1. Motiváció

C nyelvben a `printf` és a `scanf` függvények használata a programozótól nagy körültekintést igényel. A változó számú argumentum miatt a fordító nem tudja ellenőrizni, hogy a formátum mezőben megadott, kiírandó adatok száma és típusa megegyezik-e a paraméterlistával.

```
void foo()
{
  int i=19,
  double pi=3.14;
  printf("%d\n", i);  // OK
  printf("%d␣%d\n", i, 19, pi); // hibás formátum mező, vagy paraméter
}
```

A C++ nyelvben a függvény nevek túlterhelésével biztonságosabbá tehetnénk az adatok kiírását és beolvasását.

```
void write(const int i) {
  printf("%d", i);
}
void write(const double r) {
  printf("%f", r);
}
void write(const char *s) {
  printf("%s", s);
}
.....

void foo()  {
  int i=4,
  double pi=3.14;

  write("4*Pi=");
  write( 4*pi );
  write( "\n" );
}
```

Ez a megoldás kényelmetlen. A C++ nyelvben a « és a » operátorok túlterhelésével oldották meg a feladatot. Az előző kiíratás megoldása:

```
cout << "4*pi=" << 4*pi << '\n';
```

A túlterhelt operátorokat a C++ fordító függvényhívásokra fordítja. Az egyes függvények `ostream&` értékkel térnek vissza, ami lehetőséget ad, a függvények láncolására. Az előzővel ekvivalens programrészlet:

```
operator<<( (operator<<(cout,"4*pi=")).operator<<(4*3.14), '\n');
```

« Inserter (beszúrás) a kiírandó folyamba beszúrja a megjelenítendő adatokat.

» Extractor (kinyerés) a bejövő adatfolyamból kinyeri a soron következő adatot.

Előre definiált streamek.

`cin` standard input

`cout` standard output

`cerr` standard error

`clog` buffer-olt változata a standard errornak.

2. Kiírás

2.1. Felhasználó által definiált inserter

```
struct Pair{ int x; int y; };
ostream& operator<<(ostream& o, const Pair& p) {
   return o <<'(' << p.x << ',' << p.y << ')';
}
void foo() {
   Pair p1={2,5};
   cout << p1 << endl;
}
```

2.2. Azonnali output megjelenítése

A hatékonyság érdekében az » inserter a karaktereket bufferbe teszi, és csak nagyobb egységekben jeleníti meg. Interaktív programok esetén, szükséges lehet kevesebb adat azonnali megjelenítése is, ezt a `flush` speciális adat (manipulátor) kiírásával érjük el.

```
   cout << "Kérem␣az␣x␣értékét␣\n" << flush;
```

A sorvége `endl` érték nem csak a `NL` karaktert teszi az outputra, hanem azonnali megjelenítést kér. Az előző példával ekvivalens programrész:

```
   cout << "Kérem␣az␣x␣értékét␣" << endl;
```

2.3. Bináris kiíratás

```
   int c='A';
   cout.put(c);      // egy karaktert ír ki
   cout << (char)c; // egy karaktert ír ki
   cout.write( (char*)&c, sizeof(c) );  // egész bináris kiíratása
```

2.3.1. Tetszőleges adatok konvertálása karakterfüzérré

Akik használták a `sprintf` függvényt , azoknak jó hír, hogy C++-ban az `ostrstream` segítségével ugyanezt tehetik, biztonságosabban és egyszerűbben.

```
#include <sstream>
#include <iostream>
using namespace std;
int main() {
   ostringstream os;
   os << 123 << '␣' << 456 << ends;
   cout << os.str() << endl;
   return 0;
}
```

3. Beolvasás

```
int i;
double r;
cin >> i >> r;
```

3.1. Adatfolyam állapot lekezelése

A következő függvények az adatfolyam állapotát adják vissza:

`bool good()` A beolvasás sikeres volt.

`bool eof()` Fájl vége volt.

`bool fail()` Az utolsó beolvasás sikertelen volt, de adatvesztés nem lépett fel.

`bool bad()` Javíthatatlan hiba. Ilyenkor a `fail()` függvény is jelez.

Ezeket a függvényeket ritkán hívjuk meg, mert a szabványos input, output könyvtár tervezésénél a következő operátorok túlterhelésével olyan trükkel éltek, ami leegyszerűsíti a hibakezelést.

`operator void*()` A cast operator nem NULL pointert (C++-ban nem 0-át) ad vissza, ha `!fail()`.

`bool operator!()` Meghívja a `fail()` függvény, és annak a visszatérési értékét adja vissza.

```
int i;
if( cin >> i ) { .... } // rendben volt a beolvasás
else { .... } // hibás input
```

A `cin » i` kifejezés megegyezik a `cin.operator»( i )` függvény hívással, ami `istream&` értékkel tér vissza. Ezzel a típussal a fordító az `if(....)` utasítás feltételrészében nem tud mit kezdeni, ezért automatikusan meghívja az `operator void*()` típuskonvertáló operátort, és ez a `fail()` függvény alapján 0 vagy nem 0 címmel tér vissza, ami megfeleltethető egy logikai értéknek.

3.2. Beolvasó ciklus szervezése

```
    const int MAXPAIR=10;
    Pair vp[MAXPAIR];
    int x,y;
    int npair=0;
    while( npair< MAXPAIR && cin>>x && cin >>y )
    {
        vp[npair].x=x;
        vp[npair++].y=y;
    }
```

3.3. Felhasználó által definiált extractor

```
istream& operator>>(istream& i, Pair& pair)
{
   i >> pair.x >> pair.y;
   return i;
}
```

3.4. Saját hibaállapot állítás

Az előző kódrészletben az olvasás például akkor volt sikertelen, ha egész szám helyett nem számjegy karakter jött. Legyen az a feladat, hogy akkor is hibát jelzünk, ha negatív egész számokat olvasunk.

```
istream& operator>>(istream& i, Pair& pair)
{
   i >> pair.x >> pair.y;
   if( !i ) return i; // hiba volt
   if( pair.x <0 || pair.y<0 )
   {
      i.clear(ios::badbit|i.rdstate() );
   }
   return i;
}
```

A `clear()` függvény neve sajnos félrevezető, mert nem törli, hanem beállítja a hiba flag-et. Az `rdstate()` függvény visszaadja az aktuális hibaállapotot, és ezzel bitenkénti vagy kapcsolattal egybe állítjuk a `badbit` flag-et.

3.5. Maximálisan megengedett karakterszám beállítása

String `char*` beolvasása esetén vigyázni kell, nehogy több adatot olvassunk, mint amennyi memória rendelkezésre áll. Veszélyes kód (buffer overflow):

```
void foo()
{
   char p[5];
   cin >> p;
}
```

A `width(int)` függvény segítségével megadhatjuk a maximálisan beolvasható karakterek számát, a stringet lezáró nullát is beleszámítva.

```
void foo()
{
   char p[5];
   cin.width( sizeof(p) );
   cin >> p;
}
```

Ugyanezt kényelmesebben megtehetjük a `setw(int)` manipulátorral.

```
void foo()
{
   char p[5];
   cin >> setw(sizeof(p) >> p;
}
```

Egy sor beolvasásához használhatjuk a `getline` függvényt. A függvény harmadik paramétere a sor végét jelző karakter értéke, alapértéke természetesen a `NL` karakter.

```
void foo()
{
   char line[100];
   cin.getline( line, sizeof(line) );
}
```

3.6. Szóközök beolvasása

Ha a karakterek olvasásánál mi szeretnénk a whitespace-eket feldolgozni, akkor használjuk a `get(char)` függvényt, vagy a `noskipws` manipulátort. A `get(char)` függvény mindig beolvassa a szóközöket is.

```
void foo()
{
   char c;
   cin >> noskipws;
   while( cin >> c ) cout<<c;
   cin >> noskipws;

   cin.get(c);
   ....
   while( cin.get(c) ) { ..... }
}
```

3.7. Bináris input

Tetszőleges bináris adat olvasásánál a `read()` függvényt használjuk.

```
void foo()
{
   int v[100];
   ....
   cin.read( (char*)v, sizeof(v) );
}
```

3.7.1. Karakterfüzér konvertálása tetszőleges adatra

Akik használták az `sscanf` függvényt , azoknak jó hír, hogy C++-ban az `istrstream` segítségével ugyanezt tehetik, biztonságosabban és egyszerűbben.

```
#include <sstream>
void foo()  {
  int iv1, iv2;
  istringstream is("123 456");
  is >> iv1 >> iv2; // konvertálás
}
```

4. Formázás

4.1. Mezőszélesség

A mezőszélesség beállítás csak az utána következő egy adatra érvényes. Kiírásnál csak akkor hatásos, ha a megadott érték nagyobb, mint a kiírandó karakterek száma.

```
   cout.width(5);
   cout << 12 << ' ' << 34 << ' ' << 56 << endl;
```

Egyszerűbben manipulátorral:

```
   cout << setw(5) << 12 << ' ' << 34 << ' ' << 56 << endl;
```

A mezőszélesség beállítás érvényes a beolvasásnál is, erre már adtunk meg példát.

4.2. Kitöltés és jobbra, balra igazítás

Ha a mezőszélesség megadásánál nagyobb helyet adunk meg, mind amekkora a kiírandó szöveg hossza, akkor megadhatjuk, hogy a szöveg a mező jobb vagy bal széléhez legyen igazítva, és az üres helyekre milyen kitöltő karakter kerüljön.

```
cout.fill('*');
cout << left << setw(5) << 13 << ",";
cout << setw(5) << 25 << ",";
cout.fill('#');
cout << right << setw(5) << 14 << "," << endl;
```

output: `13***,25***,###14,`

4.3. Számrendszer megadása

A kiírásnál és beolvasásnál választhatunk decimális, oktális és hexadecimális számrendszerek közül. A `showbase` flag beállításával a számrendszer alapja is kiolvasható a szám alapján.

```
int x=64;
cout << dec << x << '␣'
     << hex << x << '␣'
     << oct << x << '␣' << endl;
cout << showbase
     << dec << x << '␣'
     << hex << x << '␣'
     << oct << x << '␣' << endl;
```

output :

```
64 40 100
64 0x40 0100
```

Olvasásnál:

```
cin >> dec >> x
    >> hex >> y
    >> oct >> z
```

4.4. Manipulatorok

név	jelentés
boolalpha	bool típus szövegesen (true, false)
dec	tízes számrendszer
endl	newline és flush
ends	null karakter
fixed	tizedes tört alak
flush	stream buffer kiürítés
hex	hexadecimális számrendszer
left	balra igazítás
noboolalpha	bool formátuma szám (0,1)
noshowbase	ne mutassa a számrendszer alakját
noshowpoint	ne írjon tizedesvesszőt (pontot)
noshowpos	ne írjon pozitív előjelet
noskipws	ne olvassa át a whitespace karaktereket
nouppercase	számoknál (E) és hexa értékeknél (A-F) ne írjon nagybetűt
oct	nyolcas számrendszer
resetiosflags	formátum flagek törlése
right	jobbra igazít
scientific	exponenciális alak
setfill	kitöltő karakter beállítás
setiosflags	beállítja a megadott jelzőbiteket
setprecision	kiírandó tizedesjegyek beállítása
setw	mezőszélesség
showbase	mutassa a számrendszer alakját
showpoint	írjon tizedesvesszőt (pontot)
showpos	írjon pozitív előjelet is
skipws	whitespacek átolvasása
uppercase	számoknál (E) és hexa értékeknél (A-F) nagybetűt használjon

4.5. Fájlkezelés

```
// using fstream constructors.
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
   fstream os("test.txt", fstream::in | fstream::out);

   os << "PI:" << 3.14 << endl;
   .....
   os.close();

   return 0;
}
```

Megnyitási módok:

flag	megnyitási mód
app (append)	minden egyes kiírás előtt a fájl végére pozícionál
ate (at end)	megnyitás után a fájl végére pozícionál
binary (binary)	bináris mód
in (input)	olvasás engedélyezése
out (output)	írás engedélyezése
trunc (truncate)	megnyitáskor a fájl tartalmának a törlése

Fájl megnyitás `open()` függvény segítségével:

```
// fstream::open
#include <fstream>
using namespace std;

int main ()
{
  fstream fio;

  fio.open ("test.txt", fstream::in | fstream::out | fstream::app);

  // >> i/o operations here <<

  fio.close();

  return 0;
}
```