TÓTH BERTALAN

# A C++11 NYELV ÚJ LEHETŐSÉGEINEK ÁTTEKINTÉSE

Tóth Bertalan:

# A C++11 nyelv új lehetőségeinek áttekintése

– 2013 –

Lektorálta: Juhász Tibor

# Tartalom

# A C++11 nyelv új lehetőségeinek áttekintése

## Bevezetés

A C++11 (ISO/IEC 14882:2011) a C++ programozási nyelv 2011 szeptemberében megjelent szabványa, amely egy modern, az eddigieknél is hatékonyabb programozási nyelvet ad a programozók kezébe. A nyelv 2003-as változata elsősorban hibajavítás volt az eredetileg 1998-as szabványhoz képest, a mostani azonban egy sor újdonságot is tartalmaz. Összefoglalónkban csak az új elemeket mutatjuk be, feltételezve az eredeti C++ nyelv ismeretét. A szabványosítási folyamat ezzel nem zárult le, már ismert a C++14 és a C++17 szabványokban bevezetésre kerülő nyelvi megoldások egy része.

Az összefoglalóban a jobb áttekinthetőség érdekében önkényesen csoportosítottuk a C++11 nyelv új lehetőségeit. Ennek során az alapvető nyelvi elemektől, a sablonok használatán át jutunk el a Szabványos Sablonkönyvtár (STL) alkalmazásáig.

## Néhány hasznos hivatkozás

Sajnos napjainkban még egyik C++ fordító sem támogatja teljes mértékben az új szabvány ajánlásait. Mivel a támogatott megoldások köre napról-napra, verzióról-verzióra bővül, az Interneten érdemes utána nézni, hogy az általunk használt fordítóprogrammal, mit használhatunk az összefoglalóban bemutatott, és a nem ismertetett nyelvi elemek közül.

Néhány hasznos hivatkozás:

C++ 11 szabvány
http://isocpp.org/std/the-standard

C11 szabvány
http://www.open-std.org/jtc1/sc22/wg14/

C/C++ referencia
http://en.cppreference.com/w/cpp

C/C++ referencia
http://www.cplusplus.com/reference/

C++11 Wikipedia
http://en.wikipedia.org/wiki/C++11

C11 Wikipedia
http://en.wikipedia.org/wiki/C11_(C_standard_revision)

C++11 jellegzetességek a Visual Studio 2012-ben (Microsoft Development Network)
http://msdn.microsoft.com/en-us/library/vstudio/hh567368.aspx

C++0x/C++11 támogatás a GCC-ben
http://gcc.gnu.org/projects/cxx0x.html

# 1 Változások a C++ nyelv alapelemei szintjén

Ebben részben áttekintjük a C++11 nyelv új alaptípusait és konstansmegadási módjait:

- új, illetve módosított értelmű C++ kulcsszavak,
- új egész típusok,
- új lebegőpontos értékek,
- szigorúan típusos felsorolások – **enum** osztályok,
- a **nullptr** és a ***nullptr_t***,
- sztring literálok,
- felhasználó által definiált literálok,
- fordítás idejű konstans kifejezések (**constexpr**), a **static_assert** kulcsszó,
- kivételek továbbításának tiltása (**noexcept**),
- változók memóriahatárra igazítása.

## 1.1 Új, illetve módosított értelmű C++ kulcsszavak

| ***Új foglalt szavak*** | ***Módosított kulcsszavak*** |
|---|---|
| **alignas** | **auto** |
| **alignof** | **default** |
| **char16_t** | **delete** |
| **char32_t** | **export** |
| **constexpr** | **using** |
| **decltype** | |
| **final** | |
| **noexcept** | |
| **nullptr** | |
| **override** | |
| **static_assert** | |
| **thread_local** | |

## 1.2 Új egész típusok

A C++ szabvány most már hivatalosan is támogatja a **long long** és az **unnsigned long long** típusú egészeket, amelyek legalább 64 biten tárolódnak. Minden egészekre vonatkozó művelet elvégezhető velük, és a C++ könyvtári elemei is segítik a használatukat. Konstans értékekben az *ll*, *LL*, illetve az *ull*, *ULL* utótaggal jelölhetjük őket. A típusok értékhatárait hagyományosan a <*climits*> fejállomány (*LLONG_MIN*, *LLONG_MAX*, *ULLONG_MIN*, *ULLONG_MAX*), és a <*limits*> fájlban definiált ***numeric_limits***<*típus*> osztálysablon adattagjai biztosítják számunkra.

```
int main() {
    long long int x=123456789012345678LL;
    long long z = 0LL;
    unsigned long long y = x / 712ULL;
    cout << "y  = " << y << endl;
    cout << "z  = ";
    cin >> z;
    cout << "z^2 = " << z*z << endl;
}
```

Platformfüggetlen, rögzített méretű egész típusokat, és az értékhatárokat tároló makrókat találunk a <***cstdint***> fejállományban, melyek közül a leggyakrabban használtak:

| | |
|---|---|
| ***int8_t*** | **signed char** |
| ***uint8_t*** | **unsigned char** |
| ***int16_t*** | **short** |
| ***uint16_t*** | **unsigned short** |

| ***int32_t*** | **int** |
|---|---|
| ***uint32_t*** | **unsigned** |
| ***int64_t*** | **long long** |
| ***uint64_t*** | **unsigned long long** |

Az értékhatár makrók ***INTn_MIN***, ***INTn_MAX***, ***UINTn_MIN***, ***UINTn_MAX***, ahol *n* értéke 8,16, 32 vagy 64 lehet.

## 1.3 Új lebegőpontos értékek

A C++11 szabvány a **float**, **double** és a **long double** lebegőpontos típusok körét nem bővíti, azonban a lebegőpontos értékek területen új eszközöket kínál a programozók számára. Egyrészt lehetőséget biztosít a végtelen (***INFINITY***) és a „nem szám” (***NAN***) értékek kezelésére. Másrészt pedig a lebegőpontos konstans értékeket pontosabban adhatjuk meg hexadecimális formában (a mantisszát 16-os, míg a p/P betű után a 2 kitevőjét 10-es számrendszerben kell szerepeltetnünk.) Például, a *0x1.004p7* decimális értéke *128.125*, a *0x1Ab2P2* konstansé pedig *27336*.

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

int main() {
    cout << 0x1.ap+0f << endl;
    cout << 0x1.0P+10 << endl;
    cout << 0x0.C90FDAA22168CP2 << endl;
    cout << "1/0.0 = " << 1/0.0 << endl;
    cout << "-1/0.0 = " << -1/0.0 << endl;
    cout << "0.0/0.0 = " << 0.0/0.0 << endl;
    cout << "sqrt(-1.0) = " << sqrt(-1.0) << endl;
    cout << "INFINITY/INFINITY = " << INFINITY/INFINITY << endl;
}
```

A program futásának eredménye:

```
1.625
1024
3.14159
1/0.0 = inf
-1/0.0 = -inf
0.0/0.0 = nan
sqrt(-1.0) = nan
INFINITY/INFINITY = nan
```

## 1.4 Szigorúan típusos felsorolások – enum osztályok

A felsorolás osztályok használatával minden **enum** önálló típusként jelenik meg, melynek tagjait az **enum** nevével és a hatókör operátorral minősítve érhetjük el:

```
enum class Szin {FEKETE=0, KEK, ZOLD, PIROS=4};

int main() {
    Szin alap = Szin::FEKETE;
    if (Szin::PIROS == alap) {

    }
    alap = Szin(2);
    cout << int(alap) << endl; // 2
}
```

További lehetőség a felsorolás elemtípusának megadása:

```
enum class Szin : char {FEKETE=0, KEK, ZOLD, PIROS=4};
```

Az elemek típusaként tetszőleges előjeles vagy előjel nélküli egész típust szerepeltethetünk.

## 1.5 A nullptr és a nullptr_t

Hagyományosan a mutatók inicializálására a ***0*** vagy a ***NULL*** értékeket használtuk. A C++11 egyértelművé teszi ezt az állapotot a **nullptr** kulcsszó bevezetésével. A **nullptr** érték típusa a ***nullptr_t*** típus.

## 1.6 Sztring literálok

Ez idáig kétféle sztring konstans használhattunk a szabványos C++ nyelvben:

- Az első a szokásos " " közötti szöveg, ami **const char** tömbként tárolódik, 0-ás bájttal a végén.
- A második az L"..." jelöléssel használható,melynek eredményeként egy **const wchar_t** tömb jön létre, 0 értékű utolsó elemmel. (Emlékeztetőül a **wchar_t** a széles karakterek típusa.)

A széles sztringekben hagyományos, rögzített 16 bit kódolással tárolhatunk Unicode szöveget, azonban nem alkalmasak nagyobb kódméretű szövegek tárolására.

A C++11-ben három újabb Unicode kódolás támogatása is megtalálható: UTF-8, UTF-16 és UTF-32. Az UTF-32 a karakter kódját mindig 32 biten tárolja, míg az UTF-16 1 vagy 2 egységet, az UTF-8 pedig 1-4 egységet használ erre a célra.

A C++ programokban a karakterkódolással összefüggő kaotikus állapotok valamelyest csökkentek, azonban a felhasználás további kívánnivalókat hagy maga után. Az alábbi táblázatban foglaltuk össze a lehetőségeket:

| | ***karakter típus*** | ***karakter konstans*** | ***sztring típus*** |
|---|---|---|---|
| normál karakterek | **char** | `'A', '\x41'` | ***string*** |
| széles karakterek | **wchar_t** | `L'A', L'\x0041'` | ***wstring*** |
| UTF-8 kódolás | **char** | - | ***string*** |
| UTF-16 kódolás | **char16_t** | `u'A', u'\u0041'` | ***u16string*** |
| UTF-32 kódolás | **char32_t** | `U'A', U'\U00000041'` | ***u32string*** |

A 𝄞 (violinkulcs: U+1D11E) és a π (pi: U+03C0) jelet tartalmazó szövegek tárolása karakter tömbökben:

```
wchar_t cpw[] = L"violinkulcs : \U0001D11E, pi: \u03C0";
char cpu8[] = u8"violinkulcs : \U0001D11E, pi: \u03C0";
char16_t cpu16[] = u"violinkulcs : \U0001D11E, pi: \u03C0";
char32_t cpu32[] = U"violinkulcs : \U0001D11E, pi: \u03C0";
```

és sztring típusokban:

```
wstring sw = L"violinkulcs : \U0001D11E, pi: \u03C0";
string s8 = u8"violinkulcs : \U0001D11E, pi: \u03C0";
u16string s16 = u"violinkulcs : \U0001D11E, pi: \u03C0";
u32string s32 = U"violinkulcs : \U0001D11E, pi: \u03C0";
```

A különböző kódolású karaktersorozatok közötti átalakításokhoz használhatjuk a <*locale*> fejállomány ***wstring_convert***<> valamint a <*codecvt*> *include* fájl ***codecvt_utf8***<>, ***codecvt_utf16***<> és *codecvt_utf8_utf16*<> osztálysablonjait.

### *1.6.1 Nyers (raw) sztring literálok*

A hagyományos és a raw sztring literálok közötti alapvető különbség, hogy az utóbbiban az ESC-szekvenciákat nem dolgozza fel fordító.

```
int main() {
    setlocale(LC_ALL,"Hun");
    string normal_str="Első sor.\nmásodik sor.\nutolsó sor.\n";
    string raw_str=R"(Első sor.\nmásodik sor.\nutolsó sor.\n)";
    cout << normal_str << endl;
    cout << raw_str << endl;
}
```

A program futásának eredménye:

```
Első sor.
második sor.
utolsó sor.

Első sor.\nmásodik sor.\nutolsó sor.\n
```

A nyers szöveg kijelölésére egy legfeljebb 16-karakteres tagoló karaktersorozat is megadható:

```
string raw_str=R"tagol(Első sor.\nmásodik sor.\nutolsó sor.\n)tagol";
```

## 1.7 Felhasználó által definiált literálok

Az új C++ nyelvben lehetőség van saját literálok készítésére. Az új literált az utótagja alapján lehet azonosítani, definiálása pedig az "" operátor túlterhelésével történik:

```
típus operator"" utótag(unsigned long long n) {  }
típus operator"" utótag(long double d) {  }
típus operator"" utótag(const char* cstr) {  }
```

Az egész és lebegőpontos konstans értékek feldolgozását számként, illetve karaktersorozatként egyaránt elvégezhetjük:

```
inline double operator"" _fok (long double fok) {
    return fok*3.14159265/180;
}
double szog = 90.0_fok; // szog = 1.570796325

unsigned operator "" B(const char* bs) {
   unsigned b=0;
   for (int i=0; bs[i]; i++)
      b = b*2 + (bs[i]=='1');
   return b;
};
int maszk = 110011B; // maszk = 51
```

Karakter és sztring literálok esetén az alábbi formák használhatók, ahol a *chtípus* a különböző karter-típusokat jelöli (**char**, **wchat_t**, **char16_t**, **char32_t**).

```
típus operator"" utótag(char c) {  }
típus operator"" utótag(const chtípus* str, size_t strhossz) {  }
```

## 1.8 Fordítás idejű konstans kifejezések (constexpr)

A C++11 lehetővé teszi, hogy olyan konstansokat, függvényeket és osztályokat hozzunk létre, amelyek felhasználhatók más programelemek definiálása során. Ennek egyetlen feltétele, hogy a fordító által meghatározható konstans értékeket szolgáltassanak. Az ilyen konstansokat, függvényeket és változókat a **constexpr** kulcsszóval kell definiálnunk.

```
template<class T, int N>
constexpr int TombMeret(const T (&a)[N]) { // const is jó
    return N;
}

struct RGB {
    unsigned char r, g, b;
    constexpr RGB(unsigned char x, unsigned char y, unsigned char z):
                  r(x), g(y), b(z) {}
};

constexpr RGB operator+(const RGB& x, const RGB& y) { // const is jó
      return RGB(x.r + y.r, x.g + y.g, x.b + y.b);
}
```

```
int main() {
    constexpr double pi=3.14159265; // const is jó
    int a[int(pi+1)] = { 12, 23, 34, 45};
    int b[TombMeret(a)];
    constexpr RGB piros{255, 0, 0}; // const is jó
    constexpr RGB zold{0, 255, 0};  // const is jó
    constexpr RGB kek{0, 0, 255};   // const is jó
    RGB sarga = piros+zold;
    RGB cian = kek+zold;
}
```

A példában piros színnel jelöltük a fordító által ismert konstans kifejezéseket.

### *1.8.1 A static_assert kulcsszó*

A fenti megoldások használata során szükség lehet bizonyos ellenőrzések elhelyezésére a fordító által feldolgozott kódban. A fordításidejű ellenőrzésekre vezették be a **static_assert** kulcsszót (szemben az ***assert*** makróval, amely futás idejű vizsgálatokra szolgál):

```
static_assert(konstans kifejezés, sztring);
```

A fordító kiértékeli a *konstans kifejezést*, és megjeleníti a *sztring* üzenetet, ha a kiértékelés eredménye hamis. A fenti példában ellenőrizhetjük, hogy a pi konstans értéke 3 és 4 közé esik-e:

```
const double pi=1.14159265;
static_assert(pi>=3 && pi<=4, "Nem jó az érték");
```

A következő fordítási hibaüzenetet kapjuk: *„error: static assertion failed: Nem jó az érték”*.

## 1.9 Kivételek továbbításának tiltása (noexcept)

A C++11 szabvány szerint a függvényekben keletkező kivételek továbbításáról (a hívó felé) új módon is dönthetünk. A függvényfejben a **noexcept** (vagy **noexcept**(**true**)) előírást megadva, a kivételek nem továbbítódnak a függvényből, hatásukra a program futása megszakad. Az előírást elhagyva, illetve a **noexcept**(**false**) formában használva, a kivételek továbbítódnak.

A **noexcept** kulcsszó után zárójelben egy fordítás idejű kifejezés állhat, amely 0 vagy nem 0 értéke vezérli az előírás működését. Ebben a kifejezésben használhatjuk a ***noexcept()*** operátort, amely **true** értékkel tér vissza, ha az argumentuma nem továbbít kivételeket.

```
// Engedélyezi a kivételek továbbítását amennyiben a vector::operator[]
// is engedélyezi
void Ellenoriz(const vector<int>& v) noexcept(noexcept(v[0])) {
   if (v[0] > 0) cout << "OK\n";
}

// Engedélyezi a kivételek továbbítását amennyiben a vector::push_back(T)
// is engedélyezi
void HozzaAd(vector<int>& v, const int elem)
                                    noexcept(noexcept(v.push_back(elem))) {
    v.push_back(elem);
}

// Nem továbbítódik kivétel
int Meret(const vector<int>& v) noexcept {
    return v.size();
}
```

```
int main() {
   vector<int> v{1,2,3};
   try {
      Ellenoriz(v);
      HozzaAd(v,4);
      cout << Meret(v) << endl;
   }
   catch(...) {
      cout << "Kivetel történt" << endl;
   }
}
```

## 1.10 Változók memóriahatárra igazítása

A hagyományos C++ nyelvben a ***#pragma pack(****n)* előírás segítségével gondoskodhattunk a változók memóriahatárra való igazításáról. A C++11 az **alignas** kulcsszó bevezetésével egyszerűbbé teszi ezt a feladatot. Az alábbi 10-elemű tömb minden eleme 8-bájtos határon helyezkedik el:

```
alignas(double) int tomb[10];
```

Az *Adatok* struktúra minden példánya 16-bájtos határon kezdődik:

```
struct alignas(16) Adatok {
  float d[4];
};
```

A változók és a típusok esetén alkalmazott kiigazításról az **alignof** kulcsszó segítségével szerezhetünk információt:

```
cout << alignof(Adatok) << endl;
cout << alignof(tomb) << endl;
cout<< alignof(double) << endl;
```

A típusokkal a <*type_traits*> fejállomány ***alignment_of<>*** osztálysablonját is alkalmazhatjuk:

```
cout << alignment_of<Adatok>::value << endl;
cout << alignment_of<int>::value << endl;
cout << alignment_of<double>::value << endl
```

## 2 Új, szokatlan nyelvi megoldások

Ebben a csoportban egy sor olyan dolgot gyűjtöttünk csokorba, amelyek eddig idegenek voltak a C++ nyelvtől, bár más programozási nyelvekben találhatunk példákat:

- automatikus típus-meghatározás,
- jobbérték hivatkozások (&&),
- adatok áthelyezése (***forward*** és a ***move***),
- „okos” (*smart*) mutatók,
- egységes inicializálás,
- tartományalapú **for** ciklus, a ***begin*** és az ***end*** hívások,
- λ - lambda kifejezések (függvények),
- programkód párhuzamos futtatása,
  - aszinkron kódvégrehajtás,
  - szálak futtatása,
  - szálak szinkronizálása kizárással,
  - szálak szinkronizálása atomi változókkal.

### 2.1 Automatikus típus-meghatározás

A C++11 újradefiniálta az **auto** kulcsszó jelentését. Segítségével közölhetjük a fordítóval, hogy rábízzuk a kezdőértékkel ellátott változó típusának meghatározását. A fordító a jobb oldal típusát rendeli a változóhoz:

```
auto darab = 7;           // darab - int
auto pi = atan(1)*4.0;    // pi - double
auto p = new int[123];    // p - int *
```

Az **auto** kulcsszóval hagyományos függvényeket nem definiálhatunk, azonban az új visszatérési szintaxissal már alkalmazhatjuk:

```
auto Osszead(int a, double b) -> int {
    return a+b;
}
```

Az előző megoldással közeli rokonságban áll a **decltype** használata. A **decltype** segítségével egy már ismert típusú változó (kifejezés) típusát használjuk a deklaráláshoz:

```
int x = 7;
decltype(x) y;      // int y;
decltype(x) z = y; // ugyanaz, mint az auto z = y;
```

Az **auto** és a **decltype** együttes használatával az *Osszead()* függvényünk is értelmet nyer,

```
auto Osszead(int a, double b) -> decltype(a+b) {
    return a+b;
}
```

bár az igazi előnye a bemutatott megoldásoknak a sablonok használata során jelentkezik:

```
template <typename T1, typename T2>
auto Osszead(T1 a, T2 b) -> decltype(a+b) {
    return a+b;
}
```

Megjegyezzük, hogy minkét deklaráció előtt szerepelhetnek a **const** és a **volatile** típusminősítők.

Nézzünk egy olyan példát, amikor igazán segíti a programozó munkáját az automatikus típus-meghatározás:

```
// C++ 98 megoldás
int t[4] = {12, 23, 34, 45};
vector <int> v(t, t+4);
for (vector <int>::iterator vp = v.begin(); vp!=v.end(); vp++)
    cout << *vp << endl;
}

// C++11 megoldás az auto alkalmazásával
int t[4] = { 12, 23, 34, 45};
vector <int> v(t, t+4);
for (auto vp = v.begin(); vp!=v.end(); vp++)
    cout << *vp << endl;
```

## 2.2 Jobbérték hivatkozások (&&)

A C++ 98-ban a nem konstans referenciákat balértékekhez (lvalue) köthettük, a konstansokat pedig bal-, és jobbértékekhez egyaránt. Viszont nem volt semmi, amit hozzá lehetett volna kapcsolni egy nem konstans jobbértékhez:

```
int a = 7;
int& ar = a;
const long& br = a;
const int& cr = 12;
```

Ezen a helyzeten változtatott a C++11, ahol a hivatkozások általánosításaként tekinthetünk a && operátorra:

```
int&& xr = 23;
int&& yr = f();              // int f() { return 123;}
int&& zr = a;                // hiba, a lvalue!
const int&& wr = 123;

// de!

auto&& dr = a;               // balérték hivatkozás a-hoz
```

A jobbérték hivatkozások bevezetésének igazi értelmét a függvények paraméterezésében és a következő részben bemutatásra kerülő áthelyezés (*move*) szemantikában találjuk meg.

Példaként tekintsük az alábbi két túlterhelt függvényváltozatot:

```
const int& f(const int& a) {
    return a;
}

int&& f(int&& a) {
    return a+0;
}
```

Definiáljunk változókat és hívjuk meg az *f()* függvényt!

```
int x = 123, e;
e = f(x);       // f(const int & a)

e = f(1234);    // f(int&& a)

int& y = x;
e = f(y);       // f(const int & a)

int&& z = 7;
e = f(z);       // f(const int & a)

e= f(x+y*z);    // f(int&& a)
```

Láthatjuk, hogy balérték argumentum esetén az első függvény, míg jobbérték argumentummal a második aktiválódott.

## 2.3 Adatok áthelyezése (forward és a move)

A C++ programokban általában nagymennyiségű ideiglenes változó keletkezik/törlődik, amelyek többségét a fordító kezeli. A C++11 megoldásokat kínál az ideiglenes változók számának csökkentésére, ezzel növelve a futó program hatékonyságát. Az adatáthelyezés működését egyszerű példákon szemléltetjük, azonban a sablonok készítésénél nyújtanak igazán hathatós segítséget.

A ***forward()*** függvénysablon egy függvény paramétereinek továbbítását segíti, más belőle hívott függvényeknek:

```
void f(int& a) { a *= 10; }

int g(int b) { return b + 7; }

int h(int x) {
    f(forward<int&>(x));
    return g(forward<int>(x));
}

int main() {
    int a = 12;
    int b = h(a); // 127
}
```

A ***move()*** függvény sokkal szélesebb körben használhatjuk, elsősorban nagyobb méretű objektumok esetén.

```
int a[5] = {2, 3, 1, 7, 9};
vector<int> v1(a, a+5);
vector<int> v2 = v1;              // másolás
vector<int> v3(a, a+5);
vector<int> v4 = move(v3);        // áthelyezés
```

A műveletek eredményeként a *v3* elveszíti az adatait, hisz azok áthelyeződtek a *v4*-hez. A definíciós sor után a *v1*, *v2* és *v4* vektorok tartalmazzák az a tömb elemeit, míg a *v3* egy elemek nélküli vektor lett.

Amennyiben definiáljuk az alábbi *Csere()* függvénysablont, a vektorok felcserélése is gyorsan elvégezhető:

```
template <typename T> void Csere(T& x, T& y) {
     T z(move(x));    // a move konstruktor hívása
     x = move(y);     // a move operátor hívása
     y = move(z);     // a move operátor hívása
}
```

Amennyiben a *T* típus nem rendelkezik áthelyező konstruktorral és operátorral, a minden típushoz értelmezett másoló konstruktort és operátort hívja a fordító.

Osztályok esetén, az eddig ismert konstruktorok és a másoló operátor kibővült az áthelyező konstruktorral és operátorral. Egy saját készítésű dinamikus tömb osztály részletén mutatjuk be az elmondottakat:

```
class DTomb { // részlet
  public:
    // default és paraméteres konstruktor
    DTomb (int n = 12) : p( new int[n]), db(n)  {}

    // copy konstruktor
    DTomb (const DTomb& b) : p(new int[b.db]), db(b.db) {
        for (int i = 0; i < db; ++i)  p[i]=b.p[i];
    }

    DTomb (DTomb&& b) : p(b.p), db(b.db) { // move konstruktor
        b.db = 0; b.p = nullptr;
    }

    DTomb& operator=(const DTomb& b) { // copy operátor
        delete[] p;
        db = b.db;
        p = new int [db];
        for (int i = 0; i < db; ++i) p[i]=b.p[i];
        return *this;
    }

    DTomb& operator=(DTomb&& b) { // move operátor
        delete[] p;
        p = b.p; db = b.db;
        b.db = 0; b.p = nullptr;
        return *this;
    }

    ~DTomb() { delete[] p; }
  private:
    int *p, db;
};

int main() {
    DTomb a, b, c;
    DTomb d = a, e(b);              // másoló konstruktor
    DTomb f=move(a), g(move(b));    // áthelyező konstruktor
    c = e;                          // másoló operátor
    c = move(e);                    // áthelyező operátor
}
```

A ***move()*** függvény valójában egy jobbérték referenciát állít elő az argumentumához.

## 2.4 „Okos" (smart) mutatók

C++11 három új mutatótípust (osztálysablont) is bevezetett, amelyek a hivatkozások számlálásra és az elért memóriaterület automatikus felszabadítására is képesek (<*memory*> fejállomány).

***unique_ptr***: (egyedi mutató) akkor használjuk, amikor a lefoglalt memóriát nem osztjuk meg (azaz az osztály nem rendelkezik *copy* konstruktorral), azonban átvihető más unique_ptr-hez (létezik *move* konstruktor).

***shared_ptr***: (megosztott mutató) akkor alkalmazzuk, amikor a lefoglalt memóriát meg kívánjuk osztani (pl. másolással).

***weak_ptr***: (gyengén megosztott mutató) ***shared_ptr*** által kezelt objektumra hivatkozó referenciát tartalmaz, azonban nincs hatással a hivatkozás-számlálásra.

Az új mutatók bevezetésével az ***auto_ptr*** elavult lett, így használata nem javasolt. Az egyedi mutatók alkalmazását az alábbi példaprogram szemlélteti:

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

struct Pelda {
    int a;
    Pelda()      { cout << "Pelda::Pelda()\n";  }
    ~Pelda()     { cout << "Pelda::~Pelda()\n"; }
    void TagFv() { cout << "Pelda::TagFv()\n";  }
};

void KulsoFv(const Pelda& x) {
    std::cout << "KulsoFv(const Pelda&)\n";
}

int main() {
    unique_ptr<Pelda> p1(new Pelda);  // p1 a Pelda tulajdonosa
    if (p1) {
        p1->a = 123;
        p1->TagFv();
    }

    {
        unique_ptr<Pelda> p2(move(p1));  // p2 lett a Pelda tulajdonosa
        KulsoFv(*p2);
        p1 = move(p2);  // újra p1 a Pelda tulajdonosa
        cout << "p2 törlése ...\n";
    }

    if (p1) {
        p1->TagFv();
        cout << p1.get()->a << endl;
    }

    // a Pelda objektum megszüntetése
}
```

A program kimenete jól mutatja az ***unique_ptr*** sablonok működését:

```
Pelda::Pelda()
Pelda::TagFv()
KulsoFv(const Pelda&)
p2 törlése ...
Pelda::TagFv()
123
Pelda::~Pelda()
```

A megosztott memóriaterületeket is kezelhetünk a programjainkban:

```
int main() {
    // megosztott mutatóhoz területfoglalás
    auto p = make_shared<int>(123);
    weak_ptr<int> wp = p;
    {
      // megosztott mutató készítése a hivatkozott objektumhoz
      auto sp = wp.lock();
      cout << *sp << endl;  // 123
    }
    // a hivatkozott objektum tulajdonjogának törlése, ha ez az utolsó
    // shared hivatkozás volt, az objektum is törlődik
    p.reset();
    // a hivatkozott objektum létezésének vizsgálata
    if(wp.expired())
      cout << "már törlődött" << endl;
}
```

## 2.5 Egységes inicializálás

A C++98 nyelvben csak korlátozottan tudtunk kezdőértéket adni a több érték tárolására alkalmas változóknak. A C++11 egységesítette a kapcsos zárójelek között megadott, vesszővel tagolt értéklistával történő inicializálást. Ezzel a megoldással lényegesen egyszerűbbé válik az STL tárolók használata:

```
#include <vector>
#include <iostream>
#include <map>
#include <set>
#include <string>
using namespace std;

long Osszeg(const vector<int>& v) {
    long s = 0;
    for (int e : v) // lásd 2.6 fejezet
        s+=e;
    return s;
};

int main() {
    int tomb1[] {1,2,3,4,5};
    int tomb2[] = {1,2,3,4,5};
    vector<int> v1 {1,2,3,4,5};
    vector<int> v2 = {1,2,3,4,5};
    set<int> s1 {1,2,3,4,5};
    set<int> s2 = {1,2,3,4,5};
    map<int,string> m1 { {0,"nulla"}, {1,"egy"}, {2,"kettő"} };
    map<int,string> m2 = { {0,"nulla"}, {1,"egy"}, {2,"kettő"} };
    cout << Osszeg(v1) << endl;
    cout << Osszeg({10,20,30,40}) << endl;
}
```

A kezdőérték-adás fenti módja természetesen más, akár egyértékű változókkal is sikeresen alkalmazható. Külön felhívjuk a figyelmet a dinamikus helyfoglalású tömb inicializálására:

```
int main() {
    double b {};        // a típus alapértékével inicializál: 0.0
    char ct[7] {};      // ct minden eleme '\0' lesz
    float *fp {};       // a nullptr értéket veszi fel
    int a {123};
    string s {"C++11"};
    long *p = new long[3] {12,23,34};
    delete []p;

    struct {
        int kor;
        string nev;
    } diak {9, "Iván"};
}
```

Saját készítésű osztályainkat egy ***initializer_list<>*** típusú paramétert fogadó konstruktorral felkészíthetjük az inicializáló lista feldolgozására.

## 2.6 Tartományalapú for ciklus, a begin és az end hívások

A C++11 bevezette a más nyelvekből ismert *foreach* vezérlési szerkezetet. (Nem keverendő össze a szabványos könyvtár ***for_each()*** függvényével, amely egy függvényobjektumot hajt végre a megadott intervallumon!) A ciklus formája:

```
for ( elem_deklaráció : tartomány_kifejezés) utasítás;
```

ahol a *tartomány_kifejezés* helyén olyan kifejezés állhat, melyre a szintén újonnan bevezetett ***begin()*** és ***end()*** függvények egyértelműen azonosítják (mutatókkal/iterátorokkal) a tartomány elejét és végét.

A ciklus futása során egy mutató/iterátor végigmegy a fentiekben kijelölt tartományon, és minden lépésében az *elem_deklarációban* megadott változón keresztül érhetjük el a meghivatkozott elemet.

Példaként vegyük a hagyományos (statikus) tömböket:

```
int a[] = {2, 7, 10, 11, 12, 23, 29, 30};
```

Az e változó sorban felveszi az a tömb minden elemének értékét:

```
for (int e : a) cout << e << " " ;
```

illetve hivatkozását (ekkor e tömb elemei módosíthatók):

```
for (int& e : a) e++;
```

Hasonló megoldásokhoz jutunk, amennyiben az elemtípus megállapítását a fordítóra bízzuk:

```
for (auto e : a) cout << e << " " ;
for (auto& e : a) e++;
```

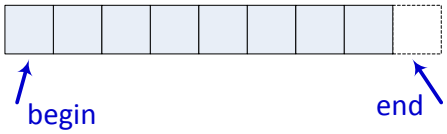
Szükség esetén a **const** és **volatile** típusminősítők is alkalmazhatók:

```
for (const auto e : a) cout << e << " " ;
```

A *tartomány_kifejezésben* elsősorban a már ismert és az új STL tárolókat használjuk, de alkalmazhatunk tömböket (***array***, ***valarray***) és inicializáló listákat is

```
vector <double> v = {1.2, 2.3, 3.5};
for (auto e : v) cout << e << " ";
array<double,3> ar = {1.2, 2.3, 3.5};
for (auto e : ar) cout << e << " ";
valarray<double> var = {1.2, 2.3, 3.5};
for (auto e : var) cout << e << " ";
for (auto e : { 12, 23, 34, 45} ) cout << e << " ";
```

Mint említettük, a **for** ciklussal bejárt tartomány határait a ***begin*** és az ***end*** hívásokkal kérdezi le a fordító, amelyek osztályok esetén a ***begin()*** és az ***end()*** tagfüggvények hívását jelenti. Minkét függvény általánosított mutatókkal (iterátorokkal) tér vissza, melyek a tárolt adatok elejére, illetve a végét követő pozícióra hivatkoznak:



Ezek ismeretében magunk is készíthetünk tároló osztályokat, amelyek például dinamikus helyfoglalású tömbök esetén lehetővé teszik a tartományalapú **for** ciklus alkalmazását. A példában a *Fibo* osztály konstruktora adott *n* méretű dinamikus tömbben tárolja a Fibonacci számsor első *n* elemét.

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

class Fibo {
    int *fb = nullptr;
    int db = 0;
  public:
    Fibo(int n = 7) {
       db = n<2 ? 2 : n;
       fb = new int[db] {0,1};
       for (int i=2; i<db; i++)
         fb[i] = fb[i-1] + fb[i-2];
    }
    ~Fibo() { delete[] fb;}
    int* begin() {return fb;}      // mutató a tömb elejére
    int* end() {return fb+db;}     // mutató a tömb utáni pozícióra
};

int main() {
    Fibo fibo(12);
    for (int a: fibo)
        cout << a << ' ';

    for (int *p=begin(fibo); p!=end(fibo); p++)
        cout << *p << ' ' ;
}
```

A ***begin*** és az ***end*** sablonokat az STL konténereivel is sikeresen alkalmazhatjuk, például az értéktömbbel (***valarray***):

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

int main() {
    valarray<double> va {25, 9, 36, 4};
    sort(begin(va), end(va));
    for (auto e : va)
        cout << e << ' '; cout << endl; // 4 9 25 36
    valarray<double> vb = sqrt(va);
    for (auto e : vb)
        cout << e << ' '; cout << endl; // 2 3 5 6
    cout << va.sum() << endl;           // 74
    cout << vb.max() << endl;           // 6
    va = va + vb;
    for (auto e : va)
        cout << e << ' '; cout << endl; // 6 12 30 42
    fill(begin(va), end(va), 0);
    for (auto e : va)
        cout << e << ' '; cout << endl; // 0 0 0 0
}
```

## 2.7 λ - lambda kifejezések (függvények)

A lambda függvények lehetővé teszik, hogy egy vagy többsoros névtelen függvényeket definiáljunk a forráskódban, ott ahol éppen szükség van rájuk. A lambda kifejezések szerkezete nem kötött, a fordító feltételezésekkel él a hiányzó részekkel kapcsolatban. Az alábbi példában a piros színnel kiemelt rész maga a lambda függvény

```
int a = [] {return 12*23;} ();
```

A bevezető szögletes zárójelpár jelzi, hogy lamda (kifejezés) következik. Ez után áll a függvény törzse, amely **return** utasításából a fordító meghatározza függvény értékét és típusát. Az utasítást záró kerek zárójelpár a függvényhívást jelenti.

Amennyiben paraméterezni kívánjuk a lambdát, a fenti két rész közé egy hagyományos paraméterlista is beékelődik:

```
int b = [] (int x, int y) { return x*y;} (12,23);
```

Szükség esetén a függvény típusát is megadhatjuk a C++11-ben bevezetett formában:

```
double c = [] (int x, int y) -> double { return x*y;} (12,23);
```

Amennyiben egy lambda függvényt többször szeretnénk hívni, hozzárendelhetjük egy függvénymutatóhoz. Az alábbi példában egy tabellázó függvény argumentumaként használunk lambdákat:

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

void Tabellaz(double a, double b, double dx, double(*pfv)(double))
{
    for (double x = a; x <= b; x += dx)
        cout << x << "\t" << pfv(x) << endl;
}

double (*fvptr1)(double) = [](double x)-> double {
                                return x*sqrt(x);
                            };

auto fvptr2 = [](double x)-> double {return log(x)*sqrt(x);};

int main() {
    Tabellaz(1, 2, 0.2, [](double x) {return x*x;});
    Tabellaz(2, 3, 0.1, fvptr1);
    Tabellaz(3, 4, 0.2, fvptr2);
}
```

A lambda függvények legfontosabb alkalmazási területe az STL algoritmusok hívása. Az alábbi utasítások a *vector* és az *algorithm* fejállományok beépítése után működőképesek:

```
vector <int> adatok = { 1, 1, 2, 3, 5, 8, 13, 21 };

int db= count_if(begin(adatok), end(adatok), [](int e){return e%2;} );

cout << "Páratlan elemek száma: " << db << endl;

for_each(begin(adatok), end(adatok), [](int& e) { e = 2*e;} );

sort(begin(adatok), end(adatok), [](int e1, int e2) { return e1>e2;} );

for_each(begin(adatok), end(adatok), [](int e) { cout << e << " ";} );
```

Először megszámláljuk az adatok vektor páratlan elemeit, majd minden elemet a duplájára növelünk, csökkenő sorrendben rendezzük a vektort, végül pedig megjelenítjük az elemeket.

A fenti példáinkban a lambda függvények csak a paramétereken keresztül tartották a kapcsolatot a környezetükben (hatókörükben) elérhető változókkal. Ellentétben a hagyományos függvényekkel, a lambda kifejezésekben elérhetjük a lokális hatókör változóit.

A fájlszintű és lokális, statikus élettartamú nevek elérése minden további nélkül működik:

```
double a = 3.14159265;

int main() {
    static int b = 11;
    double x = [] () { return a+b;} ();
}
```

A lokális, nem statikus függvényváltozók, illetve az osztály adattagjai esetén intézkedhetünk az elérés módjáról. Amennyiben a fenti példában az *a* és *b* lokális változók, a velük azonos hatókörben megadott lambda a következőképpen módosul:

```
int main() {
    double a = 3.14159265;
    int b = 11;
    double x = [a,b] () { return a+b;} ();
    cout << x << endl;
}
```

A lambdát és az elért változókat együtt szokás bezárásnak (*closure*) nevezni, míg a felhasznált változókra, mint elkapott (*captured*) változókra hivatkozunk. A változókat értékük szerint és hivatkozásukkal egyaránt elkaphatjuk, az alábbi példákban bemutatott módon:

| | |
|---|---|
| `[]` | Egyetlen helyi változót sem kívánunk elkapni. |
| `[=]` | Az összes helyi változót érték szerint kapjuk el. |
| `[&]` | Az összes helyi változót referenciájával kapjuk el. |
| `[a, b]` | Csak az *a* és *b* változókat kapjuk el, érték szerint. |
| `[a, &b]` | Az *a* változó érték szerint, a b változót azonban a referenciájával kapjuk el. |
| `[=, &b]` | Az összes helyi változót érték szerint kapjuk el, kivéve a *b* változót, amelyet a hivatkozása révén. |
| `[&, a]` | Az összes helyi változót a referenciájukkal kapjuk el, kivéve az *a* változót, amelyet érték szerint. |
| `[this]` | Az osztályon belül definiált lambda kifejezésekben használhatjuk a **this** mutatót, vagyis elérhetjük az osztály tagjait. |

A fordító alaphelyzetben nem engedi az érték szerint elkapott változók értékének módosítását. Ha azonban a függvény törzse előtt megadjuk a **mutable** kulcsszót, a módosítás elvégezhető, azonban érvényessége a lambdára korlátozódik.

```
int main() {
    int x = 7, y = 12;
    int z = [=] () mutable { ++x, ++y; return x*y;} ();
    cout << x << y << z << endl; // 7 12 104
}
```

Referenciával történő elkapás esetén egészen más a helyzet:

```
int main() {
    int x = 7, y = 12;
    int z = [&] () { ++x, ++y; return x*y;} ();
    cout << x << y << z << endl; // 8 13 104
}
```

A lambda függvények esetén is szabályozhatjuk a kivételek továbbítását. Alaphelyzetben minden kivétel továbbítódik a hívó függvénynek:

```
auto lambda1 = []() { };
```

De igény szerint korlátozhatjuk a továbbított kivételeket:

```
auto lambda3 = []() throw(int, char*) { };
```

vagy akár le is tilthatjuk azok továbbítását:

```
auto lambda2 = []() throw() { };
```

A C++11 nyelvben bevezetett ***function***<> sablon (*functional* fejállomány) nagyban segíti a lambda függvények használatát. Segítségével a tabellázó programunk jóval olvashatóbb formát ölt:

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

void Tabellaz(double a, double b, double dx,
                                    function<double(double)> fv) {
    for (double x = a; x <= b; x += dx)
        cout << x << "\t" << fv(x) << endl;
}

function<double(double)> fv1 = [](double x)-> double {
                                    return x*sqrt(x);
                                };

function<double(double)> fv2 = [](double x)-> double
                                        {return log(x)*sqrt(x);};

int main() {
    Tabellaz(1, 2, 0.2, [](double x) {return x*x;});
    Tabellaz(2, 3, 0.1, fv1);
    Tabellaz(3, 4, 0.2, fv2);
}
```

A ***function***<> sablon felhasználásával rekurzív lambdát is készíthetünk:

```
int main() {
    function<int(int)> fakt = [&fakt] (int n) {
                        if (n < 2) return 1; else return n*fakt(n-1);};
    cout << fakt(5) << endl; // 120
}
```

## 2.8 Programkód párhuzamos futtatása

A párhuzamosság (*concurrency*) és a többszálúság (*multithreading*) azt jelenti, hogy programunk bizonyos részei (függvényei) egyszerre hajtódnak végre. Többmagos processzorok világában ezzel jelenősen növelhetjük programunk hatékonyságát.

A C++98 futtató környezete egyetlen programszál futtatására volt felkészítve. Ennek következtében a többszálúság bevezetéséhez a C++11 futtató környezetét alapvetően át kellet dolgozni. A nyelv definíciója csupán a **thread_local** tárolási osztály bevezetésével támogatja a többszálú programok készítését, az igazi „tudást” C++ sablontár elemeivel érjük el. Egy **thread_local** változó a rá hivatkozó programszál indításakor jön létre, és a szál befejezésekor megszűnik, így minden szál saját másolattal rendelkezik belőle.

Mint említettük, a C++11 gazdag könyvtárat biztosít a többszálú programok készítéséhez az <*atomic*>, a <*thread*>, a <*mutex*>, a <*future*> és a <*condition_variable*> fejállományok definícióival.

A párhuzamosan indított függvények futhatnak egymástól függetlenül, aszinkron módon (*async*), illetve egymáshoz szinkronizálva (*thread*). Az aszinkron függvények, amelyek visszatérési értéke elérhető, jól használhatók bizonyos számítások elvégzéséhez, illetve perifériák kezelésére. Bármelyik esetet is választjuk, a futtatni kívánt függvényt (funkcionált) többféleképpen is megadhatjuk:

- függvénymutatóval (a függvény nevével),
- tagfüggvény mutatóval,
- lambda kifejezésként,
- függvény objektummal (amely definiálja az ***operator()*** tagfüggvényt).

### *2.8.1 Aszinkron kódvégrehajtás*

Függvények aszinkron futtatását a <*future*> fejállományban deklarált ***async()*** függvénysablon segítségével kezdeményezhetjük.

A függvénysablonnak első argumentumként a futtatni kívánt kódot tartalmazó funkcionált adjuk át, a további argumentumok pedig a hívott függvényhez továbbítódnak. (Hivatkozással történő paraméterátadáshoz az argumentumokat a ***ref()*** függvénysablonnal kel beburkolni.)

```
future<void> fv = async(Kiir, "async: ");
```

Az ***async()*** sablon másik változatában az első argumentum a futtatás módját határozza meg:

```
auto fv = async(launch::deferred, Kiir, "async: ");
```

ennek lehetséges értékei:

- *launch::async* – a kód egy új programszálban fut (ez az alapértelmezés),
- *launch::deferred* – a kód az aktuális szálban hajtódik végre aszinkron módon („lusta végrehatás”).

Az ***async()*** hívás egy ***future***<*függvénytípus*> objektummal tér vissza, mely objektumon keresztül elérhetjük az aszinkron függvény visszatérési értékét (a ***get()*** tagfüggvény hívásával), valamint a futás során keletkezett kivételeket.

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

void Kiir(string const& str) {
    for (int i=0; i<100; i++)
        cout << str << i << endl;
}

int main() {
    future<void> fv = async(Kiir, "async: ");
    for (int i=0; i<100; i++)
        cout << "main:  " << i << endl;
    fv.wait();
}
```

```
main:  0
async: 0
main:  1
async: 1
async: 2
main:  2
async: 3
main:  3
async: 4
main:  4
async: 5
async: 6
main:  5
async: 7
async: 8
main:  6
   …
```

A fenti példában, a főszálban (*main()*) és az aszinkron szálban (*Kiir()*) egy-egy **for** ciklus fut, melyek futási eredményének első pár sora a jobb oldali oszlopban látható. Az aszinkron kód befejezését a ***future*** *<>* osztálysablon ***wait()*** tagfüggvényével várhatjuk meg.

További lehetőségeket biztosítanak a *wait_for()* és a *wait_until()* tagfüggvények, amelyek a ***future_status*** felsorolás elemeivel jelzik, hogy az eredmény rendelkezésre áll-e adott idő múlva, illetve adott időpont elérését követően, vagy sem.

```
#include <future>
#include <iostream>
#include <string>
#include <chrono>
using namespace std;

int main() {
   cout << "várakozik..." << endl;
   int allapot;
   future<int> ft = async(launch::async, [] () {
                        this_thread::sleep_for(chrono::seconds(2));
                        return 123;
                    });
   do {
        allapot = ft.wait_for(chrono::milliseconds(500));
        if (allapot == future_status::deferred)
           cout << "még nem indult el" << endl;
        else if (allapot == future_status::timeout)
           cout << "timeout" << endl;
        else if (allapot == future_status::ready)
           cout << "kész" << endl;
    } while (allapot != future_status::ready);
   cout << "az eredmény " << ft.get() << '\n';
}
```

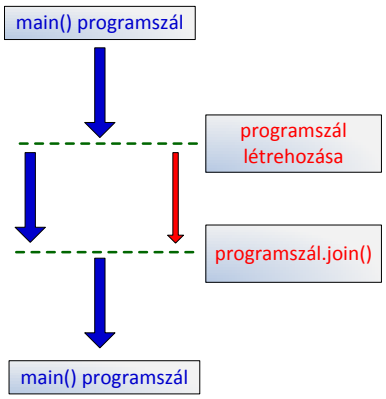
A program futásának eredménye:

```
várakozik...
timeout
timeout
timeout
kész
az eredmény 123
```

A <*future*> fejállomány további osztálysablonjai (***promise<>***, ***packaged_task<>***, ***shared_future<>*** stb.) bővített lehetőségekkel támogatják az aszinkron kódrészletekből felépülő programok széleskörű alkalmazását.

### *2.8.2 Szálak futtatása*

Futtatható szálat a ***thread*** osztály objektumaként hozzuk létre, amely a létrehozással el is indul. A szálobjektum ***get_id()*** tagfüggvényének hívásával megtudhatjuk a szál azonosítóját, a ***join()*** hívásával megvárhatjuk a szál lefutását, míg a ***detach()*** hívással leválaszthatjuk a futó szálat a szálobjektumról.




Felhívjuk a figyelmet arra, hogy nincs mód a szálban futó függvény visszatérési értékének elérésére! (Szükség esetén a függvényből referencia paraméterek segítségével nyerhetünk ki adatot.)

A fenti aszinkron példaprogram programszálat használó változata:

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

void Kiir(string const& str) {
    for (int i=0; i<100; i++)
        cout << str << i << endl;
}

int main() {
    thread td (Kiir, "thread: ");
    for (int i=0; i<100; i++)
        cout << "main:   " << i << endl;
    td.join();
    cout << "main thread ID: "
         << this_thread::get_id() << endl;
    cout << "Kiir thread ID: " << td.get_id() << endl;
}
```

```
thread: 0
main:   0
main:   1
thread: 1
thread: 2
thread: 3
thread: 4
thread: 5
thread: 6
main:   2
thread: 7
thread: 8
main:   3
main:   4
main:   5
main:   6
main:   7
main:   8
main:   9
    …
```

Az aktuális szálra (a szálat megvalósító függvényen belül) a ***this_thread*** névtér függvényeivel is hivatkozhatunk:

- lekérdezhetjük a szál azonosítóját — ***get_id()***,
- javasolhatjuk, hogy vezérlés a száltól adódjon át más szálakhoz: — ***yield()***,
- megadott ideig felfüggeszthetjük a szál futását: — ***sleep_for***(*időtartam*),

```
#include <chrono>
#include <thread>
using namespace std::chrono;
int main() {
    std::this_thread::sleep_for(seconds(10));
}
```

- adott időpontig is blokkolhatjuk a szál futását: — ***sleep_until***(*időpont*).

```
#include <chrono>
#include <thread>
using namespace std::chrono;
int main() {
  std::this_thread::sleep_until(system_clock::now() + seconds(10));
}
```

### *2.8.3 Szálak szinkronizálása kizárással*

A ***mutex*** (*mutual exclusion* – közös kizárás) objektum több szál által használt, osztott erőforrások védelmére alkalmas eszköz. Segítségével a szálon belül kritikus szekciót hozhatunk létre, amelyet egyszerre csak egy szál érhet el. A futó szál mellett a többi, ugyanazt a függvényt végrehajtó szál várakozik.

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

void fv(char t, mutex& m) {
   for (auto i = 0; i < 5; i++) {
      m.lock();
      cout << t << ": " << i << endl;
      m.unlock();
   }
}

int main() {
    mutex m;
    thread t1(fv, 'A', ref(m));
    thread t2(fv, 'B', ref(m));
    thread t3(fv, 'C', ref(m));
    t1.join();
    t2.join();
    t3.join();
}
```

```
mutex nélkül:
    A: 0
    B: 0
    B: 1
    B: 2
    B: 3
    B: 4
    C: 0
    C: 1
    C: 2
    A: 1
    A: 2
    A: 3
    A: 4
    C: 3
    C: 4
```

```
mutex-szel:
    A: 0
    B: 0
    C: 0
    A: 1
    B: 1
    C: 1
    A: 2
    B: 2
    C: 2
    A: 3
    B: 3
    C: 3
    A: 4
    B: 4
    C: 4
```

A kizárás elvégzésére több, a <*mutex*> fejállományban deklarált osztály közül is választhatunk, melyek neve utal a felhasználási módjukra: ***mutex***, ***recursive_mutex***, ***timed_mutex***, ***recursive_timed_mutex***. Mindegyik osztály biztosítja számunkra a ***lock()***, ***try_lock()*** és az ***unlock()*** tagfüggvényeket.

```
mutex m;
if (m.try_lock() == true) {
    cout << " Megtörtént a kizárás" << endl;
    m.unlock();
} else
    cout << "Sikertelen kizárás" << endl;
```

A rekurzív *mutex*-et akkor használjuk, ha egy szálban több kizárás is előfordul. Normál kizárás (***mutex***) esetén az ilyen szálak nem tudják befejezni a futásukat (előáll *deadlock* állapot). Az időzített kizáró objektumokkal adott ideig ***try_lock_for()*** vagy adott időpontig ***try_lock_until ()*** próbálkozhatunk a kizárással.

További lehetőséget biztosít a ***lock_guard<>*** osztálysablon használata, melynek konstruktorában jön létre a kizárás, míg a destruktorában automatikusan meg is szűnik az.

```
int n = 0;
mutex n_mutex;

void BiztonsagosLeptetes() {
    lock_guard<mutex> kizaras(n_mutex);
    ++n;
}
```

További megoldást ad a kezünkbe az ***unique_lock<>*** , az univerzális kizáró osztálysablon, amely a fent bemutatott megoldásokon túlmenően lehetővé teszi a kizárási stratégia (*defer_lock*, *try_to_lock*, *adopt_lock*) megadását is. Az ***unique_lock<>*** sablonnal együtt használhatjuk a <*conditional_variable*> fejállományban deklarált a ***conditional_variable*** osztályt, amely a kizárást értesítésküldési lehetőséggel vértezi fel.

### *2.8.4 Szálak szinkronizálása atomi változókkal*

A C++11 az <*atomic*> fejállományban definiált ***atomic<>*** osztálysablonnal, és annak egészekre specializált változataival támogatja az atomi műveletek végzését. Az atomi műveletek oszthatatlanok, végrehajtásuk nem szakadhat meg. Ezáltal az atomi változók és a rajtuk végzett műveletek szál biztosak (*thread-safe*), így felhasználhatók a szálak automatikus szinkronizálására.

Az alábbi példában definiált *Szamlalo<>* osztálysablon tagfüggvényei szálbiztos módon végzik a *szam* tagjának léptetését, értékének beállítását és lekérdezését:

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

template <typename T>
struct Szamlalo {
    Szamlalo(T e) { Set(e); }
    atomic<T> szam;
    void Novel() { ++szam; }
    void Csokkent() { --szam; }
    T Get() { return szam.load(); }
    void Set(T e) { szam.store(e); }
};

int main() {
    Szamlalo<int> n(12);
    for(int i=0; i<112; i++) n.Novel();
    n.Csokkent();
    cout << n.Get() << endl;
}
```

Az utolsó, több szálat futtató példában a szálakkal két nagy vektor skaláris szorzatát számoljuk, felosztva a tömb elemeit a szálak között:

```
#include <iostream>
#include <thread>
#include <vector>
#include <atomic>
using namespace std;

atomic<int> eredmeny(0);

void SkalarSzorzat(const vector<int> &v1, const vector<int> &v2,
                   int kezdo, int vegso) {
    int reszOsszeg = 0;
    for(int i = kezdo; i < vegso; ++i) {
        reszOsszeg += v1[i] * v2[i];
    }
    eredmeny += reszOsszeg;
}

int main() {
    const int elemszam =100000;
    const int szalszam = 5;
    vector<thread> szalak;
```

```
    // v1={2,2,2,2,...,2},  v2={3,3,3,3,...,3}
    // A skalárszorzat eredménye 600000
    vector<int> v1(elemszam,2), v2(elemszam,3);

    // Az elemek felosztása a szálak között
    vector<int> hatarok;
    for (int n=0; n<szalszam; n++)
        hatarok.push_back(n*elemszam/szalszam);
    hatarok.push_back(elemszam);

    // A szálak indítása:
    for (int i = 0; i < szalszam; i++)
        szalak.push_back(move( thread(SkalarSzorzat, cref(v1),
                               cref(v2), hatarok[i], hatarok[i+1])) );

    // A szálak lefutásának bevárása
    for (auto &t : szalak)
        t.join();
    cout << eredmeny << endl;
}
```

## 3. Változások az osztályok és a sablonok körében

A C++11 nyelv összes lehetőségét mérlegre helyezve, a mérleg nyelve ebben a csoportban bemutatott megoldások oldalára billen. Önmagukban az osztályok készítésének szabályai ugyan alig változtak, azonban a sablonok szintjén történő fejlesztés előtt új távlatok nyíltak:

- konstruktorok delegálása, adattagok inicializálása, **sizeof** adattagra,
- operátorfüggvények explicit típus-átalakítással,
- az **override** és a **final** módosítók,
- a **delete** és a **default** tagfügvények,
- korlátozások nélküli union típus,
- sablonok és a C++11,
    - szintaxis,
    - külső sablonok,
    - sablon álnevek (*template alias*),
    - sablonok változó hosszúságú argumentumlistával (*variadic templates*),
    - hátravetett visszatérési típus (*suffix return types*).

### 3.1 Konstruktorok delegálása, adattagok inicializálása, sizeof adattagra

A konstruktorok delegálása lehetővé teszi, hogy ugyanazon osztályon belül az egyik konstruktorból egy másikat hívjunk. A másik konstruktor hívását a konstruktorfejben, az inicializáló listán kell elhelyeznünk.

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

class Pont {
    int x,y;
  public:
     Pont(int a, int b) :x(a), y(b) {}
     Pont() : Pont(0,0) {}
     Pont(Pont& p) : Pont(p.x, p.y) {}
     // ...
};

int main() {
   Pont a, b(12,23);
   Pont c {b};
}
```

Ez a megoldás kellő odafigyelést igényel, hisz nem ajánlott rekurzív konstruktorhívásba keverednünk.

Nem túl lényeges újdonság, azonban érdemes tudni róla, hogy ezentúl a **sizeof** művelet az osztályok adattagjaira is elvégezhető, az osztályok példányosítása nélkül. Ugyancsak ritkán használt megoldásként a C++11 biztosítja, hogy az osztályok nem statikus és nem konstans adattagjait kezdőértékkel lássuk el az osztálydefiníción belül.

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

class Adattar {
  private:
     double d=12;
  public:
     long b=123;
     char v[7] = {'O', 'O', '\0'};
     void Kiir() {
        cout << sizeof(Adattar::d) << endl; // 8
     }
```

```
};
int main() {
   cout << sizeof(Adattar::b) << endl;  // 4
   cout << sizeof(Adattar::v) << endl;  // 7
   Adattar a;
   cout << sizeof(a.b) << endl;         // 4
   a.Kiir();
}
```

### 3.2 Operátorfüggvények explicit típus-átalakítással

A C++ 98 az **explicit** kulcsszóval azt jelöli, hogy egy konstruktor nem használható implicit konverzióra. Az új szabvány lehetővé teszi, hogy a kulcsszót konverziós operátorokkal is alkalmazzuk, kizárva ezzel az implicit típus-átalakításokat.

Az alábbi *Komplex* osztályban példát látunk az explicit kulcsszó minkét alkalmazására. Az explicit átalakítások kijelölésével készített ***main()*** függvény természetesen a kevésbé szigorú esetben is jól működik. (A lap jobb oldalán az implicit átalakításokkal készített ***main()*** függvény látható.)

```
class Komplex {
 public:
   Komplex(double a, double b) {re=a; im=b;}
   // konverziós konstruktor
   explicit Komplex(double a) { re=a; im=0; }
   // konverziós operátor
   explicit operator double() const {return sqrt(re*re+im*im);}
 private:
  double re, im;
};
int main() {
  Komplex a(3,4);
  Komplex b(123.0);
  double x = static_cast<double>(a);
  double y = double(b);
  int c = double(Komplex(123));
}
```

```
int main() {
  Komplex a(3,4);
  Komplex b = 123.0;
  double x = a;
  double y = b;
  int c = Komplex(123);
}
```

### 3.3 Az override és a final módosítók

A C++ nyelven az osztályok tagfüggvényeinek virtuális volta nem kötelező (mint például a Java nyelvben). A programozó a **virtual** kulcsszó segítségével jelöli ki a virtualitás kezdetét egy öröklési láncon belül.

Amennyiben a virtuális tagfüggvényt tartalmazó alaposztályból új osztályt származtatunk, és ott a virtuális függvénnyel megegyező nevű függvény készítünk, két eset áll fenn:

- ha az új függvény prototípusa megegyezik a virtuális függvényével, a fordító lecseréli az alaposztálybeli virtuális függvényt az új függvénnyel (újradefiniálás, *overriding*),
- ha azonban a paraméterek típusa eltérő, az újradefiniálás helyett a túlterhelés (*overloading*) mechanizmusa érvényesül, és mindkét függvény elérhető lesz az új osztályban. Az alaposztály mutatóján keresztül hivatkozva az új objektumra, az alaposztálybeli függvény hívódik meg. Erre látunk példát az alábbi egyszerű programban:

```
class B {
  public:
    virtual void fv(int) {cout << "B::fv()" << endl;}
};
```

```
class D : public B {
  public:
     void fv(long) {cout << "D::fv()" << endl;}
};

int main() {
   B b;
   B *p = &b;
   p->fv(12);      // B::fv()
   D d;
   p = &d;
   p->fv(23);      // B::fv()
}
```

Amennyiben a *D* osztály **int** típusú paraméterrel definiálja a *D::fv()* tagfüggvényt, a második híváskor a D-beli *fv()* aktivizálódik.

Az újonnan bevezetett **override** módosítóval jelezhetjük a fordítónak, hogy virtuális függvényt szeretnénk újradefiniálni. Ha azonban ennek feltételei nem állnak fenn (pl. eltérő paramétertípusok), fordítási hibát kapunk.

```
class D : public B {
  public:
     void fv(long) override {cout << "D::fv()" << endl;}
};
```

A **final** módosítóval pedig épp ellenkezőleg, megakadályozhatjuk az adott virtuális függvény lecserélését a származtatott osztályokban:

```
class B {
  public:
    virtual void fv(int) final  {cout << "B::fv()" << endl;}
};
```

A két módosító megfelelő kontextusban együtt is használható, így egy későbbi származtatás során meggátolva a lecserélt virtuális függvény további újradefiniálását:

```
class B {
  public:
    virtual void fv(int) {cout << "B::fv()" << endl;}
};

class D : public B {
  public:
    void fv(int) override final {cout << "D::fv()" << endl;}
};

int main() {
   B b;
   B *p = &b;
   p->fv(12);      // B::fv()
   D d;
   p = &d;
   p->fv(23);      // D::fv()
}
```

Amennyiben a **final** előírást az osztály neve után adjuk meg, akkor abból az osztályból nem lehet új osztályt származtatni:

```
class A final {};
class B : public A {}; // hiba!
```

### 3.4 A delete és a default tagfügvények

A C++ fordító egy sor tagfüggvényt hoz létre automatikusan, amennyiben mi nem definiáljuk azokat. C++98-ban egyetlen szabály volt, hogy a fordító által készített paraméter nélküli konstruktor nem jön létre, amennyiben bármilyen konstruktort definiálunk. C++11-ben minden automatikusan létrejövő tagfüggvényről magunk dönthetünk, kérhetjük az elkészítésüket (**default**), illetve elutasíthatjuk azt (**delete**). A kulcsszavak használatát és működését az alábbi osztály szemlélteti:

```
class C {
  public:
    C() = default;                          // C a; ok
    C(const C&) = default;                  // C b(c); ok
    C(C&&) = delete;                        // C c(move(b)); nem ok
    C& operator=(const C&) = default;       // b = a; ok
    C& operator=(C&&) = delete;             // a = move(c); nem ok
    virtual ~C() = default;
};
```

A C++ osztályokhoz osztályszintű operátor-függvényeket (*&, *, ->, ->*, new, delete*) definiál a fordító, melyek használatát meg is tilthatjuk. Az alábbi példában az *S* osztály nem példányosítható a dinamikus tárterületen:

```
struct S {
    void *operator new(size_t) = delete;
    void *operator new[](size_t) = delete;
};

int main() {
    S *p = new S;     // hiba
    S *q = new S[3];  // hiba
    S s;
    S t[7];
}
```

Megjegyezzük, hogy saját (virtuális) tagfüggvényeinket is elláthatjuk a **delete** kulcsszóval. Ezzel a származtatott osztályban megakadályozhatjuk a törölt függvénnyel megegyező prototípusú függvény készítését.

### 3.5 Korlátozások nélküli union típus

A C++11 jelentősen csökkentette azokat a korlátozásokat, amelyek megszabták, hogy mit helyezhetünk el a **union** típusban (például nem lehetett konstruktorral vagy destruktorral rendelkező típusú tagja). Azonban továbbra sem lehetnek az uinók tagjai a virtuális függvények és a referenciák.

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

struct Vektor2D {
    int x, y;
    Vektor2D(int x = 0, int y = 0) : x(x), y(y) {}
};
```

```
union Unio {
    int a;
    double b;
    Vektor2D p;
    Unio() {new(&p) Vektor2D(12,23);}
    ~Unio() {};
};

int main() {
    Unio u;
    cout << u.a << endl;  // 12
    cout << u.b << endl;  // 4.88059e-313
}
```

Az uniók használatának szabaddá tételét elsősorban az alkalmazásfejlesztők támogatják, a hardver közeli programozók inkább veszélyesnek tartják, és javasolják megmaradni a korábbi korlátozásoknál.

## 3.6 Sablonok és a C++11

A fent bemutatott megoldások többsége egyetlen irányba, a sablonok irányába mutat. Egyaránt egyszerűbbé és hatékonyabbá teszik a sablonok használatát és készítését.

### *3.6.1 Szintaxis*

C++98-ban a >> karaktersorozat minden esetben a jobbra való biteltolás műveletét jelentette, így egymásba ágyazott sablonok alkalmazásakor tagoló karaktert kellett a két záró > jel közé tenni. Az új szabványban azonban az egymásba ágyazott sablonok lezárása élvez elsőbbséget, így nem szükséges szóközzel elválasztanunk a nagyobb jeleket.

```
map <int, vector<double>> b;       // C++98, C++11 OK

map <int, vector<double>> a;       // C++11 OK
```

### *3.6.2 Külső sablonok*

Amennyiben egy teljesen specializált sablon előfordult egy fordítási egységben, akkor a C++ 98 fordítónak kötelező volt a fordítási egységen belül példányosítania. Ez szerencsétlen esetben igen megnövelhette a fordítási időt.

A C++11 engedélyezi **extern** kulcsszó használatát a specializált sablonokra, mellyel jelezhetjük a fordítónak, hogy az adott fordítási egységben ne példányosítsa az adott sablont:

```
// Sablon.h: - a sablon-deklaráció
template <typename T>
class Sablon {
  public:
    void fv(T a);
};

// Sablon.cpp - a sablon-definíció
#include <iostream>
#include "Sablon.h"
template <typename T>
void Sablon<T>::fv(T a) {
    std::cout << a << std::endl;
}

// explicit példányosítás
template class Sablon<int>;
```

```
// main.cpp
#include "Sablon.h"

extern template class Sablon<int>;

int main() {
    Sablon<int> teszt;
    teszt.fv(123);
    return 0;
}
```

### *3.6.3 Sablon álnevek (template alias)*

A C++98-ban a **typedef** kulcsszóval csak konkrét típusokhoz rendelhettünk szinonim nevet, azonban nem volt lehetőség arra, hogy a sablonok néhány paraméterét lekötve, új sablonokat hozzunk létre. A C++11 szabvány a **using** kulcsszó felhasználásával megoldást kínál erre a problémára.

```
#include <string>
#include <map>
#include <array>
using namespace std;

template <typename T1, typename T2>
    using dictionary = map<T1, T2>;

template <typename T>
    using strmap = map<string, T >;

template <size_t N>
    using intarray = array<int, N>;

int main() {
    dictionary<int, int> trans;
    trans[12] = 23;
    trans[7] = 29;

    strmap<string> szotar;
    szotar["one"] = "egy";
    szotar["two"] = "kettő";

    intarray<5> itomb = {12,23, 34, 45, 56};
    itomb[3] += 102;
}
```

Osztálysablonok esetén egyaránt készíthetünk álnevet az eredeti sablonhoz, vagy annak tetszőleges specializációjához. Függvénysablonok esetén azonban nem érünk célt ilyen egyszerűen.

```
template <typename T1, typename T2>
double f(T1 a, T2 b) {
   return double((a+b)/2.0);

template <typename T>
double fa(T x) {
    return f<int, T>(100, x);
}
```

Egy általánosabb megoldáshoz további sablonokkal kapcsolatos megoldásokkal is meg kell ismerkednünk:

```
template <typename... Args>
auto g(Args&&... args) -> decltype(f(forward<Args>(args)...)) {
  return f(forward<Args>(args)...);
}
```

### *3.6.4 Sablonok változó hosszúságú argumentumlistával (variadic templates)*

A sablonok használata a C++ nyelv egyik legfontosabb jellemzője, amely a változó hosszúságú argumentumlista kezelésével még hatékonyabb eszközt ad a programozók kezébe. Segítségükkel a szabványos könyvtár több, tetszőleges számú argumentumot - típusbiztos módon - fogadó függvénysablonnal bővült (***thread()***, ***bind()***, ***function()***), továbbá megjelent a tetszőleges méretű és szerkezetű rekordok készítését támogató osztálysablon (***tuple***).

A variáns osztálysablonok készítésének szintaxisa megegyezik a szabványos sablonokéval azzal a különbséggel, hogy az utolsó paraméter előtt három pont (...) szerepel. Például, a ***tuple*** osztálysablon definíciója:

```
template <typename... Tipusok>
class tuple;
```

A variáns argumentum típusokat, vagy skalár értékeket tartalmazhat. A variáns argumentumokat sabloncsomagnak nevezzük (*template pack*), amely nulla vagy több aktuális paraméterből áll.

Az első példában olyan osztálysablont hozunk létre, amely példányai adott (egész) típusú statikus tömböt tartalmaz (*tomb*), melyet variáns argumentumokként megadott elemekkel fel is töltünk. A példában a **sizeof**... operátorral a paramétercsomagban megtalálható paraméterek számát kérdeztük le.

```
template <typename T, T... elemek>
struct StatikusTomb {
   T tomb[sizeof...(elemek)] = {elemek...};
};

int main() {
    StatikusTomb<int, 12,23,34,45> x;
    for (int e : x.tomb)
       cout << e << " ";

    StatikusTomb<char, 'A', 'B', 'C'> y;
    for (char e : y.tomb)
       cout << e << " ";
}
```

Az alábbi variáns *Rekord* osztálysablon tetszőleges számú és típusú adat együttest képes kezelni, bár eléggé körülményes módon.

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

// A sablon deklarációja
template<typename ... Tipusok>
class Rekord;

// A rekurzív feldolgozás végét jelző, üres paraméterlistájú sablon
template<>
class Rekord<> {};

// Legalább egy paramétert tartalmazó specializált sablonváltozat.
// Az N elemű paraméterlistát, egy elem plusz N-1 elemű
// paraméterlistaként dolgozza fel, rekurzív módon.
template<typename Elso, typename ... Maradek>
```

```
class Rekord<Elso, Maradek...> : Rekord<Maradek...> {
    Elso adat;
   public:
    Rekord(Elso const& elso, Maradek const& ... maradek) :
                          Rekord<Maradek...>(maradek...), adat(elso) { }
    Elso const& Fej() const {
        return adat;
    }
    Rekord<Maradek...> const& Tobbi() const {
        return *this;
    }
};

// A Rekord sablon használata
int main() {
    Rekord<int, double, string> r(2011, 8.12, "C++11");
    cout << r.Fej() << endl;                    // 2011
    cout << r.Tobbi().Fej() << endl;            // 8.11
    cout << r.Tobbi().Tobbi().Fej() << endl;    // C++11
}
```

A variáns függvénysablonok típusparamétereit az osztályokhoz hasonló módon használhatjuk, azonban itt tetszőleges számú függvényargumentumból áll elő a típuslista. A feldolgozást rekurzív módon végezzük, ahol a rekurziós lánc végét egy egyparaméteres, túlterhelt függvénysablon biztosítja.

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

template <typename T>
void Kiiras(T t) {
   cout << t;
}

template <typename Elso, typename... Argumentomok>
void Kiiras(Elso e, Argumentomok... argumentumok) {
    cout << e << " ";
    Kiiras(forward<Argumentomok>(argumentumok)...);
}

template <typename T>
T Osszeg(T t) {
    return t ;
}

template <typename T, typename ...P>
T Osszeg(T t, P ...p) {
    if (sizeof...(p))
        t += Osszeg(p...);
  return t;
}

int main() {
    int a=12;
    double b=12.23;
    string c = "C++11";
    Kiiras(a, b, c, '\n');  // 12 12.23 C++11
    cout << Osszeg(12,23) << endl; // 35
}
```

### *3.6.5 Hátravetett visszatérési típus (suffix return types)*

A C++98 nyelven sokszor problémát jelentett a függvénysablonok visszatérési típusának megadása. Gyakran alkalmazott megoldás volt, hogy egy további sablonparamétert használtunk erre a célra:

```
template <typename R, typename S, typename T>
R Atlag1(S a, T b) {
    return (a+b)/2;
}
```

Az *Atlag1()* függvény hívásai:

```
cout << Atlag1<int, int, int>(9, 4) << endl;         // 6
cout << Atlag1<double, double, int>(9.5, 4) << endl; // 6.75
```

A C++11 új függvénymegadási formát vezetett be, amelyben a **decltype** kulcsszó is használható a visszatérési érték meghatározására:

```
template <typename S, typename T>
auto Atlag2(S a, T b) -> decltype ((a+b)/2) {
    return (a+b)/2;
}
```

Az *Atlag2()* függvény hívásai:

```
cout << Atlag2(9, 4) << endl;    // 6
cout << Atlag2(9.5, 4) << endl;  // 6.75
```

Természetesen ez a megoldás osztálysablonok tagfüggvényei esetén is alkalmazható.

# 4. A C++11 könyvtár újdonságai

A szabványos C++ könyvtár több oldalról is bővült. Egyrészt a C11 nyelvből átvett megoldások miatt új, hasznos függvények jelentek meg, amelyek a nevükben hordozzák a kapcsolódó típust. Másrészt pedig az STL rész újabb elemekkel gazdagodott. Valójában a C++98 sablontár is újjászületett, hiszen a sablonok többsége támogatja az áthelyezést (*move*) valamint az inicializáló lista alkalmazását.

Ebben a fejezetben követjük ezt a kettősséget, azonban az új sablonokat csoportosítva mutatjuk be. Felhívjuk a figyelmet arra, hogy semelyik rész sem helyettesíti a hivatalos C++11 STL hivatkozások gyűjteményét. Célunk egy átfogó képet adni a C++11 szabványos könyvtára által nyújtott lehetőségekről:

- új függvények és tagfüggvények a C++ könyvtárban,
  - C-sztring tartalmának számmá alakítása,
  - a string típusú és a numerikus adatok közötti átalakítások,
  - Unicode karakterek és sztringek átalakítása,
  - új matematikai függvények,
  - a bővített complex osztálysablon,
  - fordítás idejű racionális aritmetika,
- véletlen számok előállítása,
- idő- és dátumkezelés,
- reguláris kifejezések könyvtára,
- kibővített típuskezelő könyvtár,
- polimorf függvényobjektumok,
- inicializáló lista saját osztályokban,
- változások a konténerek (tárolók) könyvtárában,
  - kezdőértékadás a konténereknek,
  - a move szemantika és az STL konténerek,
  - a begin, end és swap függvénysablonok használata,
- kibővített algoritmus könyvtár,
- rekordok új megközelítésben (tuple),
- a szabványos C++ nyelv deklarációs állományai.

## 4.1 Új függvények és tagfüggvények a C++ könyvtárban

A szabványos C11 nyelvvel való minél nagyobb kompatibilitás érdekében, a C függvénykönyvtár elemeinek többsége is megjelent a C++11 nyelv szabványos könyvtárában. Emellett azonban néhány C++ függvény is készült, a számok és a szövegek közötti átalakítások támogatására.

### *4.1.1 C-sztring tartalmának számmá alakítása*

A konverziós függvényeket a <*cstdlib*> deklarálja. A függvények nevének utolsó betűi a céltípusra utalnak:

```
long long atoll( const char *str );
long long strtoll( const char *str, char **str_end, int base );
unsigned long long strtoull( const char *str, char **str_end, int base );
float strtof( const char* str, char** str_end );
long double strtold( const char* str, char** str_end );
```

A <*cinttypes*> fejállományban további függvényeket találunk:

```
intmax_t strtoimax( const char* nptr, char** endptr, int base );
uintmax_t strtoumax( const char* nptr, char** endptr, int base );
```

A fenti függvények **char** típusú karaktereket tartalmazó sztringeket használnak. A <*cwchar*> és a <*cinttypes*> fejállományokban megtaláljuk az széles karakterekből álló sztingeket átalakító függvényeket: ***wcstoll()***, ***wcstoull()***, ***wcstof()***, ***wcstold()***, ***wcstoimax()***, ***wcstoumax()***.

### 4.1.2 A string típusú és a numerikus adatok közötti átalakítások

A <*string*> fejállomány egy sor konverziós függvényt biztosít a címben jelzett átalakítások elvégzésére. Sztringet egyszerűen számmá alakíthatunk:

```
int stoi( const string& str, size_t *pos = 0, int base = 10 );
long stol( const string& str, size_t *pos = 0, int base = 10 );
long long stoll( const string& str, size_t *pos = 0, int base = 10 );
unsigned long stoul( const string& str, size_t *pos = 0, int base = 10 );
unsigned long long stoull( const string& str, size_t *pos=0, int base=10 );
float stof( const string& str, size_t *pos = 0 );
double stod( const string& str, size_t *pos = 0 );
long double stold( const string& str, size_t *pos = 0 );
```

A ***to_string()*** függvényének változatai gyors konverziós lehetőséget biztosítanak az egyszerű C++ numerikus típusok ***string*** típusú sztringgé való átalakításához. Széles sztringgé (***wstring***) a ***to_wstring()*** függvényekkel alakíthatjuk a számokat.

### 4.1.3 Unicode karakterek és sztringek átalakítása

A <*cuchar*> fejállományban deklarált függvények a 16- és 32-bites széles karakterek (*c16*, *c32*) valamint a nullavégű sztringben tárolt, többbájtos (*mb*) karakterek közötti konverziók elvégzésére szolgálnak:

```
mbrtoc16(), c16rtomb(), mbrtoc32(), c32rtomb()
```

A <*locale*> fejállományban található ***wstring_convert<>*** osztálysablont együtt használva a <*codecvt*> fájlban definiált ***codecvt_utf8<>*** és ***codecvt_utf16<>*** konverziós leírásokkal (*facet*) UTF-8 és UTF-16 kódolású sztringet 2- vagy 4-bájtos UCS (*unicode*) kódolású karaktersorozattá alakíthatunk. A ***codecvt_utf8_utf16<>*** sablonnal a kétféle UTF kódolás között konvertálhatunk. Az átalakítás lépéseit az alábbi példa szemlélteti:

```
#include <iostream>
#include <string>
#include <locale>
#include <codecvt>
using namespace std;

int main() {
    string utf8 = u8"gk:\U0001D11E pi:\u03C0";
    wstring_convert<codecvt_utf8<char16_t>, char16_t> ucs2conv;
    u16string ucs2 = ucs2conv.from_bytes(utf8);
    cout << ucs2.size() << " darab karakter" << endl;
    for (auto p = begin(ucs2); p!=end(ucs2); ++p)
        cout << hex << showbase << *p << endl;
}
```

A program futásának eredménye:

```
9 darab karakter
0x67
0x6b
0x3a
0xd11e
0x20
0x70
0x69
0x3a
0x3c0
```

A következő példában a konverziókat bináris fájlból történő olvasás során alkalmazzuk:

```
#include <fstream>
#include <iostream>
#include <string>
#include <locale>
#include <codecvt>
using namespace std;

void FajlbaIras() {
    // UTF-16le adatok (little-endian)
    char16_t utf16le[] = u"gk:\U0001D11E, pi:\u03C0"
    ofstream fout("adatok.dat");
    fout.write( reinterpret_cast<char*>(utf16le), sizeof utf16le);
    fout.close();
}

int main() {
    FajlbaIras();
    wifstream fin("adatok.dat", ios::binary);
    fin.imbue(locale(fin.getloc(),
              new codecvt_utf16<wchar_t, 0x10ffff, little_endian>));
    for (wchar_t ch; fin.get(ch); )
            cout << showbase << hex << ch << endl;
    fin.close();
}
```

Mindkét megoldásban elveszett a violin kulcs karaktere, hisz az átalakítás 16-bites UCS2 kódolásra történt.

### 4.1.4 Új matematikai függvények

A C++11 kibővíti a matematikai függvények sorát, és mindhárom valós típushoz biztosítja a megfelelő túlterhelt változatot <*cmath*>:

| Függvény | Leírás |
|---|---|
| ***acosh**(x)* | Árkusz-koszinusz hiperbolikus függvény. |
| ***asinh**(x)* | Árkusz-szinusz hiperbolikus függvény. |
| ***atanh**(x)* | Árkusz-tangens hiperbolikus függvény. |
| ***cbrt**(x)* | Köbgyökfüggvény. |
| ***copysign**(x,y)* | A visszaadott érték *x* abszolút értéke, előjele pedig megegyezik *y* előjelével. |
| ***erf**(x)* | Hibafüggvény. |
| ***erfc**(x)* | Kiegészítő hibafüggvény. |
| ***exp2**()* | 2 hatványa ($y=2^x$) |
| ***expm1**(x)* | $e^x-1$ függvény értéke. |
| ***fdim**(x,y)* | A két argumentum pozitív különbségének számítása (=*fmax(x-y,0)*) |
| ***fma**(x,y,z)* | x*y+z függvény értéke. |
| ***fmax**()* | A két argumentum közöl a nagyobb értékével tér vissza. |
| ***fmin**()* | A két argumentum közöl a kisebb értékével tér vissza. |
| ***fpclassify**(x)* | Minősíti az argumentum értékét (*NaN*, normalizált, végtelen stb.) |
| ***hypot**(x,y)* | Derékszögű háromszög befogóihoz tartozó átfogó kiszámítása. |
| ***ilogb**(x)* | A kettes számrendszerben ábrázolt lebegőpontos szám kitevője egészként. |
| ***isfinite**(x)* | true értékkel tér vissza, ha az argumentum véges. |
| ***isgreater**(x,y)* | x > y? |
| ***isgreaterequal**(x,y)* | x ≥ y? |
| ***isinf**(x)* | true értékkel tér vissza, ha az argumentum végtelen. |
| ***isless**(x,y)* | x < y? |

| | |
|---|---|
| ***islessequal**(x,y)* | x ≤ y? |
| ***islessgreater**(x,y)* | x < y vagy x > y? |
| ***isnan**(x)* | true értékkel tér vissza, ha az argumentum nem szám (*NaN*). |
| ***isnormal**(x)* | true értékkel tér vissza, ha az argumentum normalizált. |
| ***isunordered**(x,y)* | Nem összehasonlítható? |
| ***lgamma**(x)* | A gamma-függvény abszolút értékének természetes alapú logaritmusa. |
| ***llrint**()* | Kerekítés a legközelebbi egészre (**long long** típussal tér vissza). |
| ***llround**()* | Kerekítés a nulla felőli legközelebbi egészre (**long long** típussal tér vissza). |
| ***log1p**(x)* | *1+x* természetes alapú logaritmusa. |
| ***log2**(x)* | Kettes alapú logaritmus. |
| ***logb**()* | A kettes számrendszerben ábrázolt lebegőpontos szám kitevője valósként. |
| ***lrint**()* | Kerekítés a legközelebbi egészre (**long** típussal tér vissza). |
| ***lround**()* | Kerekítés a nulla felőli legközelebbi egészre (**long** típussal tér vissza). |
| ***nan**(str)* | Az *NAN* (nem szám) értékkel tér vissza. |
| ***nanf**(str)* | Az *NAN* (nem szám) értékkel tér vissza. |
| ***nanl**(str)* | Az *NAN* (nem szám) értékkel tér vissza. |
| ***nearbyint**(x)* | Az argumentum kerekítése egész értékre a beállított kerekítési mód alapján. |
| ***nextafter**(x,y)* | Megadja a következő ábrázolható értéket (*y* irányában). |
| ***nexttoward**(x,y)* | Mint a ***nextafter**()*, azonban *y* **long double** típusú |
| ***remainder**(x,y)* | Kiszámolja az osztási maradékot. |
| ***remquo**(x,y,p)* | Mint a ***remainder**()*, azonban a hányadost is visszaadja (*p* mutató). |
| ***rint**(x)* | **int** értékre kerekít, azonban hibát hoz létre, ha az eredmény különbözik az argumentumtól. |
| ***round**(x)* | Kerekítés **int** típusú egészre. |
| ***scalbln**(x,n)* | x * *FLT_RADIX*$^n$ (*n* egész) |
| ***scalbn**(x,n)* | x * *FLT_RADIX*$^n$ (*n* egész) |
| ***signbit**(x)* | true értékkel tér vissza, ha az argumentum negatív. |
| ***tgamma**(x)* | Gamma-függvény. |
| ***trunc**(x)* | Kerekítés a legközelebbi egészre, nulla irányában. |

A <*cstdlib*> és a <*cinttypes*> fejállományok nagyméretű egész számokkal alkalmazható műveleteket deklarálnak:

```
long long abs( long long n );
long long llabs( long long n );
intmax_t abs( intmax_t n );
intmax_t imaxabs( intmax_t n );

lldiv_t div( long long x, long long y );
lldiv_t lldiv( long long x, long long y );
imaxdiv_t div( intmax_t x, intmax_t y );
imaxdiv_t imaxdiv( intmax_t x, intmax_t y );
```

### *4.1.5 A bővített complex osztálysablon*

A hagyományos C++ ***complex**<>* típus további tagfüggvényekkel bővült ( <*complex*> fejállomány):

| | |
|---|---|
| ***proj**(z)* | megadja az argumentum komplex szám vetületét a Riemann gömbön, |
| ***asin**(z), **acos**(z), **atan**(z)* | az argumentum árkusz szögfüggvényeinek számítása, |
| ***asinh**(z), **acosh**(z), **atanh**(z)* | az argumentum árkusz hiperbolikus szögfüggvényeinek meghatározása. |

### *4.1.6 Fordítás idejű racionális aritmetika*

A ***ratio**<>* osztálysablon (<*ratio*>) egy jól definiált fordításidejű interfészt biztosít a törtekkel végzett műveletekhez. A ***ratio**<>* sablon első paramétere a tört számlálója (***num***), míg a második a tört nevezője(***den***) lesz. A ***num*** és a ***den*** a létrejövő osztály statikus adattagjai.

A megoldás érdekessége, hogy a műveleteket is típusok (osztálysablonok) valósítják meg. A négy alapművelet: ***ratio_add<>***, ***ratio_subtract<>***, ***ratio_multiply<>*** *és* ***ratio_divide<>***. Ugyancsak osztálysablonokat használhatunk az összehasonlítások elvégzésére: ***ratio_equal<>***, ***ratio_not_equal<>***, ***ratio_less<>***, ***ratio_less_equal<>***, ***ratio_greater<>*** és ***ratio_greater_equal<>***.

Néhány előre definiált típus lefedi a szokásos egységeknek megfelelő törtszámokat:

| *Típus* | *Definíció* | *Leírás* |
|---|---|---|
| ***yocto*** | `ratio<1,1000000000000000000000000>` | $10^{-24}$ * |
| ***zepto*** | `ratio<1,1000000000000000000000>` | $10^{-21}$ * |
| ***atto*** | `ratio<1,1000000000000000000>` | $10^{-18}$ |
| ***femto*** | `ratio<1,1000000000000000>` | $10^{-15}$ |
| ***pico*** | `ratio<1,1000000000000>` | $10^{-12}$ |
| ***nano*** | `ratio<1,1000000000>` | $10^{-9}$ |
| ***micro*** | `ratio<1,1000000>` | $10^{-6}$ |
| ***milli*** | `ratio<1,1000>` | $10^{-3}$ |
| ***centi*** | `ratio<1,100>` | $10^{-2}$ |
| ***deci*** | `ratio<1,10>` | $10^{-1}$ |
| ***deca*** | `ratio<10,1>` | $10^{+1}$ |
| ***hecto*** | `ratio<100,1>` | $10^{+2}$ |
| ***kilo*** | `ratio<1000,1>` | $10^{+3}$ |
| ***mega*** | `ratio<1000000,1>` | $10^{+6}$ |
| ***giga*** | `ratio<1000000000,1>` | $10^{+9}$ |
| ***tera*** | `ratio<1000000000000,1>` | $10^{+12}$ |
| ***peta*** | `ratio<1000000000000000,1>` | $10^{+15}$ |
| ***exa*** | `ratio<1000000000000000000,1>` | $10^{+18}$ |
| ***zetta*** | `ratio<1000000000000000000000,1>` | $10^{+21}$ * |

A csillaggal jelzett típusok megvalósítása opcionális, csak akkor léteznek, ha az <*cstdint*> ***intmax_t*** típusa lefedi a számláló és a nevező értékeit.

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

int main()
{
    typedef ratio<3,7> tort1;
    typedef ratio<6,21> tort2;

    typedef ratio_add<tort1, tort2> tort3;
    cout << tort3::num << '/' << tort3::den << endl;
    if (ratio_less<tort1, tort2>::value)
        cout << "3/7 < 6/21\n";
    else
        cout << "3/7 >= 6/21\n";

    constexpr int kg=123;
    constexpr int gramm = ratio_divide<ratio<kg,1>, milli>::num;
    cout << "123 kg = " << gramm << " gramm" << endl;
}
```

A program futásának eredménye:

```
5/7
3/7 >= 6/21
123 kg = 123000 gramm
```

## 4.2 Véletlen számok előállítása

A <*random*> fejállomány osztálysablonok sorával segíti a véletlen számok különböző módszerrel történő előállítását. A többféle megoldás eléréséhez külön ki kell választanunk egy véletlen szám motort (*engine*), egy eloszlást (*distribution*), melyeket aztán összekapcsolva előáll a véletlen szám generátor.

A ***véletlen szám motor*** valamilyen algoritmus alapján állítja elő a számokat. A <random> több motort is tartalmaz:

- ***linear_congruential_engine<>*** lineáris kongruens algoritmus (mint a ***rand()*** esetén),
- ***mersenne_twister_engine<>*** Mersenne twister algoritmus,
- ***subtract_with_carry_engine<>*** kivonás átvitellel algoritmus.

A motorokhoz adaptációk is tartoznak, amelyek valamilyen szempont szerint módosítják a megadott alapmotor működését:

***discard_block_engine<>, independent_bits_engine<>, shuffle_order_engine<>***

Mivel az motorok és az adaptációik helyes paraméterezése nem egyszerű feladat, a fejállomány ***előre felparaméterezett motorokat*** is a rendelkezésünkre bocsájt: ***seed_seq<>, minstd_rand, mt19937, ranlux24, knuth_b, default_random_engine*** stb.

Még nagyobb az eloszlások választéka. Számtalan *uniform*, *Bernoulli*, *Poisson*, *normál* vagy *mintavételes* eloszlás közül választhatunk – csak néhány ízelítőül:

***uniform_int_distribution<>, bernoulli_distribution<>, poisson_distribution<>, normal_distribution<>, discrete_distribution<>***

A definíciók alkalmazását az alábbi példaprogram szemlélteti:

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

int main() {
    mt19937 motor;
    motor.seed(time(nullptr)); // a generáló algoritmus inicializálása
    double kozepertek = 2.5, szoras = 1;

    // Különböző eloszlások készítése:
    // alapértelmezett intervallum [0, MAX]
    uniform_int_distribution<uint32_t> uintEloszlas;
    // intervallum [0,10]
    uniform_int_distribution<uint32_t> uintEloszlas10(0,10);
    // N(középérték, szórás)
    normal_distribution<double> normalEloszlas(kozepertek, szoras);

    // véletlen számok előállítása:
    for (int i=0; i<10; i++) {
       cout << uintEloszlas(motor) << '\t'
            << uintEloszlas10(motor) << '\t'
            << normalEloszlas(motor) << endl;
    }
}
```

A program futásának eredménye:

```
4097594794      5       0.71713
3284206494      6       3.34636
1585982901      2       2.85585
2813963803      3       0.601554
2881504895      8       2.89266
3379910637      9       1.56064
431323802       8       1.85685
3496590794      9       3.89834
661944248       3       2.57102
2108095663      8       3.01404
```

Még egy fontos ok, ami miatt inkább a <*random*> könyvtárat használjuk a hagyományos ***rand()*** függvény helyett. Az új eszközökkel a véletlen számok generálása szálbiztos: vagy minden szál saját lokális motort alkalmaz, vagy egy közös motorobjektumot érnek el szinkronizálva.

### 4.3 Idő- és dátumkezelés

A C++11 új időmodellel segíti a programok és a szálak időkezelési feladatait. A modell alapelemei az ***órák*** (*clocks*), amelyek meghatározzák az időszámítás kezdetét (*epoch*, pl. 1970. január 1.) valamint az óraütések gyakoriságát (vagyis az óra pontosságát, pl. 1 ms). Az óraosztályokat a <chrono> fejállomány beépítését követően a ***chrono*** névtérben érjük el:

| | |
|---|---|
| *chrono::**system_clock*** | a rendszer által használt valós idejű óra (beállítható), |
| *chrono::**high_resolution_clock*** | a rendszerben fellelhető legpontosabb óra (beállítható), |
| *chrono::**steady_clock*** | pontos időméréshez jól használható monoton óra, amely nem állítható be. |

A másik fontos eleme az időkezelésnek az ***időtartam*** (*duration*), amely adott időegységek számát jelöli. Az időtartamot a *chrono::**duration**<>* osztálysablon képviseli, melyet a ***ratio**<>* sablonnal együtt használva saját időtartamegységeket is készíthetünk, az előre definiáltakon (*chrono::**nanoseconds**, chrono::**microseconds**, chrono::**milliseconds**, chrono::**seconds**, chrono::**minutes**, chrono::**hours***) túlmenően. (A másodperc az alapértelmezett időtartam egység.)

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

int main() {
    typedef duration<int, ratio<1,2>> felmasodperc; // félmásodperc
    felmasodperc d1(10);
    seconds d2 = duration_cast<seconds>(d1);
    cout << "a megadott idő: " << d2.count() << " mp" << endl; // 5 mp
}
```

A ***duration**<>* sablon első paramétere az óraütemek számát tároló típus, míg a második a másodpercekhez képest definiálja az időtartamegységet. Az időegységek számát az időtartam osztály ***count()*** tagfüggvényével kérdezhetjük le, míg a különböző időtartamok közötti átváltáshoz a ***duration_cast**<>()* függvénysablont kell használnunk.

Az időtartam osztály további statikus tagjaival lekérdezhetjük az időtartamhoz tartozó nullás, minimális és maximális értéket:

```
cout << seconds::zero().count()<< endl; // 0
cout << seconds::min().count() << endl; // -9223372036854775808
cout << seconds::max().count() << endl; // 9223372036854775807
```

Az időtartam objektumokkal használhatjuk az értékadást (=), az öt alapműveletet (+, -, *, /, %) (összetett értékadásban is) valamint az összehasonlító műveleteket (==, !=, <, <=, >, >=). Felhívjuk a figyelmet arra, hogy a szorzás, osztás és maradékképzés esetén csak az egyik operandus lehet időtartam, a másik operandusnak számnak kell lennie!

```
minutes m1(3), m2(6), m3(-1);
m3 += 5 * m1 - m2 / 2 + m1 % 2;
cout << m3.count() << endl;  // 12
cout << (m3 <= m1) << endl;  // 0
```

Az időmodell harmadik pillére az ***időpont*** (*timepoint*), amelyet a *chrono::**time_point**<>* osztálysablon valósít meg. Az időpont valójában az óra időszámításának kezdetét és az eltelt időtartamot kombinálja. A Unix/Linux rendszerekben a rendszeróra által visszaadott beépített időpontok:

```
system_clock::time_point tp;                 Thu Jan 1 01:00:00 1970
tp = system_clock::time_point::min();        Sat Mar 5 17:27:38 1904
tp = system_clock::time_point::max();        Mon Oct 29 06:32:22 2035
```

Időpont objektumot többféleképpen is létrehozhatunk. Használhatjuk az óra osztályba épített ***time_point*** típust, vagy magát a ***time_point**<>* sablont. Első esetben a tárolt időtartam másodperc lesz, míg második esetben ez szabadon megadható, az alábbi példában látható módon:

```
#include <chrono>
#include <iostream>
#include <ctime>
#include <string>
using namespace std;
using namespace chrono;

string Idopont2Str (const system_clock::time_point& tp) {
     // az időpont átalakítása time_t típusú idővé:
     time_t t = system_clock::to_time_t(tp);
     string ts = ctime(&t);
     ts.resize(ts.size()-1);  // az újsor karakter törlése
     return ts;
}

int main() {
    // a most lekérdezése különböző módon:
    system_clock::time_point most1 = system_clock::now();
    time_point<system_clock> most2 = most1;
    time_point<system_clock, milliseconds> most_ms;
    most_ms = time_point_cast<milliseconds>(most2);
    cout << Idopont2Str(most1) << endl;

    // 7 nap 12 perc 23 másodperc múlva:
    seconds kesobb = duration_cast<seconds>(7*hours(24) +
                                            minutes(12) + seconds(23));
    most1 += kesobb;
    most2 = most2 + kesobb;
    most_ms += kesobb;
    cout << Idopont2Str(most1) << endl;
}
```

A program futásának eredménye:

```
Sat Sep 21 17:25:57 2013
Sat Sep 28 17:38:20 2013
```

Az időpont objektumokhoz tartozó +, += és -, -= műveletekkel az időpontot a megadott időtartammal módosíthatjuk, és az időpontokat össze is hasonlíthatjuk (==, !=, <, <=, >, >=).

Az új időmodell nem rendelkezik humán interfésszel, nem ismeri a naptárunkat. Az órák ***to_time_t()*** tagfüggvénye – a hagyományos C időkezeléshez – ***time_t*** típusúvá alakítja az időpontot. (Ezt alkalmaztuk az előző példa *Idopont2Str()* függvényében a megjelenítéshez). A ***from_time_t()*** tagfüggvénynyel pedig konkrét időpontot is megadhatunk. Az következő példaprogram megmondja, hogy hányat kell még Karácsonyig aludnunk:

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

int main() {
    tm xmas1 = {0};
    xmas1.tm_year = 2013-1900;
    xmas1.tm_mon = 12-1;
    xmas1.tm_mday = 25;
    time_t xmas2 = mktime(&xmas1);

    system_clock::time_point xmas3 = system_clock::from_time_t(xmas2);
    system_clock::time_point most = system_clock::now();
    typedef duration<int, ratio<24*3600,1>> nap;
    nap napok = duration_cast<nap>(xmas3-most);
    cout << "meg aludni kell: " << napok.count() << " napot.\n"; // 94
}
```

Végezetül nézzünk egy egyszerű futásidőt mérő programot, amelyben megmérjük, hogy mennyi idő szükséges 10 millió elemet tartalmazó vektor rendezéséhez!

```
#include <iostream>
#include <chrono>
#include <vector>
#include <algorithm>
using namespace std;
using namespace chrono;

int main() {
    const int meret = 10000000;
    vector<int> v(meret);

    srand(unsigned(time(nullptr)));
    for_each (begin(v), end(v), [](int& e) { e = rand();});
    random_shuffle(begin(v), end(v));

    auto start = high_resolution_clock::now();
    sort(begin(v), end(v));
    auto stop = high_resolution_clock::now();

    auto eltelt = duration_cast<milliseconds>(stop - start);
    cout << "mért idő: " << eltelt.count() << " ms" << endl; // 748 ms
}
```

### 4.4 Reguláris kifejezések könyvtára

A C++11 az STL filozófiájának megfelelően osztálysablonok és függvénysablonok segítségével valósítja meg a reguláris kifejezését kezelését. Az alaposztályok (***basic_regex***, ***sub_match***, ***match_results***) mellett algoritmusok (***regex_match***, ***regex_search***, ***regex_replace***), iterátorok (***regex_iterator***, ***regex_token_iterator***) valamint előre definiált konstansok (***std::regex_constants*** névtér) segítik a munkánkat.

A ***basic_regex*** sablon **char** és a **wchar_t** tusokkal specializált változatait (***regex***, ***wregex***) használjuk a szabályos kifejezések tárolására:

```
regex szo("[[:alpha:]]+");
regex szokozok("[[:space:]]*(.+)", regex::extended);
```

A reguláris kifejezések megadásánál háromféle szabálygyűjtemény közül is választhatunk. Az alapértelmezett az *ECMAScript* (*ECMA-262*, *JavaScript*, *Perl 5*) szintaxis, azonban a második argumentumban a *POSIX* alap (*basic*) és bővített (*extended*) nyelvtanának elkalmazását is kérhetjük.

A következő példaprogram a megadott sztringről megállapítja, hogy egész számot tartalmaz-e vagy sem:

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

int main() {
    string input;
    regex egesz("(\\+|-)?[[:digit:]]+");
    while (true) // A q billentyűlenyomásáig inputot vár
    {
        cout<<"Kérek egy egész számot!"<<endl;
        cin>>input;
        if ( input == "q" )
            break;
        if ( regex_match(input, egesz) )
            cout << "egész" << endl;
        else
            cout << "hibás input" << endl;
    }
}
```

Amennyiben a C++ nyelv valós számformátumának megfelelő adatokat szeretnénk ellenőrizni, az alábbi reguláris kifejezést kell használnunk:

```
regex valos("((\\+|-)?[[:digit:]]+)(\\.(([[:digit:]]+)?))?((e|E)"
            "((\\+|-)?)[[:digit:]]+)?");
```

Bizonyos műveletek esetén a ***match_results*** osztálysablon karakteres specializált változatai (***cmatch***, ***wcmatch***, ***smatch***, ***wsmatch***) tárolják az egyeztetés eredményét.

Az alábbi példában reguláris kifejezések segítségével egy sztringből eltávolítjuk a bevezető szóközöket, illetve szétválogatjuk a benne tárolt szavakat és számokat:

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

int main() {
    regex egesz("(\\+|-)?[[:digit:]]+");
    regex szo("[[:alpha:]]+");
    regex szokozok("[[:space:]]*(.+)");

    // A bevezető szóközök eltávolítása
    string input ="         Ivan9 Anna23 Adrienn26 Aliz33";
    smatch eredmeny;
    regex_search(input, eredmeny, szokozok);
    cout << eredmeny[1].str() << endl;

    // Szétválasztja az inputból a neveket és a számokat
    const string formatum ="";  // helyettesítő formátumsztring
```

```
    string szavak, szamok;
    szamok = regex_replace(eredmeny[1].str(), egesz, formatum,
             regex_constants::format_default);
    szavak = regex_replace(eredmeny[1].str(), szo, formatum,
             regex_constants::format_default);
    cout << szavak << endl;
    cout << szamok << endl;
}
```

A program futásának eredménye:

```
Ivan9 Anna23 Adrienn26 Aliz33
9 23 26 33
Ivan Anna Adrienn Aliz
```

### 4.5 Kibővített típuskezelő könyvtár

A C++ típuskezelő könyvtár újabb alaptípusokkal (*<cstddef>*), illetve a típusok fordításidejű (*<type_*traits>) és futásidejű (*<typeindex>*) kezelését segítő sablonokkal bővült.

A típusjellemzők fordítás közbeni ellenőrzését segítik az ***is_void<>***, ***is_integral<>***, ***is_class<>***, ***is_copy_constructible<>*** stb. sablonok (*<type_traits>*)

```
#include <type_traits>
#include <iostream>
#include <algorithm>
#include <limits>
using namespace std;

template <typename T>
    T Lkt(const T& a, const T& b) {
       static_assert(is_integral<T>::value,
                     "T-nek egész típusúnak kell lennie!");
       for (int lkt = max(a, b); lkt<=a*b; lkt++)
          if (lkt%a==0 && lkt%b==0) return lkt;
    }

int main() {
    cout << numeric_limits<int>::lowest() << endl;
    cout << Lkt(12,8) << endl;            //  24
    cout << Lkt(12.3,7.8) << endl;        //  fordítási hiba
}
```

Ugyancsak itt találunk egy sor típuskészítő és -átalakító sablont: ***integral_constant<>***, ***add_const<>***, ***result_of<>*** stb.

A futás közbeni típusazonosítást segíti a típusokhoz rendezett indexsort rendelő ***type_index*** osztály (*<typeindex>*):

```
#include <iostream>
#include <typeinfo>
#include <typeindex>
#include <unordered_map>
#include <string>
using namespace std;

class A {
  public:
     virtual ~A() {}
};
```

```
int main() {
    unordered_map<type_index, string> tipusNevek;
    tipusNevek[type_index(typeid(int))] = "egész";
    tipusNevek[type_index(typeid(double))] = "valós";
    tipusNevek[type_index(typeid(A))] = "A osztály";

    int i;
    double d;
    A a;

    cout << "i típusa " << tipusNevek[type_index(typeid(i))] << endl;
    cout << "d típusa " << tipusNevek[type_index(typeid(d))] << endl;
    cout << "a típusa " << tipusNevek[type_index(typeid(a))] << endl;
}
```

## 4.6 Polimorf függvényobjektumok

A függvényobjektumok olyan objektumok, amelyekben definiált a függvényhívás operátora. A C++ több könyvtári függvényobjektummal rendelkezik, azonban az ilyen objektumok létrehozását és kezelését is segíti. A C++11 sok új elemmel (***function<>***, ***bind<>*** stb.) bővíti a *<functional>* fejállomány definícióit, azonban több elemet (***unary_function<>***, ***binder1st<>*** stb.) elavultnak nyilvánít.

A ***hash<>*** sablon specializált változatai az alaptípusokhoz és a karaktersorozat könyvtári típusokhoz *hash* (hasító) kódot számítanak.

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

int main() {
    hash<string> hashFv;
    string str = "C++ programozási nyelv";
    size_t hashKod = hashFv(str);
    cout << hashKod << '\n';        // 4265739948
}
```

A *template< class R, class... Args > class function<R(Args...)>;* deklarálású ***funtion<>*** sablon egy általános célú függvényburkoló (*function wrapper*), amellyel normál függvényekre, tagfüggvényekre, lambda függvényekre valamint funktorokra egyaránt hivatkozhatunk. (Funktor – *functor* olyan osztály, amely definiálja a függvényhívás operátorát.)

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

int Kerekit(double x) {
    return int(x+0.5);
}

class A {
  public:
    long Levag(double x) { return long(x);}
};

class Funktor {
   public:
     long operator()(double x) const {
        return (long)x;
     }
};
```

```
int main() {
    function<int(double)> kerekitHivas = &Kerekit;
    cout << kerekitHivas(12.7) << endl;         // 13

    A a;
    function<long(A&, double)> levagHivas = &A::Levag;
    cout << levagHivas(a, 2.9) << endl;         // 2
    function<long(A*, double)> levagHivasPtr = &A::Levag;
    cout << levagHivasPtr(&a, 2.9) << endl;     // 2

    function<long(long a)> Kob = [](long x){ return x*x*x; };
    cout << Kob(10)<< endl;                     // 1728
    auto Negyzet = [](long x){ return x*x; };
    cout << Negyzet(10) << endl;                // 144
    Kob = Negyzet;

    Funktor fu;
    function <long(double)> fuHivas = fu;
    cout << fuHivas(12.34) << endl;             // 12
}
```

A ***mem_fn***<> sablon szintén függvényburkoló objektumot készít, azonban egy megadott tagfüggvény mutató alapján.

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

class A {
  public:
    long Levag(double x) { return long(x);}
};

int main() {
    A a;
    auto levagFv = mem_fn(&A::Levag);
    cout << levagFv(a, 12.7) << endl;          // 12
}
```

A *function* objektum rendelkezik a **bool** típusra konvertáló operátorral, mely művelet eredményéből megtudhatjuk, hogy a függvényburkoló kapott-e értéket:

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

int main () {
        function<void()> f;
        cout << (bool) f << endl;        // 0
        f = [](){};                      // üres lambda függvény!
        cout << (bool) f << endl;        // 1
}
```

A ***bind<>*** függvény segítségével argumentumokat köthetünk a függvényobjektumokhoz. A ***ref<>*** és ***cref<>*** sablonok ***reference_wrapper*** típusú argumentumokat készthetünk.

Az alábbi példában a ***bind<>*** és a helyfoglaló objektumok (***_1***, ***_2***, …) segítségével az argumentumokat különböző sorrendben továbbítjuk a hívott függvénynek:

```
#include <functional>
#include <string>
#include <iostream>
```

```
using namespace std;
using namespace std::placeholders;

void Kiir(string a, string b, string c) {
   cout << a << " " << b << " " << c << endl;
}

int main() {
    auto fa = bind(Kiir, _1, _2, _3);
    auto fb = bind(Kiir, _3, _1, _2);
    auto fc = bind(Kiir, "delta", _1, _1);
    fa("alfa", "beta", "gamma");
    fb("alfa", "beta", "gamma");
    fc("alfa", "beta");
}
```

A program futásának eredménye:

```
alfa beta gamma
gamma alfa beta
delta alfa alfa
```

A ***bind<>*** függvénnyel együtt használjuk a ***ref<>*** és a ***cref<>*** sablonokat, amelyekkel a hívni kívánt függvényhez referencia, illetve konstans referencia típusú argumentumokat köthetünk:

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

void Fv(int& a, int& b, const int& c) {
    cout << "Függvényen belül: " << a << ' ' << b << ' ' << c << endl;
    ++a, ++b;
}

int main() {
    int a = 1, b = 2, c = 3;
    function<void()> fvHivas = bind(Fv, a, ref(b), cref(c));
    a = 10, b = 20, c = 30;
    cout << "Fv hívása előtt: " << a << ' ' << b << ' ' << c << endl;
    fvHivas();
    cout << "Fv hívása után:  " << a << ' ' << b << ' ' << c << endl;
}
```

A program futásának eredménye:

```
Fv hívása előtt: 10 20 30
Függvényen belül: 1 20 30
Fv hívása után:  10 21 30
```

## 4.7 Inicializáló lista saját osztályokban

Az inicializáló lista feldolgozása során a fordító a sablonhívásban megadott típusú adatokból egy tömböt állít elő, amelyhez kapcsolt objektum rendelkezik a ***size()***, ***begin()*** és ***end()*** tagfüggvényekkel.

Az ***initializer_list<>*** típusú objektumokat önállóan is felhasználhatjuk a programokban:

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

void Kiir(const initializer_list<int>& il) {
    for (int x : il) cout << x << ' '; cout << endl;
    cout << "Elemek szama: " << il.size() << endl;
}
```

```
int main() {
    initializer_list<int> uresLista;
    Kiir(uresLista);
    Kiir({});

    initializer_list<int> egeszek1 {12, 23, 34, 45};
    Kiir(egeszek1);
    auto egeszek2 = {120, 230, 340, 450};
    Kiir(egeszek2);
    Kiir({1, 2, 3, 4, 5, 6});
}
```

A program futásának eredménye:

```
Elemek szama: 0

Elemek szama: 0
12 23 34 45
Elemek szama: 4
120 230 340 450
Elemek szama: 4
1 2 3 4 5 6
Elemek szama: 6
```

Az osztályaikat egyszerűen elláthatjuk inicializáló listával. Mindössze egy olyan konstruktort kell készítenünk, amely egy ***initializer_list<>*** típusú paraméterrel rendelkezik:

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

template<typename T>
class Vektor {
   T* adatok;
   int meret;
  public:
   Vektor(initializer_list<T> lista) {
       meret = (int)lista.size();
       adatok = new T[meret];
       uninitialized_copy(begin(lista), end(lista), adatok);
   }
   ~Vektor() { delete []adatok; }
    void Kiir() {
       for (int i=0; i<meret; i++)
          cout << adatok[i] << " ";
       cout << endl;
   }
};
int main() {
    Vektor<int> v {1,2,3,4,5};
    v.Kiir();
}
```

Felhívjuk a figyelmet arra, hogy az egyszerű konstruktorhívásokban is alkalmazhatunk inicializáló listát, amelynek feldolgozása, és a konstruktorhívás elvégzése a fordító feladata. Az ilyen listák akár különböző típusú elemeket is tartalmazhatnak:

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

```
struct Adatok {
    int ma;
    double mb;
    string ms;
    Adatok(int a=0, double b=0.0, string s="") {
        ma=a;  mb=b; ms = s;
    }
};
int main() {
    Adatok a1(12), a2 = {12}, a3 {12};
    Adatok b1(12, 23.4), b2 = {12, 23.4}, b3 {12, 23.4};
    Adatok c1(12, 23.4, "C++");
    Adatok c2 = {12, 23.4, "C++"};
    Adatok c3 {12, 23.4, "C++"};
}
```

## 4.8 Változások a konténerek (tárolók) könyvtárában

A C++11 szabványos sablontára több új tárolóval egészíti ki a már jól bevált konténereket. Az alábbi táblázatban összefoglaltuk a hagyományos és az új tároló osztálysablonokat:

| | *Szekvenciális tárolók* | *Asszociatív tárolók* |
|---|---|---|
| C++98 | ***vector*** – dinamikus, folytonos tárolású tömb,<br>***deque*** – kétvégű sor,<br>***list*** – kétszeresen láncolt lista. | ***set*** – rendezett halmaz egyedi kulcsokkal,<br>***map*** – rendezett szótár egyedi kulcsokkal,<br>***multiset*** – rendezett halmaz,<br>***multimap*** – rendezett szótár. |
| C++11 | ***array*** – folytonos tárolású statikus tömb,<br>***forward_list*** – egyszeres láncolású lista. | ***unordered_set*** – hasított halmaz egyedi kulcsokkal,<br>***unordered_map*** – hasított szótár egyedi kulcsokkal,<br>***unordered_multiset*** – hasított halmaz,<br>***unordered_multimap*** – hasított szótár. |

Az ***array<>*** típus a C-stílusú egydimenziós tömbök kezelésének egyszerűségét ötvözi a konténerek hasznos tulajdonságaival:

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

int main() {
   array<int,5> a = {1,2,3};
   array<int,5> b;
   b = a;                              // másolás
   for (int i=0; i<b.size(); i++)      // méret lekérdezése
       cout << b[i] << ' ';            // 1 2 3 0 0
}
```

Többdimenziós tömbök esetén szintén a C-stílusú megoldáshoz hasonló módon járhatunk el, például 2x3-as mátrix esetén:

```
array<array<int,3>, 2> m;
for (int i=0; i<m.size(); i++)       // sorok
   for (int j=0; j<m[i].size(); j++) // oszlopok
      m[i][j]=i+j;

0       1       2
1       2       3
```

A ***forward_list<>*** gyors, egyirányú listakezelést valósít meg. A listába való beszúráshoz, illetve a törléshez az elemek számától független idő szükséges. Az alábbi példaprogramban röviden összefoglaltuk a listakezelés alapvető műveleteit:

```
#include <forward_list>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>
using namespace std;

void ListaKiir(const string& s, const forward_list<int>& lista) {
    cout << s << '\t';
    copy(lista.cbegin(), lista.cend(),
                         ostream_iterator<int>(cout," "));
    cout << endl;
}


int main() {
    // A lista létrehozása és feltöltése
    forward_list<int> lista = { 1, 2, 3, 4 };
    ListaKiir ("A kiinduló lista:", lista);

    // Elem beszúrása a lista elejére
    lista.insert_after(lista.before_begin(), 10);
    lista.push_front(11);
    lista.insert_after(lista.before_begin(), {21,22,23,3,11,12} );
    ListaKiir ("8 új elem az elejére:", lista);

    // Az első két elem törlése
    lista.erase_after(lista.begin());  // 2. elem
    lista.pop_front();  // 1. elem
    ListaKiir ("Az első 2 elem törölve:", lista);

    // A lista rendezése és az ismétlődő elemek törlése
    lista.sort();
    lista.unique();
    ListaKiir ("Rendezett, kiegyelt:", lista);
}
```

A program futásának eredménye:

```
A kiinduló lista:       1 2 3 4
8 új elem az elejére:   21 22 23 3 11 12 11 10 1 2 3 4
Az első 2 elem törölve: 23 3 11 12 11 10 1 2 3 4
Rendezett, kiegyelt:    1 2 3 4 10 11 12 23
```

A szabványos C++11 nyelv a korábbi asszociatív konténerekhez elérhetővé tette az elemeket rendezettség nélkül tároló párjukat: ***unordered_set<>***, ***unordered_multiset<>***, ***unordered_map<>*** és ***unordered_multimap<>***.

Míg a C++98 halmaz és szótár objektumai az elemeket bináris rendezőfában tárolják, addig az új megoldások hasító (*hash*) táblát alkalmaznak. A két csoport osztályainak programozói felülete majdnem teljes egészében megegyezik, mint ahogy az alábbi példában is látható. Természetesen az ***unordered_xxx<>*** konténerek nem rendelkeznek a rendezettségre épülő tagfüggvényekkel (***rbegin()***, ***lower_bound()***, ***value_comp()*** stb.).

```
#include <iostream>
#include <unordered_map>
#include <map>
#include <string>
using namespace std;
```

```
int main() {
    unordered_map<string,int> win;
    // Elemek bevitele
    win["NT"] = 0;
    win["XP"] = 1;
    win["Vista"] = 6;
    win["7"] = 9;
    win["8"] = 12;

    // A tárolt évszámok módosítása
    for (auto p = win.begin(); p != win.end(); ++p)
        p->second += 2000;

    // Az "NT" átnevezése "2000"-re elemcserével
    win["2000"] = win["NT"];
    win.erase("NT");

    // Az elemek érték szerint rendezve a multimap-ben
    multimap<int, string> winmm;
    for (auto p = win.begin(); p != win.end(); ++p)
        winmm.insert(pair<int, string>(p->second, p->first));

    // Az elemek kiírása
    for (auto p = winmm.begin(); p != winmm.end(); ++p)
        cout << "Windows " << p->second << " " << p->first << endl;
}
```

A program futásának eredménye:

```
Windows 2000 2000
Windows XP 2001
Windows Vista 2006
Windows 7 2009
Windows 8 2012
```

Az ***unordered_xxx<>*** osztálysablonok argumentumaként saját készítésű hasító (*hash*) kódot számító és azonosságot vizsgáló konstans funkcionált is megadhatunk. Ezzel a módszerrel saját típusú adatokat is tárolhatunk a rendezés nélküli asszociatív konténerekben:

```
#include <functional>
#include <iostream>
#include <unordered_set>
#include <string>
using namespace std;

struct Komplex {
  double re, im;
  Komplex(double a=0, double b=0) : re(a), im(b) {}
};

struct KomplexEqual {
    bool operator()(const Komplex& k1, const Komplex& k2) const {
        return (k1.re == k2.re) && (k1.im == k2.im);
    }
};

struct KomplexHash {
    size_t operator()(const Komplex& k) const {
        return hash<double>()(k.re) | hash<double>()(k.im);
    }
};
```

```
int main() {
    unordered_set<Komplex, KomplexHash, KomplexEqual> komplexSet;
    komplexSet.insert(move(Komplex(1,2)));
    komplexSet.insert(move(Komplex(2,3)));
    komplexSet.insert(move(Komplex(3,1)));

    auto p = komplexSet.find(Komplex(2,3));
    if (p == komplexSet.end())
        cout << "nincs meg" << endl;
    else
        cout<< p->re << "+i" << p->im << endl;
}
```

### 4.8.1 Kezdőértékadás a konténereknek

A C++98 szabvány konténer objektumait csak a konstruktor adta lehetőségekkel, illetve külön tagfüggvényekkel inicializálhatjuk.

```
vector<int> iv(10,2);

vector<string> sv;
sv.push_back("alfa");
sv.push_back("béta");
sv.push_back("gamma");

map<string, int> m1;
m1["egy"] = 1;
m1["kettő"] = 2;
m1.insert(pair<string, int>("három", 3));
```

C++11-ben a kezdőértékadás az inicializáló lista segítségével egyszerűen elvégezhető:

```
vector<int> iv2 = {2, 7, 12, 23, 27, 30};

vector<string> sv2 {"alfa", "béta", "gamma"};

map<string, int> m2 { { "egy", 1}, {"kettő", 2}, {"három", 3} };
```

### 4.8.2 A move szemantika és az STL konténerek

Az áthelyezés (*move*) szemantika bevezetése minden konténer osztálysablon kibővítését jelentette, hiszen el kellett látni őket az áthelyező konstruktorral és operátorral. A konténerekbe új adatot helyező tagfüggvényeknek megjelent a jobbérték referenciával működő párjuk, például, a ***vector***<> osztálysablon esetén:

```
void push_back( const T& value );        // másol
void push_back( T&& value );             // áthelyez
```

A <*utility*> fejállományban definiált ***move*** függvénysablon a balérték hivatkozásokat jobbérték hivatkozásokká alakítja, így a másoló konstruktor és a másolással elvégzett értékadás helyett az áthelyező konstruktor és az áthelyező értékadó operátor hívódik meg. A megoldás célja a sok ideiglenes objektumpéldány számának csökkentése, így a programfutás hatékonyságának növelése. A ***move*** hívás eredménye valójában egy *xvalue* referencia, ami azt jelenti, hogy a fordító az áthelyező tagfüggvények (*rvalue*) hiányában a megfelelő másoló tagfüggvényeket (*lvalue*) hívja.

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

typedef complex<double> komplex;
```

```
const vector<double> valosErtekek (const vector<komplex>& v) {
    vector<double> valos( v.size() );
    for (auto c : v)
        valos.push_back( c.real() );
    return move(valos);
}

int main() {
    vector<komplex> cv;
    for ( int i = 10; i < 1000; i++ )
        cv.push_back( move(komplex(i*1.2, i/2.3)));
    vector<double> v = valosErtekek ( cv );
}
```

Az ideiglenes objektumok elkerülésének másik módja a konténer osztályok bővítése az ***emplace()*** (az ***insert()*** helyett) és az *emplace_xxx()* tagfüggvényekkel, amelyek „helyben” hozzák létre a tárolandó objektumot.

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

int main() {
    vector<complex<double>> cv { {12,7} };

    cv.push_back(complex<double>(12,23));
    cv.emplace_back(7,29);

    cv.insert(cv.begin(), complex<double> (10,2));
    cv.emplace(cv.begin(), 11,30);

    for (const auto& c : cv)
        cout << c.real() << " + i" << c.imag() << endl;
}
```

A program futásának eredménye:

```
11 + i30
10 + i2
12 + i7
12 + i23
7 + i29
```

A másolás és az áthelyezés az iterátorok szintjén is megkülönböztethető az <*iterator*> fejállományban definiált ***move_iterator<>*** osztálysablon és a ***make_move_iterator<>()*** függvénysablon segítségével. Ugyancsak itt találjuk meg az iterátorokat adott számú elemmel előre (***next***), illetve visszaléptető (***prev***) függvénysablonokat.

```
#include <iostream>
#include <list>
#include <vector>
#include <string>
#include <iterator>
#include <algorithm>
using namespace std;

void Kiir(const string &s, const vector<string>& vs){
    cout << s;
    for (const auto e : vs)
        cout << "\"" << e << "\" ";
    cout << endl;
}
```

```
int main() {
    list<string> lsgr{"C++", "Java", "C#"};

    vector<string> v1gr(lsgr.begin(), lsgr.end());       // másolás

    vector<string> v2gr(make_move_iterator(lsgr.begin()),
                        make_move_iterator(lsgr.end())); // áthelyezés

    cout << "lsqr tartalma: ";
    for (const auto s : lsgr)
            cout << "\"" << s << "\" "; cout << endl;

    Kiir("v1gr tartalma: ", v1gr);
    Kiir("v2gr tartalma: ", v2gr);

    typedef vector<string>::iterator tmutato;
    string s = accumulate(move_iterator<tmutato>(v1gr.begin()),
                          move_iterator<tmutato>(v1gr.end()), string());
    cout << s << endl;
    Kiir("v1gr tartalma: ", v1gr);
}
```

A program futásának eredménye:

```
lsqr tartalma: "" "" ""
v1gr tartalma: "C++" "Java" "C#"
v2gr tartalma: "C++" "Java" "C#"
C++JavaC#
v1gr tartalma: "" "" ""
```

### *4.8.3 A begin, end és swap függvénysablonok használata*

A C++11 szabvány a tartományalapú **for** ciklus működéséhez definiálja a ***begin*** és az ***end*** függvénysablonokat, amelyek konténerek esetén a ***begin()*** illetve az ***end()*** tagfüggvényeket hívják. A sablonos megoldás előnye, hogy az új osztályokat egyszerűen elláthatjuk a ***begin*** és az ***end*** használatának képességével, csupán a sablonokat kell az új típusra specializálni. A C++11-ben megjelent egy harmadik függvénysablon, a ***swap***. A vele elvégezhető felcserélés művelet minden konténer tagfüggvényeként is létezik. (Ezekkel a specializált sablonokkal a ***valarray<>*** típusú objektumok, valamint a hagyományos tömbök szintén feldolgozhatók a tartományalapú **for** ciklussal.)

A ***begin,*** az ***end*** és a ***swap*** függvénysablonok túlterhelt változatait több fejállomány is tartalmazza (<*iterator*>, <*vector*>, <*utility*> stb.)

A ***begin*** és az ***end*** függvénysablonokkal, valamint a tartományalapú ciklus alkalmazásával a konténerekben tárolt adatok feldolgozása többféleképpen elvégezhető. Példaként tekintsük az egyirányú listát, amely nem rendelkezik az indexelés operátorával, így az elemek eléréséhez általában iterátorokat használunk:

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

int main() {
    forward_list<int> fl {12, 7, 23, 10, 2};
}
```

Az elemek megjelenítése a C++98 által is támogatott egyetlen megoldást alkalmazva:

```
forward_list<int>::iterator flp;
for (flp = fl.begin(); flp != fl.end(); flp++)
    cout << *flp << endl;
```

Az automatikus típusadás és a ***begin*** és ***end*** sablonok segítségével a megoldás sokkal barátságosabbá válik:

```
for (auto flp = fl.begin(); flp != fl.end(); flp++)
    cout << *flp << endl;

for (auto flp = begin(fl); flp != end(fl); flp++)
    cout << *flp << endl;
```

A tartományalapú ciklus alkalmazásával az iterátorokról is megfeledkezhetünk:

```
for (const auto& listaElem : fl)
    cout << listaElem << endl;
```

### 4.9 Kibővített algoritmus könyvtár

A *move* szemantika bevezetése az algoritmusoknál szintén változásokat hozott. A már korábban meglévő függvénysablonok művelet paraméterének függvénymutatót és áthelyező vagy másoló konstruktorral rendelkező (*move constructible*) függvényobjektumot (lambda vagy funktor) is átadhatunk. Az alábbi példában a ***for_each()*** algoritmussal szemléltetjük a lehetőségeket:

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

void Add5(int & n) { n += 5; }

struct Osszeg {
    Osszeg() { osszeg = 0; }
    void operator()(int n) { osszeg += n; }
    int osszeg;
};

int main() {
    vector<int> v{3, -2, 26, 8, 19, 25, 6, 7};

    for_each( v.begin(), v.end(), Add5 );

    for_each( begin(v), end(v), [](int &n){ n--; } );

    Osszeg sum = for_each( begin(v), end(v), Osszeg() );

    cout << "vektor: ";
    for (auto n : v) cout << n << ' ';
    cout << endl;
    cout << "összeg: " << sum.osszeg << endl;
}
```

A program futásának eredménye:

```
vektor: 7 2 30 12 23 29 10 11
összeg: 124
```

Az alábbi táblázatban összegyűjtöttük az új STL algoritmusokat <*algorithm*>, amelyek jól kiegészítik a már meglévőket.

| | |
|---|---|
| **bool** ***all_of***(InIter kezdet, InIter vég, UnPred pred) | **true** értékkel tér vissza, ha a *pred* a [*kezdet*, *vég*) tartomány minden elemére **true** értéket ad, vagy ha a lista üres. |
| **bool** ***any_of***(InIter kezdet, InIter vég, UnPred pred) | **true** értékkel tér vissza, ha a *pred* a tartomány legalább egy elemére **true** értéket ad. Ha a lista üres, **false** értéket ad. |
| **bool** ***none_of***(InIter kezdet, InIter vég, UnPred pred) | **true** értékkel tér vissza, ha a *pred* a tartomány minden elemére **false** értéket ad, vagy ha a lista üres |

| | |
|---|---|
| `InIter find_if_not(InIter kezdet, InIter vég, UnPred pred)` | Visszaad egy iterátort a tartomány azon elemére, amelyre a *pred* **false** értékkel tér vissza. Ha nincs ilyen elem, a visszatérési érték a *vég* hivatkozás. |
| `bool is_permutation(FwIter kezdet1, FwIter vég1, FwIter kezdet2)`<br>`bool is_permutation(FwIter kezdet1, FwIter vég1, FwIter kezdet2, BinPred pred)` | **true** értékkel tér vissza, ha a [*kezdet1*, *vég1*) tartomány elemei meggyeznek a *kezdet2* pozíción kezdődőkkel, a sorrendtől függetlenül. Az elemek egyezésének vizsgálatára saját, kétparaméteres pred függvényt is használhatunk. |

| | |
|---|---|
| `OutIter copy_if(InIter kezdet, InIter vég, OutIter eredmény, Pred pred)` | A megadott tartomány azon elemeit másolja az *eredmény* kezdetű területre, amelyekre a *pred* **true** értékkel tér vissza. A visszaadott iterátor a célterület utolsó adata utáni pozícióra hivatkozik. |
| `OutIter copy_n(InIter kezdet, Méret n, OutIter eredmény)` | A megadott tartomány első n elemét másolja az *eredmény* kezdetű területre. |
| `OutIter move(InIter kezdet, InIter vég, OutIter eredmény)` | A megadott tartomány elemeit áthelyezi az *eredmény* kezdetű területre. Az eredeti tartomány elemeinek értéke definiálatlan lesz. |
| `OutIter move_backward(InIter kezdet, InIter vég, OutIter eredmény)` | A megadott tartomány elemeit áthelyezi az *eredmény* kezdetű területre. Egymást átfedő tartományok esetén ezt kell használni! |
| `void shuffle(RAIter kezdet, RAIter vég, RandNGen&& g)` | Átrendezi a tartomány elemeit a megadott g véletlen szám generátor felhasználásával. |

| | |
|---|---|
| `bool is_partitioned(InIter kezdet, InIter vég, UnPred pred)` | **true** értékkel tér vissza, ha a tartomány azon elemei, amelyre a predikátum **true** értéket ad, megelőzik azokat, amelyekre **false**-t ad. |
| `pair<OutIter1, OutIter2> partition_copy(InIter kezdet, InIter vég, OutIter1 ered_true, OutIter2 ered_false, UnPred pred)` | A megadott tartomány elemeit szétmásolja a predikátum értéke alapján. A visszaadott pár elemei az új tartományok végére hivatkoznak. |
| `FwIter partition_point(FwIter kezdet, FwIter vég, UnPred pred)` | A particionált tartomány első olyan elemére ad vissza hivatkozást, amelyre a predikátum értéke hamis. |

| | |
|---|---|
| `bool is_sorted(FwIter kezdet, FwIter vég)`<br>`bool is_sorted(FwIter kezdet, FwIter vég, Compare comp)` | **true** értékkel tér vissza, ha a [*kezdet*, *vég*) tartomány elemei növekvő sorrendben rendezettek. Első esetben az **operator**<, míg a másodikban a *comp* függvényt használja az algoritmus. |
| `FwIter is_sorted_until(FwIter kezdet, FwIter vég)`<br>`FwIter is_sorted_until(FwIter kezdet, FwIter vég, Compare comp)` | A tartomány első olyan elemének iterátorával tér vissza, amely nem követi a növekvő sorrendet. |
| `bool is_heap(RAIter kezdet, RAIter vég)`<br>`bool is_heap(RAIter kezdet, RAIter vég, Compare comp)` | **true** értékkel tér vissza, ha a tartomány elemei halmot alkotnak (a ***make_heap()*** hívás rendezte el őket). |
| `RAIter is_heap_until(RAIter kezdet, RAIter vég)`<br>`RAIter is_heap_until(RAIter kezdet, RAIter vég, Compare comp)` | A tartomány első olyan elemének iterátorával tér vissza, amely nem követi a halom elrendezést. |

| | |
|---|---|
| `T min(initializer_list<T> t)`<br>`T min(initializer_list<T> t, Compare comp)` | A listában megadott elemek közül a legkisebbet adja vissza. |
| `T max(initializer_list<T> t)`<br>`T max(initializer_list<T> t, Compare comp)` | A listában megadott elemek közül a legnagyobbat adja vissza. |

| | |
|---|---|
| `pair<const T&, const T&> minmax(const T& a, const T& b)`<br>`pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp)`<br>`pair<const T&, const T&> minmax(initializer_list<T> t)`<br>`pair<const T&, const T&> minmax(initializer_list<T> t, Compare comp)` | Párban adja vissza a megadott elemek közül legkisebbet (*first*) és a legnagyobbat (*second*). |
| `pair<FwIter, FwIter> minmax_element(FwIter kezdet, FwIter vég)`<br>`pair<FwIter, FwIter> minmax_element(FwIter kezdet, FwIter vég, Compare comp)` | Párban adja vissza a megadott tartomány elemei közül a legkisebbre (*first*) és a legnagyobbra (*second*) hivatkozó iterátorokat. |

| | |
|---|---|
| `void iota(FwIter kezdet, FwIter vég, T érték)` | Az adott tartomány feltölti a megadott értéket egyesével növelgetve. (<*numeric*>) |

A táblázatban használt rövidítések:

| | |
|---|---|
| *BinPred* | Kétparaméteres predikátum függvény, amely **true** értékkel jelzi, ha az argumentumaik megfelelnek a feltételeknek. |
| *Compare* | Kétparaméteres összehasonlító függvény, amely **true** értékkel jelzi, ha az első argumentum kisebb a másodiknál. |
| *FwIter* | ***ForwardIterator*** |
| *InIter* | ***InputIterator*** |
| *OutIter* | ***OutputIterator*** |
| *RAIter* | ***RandomAccessIterator*** |
| *T* | Valamilyen típus |
| *UnPred* | Egyparaméteres predikátum függvény, amely **true** értékkel jelzi, ha az argumentuma megfelel a feltételeknek. |

Az algoritmusok használatát az alábbi példa szemlélteti:

```
#include <iostream>
#include <algorithm>
#include <array>
#include <random>
#include <chrono>
using namespace std;
using namespace chrono;

int Fibo() {
    static int a0=0, a1=1;
    int a2=a0+a1;
    a0=a1; a1=a2;
    return a2;
}

auto hasonlit = [](int a, int b) { return a>b;};

int main () {
    array<int,7> a;

    generate_n(begin(a), 7, Fibo);
    reverse(begin(a), end(a));
    for (int& x: a) cout << x <<' '; cout << endl;
    bool rendezett = is_sorted(begin(a), end(a), hasonlit);
    if (rendezett)
        cout << "rendezett" << endl;

    unsigned kezdo = system_clock::now().time_since_epoch().count();
    shuffle (begin(a), end(a), default_random_engine(kezdo));
    cout << "Kevert: ";
```

```
    for (int& x: a) cout << x << ' '; cout << endl;

    auto mm = minmax_element(begin(a), end(a));
    cout << "min: " << *mm.first << "  max: " << *mm.second << endl;
}
```

A program futásának eredménye:

```
21 13 8 5 3 2 1
rendezett
Kevert: 13 5 2 3 1 21 8
min: 1  max: 21
```

## 4.10 Rekordok új megközelítésben (tuple)

A rekord (*tuple*) olyan statikus tároló, amely több, tetszőleges típusú érték tárolására képes. Hasonló megfontolások mellett alkalmazhatjuk, mint a két adat tárolását végző párt (***pair***<>, <*utility*>).

A rekord leírását és használatát segítő sablonokat a <*tuple*> fejállomány tartalmazza:

| | |
|---|---|
| ***tuple***<> | A rekordot definiáló osztálysablon. |
| ***tuple_size***<> | Az osztálysablon **constexpr *size_t*** típusú ***value*** tagja megadja a rekord elemeinek számát. |
| ***tuple_element***<> | A rekord i. elemének lekérdezését segítő osztálysablon, melynek ***type*** tagjával megfelelő típusú változót hozhatunk létre. |
| ***make_tuple()*** | Függvénysablon, amely rekordot állít elő az argumentumában szereplő értékekből. |
| ***forward_as_tupe()*** | Jobbérték referenciákból készít rekordot a függvénysablon, az argumentumában megadott értékekből. |
| ***tie()*** | A függvénysablon egy rekordot készít, amely balérték referenciákat tartalmaz az argumentumaihoz. |
| ***tuple_cat()*** | A megadott rekordok elemeit egyetlen rekordban egyesítő függvénysablon. |
| ***get()*** | A függvénysablon egy referenciát ad vissza a rekord i. eleméhez, amelyek az értékadás minkét oldalán felhasználhatunk. |
| ***ignore*** | Objektum, amely figyelmen kívül hagy minden rá irányuló értékadást. (A ***tie()*** hívásban használjuk, a nem kívánt rekordelemek kihagyására.) |

Az rekordokkal kapcsolatos sablonok alkalmazását az alábbi példaprogramban foglaltuk össze. A megjegyzések segítenek a kód megértésében:

```
#include <iostream>
#include <string>
#include <utility>
#include <tuple>
using namespace std;

void Kiiras(tuple<int&&, double&&, string&&> r) {
    cout << get<0>(r) << ", " << get<1>(r) << ", ";
    cout << get<2>(r) << endl;
}

int main () {
    // rekord konstruálása, az elemek számának lekérdezése és
    // az elemek egyenkénti kiolvasása:
    tuple<int,double,string> r1(2004, 10.2, "Iván");
    int elemszam = tuple_size<decltype(r1)>::value;

    tuple_element<0,decltype(r1)>::type elem0 = get<0>(r1);
    tuple_element<1,decltype(r1)>::type elem1 = get<1>(r1);
```

```
    tuple_element<2,decltype(r1)>::type elem2 = get<2>(r1);

    cout << "elemek száma: " << elemszam << endl;
    cout << "1. elem: " << elem0 << endl;
    cout << "2. elem: " << elem1 << endl;
    cout << "3. elem: " << elem2 << endl;

    // rekord létrehozása adatokból, és kiolvasása
    int a;
    double b;
    string s;
    auto r2 = make_tuple(2004, 10.2, "Iván");
    auto r3 = make_tuple<int, double, string>(2004, 10.2, "Iván");
    tie(a, b, s) = r2;
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    cout << "s: " << s << endl;

    // rekordok összekapcsolása, és részleges kiolvasása:
    pair<string, int> sn = {"MXY", 238};
    auto r4 = tuple_cat(r2, tuple<string, int>(sn));
    string ss;
    int nn;
    tie(ignore, ignore, ignore, ss, nn) = r4;
    cout << ss << nn << endl;

    // rekord módosítása, és továbbítása függvénynek
    // jobbérték rerenciával
    Kiiras(forward_as_tuple(1998, 14882.0, string("C++")));
    tuple<int&&, double&&, string&&> r5(1998, 14882.0, string("C++"));
    get<0>(r5) = 2011;
    get<1>(r5) = 14882.0;
    get<2>(r5) = string("C++11");
    Kiiras(r5);
}
```

A program futásának eredménye:

```
elemek száma: 3
1. elem: 2004
2. elem: 10.2
3. elem: Iván
a: 2004
b: 10.2
s: Iván
MXY238
1998, 14882, C++
2011, 14882, C++11
```

## 4.11 A szabványos C++ nyelv deklarációs állományai

A táblázatban kék színnel jelöltük az újonnan bevezetett, míg bordó színnel a módosított fejállományokat.

***A C++ nyelvet támogató könyvtár***

| | |
|---|---|
| Típusok (*NULL*, *size_t* stb.) | <*cstddef*> |
| Rögzített méretű típusok és formátum makrók | <*cstdint*>, <*cinttypes*> |
| Az implementáció jellemzői | <*limits*>, <*climits*>, <*cfloat*> |
| Programindítás és -befejezés | <*cstdlib*> |
| Dinamikus memóriakezelés | <*new*> |
| Típusazonosítás | <*typeinfo*>, <*typeindex*>, <*type_traits*> |
| Egyéb futásidejű támogatás | <*cstdarg*>, <*csetjmp*>, <*ctime*>, <*csignal*>, <*cstdlib*>,<*cstdbool*>, <*cstdalign*>, <*initializer_list*> |

| | |
|---|---|
| ***Hibakezelési könyvtár*** | |
| Kivételkezelés | <exception> |
| Kivételosztályok | <stdexcept> |
| *Assert* makrók | <cassert> |
| Hibakódok | <cerrno> |
| Rendszerhiba-támogatás | <system_error> |
| ***Általános szolgáltatások könyvtára*** | |
| Műveleti elemek (*STL*) | <utility> |
| Műveletobjektumok (*STL*) | <functional> |
| Memóriakezelés (*STL*) | <memory>, <scoped_allocator> |
| Speciális osztálysablonok (*STL*) | <bitset>, <tuple> |
| Dátum- és időkezelés | <ctime>, <chrono> |
| ***Sztring könyvtár*** | |
| Sztring osztályok | <string> |
| Nullavégű karakterláncok kezelését segítő függvények | <cctype>, <cwctype>, <cstring>, <cwchar>, <cuchar> |
| ***Országfüggő (helyi) beállítások könyvtára*** | |
| A helyi sajátosságok szabványos kezelése | <locale> |
| A C könyvtár helyi beállításai | <clocale> |
| Unicode konverziók támogatása | <codecvt> |
| ***A tárolók könyvtára (STL)*** | |
| Szekvenciális tárolók (*STL*) | <array>, <deque>, <list>, <vector>, <forward_list> |
| Asszociatív tárolók (*STL*) | <map>, <set> |
| Nem rendezett asszociatív tárolók (*STL*) | <unordered _map>, <unordered_set> |
| Tároló adaptációk (*STL*) | <queue>, <stack> |
| ***Iterátorok (általánosított mutatók) könyvtára (STL)*** | |
| Iterátorelemek, előre definiált iterátorok, adatfolyam iterátorok (*STL*) | <iterator> |
| ***Algoritmusok könyvtára*** | |
| Adatsorkezelés, rendezés, keresés stb. (*STL*) | <algorithm> |
| A C könyvtár algoritmusai | <cstdlib> |
| ***Numerikus könyvtár*** | |
| Komplex számok | <complex> |
| Számtömbök | <valarray> |
| Általánosított numerikus műveletek (*STL*) | <numeric> |
| A C könyvtár numerikus elemei | <cmath>, <cstdlib> |
| Fordításidejű racionális aritmetika | <ratio> |
| Véletlen szám előállítása | <random> |
| A lebegőpontos környezet kezelése | <cfenv> |
| ***Input/output könyvtár*** | |
| Forward (előrevetett) deklarációk | <iosfwd> |
| Szabványos iostream objektumok | <iostream> |
| Az iostream osztályok alaposztálya | <ios> |
| Adatfolyam pufferek | <streambuf> |
| Adatformázás és manipulátorok | <istream>, <ostream>, <iomanip> |
| Sztring adatfolyamok | <sstream> |
| Fájl adatfolyamok | <fstream>, <cstdio> |

| | |
|---|---|
| ***Reguláris kifejezések könyvtár*** | |
| Reguláris kifejezések kezelése | <*regex*> |
| ***Műveletek a konkurens programozáshoz*** | |
| Atomi típusok és -műveletek | <*atomic*> |
| ***Thread (programszál) könyvtár*** | |
| Szálak létrehozása és vezérlése | <*thread*>, <*mutex*>, <*condition_variable*>, <*future*> |