Aprakstu darbības apgabali

- Identifikatora darbības apgabals (scope) ir programmas daļa, kurā šo identifikatoru var lietot.
- Iespējamie darbības apgabali:
  - Bloks
  - Funkcija un tās prototips
  - Klase
  - Fails

Aprakstu darbības apgabali – bloks

- 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 b = 10; // sākas b darbības apgabals
      a = a; // OK
   } // beidzas b darbības apgabals

   a = a + b; // KĻŪDA! b nav pieejams
   for(int i = 0; i < 5; i++) // sākas i darbības apgals
   {
      cout << i << endl;
   }
} // beidzas a un i darbības apgabali
```

```
int i;
for (i = 0; ...
```

Aprakstu darbības apgabali – funkcija un tās prototips (deklarācija)

- 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
```

Aprakstu darbības apgabali – klase

- 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
 }
```

Aprakstu darbības apgabali – fails

- 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

Atmiņas klases

- 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();
   // ...
}
```

Atmiņas klases

- 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 mainīgiem, tāpēc tiešā veidā to raksta reti.

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

Atmiņas klases

- 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)
...
```

Atmiņas klases

- 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);
```

Atsauces

- Atsauces (angl. *reference*) tiek izmantotas kā mainīgo alternatīvie nosaukumi.
- Definējot atsauci, tās nosaukuma sākumā liek ampersanda (&) zīmi.
- 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
```

- Atsauces visbiežāk izmanto kā funkciju parametrus, tādā veidā ļaujot funkcijai mainīt nodotos parametrus un izmaiņas atdot izsaucošajai funkcijai.

- 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;
```

Kvalifikators - `const`

- 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.