PROGRAMOZÁSI NYELVTÖRTÉNET
(ELŐADÁSJEGYZET)

I. Az első programozási nyelvek:

a. 1954: Formula Translator (Ismertebb nevén FORTRAN0);
b. 1957: FORTRAN (Ez a ma ismert FORTRAN programozási nyelv);
c. 1960: ALGOL60 programozási nyelv megjelenése:
   i. Fenntartott szavak;
   ii. Változók típusát explicit meg kell adni;
   iii. Integrációs szerkezetek;
   iv. Blokkstruktúra;
   v. Rekurzió;
d. 1964: BASIC nyelv megjelenése (Beginner's All-purpose Standard Instruction Code);
e. PL/I nyelv megjelenése (kivétel-kezelési lehetőségek megjelenése);

II. Korszerűbb programozási nyelvek:

a. 1972: A C nyelv megjelenése (Dennis Richie). A C nyelv legfőbb jellemzője a hatékonysága volt;
b. 1983: ADA83 nyelv megjelenése. Az ADA programozási nyelv legfőbb erőssége viszont már a biztonság volt. Korszerűbb változata az ADA95 nyelv;
c. 1980: A C++ nyelv megjelenése (Bjarne Stroustrup);

III. Programozási nyelvek fejlődése:

a. 1990: Szabványosítás;
b. 1993: SUN, JAVA;
c. 1994: HP → STL;
d. 1998: A C++ szabványosítása;
e. 2003: A C++ szabvány revíziója (A C++ 0X a második szabványrevízió);

IV. Programozási nyelvek osztályozása:

a. Imperatív programozási nyelvek:
   i. Procedurális;
   ii. Strukturális;
   iii. Objektum Orientált (OO nyelvek, pl. Borland Delphi Enterprise);
b. Deklaratív programozási nyelvek (SQL, XSLT);
c. Funkcionális programozási nyelvek (LISP, HASKELL, CLEAN);
d. Generatív programozási nyelvek;
e. Multi-paradigmás programozási nyelvek (C++);

C++ FORDÍTÁSI FOLYAMAT
(ELŐADÁSJEGYZET)

```
#include „fájl.név” – Munkakönyvtárban keresi a fájlt;

#include <fájl.név> – Standard könyvtárban keresi a fájlt;
```

#define azonosító – speciális esetek:

```
#define SIZE 100;
int t[SIZE]; [ → int t[100]; ]
#undef SIZE

#define SIZE 200
int t2[SIZE]; → int t[200];

#define MAX(a,b) ((a) > (b) ? (a) : (b))
int a=5; 2
int b=6; 1
int z=MAX(x,y) ↔ int z = x > y ? x : y;
int z2=MAX(x,y) ↔ int z2 = ((x) > (y) ? (x) : (y))*2;

#define str(x) #x
cout << str(hello); ↔ cout << „hello”;

#define F(a,b) a##b
F(alma,korte) ↔ almakorte

#if SIZE > 50
#elif SIZE > 20
#else
#endif

#ifdef SIZE
int t[SIZE];
#else
#define SIZE 50;
#endif

#ifndef SIZE
#define SIZE 100;
#endif
```

a.h

```
#ifndef A_H
#define A_H
…
#endif
```

b.h

```
#ifndef B_H
#define B_H
#include „a.h”
…
#endif
```

Előredefiniált makrók:

WIN32

```
#ifdef WIN32
#define SLASH „\\”
#else
#define SLASH „/”
#endif

void f(int a){…};
void f(double a){…};
void f(int a, int b){…};
```

3 függvény, háromféle paraméterezés, de az azonosító ugyanaz! Ez csak C++ nyelvben működik, C-ben már nem!

```
#ifdef __cplusplus
extern „C”
#endif
…
#ifdef __cplusplus
…
#endif

#line pozitív egész //Számláló átírása

#error szöveg //Szöveg kiírása Standard Error-ra

#pragma

#pragma once (Csak Visual Studio-ban!)

#pragma warning(disable:4267) //4267-es warning kikapcsolása

__LINE__

__TIME__ //cout << __TIME__;

__DATE__

__FILE__
```

Azonosítók:

Kezdődhetnek az angol abc 26 betűjével (illetve „_” jellel), folytatódhatnak az angol abc 26 betűjével, vagy „_” jellel, illetve számokkal. Erősen ajánlott csak az angol abc 26 betűjét használni.

Fenntartott szavak:

if, else, for, while, class, strnd, stb.

alma ≠ Alma

szoveget_kiir

Egész típusok:

16 – egész érték

5+8 – egész érték

5/8==0 //egész értékek esetén

Boolean-konstansok:

FALSE, TRUE //A FALSE konvertálódhat 0-ra, a TRUE pedig minden olyan egészre, melynek értéke nem 0.

Speciális karakterek:

'a' 'b' '0' '^' '\n' '\t' '\\' - char

unsigned/signed char - előjel nélküli/előjeles char

unchar_t - unicode karakterek

Számok:

014 (oct) → 12 (dec) → 0xC (hex)

031 (oct) → 25 (dec) → 0x19 (hex)

050 (oct) → 40 (dec) → 0x38 (hex)

shortint/int/longint - integer típusok (mindkettő előjeles)

unsigned shortint/int/longint - előjel nélküli integer típusok

20000+20000+20000+20000L - az összes érték longinteger-re konvertálódik, és ebben a típusban adódnak össze.

2L 2UL

1.23, 0.15, .15, 1. - lebegőpontos számok

5./8 - ebben az esetben lebegőpontos osztás történik

Lebegőpontos típusok:

float - egyszeres pontosságú

double - kétszeres pontosságú (ajánlott)

long double - kiterjesztett

sizeof(char) - kiírja a típus által lefoglalt méretet

sizeof(char)≤sizeof(shortint)≤sizeof(int)≤sizeof(longint)

sizeof(X)=sizeof(signed X)=sizeof(unsigned X)

ALAPVETŐ C++ ISMERETEK
(GYAKORLATJEGYZET)

Fordítási módszerek:

```
g++ -o valami

g++ a.cpp b.cpp

Output-file: a.out
Futtatás: ./a.out

g++ -c a.cpp
```

Ekkor csak objektumfájl keletkezik a.o néven, mely az „ed” programmal szerkeszthető.

Az objektumfájlból a Linker program készít futtatható állományt, de készíthető belőle dinamikus könyvtár (*.dll [Windows], *.so [Linux]) is (függvények, eljárások esetén).

További kapcsolók:

-Wall (összes figyelmeztetés)

-W vagy -Wextra

-Werror (figyelmeztetések hibaként)

Bonyolult program esetén nem ajánlott kikapcsolni a figyelmeztetéseket, mert ez megnehezítheti az esetleges hibakeresést!

Bájtkód – futtatókörnyezet (JAVA, .NET)

Interpretált nyelvek: bash, sh, Perl, Python (*.py illetve *.pyc), PHP

Hello World – C program (hello.c):

```
#include <stdio.h>
int main()
{
 printf(„Hello World!”);
 return 0;
}
```

>Fordítási módszerek:

```
g++ -o hello hello.c

make hello

gcc -o hello hello.c
```

Hello World – C++ program (hello.cpp):

```
#include <iostream>
int main()
{
 std::cout << „Hello World!” << std::endl;
 return 0;
}
```

Lehet ezt ennél egyszerűbben is:

```
#include <iostream>
using namespace std;
int main()
{
 cout << „Hello World!” << endl;
 return 0;
}
```

A két módszer közti különbség a C++ által fenntartott szavak mennyisége. Utóbbi esetben nagyobb a fenntartott szavak mennyisége, így több a kódolásra vonatkozó korlátozás, mivel több modul került betöltésre.

```
int main()
{
 using std::cout;
 using std::endl;
 :
}
```

Sorok lezárása:

<< endl; (legegyszerűbb)

<< „\n”; (Linux alatt „\n” jelzi a sorok végét)

<< flush; (VMS operációs rendszer)

Példa „flush” használatra (VMS):

```
int i;
cout << „adjon meg egy szamot: ”;
cout << flush;
cin >> i;
```

Hibakezelés:

cerr << „Ez egy hiba!”; (szöveg kiírása hibaként)

exit(1); (hiba jelzése és azonnali kilépés)

Standard Input/Output:

stdio.h (C)

iostream.h (C++)

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

Lehet másképpen is:

```
#include <iostream.h>
iostream cout;
int main()
{
 …
 return 0;
}
```

```
std::cout
::cout
```

További modulok:

```
cstdio
cstdlib
```

```
std (*.c, *.h)
```

```
man printf (printf függvényre vonatkozó információk lekérése)
```

```
man 3 printf (printf-információ 3-as szekciójának lekérése)
```

Függvénydefiniálás:

```
int fv(void);
```

```
fv() extren „c” int fv(void); (C-n kívül definiált függvény)
```

PreProcesszor:

```
preprocessor.h (a preprocesszor header-állománya)
```

```
#ifdef __cplusplus
 extern „c”
#endif
int fv(void);
```

almafa.h

```
#ifndef ALMAFA_H_INCLUDED
#define ALMAFA_H_INCLUDED
 …
#endif
```

```
cout << „a1” << endl;
```

Debugger:

```
gdb (debugger program)
```

```
gdb ./hello (A „hello” program ellenőrzése)
```

```
>..run
>breakpoint main
>next       step
>continue
```

További fordítási lehetőségek:

```
g++   -g    -oo   -ggdb

      debug symbols

      -02

      -03
```

Az utóbbi két kapcsoló azt határozza meg, hogy a lefordított kód mennyire legyen összhangban a program forráskódjával (ugyanis a fordító optimalizálva készíti el a futtatható állományt, és hatékonysági szempontból nem feltétlenül a forráskódban szereplő szintaktika a legideálisabb).

Leggyakrabban használt típusok (méret szerint):

char (karakterek, lefoglalt terület: 1 Byte)

short (egész értékek, lefoglalt terület: 2 Byte)

int (integer – egész értékek, lefoglalt terület: 4 Byte)

long (egész értékek, lefoglalt terület: 4 vagy 8 Byte)

Példa a típusokra:

```
int i;

void f(long *p);
```

MŰVELETEK TÍPUSOKKAL
(ELŐADÁSJEGYZET)

Karakter-típus:

„Hello” – konstans karaktertömb

H	e	l	l	o	\0	C/C++
5	H	e	l	l	o	Pascal

```
char str[]={'H','e','l','l','o','\0'};
```

```
char str2[]="Hello"; | str2[0]='e'; (jó megoldás)
```

```
const char *s="alma"; | *s='e'; s[0]='e'; (rossz megoldás)
```

Pointerek:

```
int t[]={1,2,3};
```

Érték	1	2	3
Index	0	1	2

```
int *pt;    t[2];

pt=t;
```

Konvertálás:

Másoljuk át „a” változó tartalmát „b” változóba!

- Ha mindkettő típusa ugyanaz, akkor egyszerű másolás;
- Ha a ~ b long double, akkor a másik is azzá konvertálódik;
- Ha a ~ b double, akkor a másik is azzá konvertálódik;
- Ha a long int, b unsigned int és unsigned int értékkészlete elfér long-ben, akkor b long int lesz;
- Ha a long int, b unsigned int és unsigned int értékkészlete nem fér el long int-ben, akkor mindkettő unsigned long integer lesz;
- Ha a ~ b long int, akkor a másik is az lesz;
- Ha a ~ b unsigned int, akkor a másik is az lesz;
- Ha a ~ b int, akkor a másik is az lesz;

```
int a = -1;
unsigned int b = 2;
if(a < b)
{
 cout << „Kisebb” << endl;
}
else
{
 cout << „Nagyobb” << endl;
}
```

Operátorok:

```
név :: entitás
```

tagkiválasztás . (t.a, t → a)

indirekt címzés *

```
struct T
{
 int a;        t.a=15;
 int *ap;      *(tp.a)=15; - rossz megoldás!
}              (*tp).a=15; tp → a=15; - jó megoldás.
T*tp = new T;
```

int x;

x++; - int operator++(int t){int tmp=t; t=t+1; return tmp;}

++x; - x értéke megnövelve 1-el

++++++++x - x értéke megnövelve 4-gyel

! logikai negáció

~ bitenkénti negáció

* szorzás

/ osztás

% modulo-számítás

- kivonás

<< biteltolás balra

>> biteltolás jobbra

Logikai műveletek:

< <= == >= > (az egyenlőségnek a „==” felel meg)

& bitenkénti és

| bitenkénti vagy

&& logikai és

|| logikai vagy

További műveletek:

a += 1 (a értéke megnövelve 1-gyel)

x *= 5 (x értéke megszorozva 5-tel)

Kifejezések:

- Változók
- Konstansok
- Függvényhívások

a = b + c * d (matematikai műveleti sorrend érvényesül)

a = f() + g() + h() (nem határozható meg az összeadás sorrendje)

```
int fv(int a, int b, int c)
{
 cout << a << b << c << endl;
}
int f(){cout << 'F' << endl; return 1;}
int g(){cout << 'G' << endl; return 2;}
int h(){cout << 'H' << endl; return 3;}
fv(f(),g(),h());
```

a&&b a||b

- Ha mondjuk a [b] értéke hamis, akkor a&&b hamis (b [a] változóval nem is foglalkozik)
- Ha a [b] értéke igaz, akkor a||b igaz (b [a] változóval nem is foglalkozik)

Emiatt ha ezt írjuk:

a&&f(), a||f(), b&&f(), b||f(), akkor egyáltalán nem biztos, hogy a függvény meghívásra kerül!

```
int t[10];
int i=0;
while(i<10)
{
 t[i]=i++;
};     t[1]=0
```

y = a * b + c

„a * b" helyén létrejön egy temporális kifejezés. A temporális kifejezések csak addig élnek, amíg kiértékelődnek.

```
int x = 5 + 6;
```

Ez működőképes megoldás, ez lesz belőle fordításnál: int x=11;

FÜGGVÉNYEK ÉS TÍPUSOK KEZELÉSE
(GYAKORLATJEGYZET)

Kommentárok írása:

```
// Egysoros kommentár szövege

/*
Kommentár szövege
több sorba
tördelve
*/
```

Fahrenheit átváltása Celsiusokra (celsius.c):

```
#include <stdio.h>
int main()
{
 const int lower = -40;
 const int upper = 200;
 const int step = 10;
 int i;
 for(i=lower; i<=upper; i+=step)
 {
  /*
  Ez rossz, mert itt egeszosztas fog tortenni (5/9), illetve a celsius
  kimeneti formatuma sem megfelelo (%d)
  */
  fprintf(stdout,"fahr=%a\tcels=%d\n",i,5/9*(i-32));
  //Most kovetkezik a jo megoldas:
  fprintf(stdout,"fahr=%a\tcels=%lf/n",i,5./9*(i-32));
  //5. helyett akar 5.0-t is irhattunk volna.
 }
 return 0;
}
```

Figyelmeztetés:

Ha egy értéket 8 Byte-os long típusban tároltunk le, és ezt egy 4 Byte-os double típusban kívánjuk kiíratni, akkor nagy valószínűséggel rossz eredményt fogunk kapni, ezért tehát a neki megfelelő long típusban szabad csak kiíratni a képernyőre!

Fahrenheit átváltása Celsiusokra (celsius.cpp):

```
#include <iostream>
using namespace std;
int main()
{
 const int lower = -40;
 const int upper = 200;
 const int step = 10;
 for(int i=lower; i<=upper; i+=step)
 {
  cout << „fahr=" << i << „ cels=" << 5./9*(i-32) << endl;
  //Itt most az eredmenyt double tipusban kapjuk meg.
 }
 return 0;
}
```

Pointerek:

```
int* p; (mutató deklarálása)

int *p,q; (a p pointer lesz, de a q már nem!)
```

Visual Studio 6:

```
for(int i=0; i<=10; i=i+1)
{
 cout << i;
}
cout << i;
```

Legnagyobb meglepetésünkre a ciklus után következő „cout << i;” sorban megkapjuk, hogy i=10, pedig az i változót lokális változóként szerettük volna deklarálni! Tehát mégsem lett lokális változó!

Hogy az i változó valóban lokális változó legyen, a szükség van a következő sor beírására:

```
#define for if(0); else for
```

Most már tényleg lokális változó lesz az i változó.

A „csellengő else” problémája:

```
if(…)
 if(…)
else
 …
```

A látszattal ellentétben az else ág nem az első if(…) feltételhez fog tartozni, hanem a másodikhoz!

A Fahrenheit-program függvényként:

```
double fahr2cels(double f);
```

Ez a sor még csak deklarálta a függvényt, de semmit nem mondott annak működéséről, vagyis nem definiálta. Ez a módszer akkor használatos, amikor a függvényt a forráskód végén kívánjuk elhelyezni, vagy pl. egy osztály műveleteit deklaráljuk, stb.

A függvényt a következőképpen kell definiálni:

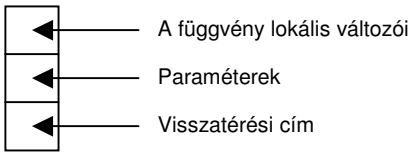
```
double fahr2cels(double f)
{
 return (5./9)*(f-32);
}
```

A függvényt pedig így lehet meghívni:

```
int i;
cout << fahr2cels(i);
```

ahol i értéke egy tetszőleges, double típusú szám.

Veremkezelés:



Deklaráció / definíció:

```
int i; (az i változó deklarációja)

int i=4; (deklaráció és definíció)
```

Makrók definiálása:

```
#define MAX(a,b)
 a > b ? a : b
```

Ha hamis a „?” előtti rész, akkor eredményként az „a” változót kapjuk vissza, egyébként pedig a „b” változót.

Makrók használata:

MAX(3,4)*12 a fentebb definiált esetben valószínűleg rossz eredményt ad, mivel a következő műveletek kerülnek behelyettesítésre:

```
3 > 4 ? 3 : 4 * 12
```

Mivel a kifejezésben nem szerepelnek zárójelek, ezért a műveletek nagy valószínűséggel nem a kívánt sorrendben lesznek elvégezve! A megoldást a megfelelő zárójelezés adja:

```
#define MAX(a,b)
 (3 > 4 ? 3 : 4)
```

Ez már jobb megoldás, mely a matematikai alapműveletek nagy részére megfelelően működik, azonban még ez a megoldás sem tökéletes… Vegyük a következő esetet:

MAX(2<<2,4), ahol „<<” a bitenkénti eltolás művelete, az eltolás mértéke pedig 2 bit. Ekkor megint nem lehet megmondani, hogy pontosan melyik művelet kerül végrehajtásra először, így valószínűleg megint rossz eredményt kapunk. Vegyük a következő példát:

```
#define F2C(a)
 (5./9*(a-32))
```

ahol a=2<<2. Ekkor a definícióban szereplő kifejezés így néz ki:

(5./9*(2<<2-32)), ahol előbb a 2-32 művelet kerül végrehajtásra, így már nem 2 lesz a biteltolás mértéke, hanem -30!

A probléma megoldását a definíciókban szereplő kifejezések 100%-os bezárójelezése adja. Az említett MAX makró esetében akkor ez lesz a megoldás:

```
#define MAX(a,b)
 ((a)>(b) ? (a):(b))
```

Ezzel a módszerrel makrónk már majdnem mindig jól működik, kivéve a következő esetben:

```
MAX(i++,33)
```

Ekkor a makróban ez a kifejezés kerül kiértékelésre:

```
((i++)>(33) ? (i++):(33))
```

Ebben az esetben az i változó értéke kétszer lesz megnövelve 1-gyel, így megint rossz eredményt kapunk.

Pointer i-re:

```
int *p = &i; (i változó címe)
```

Ha pl. i=4, attól még nyugodtan lehet, hogy *p=3.
Lehetséges címzési módszerek:

```
const int c;
int* q = &c;
q=q+1;

q++;

double *s;
```

A sizeof() operátor:

A sizeof() operátor lekérdezi az argumentumban megadott változó méretét. Példa a sizeof() operátor használatára:

```
int t[32]={1, 2, … 32};
int a;
a = sizeof(t);
```

Ekkor „a” változó értéke 32*n lesz, mivel n=sizeof(int). Megadhatjuk argumentumként egy tömbnek egy elemét is:

```
b = sizeof(t[0]);
```

Tömb elemeinek indexelése, pointerek használata:

A C/C++ nyelvben egy tömb elemeinek indexelése 0-tól kezdődik, ennek megfelelően egy n elemű t tömb első eleme t[0], utolsó eleme pedig t[n-1] lesz. Az index-túlcsordulás elkerülésére oda kell figyelni.

Azonosságok:

```
*(t+0) ≡ *t[0]

*(t+1) ≡ t[1]
```

További példák:

```
int *p;
int i;
for(i=0; i!=32; ++i)
{
 p = t + i;
 cout << *p << endl;
}
```

```
int *p = t;
for(int i=0; i!=32; ++i)
{
 cout << *p << endl;
 p++;
}
```

```
nullpointer int *p = 0;
```

Változók értékeinek megnövelése:

Egy változó értékének egyel történő megnövelésére alapvetően három utasítás létezik, melyek egy „i” integer típusú változó esetében a következők:

- `i=i+1;`
- `++i;`
- `i++;`

Az első utasítás működése egyértelmű. A második eset ugyanúgy működik, mint az első, csak rövidebb. Különbség csak a harmadik esetben van, amikor is egy „a=(i++)” utasítás esetén a=(i++) ≡ (int temp=i; i=i+1; temp).

```
cout << 3*i++ << endl;
```

esetén már megint nem kapunk jó eredményt.

```
((i+3) … (++i))
```

kifejezés esetén pedig megint nem lehet meghatározni, hogy melyik művelet kerül először végrehajtásra, így szinte biztos, hogy megint nem kapunk jó eredményt.

LÁTHATÓSÁG ÉS ÉLETTARTAM
(ELŐADÁSJEGYZET)

Névterek:

Függvények és változók logikai csoportosítása, melyekkel megoldható a változók láthatóságának problémája.

Namespace modulok:

Szabványos bemeneti/kimeneti modulok. Egy projecten belül több „using namespace” parancsot is kiadhatunk, de ha egy függvény vagy eljárás több modulban is előfordul, akkor fordításkor hibaüzenetet kapunk, mert a fordító nem tudja eldönteni, hogy melyik függvényt vagy eljárást akarjuk használni.

Ennek elkerülése végett a header (*.h) állományokban a „using namespace” parancs használata szigorúan tilos!

Névtelen névterek (h.cpp):

```
namespace
{
 int x;
 void f(){ … }
}
```

A névtelen névtereket a static-típusú függvények, eljárások és változók kiváltására találták ki, mivel a static-típusú adatok túl sok helyet foglalnak. A static-típusú adatok lényege, hogy megmaradnak a függvényből történő kilépés után is.

```
static x;
static void f(){ … }

void g()
{
 static int n=0;
 int m=5;
 ++n;
}
```

Láthatóság (main.cpp):

```
int i; (globális változó)
static int j; (static változó)
extern int n; (externális változó)
```

Példaprogram:

a.cpp:

```
int n=25;
```

b.cpp:

```
extern int n;
int main()
{
 std::cout << n;
 return 0;
}
```

Ennek eredményeként a program ki fogja írni n változó értékét, vagyis 25-öt, amennyiben a LINKER-rel megfelelően összekapcsoljuk az a.cpp és b.cpp fájlt.

Példa namespace-névtérre:

```
int i;
static int j;
extern int n;
namespace X
{
 int i; //X::i (lokális változó)
 int f()
 {
  i=n; //X::i=::n
  return i;
 }
}
namespace
{
 int k;
}
void f()
{
 int i;
 static int k;
 {
  int j=i;
  int i=j;
  std::cout << i << j << k << X::i << k << i;
 }
}
```

Függvényen belül szigorúan tilos globális változók értékeit megváltoztatni (pirossal szedett rész), mert ez gusztustalan, és áttekinthetetlenné teszi a kódot!

Változók definiálása:

Törekedni kell arra, hogy egy változót csak akkor definiáljunk, amikor arra tényleg szükségünk van (feltételek, ciklusok, stb.).

Élettartam:

Helyfoglalási methódusok:

- Statikus helyfoglalás
- Dinamikus helyfoglalás
- STACK helyfoglalás

I. Csoport: globális névterek, osztályok statikus változói (A program elején születnek és a program végéig élnek);

II. Csoport: automatikus objektumok (deklaráláskor születnek a változók és az eljárás vagy függvény lefutásának végéig élnek);

III. Dinamikus változók (a „new” operátor segítségével keletkeznek, és a delete parancs kiadásáig élnek);

IV. Nemstatikus adattagok (az őket tartalmazó objektummal együtt keletkeznek, és az őket tartalmazó objektum megszűntetéséig élnek);

V. Lokális, statikus objektum (a deklaráció első kiértékelésénél jönnek létre, és a program lefutása után szűnik meg);

VI. Temporális objektumok (egy részkifejezés vagy reguláris kifejezés kiértékelésekor jönnek létre, és a teljes kifejezés kiértékelése után szűnik meg);

Példák:

```
void g(string s1; string s2)
{
 char *sp = (s1 + s2) c_str();
 sp[0]=”a”;
 if(strlen((s1+s2) c_str())<10)
 {
  std::cout << strlen((s1+s2) c_str());
 }
}

int f()
{
 int x;
 return x;
}

int *i = new int[5];
int *t = new int[5];
delete[] i;
delete[] t;
```

Answer-program:

```
char* answer(char* question)
{
 cout << question;
 char buffer[80]; //Rossz, mert statikus a tarterulet!
 cin >> buffer;
 return buffer;
}
int main()
{
 cout << answer(„How are you now?”) << endl;
//Ez is rossz, mert a buffer lokalis tomb!
 return 0;
}
```

Javítás:

```
string answer(char* question)
{
 cout << question;
 string buffer;
 cin >> buffer;
 return buffer;
}
int main()
{
 cout << answer(„How are you?”);
 cout << answer(„Sure?”);
 return 0;
}
```

MŰVELETEK KARAKTEREKKEL, TÍPUSKONVERZIÓK
(GYAKORLATJEGYZET)

UNIX: CAT – Lebutított változat (cat.c):

```
#include <stdio.h>
int main()
{
 int ch;
 while((ch=getchar())!=EOF)
 {
  putchar(ch);
 }
 return 0;
}
```

Alternatív megoldás:

```
int ch;
ch=getchar();
while(ch!=EOF)
{
 putchar(ch);
 ch=getchar();
}
```

A getchar() függvény valójában egy [0..255] közé eső egész számot ad vissza, ez pedig nem más, mint a kiíratásra szánt karakternek a kódja.

A Linux/UNIX rendszereken, amennyiben az inputot nem fájlból, hanem billentyűről olvassuk be, akkor a CTRL+D billentyűkombináció jelzi az input végét, vagyis a fájlvége-jelet ez helyettesíti (EOF). A bevitel helyességének ellenőrzésére a „cat | grep” parancs szolgál.

CAT program – C++ változat (cat.cpp):

```
#include <iostream>
using namespace std;
int main()
{
 char ch;
 while(cin >> ch)
 {
  cout << ch;
 }
 return 0;
}
```

Amennyiben a többszörös szóközöket (wide-space) nem kívánjuk megtartani, akkor a fenti megoldás nagyon praktikus lehet. Ha viszont egy az egyben vissza akarjuk kapni a bemenetet, akkor a while-ciklus feltételét ki kell bővíteni a következő paranccsal:

```
cin >> noskipws;
```

így ez lesz az új ciklusfeltétel:

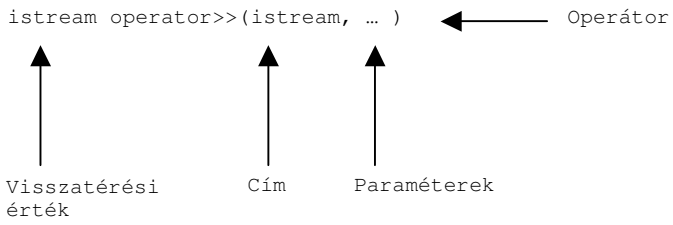
```
while(cin >> noskipws; cin >> ch)
```

Ennek megvalósításához azonban szükségünk van manipulátor-függvényekre, melyek az iomanip modulban találhatók, tehát a program elejét ezzel is ki kell bővíteni:

```
#include <iomanip>
```

Az iostream függvényekről:

iostream cin




```
int n,m;
cin >> n >> m;
```

Például itt lehet haszna a többszörös szóközök elhagyásának, ha például így adjuk meg a bemenetet:

1 _ _ _ 2

vagy

1↓
2

ahol jelen esetben „_” jelöli a szóközöket, „↓” pedig a leütött ENTER billentyűket.

Típuskonverziók:

Logikai Bool operátor (bool):

A „cin” bemeneti parancs eredménye átkonvertálható logikai bool típussá, amennyiben a beolvasás során nem lép fel semmilyen hiba, és nem fájlvége jelet olvastunk.

Bit (bit):

A bit típusnak három alapvető állapota lehet:

- fail (ha a „cout” parancs végrehajtásánál hiba lépett fel);
- bad (ha helyreállíthatatlan hiba keletkezett);
- EOF (ha a fájlnak vége);

Amennyiben a fájlvége jel után tovább olvasunk, akkor „bad” lesz az aktuális állapot.

Ha a bit típus értéke „bad” vagy „fail” akkor a SKIP parancs kerül végrehajtásra, hogy a program ne szálljon el, azonban van lehetőség kivételkezelésre is.

Beolvasás:

Karakterek beolvasása:

```
cin >> ch;

cin.get(ch);
```

Karakterek kiíratása:

```
cout << ch;

cout.put(ch);
```

Az angol ábécé nagybetűsítése:

```
int main()
{
 char ch;
 while(cin >> ch)
 {
  cout << (((ch>='a')&&(ch<='z')) ? ch-'a'+'A' : ch);
 }
 return 0;
}
```

Fontos megjegyezni, hogy az angol ábécé kisbetűinek [a..z], illetve nagybetűinek [A..Z] egymástól való távolsága a kódolásban megegyezik, ezért kezelhető viszonylag könnyen a fentebb leírt „cout" parancs.

Meg kell jegyezni még azt is, hogy az említett „cout" parancs „ch-'a'+'A'" része egy integer értéket ad vissza, mely a „:ch" rész után visszaalakul karakterré.

További típuskonverziók:

```
cout << static_cast<char>( … );
```

Template

Ez a művelet elvileg bármilyen típusra jól működik, azonban némely esetben értékvesztés is előfordulhat, például:

```
double d;
static_cast<int>(d);
```

Lehetőség van számértékek karakterláncokra, illetve számot tartalmazó karakterláncok számértékekre konvertálására is. Az ehhez szükséges modul a „cstdlib" modul. Legfőbb függvényei a következők:

atoi(x) – karakterláncok integerré konvertálása;

itoa(y) – integer érték karakterlánccá konvertálása;

CAST műveletek:

static_cast;

dynamic_cast<T*>(…);

p=dynamic_cast<T*>(…)

A művelet futási időben hajtódik végre, sikertelen átalakítás esetén p==0.

const_cast<T>(…);

```
const int a=4;
int b=const_cast<int>(a);
```

A „b” változó az „a” változó értékével dolgozik tovább.

reinterpret_cast<T>(…);

Ez a művelet elméletileg minden típust át tud alakítani bármilyen más típussá, azonban a gyakorlatban egyáltalán nem garantált a megfelelő működés. Nem működik jól például a következő esetben:

```
struct T
{
 char a;
 char b;
 char c;
}
T *t;
short s;
t=reinterpret_cast<T*>(&s);
```

ahol „&s” címre történő hivatkozás. Mivel azonban a struktúra memóriaterülete nagyobb, mint az „s” változó által lefoglalt memóriaterület, ezért az átalakítás során nem a kívánt értéket fogjuk megkapni („a” és „b” változók értékeit még jól alakítja át, azonban a „c” változónál már problémák lépnek fel).

Pointerek:

int t[8]={ … };

t+i ≡ &t[i]

Pointer-aritmetika:

t+1 ≡ &t[1]

*(t+1) ≡ *(&t[1])

A második eset nem működik, mert a „*” és a „&” jelek kiütik egymást (de-referencia).

Legyen „t” továbbra is egy tömb, „a” és „b” változók pedig két, „t” tömbön értelmezett pointer, ahol az „a” változó által hivatkozott memóriacím indexe kisebb, mint a „b” változó által hivatkozott memóriacím indexe.

Ebben az esetben van értelme az „a” és „b” pointerek aritmetikai összehasonlításának, vagyis értelmezve vannak az egész számok körében megszokott aritmetikai műveletek, illetve relációk:

a<b, a<=b, a>b, a>=b, a==b, a!=b, a-b, illetve ezek megfordítása.

Természetesen ezek a műveletek, illetve relációk csakis abban az esetben működnek, amennyiben mind az „a”, mind a „b” pointer a „t” tömbön van értelmezve, mivel az általuk hivatkozott memóriakörnyezetnek meg kell egyeznie. A pointereken értelmezett aritmetikai műveletek elvégezhetőségét tehát az általuk hivatkozott tömb azonossága biztosítja.

Két pointer különbsége mindig a köztük lévő számtani távolságot definiálja. Nincs értelmezve azonban az összeadás, szorzás és osztás művelete, mivel nem eshetünk ki az általuk hivatkozott tömbből.

Nullpointer (NULL):

```
(void *)0
```

Lehetőség van nullpointerek létrehozására is, azonban a gyakorlatban óvatosan kell bánni velük:

```
T *p=0;
*p;
```

Ebben az esetben elszáll a program, mivel a nullpointer által mutatott NULL címre próbáltunk hivatkozni!

Néhány függvényhívás:

```
int f(int);

int f(std::string*)
```

A két függvényhívás közül az elsőt fogja használni a fordító.

SIZE függvények:

```
size_t;
```

strlen(const char* s) – karakterláncok hossza;

H	e	l	l	o	\0

s: „Hello\0”

A C++ nyelvben a stringek végét a „\0” jelzi.

read, printf – karakterláncok olvasása, kiíratása

```
size_t size;
while(*s++)
{
 size++;
}
return size;
```

Ekkor a „*s” végigmegy az „s” stringen.

s++; – pointer értékének megnövelése 1-gyel

```
char *const s;
```

Ebben az esetben az s++ parancs nem működik, mivel „s” string konstans, így nem lehet megváltoztatni az értékét.

Karakterláncok indexelése:

H	e	l	l	o	\0
0	1	2	3	4	-

Legyen „t” egy string, ahol t=”Hello”. Ekkor érvényesülnek a következő azonosságok:

t[2] ≡ *(t+2) ≡ *(2+t) – Karakterek kiolvasása pointerekkel

„Hello”[2] ≡ 2[„Hello”]

sizeof(T) – megadja, hogy hány bájtot foglal le a „T” változó

Példa a sizeof() operátorra:

```
int *p;
```

Ekkor a „sizeof(p)” operátor értéke 4 vagy 8 (bájt) lesz. Hogy pontosan melyik, az architektúra-függő.

```
int t[8];
```

Ebben az esetben a „sizeof(t)” operátor értéke 32 (4*8 bájt) lesz, feltéve, hogy az integer-típus 4 bájton tárolódik.

DEKLARÁCIÓK, DEFINÍCIÓK
(ELŐADÁSJEGYZET)

Definíció:

Új változó vagy függvény bevezetése, illetve kezdőértékek megadása

Deklaráció:

Már (külső modulokban) létező változók használatba vétele

[módosító (elhagyható)] alaptípus deklarátor [inicializátor (elhagyható)]

Módosító:

extern, static, register, inline

Alaptípus:

int, double, bool, stb.

Deklarátor:

new

Operátor:

*, &, const – név elé kell írni

[], () – név után kell írni

Definíció:

Karakter-pointer:

++cptr; – pointer érték növelés

++(*cptr); – mutatott érték növelés

char* cptr; – mind a pointer, mind a mutatott terület változtatható

const char* cptr; – pointer változtatható, mutatott érték konstans

char const *cptr; – mutatott érték változtatható, pointer konstans

char* const cptr; – pointer konstans, mutatott érték nem

const char* const cptr; – mindkettő konstans

char const * const cptr; – mindkettő konstans

Tömbök:

```
int t[5];

int t[5] = {1, 2, 3, 4, 5}

int t[] = {1, 2, 3, 4, 5}
```

sizeof(t) = tömb mérete bájtokban, jelenleg 20.

```
int size = sizeof(t) / sizeof(int) – a tömb elemszámának megadása.
```

A sizeof() operátor csak a vele azonos blokkban lévő adatokra működik!

Az azonos fordítási egységben kezelt tömb és pointer lényegében egyformának tekinthetők.

b.cpp

```
int ip[]={1,2,3,4};
void f();
int* ipp=ip;
int main()
{
 f();
 return 0;
}
```

Természetesen ez a példa nem működik.

Többdimenziós tömbök:

```
double t[10][10];
```

Struktúrák:

```
struct T
{
 int x;
 int y;
};
int main()
{
 …
}
```

Ha a „;” lemarad, akkor nem működik ez a megoldás.

Legyen „p” nullpointer. Ekkor „++p”, illetve „p++” szintén nullpointer lesz, vagyis „p” értéke nem változik meg.

További deklarációk:

```
extern int i;

extern char *ptr;

extern double t[][20];

/* extern */ char *getenv(const char* var);

struct date;
```

```
struct date d;

/* static int si; */

/* const int ci = 10; */

extern const eci;
```

Forward deklaráció:

```
struct Egyik;
struct Masik
{
 Egyik *ep;
}
struct Egyik
{
 Masik *mp;
};
```

Egyszerre több változót is lehet deklarálni, például:

```
static float f, g, h;
```

auto int i=0, j, k = fun(); – automatikus változó, az új szabványban új értelmet nyer, ezért tehát javasolt kerülni a használatát.

register int i=0; – a változó értéke minél többször szerepeljen regiszterben, ezzel növelhetjük az elérés sebességét

typedef int size_t; – új típus létrehozása

Elemi típuskonstrukciók:

```
int i, *ip, t(const_kif), f(param_t par), &refi = i;
```

&refi = i – az „i” változó referenciája

Referencia:

int *ip; – memóriaterületet tárol (pointer)

int &ir = i; – ez pedig az „i” változónak lesz az álneve, és nem keletkezik új változó (eljárásoknál és operátor-felüldefiniálásnál hasznos), viszont inicializálni kell

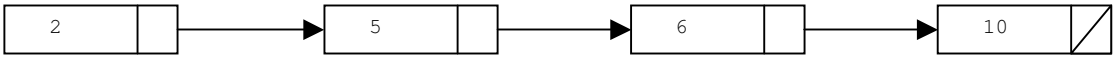
```
int &ir2 = 0;
++ip;
```

++ir ≡ ++i

ir=x ≡ i=x

Pointer-deklarálás:

```
int* ip; ≡ int *ip; – elvileg a kettő ugyanazt jelenti

int *ip1, ip2; – itt „ip2” már nem lesz pointer!

int **ipp; – pointerre mutató pointer (láncolt listák)

struct Node
{
 int x;
 Node *next;
}
```




```
int *t1[10]; – pointereket tartalmazó 10 elemű tömb

int (*t2)[10]; – pointer egy 10 elemű tömbre

double *f1(double); – függvény, ami double*-ot ad vissza

double (*f2)(double); – pointer egy double típusú függvényre (double
paraméterekkel)
```

FÜGGVÉNYHÍVÁSOK ÉS LÁNCOLT LISTÁK
(GYAKORLATJEGYZET)

Cast-függvények (C):

```
(T)a

static_cast

const_cast

reinterpret_cast
```

Memóriakezelés:

```
(double*)(0xffa4)
```

A C nyelv innentől kezdve double típusként kezeli a memóriaterületet.

Lehetséges problémák:

```
int i;
int *p;

int t[…];
int *p;
p=t;
*(p+0)=*(t+0);
t=p;
```

Ez így nem működik.

f1.cpp:

```
int t[10];
```

f2.cpp:

```
extern int *t;
extern int a;
extern „c” …
int f(int a);
double f(int a);
```

Ez így nem fordulhat elő! Egyébként pedig a C++ csak az „f(int a);” részt veszi figyelembe.

Nagybetűsítés:

```
Static_cast<char>(ch – ’a’ + ’A’)
```

Alternatív megoldás:

```
toupper(ch)

tolower(ch)
```

Ezeknek a függvényeknek a használatához be kell tölteni a „cstdlib” modult.

```
LC_ALL=c
```

$$\begin{bmatrix} en \\ hu \end{bmatrix} + \begin{matrix} us \\ u \end{matrix} .UTF-8$$

Kapcsolódó logikai függvények:

```
isupper()

isdigit()
```

Példa:

```
f(isdigit(ch)) ≡ if((ch>='0')&&(ch<='9'))
{
 …
}
```

Ezek a függvények a legtöbb kódtáblán jól működnek, azonban az IBM egyik egyedi kódtábláján problémába ütközhet használatuk.

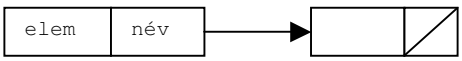
```
int r=4;
int* pr=&r;
int &rr=r;

pr++; - pointer megnövelése egyel

rr++; - mutatott érték megnövelése egyel

int &r1=1; - nem működik
```

Megoldás:

```
const int &r1=1; ≡ int(1);
```

Saját típusok definiálása:

```
struct nev{
 int elem1;
 nev* next; == 0 //Feltétel teljesül-e
};
```




```
struct Node{
 int elem;
 Node* left;
 Node* right;
};
```

A „;” karaktert kötelező kirakni a „}” után!

```
Node* tree=0;
```

```
void insert(Node* r, int e){
 r.elem=e; //csak akkor, ha már létezik az „e” változó
 if(r){
  insert(e<=r -> elem ? r -> left : r -> right; e); //pointereknél
 }else{
  r = new Node; //delete r;
  r -> left=r -> right=0;
  r -> elem=e;
 }
}
```

Ez papíron jó megoldásnak tűnik, a gyakorlatban azonban semmit sem fogunk látni belőle.

Megoldás:

```
void print(Node* r){
 f(r){
  print(r -> left);
  cout << r -> elem << „ „;
  print(r -> right);
 }
 print(tree);
}
```

Ez a megoldás már működni fog.

```
void insert2(Node* &r, int e){
 if(r){
  insert(e<=r -> elem ? r -> left : r -> right; e);
 }else{
  r = new Node; //delete r;
  r -> left=r -> right=0;
  r -> elem=e;
 }
}
```

A referencia miatt ez a megoldás is jól fog működni.

```
void destroy(Node* r){
 if(r){
  destroy(r -> left);
  destroy(r -> right);
  delete r; r=0;
 }
}

destroy(tree);
```

Ez az eljárás is jól működik, a „tree” üres fa lesz.

```
void add(int* p1, int* p2){
 (*p1) += (*p2);
}

void add2(int &r1, int &r2){
 r1 += r2;
}

int i=4;
```

add(&i, 2); – nem működik a konstans 2 miatt

Ahhoz, hogy az „add(i, 2);” működjön, az eljárásokat ki kell bővíteni a „const” kulcsszóval:

```
void add(int* p1, const int* p2){
 (*p1) += (*p2);
}

void add2(int &r1, const int &r2){
 r1 += r2;
}
```

Karakter-típus kezelése:

```
char t[10];
scanf(„%s”, t);
```

Ez a megoldás az első 10 karakterig működni fog, utána a függvény felülírja a közbülső eredményeket.

heap, new, delete – túlírható, túlolvasható, és ez rengeteg hibalehetőséget hordoz magában.

Példa a használatukra:

```
Node* a = new Node;
delete a;

Node* b = new Node[10];
delete[] b;
```

Standard Template Library (STL):

```
namespace std
{
 …
};
```

Tárolók:

string, vector, map, queue, deque

Típusdefiníciók:

```
 typedef int hossz;
```

typedef int hosszak[10]; – 10 elemből álló integer-típusú tömb

```
typedef void(*nev)(Node*, int);
nev n;
n=insert;
n(tree, 16);
```

ahol az „n” változó egy pointer

```
#include <stdint.h>
```

sys/

unsig int – 4 Byte x86 / x86 – 64 (architektúrafüggő)

signed long – 4 vagy 8 Byte (architektúrafüggő)

short 2 Byte

Unix/Linux rendszeren:

int 32_t

uint 32_t

int_t

Definiálás:

```
typedef int int32_t;

typedef long time_t; (time_h, size_t)
```

Minél több „typedef” utasítást használunk, annál inkább platform-függetlenné tehetjük vele programunkat.

```
int myfunc(const char* s, size_t len);

#ifdef__win32
…
#else
…
#endif
```

FÜGGVÉNYEK PARAMÉTER ÁTADÁSA ÉS ABSZTRAKT ADATSZERKEZETEK
(ELŐADÁSJEGYZET)

Paraméter-átadás:

I. Érték szerinti paraméter-átadás:

```
void f(int a, double v){
 …
}
```

A definícióban szereplő „a” és „v” változókat formális paramétereknek nevezzük.

Példa a függvény meghívására:

```
f(5, 3.14);
```

A függvény meghívásában szereplő változókat – melyek ebben az esetben az 5 és a 3.14 – aktuális paramétereknek nevezzük.

A formális paraméterek lokális változói lesznek a függvénynek, illetve az aktuális paraméterek belemásolódnak a formális paraméterekbe.

II. Cím szerinti paraméter-átadás: a formális paraméter átveszi az aktuális paraméter címét, így ha módosítjuk a formális paramétert, akkor az aktuális paraméter is módosul. Ebben az esetben aktuális paraméter csak címmel rendelkező objektum lehet.

III. Érték/eredmény szerinti paraméter-átadás: a formális paraméter felveszi az aktuális paraméter értékét, majd az alprogram [függvény] lefutása után a formális paraméter értéke visszamásolódik az aktuális paraméterbe.

IV. Név szerinti paraméter-átadás: működése függ az aktuális paraméter típusától. A következő esetek lehetségesek:

a. Konstans literál esetén érték szerinti paraméter-átadás történik;

b. Egyszerű típus esetén cím szerinti paraméter-átadás történik;

c. Függvény-hívás vagy összetett típusnál minden formális paraméterre való hivatkozáskor kiértékelésre kerül az aktuális paraméter;

C++ nyelvben ezenkívül létezik még az inicializálás szerinti paraméter átadás is:

```
int a=5;
void f(int a, double b){
 …
}

f(5, 3.14);
```

Értékcsere megvalósítása (rossz megoldás):

```
soid swap(int x, int y){
 int tmp=x;
 x=y;
 y=tmp;
}

int a=1;
int b=2;

swap(a,b);
```

Ez a megoldás azért rossz, mert a függvényből történő kilépés után a megváltoztatott értékek elvesznek.

Értékcsere megvalósítása (jó megoldás):

```
soid swap(int &x, int &y){
 int tmp=x;
 x=y;
 y=tmp;
}

int a=1;
int b=2;

swap(a,b);
```

Ez a megoldás azért működik, mert ezúttal az „x” és „y” változók csak referenciái, álnevei az „a” és „b” változónak, azonban hivatkozott memóriacím ugyanaz.

Absztrakt adatszerkezetek (class):

A C++ programozási nyelvben absztrakt adatszerkezeteket osztályok definiálásával hozhatunk létre. Az osztályon belüli adatokhoz háromféle hozzáférési szint létezik:

- public: a programon belül bárhonnan elérhető;
- protected: csak az osztályon belül, illetve annak származtatott osztályaiból érhető el;
- private: csak az osztályon belülről érhető el.

A „private” adatokhoz a „public” részben definiált függvényeken keresztül férhetünk hozzá.

Példa (dátum-típus):

```
class date{
 private.
  int year;
  int month;
  int day;
 public:
  void set_year(int y){ year = y; }
  void set_month(int m){ month = m; }
  void set_day(int d){ day = d; }
};
```

Egy „private” részben definiált függvény azonban így működik:

```
date&set_year(int y){ year = y; return *this; }
```

a „*this” hivatkozás mindig az aktuális objektumra mutat.

```
const date d(2008.11.15); - konstans dátum

d.get(year); - lekérdezi az évszámot

d.set_year(2005); - ez nem működik, mert a „d” dátum konstansként lett
definiálva!
```

A C++ nyelvben mind a függvényeket, mind az operátorokat túl lehet terhelni. Az operátorok is egyfajta függvényhívásként működnek.

```
++d ≡ d.operator++();

d++ ≡ d.operator++(0);
```

Globális műveletek az osztályon:

```
inline bool operator==(date d1, date d2){
 return !((d1<d2)||(d2<d1));
}
```

ÖSSZETETTEBB TÍPUSOK
(GYAKORLATJEGYZET)

Standard Template Library (Szabványos Könyvtár):

A Szabványos Könyvtár a következő fontos eszközöket tartalmazza:

- Tárolók;
- Algoritmusok;
- Funktorok (függvényként viselkedő típusok);
- Bejárók (memóriafoglalók).

Tárolók:

vector, queue, deque, list, [multi] map, [multi] set, stack

A multi-map és multi-set típusokban ugyanaz a kulcs többször is előfordulhat, egyébként pedig csak egyszer.

Tömbök használata:

```
std::string tomb[30];
std::string s;
int index=0;
while(cin >> s){
 tomb[index]=s;
 ++index;
 if(index==30){
  break;
 }
}
```

Az előre rögzített elemszám – tomb[30] – miatt ez a megoldás nem praktikus.

Normálisabb megoldás:

```
std::vector<std::string> tomb;
std::string s;
while(cin >> s){
 tomb.push_back(s);
}
```

A ciklus így is működik:

```
int index=0;
while(cin >> s){
 tomb[index]=s;
 ++index;
}
```

Kiíratás:

```
for(int i=0; i!=tomb.size; ++i){
 cout << tomb[i] << endl;
}
```

Iterátor használata:

```
std::vector<std::string>::iterator it;
for(it=tomb.begin(); it!=tomb.end(); ++it){
 cout << *it << endl;
}
int t[10]; { … };
std::iterator_traits<int>::iterator it;
```

iterator_traits == int*

Pointer-aritmetika:

```
*(it+4);

++it;

*it;
```

reverse_iterator:

… iterator p = find(tomb.begin(); tomb.end(); „almafa”);

ahol a „p” változó egy pointer.

Legyen tomb.end()=tomb.begin()+4 és p=tomb.begin()+4. Természetesen ekkor teljesül az alábbi feltétel:

p == tomb.end()

Ha a „tomb” egy vektor-típusú változó, akkor mind az elejéhez, mind a végéhez hozzáfűzhetünk további elemeket, de felül is írhatjuk a már meglevő értékeket, vagy kitörölhetjük azokat.

tomb.front(), tomb.back() – tömb eleje, illetve vége

t.insert(tomb.begin(), „almafa”); – az „almafa” elem beszúrása a tömb elejére

t.erase(tomb.begin(), tomb.begin()+4); – az első 5 elem kitörlése

map-típusú tároló:

```
map<int, std::string> m;
m[4]=”a”;
std::string s=m[2];
```

Ebben az esetben már működik az indexelés.

```
map<int, std::string>::iterator it;
```

*it – nem jó hivatkozás, mivel a „map” egy rendezett párt hozott létre.

```
it -> first;
it -> second;
```

Ez már megfelelő hivatkozás.

További példák:

```
template<class T1, class T2>
class std::Par{
 public:
  Par(){ … }
  T1 first;
  T2 second;
  //Konstruktor
  … pair(T1 f, T2, s) : first(f), second(s) { … }
  //Copy-konstruktor
  … pair(const pair &p) : first(p.first, second(p.second)) { … }
};

vector<string> v1, v2;
v1.push_back(„a”);
v2=v1;
vector<string> v3(v1);
vector<string> v4=v1;

map<int, string> m;
string s=m[2];

string()
```

Amennyiben már definiáltunk egy konstruktort, akkor az jön létre az alapértelmezett konstruktor helyett.

```
vector<int> u;
bool kisebb_mint_7(int x){
 return x<7;
}
iterator it=find_if(t.begin(), t.end(), kisebb_mint_7);
```

Ennek az algoritmusnak az a hátránya, hogy csak 7-re működik. Készítsük tehát el úgy, hogy bármilyen egész értékre működjön! Ehhez functor-okat kell alkalmazni.

```
class kisebb_mint{
 private:
  int x;
 public:
  kisebb_mint(int ertek) : x(ertek){ … }
  bool operator()(int y){
   return y<x;
  }
};
```

Ez a megoldás már minden egész értékre működni fog.

```
template<class key, class value, class cmp>
map{
 …
}
```

Deklaráció:

```
class string_less;

char*
```

Osztálydefiníció:

```
class string_less{
 public:
  bool operator()(const char*, const char*);
};

bool string_less::operator()(const char* s1, const char* s2){
 while(*s1&&*s2){
  if(*s1 >= *s2){
   return false;
  }
 }
 return true;
}
```

Modulok (#include <…>):

vector, list, queue, map, algorithm

Hasznos link:

www.cppreference.com

ABSZTRAKT ADATSZERKEZETEK (FOLYTATÁS)
(ELŐADÁSJEGYZET)

Stream-műveletek:

cout << „alma” << endl; – „alma” kiíratása standard outputra

cin >> n; – beolvasás „n” változóba a standard inputról

std::istream& operator>>(std::istream& is, date& d); – beolvasó operátor

std::ostream& operator<<(std::ostream& os, const date &d); – kiíró operátor

date d(2008.01.01);

cout << d; ↔ d.put(cout);

Verem-adatszerkezet:

A verem egy olyan adatszerkezet, mely lényegében egy tömb, melybe lehet berakni elemeket, illetve onnan kivenni is, de csak a legutoljára berakott elem vehető ki (a többi elem nem elérhető), és csak akkor lehet kivenni elemet, ha a verem nem üres.

A verem-típus működése egy rendkívül egyszerű példán keresztül lesz szemléltetve (reprezentáció dinamikus helyfoglalású tömbbel, egész értékekre deklarálva).

```
class stack{
 private:
  int max;
  int* t;
  int top;
  bool hiba;
  bool tomb;
 public:
  stack(){
   max=100;
   top=0;
   hiba=false;
   tomb=false;
  }
  void Empty(); //Verem kiuritese
  bool isEmpty(); //Ures-e a verem
  bool isFull(); //Tele van-e a verem
  push(const int e); //Elem berakasa
  pop(int &e); //Elem kivetele
  hibas(); //Volt-e hiba
};

void stack::Empty(){
 if(tomb==true){
  delete[] t;
  tomb=false;
 }
 top=0;
 hiba=false;
}

bool stack::isEmpty(){
 return top==0;
}
```

```
bool stack::isFull(){
 return top==max;
}

void stack::push(const int e){
 if(top<max){
  if(tomb==false){
   t = new int[max];
   tomb=true;
  }
  t[top]=e;
  ++top;
 }else{
  hiba=true;
 }
}

void stack::pop(int &e){
 if(top>0){
  --top;
  e=t[top];
  if(top==0){
   delete[] t;
   tomb=false;
  }
 }else{
  hiba=true;
 }
}

bool stack::hibas(){
 return hiba;
}
```

A fent definiált osztályban klasszikus értelemben vett kivételkezelés nincs, azt hogy a veremműveletek során volt-e hiba a bool-típusú „hiba” változóban kapjuk meg, melyet a szintén bool-típusú „hibas()” függvénnyel kérdezhetünk le.

A dinamikus helyfoglalású tömb létrehozása, illetve a számára lefoglalt hely felszabadítása automatikus: üres verem esetén nincs lefoglalva hely, ha pedig az üres verembe be akarunk rakni egy elemet, akkor a tömb számára automatikusan le lesz foglalva memóriaterület. Azt, hogy a verem üres-e az „isEmpty()” függvénnyel, azt pedig hogy tele van-e az „isFull()” függvénnyel kapjuk meg.

A „push(const int e)” eljárással berakunk egy elemet, a „pop(int &e)” eljárással pedig kiveszünk egy elemet, a kivett elemet az „e” változóban kapjuk meg – ide mindenképpen szükséges a referencia szerinti hivatkozás, ellenkező esetben a kivett érték elveszne.

A vermet az „Empty()” eljárással tudjuk kiüríteni, mely egyúttal a verem tartalmát reprezentáló tömb számára lefoglalt memóriaterületet is felszabadítja – feltéve, hogy a tömb számára már le lett foglalva memóriaterület, vagyis hogy a verem nem üres.

FELHASZNÁLÓI TÍPUSOK
(GYAKORLATJEGYZET)

Memória-felszabadítás:

free() – (C#)

delete[] – (C++)

Típusok létrehozása:

- Konstruktorok (létrehozzuk a tagfüggvényeket);
- Destruktorok (töröljük a tagfüggvényekhez rendelt adatokat);
- Tagfüggvények (a típuson belül deklarált függvények, melyek hozzáférnek a típus minden adatához);
- Tagváltozók (a típuson belül deklarált változók);
- Operátorok (a típussal kapcsolatos műveletek megvalósítása);

Struct-típusok:

A „struct” kulcsszóval megvalósított típusok adatainak default láthatósága public, vagyis – amennyiben nem adunk meg külön kikötést erre vonatkozóan – az itt definiált adatok a programon belül bárhonnan elérhetőek lesznek.

Dátum-típus megvalósítása:

```
struct date{
 int ev, honap, nap;
};
```

Class-típusok:

A „class” kulcsszóval megvalósított típusokban az alapértelmezett láthatóság private, vagyis ebben a típusban – megkötések nélkül definiált adatok – csak az osztály függvényein, operátorain és eljárásain keresztül lesznek hozzáférhetőek. Háromféle hozzáférési szintet különböztethetünk meg:

- Public (a programon belül bárhonnan elérhetőek az itt megadott adatok);
- protected (csak az osztályon belül, illetve annak származtatott osztályain belül érhetőek el az itt megadott adatok);
- private (csak az osztályon belül érhetőek el az itt megadott adatok);

ezen felül definiálhatunk úgynevezett barát-függvényeket, illetve operátorokat is, melyek megvalósítása történhet az osztályon kívül, azonban ezek hozzáférnek az osztály „private” adataihoz is. Ezeket a függvényeket, illetve operátorokat a „friend” kulcsszóval adhatjuk meg. Természetesen ez a struct-típusokban is működik.

Hogy mikor melyik hozzáférési szintet célszerű használni, azt a gyakorlatban a programunk hatékonysági tényezői határozzák meg.

Példák:

```
struct Date{
 int …
 Date(int ev, int ho, int nap);
};

Date::Date(int ev, … ){
 this -> ev=ev;
}
```

```
struct Date{
 int ev, ho, nap;
};

Date::Date(int ev_, int ho_, int nap_) : ev(ev_), ho(ho_), nap(nap_){}
```

Ez C++ nyelvben egy legális művelet. A kezdőértékek elhagyhatóak, sőt, a későbbi ellenőrzések miatt el is kell hagyni őket.

Konstruktor:

```
class Date{
 int ev, ho, nap //alapertelmezeskent private lesz
 public:
  enum Honap{jan=1, feb=2, … , dec=12};
  Date(int ev, Honap m, int nap) : ho = m;
  switch(m){
   class jan : if(day>31){ throw … };
   …
   class dec : if(day>31){ throw … };
  }
};
```

Destruktor:

```
struct Date{
 int …
 ~Date(){} //Ez egy destruktor
};
```

Egy struktúra destruktora mindig a „~” jellel kezdődik.

```
Date d(2008, Date::oct 22);
Date d1=d;
Date d2(d);
Date d3;
d3=d;
```

Copy-konstruktor:

```
Date(const Date &d) : ev(d.ev), ho(d.ho), nap(d.nap){}
Date d3;
```

Default konstruktor:

Date() – nincs paraméter

```
Date(int ev=2008, Honap m=oct, int nap=22);
```

```
class Date{
 Date &operator=(const Date &d){
  if(*this!=d){
   this -> nap=d.nap;
  }
  return *this;
 }
};
```

A „Date &operator=(const Date &d){” sorban egy már meglévő objektum módosítása található. Az operátor definiálása végén a „*this” objektummal kell visszatérni, amit például a következő esetek tehetnek indokolttá:

```
d1=d2=d3;

d1=(d2=d3);
```

Mindét eset egy láncolt értékadást szemléltet.

```
class Date{
 public:
  Date &operator+=(int napok){
   nap+=napok;
   if(nap>= … ){
    nap-=max; //az adott hónapra vonatkozó maximum
    ++ho;
   }
   return *this;
  }
};

date d;
d+=3;
Date d1=d+4;

Date operator+(const Date &d, int n){
 Date result=d;
 result+=n; //return result+=n;
 return result;
}
```

```
class T{
 public:
  T();
  ~T();
  T(const T&);
  T& operator=(const T&);
};
```

A „T()” akkor és csak akkor generálódik a fordító által, hogyha egyetlen konstruktort sem definiáltunk.

```
class T{
 refptr<T> create();
 private:
  T(const T&);
  T& operator=(const T&);
};
```

Lehet, hogy „T(const T&)” és „T& operator=(const T&)” soha nem kerül meghívásra, éppen ezért a törzsdefiníció nem kötelező.

```
glibmm, glib::object – függvénykönyvtárak

{
 reference();
 …
 unreference(); //ref_cnt==0 felszabadul
}

template<class T>
class refptr{
 T* object;
 public:
  refptr(T* a){
   this -> object=a;
   this -> object -> reference();
  }
  template<class T>
  refptr::refptr(const refptr& r){
   this -> object=r -> object;
   this -> object -> reference();
  }
  refptr::~refptr(){
   object -> unreference();
  }
  template<class t>
  refptr&refptr::operator=(const refptr& r){
   if(*this!=r){
    this -> object -> unreference();
    this -> object=r -> object;
    object -> reference();
   }
   return *this;
  }
  template<class T>
  T* refptr::operator->(){
   return object;
  }
};

T* a;
a -> osszead(4);

refptr<T>b= … ;
b -> osszead(4);

refptr<T> T::create(){
 T* a = new T();
 return refptr<T>(a);
}
```

ÖRÖKLŐDÉS
(ELŐADÁSJEGYZET)

Az öröklődés egy vagy több osztályból való származtatással történik, melyben a származtatott osztály megörökli a szülő-osztályok adatainak, függvényeinek és eljárásainak egy részét, a korábban kifejtett láthatósági korlátok figyelembevételével.

1. Példa:

```
class T{
 public:
  void f(){ … }
  void g(){ … }
};

class R : public T{
 void h(){ … }
};

R r;

r.f(); //Orokolve, tehat mukodik
r.g(); //Ez is orokolve, tehat ez is mukodik
r.h(); //”r” sajat eljarasa, tehat mukodik
```

Összefoglalva: az „R” osztály megörökli a „T” osztály eljárásait.

2. Példa:

```
class Ember{
 public:
  void eszik();
  void iszik();
  void alszik();
  string nev;
};

class Diak : public Ember{
 public:
  void tanul();
  string EHA;
};
```

A „Diak” osztály megörökli az „Ember” osztály eljárásait és változóját (nev), kiegészítve azt saját eljárásaival és változójával (EHA).

```
class Tanar : public Ember{
 public:
  void tanit();
};

class IgaziDiak : public Diak{
 public:
  void kocsmazik();
};
```

A gyerek-osztályokban felüldefiniálásokra is van lehetőség.

Emlékeztető – láthatóság:

- private: csak az adott osztályon belül elérhető adatok;
- public: mindenhonnan elérhető adatok;
- protected: csak az adott osztályon, illetve annak származtatott osztályaiból elérhető adatok;

Elnevezések (előző példa folytatása):

Diak rozsi, peti;

Diak: az osztály neve

rozsi, peti: a „Diak” osztály konkrét példányai

Tagfüggvények túlterhelése:

A C++ programnyelvben lehetőség van két vagy több egyforma nevű tagfüggvény deklarálására [definiálására] – vagyis túlterhelésére – amennyiben mindegyiknek más a paraméterezése (kötelezően kell lennie valamilyen különbségnek).

A tagfüggvények túlterhelése azonban nem működik megfelelően az öröklődés esetében, egy adott osztályon belül – ahol eredetileg is létrehoztuk őket, vagyis nem a származtatott osztályokban – viszont megfelelően működik.

Többszörös öröklődés:

A C++ nyelvben egy adott osztály nemcsak 1, hanem több szülő-osztálytól is örökölhet adatokat, ugyanez a JAVA nyelvben teljesen le van tiltva.

Példa:

```
class A{
 public:
  void foo(){ … }
};

class B{
 public:
  void bar(){ … }
};

class C : public A, public B{
 …
};

C c;
c.foo(); //Mukodik
c.bar(); //Ez is mukodik
```

Ezzel a konkrét példával nincs is probléma, a gondok akkor kezdődnek, hogyha a szülő-osztályok egyforma nevű adatokat (eljárások, függvények, változók) tartalmaznak. Diszjunkt osztályok esetén ez a probléma nem merül fel.

Struktúrák:

```
struct S{
 void f(){ … }
};

struct Q : public S{
 void g(){ … }
};

Q* q = new Q();

q->f(); //Mukodik
q->g(); //Ez is mukodik

S* s = new Q();

s->g(); //Meg ez is mukodik

Q* p = new S;

p->g(); //NEM JO!
```

Az utolsó sor már azért nem működik, mert az „S” struktúra nem tartalmazza a „g()” eljárást.

OPERÁTOROK
(GYAKORLATJEGYZET)

Vektor-típus:

```
Vektor;

RefVektor →   operator=( … );
              int operator[](int index);
              VektData;

struct Vektor{
 int *t;
 size_t size;
};

int& Vektor::operator[](int index){
 return t[index];
}

Vektor v;
//Feltoltes
int a=v[4];
v[4]=32; //Nincs ertekadas-operator-feluldefinialas!

void RefVektor::Set(int index, int ertek){
 if(data->t[index]!=ertek){
  //masol
  data->t[index]=ertek;
 }
}

struct RefVektor{
 VektData *data;
};

struct VektData{
 int *t;
 size_t size;
 size_t refert;
};

VektData *temp=data;
data = new VektData(temp);
--temp->refert;
if(!temp->refert){
 delete temp;
}

Vektor& Vektor::operator=(const Vektor& v);
                        +=
                        /=(const int i);
                        *=
                        *
```

Komplex-számok képzése:

```
class Complex{
 double re, im;
 // …
};

Complex c(4, 3);

Complex operator+(const Complex& a, const Complex& b){
 Complex ret(a.re+b.re, a.im+b.im);
 return ret;
 //return Complex(a.re+b.re, a.im+b.im);
}

++c;
c++;
c=c++;
```

Ha az operátor a „Complex” osztály egyik tagfüggvénye, akkor az operátor bal oldalán csak „Complex” típusú változó állhat, ellenkező esetben más adattípus is.

```
Complex operator+(int a, const Complex& b);
Complex operator+(const Complex &a, int b){
 return b+a;
}
```

Az utóbbi függvényben megtehetjük, hogy „b+a” értéket adjuk vissza, mivel a visszatérési értékben szereplő „+” operátor paraméterezésével kapcsolatos műveletet az előző függvényben már definiáltuk.

```
T operator@(const T&, const T&);
T& T::operator@=const(const T&);

a@b

operator+( … ){
 Complex ret(a);
 return ret+=b;
}
```

```
                              ++() // prefix _ forma
Complex& Complex::operator
                              ++(int) // postfix _ forma
{
 ths->re+=1;
 return *this;
}

Complex Complex::operator++(int){
 Complex ret=this;
 ++(*this); //Ronda, de mukodik
 return ret;
}
```

Template-k bevezetése:

```
Template<T>;
class std::auto_ptr{
 T* ap;
 public:
  T* operator->(){
   return ap;
  }
};

Complex *c = new Complex();
std::auto_ptr<(Complex) ap(c);
```

c->re

ap->re

ap.operator->()->re

```
double Complex::operator double(){
 return re;
}

void f(const Complex&);
void f(double);
```

c.operator()

c()

```
double Complex::operator()(){
 return re;
}
```

For_each használata:

```
std::vector<int> v;

v[i];

*it;
```

for_each algoritmus bevezetése:

```
class F{
 public:
  int sum;
  void operator()(int e){
   sum+=e;
  }
 F() : sum(0){}
};

F f; //Funktor

for_each(v.begin, v.end, f){ f.sum; }

std::map<kulcs, ertek , rendezes>
                      <

std::string
```

```
kulcs : char*
```

```
template< … >

class Nev{
 …
};

int f( … );
```

```
template<class T> //template<class T=int>**

class Vektor{
 T* tomb;
};

operator=

operator==

!=(const T& a, const T& b){ return !(a==b); }

T a;

a.f();

**

Vektor<double> dv;

Vektor<> iv;

template<class T, class u=typename T::tipus>

template< typename / class T=int>

class Vektor{
 T* tomb;
 typedef T ElemTipus;
}
```

ÖRÖKLŐDÉS ÉS VIRTUÁLIS TAGFÜGGVÉNYEK
(ELŐADÁSJEGYZET)

Struktúrák:

```
struct S1{
 S1() : a(0), b(0), c(0){}
 private:
  int a;
  int b;
  int c;
};

struct S2{
 S2(){ a=0; b=0; c=0; }
 private:
  int a;
  int b;
  int c;
};
```

Az első esetben (S1) a változók már eleve úgy jönnek létre, hogy megkapják az alapértelmezett 0 értéket, míg a második esetben (S2) előbb létrejönnek a változók, azután pedig megkapják a 0 értéket.

```
struct S3}
 S3(int p1, int p2) : a(p1), b(0), c(p2){}
 private:
  int& a;
  const int b;
  int& c;
};
```

Az S3 struktúra jól fog működni, mert a referencia-változók már eleve inicializáltan jönnek létre.

```
struct S4{
 S4(){ a=0; b=0; c=0; }
 private:
  int& a;
  const int b;
  int& c;
};
```

Az S4 struktúra hibás megoldás, mert referencia nem jöhet létre inicializálatlanul.

Virtuális tagfüggvények:

Virtuális tagfüggvényekre abban az esetben van szükség, amikor azt akarjuk, hogy egy származtatással létrehozott gyerek-osztály dinamikus-típusát megkapja a szülő-osztály statikus típusa, amennyiben a szülő-osztály ezzel kapcsolatos függvénye virtuális tagfüggvény (a példában látható módon):

Példa:

```
struct A{
 virtual void f(){ std::cout << „A::f()” << std::endl; }
 void g(){ std::cout << „A::g()” << std::endl; }
};
```

```
struct B : public A{
 /* virtual */ void f(){ std::cout << „B::f()” << std::endl; }
 void g(){ std::cout << „B::g()” << std::endl; }
};

A *a = new B(); //! A statikus tipusa: A*, dinamikus tipusa: B*
a->f(); //! A::f();
a->g(); //! B::f();

A b=B();

a.f(); //! A::f();
b.f(); //! B::f();

struct C : public B{
};

A* a = new C();
a->g();
```

Tisztán virtuális tagfüggvények:

```
struct P{
 virtual void foo()=0;
};

P p;
```

Ez a deklaráció nem működik, mert P-ből nem lehet konkrét objektumot létrehozni!

```
struct Q : public P{
 /* virtual */ void foo(){ std::cout << „Q::foo()” << std::endl; }
};

Q q; //Mukodik
P* p = new Q; //Ez is mukodik
```

- Egy osztály polimorfikus, ha van virtuális tagfüggvénye;
- A dynamic_cast csak polimorfikus osztályok esetén működik;
- Egy osztály absztrakt, ha van tisztán virtuális tagfüggvénye.

Virtuális tagfüggvény-hívás a konstruktorban:

A virtuális metódus-tábla csak az objektum felépítése után jön létre, így ha a konstruktorban virtuális tagfüggvényt hívunk meg, akkor az ugyanúgy fog viselkedni, mintha nem lenne virtuális.

```
struct Base{
 Base(){ f(); }
 virtual void f(){ std::cout << „Base::f()” << std::endl; }
};

struct Derived : public Base{
 Derived() : Base(){}
 virtual void f(){ std::cout << „Derived::f()” << std::endl; }
};
```

Konstruktorok explicit hívása:

Ha a szülő-osztálynak nincs default konstruktora (vagy pedig nem a default konstruktorral szeretnénk felépíteni), akkor a gyerek-osztály konstruktorának inicializáló listájában explicit meg kell hívni.

Virtuális destruktor:

Ökölszabály, hogy a polimorfikus osztályok destruktorának mindig virtuálisnak kell lennie.

SABLONOK
(GYAKORLATJEGYZET)

A sablonok (template) akkor tudnak nagyon hasznosak lenni, hogyha egy – már megírt és biztosan jól működő – osztályt szeretnénk általánosítani, hogy ne csak egy, hanem jóval több típusra is jól működjön (például a verem esetében: $D_v = integer \mid double \mid string \mid char \mid bool \mid \ldots stb$).

Sablonokat a következő módon lehet készíteni:

template< sablon-paraméterek >

Példák:

```
template< class T, class U>
T f(U u);

template< class T, class U>
T implicit_cast(U u){
 return (T) u;
}
```

Úgy is lehet konvertálni, hogy megadjuk mindkét típust, pontosan meghatározva, hogy mit mire akarunk konvertálni, például:

```
int a = implicit_cast<char, int>(32);

int b = implicit_cast(32);
```

Vektor-típusnál:

```
template<class T, int i>
class Vector{
 T elemek[i];
};

Vector<int, 16> a;

template<class T, int i>
Vector<T, i>::Vector()
                  ↑
                <T, i>
```

SWAP-eljárás:

```
void swap(int& a, int& b){
 int temp=a;
 a=b;
 b=temp;
}
```

Általános SWAP-sablon:

```
template<class T>
void swap(T& a, T& b){
 T temp=a;
 a=b;
 b=temp;
}
```

```
template<>
void swap(char& a, char& b){
 temp^=a;
 a^=b;
 b^=temp;
}
```

swap(x, y); $\left.\begin{matrix} short & x \\ int & y \end{matrix}\right\} T = int$

```
swap<int>((int) x, y);
```

```
std::vector<class T, … >

vector<bool> - 8-32 bit

bitset<4>
```

```
template<int size>
bitset{
 char tomb[size];
};
```

$$\frac{size}{8}+1$$

```
template<class T, class q = std::deque<T>>

std::queue
```

```
queue<std::vector<int>> a;
template<class T>
bool less(T a, T b){
 return a<b;
}
```

Ez pointerek esetén rosszul fog működni, mert a pointerek által mutatott memóriacím lesz összehasonlítva, nem pedig az általuk mutatott érték.

Megoldás:

```
template<class T>
bool less(T* a, T* b){
 return *a < *b;
}
```

Ez a megoldás már jól fog működni a pointerekre.

```
T tomb[ … ];
if(less(tomb[i], tomb[i+1]){
…
}

template<class T>
void sort(T* tomb, int size){
 …
 //less, eq, swap
}

vector v;
```

```
std::cout << v;

cout.operator<<(const vector&) → cout.operator<<(v);
ostream& operator<<(ostream& os, const vector& v);

std::operator<<(std::ostream);
```

A kiíró operátor megírásának hiányában kapunk egy képernyőnyi hibaüzenetet fordítás közben, amiből a következő sor a lényeges:

```
a.cpp: 3: instantied from here
```

Class-típusok, Struct típusok:

Emlékeztető: a class és struct adattípusok közötti különbség az alapértelmezett láthatóság. Class-típusnál ez private, struct-típusnál pedig public.

```
class Allat{ … };
class Kutya : public Allat{
 void kiir(const Allat& a){
  a.kiir();
 }
};
```

A gyerek osztályok függvényeinek / eljárásainak / operátorainak paramétereként a származtatott (szülő) osztály valamely objektumát is megadhatjuk, ezt szemlélteti a fenti példa.

```
class Allat{
 public:
  virtual void kiir(){
   std::cout << „Allat” << std::endl;
  }
};

class Kutya : public Allat{
 virtual void kiir(){
  std::cout << „Kutya” << std::endl;
 }
};

Allat a;
Kutya k;

a.kiir();
k.kiir();

void kiir(const Allat* a){
 a->kiir();
}

kiir(k);
```

A „virtual” kulcsszót az „Allat” osztály eljárásának definíciója elé mindenképpen ki kell tenni, különben a függvény nem fog megfelelően működni a származtatások esetén.

```
kiir(int i=4); //Hibas!
```

Virtuális függvénynek – gyakorlatilag – bármikor lehet kezdőértéket adni. A fenti példa azért nem működik, mert a meghívott „kiir(int i=4);” függvény nem virtuális tagfüggvény.

```
class Base{
 virtual void print()=0;
 public:
  virtual void print()=0;
};

void Base::print(){ … } //Tisztan virtualis tagfuggveny

class Derived{
 void print(){
  std::cout << „Derived” << std::endl;
 }
};

Base b; //Hiba!

Derived d; //Ez viszont mukodik.
d.print(); //Ez is mukodik.
```

```
class Derived{
 void print(){
  Base::print();
 }
};
```

SZABVÁNYOS KÖNYVTÁRI CSOMAG (STANDARD TEMPLATE LIBRARY)
(ELŐADÁSJEGYZET)

Szabványosítás:

A Standard Template Library (STL) 1998-ban lett szabványosítva és beépítve a C++ nyelvbe.

Három fő elemét különböztetjük meg:

- Tárolók;
- Algoritmusok;
- Iterátorok (az első kettőt kötik össze).

A C++ hamarosan megjelenő új szabványában – többek között – a Standard Template Library is ki lesz bővítve.

Tárolók:

A tárolóknak két típusa van:

- Szekvenciális tárolók;
- Asszociatív tárolók.

Szekvenciális tárolók:

- Lista: kétirányú láncolt lista;
- Vektor: gyakorlatilag ez egy tömb, mely abban az esetben ha betelik, akkor létrejön egy kétszer nagyobb tömb, melybe átmásolódnak az elemek, az újonnan beszúrandó elem pedig ennek a tömbnek kerül be a végére. A tömb kezdőméretét a konstruktorban lehet megadni;
- Deque: ez egy speciális tömb, mely lényegében egy átmenet az előző két adatszerkezet között. Előnye, hogy az elejéhez és végéhez konstans idő alatt tudunk új elemeket hozzáfűzni, illetve konstans idő alatt érjük el az elemeit is. Használata azonban komolyabb megfontoltságot igényel, mivel egy nagyméretű program esetében pillanatok alatt tele lehet vágni vele a memóriát szeméttel!

Adaptorok:

Ezek „nem valódi” adatszerkezetek, azonban ha az alapvető adattípusokból kapnak értéket, akkor annak interfészét úgy alakítják át, minta az a három felsorolt típus valamelyike lenne (meghívják a mögöttes adatszerkezetek megfelelő függvényeit).

- Queue (sor);
- Stack;
- Priority-queue.

Asszociatív tárolók:

- Halmaz (set): olyan elemeket tartalmaz, melyek a „<” matematikai reláció szerint összehasonlíthatók egymással. Megvalósítása keresőfa adatszerkezettel történik. Ebben a tárolóban minden elem csak egyszer szerepelhet;

- Zsák (multi-set): hasonló az előző pontban definiált tárolóhoz, azzal a különbséggel, hogy egy elem többször is szerepelhet a tárolóban, ennek megfelelően pedig a multiplicitás értéke is tárolva van;

- Map: kulcs-érték párokat tárol, melyekben a kulcsok a „<” matematikai reláció szerint összehasonlíthatóak egymással. Ez a típus úgy tud viselkedni, mintha egy tömb lenne, azonban ha nem létező elemre próbálunk meg hivatkozni, akkor az elem automatikusan létrejön a meghivatkozott kulccsal, tehát ezzel is vigyázni kell!

  Példa:

  ```
  map<string, int> m;
  m[„Peter”]=5;
  int x=m.find(„Dezso”)->second;
  ```

  Az „m.find(„Dezso”)” visszaadja a „Dezso” kulcsú elempárt (ha van ilyen), vagyis ez jelenti a megoldást a nem létező elemekre való hivatkozás elkerüléséhez;

- Multimap;

Algoritmusok:

```
for(int i=0; i<l; ++i){
 if t[i]==<keresett ertek> return true;
}
```

A tárolók iterátorokat is tartalmaznak, így az algoritmusok problémamentesen le tudnak futni a különböző típusokra.

Konstans iterátorral csak olvasni lehet a megfelelő adatszerkezetben, egyébként pedig értéket adni is.

Tárolók száma: 3+4 db (lásd: Tárolók). Algoritmusból jóval több van. Legismertebb: első előfordulás megkeresése (lineáris keresés).

STL intervallum:

Az Standard Template Library intervallumai a klasszikus értelemben vett, matematikai definíció szerint balról zártak és jobbról nyitottak.

Amennyiben pl. egy „s” tárolónál „s.end” egy iterátor értéke, akkor nyilvánvalóan már túlmentünk az „s” tároló tartalmán (pl.: ha egy lineáris keresésre nem volt találatunk, akkor ez lesz az iterátor értéke).

További példák:

```
template<typename T>
struct S{
 typedef int x;
}

template<>
struct S<bool>{
```

```
 static const int x=2;
};

S<double>::X *x; // ≡ int *x;
S<bool>::X *x; // ≡ 2*x;

template<typename U>
void f(){
 S<U> X *x; // ≡ 2*x;
 typename S<U> X *x;
}

std::pair
```

ÖRÖKLŐDÉS (FOLYTATÁS)
(ELŐADÁSJEGYZET)

Absztrakt és származtatott osztályok:

```
class Base{
 public:
  void f();
  virtual void g();
  virtual void h(int i)=0;
  Base(int); //Konstruktor
  virtual ~Base(); //Destruktor
};
```

A „Base” egy absztrakt osztály, mivel van egy tisztán virtuális tagfüggvénye: „virtual void h(int i)=0”. Absztrakt osztályból nem lehet konkrét példányt létrehozni, csak származtatással. Itt van szerepe az öröklődésnek.

```
class Derived : public Base{
 void f(); //Hiba!
 virtual void g(); // „virtual” kulcsszo nelkül is mukodik
 virtual void h(int); //OK
 virtual void h(short); //Hiba!
};

Derived::Derived(int a) : Base(a){
 …
};

class Derived2 : private Base{ … };
Derived2 b;

fv(&b); //Hiba!

Derived a(42);
fv(&a);

void f(Base *b);

Base::f;
Derived::g;

b->f(); //Az os-osztalyban levo f() fuggveny lesz.
b->g(); //Derived2 osztaly tagfuggvenye lesz.
```

```
class Derived2{
 private:
  Base *b;
  Derived2(){
   b = new Derived(3);
  }
};
```

Ős-osztály és származtatott osztály kapcsolata:

Ha egy osztályból származtatunk két másikat, majd azokból közösen egy negyediket, abból problémák lehetnek az azonos nevű függvények használata területén. Ugyanez vonatkozik az azonos nevű változókra is.

Ebben az esetben pontosan meg kell jelölni, hogy melyik osztálynak az adattagjára kívánunk hivatkozni, úgy, hogy a megfelelő osztály megnevezése, és „::” után írjuk a meghivatkozni kívánt adattagot.

Virtuális tagfüggvények öröklődésére ez a megkötés nem vonatkozik.

```
class Valami_Impl; //Nincs implementacio

class Valami{
 private:
  Valami_Impl *impl; //A pointer miatt mukodik
};

int Valami::f(){
 return impl->f();
}
```

patakino.web.elte.hu/pny2