Programozás alapjai 2

Nem OO újdonságok (1. előadás)

Deklaráció: szólunk h a cucc létezik   int fv(int z);
Definíció: megmondjuk, h hogy viselkedik / mi az   int fv(int z) {return 2*z;}

Bool ← új típus 0=hamis, több = igaz
const: típus módosító mely megtiltja a típus átírását
Referencia: egy változó alternatív neve   void inc(int& a) megfuttatható → inc(x);
↳ Olyankor olyan mintha a fgv nem csak a változó másolatát kapná meg.

There exists default paraméterek
also: f(), f(void) nincs paraméter
f(...) nem kell ellenőrizni a paramétereket
f(int a, int) nem fogjuk használni

inline fgv: olyan mint egy rendes fgv, csak bemásolja a fordító az adott helyre, így gyorsabb valami(?)

Overload: azonos nevű függvényből lehet több is, feltéve h az argumentumok mások
Balértékként fgv: Referencia értékű fgv lehet bal oldalon is

Névterek, függvénysablonok, dinamikus memóriakezelés (2. előadás)

∃ névterek melyekre a using namespace, vagy a ::-vel vonatkozhatunk
↳ using namespace: teljes névtér, vagy névtér részének láthatóvá tétele using namespace std::cout;
Ha biztosítani akarjuk, hogy egy kódrészlet csak az adott fordítási egységből legyen elérhető ⇒ névtér nélküli névtér
A névterek egymásba ágyazhatóak

cin>>ws eldobja az összes whitespacet
Sablonok Iroi:

```
template <typename T>
inline T max(T a, T b) {
    return a < b ? b:a;
}
```

a idáig tart a template értékére (a template-t követő definíció / deklaráció vége)

Alternatíva:

```
template < typename T, typename C>
T max(T a, T b, C cmp) {
    return cmp(a,b) ? b:a;
}
```

← ezzel fgv-t lehet átadni paraméterként

vagy teljes specializáció:

```
template <>
inline const char* max(const char* a, const char* b) {
    return strcmp(a,b) >0 ? a:b;
}
```

A fgvsablonok fordítási időben generálódnak

típusok: - végzetes
- nem végzetes

- meg kell állítani a programot
- helyben kezelhető
- a program más részében kezelhető
- elhalasztjuk (másra bízzuk) a döntést

```
double osztás (int y) {
    if (y==0)
        throw "Osztás nullával";    // const char* kivétel
    return (5.0/y);
}

int main () {
    try {
        cout << "5/2" =" << osztás (2) << endl;
        cout << "5/0 =" << osztás (0) << endl;
    }
    catch (const char* p) {        // elkapja ...ut y==0 const char* kivételt dob
        cout<< p<<endl;            // kivétel kezelése
    }
}
```

OO (3. előadás)

Az objektum testesíti meg a konkrét adatot és a rajta végezhető műveletet, egyedileg azonosítható, viselkedéssel és állapottal jellemezhető, felelőssége és jogköre van, a belső működés, adatszerkezet rejtve marad.
OO paradigmák: - egységbezárás (encapsulation): osztályok
- többalakúság (polymorphism: a műveletek paraméter/tárgy függőek)
- példányosítás (instantiation)
- generikus adatszerkezetek/algoritmusok

Osztály ≠ Objektum. Az objektum egy viselkedési osztály egy konkrét példánya
this pointer ≡ példányra mutató ptr ≠ erre a konkrét példányra
Konstans tagfüggvény: nem változtathatja meg az objektum állapotát

```
double getRe() const {return re;}
```

Automatikusan keletkező (implicitan deklarált) tagfüggvények:

- Konstruktor: létrehozza az objektumot implicit: Komplex() {}
- Destruktor: megszünteti az objektumot implicit: ~Komplex() {}
- Másoló konstruktor
- Értékadás
- Címképző s dereferáló operátorok

Paraméter nélküli default konstruktor, nem keletkezik implicitan, ha van legalább egy explicit
pl.: Komplex k1;

Az operátorok túlterhelhetőek, ha a bal oldal nem objektum, akkor globális fgv kell. Minden túlterhelhető, kivéve: . ; :: ; ?: ; sizeof

- operator++() → pre increment
- operator++(int) → post increment
- operator double() → cast (double)
- operator()() → fgv hívás

Konstruktor, értékadás, dinamikus szerkezetek, statikus tagok (4. előadás)

Inicializáló lista:
Valami (double c) {c1=c;} ⟺ Valami (double c) :c1(c) {} //konstans s ref. tag csak így inicializálható

A friend fgv.-ek hozzáférnek az osztály nem publikus változóihoz is (akár globális vagy más osztály fgv.-e is

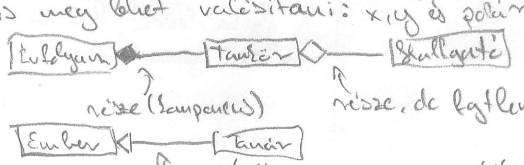
Dinamikus adatok esetén fontos a másoló konstruktor s az értékadó átdefiniálása!

Másoló konstruktorhoz referencia!

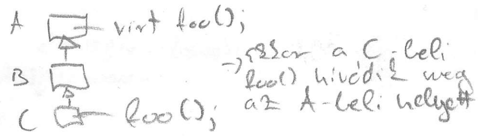
Az osztályban statikusan deklarált tagok nem példányosodnak, minden objektum ugyanazt a tagot éri el
Kívül pl.: bool String::ucase = false; // Definíció
Ha a statikus tag const int/enum típusú, akkor inicializálható az osztály deklarációjában is, ez a definíciót is kiváltja

Öröklés, virtuális tagfüggvény (5. előadás)

Osztály vs típus. A típus egy objektum-halmaz viselkedését specifikálja. Pl.: a komplex osztályt többféleképp is meg lehet valósítani: x,y és polár koordinátákkal is, belül különböznek, de kívülről azonosak



A típusú objektum kompatibilis B-vel, ha A típusú objektum bárhol s bármikor alkalmazható, ahol B használata megengedett. Pl.: állat ← madár ← veréb



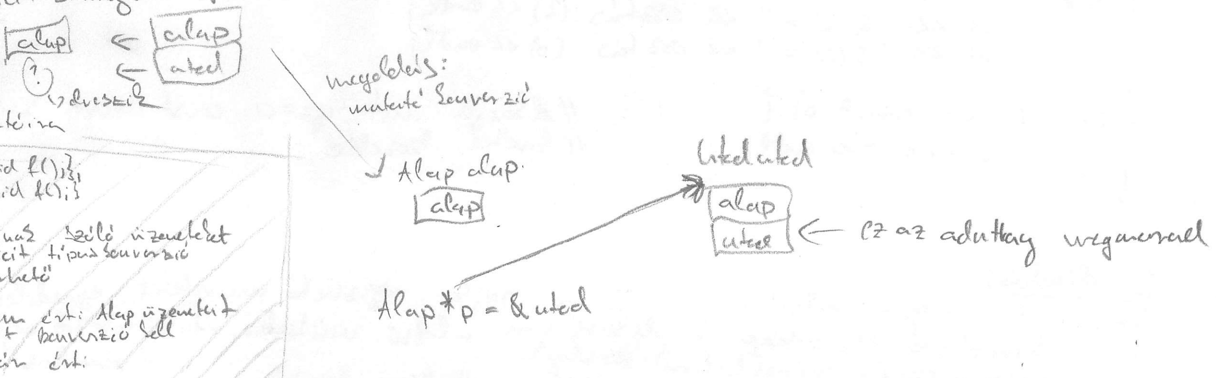
A leszármaztatás fele a protected szóval lehet gyengíteni a védelmet
Virtuális fgvt mindig az adott osztályból hívjuk (ha van)
Absztrakt alaposztályok, melyek csak a hierarchia kialakítása miatt léteznek
Nem példányosíthatók, tisztán virtuális fgv-ek: virtual void rajzol()=0;
Virtuális destruktor!!!!!! → A származtatott osztály destruktorát is lefuttatja.
Analitikus öröklés vs korlátozó öröklés: A korlátozó öröklés eltakarja az osztály egyes részeit pl.: class Négyzet: private Téglalap

Mutató konverzió, heterogén kollekció (6. előadás)

Az alaposztály konstruktorát illik hívni
Point3D(int x1, int y1, int z1): Point(x1,y1), z(z1) {}

Egyparaméteres konstruktorok egyben automatikus konverziót is jelentenek: String a = "hello"; → String a = String("hello");
Ennek kikapcsolása: explicit String(int i);

Slicing: Alap alap
Utod utod
alap = utod

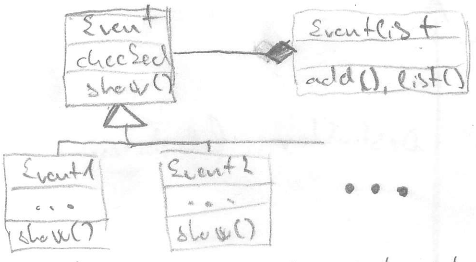


Konverzió alaposztály mutatóra

```
struct Alap {void f();};
struct Pub : public Alap { void f();};
struct Priv : private Alap { void f();}
Alap* p1B, * p2B;
Pub pub;          // pub kaphat Alapnak szóló üzeneteket
p1B = &pub;       // nem kell explicit típuskonverzió
p1B->f();         // alap o. f() elérhető
Priv priv;        // priv nem érti Alap üzeneteit
p2B = (Alap*)&priv; // explicit konverzió kell
p2B->f();         // így már érti
```

Kompatibilitás esetén a konverzió automatikus

Heterogén gyűjtemények: különböző típusú objektumokat egy közös gyűjteménybe tesszük, egységes kezelés valamely viselkedési kompatibilitásra építve. Egy öröklési hierarchiából származó objektumokat kezelünk heterogén szerkezetbe.



A virtuális destruktor és destruktora mindig meghívódik.

Generikus szerkezetek, template reloaded (7. előadás)

Sablonos osztály fgv-ek definíciója:

```
template <typename T>
T& Array <T>::operator [] (size_t i){
    return tp[i];
}
```

Sablonban is lehet default paraméter

Részleges vs teljes specializáció:

```
template <class R, class S, class T>
struct A  {...};
template <class S, class T>
struct A <char, S, T>  {...};

template <>
struct A <char, char, char>  {...};
```

A sablonparaméterek fgv-nél általában levezethetők, ha nem => ki kell írni.

A sablonok fordítási idejűek, ezt egy fordítási egységbe kell őket helyezni.

Predikátum sablonnal:

```
template <typename T, class S>
T leg1 (T a[], int n) {
    T tmp = a [0];
    for (int i=1; i<n; ++i)
        if (S:: select (a[i], tmp)) tmp=a[i]    // Az S osztálynak van egy select tagfüggvénye
    return tmp
}
```

Funktor: függvényként használható objektum. funktor osztály:

```
template <classT>
struct kisebbObj {
    bool operator () (const T& a, const T& b) { return a<b; }
};
```

leg-elem funktorral:

```
template <typename T, typename S>
T legElem (const T a[], int n, S sel){
    ...
    if ( sel (...))
    ...
}
```

Iterátorok (8. előadás)

Az iterátorok általánosított adatsorozat elemeire hivatkozó elemet mutató objektum. Legfontosabb műveletei:

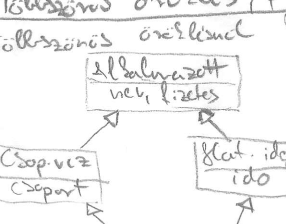
- Akt. elem elérése (*, ->)
- Következő elem elérése (++)
- Összehasonlítás (==, !=)
- Mutató objektum létrehozása az első s utolsó utáni elemre (begin(), end())

```
for (i=list.begin(); i!=list.end(); i++) {
    int x= *i;
    ...
```

Az iterátor miatt nem kell ismerni a tároló belső adatszerkezetét.

Többszörös öröklés, perzisztencia (8. előadás)

Többszörös öröklésnél külön figyelni kell arra, ha egy alaposztály leszármazottjai összetalálkoznak. "diamond" struktúra



Ilyenkor amikor inicializálnánk az objektumot akkor kétszer adnánk nevet s fizetést

nek: (mert meg kell hívni a CsopVez s a Hat.idejű konstruktorát is)

```
HatIdCsV (String n, double f, csop_t cs, time_t, t): CsopVez(n,f,cs), HatIdeju(n,f,t) {}
```

This is problem!

Ezért virtuális alaposztályt alkalmazunk, ez azt jelenti, hogy csak az öröklési lánc végén hívódik meg az alaposztály konstruktora

```
class CsopVez: virtual public Alkalmazott {...};
class HatIdeju: virtual public Alkalmazott {...};
class HatIdCsV: public CsopVez, public HatIdeju {
    HatIdCsV(String n, double f, csop_t cs, time_t t): Alkalmazott (n,f), CsopVez(NULL, 0, cs),
                                                       HatIdeju (NULL, 0, t) {}
};
```

A virtuális alaposztály adattagjai nem épülnek be a származtatott osztály adattagjaiba, hanem indirekt elérésűek lesznek. A származtatott osztály konstruktorait, nem az első származtatott osztály konstruktora fogja hívni, hanem az öröklési lánc legvégén szereplő osztály konstruktorai.

Konstruktor feladatai:
- öröklési lánc végén hívja a virtuális alaposztályok konstruktorait
- hívja a közvetlen nem virtuális alaposztályok konstruktorait
- létrehozza a saját részt:
  - -> beállítja a virtuális alaposztály mutatóit
  - -> beállítja a virtuális függvények mutatóit
  - -> hívja a tartalmazott objektumok konstruktorait
  - -> végrehajtja a programozott részt

Destruktor feladatai

A konstruktor NEM lehet virtuális, a destruktor viszont esetenként még kell is, h virtuális legyen.

dynamic_cast<T>(V)
  V - alaposztályra vonatkozik
  T - származtatott osztály vagy void pointer

```
struct A { virtual ~A(); };
struct B : A { };
struct C : A { };
A* apB = new B;
A* apC = new C;
B* bp1 = dynamic_cast <B*>(apB);
B* bp2 = dynamic_cast <B*>(apC);   //NULL
```

A dynamic_cast alap típusról konvertál, futási időben értékelődik ki, ha nem jó akkor NULL lesz.

A static_cast olyan mint a dynamic_cast de fordítási időben értékelődik ki.

const_cast<T>(V) a const vagy volatile minősítő előírására vagy eltávolítására használható.
reinterpret_cast<T>(V) ellenőrzés nélküli castol.

Perzisztens objektumok állapota elmenthető s visszatölthető később s akár másik gépen is. A visszatöltött objektum "folytatja" a működést.
  Saját állapotát képes kiírni egy adatfolyamra -> Szerializálható.
  Saját állapotát képes beolvasni egy adatfolyamról -> Deszerializálható.
A Serializable classt előre kell írni ... ... idő

Kivételkezelés reloaded, standard library (10. előadás)
Ha valamit lefoglalunk, fontos h hibakezeléskor is felszabadítsuk
A dobott kivétel alap & származtatott objektum is lehet.

```
try {
  throw E();
} catch(H) {
  // sg sg
}
```

A sg sg részhez akkor jutunk ha:
- H és E azonos típusú
- H báziosztálya E-nek
- Ha H és E pointer/referencia s teljesül rájuk a fentiek egyike

A dobás értékparaméterként dob ezért számolni kell az adatvesztéssel (alaposztályra konverzió)
Függvény deklarálásakor megadható h milyen kivételeket generálhat, ha mást generál => automatikusan meghívja az unexpected() handlert.

```
void f1() throw (E1, E2);   // csak E1, E2
void f2() throw();          // semmi
void f3();                  // bármi
void f4() noexcept (true);  // semmi (C++11-től)
```

A catch blokkból tovább tudjuk dobni az eredeti dolgot

```
try{
  throw E();
} catch (H){
  if (kezelhető) {...}
  else throw;
}
```

Ugyanaz dobódik tovább

Minden kivétel elkapása: catch(...)
Dobás után minden a blokkban deklarált objektum destruktora meghívódik.
A C++ STL: Általános célú újra felhasználható elemeket tartalmaz:
- tárolók
- algoritmusok
- függvények
- bejárók
- kivételek
- memóriakezelők
- adatfolyamok

Tárolók által definiált típusok:
- ↳ value_type, allocator_type
- -> reference, const_reference
- -> pointer, const_pointer
- -> iterator, const_iterator
- -> reverse_iterator, const_reverse_iterator
- -> difference_type
- -> size_type

∀ Asszociatív tárolókra:
- -> key_template

Funkciók
- ↳ konstruktor/destruktor
- -> értékadás: assign(), operator=
- -> iterator példányosítás: begin(), end(), rbegin(), rend()
- -> méret lekérdezés: size(), max_size(), capacity(), empty()
- -> módosítók: insert(), erase(), clear(), swap()

∀ sorozattároló kivéve listen:
- -> elemek elérése: front(), back(), operator[], at()
- -> módosítók: push_back(), pop_back()

∀ asszociatív tárolókra:
- -> count(), find(), lower/upper_bound(), equal_range()

vector <T, Alloc>
↳ Speciális műveletek: • capacity()
• reserve()
• resize (n), resize (n, val)

→ Nincs: • push_front, pop_front
• asszociatív op.

list <T, Alloc>
↳ Speciális műveletek: • merge (list), merge (list, bin_pred)
• remove (val)
• remove_if (un_pred)
• resize (n), resize (n, val)
• sort (), sort (bin_pred)
• splice (p, list), splice (p, list, first), splice (p, list, first, last)
• unique (), unique (b-pred)

→ Nincs: • at(), operator[]()
• asszociatív op.

deque <T, Alloc>
Kétvégű sor
↳ Speciális műveletek: • resize (n), resize (n, val)
→ Nincs: asszociatív op.

stack <T, deque>
Elrejti a deque nem verem stílusú műveleteit

queue <T, deque>
Elrejti a deque nem sor stílusú műveleteit

priority_que <T, vector, Cmp>
Prioritási sor, alapvetően a < operátorral hasonlít.

map <Key, T, Cmp, Alloc>
Asszociatív tömb. kulcs - érték párt tárol, alapértelmezés szerint <-el hasonlít

pair <const Key, mapped_type>
map bejárásakor párok sorozatát kapjuk.

set < Key, Cmp, Alloc>
Halmaz.

STL algoritmusok (11. előadás)

A tárolók nem csak tárolják hanem kiértékelik is az elemeket.

Algoritmusok típusai:
- → Nem módosító sorozatműveletek
- → Sorozatmódosító műveletek
- → Rendezés, rendezett sorozat műveletei
- → Halmazműveletek
- → Kupacműveletek
- → Minimum, maximum
- → Permutációk

count_if: megszámolja hány tagra igaz a predikátum
adjacent_find: visszaadja az első adatra mutató pointert aminek a rákövetkező adatra igaz volt a predikátum, az utolsó pointert adja vissza ha nem volt ilyen
mismatch: két tárolóban megnézi meddig igaz rájuk a predikátum, ha nem az visszaadja őket egy párként

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};
struct Valami : public unary_function <int, bool> {
    ......
};
Valami :: argument_type ...
Valami :: result_type ...
```

Lekötők:
Ha egy fgv-hez két argumentum kell, de csak egyet adunk meg: akkor ezt lekötővel oldhatjuk meg
count_if(v, v+8, bind2nd(less<int>(), 11)) ← kisebb mint 11

Backtrack algoritmusok (12. előadás)

Példa probléma: 8 királynő elhelyezése a sakktáblán
Megoldás: Elkezdjük elhelyezgetni a királynőket egyesével a 8 oszlopban, ha az egyiknél már nincs több hely akkor lépjünk vissza az előző oszlopra
Rekurzióval addig próbálkozunk amíg ki nem jön