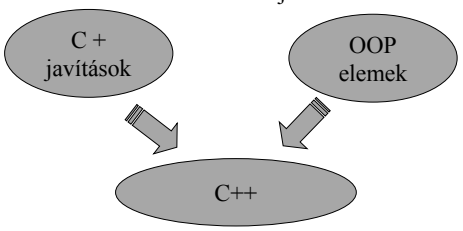
Programozás alapjai II.
(13. ea) **C++**

Év végi összefoglalás

Szeberényi Imre
BME IIT
<szebi@iit.bme.hu>

C++ kialakulása

Veszélyforrások csökkentése — Objektum orientált szemlélet



Javítások egy része átkerült az ANSI C-be.

C Javítások, bővítések

- Struktúranév típussá válik
- Csak preprocesszorral megoldható dolgok nyelvi szintre emelése (const, enum, inline)
- Kötelező prototípus, névterek
- Referencia, cím szerinti paraméterátadás
- Többarcú függvények (overload)
- Alapértelmezésű (default) argumentumok
- Dinamikus memória nyelvi szint. (new, delete)
- Változó definíció bárhol

Névterek, scope operátor

- A moduláris programozás támogatására külön névterületeket definiálhatunk.
- Ezek neveire (azonosítóira) a hatókör (scope) operátorral (::), vagy a using namespace direktívával hivatkozhatunk.

```
namespace Proba {
  int alma;
  float fv(int i);
  char *nev;
}
```

```
Proba::alma = 12;
float f = Proba::fv(3);
```

```
using namespace Proba;
alma = 8; float f = fv(3);
```


Objektum

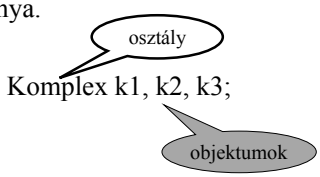
- OBJEKTUM: konkrét adat és a rajta végezhető műveletek megtestesítője
- egyedileg azonosítható
- viselkedéssel és állapottal jellemezhető
- felelőssége és jogköre van
- képes kommunikálni más objektumokkal
- a belső adatszerkezet, és a műveleteket megvalósító algoritmus rejtve marad
- könnyen módosítható
- újrafelhasználható
- általánosítható

OO paradigmák

- egységbezárás (encapsulation)
  - osztályok (adatszerkezet, műveletek összekapcsolása)
- többarcúság (polymorphism)
  - műveletek paraméter függőek, tárgy függőek (kötés)
- példányosítás (instantiation)
- öröklés (inheritance)
- generikus adatszerkezet alapú megoldások

Class

- Objektum osztály ≡ objektum fajta, típus (viselkedési osztály)
- Osztály ≠ Objektum
- Objektum ≡ Egy viselkedési osztály egy konkrét példánya.



Létrehozás, megsemmisítés, értékadás

- Konstruktor // létrehozás
  - default: X() // nincs paramétere automatikusan létrejön, ha nincs másik konst.
  - másoló: X(const X&) // referencia paramétere van, automatikusan létrejön: meghívja az adattagok és ősök másoló konst.-rát, ha obj. egyébként bitenként másol.
- Destruktor // megsemmisítés
  - automatikusan létrejön: meghívja az adattagok és ősök destruktorát
- operator=(const X&) // értékadó operátor automatikusan létrejön: meghívja az adattagok és ősök értékadó operátorát, ha objektum, egyébként bitenként másol.

Konstruktor és destruktor

KONSTRUKTOR: Definíció és inicializálás összevonása.
DESTRUKTOR: Az objektum megszüntetése.

```
class Komplex {
   double re, im;
 public:
   Komplex() { }      // konstruktornak nincs típusa
   Komplex(double r, double i)      { re = r; im = i; }
   ~Komplex()   {}    // destruktornak nincs paramétere
   ...
};
Komplex k1;
Komplex k2 = k1;     // másoló (copy) konstruktorral
Komplex k3 = Komplex(1.2, 3.4);
```

ez az alapértelmezés

ideiglenes objektum

Inicializálás miért más mint az értékadás?

- A kezdeti értékadáskor még inicializálatlan a változó (nem létezik), ezért nem lehet a másolással azonos módon kezelni.
- Mikor hívódik a másoló konstruktor?
  - inicializáláskor (azonos típussal inicializálunk)
  - függvény paraméterének átadásakor
  - függvény visszatérési értékének átvételekor
  - ideiglenes változók összetett kifejezésekben
  - kivétel átadásakor

Operátorok kiértékelése

- ha a bal oldal osztály, akkor meg kell vizsgálni, hogy van-e megfelelő alakú tagfüggvény
- ha nincs, vagy beépített típus, de a jobb oldal osztály, akkor
- meg kell vizsgálni, hogy van-e megfelelő alakú globális függvény

Mikor kell globális függvény?

- ha a bal oldal nem osztály (pl. 2 + X)
- vagy nincs a kezünkben (pl. cout << X)

A védelem enyhítése: friend

```
class Komplex {
  double re, im;
public:  .....
  Komplex operator+(const Komplex& k)
        { Komplex sum(k.re + re, k.im + im); return(sum); }
  Komplex operator+(const double r)
        { return(operator+(Komplex(r))); }
  friend Komplex operator+(const double r, const Komplex& k)
        { return(Komplex(k.re + r, k.im)); }
  friend ostream& operator<<(ostream& s, const Komplex& k)
        { s << k.re << ',' << k.im << 'j'; return(s); }
};
```

Így láncolható

```
cout << k1 << k2;
```

Tagok inicializálsa és a const fv.

```
class Komplex {
   double re, im;
 public:
   Komplex(double r = 0, double i = 0) : re(r), im(i) { }
   double Re() const { return(re); }
   double Im() const { return(im); }
   double Abs() const;
};
double Komplex::Abs() const
{
        return(sqrt(re*re + im*im));
}
```

Tagok konstruktorának hívása

Nem változtathatja meg az állapotot (adatokat)

Konstans és referencia tag

- A konstans és a referencia kötelezően inicializálandó.
- Az inicializálást a konstruktor inicializáló listáján kell elvégezni, kivéve ha static.

```
class Valami{
   static const double pi = 3.14;
   const double szorzo;
   int &ref;
 public:
   Valami(double x, int& r) :ref(r), szorzo(x) {....}
};
```

Baj lenne, ha nem ref. lenne?

Enum konstansok helyett

- Gyakori, hogy enum-ot használunk konstansok helyett
- Ha kívülről is el kell érni, akkor ugyanúgy publikussá kell tenni.

```
class Valami{
        enum { c0, c1, c5 = 5, c8 = 8 };
public:
        enum { jobbra, le, balra, fel };
....
};
```


Statikus tag

- Az osztályban statikusan deklarált tag nem példányosodik.
- Pontosan egy példány létezik, amit explicit módon definiálni kell (létre kell hozni).
- Minden objektum ugyanazt a tagot éri el.
- Nem szükséges objetummal hivatkozni rá.
  pl: String**::**SetUcase(true);
- Statikus tagként az osztály tartalmazhatja önmagát.
- Felhasználás: globális változók elrejtése

Statikus tagfüggvény

- Csak egy példányban létezik.
- Statikus tagfüggvény nem éri el az objektumpéldányok nem statikus adattagjait.

```
class A {
   int a;
public:
   static void f() { cout << a;}
};
```

Nem éri el!

Alapértelmezett tagfüggvények

- Konstruktor
  - default: X() // nincs paramétere
  - másoló: X(const X&) // referencia paraméter
- Destruktor
- operator=(const X&) // értékadó
- operator&() // címképző
- operator,(const X&) // vessző

A konstruktor/destruktor és az értékadó operátor alapértelmezés szerint meghívja az adattagok és ősök megfelelő tagfüggvényét. Ha azonban saját függvényünk van, akkor abban csak az történik, amit beleírtunk. Kivéve a konstruktor/destruktor alapfunkcióit (default létrehozás, megsemmisítés.)

Saját konstruktor és értékadás

```
void Pr(char *p) { cout << p << endl; }
class A {
public:
  A() { Pr("AK"); }
  A(const A&) { Pr("AC"); }
  void operator=(A&) {Pr("A="); }
};
class B {
  A a;
public:
  B(const B& b) _____{Pr("BC"); }
  void operator=(B& b) {
                _____ Pr("B="); }
};
```

Default hívódott meg

:a(b.a)

a = b.a;

```
B b1;         // AK
B b2 = b1; // AKBC
b1 = b2;    // B=
```

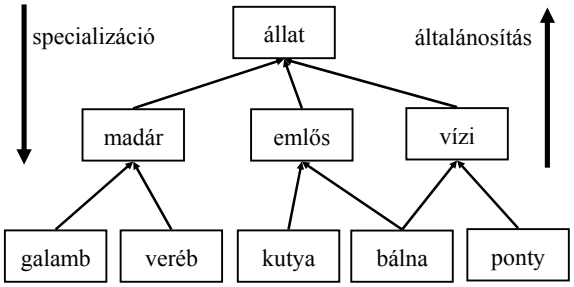
Értékadásnál nem hívódott meg a tartalmazott osztály értékadó operátora!

Ügyelni kell a tartalmazott objektumok és ősök helyes létrehozására/értékadására!

Öröklés

- Az öröklés olyan implementációs és modellezési eszköz, amelyik lehetővé teszi, hogy egy osztályból olyan újabb osztályokat származtassunk, melyek rendelkeznek az eredeti osztályban már definiált tulajdonságokkal, szerkezettel és viselkedéssel.
- Újrafelhasználhatóság szinonimája.
- Nem csak bővíthető, hanem a tagfüggvények át is definiálhatók.

Kompatibilitás/2



Védelem összefoglalása

	külső	származtatott	tagfüggvény és barát
public:	√	√	√
protected:		√	√
private:			√

Mutatókonverzió

- Mutatókonverzió = rejtett objektumkonverzió
- Kompatibilitás: öröklés
  - kompatibilis memóriakép
  - kompatibilis viselkedés (tagfüggvények)

class Base { .... };
class PublicDerived : public Base { .... };
class PrivateDerived: private Base { .... };

Konstruktor feladatai

- Öröklés 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 törzset

Lehet-e konstruktorban virt. fv?

- Nem lehet sem virtuális konstruktor, sem virtuális fv. a konstruktorban.
- Pontosabban fv. lehet, de nem az történik, amit várunk. Az alaposztály konstruktora a virtuális függvénypointerek beállítása előtt lefut.

```
class A { public:
   A() { f(); }
   virtual void f() { cout << "f:A\n";}
};
```

```
class B :public A { public:
   B() { }
   void f() {  cout << "f:B\n"; }
};
```

Kiírás: f:A

Destruktor feladatai

- Megszünteti a saját részt
  - végrehajtja a programozott törzset
  - tartalmazott objektumok destruktorainak hívása
  - virtuális függvénymutatók visszaállítása
  - virtuális alaposztály mutatóinak visszaállítása
- Hívja a közvetlen, nem virtuális alaposztályok destruktorait
- Öröklés lánc végén hívja a virtuális alaposztályok destruktorait

Lehet-e destruktorban virt. fv?

- Lehet, de a működés hasonló.
- Virtuális destruktor használata dinamikus tag esetén viszont nagyon fontos.

```
class A {
   char *p;
public:
   A() { ... }
   virtual ~A() { delete p; }
};
```

```
class B :public A {
   char *p;
public:
   B() { ... }
   ~B() { delete p; }
};
```

```
A *pa = new B;
delete pa;
```

ha nem virtual, akkor nem hívódik meg ~B()

Szabványos könyvtár (STL)

Általános célú, újrafelhasználható elemek:
- tárolók, majdnem tárolók
- algoritmusok
- függvények
- bejárók
- memóriakezelők
- adatfolyamok

http://www.sgi.com/tech/stl/
http://www.cppreference.com/cppstl.html
http://www.cplusplus.com/reference/stl/

Tárolók (konténerek)

- Tetszőleges adatok tárolására
- Sorban, vagy tetszőleges sorrendben érhetők el az adatok.
- Tipizált felületek

```
vector<int> i1(10, -3);
vector<double> d1(100);
i1[8] = 12; i1.at(9) = 13;
for (vector<double>::size_type i = 0;
        i < d1.size(); i++) d1[i] = 3.14;
for (vector<int>::iterator i = i1.begin();
        i < i1.end(); i++) cout  << *i << endl;
```


Szabványos tárolók

- vector
- list
- deque
- stack
- queue
- priority_queue

- map
- multimap
- set
- multiset

- string
- array
- valarray
- bitset

Tárolókhoz tartozó műveletek

Bejárók műveletei:
- begin()
- end()
- rbegin()
- rend()

Hozzáférés műveletei:
- front()
- back()
- []
- at()

```
vector<int> iv(10, -1);
cout << iv.front();
cout << iv.back();
int i = iv.[2];
```

```
iv.back() = 7;
iv.front() = 12;
iv[12] = 8;    // nincs ellen.
iv.at(12) = 8; // hibát dob
```

```
vector<int>::reverse_iterator rit = iv.rbegin();
while (rit != iv.rend()) cout << *rit++; // ++ visszafelé!!!
```


Tárolókhoz tartozó műveletek/2

Verem és sorműveletek:
- push_back()
- pop_back()
- push_front()
- pop_front()

void !

Listaműveletek:
- insert(p, x)
- insert(p, n, x)
- insert(p, first, last)
- erase(p)
- erase(first, last)
- clear()

ez is iterator: int*

```
iv.push_back(23);
iv.push_back(88);
int i = iv.back();
iv.pop_back();
iv.erase(iv.begin());
```

```
int v[] = {1, 2, 3};
iv.insert(iv.begin(), v, v+3);
```

iterator

Tárolókhoz tartozó műveletek/3

Egyéb műveletek:
- size()
- empty()
- max_size()
- swap()
- get_allocator()
- ==, !=
- <, <=, >, >=,

Konstruktorok, destr.:
- container()
- conatiner(n)
- container(n,x)
- container(first, last)
- container(x)
- ~container()

```
int v[] = {1, 2, 3}; vector<int> iv(v, v+3);
vector<char>cv(10, '*'); vector<int> iv2(iv);
```


Tárolókhoz tartozó műveletek/4

Értékadások:
- operator=(x)
- assign(n,x)
- assign(first, last)

Asszociatív műveletek:
- operator[](k)
- find(k)
- lower_bound(k)
- upper_bound(k)
- equal_range(k)
- key_comp()
- value_comp()
- count()

```
vector<int> iv, iv1;
iv.assign(10, 0);
iv1.assign(iv1.begin(),
           iv1.end());
```

```
map<double, int> big; big[2.3] = 8; big[3.4] = 13;
map<double,int>::iterator bi = big.lower_bound(3);
```


vector<T, Alloc>

Speciális műveletek:
- capacity()
- reserve()
- resize(n), resize(n, val)

Nincs:
- push_front, pop_front
- asszociatív op.

```
int v[] = {0, 1, 2};
vector<int> iv(v, v+3);
iv.push_back(3);
iv.at(5) = 5; // hiba
```

```
iv.resize(7, -8);
iv.at(5) = 5; // nincs hiba
Print(iv) // 0,1,2,3,-8,5,-8,
```

```
template <typename T> void Print(T& t) {
  for (T::iterator i = t.begin(); i != t.end(); i++)
      cout << *i << ", ";
  cout << endl; }
```


list<T, Alloc>

Speciális műveletek:
- merge(list),
- merge(list, bpred)
- remove(val)
- remove_if(upred)
- resize(n), resize(n, val)
- sort(), sort(bpred)
- splice(p, list)
- splice(p, list, first)
- splice(p, list, first, last)
- unique(),unique(bpred)

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

```
list<int> il(2, -3);
il.push_front(9);
il.push_back(2);
il.sort();
Print(il); // -3, -3, 2, 9,
il.unique();
list<int> il2(3, 4);
il.merge(il2);
Print(il); // -3, 2, 4, 4, 4, 9,
```

deque<T, Alloc>

Kétvégű sor
Speciális műveletek:
- resize(n), resize(n, val)

Nincs:
- asszociatív op.

```
deque<int> dq;
dq.push_back(6);
dq.push_front(9);
Print(dq); // 9, 6,
dq.resize(6, -3);
Print(dq); //9, 6, -3, -3, -3,-3,
```

```
dq.back() = 1;
Print(dq); // 9, 6, -3, -3, -3, 1,
dq[2] = 2;
Print(dq); // 9, 6, 2, -3, -3, 1,
dq.at(3) = 0;
```

```
if (!dq.empty())
        Print(dq); // 9, 6, 2, 0, -3, 1,
```


stack<T, deque>

Elrejti a kétvégű sor nem verem stílusú műveleteit.

Műveletek:
- empty()
- push()
- pop()
- top()
- stack()
- stack(cont)

```
stack<int> s;
s.push(1);
s.push(2);
s.push(3);
s.top() = 4;
s.push(13);
```

```
while (!s.empty()) {
   cout << s.top() << ", "; s.pop();
} // 13, 4, 2, 1,
```


queue<T, deque>

Elrejti a kétvégű sor nem sor stílusú műveleteit.

Műveletek:
- empty()
- push() –> push_back()
- pop() –> pop_front()
- front()
- back()
- queue(), queue(cont)

```
queue<int> q;
q.push(1);
q.push(2);
q.push(3);
q.back() = 4;
q.push(13);
```

```
while (!q.empty()) {
   cout << q.front() << ", "; q.pop();
} // 1, 2, 4, 13,
```


priority_queue<T, vector, Cmp>

Prioritásos sor. Alapesetben a < operátorral hasonlít.

Műveletek:
- empty()
- push()
- pop()
- top()
- priority_queue()

```
priority_queue<int> pq;
pq.push(1);
pq.push(2);
pq.push(3);
pq.push(-2);
pq.push(13);
```

```
while (!pq.empty()) {
   cout << pq.top() << ", "; pq.pop();
} // 13, 3, 2, 1, -2,
```


map<Key, T, Cmp, Alloc>

Asszociatív tömb
- (kulcs, érték) pár tárolása
- alapértelmezés szerint < operátorral hasonlít
- map maga is összehasonlítható

```
map<string, int> m;
m["haho"] = 8;
m["Almas"] = 23;
cout << m["haho"] << endl;
cout << m["Almas"] << endl;
map<string, int>::iterator i = m.find("haho");
```


pair<const Key, mapped_type>

Párok
- map bejárásakor párok sorozatát kapjuk
- A kulcsra first, az értékre second mezővel hivatkozhatunk

```
map<string, int> m;
m["haho"] = 8; m["Almas"] = 23; m["xx"] = 13;
map<string, int>::iterator p;
for (p = m.begin(); p != m.end(); p++) {
   cout << p->first << ": ";
   cout << p->second << ", ";
} // almas: 23, haho: 8, xx: 13
```

set<Key, Cmp, Alloc>

Halmaz
- olyan map, ahol nem tároljuk az értéket
- alapértelmezés szerint < operátorral hasonlít
- map-hoz hasonlóan összehasonlítható

```
set<long> s;
s.insert(3); s.insert(3);
s.insert(7); s.insert(12); s.insert(8);
cout << s.count(6) << endl;  // 0
cout << s.count(3) << endl;  // 1
set<long>::iterator i = s.find(3);
Print(s); // 3, 7, 8, 12,
```


Algoritmusok általánosítása

- Sablonok segítségével az algoritmusok általánosíthatók. Általános működés ún. predikátummal módosítható.
- (Sablon)paraméterként egy eljárásmódot (függvényt) is átadhatunk.
- Példa: Írjunk egy általános kiválasztó algoritmust, ami képes kiválasztani a legkisebb, legnagyobb, leg... elemet.

Sablonparaméterrel

```
template<class T, int n, class S> T keres(T t[n]) {
   T tmp = t[0];
   for (int i = 1; i < n; i++) if (S::select(t[i], tmp)) tmp = t[i];
   return tmp;
}
```

```
template<class T> struct Min {        // szokásos min. kereséshez
   static bool select(T a, T b) { return a < b; }
};

template<class T> struct Max {        // szokásos max. keresésez
   static bool select(T a, T b) { return a > b; }
};
```


Függvényparaméterrel

```
template<class T, class S> T keres2(T t[], int n, S sel) {
   T tmp = t[0];
   for (int i = 1; i < n; i++) if (sel(t[i], tmp)) tmp = t[i];
   return tmp;
}
template<class T> struct Min2 {
   bool operator()(T a, T b) { return a < b; }
};
template<class T> bool kisebb(T a, T b) { return a < b; }

cout << keres2<int >(tomb, 5, Min2<int>());
cout << keres2<int>(tomb , 5, kisebb<int>);
```

Függvényobjektum

Példány

STL Algoritmusok

- accumulate()
- adjacent_find()
- binary_search()
- count(), count_if()
- find(), find_if()
- for_each()
- make_heap()
- max(), max_element()
- merge()
- min(), min_element()
- mismatch()
- next_permutation()
- nth_element()
- partial_sort()
- pop_heap(),
- push_heap()
- random_shuffle()
- remove_if()
- search_n()
- sort(), sort_heap()
- swap(), swap_ranges()
- transform()
- unique()
- upper_bound()

Algoritmus példák/1

```
int v[] = {10,20,30,30,20,10,10,20};
int *ip = adjacent_find (v, v+8);    // a pointer is iterator!
vector<int> iv(v, v+8);
vector<int>::iterator it
it = adjacent_find(iv.begin(), iv.end()); // első ismétlődés
it = adjacent_find (++it, iv.end());      // második ism.

cout << count_if(v, v+8, bind2nd(less<int>(), 11));
```

predikátum

lekötő

hasonlító függvény

Algoritmus példák/2

```
bool odd(int i) { return i&1; } ....

int v[] = {10,20,15,35,92}; vector<int> iv(v, v+5);
make_heap(iv.begin(), iv.end());
Print(iv);  // 92, 35, 15, 10, 20,
sort_heap(iv.begin(), iv.end());
Print(iv);  // 10, 15, 20, 35, 92,

vector<int>::iterator it =
                    remove_if(iv.begin(), iv.end(), odd);
iv.erase(it, iv.end());
Print(iv);  // 10, 20, 92,
```

Print: 36. slide-ról

Algoritmus példák/3

```
int pow(int i) { return i*i; } ....

int v[] = {1,2,5,7,10}; vector<int> iv(v, v+5);
transform(iv.begin(), iv.end(), iv.begin(), pow);
Print(iv);  // 1, 4, 25, 49, 100,

iv.pop_back(); iv.pop_back();
do {
    Print(iv);
} while (next_permutation(iv.begin(), iv.end()));
```

```
1, 4, 25,
1, 25, 4,
4, 1, 25,
4, 25, 1,
25, 1, 4,
25, 4, 1,
```

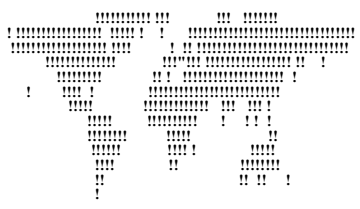

Nem csak C++-t tanultunk!

- OO tervezés
  - dolgok szereplők számbavétele
  - viselkedésük modellezése
  - újrafelhasználhatóság, generikusság
- OO modell leírása
- Fejlesztő környezet
- Dokumentálás
- pici UNIX

A forma is fontos !

```
          main(l
        ,a,n,d)char**a;{
     for(d=atoi(a[1])/10*80-
    atoi(a[2])/5-596;n="@NKA\
   CLCCGZAAQBEAADAFaISADJABBA^\
   SNLGAQABDAXIMBAACTBATAHDBAN\
   ZcEMMCCCCAAhEIJFAEAAABAfHJE\
   TBdFLDAANEfDNBPHdBcBBBEA_AL\
    H E L L O,    W O R L D! "
      [l++-3];)for(;n-->64;)
         putchar(!d+++33^
              l&1);}
```


WhereAmI 47 19



UNIX-szal kezdtük a félévet

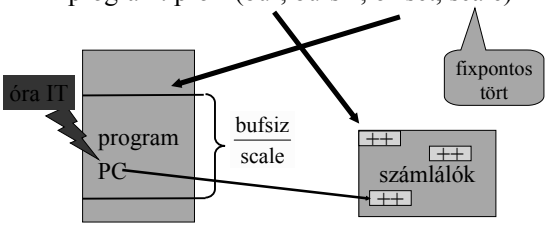
- Néhány gondolat a UNIX-os fejlesztőeszközökről befejezésül:
- Számtalan eszköz, többek által ma már elavultnak tekintett, ugyanakkor hatékony, egységes parancssoros felülettel.
- Kiemelt a szövegfeldolgozás/elemzés hatékony támogatása.
- Egyre több next, next, next, finish program, a PC-s változatokban, ami nem mindig jelent előnyt.

Fejlesztést segítő eszközök

- make
- sccs, rcs, cvs, svn
- prof
- lex
- yacc
- awk
- perl
- .....

Futási idő analízis

- A profil() rendszerhívás segítségével megmérhető, hogy hol, mennyi időt töltött a program: profil(buf, bufsiz, offset, scale)



Futási idő analízis/2

- A futás végén a buffer számlálóit kiírva a statisztika feldolgozható (prof, gprof).
- Megfelelő starup és exit kódot linkelve ez automatikussá tehető (cc -p, gcc -xpg).
- A függények prológ kódját módosítva a függvényekbe való belépések száma könnyen mérhető.

gcc –pg –o pogram program.c

Futási idő analízis/3

Ekkor a program futtatásakor keletkezik egy (g)mon.out, amint a gprof programmal lehet feldolgozni: pl: (g)prof program

```
...
%time  cumsecs  #call  ms/call  name
  61.5     0.08      1    80.00  _vege
  38.5     0.13    500     0.10  _xyz
   0.0     0.13      1     0.00  _exit
   0.0     0.13      1     0.00  _main
   0.0     0.13      1     0.00  _printf
   0.0     0.13      1     0.00  _write
```


lex (csak gondolat ébresztőként)

- Lexikai analizátor generátor
- Reguláris kifejezésekkel megadott lexikai elemek felismeréséhez C programot generál.
- Önállóan is felhasználható (–ll), de legtöbbször beépítik egy másik programba.

yacc (csak gondolat ébresztőként)

- Compiler generáló eszköz környezetfüggetlen nyelvhez.
- A nyelvtani szabályokból előállítja a nyelvtant felismerő C programot.
- Önállóan is használható (-ly), de legtöbbször beépítik másik programba.

Péda: római számok konverziója

```
%token RDIG
%{
  int last = 0;
%}
%%
list:
  | list '\n'
  | list number '\n' {
    printf("-> %d\n", $2); last = 0; }
  | list error '\n'  { yyerrok; };

number: RDIG    { last = $$=$1; }
  | RDIG number {
    if ($1 >= last) $$ = $2 + (last=$1);
    else            $$ = $2 - (last=$1); };
```


romailex.l

```
%}
   extern int yylval;
%}
%%
I  { yylval=   1; return RDIG; }
V  { yylval=   5; return RDIG; }
X  { yylval=  10; return RDIG; }
L  { yylval=  50; return RDIG; }
C  { yylval= 100; return RDIG; }
D  { yylval= 500; return RDIG; }
M  { yylval=1000; return RDIG; }
[^IVXLCDM]     { return(yytext[0]); }
```


Köszönöm a figyelmet