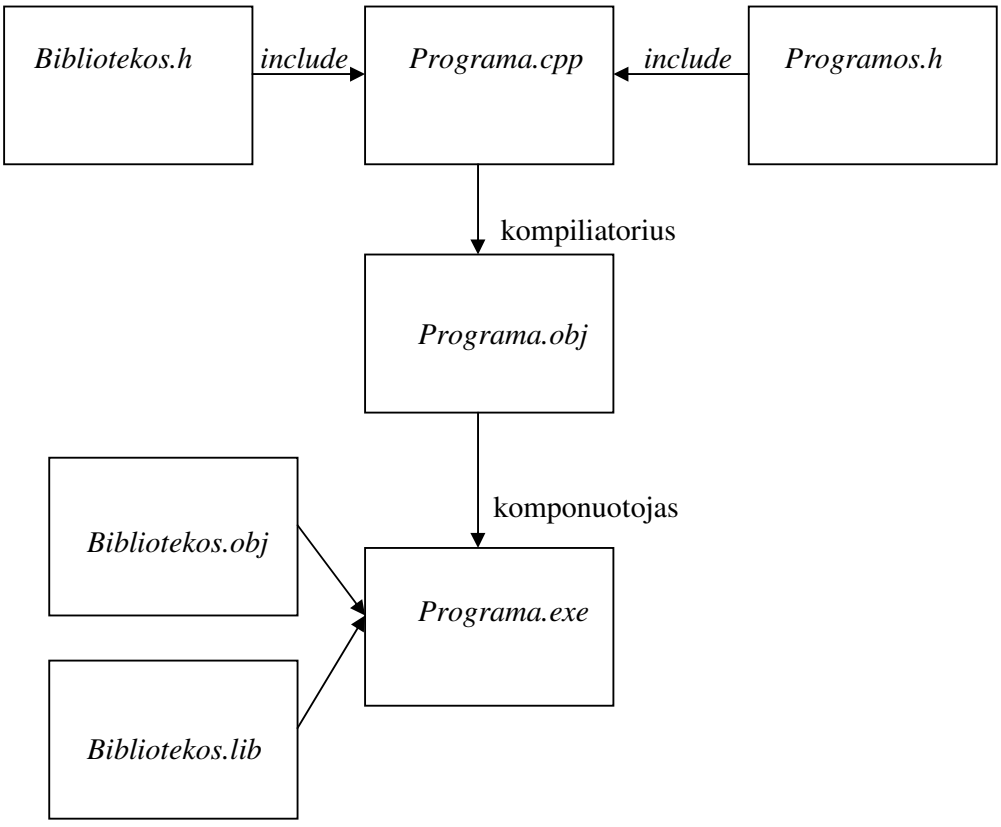
12-13 PASKAITOS

Turinys:
Daugiafailės programos. Tarpfailiniai ryšiai. Antraštiniai failai.
Vardų sritys.
Duomenų tipų pervardinimas.

Daugiafailės programos

Daugiafailė programa sudaryta iš kelių failų. Tai leidžia sudalyti sudėtingos programos rašymą keliems programuotojams bei naudotis klasių bibliotekomis (kad ir svarbiausia iš bibliotekų – STL (Standard Template Library)).
Paprastai biblioteką sudaro sąsaja (interface) ir realizacija (implementation). Sąsajos informacija: klasių skelbimai (laukai ir metodai); be tokios informacijos naudotis klase neįmanoma. Informacija sudedama į antraštinius failus, prie programos prijungiamus direktyva *include*. Realizacija programuotojui paprastai neprieinama; ji teikiama **.obj* arba **.lib* failų pavidalu.
Principinė programos schema:



include direktyvas galima teikti ir kabutėse:

#include "Bibliotekos.h"

Tai reiškia, kad kompiliatorius **.h* failo ieškos darbiniame aplanke, o ne pajungiamų failų aplanke. Patogiausia visus failus (*Programa.cpp, Bibliotekos.obj, Bibliotekos.h*) sudėti į tą pat aplanką. Bibliotekinius **.h* failus (pavyzdžiui, *iostream* ar *string*) kompiliatorius DeveloperStudio terpėje saugo aplanke INCLUDE, esančiame aplanke Microsoft Visual Studio\VC<+kompiliatoriaus išleidimo metai>.
Daugiafailes programas programavimo terpė saugo vadinamuosiuose projektuose (MS DeveloperStudio jų failams teikia plėtinius **.dsp*). Projektas saugo visus reikiamus failus ir saistymo instrukcijas. Į projektą **.cpp, *.lib* ir **.obj* failus įjungti galima rankiniu būdu ar automatiškai. Atnaujindama projekto failą, programavimo terpė perkompiliuoja tik keistus pradinius failus **.cpp*.

Tarpfailiniai ryšiai

Failai kompiliuojami atskirai, o vėliau susiejami kartu. Kaip tarp atskirų programos failų siesis programos kintamieji, funkcijos ir klasės?

1. Kintamieji.

Lokalieji programos kintamieji matomi tik savo failo atitinkamoje matomumo srityje. Globalieji kintamieji gali būti matomi visuose failuose. Tokie kintamieji turi būti skelbiami visuose failuose, o apibrėžiami tik viename kuriame nors faile.
Toks skelbimas ir apibrėžimas

```
//failas1.cpp                    //failas2.cpp
. . .                            . . .
int globalVariable;              int globalVariable;
. . .                            . . .
```

– klaidingas – saistymo klaida – keletas apibrėžimų.

Toks būdas

```
//failas1.cpp                    //failas2.cpp
. . .                            . . .
int globalVariable;              globalVariable = 10;
. . .                            . . .
```

– klaidingas – kompiliavimo klaida – antrame faile *globalVariable* nepaskelbtas.

Reikia skelbimus teikti visuose failuose, o apibrėžimą palikti tik viename. Skelbimą be apibrėžimo nurodo raktažodis *extern*:

```
//failas1.cpp                    //failas2.cpp
. . .                            . . .
int globalVariable;              extern int globalVariable;  // tik skelbimas
                                 globalVariable = 10;        // inicializavimas
. . .                            . . .
```

Inicializavimas galimas bet kur; vien tik skelbimo ir inicializavimo sutapdinti į vieną operatorių negalima.
Jei keliuose failuose reikia skirtingų globaliųjų kintamųjų vienodais vardais, juos reikia skelbti *static*: matomumo sritis susiaurės iki failo.

2. Funkcijos.
Failuose būtina teikti tik funkcijos skelbimus. Galima funkciją skelbti viename faile, o apibrėžti – kitame.

```
//failas1.cpp                    //failas2.cpp
. . .                            . . .
int f( int a, int b ){           int f( int, int );   // tik skelbimas
        return ( a+b);           . . .
}                                cout<<f( 1, 2 );
. . .                            . . .
```

Kaip ir kintamiesiems, paskelbus funkciją *static*, ji kitame faile nebus matoma.

3. Klasės.
Klasėms skelbimų skirtinguose failuose nepakanka, kadangi skelbimas tėra tik

class className;

Kompiliatorius gi turi žinoti visus klasės laukus ir jos metodų skelbimus. Todėl klases visuose failuose reikia arba apibrėžti (tai nepatogu), arba į visus failus įjungti antraštinius failus su reikiama informacija apie klases.

Antraštiniai failai

Juose talpinami kintamųjų ir funkcijų skelbimai, klasių apibrėžimai.

```
//failash.h
extern int globalVariable;
int globalFunction( int );
class someClass{
        private:
                int field;
        public:
                int method( int n ){
                        return n;
                }
};
```

```
//failas1.cpp                    //failas2.cpp
int globalVariable;              #include "failash.h"
int globalFunction( int n ){     . . .
        return n;                globalVariable = 10;
}                                cout<<globalFunction( globalVariable );
```

```
. . .                                   . . .
                                        int function( ){
                                                someClass sc;
                                                int variable = sc.method( 20 );
                                        . . .
                                        }
                                        . . .
```

Antraštiniame faile metodo apibrėžimo gali ir nebūti; jį galima tik skelbti, o apibrėžti bet kuriame faile:

```
//failash.h
extern int globalVariable;
int globalFunction( int );
class someClass{
        private:
                int field;
        public:
                int method( int );
};

//failas2.cpp
#include "failash.h"
int someClass::method( int n ){
        return n;
}
. . .
```

Įjungiant į programą kelis antraštinius failus, gali pasitaikyti, kad vienas kintamasis *globalVariable* bus apibrėžtas keliuose jų – tada būtų deklaruota kompiliavimo klaida. Nuo tokių klaidų galima apsidrausti direktyva-konstrukcija *if defined – define*:

```
. . .
#if !defined( globalVariable )  // jei neapibrėžta -
#define globalVariable          // apibrėžti globalVariable
#endif                          // konstrukcijos pabaiga
. . .
```

Aišku, tai neapsaugo nuo pakartotinio apibrėžimo antraštiniame faile ir **.cpp* faile.

Vardų sritys

Tai – įvardinta programos matomumo sritis. Pavyzdžiui,

```
namespace single{
        const float PI     = 3.14159f;
        const float RAD = 57.2829f;
}
. . .
```

```
cout<<PI<<endl;   //  klaida!: PI nematomas
cout<<single::PI<<endl;   // gerai
. . .
```

Patogiau naudoti ne priklausomybės operaciją *::* , o direktyvą *using*:

```
namespace single{
        const float PI     = 3.14159f;
        const float RAD = 57.2829f;
}
. . .
using namespace single;
cout<<PI<<endl;   //  gerai
. . .
```

Direktyva vardų sritį matoma paverčia nuo jos iki pat programos pabaigos. Norint galima apriboti tai, įdedant direktyvą į kurią nors matomumo sritį arba sukuriant tokią sritį.
Leidžiami keli tos pat vardų srities skelbimai – tai apjungia tas sritis į vieną:

```
namespace single{
        const float PI = 3.14159f;
}
. . .
namespace single{
        const float RAD = 57.2829f;
}
. . .
namespace single{
        int method( int n ){
                return n;
        }
}
. . .
```

Tokia vardų sričių savybė patogi vardų sritis skelbiant ir apjungiant **.h* failuose.
Vardus į vardų sritį įjungti galima ir iš už srities ribų:

```
. . .
const float single::PI  = 3.14159f;
. . .
```

Galima ir bevardė vardų sritis. Tokios srities elementai matomi tik iš jos failo. Tai – alternatyva globalių kintamųjų skelbimui *static*:

```
//failas1.cpp                       //failas2.cpp
namespace{                          namespace{
        int globalVariable = 10;            int globalVariable = 20; // gerai, vardų
}                                   }                                // konflikto nėra
. . .                               . . .
```

Duomenų tipų pervardinimas

Duomenų tipus (įskaitant klases) galima pervardinti
typedef senasVardas naujasVardas; operatoriumi. Senieji duomenų tipų pavadinimai taip pat galioja:

```
typedef int feet;
typedef float inches;
typedef int* pointerInt;
feet f1, f2;
inches in1;
pointerInt p1, p 2;
int f3;
float in2, in3;
int* p3, *p4;
. . .
```