# Objektumorientált programozás

## Objektumalapú programozás a C++ programozási nyelvben

Adatfolyamok

**Darvay Zsolt**

# A standard könyvtár

17. A standard könyvtár felépítése
18. Adatfolyamok
19. Algoritmusok

# 18. Adatfolyamok

Bevitel és kivitel megvalósítása osztályokkal

# Áttekintés

18.1. Az adatfolyam fogalma

18.2. Formátumozott kimenet

18.3. Formátumozott bemenet

18.4. Állománykezelés

# 18.1. Az adatfolyam fogalma

- Az adatfolyam (stream) alatt az adatok áramlását értjük egy bizonyos forrástól egy cél irányába.
- A forrás lehet: a billentyűzet, egy állomány, vagy egy memóriaterület.
- A cél lehet a képernyő, egy állomány, vagy egy memóriaterület.

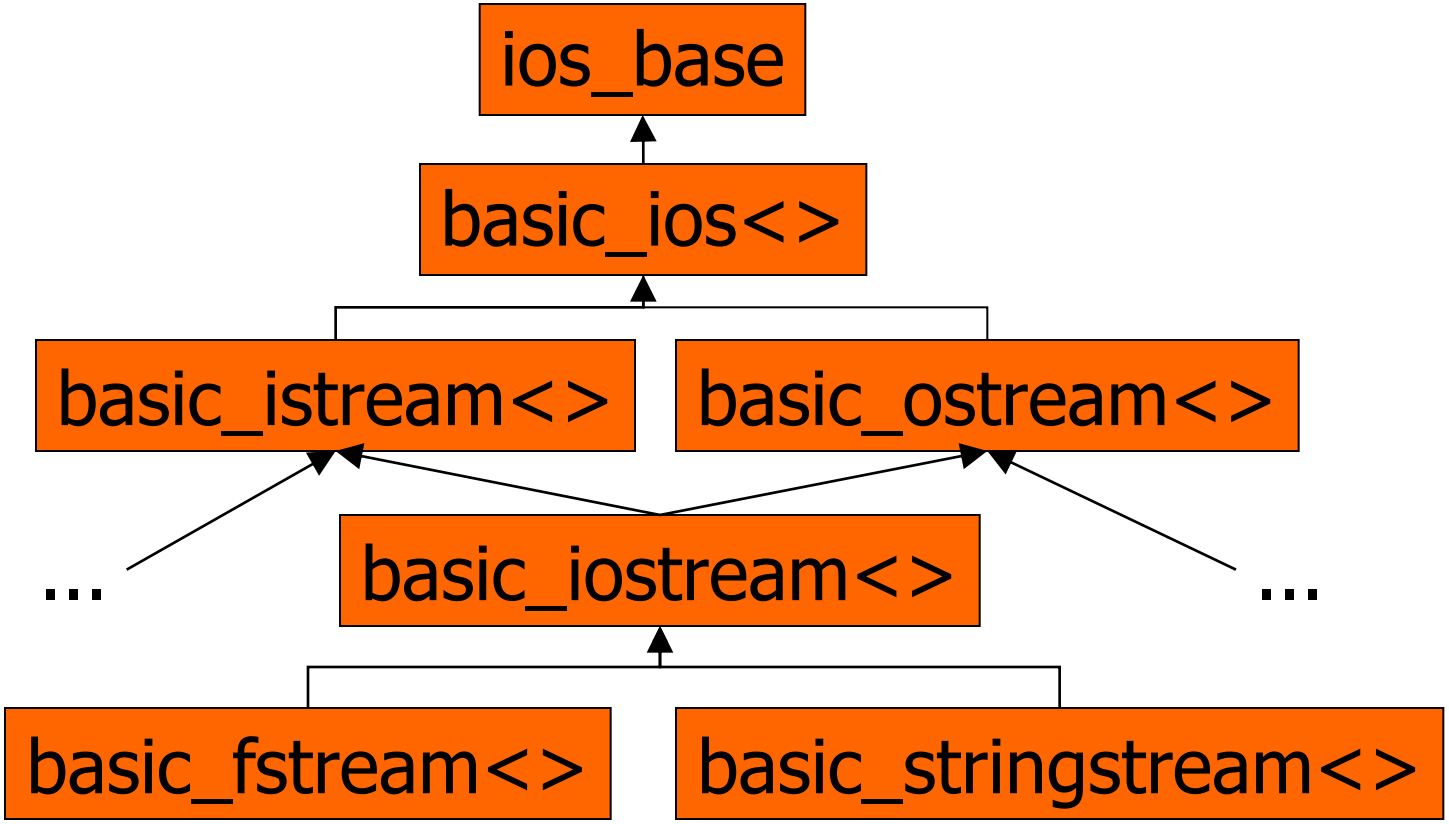
# Az iostream és iostream.h közti különbség

- Az adatfolyamok használatához az iostream fejállományt kell beékelni.
- Az adatfolyamokra vonatkozó osztályokat sablonokként definiálja a rendszer.
- A régebbi változatokkal való kompatibilitás céljából a Visual C++ 6.0-ban még megőrizték az iostream.h fejállományt, amely nem használ sablonokat. A jelenlegi Visual C++-ban az iostream.h fejállomány már nem használható.

# Az adatfolyamokra vonatkozó rendszer felépítése

- Az ios_base alaposztállyal rendelkező hierarchia.
- Az átmeneti tárak (pufferek) kezelése a basic_streambuf bázisosztályból származtatott osztályokkal.
- A formázással kapcsolatos helyi sajátosságok megadása a locale objektummal.

# Az ios_base alaposztállyal rendelkező hierarchia



# Az osztályhierarchia további részei

- A basic_istream osztály a basic_istringstream és basic_ifstream származtatott osztályokkal is rendelkezik.
- A basic_ostream osztály a basic_ostringstream és basic_ofstream származtatott osztályokkal is rendelkezik.
- A basic_ios virtuális bázisosztálya a basic_istream és basic_ostream osztályoknak.
- Ily módon a basic_ios adattagjait csak egy példányban örökli a basic_iostream.

# A basic_ előtag elhagyásával kapott osztályok

- A <> karakterek az osztályhierarchiában arra utalnak, hogy az illető osztály egy sablon.
- A basic_ előtag elhagyásával olyan osztályokat kapunk, amelyekre a sablonparaméter char típussal van behelyettesítve.
- Ha a sablonparaméter helyén wchar_t van, akkor az adatfolyam széles karaktereket dolgoz fel.

# Az istream, ostream, iostream, ... osztályok

- Típusdeklarációkkal vannak megadva:

```
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
typedef basic_iostream<char> iostream;
```

# Szabványos adatfolyamok

- Kimeneti adatfolyam karakterekre:

  ostream cout;

- Hibaüzenetek kimeneti adatfolyama:

  ostream cerr; // pufferkezelés nélkül

  ostream clog; // pufferkezeléssel

- Bemeneti adatfolyam karakterekre:

  istream cin;

# Az wistream, wostream, wiostream, ... osztályok

- Típusdeklarációkkal vannak megadva:

```
typedef basic_istream<wchar_t> wistream;
typedef basic_ostream<wchar_t> wostream;
typedef basic_iostream<wchar_t> wiostream;
```

# Szabványos adatfolyamok széles karakterekre

- Kimeneti adatfolyam wchar_t típusra:

  wostream wcout;

- Hibaüzenetek kimeneti adatfolyama:

  wostream wcerr;  // pufferkezelés nélkül

  wostream wclog;  // pufferkezeléssel

- Bemeneti adatfolyam wchar_t típusra:

  wistream wcin;

# 18.2. Formátumozott kimenet

- A formátum szerinti beolvasását és kiírást a ios_base osztályban definiált jelzőbitek segítségével valósíthatjuk meg.
- A jelzőbitekhez egy-egy nevet rendeltek, oly módon, hogy a bitekre az

  ios_base::bitnév

- alakban lehessen hivatkozni. A jelzőbitek által alkotott adattag típusa ios_base::fmtflags. Ez általában int.

# A jelzőbitek nevei

- skipws: beolvasáskor a fehér karaktereket figyelmen kívül hagyja;
- unitbuf: a puffert üríti kimenet után;
- uppercase: nagybetűvel írja a hexadecimális számokat, és az exponenciális alakot;
- showbase: a számrendszer alapja is megjelenik;
- showpoint: a tizedespont és az utána következő esetleges nullák is megjelennek;
- showpos: a pozitív egészeket is előjellel írja ki;

# A jelzőbitek nevei

- left: balra igazítás;
- right: jobbra igazítás;
- internal: a töltőkarakterek az előjel, illetve a számrendszer alapja, és a szám közé kerülnek;
- dec: tízes számrendszer;
- oct: nyolcas számrendszer;
- hex: tizenhatos számrendszer;
- scientific: exponenciális alakos kiírás;
- fixed: tizedespontos kiírás;
- boolalpha: a logikai típust szavakkal írja ki.

# A jelzőbitek csoportosítása

- Három csoport:
  - adjustfield: left, right, internal;
  - basefield: dec, oct, hex;
  - floatfield: scientific, fixed.
- A csoportok nevére is az ios_base::név alakban hivatkozhatunk, ahol a név a csoport neve.

# A setf tagfüggvény

- Az ios_base tagfüggvénye. Két alakja van:
  - fmtflags setf(fmtflags f);
  - fmtflags setf(fmtflags f, fmtflags cs);
- Az első változat a megadott biteket állítja be.
- A második egy adott csoporton belüli bitet állít be, a régit törli.
- Mindkét függvény a régi állapotot téríti vissza.

# A flags tagfüggvény

- Az ios_base tagfüggvénye. Két alakja:

  fmtflags flags() const;

  fmtflags flags(fmtflags fmtfl);

- Az első a jelzőbitekre vonatkozó adattagot téríti vissza.
- A második módosítja a jelzőbiteket a paraméternek megfelelően, és a régi értéket téríti vissza.

# Példa a setf és flags tagfüggvényekre

- Visual C++-ban érvényes alakban kiírjuk a bitek nevét, és az adattagot is bitenként (az int mérete 32 bit).

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

# A binaris_c függvény

```
void binaris_c(int x) {
    printf("bitenkent: ");
    for (int i = 8 * sizeof(int) - 1; i >= 0; i--) {
        printf("%d", (x >> i) & 1);
        if (!(i % 8)) printf(" ");
    }
    printf("\n");
}
```

# A bit_nevek függvény

```
void bit_nevek(const char* s[], int x)
{
   for (int i = 0; i < 15; i++)
   if ((x >> i) & 1)
        printf("%-16s", s[i]);
   printf("\n");
}
```

# A kiiras függvény

```
void kiiras(const char* s[], int x)
{
   binaris_c(x);
   bit_nevek(s, x);
}
```

# Az enum_nevek tömb

```
const char* enum_nevek[] = {
   "skipws",      "unitbuf",      "uppercase",
   "showbase",    "showpoint",    "showpos",
   "left",        "right",        "internal",
   "dec",         "oct",          "hex",
   "scientific",  "fixed",        "boolalpha"
};
```

# A fő függvény

```
int main() {
    int x = 64;
    kiiras(enum_nevek, cout.flags());
    cout << x << endl;
    cout.setf(ios_base::oct, ios_base::basefield);
    kiiras(enum_nevek, cout.flags());
```

# A fő függvény

```
    cout << x << endl;
    cout.setf(ios_base::showbase);
    cout.setf(ios_base::hex, ios_base::basefield);
    kiiras(enum_nevek, cout.flags());
    cout << x << endl;
    return 0;
}
```

# A kimenet

```
bitenkent: 00000000 00000000 00000010 00000001
skipws          dec
64
bitenkent: 00000000 00000000 00000100 00000001
skipws          oct
100
bitenkent: 00000000 00000000 00001000 00001001
skipws          showbase        hex
0x40
```

# A width tagfüggvény

- Az ios_base tagfüggvénye. Két alakja:

  streamsize width() const;

  streamsize width(streamsize w);
- A streamsize egy egész típus, általában int.
- Az első függvény a mezőszélességet meghatározó adattagot téríti vissza. A második módosítja azt, és a régit téríti vissza.
- A beállított mezőszélesség csak az első kimeneti műveletre vonatkozik.

# A fill tagfüggvény

- Az basic_ios tagfüggvénye. Két alakja:

  E **fill**() const;

  E **fill**(E ch);

- Itt E a sablonparaméter, ami általában char.
- Az első változat a töltőkaraktert kérdezi le.
- A második beállítja azt, és a régit téríti vissza.
- Az alapértelmezett töltőkarakter a szóköz.

# Valós típusok kimenete

- A kimenetet a formátum és a pontosság határozza meg.
- A formátum lehet:
  - általános
  - tudományos (scientific)
  - fixpontos (fixed)

# Az általános formátum

- A lehető legkevesebb számú karakterrel igyekszik kiírni.
- A pontosság a kiírandó számjegyek maximális számát jelenti.
- A %g konverzióelőírásnak felel meg.
- Ha más formátum az aktív, akkor a

```
cout.setf(0, ios_base::floatfield);
```

- segítségével állíthatjuk be.

# A tudományos formátum

- Az exponenciális alakot használja a kiírásra.
- Például: 2.345e10
- A pontosság a tizedespont utáni számjegyek maximális száma.
- A %e konverzióelőírásnak felel meg.
- Beállítás:

```
cout.setf(ios_base::scientific, ios_base::floatfield);
```

# A fixpontos formátum

- A tizedespontos alakot használja.
- Például: 234.56789
- A pontosság ebben az esetben is a tizedespont utáni számjegyek maximális száma.
- A %f konverzióelőírásnak felel meg.
- Beállítás:

```
cout.setf(ios_base::fixed, ios_base::floatfield);
```

# A precision tagfüggvény

- Az ios_base tagfüggvénye. Két alakja:

  streamsize precision() const;

  streamsize precision(streamsize prec);
- A pontosságot kérdezi le, illetve állítja be. A pontosság előző értékét téríti vissza.
- A beállított pontosság mindaddig érvényben marad, amíg újra meg nem változtatjuk.

# Példa a width, fill és precision tagfüggvényekre

```
#include <iostream>
#include <cstdio>
using namespace std;
int main() {
   const double x = 987.65432;
   cout << "*";
   cout.width(11);
   cout.precision(4);
   cout.fill('0');
```

# Példa a width, fill és precision tagfüggvényekre

```
    cout << x << "*\n";
    cout << "*";
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << x << "*\n";
    cout << "*";
    cout.width(11);
    cout << x << "*\n";
    printf("*%011.4lf*\n", x);
}
```

# A kimenet

| | |
|---|---|
| *000000987.7* | kerekítés, négy számjegy |
| *987.6543* | alapértelmezett mezőszélesség |
| *000987.6543* | mezőszélesség: 11 |
| *000987.6543* | printf függvénnyel |

# Módosítók (manipulátorok)

- Olyan sajátos tagfüggvények, amelyek az adatfolyamra hivatkozó referenciát térítenek vissza.
- Ez által a tagfüggvénymeghívások egymáshoz láncolhatók.
- Egyes módosítók az iomanip fejállományban vannak deklarálva.

# Példa módosítóra

```
#include <iostream>
#include <iomanip>
#include <cstdio>

using namespace std;
```

# Példa módosítóra

```
int main() {
   const double x = 987.65432;
   cout << "*" << fixed << setw(11)
        << setprecision(4) << setfill('0')
        << x << "*" << endl;
   printf("*%011.4lf*\n", x);
}
```

# A kimenet

*000987.6543*

*000987.6543*

- A programban szabványos módosítókat használtunk.
- Lehetőség van arra is, hogy a programozó definiáljon saját módosítót.

# A << operátor túlterhelése

- Szabványos típusokra a << operátor túl van terhelve.
- Saját típusokra ezt mi tehetjük meg az

ostream& operator <<(ostream& s, const oszt &o);

- alakú függvénnyel, ahol oszt az illető típust definiáló osztály neve.
- Ha ez az operátor az oszt védett tagjaira kell hivatkozzon, akkor ezt általában az oszt egy nyilvános tagfüggvényén keresztül valósíthatjuk meg.

# Példa a << operátor túlterhelésére

```
#include <iostream>
using namespace std;
class tort {
   int sz;
   int n;
public:
   tort(int a, int b) { sz = a; n = b; }
   ostream& kiir(ostream& s) const;
};
```

# A kiir tagfüggvény

```
ostream& tort::kiir(ostream& s) const
{
   s << sz << " / " << n;
   return s;
}
```

# A << operátor

```
ostream& operator<<(ostream& s, const tort& t)
{
   return t.kiir(s);
}
```

# A fő függvény

```
int main() {
   tort t1(3, 5);
   cout << t1 << endl;
}
```

Kimenet:

3 / 5

# Példa függvénysablonra (típusazonosítás)

```
#include <iostream>
using namespace std;
template <class T>
void kiir_tipus(T a) {
   cout << a << "\t";
   cout << typeid(T).name() << endl;
}
```

# A tort osztály

```
class tort {
   int sz;
   int n;
public:
   tort(int a, int b);
   ostream& kiir(ostream& s) const;
};
```

# A konstruktor

```
tort::tort(int a, int b)
{
   sz = a;
   n = b;
}
```

# A kiir tagfüggvény

```
ostream& tort::kiir(ostream& s) const
{
   return s << sz << " / " << n;
}
```

# A << operátor

```
ostream& operator <<(ostream& s, const tort& t)
{
   return t.kiir(s);
}
```

- Ha az operátort nem terhelnénk túl, akkor a kiir_tipus függvénybeli

  cout << a << "\t";

  kiírást nem tudná végrehajtani a rendszer.

# A fő függvény

```
int main() {
   kiir_tipus(3);
   kiir_tipus(5.9);
   tort y(4, 9);
   kiir_tipus(y);
}
```

Kimenet:

```
3        int
5.9      double
4 / 9    class tort
```

# Az előző példa módosítása

```
#include <iostream>
#include <iomanip>
using namespace std;
template <class T>
void kiir_tipus(T a) {
   cout << left << setw(10) << a << "\t";
   cout << typeid(T).name() << endl;
}
```

# A fő függvény

```
int main() {
   kiir_tipus(3);
   kiir_tipus(5.9);
   tort y(4, 9);
   kiir_tipus(y);
}
```

Kimenet:

```
3                  int
5.9                double
4          / 9     class tort
```

- A setw csak a számlálóra vonatkozott.

# További módosítás

```
#include <iostream>
#include <iomanip>
#include <sstream>
using namespace std;
template <class T>
void kiir_tipus(T a) {
   cout << left << setw(10) << a << "\t";
   cout << typeid(T).name() << endl;
}
```

# A kiir tagfüggvény

```
ostream& tort::kiir(ostream& s) const {
   ostringstream str_buf;
   str_buf << sz << " / " << n;
   return s << str_buf.rdbuf()->str();
}
```

- Részletek a következő oldalon.

# Az ostringstream használata

- A string osztály egy objektumába írunk.
- Az rdbuf tagfüggvény a basic_stringbuf egy objektumára hivatkozó mutatót térít vissza.
- Ennek az str tagfüggvénye a string osztály objektumát téríti vissza.

# A fő függvény

```
int main() {
   kiir_tipus(3);
   kiir_tipus(5.9);
   tort y(4, 9);
   kiir_tipus(y);
}
```

Kimenet:

```
3          int
5.9        double
4 / 9      class tort
```

# 18.3. Formátumozott bemenet

- A bemeneti műveletek a >> operátor segítségével végezhetők.
- A kimenethez hasonló formázási műveleteket alkalmazhatunk ebben az esetben is.
- Ha beolvasáskor egy hiba jelentkezik, akkor az adatfolyam hibaállapotba kerülhet.

# A hibaállapot

- A hibaállapotot is jelzőbitek segítségével lehet felismerni:
- goodbit: nincs hiba;
- eofbit: állomány vége jelhez jutottunk;
- failbit: a következő művelet nem lesz sikeres;
- badbit: sérült adatfolyam.

# A >> operátor túlterhelése

- Szabványos típusokra a >> operátor túl van terhelve.
- Saját típusokra ezt mi tehetjük meg az

  istream& operator >>(istream& s, oszt &o);

- alakú függvénnyel, ahol oszt az illető típust definiáló osztály neve.
- Ha ez az operátor az oszt védett tagjaira kell hivatkozzon, akkor ezt általában az oszt egy nyilvános tagfüggvényén keresztül valósíthatjuk meg.

# Példa a >> operátor túlterhelésére

```
class tort {
   int sz;
   int n;
public:
   tort(int a, int b) { sz = a; n = b; }
   ostream& kiir(ostream& s) const;
   istream& beolvas(istream& s);
};
```

# A beolvas tagfüggvény

```
istream& tort::beolvas(istream& s)
{
   cout << "Szamlalo: ";
   s >> sz;
   cout << "Nevezo: ";
   s >> n;
   return s;
}
```

# A >> operátor

```
istream& operator>>(istream& s, tort& t)
{
   return t.beolvas(s);
}
```

# A fő függvény

```
int main() {
   tort t1(3, 5);
   cout << t1 << endl;
   cin >> t1;
   cout << t1 << endl;
}
```

Kimenet:

```
3 / 5
Szamlalo: 4
Nevezo: 7
4 / 7
```