

Objektumorientált programozás



Objektumalapú programozás a C++ programozási nyelvben

Alapok

Darvay Zsolt

Alapok (C és C++)



Alapok

Alapok (C és C++)



1. Típusok és nevek a forráskódban
2. Állandók és változók
3. Hatókörök, névterek és az előfeldolgozó
4. Bevitel és kivitel
5. Kifejezések
6. Utasítások
7. Mutatók
8. Függvények
9. Struktúrák és típusok
10. Állománykezelés

1. Típusok és nevek a forráskódban



1.1. A forrásprogramok felépítése

1.2. Nevek és kulcsszavak

1.3. Alapvető típusok

1.1. A forrásprogramok felépítése



- ▶ A program felépítése:
 - ▶ függvényekből áll;
 - ▶ a fő függvény neve: **main**.
- ▶ A program tárolása:
 - ▶ általában .c vagy .cpp kiterjesztésű állomány (forráskód) - szövegszerkesztés.
- ▶ Fordítás -> lefordított program (.obj).
- ▶ Szerkesztés -> végrehajtható állomány.

1.2. Nevek és kulcsszavak

- ▶ Név (azonosítónév): betűk, '_' karakter (aláhúzás) és számjegyek sorozata. Betűvel vagy '_' karakterrel kezdődik.
 - ▶ Kisbetűk és nagybetűk különböznek!
- ▶ A névnek jellemzőnek kell lennie.
- ▶ Kulcsszó (lefoglalt szó)
 - ▶ pl. if, for, while, stb.
- ▶ A név nem lehet kulcsszó.

1.3. Alapvető típusok

- ▶ char - karakter;
- ▶ double - valós, kétszeres pontosság;
- ▶ float - valós, egyszeres pontosság;
- ▶ int - egész;
- ▶ bool - logikai (csak a C++-ban).
- ▶ Módosítójelzők:
 - ▶ long (hosszú), short (rövid),
 - ▶ signed (előjeles), unsigned (előjel nélküli).

A módosítójelzők használata

- ▶ Az ismertetett típusokhoz csatolva új típus képezhető.
- ▶ Például:
 - ▶ unsigned - ugyanaz mint: unsigned int,
 - ▶ unsigned long - ugyanaz mint: unsigned long int.

2. Állandók és változók



2.1. Literálok

2.2. Változódeklarációk

2.3. Változók kezdőértéke

2.4. Konstansdeklarációk

2.1. Literálok



- ▶ Olyan állandó (konstans), amelyhez nem rendelünk azonosítónevet.
- ▶ A típusa és értéke jellemzi.
- ▶ Az érték nem módosítható futtatás közben.
- ▶ Egész literálok
- ▶ Lebegőpontos literálok
- ▶ Karakterliterálok
- ▶ Karakterlánc-literálok

Egész literálok

- ▶ Decimális (10-es számrendszerbeli),
- ▶ hexadecimális (16-os számrendszerbeli),
- ▶ oktális (8-as számrendszerbeli) állandók.
- ▶ Az egész literálok tárolása függ a fordítóprogramtól. Például a Borland C++ 3.1-ben az int típus mérete 2 byte, a Visual C++ .NET-ben pedig 4 byte, és ez hatással van az állandók tárolására is.

Decimális literálok

- ▶ Példák:
- ▶ 357 (int), -6962 (int),
- ▶ 12300L vagy 12300l (long),
- ▶ 14000U vagy 14000u (unsigned),
- ▶ 11700LU vagy 11700UL vagy 11700lu vagy 11700ul (unsigned long).

Decimális literálok tárolása

- ▶ Borland C++ 3.1-ben:
- ▶ 72000 (long),
- ▶ 85000U vagy 85000u (unsigned long).
- ▶ Visual C++ .NET-ben:
- ▶ 72000 (int),
- ▶ 85000U vagy 85000u (unsigned int).

Hexadecimális és oktális literálok



- ▶ 0..... (8-as számrendszer),
- ▶ 0x..... vagy 0X..... (16-os számrendszer).
- ▶ Példák:
- ▶ 0100 a 10-es számrendszerben: 64,
- ▶ 0x100 a 10-es számrendszerben: 256.
- ▶ Ebben az esetben is használható az 'U', 'u' illetve 'L', 'l' utótag.

Lebegőpontos literálok

- ▶ Tárolás double. Ha a 'F' vagy 'f' utótagot használjuk, akkor float lesz, illetve 'L' vagy 'l' esetén long double.
- ▶ Példák valós (lebegőpontos) literálokra:
- ▶ 311. (double), -5.24 (double),
- ▶ 5.1e3 (double), .1e-5 (double),
- ▶ -2.54f (float), 1.5L (long double).

Karakterliterálok



- ▶ 'A' - egy karakter.
- ▶ C-ben int típusú, C++ -ban pedig char.
- ▶ Általában a karakternek megfelelő ASCII kódot tároljuk.
- ▶ Escape karakterek
- ▶ Fehér (láthatatlan) karakterek
- ▶ „Széles” karakterliterálok

Escape karakterek

▶ \n	NL (LF)	újsor (sortörés)	10
▶ \r	CR	kocsivissza	13
▶ \t	HT	vízszintes tabulátor	9
▶ \"	"	idézőjel	34
▶ \'	'	aposztróf	39
▶ \\	\	fordított perjel	92
▶ \ooo	ooo	oktális szám	
▶ \xhh...	hh...	hexadecimális szám	

Példák escape karakterekre

- ▶ Escape szekvenciák (escape jelsorozatok):
- ▶ 34, `"\"`, `\42` és `\x22` ugyanaz.
- ▶ 65, `'A'`, `\101` és `\x41` ugyanaz.
- ▶ A null karakter: `\0` (az értéke 0). Különbözik a `'0'` karaktertől, aminek az értéke 48 az ASCII készletben.
- ▶ A null karakternek fontos szerepe van a karakterlánc-literálok tárolásában.

Fehér karakterek



- ▶ Láthatatlan karakterek,
- ▶ „üres hely” karakterek (térközök):
 - ▶ szóköz: ' ',
 - ▶ vízszintes tabulátor (behúzás): '\t',
 - ▶ újsor: '\n'.
- ▶ Bizonyos C, illetve C++, fordítók ezeken kívül még más karaktereket is fehér karakterként kezelhetnek.

„Széles” karakterliterálok

- ▶ Nagyobb karakterkészletek (például a Unicode) tárolására: `wchar_t`.
- ▶ C-ben: `wchar_t` egy típusdeklarációval (`typedef`) van definiálva.
- ▶ C++ -ban: `wchar_t` beépített típus.
- ▶ Például a Visual C++ .NET-ben:
- ▶ L'a' unsigned short 2 byte
- ▶ 'a' char 1 byte

Karakterlánc-literálok

- ▶ Karaktersorozatok (C stílusú string-ek).
- ▶ A null karakterrel végződnek. Például:
- ▶ "ABC" tárolása: 'A', 'B', 'C', '\0'.
- ▶ Használhatjuk az escape karaktereket is.
- ▶ Karakterlánc-literálok típusa
- ▶ Hosszú karakterláncok
- ▶ L előtagú karakterláncok

Karakterlánc-literálok típusa

- ▶ Típusa C-ben, és a C++ régebbi változataiban: `char *` (mutató egy karakterhez).
- ▶ Típusa C++ -ban: megfelelő méretű állandó karakterből álló tömb. Például "ABC" típusa `const char[4]`.
- ▶ A karakterlánc-literálokat egy `char *` típusú mutatónak átadhatjuk.

Hosszú karakterláncok

- ▶ A következő sorban folytatni lehet.

- ▶ 1. példa (''-al jelöljük a szóközt):

"abcd\
...

x"

A kapott karakterlánc: "abcd...x"

- ▶ 2. példa:

"abcd"

...."efg"

A kapott karakterlánc: "abcdefg".

L előtagú karakterláncok

- ▶ Széles karakterekből állnak.

- ▶ Példa:

L"xyzt"

- ▶ C++ -ban típusuk egy megfelelő méretű `const wchar_t[]`.

2.2. Változódeklarációk



- ▶ Változó: adat, ami módosíthatja az értékét a program futtatása közben.
- ▶ Minden változót deklarálni kell.
- ▶ Deklarációk szerkezete
- ▶ Egyszerű típusú változók deklarálása
- ▶ Tömbök deklarálása

Deklarációk szerkezete



- ▶ Tetszőleges deklarációra (nem csak a változókra) vonatkozik:
 - ▶ minősítő (például: virtual, extern)
 - ▶ alaptípus
 - ▶ deklarátor
 - ▶ kezdőérték-adó kifejezés
- ▶ A minősítő és a kezdőérték-adó kifejezés elmaradhat.

Egyszerű típusú változók deklarációja

- ▶ Deklaráció:
- ▶ típus nevek_listája;
- ▶ Példák:
- ▶ `int x, y, z;`
- ▶ `char c;`
- ▶ `int* p, u, v; // az operátor csak egy //névre vonatkozik (csak a p lesz mutató).
//Ha lehet kerülni kell az ilyen deklarációt.`

Tömbök deklarálása

- ▶ Deklaráció: tömb (táblázat)
- ▶ típus név $[h_1][h_2]\dots[h_n]$;
 - ▶ típus: elemek típusa;
 - ▶ h_i : i -edik index felső határa, tehát
 - ▶ az indexek 0 és h_i-1 között változhatnak;
 - ▶ n : tömb dimenziója, tehát
 - ▶ $n=1$ vektor, $n=2$ mátrix.
- ▶ Hivatkozás: név $[i_1][i_2]\dots[i_n]$

Példák tömbökre

- ▶ `int a[10];` // egész elemekből álló tömb.
- ▶ Hivatkozás az elemekre:
 - `a[0]` - első elem
 - `a[9]` - tízedik elem (utolsó)
 - `a[10]` - hiba, mivel csak tíz elem számára van lefoglalva memóriaterület.
- ▶ `int b[10][20];` // egy 10 sorból és 20 oszlopból álló mátrix.
- ▶ `b[i][j]` - i-edik sor, j-edik oszlop

Karakterekből álló tömbök

- ▶ `char s[20];` //egy karakterekből álló vektor, tulajdonképpen ez egy karakterlánc.
- ▶ `s[0]` az első karakter.
- ▶ Nem használhatók operátorok a karakterláncokra. E helyett a `string.h` (illetve `cstring`) állományokban standard függvények vannak deklarálva. Például: `strcpy`, `strcat`, `strcmp`.

2.3. Változók kezdőértéke

- ▶ Deklaráció: minősítő, alaptípus, deklarátor, **kezdőérték-adó kifejezés**.
- ▶ Példák:
 - ▶ `int n = 6;`
 - ▶ `int t[] = {11, 22, 33, 44, 55, 66};`
 - ▶ `static int *p = t;`

Alapértelmezett kezdeti érték

- ▶ A globális, a névtéren belül megadott és a helyi statikus változók kezdeti értéke zéró lesz alapértelmezés szerint. Ezeket **statikus objektumoknak** is nevezzük.
- ▶ A helyi, de nem statikus változók (**automatikus objektumok**), illetve a dinamikus módon létrehozott változók (**dinamikus objektumok**) nem rendelkeznek alapértelmezett kezdeti értékkel. Lásd még:

[Memóriakezelés](#)

Statikus változók a függvény testében

```
#include <iostream>
using namespace std;
void kiir_elemek(int *t, int n)
{
    static int *p = t;
    static int k = 0;
    int *x = t;
    cout << "Statikus: " << *p;
    cout << "\tNem statikus: " << *x << endl;
    if ( k < n-1 ) { p++; x++; k++; }
    else { p = t; x = t; k = 0; }
}
```

Fő függvény

```
void main()
{
    int t[] = {11, 22, 33, 44, 55, 66};
    int n = 6;
    for(int i = 0; i < 10; i++)
        kiir_elemek(t, n);
}
```

- ▶ Minden változónak érdemes kezdeti értéket adni akkor is, ha van alapértelmezett kezdeti értéke.

Kimenet



Statikus: 11	Nem statikus: 11
Statikus: 22	Nem statikus: 11
Statikus: 33	Nem statikus: 11
Statikus: 44	Nem statikus: 11
Statikus: 55	Nem statikus: 11
Statikus: 66	Nem statikus: 11
Statikus: 11	Nem statikus: 11
Statikus: 22	Nem statikus: 11
Statikus: 33	Nem statikus: 11
Statikus: 44	Nem statikus: 11

2.4. Konstansdeklarációk



- ▶ Nem változtathatják értéküket futási időben. Különben már a fordításkor hibát észlelnénk.
- ▶ A **const** minősítővel jelezzük, hogy konstansról van szó.

Példa

```
#include <iostream>
using namespace std;
void main() {
    const int x = 10;
    // x++; //hiba
    int y = 20; y++;
    cout << y << endl;
    char allat[] = "kecske";
    char madar[] = "galamb";
    char *p = allat; p[0] = 'f';
    cout << p << endl;
    p = madar;
    cout << p << endl;
```

```
const char *s = allat;
// s[0] = 'f'; //hiba
s = madar;
cout << s << endl;
char * const t = allat;
t[0] = 'f';
// t = madar; // hiba
cout << t << endl;
const char * const w = allat;
// w[0] = 'f'; // hiba
// w = madar; // hiba
}
```

Kimenet



21

fecske

galamb

galamb

fecske

3. Hatókörök, névterek és az előfeldolgozó



3.1. Az előfeldolgozó

3.2. Lokális és globális hatókörök

3.3. Memóriakezelés

3.4. A névtér és tagjai

3.5. A using deklaráció és direktíva

3.1. Az előfeldolgozó



- ▶ Előfeldolgozó (előfordító, preprocesszor).
- ▶ Egy makró-feldolgozó rendszer.
- ▶ Fordítás előtti feldogozása a programnak.
- ▶ Forráskód beékelése
- ▶ Szimbolikus állandók definiálása
- ▶ Makrók definiálása
- ▶ Feltételes fordítás

Forráskód beékelése

- ▶ Standard állományok:

`#include <állománynév>`

- ▶ Saját állományok:

`#include "állománynév"`

- ▶ Elérési útvonalat is tartalmazhatnak (a karakterlánc belsejében a `\` karaktert meg kell duplázni).
- ▶ Általában fejállományokat (header) ékelünk be (kiterjesztésük `.h`). Az újabb C++ fordítók szabványos fejállományainak nincs kiterjesztése.

Szimbolikus állandók

- ▶ Jelképes konstansoknak is nevezzük.

`#define név karakterek_sorozata`

- ▶ A definiálás helyétől az állomány végéig a név összes előfordulását a karakterek sorozatára cseréli.
- ▶ Ha az

`#undef név`

jelen van, akkor csak addig a sorig.

3.2. Lokális és globális hatókörök



- ▶ Egy név deklarációja azt jelenti, hogy az illető név egy bizonyos hatókörben használható.
- ▶ A névterek olyan hatókörök, amelyeket egy névvel láttunk el.
- ▶ Lokális (helyi) hatókörök
- ▶ Globális hatókörök
- ▶ A lokális és globális hatókörök is névterek.

Lokális hatókörök

- ▶ Egy függvény lokális változója csak a függvény belsejében használható. Ez a blokk az illető változó hatóköre. Példa:

```
void f()  
{  
    int x;  
    ...  
}
```

Az x változó csak a függvény belsejében használható. Az x hatóköre ez a blokk.

Globális hatókörök



- ▶ Egy globális név hatóköre a deklaráció helyétől az állomány végéig terjed. Egy extern típusú deklaráció kiterjesztheti ezt a hatókört más állományra is.
- ▶ Egy név akkor globális, ha a függvényeken, osztályokon és névtereken kívül van megadva.

3.3. Memóriakezelés

- ▶ A változók helyet kaphatnak:
 - ▶ a statikus memóriában;
 - ▶ az automatikus memóriában;
 - ▶ a szabad tárban.
- ▶ Lásd még:

[Alapértelmezett kezdeti érték](#)

A statikus memória



- ▶ A statikus memóriában lesznek elhelyezve a globális változók, a névtereken belül megadott változók, a függvények belsejében deklarált statikus változók, és az osztályok statikus tagjai.
- ▶ A szerkesztő (linker) foglalja le a statikus memóriában a helyet, a program futtatásának teljes időtartamára.

Az automatikus memória



- ▶ A függvények paramétereit és a helyi változókat az automatikus memóriában hozza létre a rendszer.
- ▶ Az adott blokk minden egyes végrehajtásakor új példányok jönnek létre ezekből a változókból, a blokk végén pedig automatikusan fel lesz szabadítva az általuk lefoglalt memóriaterület.
- ▶ Az automatikus változók a **veremben** vannak elhelyezve.

A szabad tár



- ▶ A szabad tárban elhelyezett elemek memóriakezelését a programozó hivatott megvalósítani.
- ▶ Sem a memóriaterület lefoglalását, sem a felszabadítását nem végzi el automatikus módon a rendszer.
- ▶ A változók a **dinamikus memóriában**, vagy a **kupacban (heap)** vannak elhelyezve.

3.4. A névtér és tagjai



- ▶ A névtér fogalma
- ▶ Példa névtérre
- ▶ A névtér tagjaira való hivatkozás

A névtér fogalma

- ▶ Egy névtér segítségével bizonyos deklarációk csoportosítását valósíthatjuk meg.
- ▶ Az összetartozó neveket ugyanabba a névtérbe helyezhetjük el.

- ▶ A névtér megadása:

namespace név

{

// deklarációk, definíciók

}



A névtér neve

Példa névtérre

```
#include <iostream>
namespace vektor {
    int *elem;
    int dim;
    void init(int *e, int d);
    void felszabadit();
    void negyzetreemel();
    void kiir();
}
```

Visual C++ 6.0-ban
iostream.h is lehetne
(nem ugyanaz).

Egy egész elemekből
álló vektor kezelésére
vonatkozó névtér.

A névtér tagjaira való hivatkozás

- ▶ A hatókör operátort (`::`) használjuk.
- ▶ Hivatkozás: `névtér::tag`.
- ▶ Példa: `vektor::dim = 6;`
- ▶ Ez a hivatkozás a névtér függvényeire is érvényes. Ebben az esetben a függvény neve elé kerül a `névtér::`.
- ▶ A tagokra való hivatkozás egyszerűbben megvalósítható, ha „using” deklarációt vagy direktívát használunk.

3.5. A using deklaráció és direktíva



- ▶ A using deklaráció
- ▶ A using direktíva
- ▶ Az iostream és iostream.h közti különbség

A using deklaráció

- ▶ Ha egy névtér tagjára többször kell hivatkoznunk a névteren kívül, akkor a „using” deklarációval adhatjuk meg, hogy a tag ezentúl önállóan is használható.
- ▶ A deklaráció:
`using névtér_neve::tag_neve;`
- ▶ Például: `using vektor::dim;`
- ▶ Ekkor `dim = 14;`
ugyanaz, mint `vektor::dim = 14;.`

A using direktíva

- ▶ Akkor használjuk, ha a névtér összes tagját elérhetővé szeretnénk tenni.

- ▶ Alakja:

using namespace névtér;

- ▶ Példa: `using namespace vektor;`
- ▶ A globális using direktívákat a C++ régebbi változataival való kompatibilitás megvalósítására használhatjuk.

Az iostream és iostream.h közti különbség



- ▶ A Visual C++ 6.0-ban:
- ▶ Az iostream fejállomány az adatfolyamokra vonatkozó deklarációkat az **std** standard névtérbe helyezi.
- ▶ Az iostream.h a deklarációkat nem helyezi névtérbe.
- ▶ A Visual C++ .NET 2003-ban már nem használható az iostream.h mivel ezt elavultnak tekintik.

Az iostream fejállomány használata



//using direktívával

```
#include <iostream>
using namespace std;
void kiiras() {
...
cout << endl;
}
```

//using direktíva nélkül

```
#include <iostream>
void kiiras() {
...
std::cout << std::endl;
}
```

Az `iostream.h` fejláallomány használata

```
#include <iostream.h>
```

```
void kiiras() {
```

```
...
```

```
cout << endl;
```

```
}
```

- ▶ Ebben az esetben nem szükséges az `std` névtér. Csak régebbi C++ fordítók esetén használható.

Az init függvény



```
void vektor::init(int *e, int d)
{
    elem = new int[d];
    dim = d;
    for(int i=0; i < dim; i++)
        elem[i] = e[i];
}
```

A felszabadít és negyzetreemelés függvények

```
void vektor::felszabadit() {  
    delete []elem;  
}  
  
void vektor::negyzetreemel() {  
    for(int i = 0; i < dim; i++)  
        elem[i] *= elem[i];  
}
```

A kiír függvény

```
void vektor::kiir() {  
    for(int i = 0; i < dim; i++)  
        std::cout << elem[i] << '\t';  
    std::cout << std::endl;  
}
```

- ▶ Ha a **using namespace std;** jelen van, akkor az **std::** elhagyható.

A fő függvény



```
void main() {  
    int t[]={11, 22, 33, 44};  
    using namespace vektor;  
    init(t, 4);    //ha nincs using, vektor::init  
    negyzetreemel();  
    kiir();  
    felszabadit();  
}
```

4. Bevitel és kivitel



4.1. Bevitel és kivitel a C-ben

- ▶ A printf és a scanf függvények
- ▶ A getchar és putchar makrók
- ▶ A gets és puts függvények

4.2. Adatfolyamok

5. Kifejezések

$$x += y / f(z) - (b < < 3)$$

5. Kifejezések



- 5.1. Operandusok és operátorok
- 5.2. Kiterjesztés és konverzió
- 5.3. Aritmetikai operátorok
- 5.4. Összehasonlító és logikai operátorok
- 5.5. Bitenkénti operátorok
- 5.6. Értékadó operátorok
- 5.7. Léptető operátorok

5. Kifejezések



5.8. A sizeof operátor

5.9. A cím operátor és a referencia típus

5.10. A zárójel operátorok

5.11. A feltételes operátor

5.12. A vessző operátor

5.13. A hatókör operátor

5.14. A típusazonosító operátor

5.15. Precedencia és kiértékelési irány

5.1. Operandusok és operátorok



- ▶ Egy kifejezés operandusokból és operátorokból áll.
- ▶ Az operátor egy művelet. Ez vonatkozhat egy operandusra (unáris), vagy két operandusra (bináris). Van egyetlen három operandusra vonatkozó operátor (feltételes operátor).
- ▶ Az operandust a típus és érték jellemzi.

Operandusok



Az operandus lehet:

- ▶ állandó
- ▶ szimbolikus állandó
- ▶ változónév
- ▶ tömbnév
- ▶ struktúranév
- ▶ típusnév
- ▶ függvéynév
- ▶ tömb eleme
- ▶ struktúra eleme
- ▶ függvénymeghívás
- ▶ (kifejezés)

Kifejezések kiértékelése

- ▶ Figyelembe kell venni:
 - ▶ a műveletek sorrendjét (precedencia, prioritás);
 - ▶ az azonos prioritású műveletek kiértékelési irányát („kötési” szabályok, vagy „asszociativitás” - nem a legjobb elnevezés);
 - ▶ az automatikus típuskonverzió szabályait.

5.2. Kiterjesztés és konverzió

- ▶ Automatikus típuskonverzió (más néven: alapértelmezés szerinti konverzió, implicit típuskonverzió).
- ▶ Ha az operandusok azonos típusúak akkor az eredmény típusa megegyezik az operandusok típusával.
- ▶ Kiterjesztés (automatikus konverzió, amely megőrzi az értékeket, angolul: promotion)
- ▶ Konverzió

Kiterjesztés a C-ben

- ▶ A karakter típusok egész típussá lesznek konvertálva.
- ▶ A felsoroló típusok egész típusra lesznek alakítva.
- ▶ Példa:

```
enum szinek {piros, sarga, zold} s;
```

```
int x;
```

```
s = piros;    // zéró
```

```
x = s + 1;    // x = 1 lesz.
```


Kiterjesztés a C++-ban

- ▶ Egész típusú kiterjesztés:
 - ▶ egy aritmetikai operátor alkalmazása előtt az int típusnál „alacsonyabb” típusok, és a logikai típus, int típusra lesznek alakítva.
 - ▶ Ha lehet, ugyancsak int típusra alakítja a rendszer a wchar_t típust, a felsoroló típusokat, és a bitmezőket is.
 - ▶ Ha az int nem tudja az illető típus összes értékét tárolni, akkor ezek a típusok unsigned típusra lesznek alakítva (ez lesz a céltípus).

Egész típusú kiterjesztés a C++-ban

- ▶ Az int-nél alacsonyabb típusok: char, signed char, unsigned char, short és unsigned short.
- ▶ A wchar_t típus és a felsoroló típusok esetén, ha az unsigned sem tudja az összes értéket tárolni, akkor a long illetve unsigned long típusokkal próbálkozik a rendszer, ebben a sorrendben.
- ▶ A bitmezők esetén, ha az unsigned sem felel meg a céltípusnak, akkor nem történik kiterjesztés.
- ▶ A logikai típusok esetén a false 0-ra, a true 1-re lesz alakítva.

Konverzió



- ▶ Többféle konverzióra van lehetőség.
- ▶ Ezek közül az aritmetikai konverzió a leggyakoribb.

Aritmetikai konverzió



- ▶ Ha a két operandus típusa különböző, akkor az „alacsonyabb” típusú a „magasabb” típusra lesz konvertálva (csökkenő: long double, double, float, unsigned long, long, unsigned, int).
- ▶ A C++-ban, ha a long tudja tárolni az összes unsigned értéket, akkor egy long és egy unsigned operandus esetén az eredmény long lesz, ellenkező esetben pedig unsigned long.

Meghatározott típuskényszerítés



- ▶ Explicit típuskonverzió (típuskényszerítés) a C-ben (a típusmódosító operátor).
- ▶ Meghatározott típuskényszerítés a C++ programozási nyelvben
- ▶ Típuskonverzió konstruktorral a C++-ban

Explicit típuskonverzió a C-ben



- ▶ Típusmódosító operátor:
(típus) kifejezés
- ▶ A kifejezést az adott típusra alakítja.
- ▶ A típusmódosító operátor magas precedenciája miatt a kifejezés általában egy változónév lesz.
- ▶ Ellenkező esetben a kifejezés helyén (kifejezés) lehet.

Meghatározott típuskényszerítés a C++-ban

- ▶ fordítási időben ellenőrzött típuskonverzió
`static_cast<típus>(kifejezés)`
- ▶ futási időben ellenőrzött típuskonverzió
`dynamic_cast<típus>(kifejezés)`
- ▶ nem ellenőrzött típuskonverzió
`reinterpret_cast<típus>(kifejezés)`
- ▶ konstans típuskonverzió
`const_cast<típus>(kifejezés)`

Típuskonverzió konstruktorral a C++-ban

- ▶ Alakja:

típus(kifejezés)

- ▶ Beépített típusokra ez ugyanaz, mint:

(típus)kifejezés,

- ▶ esetleg (típus)(kifejezés), ha a precedenciára vonatkozó szabályok ezt megkövetelik.

- ▶ Ez a típuskényszerítés saját típusokra is alkalmazható.

5.3. Aritmetikai operátorok

- ▶ unáris: + és -
- ▶ bináris:
 - ▶ multiplikatív: * / %
 - ▶ additív: + és -
- ▶ Példák:
 - ▶ $14 / 3$ eredmény: 4 (egész osztás, hányados).
 - ▶ $14 \% 3$ eredmény: 2 (osztási maradék).

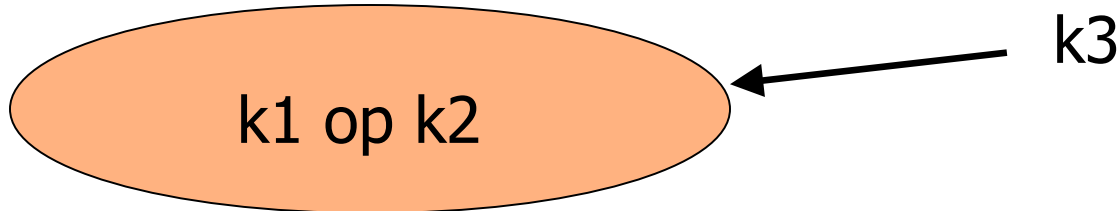
5.4. Összehasonlító és logikai operátorok



- ▶ Összehasonlító operátorok
 - ▶ Relációs operátorok
 - ▶ Egyenlőség és különbözőség
- ▶ Logikai operátorok
- ▶ Összehasonlító és logikai operátorok a C++-ban

Relációs operátorok

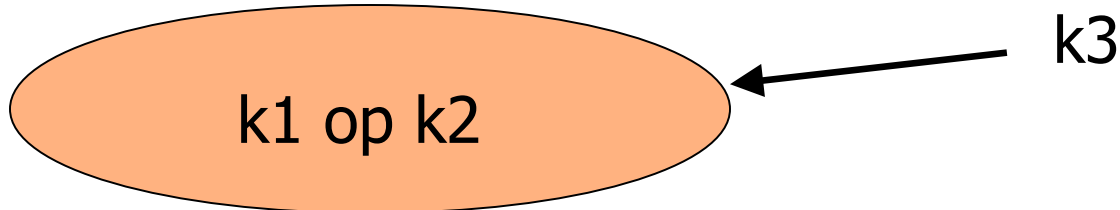
▶ Ezek: < <= > >= (op)



- ▶ k1, k2 és k3 kifejezések
- ▶ A C-ben nincsenek logikai típusok:
- ▶ 0 (hamis), nemzéró (igaz).
- ▶ A C-ben a k3 értéke 1 vagy 0 lesz attól függően, hogy fennáll a feltétel, vagy nem.

Egyenlőség és különbözőség

- ▶ Ezek: == és != (op)



- ▶ k1, k2 és k3 kifejezések
- ▶ Egyenlőség: ==
- ▶ Különbözőség: !=
- ▶ A C-ben a k3 értéke 1 vagy 0 lesz attól függően, hogy fennáll a feltétel, vagy nem.

Logikai operátorok

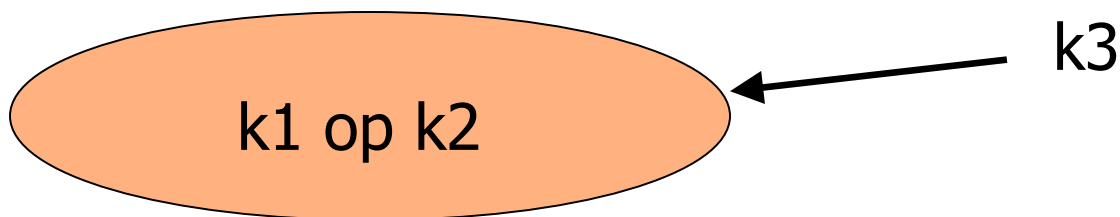
- ▶ Unáris: ! (tagadás)

!kifejezés

- ▶ Értéke 0, ha a kifejezés igaz, illetve 1, ha a kifejezés értéke zéró (hamis).
- ▶ Bináris:
- ▶ && (logikai „és”)
- ▶ || (logikai „vagy”)

Bináris logikai operátorok

- ▶ Ezek: `&&` és `||` (op)



- ▶ `k1`, `k2` és `k3` kifejezések
- ▶ Logikai „és”: `&&`
- ▶ Logikai „vagy”: `||`
- ▶ A C-ben `k3` értéke 1 vagy 0 lesz.

Összehasonlító és logikai operátorok a C++-ban

- ▶ A C++-ban létezik a bool logikai típus.
- ▶ Két értéke lehet: true és false.
- ▶ Kifejezésekben egészzé alakulhatnak. Ekkor a true értéke 1, a false pedig 0 lesz.
- ▶ Ha az egész típusú kifejezés logikai típussá alakul, akkor a zérónak felel meg a false, és a nem zérónak a true.
- ▶ Az összehasonlító és logikai kifejezések értéke a C++-ban logikai típus lesz.

5.5. Bitenkénti operátorok

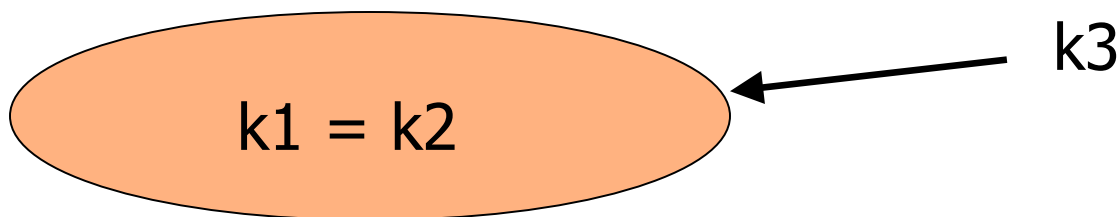
- ▶ Ide tartozik a:
 - ▶ \sim (komplementum)
 - ▶ $<<$ (eltolás balra)
 - ▶ $>>$ (eltolás jobbra)
 - ▶ $\&$ (bitenkénti „és”)
 - ▶ \wedge (bitenkénti „kizáró vagy”)
 - ▶ $|$ (bitenkénti „vagy”)
- ▶ Egész típusokra alkalmazzák. A tárolási területre vonatkoznak.

5.6. Értékadó operátorok

- ▶ Balérték (lvalue): olyan kifejezés, amely szerepelhet az értékadások bal oldalán.
- ▶ Kivételt képeznek a const segítségével megadott balértékek.
- ▶ A nem const-ként megadott balértéket módosítható balértéknek is nevezzük.
- ▶ Egyszerű értékadás
- ▶ „Művelet” és értékadás

Egyszerű értékadás

- ▶ Az „=” operátorral fejezzük ki:



- ▶ `k1`: módosítható balérték kell legyen.
- ▶ A `k1` felveszi `k2` értékét.
- ▶ A `k3` értéke a `k1` megváltoztatott értékével (az átadott értékkel) egyezik meg.

Példák egyszerű értékadásra

- ▶ Kiértékelési irány: jobbról balra

```
int x, y, z;
```

```
x = y = z = 10;
```

- ▶ Az összes változó értéke 10 lesz.

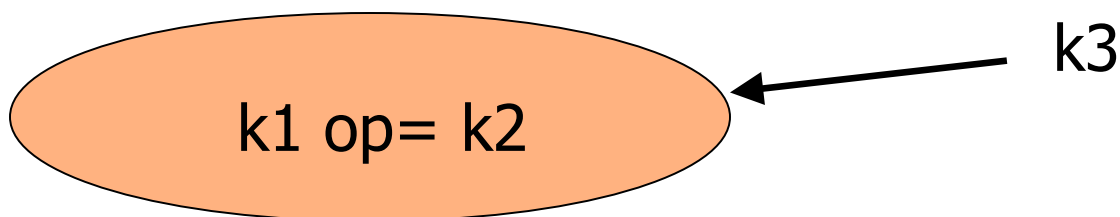
```
int *p;           // mutató
```

```
...
```

```
*p = 20;           // az int típusú szám értéke,  
                   // ahova a p mutat, 20 lesz.
```

„Művelet” és értékadás

- ▶ Az „op” tetszőleges bináris aritmetikai vagy bitenkénti operátor lehet.



- ▶ `k1`: módosítható balérték kell legyen.
- ▶ A kifejezés `k1 = k1 op (k2)` -vel egyenértékű.
- ▶ A `k3` értéke a `k1` megváltoztatott értékével (az átadott értékkel) egyezik meg.

5.7. Léptető operátorok

- ▶ Növelő operátor: ++
- ▶ Csökkentő operátor: --
- ▶ Használható előtagként és utótagként is.
- ▶ Előtagként: ++k1 vagy --k1.
- ▶ Utótagként: k1++ vagy k1--.
- ▶ A k1 módosítható balérték kell legyen.
- ▶ A ++ növeli a k1 értékét 1-el, a -- pedig csökkenti 1-el.

Más elnevezés



- ▶ növelő operátor = inkrementáló operátor
- ▶ csökkentő operátor = dekrementáló operátor
- ▶ postfix operátor = utótagként használt operátor
- ▶ prefix operátor = előtagként használt operátor

Léptetés előtaggal

- ▶ Ha a léptető operátort előtagként használjuk, akkor először módosítjuk a balértéket (k1-et), majd a kapott értéket használjuk a kifejezésben.
- ▶ Példa:

```
int x = 20;  
int y;  
y = ++x; // ugyanaz, mint y = (x+=1);  
          // a zárójel itt el is hagyható.
```
- ▶ Ekkor $x = 21$ és $y = 21$ lesz.

Léptetés utótaggal

- ▶ Ha a léptető operátort utótagként használjuk, akkor először használjuk a k1-et a kifejezésben, majd ezt követően módosítjuk az értékét.
- ▶ Példa:
 `int x = 20;`
 `int y, z;`
 `y = x++; // ugyanaz, mint y = (z=x, x+=1, z);`
 `// a zárójel nem hagyható el.`
- ▶ Ekkor `x = 21` és `y = 20` lesz.

5.8. A sizeof operátor

- ▶ Egy adatnak, illetve egy típusnak a méretét fejezi ki bájtokban.

- ▶ Alakja:

sizeof kifejezés,

- ▶ illetve

sizeof(típus).

5.9. A cím operátor és a referencia típus

- ▶ A cím operátor használata (C és C++):
 $\& k1$
- ▶ A $k1$ kifejezés módosítható balérték kell legyen.
- ▶ A kifejezés értéke a $k1$ -nek megfelelő adat memóriabeli címe.
- ▶ Ez annak a memóriaterületnek a kezdete, ahol a $k1$ tárolva van.
- ▶ A `scanf` függvény esetén gyakran használjuk.

A referencia típus (C++)

- ▶ Más néven: alternatív név, szinonima, álnév, hivatkozás, „alias”.
- ▶ A cím operátort (&) használjuk.
- ▶ Két változat:

típus & név = adat;
- ▶ vagy

típus & formális_paraméter
- ▶ A **típus &** egy új típus lesz (a referencia típus).

Példa referencia típusra

```
void main() {  
    int x[4] = {10, 20, 30, 40};  
    int& y = x[0]; // y és x[0] ugyanaz  
    int* z = x;    // *z és x[0] ugyanaz  
    y = 50;        // y, x[0] és *z módosul  
    *z = 60;       // y, x[0] és *z módosul  
}
```

5.10. A zárójel operátorok

- ▶ Függvényhívás:

kifejezés(kifejezés_lista)

- ▶ A () operátort a műveletek sorrendjének megváltoztatására is használhatjuk.

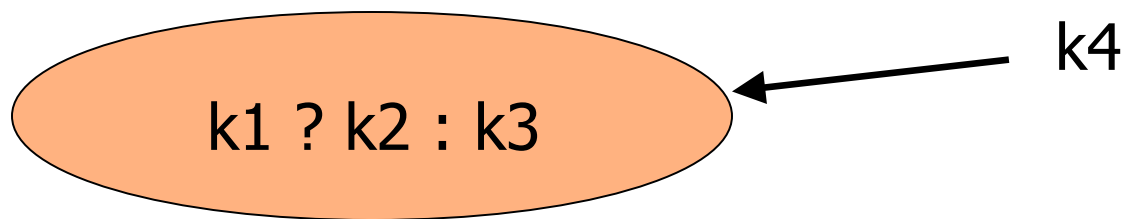
- ▶ Indexelés:

mutató[kifejezés]

- ▶ A tömb neve is egy mutatóként kezelhető.

5.11. A feltételes operátor

- ▶ Alakja:



- ▶ A `k1` egy feltétel. Ennek az értékétől függ a kifejezés értéke (`k4`).
- ▶ Ha a `k1` igaz (nem zéró, vagy `true` a C++-ban), akkor a `k4` értéke `k2`-vel, ellenkező esetben pedig `k3`-al fog megegyezni.

Példa feltételes operátorra

```
int a = 25;
```

```
int b = 34;
```

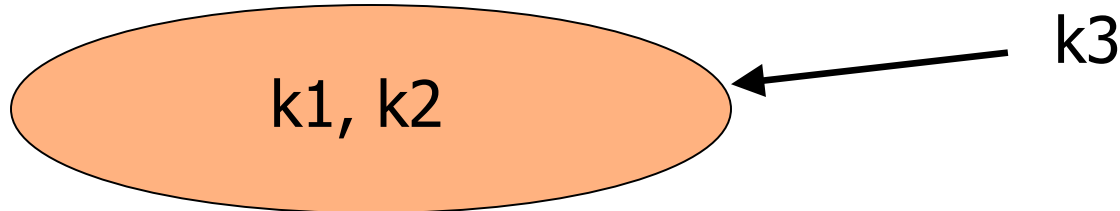
```
int m;
```

```
...
```

```
m = (a > b) ? a : b; // maximum
```

5.12. A vessző operátor

- ▶ Más név: műveletsor. Alakja:



- ▶ A $k3$ értéke és típusa megegyezik $k2$ -vel.
- ▶ Akkor használjuk, ha a szintaxis egyetlen kifejezést követel meg, de mi többet szeretnénk használni.

5.13. A hatókör operátor

- ▶ Más néven: hatókör feloldó operátor, érvényességi kör operátor.
- ▶ Lehetséges alakjai:

osztálynév :: tag

osztály tagja

névtérnév :: tag

névtér tagja

:: név

:: minősített_név

globális hatókör

Globális hatókör

- ▶ Egy globális névre hivatkozhatunk, ha az újra van deklarálva. Példa:

```
int x = 100;
void main() {
    char x = 'Z';
    cout << x << endl;    // helyi (karakter: 'Z' )
    cout << ::x << endl; // globális (egész: 100)
}
```

Más példa

```
#include <iostream>
using namespace std;
namespace X {
    int x_nev;
    namespace Y {
        int x_nev;
    }
}
double x_nev = 5.25;
```



A fő függvény

```
void main() {  
    char x_nev = 'A';  
    X::x_nev = 10; // az X névtérbeli  
    X::Y::x_nev = 20; // Y::x_nev minősített név  
    cout << x_nev << endl;    // helyi ('A')  
    cout << X::x_nev << endl; // 10  
    cout << ::x_nev << endl;  // globális (5.25)  
    cout << X::Y::x_nev << endl; // 20  
}
```

5.14. A típusazonosító operátor

- ▶ A typeid operátor:

typeid(típus_név)

vagy

typeid(kifejezés)

- ▶ Egy objektumot térít vissza, ami lehetővé teszi a típus futási időben történő meghatározását.

Futási idejű típusazonosítás

```
#include <iostream>
#include <typeinfo.h>
using namespace std;
void main() {
    cout << typeid( 97 ).name() << endl;
    cout << typeid( 97.0 ).name() << endl;
    cout << typeid( 97.0f ).name() << endl;
    cout << typeid( 'a' ).name() << endl;
    cout << typeid( static_cast<int>('a') ).name() << endl;
}
```

A kimenet



int

double

float

char

int

- ▶ A **typeinfo.h** fejláallományra is szükség van a typeid operátor használatához.

5.15. Precedencia és kiértékelési irány



- ▶ Precedencia (prioritás, a műveletek sorrendje)
- ▶ Kiértékelési irány („kötési” szabályok)

Precedencia



- ▶ A precedenciára vonatkozóan pontos szabályok léteznek.
- ▶ Egyes szabályok a matematikából ismertek. Például a bináris multiplikatív műveletek magasabb prioritással rendelkeznek, mint a bináris additív operátorok.
- ▶ A műveletek sorrendjét mindig meghatározhatjuk a zárójelek használatával.

Kiértékelési irány

- ▶ Általában balról jobbra történik a kiértékelés. Például: $x + y - z$ ugyanaz, mint $(x + y) - z$.
- ▶ Az unáris, illetve az értékadó operátorok jobbról balra lesznek kiértékelve.
- ▶ Például $*++p$ ugyanaz, mint $*(++p)$.
- ▶ Továbbá: $x=y=z$ ugyanaz, mint $x=(y=z)$.

6. Utasítások



6.1. Általános utasítások

6.2. Elágazások

- ▶ Az if utasítás
- ▶ A switch utasítás

6.3. Ciklusok

- ▶ A while utasítás
- ▶ A do while utasítás
- ▶ A for utasítás

6.4. Az exit függvény és a break utasítás

7. Mutatók

int *p;

7. Mutatók



7.1. Deklaráció

7.2. Kapcsolat a tömbökkel

7.3. Műveletek mutatókkal

7.4. A dinamikus memória kezelése

- ▶ C stílusú memória foglалás és felszabadítás
- ▶ Az elhelyező és felszabadító operátorok

7.5. Mutatókból álló tömbök és a parancssor paraméterei

7.1. Deklaráció



- ▶ A mutató (pointer) olyan változó, amely címeket tartalmaz.
- ▶ A mutatók segítségével olyan változók értékeire hivatkozhatunk, amelyeknek a címét ismerjük.
- ▶ Mutató típus: **típus ***
- ▶ Deklaráció:

típus * név;

A cím operátor

- ▶ Egy változó címét határozza meg. Példa:

```
int x = 10;
```

```
double y = 5.9;
```

```
int *p;
```

```
double *q;
```

```
p = &x; // p egy x-re hivatkozó mutató
```

```
p = &y; // hiba
```

```
q = &y; // helyes
```

Az indirekció operátor

- ▶ Más néven: „dereferencia”
- ▶ A * (unáris) operátort használjuk.

▶ Példa:

```
int x = 10;
```

```
int *p = &x;
```

```
*p = 50; // *p annak a változónak az értéke,  
        // amely az illető címen tárolva van.  
        // Ugyanaz mint: x = 50.
```


Void típusra hivatkozó mutatók



```
void *p;  
int x = -1;  
p = &x;  
int *q1 = (int *)p; // explicit típuskényszerítés  
printf("%d\n", *q1);  
unsigned *q2 = (unsigned *)p; // vigyázat!  
printf("%u\n", *q2);  
int z = *(int *)p;  
printf("%d\n", z);
```

A C++ változat



```
void *p;  
int x = -1;  
p = &x;  
int *q1 = static_cast<int *>(p);  
cout << *q1 << endl;  
unsigned *q2 = static_cast<unsigned *>(p);  
cout << *q2 << endl;  
int z = *static_cast<int *>(p);  
cout << z << endl;
```

Az eredmény



- ▶ Visual C++ .NET fordítóval mindkét esetben:
-1
4294967295
-1
- ▶ A $4294967295 = 2^{32} - 1$ szám a maximális unsigned.

Típuskényszerítés

```
int *p;    // most nem void * típus lesz!  
int x = -1;  
p = &x;  
unsigned *r1 = (unsigned *)p;  
cout << *r1 << endl;  
unsigned *r2 = static_cast<unsigned *>(p); //hiba  
...  
unsigned *r3 = reinterpret_cast<unsigned *>(p);  
cout << *r3 << endl;
```

7.2. Kapcsolat a tömbökkel

- ▶ A tömb neve úgy tekinthető, mint egy mutató az első elemre.
- ▶ Példa:

```
int t[50];  
int *p;  
p = t; // helyes  
t = p; // fordítási hiba
```
- ▶ A tömb neve egy állandó pointerként kezelhető.

Tömb típusú paraméter

- ▶ Tekintsük a

```
void f(double t[]);
```

- ▶ függvénydeklarációt. Ez ugyanaz, mint:

```
void f(double *t);
```

- ▶ Meghívás:

```
double z[3]={1.1, 2.2, 3.3};
```

```
f(z);
```

- ▶ Ebben az esetben a tömbnév nem állandó mutató.

7.3. Műveletek mutatókkal

- ▶ Növelés és csökkentés
- ▶ Egész számra és mutatóra alkalmazott művelet
 - ▶ Egész szám hozzáadása mutatóhoz
 - ▶ Egész szám kivonása mutatóból
- ▶ Mutatók összehasonlítása
- ▶ Mutatók különbsége

Növelés és csökkentés

- ▶ Legyen: `típus *m;`
- ▶ Növelés: `++m` vagy `m++`.
- ▶ Csökkentés: `--m` vagy `m--`.
- ▶ Mindig `sizeof(típus)`-al módosul a megfelelő cím, tehát az `m` értéke.
- ▶ Ha az `m` egy tömb egyik eleméhez mutat, akkor a következő, illetve az előző elemre hivatkozó mutatót kapjuk.

Példa léptető operátorra

```
int t[5]={11, 22, 33, 44, 55};  
int *m = t;  
printf("cim: %p\tertek: %d\n", m, *m);  
m++;  
printf("cim: %p\tertek: %d\n", m, *m);
```

► Lehetséges kimenet (Visual C++ .NET):

cim: 0066FDE4 érték: 11

cim: 0066FDE8 érték: 22

Egész számra és mutatóra alkalmazott művelet

- ▶ Legyen:

típus *m;

int x; // vagy más egész típus

- ▶ Akkor

m + x : m értéke növelve $x * \text{sizeof}(\text{típus})$ -al.

m - x : m értéke csökkentve $x * \text{sizeof}(\text{típus})$ -al.

- ▶ Ha t egy tömb neve, akkor **t[i]** megegyezik ***(t+i)** -vel.

Példa hozzáadásra és kivonásra

```
int t[5]={11, 22, 33, 44, 55};  
int *m = t;  
int *q;  
q = m + 2;    // 33  
q = 1 + q;    // 44  
q = -1 + q;   // 33  
q = q - 2;    // 11  
q += 3;       // 44  
q -= 3;       // 11
```

Mutatók összehasonlítása

- ▶ Akkor végezhető el, ha a két mutató ugyanannak a tömbnek az elemeire hivatkozik.
- ▶ Ha a p pointer $t[u]$ -ra mutat, és a q pointer $t[v]$ -re mutat, akkor:

$p \text{ op } q$ ugyanaz, mint $u \text{ op } v$,

- ▶ ahol op tetszőleges összehasonlító operátor lehet.
- ▶ op : $<$ $<=$ $>=$ $>$ $==$ $!=$

Mutatók különbsége

- ▶ Ebben az esetben is a két mutató ugyanannak a tömbnek az elemeire kell hivatkozzon.
- ▶ Ha a p pointer $t[u]$ -ra mutat, és a q pointer $t[v]$ -re mutat, akkor:
 $p - q$ ugyanaz, mint $u - v$.

Példa mutatók különbségére

```
int t[5]={11, 22, 33, 44, 55};
```

```
int *p = t+1;
```

```
int *q = t+4;
```

```
printf("%d\n", q-p);    // 3
```

```
printf("%d\n", p-q);    // -3
```

7.4. A dinamikus memória kezelése



- ▶ Más néven: dinamikus memóriakiosztás, memóriaterület lefoglalása és felszabadítása.
- ▶ C stílusú memórafoglalás és felszabadítás (szabványos függvényekkel)
- ▶ Az elhelyező és felszabadító operátorok (C++)

C stílusú memóriafoglalás és felszabadítás



- ▶ Lefoglalás
 - ▶ A malloc függvény
 - ▶ A calloc függvény
- ▶ Felszabadítás
 - ▶ A free függvény

Az malloc és calloc függvények

▶ Példa:

```
double *m;
```

```
int x=20;
```

```
m = (double *)malloc(x*sizeof(double));
```

▶ Húsz valós számára foglal le memóriaterületet.

▶ Ugyanaz, mint:


```
m = (double *)calloc(x, sizeof(double));
```

▶ Típuskonverzióra van szükség, mivel a visszatérített típus void *.

A free függvény

- ▶ Az előző példa esetén a:
`free(m);`
- ▶ felszabadítja a lefoglalt memóriaterületet.
- ▶ A free függvénynek void * típusú a formális paramétere, de az m automatikusan erre a típusra lesz konvertálva.

Az elhelyező és felszabadító operátorok (C++)



- ▶ Lefoglalás
 - ▶ A new operátor (elhelyező operátor)
- ▶ Felszabadítás
 - ▶ A delete operátor (felszabadító operátor)

A new operátor



- ▶ Három változat:
 - a) new típus
 - b) new típus(kifejezés1)
 - c) new típus[kifejezés1]
- ▶ A b) esetben a memóriaterület inicializálva lesz kifejezés1-el.
- ▶ Az a) és b) esetben sizeof(típus) byte számára, a c) esetben pedig kifejezés1*sizeof(típus) byte számára foglal le memóriát.

Példa a new operátorra

- ▶ Az malloc függvényre adott példa így írható:

```
double *m;  
int x=20;  
m = new double[x];
```

- ▶ Továbbá

```
int *p = new int(70);
```

- ▶ ugyanaz, mint:

```
int *p = (int *)malloc(sizeof(int));  
*p = 70;
```

A delete operátor

- ▶ Ha a lefoglalás a new operátorral az a) vagy a b) módon történt, akkor a
delete m
- ▶ ha a c) módon, akkor a
delete [kifejezés1] m
- ▶ szabadítja fel a memóriát, ahol m a lefoglalt memóriaterület kezdetére mutat.
- ▶ A kifejezés1 általában elmaradhat, de a szögletes zárójel jelen kell legyen.

7.5. Mutatókból álló tömbök és a parancssor paraméterei

- ▶ Mutatókból álló tömbök (a tömb elemei pointerek).
- ▶ A parancssor paraméterei (a fő függvénynek átadott paraméterek).
- ▶ Dos: például legyen a végrehajtható állomány neve: `cprogr.exe`). Ekkor
C:\Temp>cprogr param1 param2 ...
- ▶ Windows: Start->Run ablakban.

Mutatókból álló tömbök

▶ Példa:

```
char *mese[] = {  
    "Piroska",  
    "farkas",  
    "nagymama"  
};
```

```
printf("%s\n", mese[0]); // Piroska
```

- ▶ A tömb karakterlánc-literálokra hivatkozó mutatókból áll.

A parancssor paraméterei

- ▶ A fő függvény fejléce:

```
int main(int argc, char *argv[])
```

- ▶ A visszatérített érték típusa void is lehet.
- ▶ A formális paraméterek jelentése:
- ▶ `argc` = a parancssor paramétereinek a száma + 1;
- ▶ `argv[0]` = mutató a végrehajtható állomány nevéhez (az útvonalat is tartalmazza).
- ▶ `argv[i]` = egy karakterlánc-literálhoz mutat, amely az i -edik ($1 \leq i \leq \text{argc}-1$) paramétert tartalmazza.

8. Függvények



void fct(int x);

8. Függvények



8.1. Deklaráció és definíció

8.2. Függvények meghívása és a visszatérített érték

8.3. Paraméterátadás

8.4. A formális paraméterek kezdeti értéke

8.5. Függvényekre hivatkozó mutatók

8.6. Referencia típust visszaadó függvények

8.7. Túlterhelés

8.8. Inline függvények

8.1. Deklaráció és definíció



- ▶ Függvénydefiníció
- ▶ Függvénydeklaráció (függvény-meghatározás)
- ▶ Példák

Függvénydefiníció

- ▶ Definíció – teljes szerkezet megadása.

típus név(formális paraméterek listája)

{

 deklarációk

 utasítások

}

Függvények osztályozása a visszatérített típus szerint

- ▶ Két féle függvény:
- ▶ visszaad egy értéket
 - ▶ "típus" a visszaadott érték típusa;
 - ▶ algoritmikus nyelvbeli "függvény";
 - ▶ nem void típust visszaadó függvény (nem void függvény);
- ▶ nem ad vissza értéket
 - ▶ a "típus" void lesz;
 - ▶ algoritmikus nyelvbeli "eljárás";
 - ▶ void függvény.

A definíció két része

- ▶ Fejléc: **típus név(formális paraméterek listája)**
- ▶ A függvény belseje (a függvény teste, a függvény törzse, blokk):

```
{  
    deklarációk  
    utasítások  
}
```
- ▶ A formális paraméterek listája: változódeklarációk, vesszővel elválasztva. Lehet üres is, de a zárójelek ekkor is jelen kell legyenek.

Függvénydeklaráció



- ▶ Deklaráció:
típus név(formális paraméterek listája);
- ▶ Tehát: fejléc;
- ▶ Ezt még prototípusnak is nevezzük.
- ▶ Deklaráláskor a formális paraméterek nevei elmaradhatnak (a típusoknak mindenképpen jelen kell lenniük).

Példák függvénydeklarációkra

- ▶ A formális paraméterek listája üres:

`void f();`

- ▶ Két paraméter (egy egész és egy valós):

`void g(int x, double y);`

- ▶ Ugyanaz mint:

`void g(int, double);`

- ▶ Tömb típusú paraméter:

`void h(int t[]);`

- ▶ Ekkor, az első index felső határa elmaradhat.

8.2. Függvények meghívása és a visszatérített érték



- ▶ Függvénymeghívás
- ▶ Visszatérés a függvényből

Függvénymeghívás

- ▶ Különálló utasításként:
 név(aktuális paraméterek listája);
- ▶ Kifejezésben, mint operandus:
 név(aktuális paraméterek listája)
- ▶ Ekkor a függvény vissza kell térítsen egy értéket.
 Ezt használjuk a kifejezésben.
- ▶ A formális és aktuális paraméterek közti
 megfeleltetést ellenőrzi a fordító. Ha ezek a típusok
 nem egyeznek meg, akkor a megfelelő formális
 paraméter típusára konvertál.

Példa függvénymeghívásra

- ▶ Függvény deklaráció (karakterlánc hossza):

```
int hossz(char s[]); //dimenzió elmaradhat
```

- ▶ Fő függvényben:

```
char u[]="karakter"; //dimenzió elmaradhat
```

```
char v[10]="abc";
```

```
printf("%d\n", hossz(u)); //eredmény: 8
```

```
printf("%d\n", hossz(v)); //eredmény: 3
```

Visszatérés a függvényből

- ▶ Három lehetőség:
 - (1) `return;`
 - (2) `return` kifejezés;
 - (3) a függvény belsejében lévő utolsó utasítás is végre volt hajtva.
- ▶ Az (1) és (3) `void` függvény esetén használható.
- ▶ Az (1) esetén a `return` utasításnak általában egy elágazó utasítás (`if`, `switch`) egyik ágán kell megjelenni.

A visszaadott érték megadása

- ▶ A void függvények nem adnak vissza értéket.
- ▶ A nem void-ként deklarált függvényeknek értéket kell visszaadni.
 - ▶ C++-ban ez kötelező, kivétel a main, amit lehet nem void függvényként deklarálni, de nem kell feltétlenül megadni a visszatérített értéket.
 - ▶ C-ben, és régebbi C++ fordítók esetén tetszőleges függvényre csak figyelmeztető üzenet jelenik meg.

Visszatérési érték



- ▶ return kifejezés;
- ▶ Akkor használható, ha a függvény nem void típust térít vissza. Kivétel C++-ban az az eset, amikor a kifejezés egy másik void függvény meghívása (Visual C++ 6.0-ban még nem működik).
- ▶ A kifejezés értékét adja vissza.
- ▶ Ha a kifejezés típusa különbözik a fejlécben megadott típustól, akkor a fordító a deklarációban megadott típusra konvertálja.

8.3. Paraméterátadás



- ▶ Példa
- ▶ Paraméterátadás a C-ben
- ▶ Cím szerinti paraméterátadás megvalósítása a C-ben
 - ▶ Tömbökkel
 - ▶ Mutatókkal
- ▶ Cím szerinti paraméterátadás a C++-ban (tulajdonképpen cím szerinti paraméterátadás)

A „cserél” függvény

```
void cserel(int a, int b)
{
    int x;
    x = a;
    a = b;
    b = x;
}
```

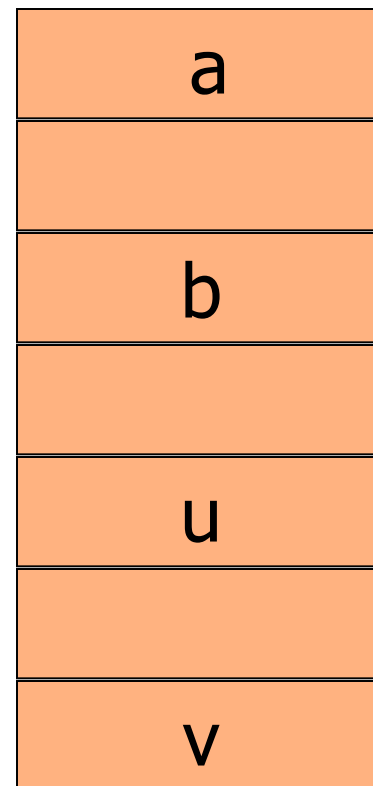
- ▶ Kérdés: valóban cserét fog végezni?

A fő függvény

```
void main()
{
    int u = 3;
    int v = 5;
    cserel(u, v);
    printf("u=%d\tv=%d\n", u, v);
}
```

- ▶ Kimenet: u=3 v=5
- ▶ Tehát nem történt csere.

▶ Tárolás:



Lehetséges megoldások



- ▶ Megváltoztatjuk a függvény nevét: „cserél” helyett „nemcserél”.
- ▶ C-ben:
 - ▶ mutatókat vagy tömböket használunk.
- ▶ C++-ban:
 - ▶ referencia típus segítségével valósítunk meg tulajdonképpeni cím szerinti paraméterátadást.

Paraméterátadás a C-ben



- ▶ Mindig érték szerinti.
- ▶ Az aktuális és formális paraméterek különböző helyeken vannak tárolva a memóriában.
- ▶ A formális paraméterek megváltozása nincs hatással az aktuális paraméterekre.
- ▶ A függvény meghívásakor az aktuális paraméterek átadják az értéküket a formális paramétereknek, és ez az értékadás egyirányú.

Cím szerinti paraméterátadás megvalósítása tömbökkel

```
void cserel_tomb(int t[])
{
    int x;
    x = t[0];
    t[0] = t[1];
    t[1] = x;
}
```

- ▶ A függvény kicseréli egymás közt az aktuális paraméterként megadott tömb első két elemét.

Tömb paraméterek

- ▶ Fő függvényben:

```
int m[2];
```

```
m[0] = 3;
```

```
m[1] = 5;
```

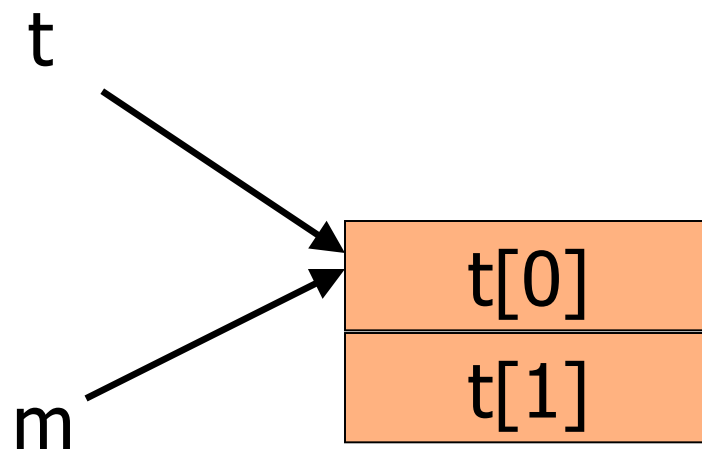
```
cserel_tomb(m);
```

- ▶ Valóban csere történik.

- ▶ Indoklás:

- ▶ t formális paraméter,

- ▶ m aktuális paraméter.



Cím szerinti paraméterátadás megvalósítása mutatókkal

```
void cserel_p(int *p, int *q) // pointerekkel
{
    int x;
    x = *p;
    *p = *q;
    *q = x;
}
```

- ▶ Az aktuális paraméter egy cím kell legyen.

Mutató típusú paraméterek

- ▶ Fő függvényben:

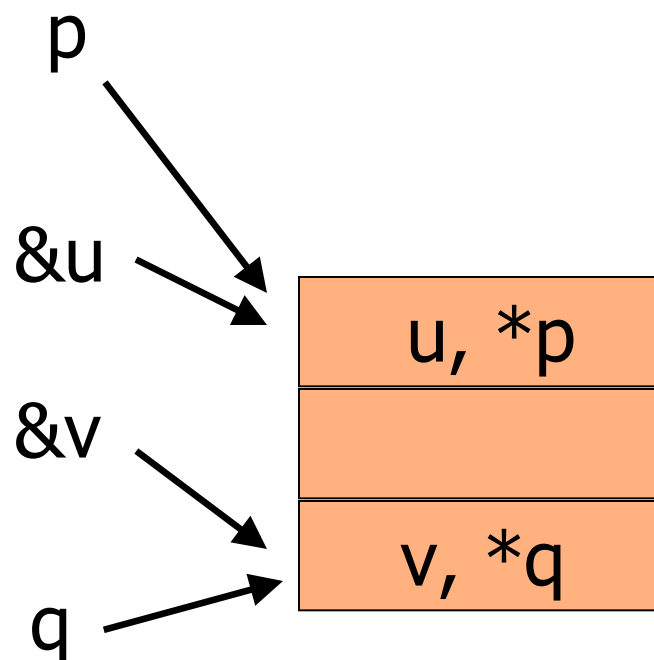
```
int u = 3;
```

```
int v = 5;
```

```
cserel_p(&u, &v);
```

- ▶ Valóban csere történik.
- ▶ p, q formális paraméter
- ▶ &u, &v (u és v címe) aktuális paraméter.

- ▶ Tárolás:



Cím szerinti paraméterátadás a C++-ban

- ▶ Referencia típus (alternatív név, szinonima, álnév, hivatkozás, „alias”)
típus & formális_paraméter
- ▶ Ebben az esetben az aktuális paraméter egy alternatív neve a formális paraméternek.
- ▶ Az aktuális és formális paraméter pontosan ugyanazt a memóriaterületet fogja jelenteni.
- ▶ Ezt használjuk a cím szerinti paraméterátadás esetén.

Referencia típus használata

```
void cserel_r(int& a, int& b) //referencia
{
    int x;
    x = a;
    a = b;
    b = x;
}
```

- ▶ Ez csak abban különbözik a „cserel” függvényétől, hogy itt használjuk a referencia típust.

Referencia típusú paraméterek

- ▶ Fő függvényben:

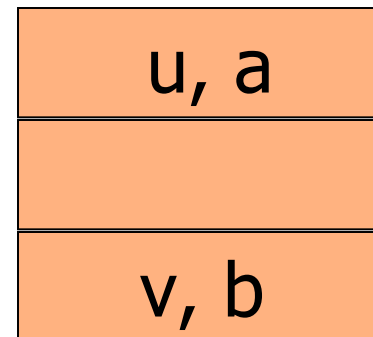
```
int u = 3;
```

```
int v = 5;
```

```
cserel_r(u, v);
```

- ▶ Valóban csere történik.
- ▶ a, b formális paraméter
- ▶ u, v aktuális paraméter.

- ▶ Tárolás:



8.4. A formális paraméterek kezdeti értéke

- ▶ Az inicializált paramétereket alapértelmezett paramétereknek is nevezzük.
- ▶ A formális paramétereknek egy alapértelmezés szerinti értéket adhatunk.
- ▶ A formális paraméterek listájában a
típus név = kifejezés
- ▶ alakot használjuk.
- ▶ Az alapértelmezett paraméterek a nem inicializáltak után kell elhelyezkedjenek a paraméterlistában.

Példa alapértelmezett paraméterekre

```
#include <iostream>
#include <cmath>
using namespace std;
int kerulet(int x2, int y2, int x1 = 0, int y1 = 0)
{
    return 2*(abs(x2-x1)+abs(y2-y1));
}
```

- ▶ A függvény egy téglalap kerületét számolja ki. Az egyik csúcs alapértelmezés szerint az origó.

A fő függvény

```
void main() {  
    cout << kerulet(7,4) << endl;    // 22  
    cout << kerulet(7,4,3) << endl;  // 16  
    cout << kerulet(7,4,3,1) << endl; // 14  
    cout << kerulet(-7, -4) << endl; // 22  
}
```

8.5. Függvényekre hivatkozó mutatók

- ▶ Egy függvény címét átadhatjuk egy olyan mutatónak, amely a függvényre hivatkozik.
- ▶ Például:

```
double f_negyzet(double x) { return x*x; }  
void main() {  
    double (*g)(double); // mutató egy függvényre  
    g = &sin;           // a math.h fejlécből  
    g = f_negyzet; // a cím operátor (&) elmaradhat  
}
```

Függvény típusú paraméterek

- ▶ Példa (kiírás C++-ban):
 - ▶ A kiir_f függvény
 - ▶ A kiir2_f függvény
 - ▶ A deklarációk
 - ▶ A meghívás
 - ▶ Az eredmény

A kiir_f függvény

```
void kiir_f( double (*pf)(double), double a)
{
    cout << "a = " << a;
    cout << "\tf(a) = " << (*pf)(a) << endl;
    // cout << "a = " << a;
    // cout << "\tf(a) = " << pf(a) << endl;
}
```

- ▶ A meghíváskor az indirekció operátor (*) elhagyható.

A kiir2_f függvény

```
void kiir2_f( double f(double), double a)
{
    cout << "a = " << a;
    cout << "\tf(a) = " << f(a) << endl;
    // cout << "a = " << a;
    // cout << "\tf(a) = " << (*f)(a) << endl;
}
```

- ▶ A formális paraméter egy függvénynév is lehet.

A deklarációk

- ▶ A formális paraméter neve elmaradhat. Tehát
`void kiir_f(double (*pf)(double), double a);`
- ▶ ugyanaz, mint
`void kiir_f(double (*)(double), double);`
- ▶ Ha a formális paraméter függvénynév, akkor nem hagyható el:
`void kiir2_f(double f(double), double);`

A meghívás



```
void main() {  
    double (*g)(double);  
    kiir_f(f_negyzet, 1.5);  
    kiir_f(sin, 3.1415 / 2);  
    kiir2_f(f_negyzet, 1.1);  
    kiir2_f(sin, 0);  
    g = f_negyzet;  
    kiir_f(g, 2);  
}
```

Az eredmény



$$a = 1.5 \quad f(a) = 2.25$$

$$a = 1.57075 \quad f(a) = 1$$

$$a = 1.1 \quad f(a) = 1.21$$

$$a = 0 \quad f(a) = 0$$

$$a = 2 \quad f(a) = 4$$

8.6. Referencia típust visszaadó függvények

- ▶ A visszaadott érték referencia típus lesz, tehát a függvény fejléce a következő:
típus & név(formális paraméterek)
- ▶ A referencia típust vissztérítő függvények meghívása egy balérték (lvalue).
- ▶ A függvénymeghívás szerepelhet értékadások bal oldalán.

Példa referencia típust visszaadó függvényre

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int& fekete_piros( int& r_fekete, int& r_piros)
{
    if ( rand() % 2 )
        return r_fekete;
    return r_piros;
}
```

A fő függvény

```
void main() {  
    int fekete = 0; int piros = 0; int szam;  
    cout << "Golyok szama: "; cin >> szam;  
    srand( (unsigned)time( NULL ) );  
    for( int i = 0; i < szam; i++ )  
        fekete_piros( fekete, piros )++;  
    cout << "Fekete golyok: " << fekete  
        << "\nPiros golyok: " << piros;  
}
```


A rand és srand függvények

- ▶ A rand függvény a [0, RAND_MAX] intervallumba eső véletlenszámot generál.
- ▶ A RAND_MAX értéke a Visual C++ .NET-ben a cstdlib állományban van megadva:

```
#define RAND_MAX 0x7fff
```
- ▶ Tehát a RAND_MAX értéke 32767.
- ▶ A generátor inicializálását az srand függvénnyel végezzük, melynek egy unsigned típusú paramétere van. Ezt az aktuális rendszeridő függvényében határoztuk meg.

8.7. Túlterhelés



- ▶ A C-ben két különböző függvénynek nem lehet ugyanaz a neve.
- ▶ A C++-ban a függvények túlterhelhetők, tehát különböző függvényeknek lehet azonos a neve.
- ▶ Ebben az esetben a paraméterlisták kell különbözzenek.
- ▶ Pontos szabályok léteznek arra vonatkozóan, hogy az azonos nevű függvények közül hogyan választja ki a rendszer azt, amit meg fog hívni.

Túlterhelt függvény meghívása

► Négy lépés:

1. Ha egy függvény esetén a formális paraméterek típusa megegyezik az aktuális paraméterek típusával, akkor ezt hívja meg a rendszer.
2. Máskülönben, a függvényt a standard típusokra alkalmazott alapértelmezés szerinti konverzióval próbálja meghatározni a rendszer (például az **int** típust **double** típusra alakítja). Ebben az esetben nem történhet információvesztés.

Túlterhelt függvény meghívása



3. Ha így sem sikerült meghatározni a függvényt, akkor a standard típusokra alkalmazható más konverzióval próbálkozik a rendszer. Ez a konverzió esetleg információvesztéssel járhat (például a **double** típus **int** típusra alakítása).
4. Ha ebben az esetben sem sikerült meghatározni a függvényt akkor a programozó által definiált típusokra is konverziót próbál alkalmazni a rendszer.

Példa a tangens függvény inverzére

- ▶ A C-ben a math.h fejlécfájlban a következő függvények vannak deklarálva:

```
double atan(double x);
```

```
double atan2(double y, double x);
```

```
long double atanl(long double x);
```

```
long double atan2l(long double y, long double x);
```

```
float atanf(float x);
```

```
float atan2f(float y, float x);
```

A fő függvény



```
#include <iostream>
#include <cmath>
using namespace std;
void main() {
    // ...
}
```

- ▶ A **cmath** fejlálcím a **math.h** megfelelője az **std** névtérben.

A ✓kiszámítása

```
void main() {  
    cout.precision(17);    // 17 számjegyet ír ki  
    cout << 4 * atan( 1.0 ) << endl;  
    cout << 4 * atanl( 1 ) << endl;  
    cout << 4 * atanf( 1 ) << endl;  
    cout << atan2(0.0, -1.0) << endl;  
    cout << atan2l(0, -1) << endl;  
    cout << atan2f(0, -1) << endl;  
}
```

A kimenet



3.1415926535897931

3.1415926535897931

3.1415927410125732

float!

3.1415926535897931

3.1415926535897931

3.1415927410125732

float!

Futási idejű típusazonosítás

```
void main() {  
    cout << typeid( atan( 1.0 )).name() << endl;  
    cout << typeid( atanl( 1 )).name() << endl;  
    cout << typeid( atanf( 1 )).name() << endl;  
    cout << typeid(atan2(0.0, -1.0)).name() << endl;  
    cout << typeid(atan2l(0, -1)).name() << endl;  
    cout << typeid(atan2f(0, -1)).name() << endl;  
}
```

- ▶ Ha nem használjuk az **std** névtért, akkor a **typeinfo.h** fejlécállományra van szükség a typeid operátor használatához.

A kimenet



double

long double

float

double

long double

float

A függvénynevek túlterhelése

- ▶ Az atan függvényt úgy terheljük túl, hogy ugyanazt végezze mint az:
- ▶ atan2
- ▶ atanl
- ▶ atan2l
- ▶ atanf
- ▶ atan2f

Az atan2 függvény túlterhelt változata



```
double atan(double y, double x) {  
    return atan2(y, x);  
}
```

Az atan1 és atan2l függvények túlterhelt változatai

```
long double atan(long double x) {  
    return atanl(x);  
}
```

```
long double atan(long double y, long double x)  
{  
    return atan2l(y, x);  
}
```

Az atanf és atan2f függvények túlterhelt változatai



```
float atan(float x) {  
    return atanf(x);  
}
```

```
float atan(float y, float x) {  
    return atan2f(y, x);  
}
```

A atan függvény meghívása

```
void main() {  
    cout.precision(17); // 17 számjegyet ír ki  
    cout << 4 * atan( 1.0 ) << endl;  
    cout << 4 * atan( 1.0L ) << endl;  
    cout << 4 * atan( 1.0f ) << endl;  
    cout << atan(0.0, -1.0) << endl;  
    cout << atan(0.0L, -1.0L) << endl;  
    cout << atan(0.0f, -1.0f) << endl;  
}
```

A kimenet



3.1415926535897931

3.1415926535897931

3.1415927410125732

float!

3.1415926535897931

3.1415926535897931

3.1415927410125732

float!

Futási idejű típusazonosítás

```
void main() {  
    cout << typeid( atan( 1.0 )).name() << endl;  
    cout << typeid( atan( 1.0L )).name() << endl;  
    cout << typeid( atan( 1.0f )).name() << endl;  
    cout << typeid(atan(0.0, -1.0)).name() << endl;  
    cout << typeid(atan(0.0L, -1.0L)).name() << endl;  
    cout << typeid(atan(0.0f, -1.0f)).name() << endl;  
}
```

A kimenet



double

long double

float

double

long double

float

A „valarray” használata

```
// ...  
#include <valarray>  
void main() {  
    valarray<double> v_tomb(3);  
    valarray<double> eredmeny;  
    v_tomb[0] = 1;  
    v_tomb[1] = sqrt(3);  
    v_tomb[2] = sqrt(3) / 3;  
    // ...  
}
```

Az atan függvény „valarray”-re



```
void main() {  
    // ...  
    eredmeny = atan(v_tomb);  
    int t[] = {4, 3, 6};  
    for (int i = 0; i < 3; i++)  
        cout << t[i] * eredmeny[i] << endl;  
    cout << typeid( eredmeny ).name() << endl;  
}
```

Az eredmény



3.1415926535897931

3.1415926535897931

3.1415926535897931

class std::valarray<double>

8.8. Inline függvények

- ▶ Más néven: helyben kifejtett függvény
- ▶ Ha lehet, a függvénymeghívást a függvény törzsével helyettesíti, ugyanúgy mint a C-beli makró.
- ▶ Az **inline** minősítőt kell használni a függvény fejlécében. Például:

```
inline void f() {  
    // ...  
}
```

Példa makróra



```
#include <iostream>
using namespace std;
#define absz(x)      ((x) > 0 ? (x) : -(x))
#define absz1(x)     ( x > 0 ? x : -x )
#define absz2(x)     (x) > 0 ? (x) : -(x)
```

A fő függvény

```
void main() { int y = 10;
    cout << absz( y ) << endl;           // 10
    cout << absz(2 * y - 25 ) << endl;    // 5
    cout << absz1(2 * y - 25 ) << endl;    // -45
    cout << 14 + absz( -y ) << endl;      // 24
    cout << (14 + absz2( -y )) << endl;    // -10
    cout << absz( y++ ) << endl;          // 11
    cout << "y = " << y << endl;         // y = 12
}
```


Példa inline függvényre



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

```
inline int absz_ert( int x )  
{  
    return x > 0 ? x : -x;  
}
```

A fő függvény

```
void main() {  
    int y = 10;  
    cout << absz_ert( y ) << endl;           // 10  
    cout << absz_ert(2 * y - 25 ) << endl;    // 5  
    cout << 14 + absz_ert( -y ) << endl;      // 24  
    cout << absz_ert( y++ ) << endl;          // 10  
    cout << "y = " << y << endl;             // y=11  
}
```

Példa függvénysablonra

```
#include <iostream>
using namespace std;
template<class T>
inline T absz_ert( T x )
{
    return x > 0 ? x : -x;
}
```

Az absz_ert függvény meghívása

```
void main() {  
    cout << absz_ert( -11 ) << endl;           // 11  
    cout << absz_ert( -4.3 ) << endl;          // 4.3  
    signed char w = '\x82';  
    cout << static_cast<int>(w) << endl;        // -126  
    cout << absz_ert( w ) << endl;             // ~  
    unsigned char t = '\x82';  
    cout << static_cast<int>(t) << endl;        // 130  
    cout << absz_ert( t ) << endl;             // é  
}
```

9. Struktúrák és típusok

typedef int egész;
struct könyv A;

9. Struktúrák és típusok



9.1. Struktúradeklaráció és hivatkozás az adattagokra

9.2. Típusdeklarációk

9.3. Uniók

9.4. Bitmezők

9.5. A felsoroló típus

9.6. Önhivatkozó struktúrák

9.1. Struktúradeklaráció és hivatkozás az adattagokra

- ▶ Általános alakja:

```
struct név {  
    deklarációk  
} nevek_listája;
```

- ▶ A „nevek_listája” és a „név” is elmaradhat, de nem egyszerre.
- ▶ A „nevek_listája” struktúra típusú változónevekből áll. Vesszővel választjuk el őket.
- ▶ A „név” a struktúra neve.

Példa struktúrára

```
struct konyv {  
    char szerzo[30];  
    char cim[50];  
    char kiado[30];  
    int oldalszam;  
} x, y;
```



adattagok

- ▶ Az x és y struktúra típusú változó lesz.

A struktúra típus

- ▶ A C-ben a **struct név** egy típusnév.
- ▶ A C++-ban a **név** önállóan is használható (a struct kulcsszó nélkül).
- ▶ Példa (C-ben):

```
struct konyv u;  
struct konyv *p;
```

- ▶ Példa (C++-ban):

```
konyv v;  
konyv *q;
```

Az adattagokra való hivatkozás

- ▶ Tagkiválasztó operátor: `változónév.tag`.

- ▶ Példa:

```
printf("%s", u.szerzo);
```

- ▶ Struktúra-mutató operátor: `változónév->tag`.
Ugyanaz mint: `(*változónév).tag`.

- ▶ Példa:

```
printf("%d\n", p->oldalszam);  
printf("%d\n", (*p).oldalszam);
```

9.2. Típusdeklarációk

- ▶ Alakja:

```
typedef típus saját_típus_név;
```

- ▶ A „típus” egy létező típus neve kell legyen.
- ▶ A „saját _típus_név” ezt követően a „típus” helyett használható.
- ▶ Példa:

```
typedef float valos;
```

```
...
```

```
valos x, y;
```

A struktúra és a typedef

```
typedef struct konyv {
```

```
    ...
```

```
} KONYV;
```

```
...
```

```
KONYV a, b;
```

- ▶ Ennek csak a C-ben van értelme.
- ▶ A C++-ban a „konyv” egy önálló típusnév.

Más példák



```
typedef int *pint;  
typedef double (*mf)(double);  
pint p; // int típusra hivatkozó mutató  
int x = 20;  
p = &x;  
mf g; // mutató egy függvényhez  
g = sin;
```

9.3. Uniók



- ▶ Más néven: union típusok.
- ▶ Ugyanazon a memóriaterületen különböző típusú változók tárolhatók.
- ▶ A struktúrákhoz hasonlóan deklaráljuk, de a **struct** helyett az **union** kulcsszót használjuk.
- ▶ Az uniók tagjai közül egyszerre csak egy használható.
- ▶ A C++-ban az unió utáni név önálló típusnév, a C-ben a „union” kulcsszót is kell használni.

9.4. Bitmezők



- ▶ Egy struktúra tagjai
típus mezőnév: hosszúság_bitekben;
- ▶ alakúak lehetnek. Ekkor a mezőnév annyi bitre vonatkozik, amennyit a hosszúságban megadtunk.
- ▶ A mezőnév elmaradhat, ha bizonyos bitekre nem akarunk hivatkozni.
- ▶ A típus általában unsigned, de lehet int, bool, unsigned char vagy char is.

Példa unióra és bitmezőre

```
union {  
    int x;  
    struct {  
        unsigned b1 : 8;  
        unsigned b2 : 8;  
        unsigned b3 : 8;  
        unsigned b4 : 8;  
    } y;  
} u;
```

u.x egy egész

u.y.b1 az első byte

A fő függvény

```
void main() {  
    cout << "i =";  
    cin >> u.x;  
    cout << u.x << endl;  
    cout << u.y.b1 << endl;  
    cout << u.y.b2 << endl;  
    cout << u.y.b3 << endl;  
    cout << u.y.b4 << endl;  
}
```

Visual C++-ban az int típusú számot alkotó négy byte értékét írja ki.

Az int típus maximális értéke C++-ban

```
#include <iostream>
#include <limits>
using namespace std; // névtér
// ... union
void main() {
    u.x = numeric_limits<int>::max();
    // ... kiírás
}
```

9.5. A felsoroló típus

- ▶ Más néven: felsorolás típus. Alakja:
enum név {nevek_listája} változónevek_listája;
- ▶ Az egész számokhoz jellemző neveket rendel.
- ▶ Az első név zéró lesz, ha másképpen meg nem adjuk.
- ▶ A többi névhez a következő számokat rendeljük növekvő sorrendben.
- ▶ Ha más értéket akarunk, akkor a **név = k1** alakot használjuk, ahol **k1** egy konstans kifejezés.

Példa felsoroló típusra

```
enum torpe {Hofeherke, Tudor, Vidor, Hapci,  
Szende, Szundi, Morgo, Kuka};
```

- ▶ A C-ben az **enum torpe** egy típus, a C++-ban a **torpe** önmagában is használható.

```
enum torpe t;
```

```
t = 5; // helyes a C-ben, hibás a C++-ban.
```

```
t = Szundi; // helyes a C-ben, és a C++-ban is.
```

```
cout << t; // kimenet: 5
```

Más változat



► A C++-ban:

```
enum torpe {Tudor=1, Vidor, Hapci, Szende,  
Szundi, Morgo, Kuka} x;
```

```
x = Hapci;
```

```
cout << x << endl; // kimenet: 3
```

```
torpe y;
```

```
y = Kuka;
```

```
cout << y << endl; // kimenet: 7
```

9.6. Önhivatkozó struktúrák

- ▶ Más néven: rekurzív módon definiált struktúrák.
- ▶ Egy struktúrán belül nem adhatunk meg ugyanolyan típusú struktúrát adattagként.
- ▶ Egy adattag lehet mutató ahhoz a struktúrához, amelyben definiáljuk. A C++-ban lehet referencia is.
- ▶ Ez a „rekurzivitás” lehet indirekt is.
- ▶ Különböző adatszerkezetek definiálására használhatjuk.

Példa önhivatkozó struktúrára

- ▶ Direkt „rekurzivitás”:

```
struct név {  
    // ...  
    struct név *p;  
    // ...  
};
```

- ▶ Egyszeresen láncolt lista esetén használhatjuk.

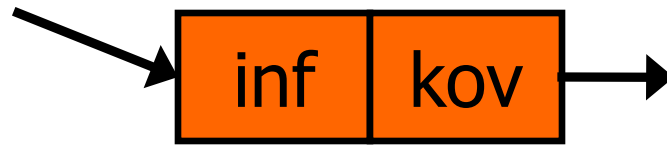
Indirekt önhivatkozás

```
struct n1;    // előzetes, nem teljes deklaráció
struct n2 {
    struct n1 *x;
    // ...
};
struct n1 {
    struct n2 *y;
    // ...
};
```

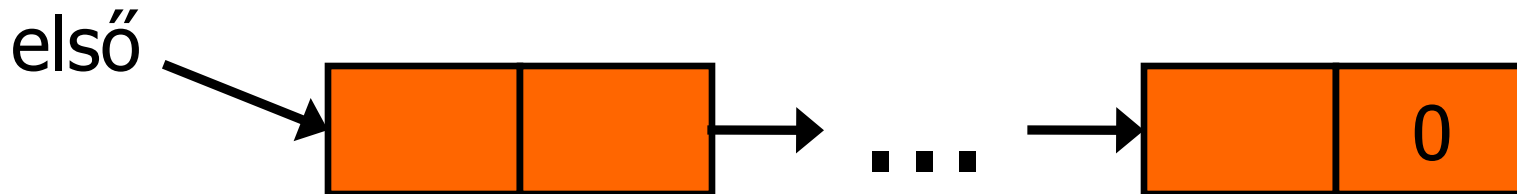
Az n2 struktúrában
hivatkozunk n1-re, és
fordítva, az n1-ben n2-re.

Egyszeresen láncolt lista

- ▶ A lista minden eleme két részből áll:
 - ▶ inf: információs rész (egy adat),
 - ▶ kov: mutató a következő listaelemre.



- ▶ A lista:



A struktúra C++-ban

```
struct elem {  
    int inf;  
    elem *kov;  
};
```

- ▶ Mutató az első elemre:

```
elem *elso;
```

- ▶ C-ben `elem` helyett `struct elem` kell.

A létrehozás



- ▶ Írjunk függvényt, amely egyetlen elemből álló listát hoz létre.
- ▶ Az információs rész legyen: -1.
- ▶ A mutató a következő elemre legyen: 0.
- ▶ A függvénynek paraméterként adjuk át a mutatót az első elemre.
- ▶ Két változat: C-ben, és C++-ban.

A létrehozás C-ben

```
void letrehoz_C(struct elem * *elso)
{
    *elso = (struct elem *)malloc(sizeof(struct elem));
    (*elso)->inf = -1;
    (*elso)->kov = 0;
}
```

▶ A meghívás:

```
struct elem *elso;
letrehoz_C(&elso);
```

Típusdeklarációval C-ben

```
typedef struct elem *pelem;  
void letrehoz_CT(pelem *elso) {  
    *elso = (pelem)malloc(sizeof(struct elem));  
    (*elso)->inf = -1;  
    (*elso)->kov = 0;  
}
```

- ▶ A meghívás:
 pelem elso;
 letrehoz_CT(&elso);

A létrehozás C++-ban

```
void létrehoz(elem * & elso)
{
    elso = new elem;
    elso->inf = -1;
    elso->kov = 0;
}
```

▶ A meghívás:

```
    elem *elso;
    létrehoz(elso);
```

Típusdeklarációval C++-ban

```
typedef elem *pelem;  
void letrehoz_T(pelem & elso) {  
    elso = new elem;  
    elso->inf = -1;  
    elso->kov = 0;  
}
```

- ▶ A meghívás:
 pelem elso;
 letrehoz_T(elso);

10. Állománykezelés



10.1. C stílusú állománykezelés

- ▶ Alacsonyszintű állománykezelés
- ▶ Magasszintű állománykezelés

10.2. Állomány-folyamok