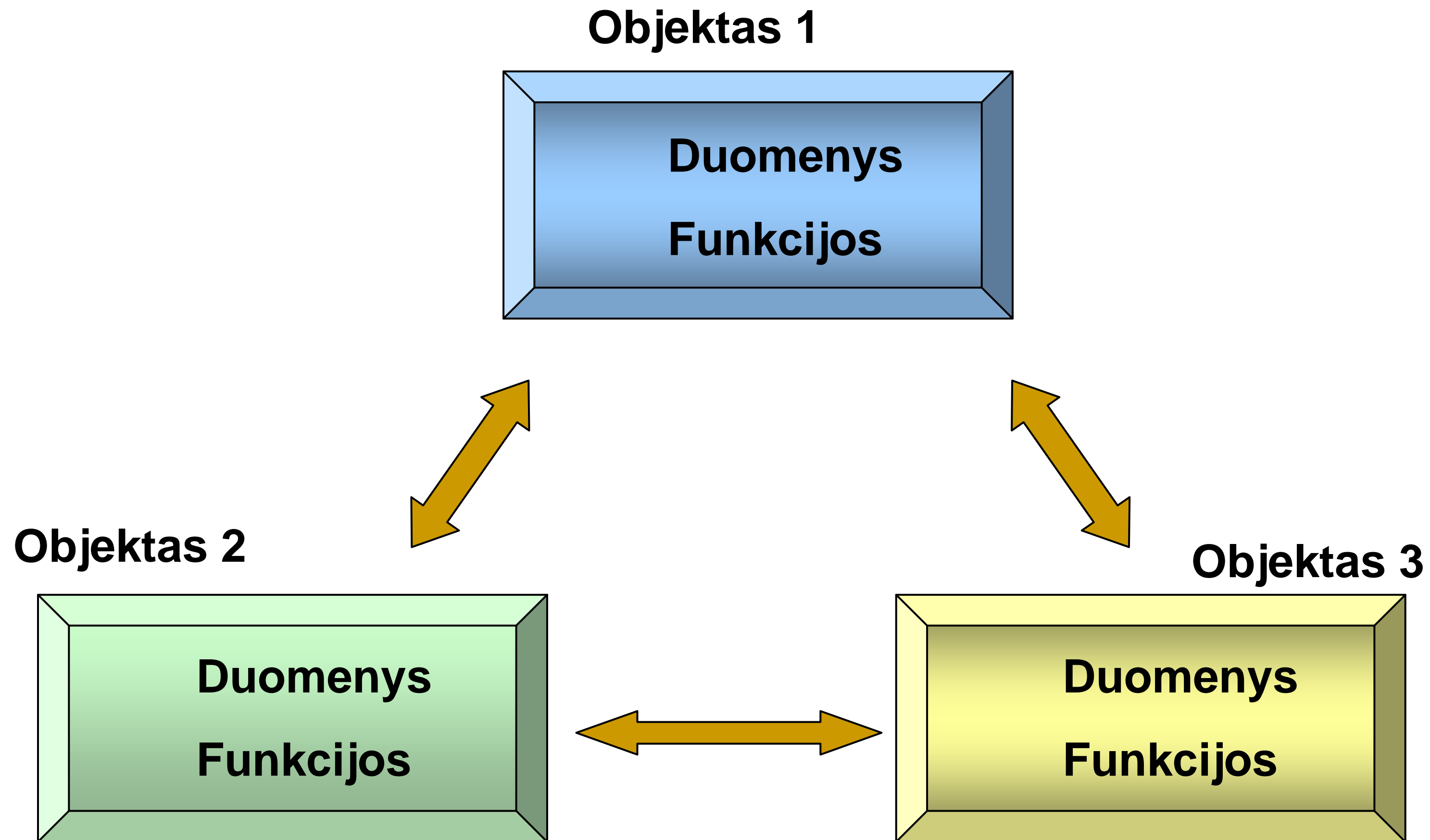
Objekto duomenys dažniausiai pasiekiami (nuskaitomi, modifikuojami ir t.t.) tik per metodus. Tai reiškia, kad duomenys yra paslėpti nuo atsitiktinio modifikavimo. Kitaip dar sakoma, kad duomenys yra **inkapsuliuoti** (*encapsulated*).

C++ programą paprastai sudaro tam tikras skaičius objektų, kurie komunikuoja tarpusavyje iškviesdami vienas kito metodus.

C++

- ANSI/ISO standartizuota 1997m.
- C++ - tai C kalbos pratęsimas, priskiriant ją OOP kalbų grupei, todėl išlieka C ir C++ suderinamumas.
- C++ geriau nei C, nes:
  - Išsamesnis klaidų tikrinamas, lengvesnė klaidų paieška;
  - Nuorodų (*reference*) panaudojimas funkcijų argumentuose ir grąžinamose reikšmėse yra patogesnis nei rodyklių (*pointer*).
  - Funkcijų perkrovimas (*overloading*) leidžia naudoti tuos pačius funkcijų pavadinimus skirtingoms funkcijoms.
  - Vardų erdvės (*namespace*) leidžia geriau kontroliuoti vardų naudojimą.

OOP



Klasė – tai šablonas, pagal kurį kuriamas objektas

Klasės apibrėžimas

C++ kalboje struktūra, jungianti savyje kintamuosius, skirtus duomenims saugoti, ir funkcijas, kurios naudoja tik tuos kintamuosius, vadinama **klase**.

Klasės kintamieji vadinami **duomenimis**, o funkcijos – **metodais**.

Objektiniame programavime priimta, kad duomenimis tiesiogiai gali naudotis tik tos klasės metodai.

Klasės aprašo struktūra:

class [*Klasės_vardas*]

{ *Duomenų elementai;*

public: *Metodai*;

} [objektų sąrašas];

Pavyzdys

```
#include <iostream>
using namespace std;
class small {
       private: int somedata;
       public: void setdata(int d)
                    {somedata = d; }
                  void showdata()
                    {cout<<“Duomenys =“<<somedata<<endl;  }
      };
void main()
{ small s1, s2;
  s1.setdata(109);  s2.setdata(111);
  s1.showdata(); s2.showdata();
}
```

Metodų aprašymas

Metodai klasėje gali būti pilnai aprašyti (prieš tai buvusioje skaidrėje). Tokius metodų aprašus tikslinga turėti, jeigu jų tekstas yra trumpas.

Dažniausiai metodų aprašai iškeliami už klasės ribų. Tuomet klasėje rašomas tik metodo prototipas (metodas deklaruojamas).

Metodo aprašo klasės išorėje sintaksė:

[Reikšmės tipas] [**Klasės Vardas**] :: [**Metodo vardas**] (Parametrų sąrašas)
{ Programos tekstas }

Klasės elementų požymiai

Klasės elementai (duomenys ir metodai) gali turėti požymius. Požymis klasėje galioja tol, kol bus sutiktas kito požymio užrašas.

Jeigu požymio užrašo nėra, tuomet pagal nutylėjimą bus priimtas private visiems elementams iki pirmojo sutikto požymio užrašo, jeigu jis iš viso bus.

C++ klasės elementų požymiai:

- **private** (lokalusis). Duomenys ir metodai prieinami tik klasės metodams.
- **public** (globalusis). Klasės elementai prieinami tiek klasės metodams tiek ir išorinėms funkcijoms.
- **protected** (apsaugotasis). Klasės elementai prieinami klasėje, kuri paveldi duotąją klasę. Paveldėtoje klasėje jie galioja *private* teisėmis.

Pavyzdys

```
#include <iostream>
using namespace std;

class Staciakampis {
    int x, y;
  public:
    void ivedimas (int,int);
    int plotas ()
       {return (x*y);}
};

void Staciakampis::ivedimas (int a, int b)
{
  x = a;
  y = b; }
```

```
int main () {
  Staciakampis rect;
  rect.ivedimas (3,4);
  cout << “Plotas: “;
  cout<<rect.plotas()<<endl;
  return 0;
}
```

Konstruktorius

Sukuriant objektą, jo duomenims paprastai turi būti priskiriamos pradinės reikšmės. Tai gali atlikti tam skirti metodai. Bet būtų patogu, jei kuriant objektą, duomenų inicializacija būtų atliekama **automatiškai**, neiškviečiant papildomų metodų.

Tai galima atlikti naudojant specialų metodą – *konstruktorių*.

Konstruktorius – tai metodas, kuris automatiškai vykdomas kiekvieną kartą, kai tik sukuriamas objektas.

Konstruktoriaus savybės:

- Turi tokį pat pavadinimą, kaip ir klasė;
- Nieko negrąžina t.y. neturi operatoriaus *return*;
- Konstruktoriaus paskirtis – priskirti pradines reikšmes klasės kintamiesiems.

Konstruktorius

Klasė

```
#include <iostream.h>
class studentas
{ private:   float svoris;
  public:
     studentas(float x)         // konstruktorius
     { svoris = x;  }
     void rodo()
     { cout << svoris << endl; }
};
void main()
{ studentas Algis(120.25);
  studentas Jonas(56.2);
  cout<<"Algio svoris "; Algis.rodo();
  cout<<"Jono svoris "Jonas.rodo();}
```

Konstruktorius

```
#include <iostream.h>
class Counter
{private:   unsigned int count;
 public:
     Counter()
     { count = 0;  }
     void increase()
     { count ++ ; }
     int show()
     { return count; }
};
```

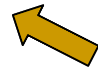
```
void main()
{ Counter C1;
  cout<<"Skaitliukas = "<<
              C1.show() << endl;
C1.increase();
cout <<"Skaitliukas =" <<
              C1.show()<<endl;
}
```

Konstruktorius

Konstruktoriuje klasės duomenims reikšmės gali būti priskiriamos naudojant įprastinę funkcijų sintaksę arba inicializacinį sąrašą.

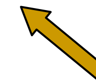
Naudojant inicializacinį sąrašą, kintamiesiems reikšmės priskiriamos prieš konstruktoriaus vykdymą. *const* tipo duomenys bei nuorodos gali būti inicializuotos tik inicializaciniame sąraše.

```
counter()
{ count = 0; }
```



Konstruktorius kaip funkcija

```
counter() : count(0)
 { }
```



Automatinė inicializacija per sąrašą

Inicializacinis sąrašas

```
class circle
{ private:
        int radius;
        int c_x, c_y;
  public:
        circle (int a, int b, int c) : radius(a), c_x(b), c_y(c)
        { }
};
```

Apibendrinus

```
AnyClass() : m1(7), m2(12), m3(23.56)
   { }

AnyClass(int x, int y, float z) : m1(x), m2(y), m3(z)
   { }
```

Konstruktorių perkrovimas (overloading)

Jei klasėje nėra apibrėžtas konstruktorius, kompiliatorius automatiškai kuria konstruktorių be argumentų sąrašo. Toks konstruktorius rezervuoja atmintį objektui, bet duomenims reikšmių nepriskiria.

Visgi tikslinga atskirai sukurti konstruktorių be argumentų sąrašo, priskiriant duomenims pradines reikšmes.

Jei klasėje yra keli konstruktoriai, sakoma, kad konstruktorius yra perkrautas (overloaded).

```
class atstumas
{ private:  float ilgis; float plotis;
  public:  atstumas() : ilgis(0.0), plotis(0.0)
  { }
  atstumas (float x, float y)
 { ilgis = x;   plotis = y; }
};
```

```
void main()
{  atstumas a1, a2;
   atstumas a3(12.4, 45.1);
}
```

Pavyzdys

```
#include <iostream>
using namespace std;

class Staciakampis {
    int ilgis, plotis;
  public:
    Staciakampis (): ilgis(3), plotis(3)
    { }
    Staciakampis (int, int);
    int plotas () {return (ilgis*plotis);}
};

Staciakampis :: Staciakampis (int a, int b)
{
  ilgis = a;
  plotis = b;
}
```

```
int main () {
 Staciakampis rect;
 Staciakampis rectb (5,6);
  cout << “Plotas: " << rect.plotas()
          << endl;
  cout << “Plotas: " <<
                 rectb.plotas() << endl;
  return 0;
}
```

Konstruktorius pagal nutylėjimą

Jei klasėje nėapibrėžtas nei vienas konstruktorius, tuomet kompiliatorius sukuria konstruktorių pagal nutylėjimą. Toks konstuktorius nepriskiria objekto duomenims jokios reikšmės, nors atmintyje išskiria kintamiesiems vietą. Konstruktorius pagal nutylėjimą naudojamas tik tuomet, kai nėra sukuriamas joks kitas konstrukorius.

```
class Staciakampis
{
   int ilgis, plotis;
public:
   int plotas (int a, int b)
 {
   ilgis = a;   plotis = b;
   return (ilgis*plotis); }
};
```

```
class Staciakampis
 {   int ilgis, plotis;
  public:
    Staciakampis (): ilgis(2), plotis(3)
    { }
    int plotas () {
    return (ilgis*plotis); }
};
Staciakampis didelis;
```

Destruktorius

Destruktorius – tai specialus metodas, kuris iškviečiamas automatiškai tuomet, kai naikinamas objektas. Jo paskirtis naikinti objektą ir atlaisvinti atmintį.

Destruktoriaus savybės:

- pavadinimas "~klasės_pavadinimas"
- nieko negrąžina
- neturi argumentų sąrašo
- destruktorius nepaveldimas
- destruktorius negali būti apibrėžtas naudojant modifikatorius *const, violatile, static, virtual*

Jei destruktorių reikia iškviesti rankomis, naudojama sintaksė:

objektas.~klasė();

Destruktorius (pavyzdys)

```
#include <iostream>
using namespace std;

class Staciakampis {
    int *ilgis, *plotis;
  public:
Staciakampis (int, int);
~Staciakampis ();
    int plotas () { return (ilgis*plotis); }
};

Staciakampis :: Staciakampis (int a, int b)
{ ilgis = new int; plotis = new int;
  *ilgis = a;
  *plotis = b;
}
```

```
Staciakampis ::~ Staciakampis () {
delete width; delete height; }


int main () {
 Staciakampis rect (3,3);
 Staciakampis rectb (5,6);
  cout << “Plotas: " << rect.plotas()
          << endl;
  cout << “Plotas: " <<
                 rectb.plotas() << endl;
  return 0;
}
```

Draugiškumas

Draugiškos funkcijos pavyzdys

```
#include <iostream.h>
class gama
{ int x, y;
  public:
      gama(int a, int b)
      { x = a;  y = b; }
      friend int draugas (gama);      // draugiška funkcija
 };
int draugas (gama AB)
{ return (AB.x - AB.y);
}

main ()
   gama obj(3,5);
   cout<< draugas(obj)<< endl;        // iškviečiama draugiška funkcija
   return 0; }
```

Draugiškos funkcijos pavyzdys

```
#include <iostream.h>
class beta;          // klasės beta deklaravimas
class alfa
{ int data;
  public:
     alfa() : data(3) { }   // ➡ Kas tai???
     friend int draugas (alfa, beta);
};
class beta
{ int data;
  public:
     beta() : data(7) { }
     friend int draugas (alfa, beta);
};
```

```
int draugas (alfa a, beta b)
{ return (a.data + b.data);
}

main ()
   alfa aa; beta bb;
   cout<< draugas(aa, bb)<< endl;
   return 0; }
```

Draugiškos klasės pavyzdys

```
class AA
{ int n, d;
    public:
          AA (int x, int y) : n(x), d(y) { }
          void  increase() { n++; d = d*10; }
          friend class BB ;
};
class BB
{ AA objA(2,6);
  public:
         void show()
         { cout<< “Pradinės reikšmės”<<objA.n<<“ “<<objA.d << endl;
            objA.increase();
          cout<< “Pakeistos reikšmės”<<objA.n<<“ “<<objA.d << endl; }
};
void main()
{ BB objB;
  objB.show(); }
```

Draugiškos klasės pavyzdys

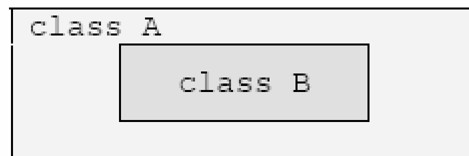
```
#include <iostream.h>
class alfa
{ int data1;
    public:
          alfa() : data1(99) { }
          friend class beta ;
};
class beta
{   public:
         void func1(alfa a)
           { cout<< “data1 reikšmė”<<a.data1<< endl;}
         void func2(alfa a)
           { cout<< “data1 reikšmė”<<a.data1++<< endl;}
};
void main()
{ alfa AA; beta BB;
  BB.func1(AA);
  BB.func2(AA); }
```

Paveldėjimas (inheritance)

Paveldėjimas

Objektiniame programavime viena svarbiausių savybių, apibrėžiančių objektų tarpusavio sąveiką, - tai **paveldėjimas** (*angl. inheritance*).

C++ klasės gali paveldėti kitos klasės ar net kelių klasių duomenis ir metodus. Paveldėjimo vienas iš didžiausių privalumų – tai galimybė ir ateityje panaudoti klasę, šiek tiek ją modifikavus.



Klasė B, kurią paveldi klasė A, vadinama **protėviu** (bazinė klasė). Klasė A, kuri šalia savo duomenų ir metodų paveldi kitą klasę B, vadinama **palikuonimi** (išvestinė klasė). Protėvio duomenys ir metodai pagal nutylėjimą paveldimi kaip *private*, t.y. juos gali naudoti tik palikuonio metodai. Jeigu norima tiesiogiai kreiptis iš išorės į protėvio metodus, reikia nurodyti paveldėjimo atributą *public*.

Išvestinės klasės apibrėžimas

Išvetinė klasė apibrėžiama:

```
class Palikuonis : [identifikatorius] Bazinė_klasė1, [identifikatorius] Bazinė_klasė2
{ ……. };
```

pvz. class Palikuonis : public Bazine
{…..};

[identifikatorius] gali turėti tokias reikšmes: *public, private, protected*. Jis nėra privalomas. Jei identifikatorius praleistas, laikoma, kad jo reikšmė *private*.

Identifikatoriaus paskirtis – nustatyti išvestinės klasės paveldimumo lygį, t.y. kokius metodus ir duomenis gali paveldėti klasė.

Bazinė klasė neturi jokiu teisių į duomenis ir funkcijas, apibrėžtus išvestinėje klasėje.

Pavyzdys

```
#include <iostream.h>
class Skaitliukas
{ protected: int x;
  public:
        Skaitliukas(): x(0)
        { }
        int Getx()
        { return x; }

       void plius()
       {x++;}
};
class MinusSkaitliukas: public Skaitliukas
{ public:
    int minus()
    { return x--;}
};
```

```
main ()
{ MinusSkaitliukas m1;
  cout<<m1.Getx()<<endl;

  m1.plius();
  cout<<m1.Getx()<<endl;
  m1.minus();
  cout<<m1.Getx()<<endl;

return 0; }
```

Paveldėjimas ir pasiekiamumas

Sritys bazinės klasėje	Pasiekimas iš bazinės klasės	Pasiekimas iš išvestinės klasės	Pasiekimas iš išorinių funkcijų
public	Taip	taip	taip
protected	taip	taip	ne
private	taip	ne	ne

Paveldėjimo lygiai

Paveldėjimo identifikatorius	Pasiekimas bazinėje klasėje	Pasiekimas išvestinėje klasėje
public	Public Protected Private	Public Protected nepasiekiami
protected	Public Protected Private	Protected Protected nepasiekiami
private	Public Protected Private	Private Private nepasiekiami

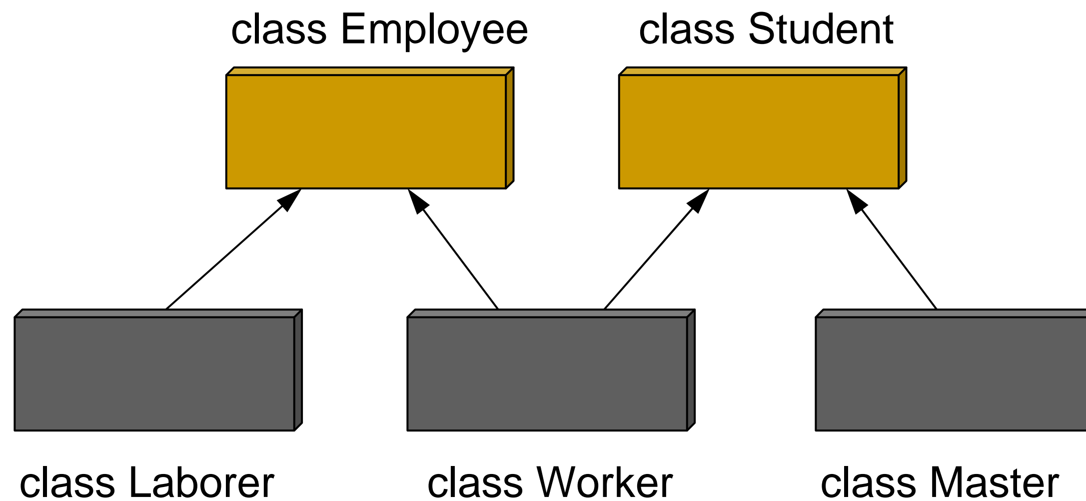
Pavyzdys

```
#include <iostream.h>

class A
{private:  int PrivateData;
 protected: int ProtectedData;
 public: int PublicData;
};

class B : public A
{public: void func()
   { int a;
     a= PrivateData;        // klaida
     a= ProtectedData;     // OK
     a= PublicData;         // OK
     }
};
```

```
class C : private A
{public: void func()
   { int a;
     a= PrivateData;        // klaida
     a= ProtectedData;     // OK
     a= PublicData;         // OK
     }
};
void main()
{ int a;   B objB;  C objC;
  a=objB.PrivateData;      // klaida
  a=objB.ProtectedData;  // klaida
  a=objB.PublicData;        // OK

  a=objC.PrivateData;      // klaida
  a=objC.ProtectedData;  // klaida
  a=objC.PublicData;        // klaida }
```

Nepaveldimi metodai, herarchija

- Konstruktorius
- Kopijos konstruktorius
- Destruktorius
- Priskyrimo operatorius, kurį apibrėžia pats programuotojas
- Draugiškos klasės



Išvestinės klasės konstruktorius

Išvestinė klasė turi apibrėžti savo konstruktorių, nes bazinės klasės konstruktorius nepaveldimas. Išvestinės klasės konstruktorius turi priskirti pradines reikšmės ir bazinės klasės duomenims, ir saviems ir **tai turi padaryti ankščiau nei savo klasės**. Jei turime klasę, kuri paveldi keletą bazinių klasių, tuomet svarbus ir bazinių konstruktorių užrašymo eiliškumas.

Apibrėžiant konstruktorių naudojama sintaksė:

Išvestinės_klasės_konstr(argumentai) : bazinės_klasės_konstr(argumentai1)
{konstruktoriaus tekstas}

Pvz.

```
Sign(int ft, float in, char sig1) : Distance (ft, in)
    {sig = sig1; }
```

Išvestinės klasės konstruktorius

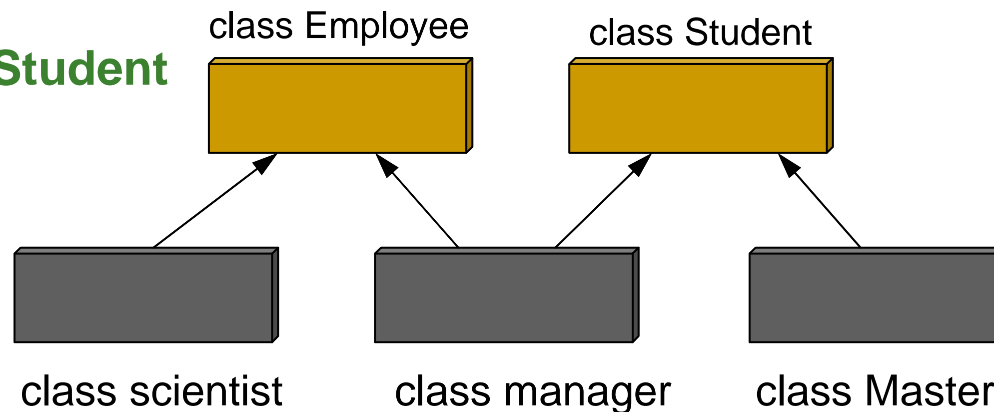
```
#include <iostream.h>

// Klasė Pirmas
   class Pirmas
   { int x;
     public:  Pirmas(int a) { x = a; }
              void Rodo()
              { cout<<"Pirmas::rodo()“<< x<<endl; }
};

// Klasė Antras
  class Antras: public Pirmas
  { int y;
   public:  Antras(int b, int c) : Pirmas(b)
            { y = c; }
            void Rodo()
        {cout<< "Antras::rodo()” << y << endl; }
};
```

```
void main ()
{ Antras B (4, 55);
  B.Rodo();
  B.Pirmas::Rodo();
}
```

Daugybinis paveldėjimas

Jei išvestinė klasė turi kelias bazines klases, turime daugybinio paveldėjimo atvejį.

```
class Student
{ };
class Employee
{ };
class manager : private Employee, private Student
{ };
class scientist : private Employee
{ };
class Master : private Student
{ };
```



Daugybinio paveldėjimo pavyzdys

```
#include <iostream.h>
class Coord
{int x,y;
   public:
   Coord(int x_init, int y_init) {x=x_init;  y=y_init)
   int GetX() {return x;}
   int GetY() {return y;}
};
class SaveMsg
{char Message[80];
 public:
      SaveMsg(char* msg) {strcpy(Message, msg); }
      void Show() { cout<< Message;}
};
class PrintMsg: public Coord, public SaveMsg
{ public: PrintMsg(int _x, int _y, char *msg): Coord(_x, _y), SaveMsg(*msg)
              { }
};
```

Perkrovimas (overloading)

Kodėl naudojamas perkrovimas ?

Funkcijų perkrovimas – tai vienas iš polimorfizmo tipų, naudojamas C++ kalboje. Jei keletas funkcijų turi tą patį vardą, jos laikomos perkrautomis (*overloaded*).

Perkrauti galima tik funkcijas, turinčias skirtingą argumentų skaičių arba jų tipą. Taip pat gali skirtis funkcijų tipas.

Argumentų tipas skiriasi, jei jiems apibrėžti naudojami skirtingi inicializatoriai.

Perkrovimas naudingas tuomet, kai siekiama supaprastinti programą t.y. funkcijoms, atliekančioms panašius veiksmus su skirtingais argumentų sąrašais, suteikti tokius pat pavadinimus.

Kuri iš perkrautų funkcijų bus iškviesta, sprendžia kompiliatorius.

Pavyzdys

```
# include <iostream.h>
// funkcijos abs perkrovimas
int abs (int a);
float abs (float b);
double abs (double c);
void main ()
{  int x = 255; float y = -1.2f; double  z = -15.0;
   cout << abs (x)<< endl;
   cout << abs (y)<< endl;
   cout << abs (z)<< endl;
}
```

```
int abs (int a)
{ return a<0? –a: a; }

float abs (float b)
{ return b<0? –b: b; }

double abs (double c)
{ return c<0? –c: c; }
```

Pavyzdys

```
# include <iostream.h>
// funkcijos repchr() perkrovimas

void repchr();
void repchr(char);
void repchr(char, int);

void main ()
{ repchr();
  repchr('=');
  repchr('+', 30);
}

void repchr()
{ for (int i=0; i<45; i++)
     cout<< '*';
  cout<<end;}

void repchr(char ch)
{ for (int i=0; i<45; i++)
     cout<< ch;
  cout<<end;}

void repchr(char ch, int n)
{ for (int i=0; i<n; i++)
     cout<< ch;
  cout<<end;}
```

Ekrane

```
*********************************************
==============================
++++++++++++++++++++++
```

Funkcijų perkrovimo apribojimai

- Perkraunamos funkcijos turi turėti skirtingus argumentų sąrašus;
- Negalimos perkrautos funkcijos, turinčios tuos pačius argumentų sąrašus ir skirtis tik grąžinamos reikšmės tipu.
- Klasės metodai nelaikomi perkrautais, jei vienas iš jų apibrėžtas kaip *static*, o kitas ne.
- *typedef* operatorius neįtakoja perkrovimo mechanizmui.

```
typedef char* ptr;
void SetVal (char * xx);
void SetVal (ptr xx);
```

Klaida, nes tai neperkrautos funkcijos (argumentų sąrašas tas pats)

Klasės metodų perkrovimo pavyzdys

```
class Lapas
{  private: int x; float y; char z;
   public:
   Lapas (int A, float B, char C) { x = A; y = B; z = C; }   // Konstruktorius
   Lapas() { x = 0; y = 0; z = 'z'; }                        // Konstruktorius
   void Rodo (char *);
     void Suma (int Sk)     { x = x + Sk; }
     void Suma (float Sk)  { y = y + Sk; }
     void Suma (char Sk)   { z = Sk; }
     void Suma (int A, char B) { x = x + 2.0 * A;  z = B + 4; }
};

void Lapas::Rodo(char *Eilute) {
cout << Eilute << " : ";
cout << " x= " << x << " y= " << y << " z= " << z <<endl; }
```

perkrautos funkcijos

Tęsinys

```
void main(void)
{
  Lapas A;
  Lapas B(5, 3.4, 'C');
  A.Rodo("Objektas A-1");
  B.Rodo("Objektas B-1");

  A.Suma(5);
  B.Suma('K');
  A.Rodo("Objektas A-2");
  B.Rodo("Objektas B-2");

  A.Suma((float)35.25);
  B.Suma(5, 'A');
  A.Rodo("Objektas A-3");
  B.Rodo("Objektas B-3");
  getch();
}
```

Klasės metodų perkrovimas

Perkrauti klasės metodai gali būti apibrėžti skirtingose klasės pasiekia-mumo dalyse (pvz. vienas *private*, kitas *public* dalyje). Tačiau tai nekeičia metodų perkrovimui galiojančių taisyklių.

```
class AnyClass
{   private :
    int Func (int a)
      { …. }
    public:
    AnyClass();
    double Func (double b, char * c)
     { ….}
};
```

```
main()
{ AnyClass * ptr = new AnyClass;
  ptr -> Func(104);        //blogai
  ptr -> Func(12.2, "Eilute");
  return 0;
};
```

Konstruktorių perkrovimas

Dažniausiai funkcijų perkrovimas naudojamas kuriant perkrautus konstruktorius. Taip daroma siekiant suteikti galimybę kurti objektus, naudojant skirtingus argumentų sąrašus.

```
Class Rect
{ private: int x,y,w,h;
  public: Rect()  {x=y=w=h=0; }
          Rect(int a, int b)  { x = a; y = b; w=h=10; }
          Rect(int a, int b, int c, int d)  { x = a; y = b; w = c; h=d; }
          Rect(const Rect&);
          Rect(const Rect&, int, int );
};
Rect::Rect(const Rect& rc)

{x=rc.x;   y=rc.y;
 w = rc.w; h = rc.h }
```

Konstruktorių perkrovimas

```
Rect::Rect( const Rect& rc, int _x, int _y)
{ x = _x;
  y = _y;
  w= rc.w;
  h = rc.h;
 }

void main ()
{ Rect ab;
  Rect bc(10, 20);
  Rect rc(3,4,5,6);
  Rect newrc(rc, 14, 34);
}
```

Operatorių perdengimas

Programavimo kalbose naudojami operatoriai pasižymi polimorfizmu (daugiavariantiškumu). Kaip pavyzdys gali būti operatorius **+**, kuris taikomas tiek *int*, tiek *float* tipo kintamiesiems, nors sudėties veiksmai procesoriaus registruose atliekami nevienodai.

Klasėse galima naujai apibrėžti operatorius, t.y. pakeisti jų veikimą. Toks naujas operatorių perorientavimas yra vadinamas **perdengimu (perkrovimu).** Operatorių aprašai nuo funkcijų skiriasi vartojamu žodžiu operator.

Operatoriaus perdengimo sintaksė :

[tipas] ***operator*** *<Operatoriaus_simbolis> (parametrai);*

Perdengimas draudžiamas operatoriams: . :: ?:

Operatorių perdengimas gali galioti lokaliai t.y. klasės ribose arba globaliai t.y. visoje programoje.

Operatoriai – tai funkcijos, todėl operatorių perkrovimas siejamas su funkcijų perkrovimu.

Pavyzdys

```
#include <iostream.h>
class Counter
{ private:   unsigned int count;
  public:    Counter(): count(0)
             {  }
             unsigned int Rodo()
              { return count;  }
             void operator ++ ()                // objektas dešinėje ++ pusėje
             {  ++ count; }
};
void main()
{Counter c1, c2;
 cout<< "c1="<<c1.Rodo()<<endl;             // Ekrane ->  c1=0
 cout<< "c2="<<c2.Rodo()<<endl;             // Ekrane ->  c2=0
 ++c1;
 ++c2;
 cout<< "c1="<<c1.Rodo()<<endl;             // Ekrane ->  c1=1
 cout<< "c2="<<c2.Rodo()<<endl;  }          // Ekrane ->  c2=1
```

Operatorių perdengimas

```
class Katinas
{  public:
  Katinas(char *);
  void operator + (char *);
  void operator - (char );
  void Rodo();
 ~Katinas();
 private:
 char *Vardas;
};
```

```
Katinas::Katinas(char *Vardas) {
      Katinas::Vardas = new char[50];
      strcpy(Katinas::Vardas, Vardas); }

void Katinas::Rodo()
      { cout << Vardas << endl; }

void Katinas::operator + (char * A)
      { strcat(Vardas, A); }

void Katinas::operator - (char C) {
      for(int i=0; *(Vardas+i) != '\0';) {
              if(*(Vardas + i) == C)
          for(int j=i; j<(strlen(Vardas)-1); j++)
               *(Vardas + j) = *(Vardas + j+1);
       i++; }
  }
```

Operatorių perdengimas

```
Katinas::~Katinas()
{ delete Vardas; }

// Pagrindinė programa
void main(void)
{
   Katinas A ("Batuotas ir piktas");
   A.Rodo();
   A + " Katinas! ";         // Ekrane ->  Batuotas ir piktas Katinas
   A.Rodo();
   A - 'a';                  // Ekrane -> Btuots ir pikts Ktins
   A.Rodo();

}
```

Perdengiami operatoriai

Operatorių perdengimas – tai funkcijos operatoriai, kurių pavadinimai siejami su raktiniu žodžiu *operator* ir po juo einančiu operatoriumi.

Taisyklė:

Funkcijos-operatoriai turi būti nestatiniai klasės metodai arba turėti klasės tipo argumentą arba nuorodą į klasę.

Sintaksė:

Binariniai operatoriai (*turi du operandus: +, -, *, /, %, ….*)

[tipas] ***operator*** *<Operatoriaus_simbolis> (par1);* *(lokalus)*

[tipas] ***operator*** *<Operatoriaus_simbolis> (par1, par2);* *(globalus)*

Unariniai operatoriai (*turi tik vieną operandą ++, --, +=, -=, ….*)

[tipas] ***operator*** *<Operatoriaus_simbolis> ();* *(lokalus)*

[tipas] ***operator*** *<Operatoriaus_simbolis> (par1);* *(globalus)*

Perdengiami operatoriai

Unary:	new	*	!	~	&	++	--	()	->	->*	
	delete										
Binary:	+	-	*	/	%	&	|	^	<<	>>	
	=	+=	-=	/=	%=	&=	|=	^=	<<=	>>=	
	==	!=	<	>	<=	>=	&&	||	[]	()	,

Pavyzdys (lokalus perdengimas)

```
class Point {
public:
        Point (int a, int b)
            {Point::x = a; Point::y = b;}
        Point operator + (Point &p)
            {return Point(x + p.x, y + p.y);}
        Point operator - (Point &p)
            {return Point(x - p.x, y - p.y);}
private:
        int x, y;
};

void main () {
Point p1(10,20), p2(10,20);
Point p3 = p1 + p2;
Point p4 = p1 - p2;
}
```

Pavyzdys (globalus perdengimas)

```
class Point {
public:
        Point (int a, int b)  {Point::x = a; Point::y = b;}
        friend Point operator + (Point &p, Point &q)  //globalus operatrorius
                {return Point(p.x + q.x, p.y + q.y);}
        friend Point operator - (Point &p, Point &q)
                {return Point(p.x - q.x, p.y - q.y);}
private:
        int x, y;
};
```

Perkrauto operatoriaus naudojimas – ekvivalentiškas funkcijos iškvietimui. Pavyzdžiui:

```
operator+(p1, p2);                    //  tas pats: Point p3=p1 + p2;
```

Inkremento operatorių perdengimas

```
#include <iostream.h>

class Counter
{ private:   unsigned int count;
  public:    Counter(): count(0)
              {  }
             Counter(int c): count(c)
              { }
             unsigned int Rodo()
             { return count;  }
             Counter operator ++ ()                    // prefiksas
             {  return Counter (++ count); }          // didina ir priskiria

             Counter operator ++ (int)                 // postfiksas
             {  return Counter( count++); }           // priskiria ir didina
};
```

Inkremento operatorių perdengimas

```
void main()
{Counter c1, c2;
 cout<< "c1="<<c1.Rodo()<<endl;        // c1=0
 cout<< "c2="<<c2.Rodo()<<endl;        // c2=0
 ++c1;
 c2=++c1;

 cout<< "c1="<<c1.Rodo()<<endl;        // c1=2
 cout<< "c2="<<c2.Rodo()<<endl;        // c2=2

c2=c1++;
cout<< "c1="<<c1.Rodo()<<endl;         // c1=3
cout<< "c2="<<c2.Rodo()<<endl;         // c2=2

}
```

Palyginimo operatorių perdengimas

```
#include <iostream.h>
class Distance
{ private:   int metrai, centimetrai;
  public:    Distance(): metrai(0), centimetrai(0)    {  }
             Distance(int m, int cm): metrai(m), centimetrai(cm)     {   }
             void ShowDist ( )
             { cout<< metrai<<"   "<<centimetrai<<endl;
             bool operator < (Distance) const;
 };

bool Distance::operator < (Distance d2) const;
{ int sum1 = metrai*100 + centimetrai;
  int sum2 = d2.metrai*100 + d2.centimetrai;
  return (sum1 < sum2) ? true : false;
}
```

Palyginimo operatorių perdengimas

```
// Pagrindinė programa

void main()
{  Distance dist1;
   Distance dist2(12, 44);
      cout<< “dist1 “; dist1.ShowDist();
      cout<< “dist2 “; dist2.ShowDist();
  if (dist1< dist2)
        cout<< “dist1 mažiau nei dist2 \n”;
 else
     cout<< “dist1 daugiau nei dist2 \n”;
}
```

[ ] perkrovimas (pavyzdys 1)

```
#include <iostream.h>
#include <process.h>              // dėl funkcijos exit()
const int LIMIT = 100;
class masyvas {
    private:  int arr[LIMIT];
    public:   int& operator [ ] (int n)
                 {   if (n < 0 || n  >= LIMIT )
                           { cout << "Indekso numeris neteisingas\n"; exit(1); }
                     return arr[n];
                  }
};
      void main()
      { masyvas sa1;
          for (int j=0; j <LIMIT; j++)
          {   sa1[ j ] = j+10;
              cout << "Elementas " << j << "lygus " << sa1[ j ]<<endl;
          }  }
```

Virtualumas

Kam reikalingos virtualios funkcijos?

C++ virtualumas suprantamas, kaip polimorfizmo ir paveldėjimo savybių apjungimas.

Iki šiol nagrinėtos perkrautos funkcijos ar operatoriai buvo nesiejami su klasių herarchija.

Virtualios funkcijos naudojamos tuomet, kai tuo pačiu pavadinimu egzistuoja funkcijos tiek bazinėje tiek ir išvestinėse klasėse.

Virtuali – reiškia neegzistuojanti realybėje, o tai suprantama, kad kviečiant funkciją iš vienos klasės, realiai vykdoma kitoje klasėje esanti funkcija.

Pavyzdys:

```
shape* ptr_array [100];           // 100 rodyklių masyvas į skirtingus objektus
for (int i=0; i<N; i++)
    ptr_array[ i ] -> draw( );    // ta pati funkcija draw() kviečiama iš
                                  //    skirtingų objektų
```

Problema…nr.1

```
BankAccount bretta;         // base-class object
Overdraft ophelia;          // derived-class object
bretta.ViewAcct();          // use BankAccount::ViewAcct()
ophelia.ViewAcct();         // use Overdraft::ViewAcct()
```

→ Vienareikšmiš-kai apibrėžta

```
BankAccount bretta;
BankAccount * bp = &bretta;     // points to BankAccount object
bp->ViewAcct();                 // use BankAccount::ViewAcct()
bp = &ophelia;                  // BankAccount pointer to Overdraft object
bp->ViewAcct();                 // which version of ViewAcc?
```

↓ Atsakymas

Pagal nutylėjimą, C++ naudoja rodyklės arba nuorodos tipą, kad nuspręstų, kokią f-ją pasirinkti. *Todėl bus naudojama BankAccount::ViewAcct()*

Problema…nr.2

Tačiau kompiliatorius kompiliavimo metu dažnai nežino, su kokiu objektu bus surišta nuoroda ar rodyklė. Pavyzdžiui:

```
cout << "Enter 1 for Brass Account, 2 for Brass Plus Account: ";
int kind;
cin >> kind;
BankAccount * bp;
if (kind == 1)
    bp = new BankAccount;
else if (kind == 2)
    bp = new Overdraft;
bp->ViewAcct();             // neaišku, kuri funkcija bus iškviesta
```

Dinaminio pririšimo (dynamic binding) aktyvizavimas

Dinaminis pririšimas aktyvizuojamas tik klasės metodams. Norint tai atlikti **bazinėje funkcijoje perkrauta funkcija apibrėžiama kaip virtuali** (naudojamas raktinis žodis **virtual**).

Jei metodas apibrėžtas kaip virtualus, jis toks išlieka visose paveldėtose (išvestinėse) klasėse.

Vienam virtualiam metodui raktinis žodis naudojamas tik vieną kartą ir tik bazinėje klasėje.

Pavyzdys

```
// bankacct.h – a simple BankAccount class with virtual functions
#ifndef _BANKACCT_H_
#define _BANKACCT_H_

class BankAccount  {
private:  const int MAX = 35;
          char fullName[MAX];
          long acctNum;
          double balance;

public:   BankAccount (const char *s = "Nullbody", long an = -1, double bal = 0.0);
          void Deposit (double amt);
          virtual void Withdraw (double amt);        // virtual method
          double Balance() const;
          virtual void ViewAcct() const;              // virtual method
};
```

Pavyzdys (tęsinys)

```
class Overdraft : public BankAccount {
private:  double maxLoan;
          double rate;
          double owesBank;

public:   Overdraft (const char *s = "Nullbody", long an = -1, double bal = 0.0,
                     double ml = 500, double r = 0.10, owes = 1.0);
          Overdraft (const BankAccount & ba, double ml = 500, double r = 0.1,
                     owes = 1.0);
          void ViewAcct() const;
          void Withdraw(double amt);
          void ResetMax(double m) { maxLoan = m; }
          void ResetRate(double r) { rate = r; }
          void ResetOwes() { owesBank = 0; }
};

#endif
```

Pavyzdys (tęsinys)

```
int main() {
    BankAccount * baps[10];
    char name[MAX];
    long acctNum;  double balance;  int acctType;
for (int i = 0; i < 10; i++) {
   // su cin >> nuskaitomi name, acctNum, balance, acctType

if (acctType == 2)
            baps[i] = new Overdraft(name, acctNum, balance, max, rate, ow );
else
           { baps[i] = new BankAccount(name, acctNum, balance);
                 if (acctType != 1)
                        cout << "I'll interpret that as a 1.\n"; }
}
for (i = 0; i < 10; i++)
    { baps[i]->ViewAcct();
      cout << endl; }
 return 0; }
```

Virtualios ir nevirtualios funkcijos pavyzdys

```
#include <iostream.h>
class Base
{ public:  virtual void VirtFunc()
           { cout << "Base::VirtFunc()\n"; }
           void show()
           {cout << "Base klases funkcija \n"; }
};
class Derived: public Base
{ public:  void VirtFunc()
           { cout << "Derived::VirtFunc()\n"; }
           void show()
           { cout << "Derived klases funkcija \n";}
};
```

Virtualios ir nevirtualios funkcijos pavyzdys

```
void main ()
{Derived A_isvestine;
 Derived *ptr_isvestine = &A_isvestine;

 Base  *ptr_base = &A_isvestine;
 ptr_base->VirtFunc();                // virtualios funkcijos iškvietimas
 ptr_base->show();                    // nevirtuali funkcija


 ptr_isvestine->VirtFunc();            // virtualios funkcijos iškvietimas
 ptr_isvestine->show();                // nevirtuali funkcija

}
```

Daugybinis paveldėjimas, virtualios bazinės klasės

Siekiant išvengti dvigubo bazinės klasės įtraukimo, naudojamos virtualios bazinės klasės. Tam prieš paveldimumo identifikatorių rašomas žodis *virtual*.

```
class IndBase
{   int x;
    public:
    int GetX() {return x; }
    void setX (int _x) { x=  _x ;}
    double var;
};
Class Base1: virtual public Indbase
{ …. };
Class Base2: virtual public Indbase
{ …. };
```

```
Class Derived: public Base1, public Base2
{ …. };

main ()
   Derived ob;
   ob.var = 5.0
   ob.SetX(0);
   int z = ob.GetX();
return 0; }
```

Abstrakčios klasės ir *pure virtual* funkcijos

Egzistuoja klasės, kurios nėra naudojamos objektams kurti, o tik tam, kad sudarytų reikiamą klasių herarchijos medį ir užtikrintų logišką paveldimumą. Tokios klasės vadinamos abstrakčiomis (*abstract class*).

Pavyzdžiui: bazinė klasė *forma* ir klasės *trikampis*, *apskritimas*, *keturkampis* ir t.t

Akstrakti klasė turi bent vieną virtualią funkciją, o tiksliau tikrą virtualią (*pure virtual*). Išvestinės klasės turi būtinai panaudoti abstrakčios klasės virtualias funkcijas, priešingu atveju jos taip pat tampa abstrakčiomis (dėl paveldimumo).

Tikrai virtualia funkcija vienareikšmiškai pasako, kad klasė, kuriai ji priklauso yra abstrakti ir objektas iš tokios klasės negali būti sukurtas.

Tikrai virtutalios funkcijos sintakse:

```
virtual <funkcijos_pavadinimas> (argumentai) = 0;
```

Pure virtual funkcijos pavyzdys

```
class Base
{ int x;
  public: Base( int xx) {x = xx;}
  virtual int GetX()      {return x;}
  virtual void PrintX () = 0;
};
class Derived: public Base
{ public: Derived(int xx): Base (xx)
              { }
             int GetX() { return 0;}
             void PrintX()
             {cout<< "x= " <<GetX()<<"  \n"; }
};
```

Šablonai

Kodėl šablonai (templates)?

Programuojant egzistuoja situacijos, kai reikia atlikti tuos pačius veiksmus su skirtingais duomenų tipais (pvz. modulio radimas, kėlimas laipsniu).
Tam patogu naudoti **šablonines** arba **parametrizuotas** funkcijas ir klases.

- Šablonai leidžia sukurti bendro pobūdžio funkcijas ir klases, neprisirišant prie konkrečių duomenų tipų.
- Leidžia kurti bendro tipo bibliotekas (STL) ir taip užtikrina daugkartinį funkcijų ar klasių panaudojimą.
- Šablonų tipai : funkcijų šablonai, klasių šalonai.

Pavyzdys:

```
int abs (int n)
    {return (n<0) ? –n : n;
    }
```

C kalboje egzistuoja abs(); fabs(); fabsl(); labs(), cabs()...

Funkcijų šablonai (function template)

Funkcijos šablonas visad apibrėžiamas raktiniu žodžiu **template** ir vienu ar keliais prametrais .

```
template <class T> Max (T, T);

template <class T> T abs(T n)
{return (n<0)? –n, n;}
```

```
template <class A> A Sqr (A skc)
{ return skc * skc;  }
```

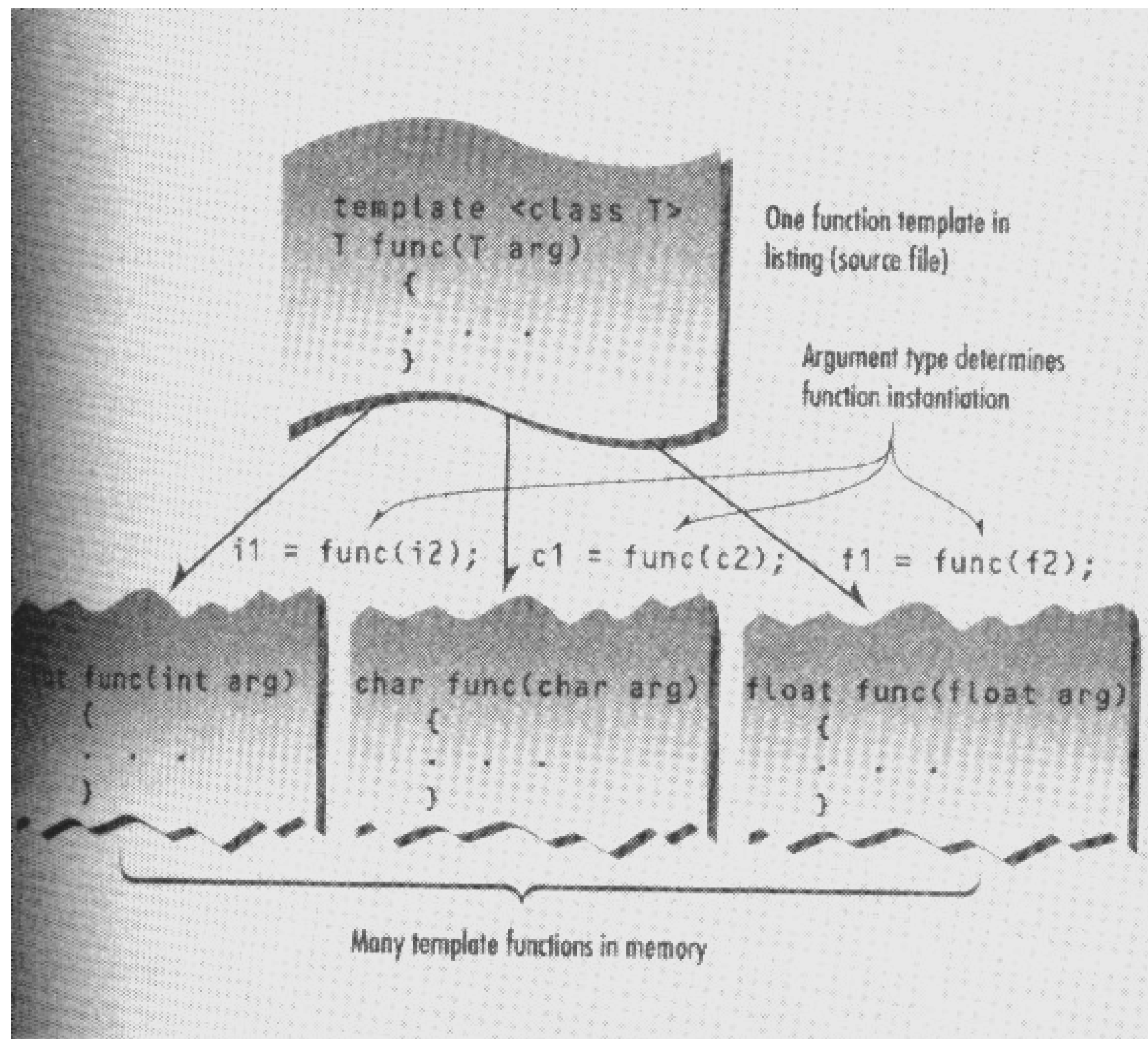
template <*typename* identifier> function_declaration
template <*class* identifier> function_declaration

Sintaksė

Argumentai, esantys <> viduje vadinami šablono argumentais.
class – raktinis žodis. Jis būtinas prieš kiekvieną argumentą!!!

Kompiliatorius kuria konkrečią funkcijos realizaciją (šabloninę funkciją – template function) programos vykdymo metu.
Kaip atrodys konkreti funkcija, priklauso nuo argumentų tipo.

Funkcijų šablonai



Funkcijų šablonai su keletu argumentų

```
template <class T1, class T2, class T3>
        T3 Relation(T1, T2);                  // ok
template <class T1, class T2>
        int Compare (T1, T1);                 // blogai! T2 neneaudojamas.
template <class T1, T2>                       // blogai! class praleistas dėl T2
        int Compare (T1, T2);
```

Pavyzdys

```
template <class T>
T Max (T val1, T val2)
{
        return val1 > val2 ? val1 : val2;
}
cout << Max(19, 5) << ' ' << Max(10.5, 20.3) << ' ' << Max('a','b') << '\n'
```

Pavyzdys

```
#include <iostream.h>

template <class atype, class btype>
int find(atype* array, atype value, btype size)
{ for (btype j=0; j<size; j++)
    if (array[ j ] == value)
        return j;
    return -1; }

void main( )
{int size = 5;
 char chArr [ ] = {1, 2, 3, 4, 5};      char ch = 5;
 int intArr [ ] = {1, 2, 3, 4, 5};         int in = 4;
 double douArr[ ] = {1.0, 2.0, 3.0, 4.0, 5.0 };
 double db = 4.0;

 cout << “char” << find (chArr, ch, size)<<endl;
 cout << “int ” << find (intArr, in, size)<<endl;
 cout << “char ” << find (douArr, db, size)<<endl; }
```

Klasių šablonai

Šablonų koncepsija gali būti išplėsta pritaikant ją klasėms. Šablonai naudojami specialioms klasėms, kurios vadinamos duomenų (container) klasėms (pvz. stekams (stacks) ir sujungtiems sąrašams (linked lists)).

Sintaksė:

```
template <class Tipas> class klasės_pavadinimas
{  };
```

```
template <class Type>
class Stack
 { private: Type st[10];
                 int top;
  public: Stack() {top = -1;}
  void push (Type var) { st[++top] = var; }
  Type pop () { return st [top--]; }
 };
```

```
void main()
{Stack<float> s1;
  s1.push(11.1F);
  cout<<s1.pop()<<endl;
  Stack<long> s2;
  s2.push(123L);
  cout<<s2.pop()<<endl;
}
```

Klasių šablonai

```
#include <iostream>
using namespace std;

template <class T>
class pair {
    T a, b;
  public:
    pair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T pair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval; }
```

```
int main () {
  pair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

Klasių šablonai

```
template <class Type>
class Stack
{ private:
            Type *stack;                    // steko masyvas
            int top;                        // steko indeksas
            const int maxSize;              // maksimalus steko dydis
  public:
    Stack (int max) : stack(new Type[max]), top(-1), maxSize(max)
    { }
   ~Stack (void)
     { delete [ ] stack; }

    void Push (Type &val);
    void Pop (void)   {  if (top >= 0) --top;}
    Type& Top (void) {return stack[top];}
    friend ostream& operator << (ostream&, Stack&);
};
```

Klasių šablonai

Jei klasės funkcija apibrėžiama už klasės ribų naudojama sekanti sintaksė:

```
template <class Type>
void Stack<Type>::Push (Type &val)
{
        if (top+1 < maxSize)
                stack[++top] = val;
}

template <class Type>
ostream& operator << (ostream& os, Stack<Type>& s)
{
        for (int i = 0; i <= s.top; ++i)
                os << s.stack[i] << " ";
        return os;
}
```

Klasių šablonai

```
#include <iostream.h>
using namespace std;
template <class Type>
class Stack
 { private:  Type st [10];
             int top;
  public:   Stack() { top = -1; }                    // Konstruktorius
             void push (Type var) { st [ ++top ] = var; }
             Type pop () { return st [top -- ]; }
 };

  void main()
  { Stack <float> s1;                        // s1 – klasės Stack<float> objektas
    s1.push(11.1F);  s1.push(22.2F);
    cout<<s1.pop()<<endl;
    cout<<s1.pop()<<endl;
    Stack <long> s2;
    s2.push(123L);     cout<<s2.pop()<<endl;
   }
```

Šablonų apibendrinimas

- Šablonai – tai C++ savybė, leidžianti kurti parametrizuoto tipo funkcijas ir klases, kurių pagalba galima kurti klases ar funkcijas keičiančias savo elgseną priklausomai nuo parametrų. Šablonai – tai galimybė pakartotinai naudoti jau sukurtus programinius kodus.
- Šablonus apibrėžia parametrizuoti kintamieji.
- Klasių šablonai gali turėti draugiškas klases ir funkcijas bei klasių ir funkcijų šablonus.
- Šablonai gali turėti ir statinius kintamuosius. Tuo atveju kiekviena šablono konkretizacija kuria atskirus savo statinių duomenų rinkinius.

I/O srautai

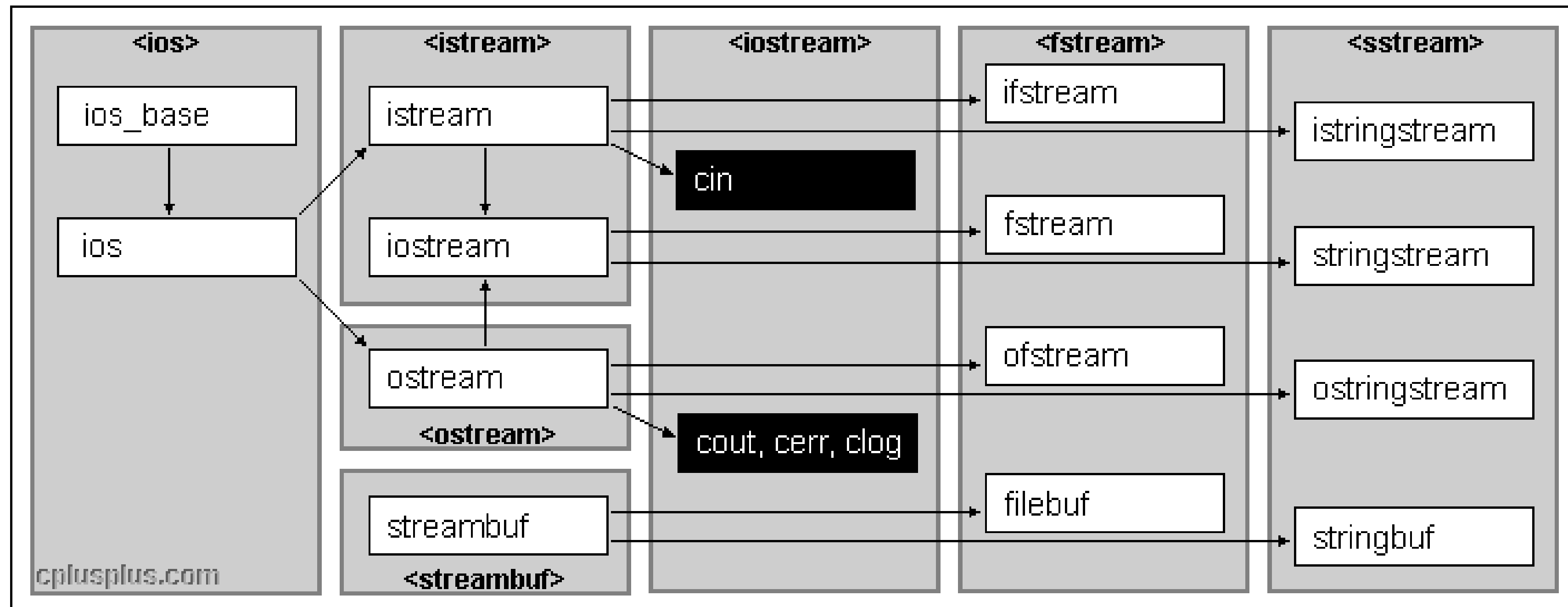
I/O biblioteka

- I/O srautas – tai loginis įrenginys informacijos gavimui iš vartotojo ir informacijos perdavimui vartotojui.
- I/O srautas siejamas su fiziniais įrenginais (ekranu, klaviatūra, pele, HDD) per C++ I/O sistemą t.y. I/O biblioteką, vadinamą iostream.
- Tokia I/O sistema užtikrina vieningą tvarką informacijos įvedimui/išvedimui.
- I/O biblioteka suskirstyta į 4 failus:
  - iostream.h
  - fstream.h
  - strstream.h
  - iomanip.h

C ir C++ srautų palyginimas

Srautas	C++	C
Įvedimo	cin	stdin
Išvedimo	cout	stdout
Klaidų	cerr	stderr
Registravimo	clog	stderr

Srautų herarchija



Strautų bibliotekų failai

Antraštės failas	Aprašymas
iostream.h	Defines a hierarchy of classes for low-level (untyped character-level) I/O and high-level (typed) I/O. This includes the definition of the *ios, istream, ostream,* and *iostream* classes.
fstream.h	Derives a set of classes from those defined in iostream.h for file I/O. This includes the definition of the *ifstream, ofstream*, and *fstream* classes.
strstream.h	Derives a set of classes from those defined in iostream.h for I/O with respect to character arrays. This includes the definition of the *istrstream*, *ostrstream*, and *strstream* classes.
iomanip.h	Defines a set of manipulator which operate on streams to produce useful effects.

iostream bibliotekos panaudojimas

I/O rūšis	Įvedimas	Išvedimas	Įvedimas ir išvedimas
Standard I/O	istream	ostream	iostream
File I/O	ifstream	ofstream	fstream
Array of char I/O	istrstream	ostrstream	strstream

Apibrėžti standartiniai srautai

Srautas (stream)	**Tipas**	**Buferi-zuotas**	**Aprašymas**
cin	istream	Taip	Siejamas su standartiniu įvedimu (klaviatūra)
cout	ostream	Taip	Siejamas su standartiniu išvedimu (monitoriumi)
clog	ostream	Taip	Siejamas su standartiniu log'ų išvedimu (monitorius)
cerr	ostream	Ne	Siejamas su standartiniu klaidų išvedimu (monitorius)

I/O operatoriai

- Įvedimo >>
- Išvedimo <<

Pavyzdys

```
#include <iostream.h>
void main()
{
   float A;
   cout << “Iveskite A reiksme: " ;
   cin >> A;
   cout << “\nA=“<<A<<endl;
}
```

```
#include <iostream>
using namespace std;
void main()
  {  float A = 18.236;
     cout << "1. A="  << A << endl;
     cout << "2. A=" <<  A*2.0 <<
     endl;
  }
```

Srauto formatavimo manipuliatoriai

C++ strautai formatuojami: *manipuliatoriais*, *žymėmis*, *funkcijomis*.

Manipuliatoriai naudojami norint formatuoti išvedamą/įvedamą informaciją.

- **setw (int** n**)** n **–** lauko plotis simboliais išvedimui,
- **setprecision (int** n**)** n **–** skaitmenų skaičius, išvedant trupmeninį skaičių.
- **setfill(char c**) c - tarpo užpildymo simbolis.
- **endl** uždedamas naujos eilutės simbolis
- **ends** uždedamas eilutės pabaigos simbolis

Jei naudojami manipuliatoriai su argumentais, įtraukiamas **iomanip.h failas**.

Jei *setw()* nurodyto lauko dydžio skaičiui nepakanka, manipuliatorius ignoruojamas. *setw()* galioja artimiausiai išvedamai reikšmei, o *setprecision()* – iki naujo nurodymo.

Pavyzdys

```
#include <iostream>
#include <iomanip>
using namespace std;
        void main()
        {
        float A = 18.2345f;
        cout << setfill('0');
        cout << "1. A=" << setw(9) << A << endl;     //  0018.2345
        cout << "2. A=" << setprecision(3) << A << endl;  // 18.2
        cout << "3. A="<< setw(10) << setprecision(5) << A <<endl;
                //000018.234

        A = 123.45678f;
        cout << "4. A=" << A << endl;  //123.46
        A = 12345678f;
        cout << "5. A=" << A << endl;  //1.2346e+07
        }
```

Srauto formatavimo funkcijos (width)

- Klasės ostream metodas width(int) skirtas nurodyti išvedamo sraute duomenų laukams. Galioja artimiausiam dydžiui. Tai *setw()* analogas

```
#include <iostream.h>
#include <iomanip.h>
void main() {
    for(int i=2; i<6; i++) {
        cout.width(3);
        cout << "i=";
        cout.width(i);
        cout << i << endl; }
}
```

Srauto formatavimo funkcijos (fill)

Metodas fill(int) leidžia pakeisti užpildymo simbolį norimu. Manipuliatoriaus *setfill()* analogas.

```
#include <iomanip.h>
#include <iostream.h>
void main() {
   cout.fill('.');
   for(int i=2; i<6; i++) {
            cout << "i=";
            cout.width(i);
            cout << i << endl; }
 }
```

Ekrane

i=.2
i=..3
i=...4
i=....5

Srauto formatavimo funkcijos (precision)

Metodas precision(int) leidžia nustatyti, kiek skaičių bus išvesta, išvedant trupmeninį skaičių.

Pagal nutylėjimą išvedami **6** skaičiai.

```
#include <iostream>
#include <cmath>
using namespace std;
void main() {
double x;
cout.precision(4);
cout.fill('0');
```

```
cout << "   x    sqrt(x)     x^2\n\n";
  for(x = 1.0; x <= 6.0; x++) {
          cout.width(7);
          cout << x<< "  ";
          cout.width(7);
          cout << sqrt(x) << "   ";
          cout.width(7);
          cout << x*x << endl; }
}
```

Failų atidarymo rėžimai

Rėžimas	Aprašymas
ios::in	Failas atidaromas skaitymui
ios::out	Failas atidaromas rašymui
ios::ate	Failas atidaromas rašymui, išvedimas pradedamas nuo failo galo (at end)
ios::app	Failas atidaromas papildymui
ios::trunc	Naikinamas failo turinys
ios::binary	Failas atidaromas dvejatainiame režime

Srautų sintaksė

- **ifstream (const char *name, ios::openmode = ios::in);**

  (nuskaitymo iš failo srautas)

- **ofstream (const char *name, ios::openmode = ios::out | ios::trunc);**

  (*rašymo į failą srautas*)

- **fstream (const char *name, ios::openmode = ios::in | ios::out);**

  (*rašymo ir skaitymo iš/į failą srautas*)

Pavyzdžiai

```
#include <fstream>
using namespace std;
void main() {
ofstream fs;
fs.open(“File.dat”, ios::app);          //ofstream fs("File.dat", ios::app);

if ( ! fs )
    cout<< “Failo nepavyko atidaryti”<<endl;

if ( fs.is_open() )                     // is_open() grąžina true/false
    cout<< “Failo nepavyko atidaryti”<<endl;
}
```