

Limbaje formale și tehnici de compilare

Curs 14: implementarea unor facilități avansate:
metode polimorfice, funcții cu context,
managementul memoriei

Facilități avansate ale LP

- ▶ Aceste facilități automatizează crearea unui cod destul de complex, care altfel ar fi trebuit implementat manual
- ▶ (1) Facilități avansate care se implementează prin generare de cod specific:
 - Metode polimorfice
 - Funcții cu context (closures)
 - Excepții
- ▶ (2) Facilități avansate care necesită o bibliotecă mai complexă de funcții care să o implementeze:
 - Managementul automat al memoriei
 - Încărcare dinamică de module
 - Metaprogramare și accesul la funcțiile compilatorului

Metode polimorfice (MP)

- ▶ Metodele polimorfice au implementări diferite în clase diferite
- ▶ Există un mecanism de selecție a implementării corespunzătoare obiectului pentru care se apelează metoda
- ▶ Ele se folosesc pentru implementarea colecțiilor eterogene
- ▶ Exemple: figuri în programe CAD, entități într-un joc, produse cu structură diferită într-un inventar, etc

[illegible]

Implementarea clasei abstracte

- ▶ Orice clasă care are MP va avea un membru ascuns, denumit "**tabelă de selecție**" (*dispatch table* sau *virtual table*) – **vTable**
- ▶ vTable este un pointer la o structură (VTable) ce conține pointeri la funcții
- ▶ Fiecare pointer din VTable corespunde unei MP

```
struct VTable{                                     // C
    double (*arie)(Figura *fig);                  // fig ⇔ this
    double (*perimetru)(Figura *fig);
}VTable;
```

```
struct Figura{
    VTable *vTable;
};
```

Implementarea claselor concrete

- ▶ O clasă concretă va avea toate câmpurile clasei de bază și propriile sale implementări ale MP

```
struct Cerc{ // C
    VTable *vTable; // moștenită din Figura
    double r;
};
```

```
double Cerc_arie(Figura *fig){ // fig ⇔ this
    double r=((Cerc*)fig)->r;
    return PI*r*r;
}
```

```
VTable Cerc_vTable={Cerc_arie, Cerc_perimetru};
```

Instanțierea claselor concrete

- ▶ La crearea unei instanțe (new), se setează vTable-ul instanței cu tabela ce cuprinde implementările MP pentru clasa respectivă

```
// Java: Cerc c=new Cerc();
```

```
Cerc *Cerc_new(){ // C
    Cerc *c=(Cerc*)malloc(sizeof(Cerc));
    c->vTable=Cerc_vTable; // vTable pentru un Cerc va
    // pointa la implementarile specifice cercului
    c->r=0;
    return c;
}
```

Apelul MP

- ▶ La apelul unei MP se selectează din vTable-ul obiectului curent pointerul la funcția corespunzătoare MP
- ▶ Se apelează funcția pointată dându-i-se ca prim argument obiectul curent (this), pentru ca funcția să aibă acces la obiect

```
// Java: fig.arie();
```

```
Figura *fig;
```

```
double a=fig->vTable->arie(fig);
```

```
// C
```

Funcții cu context (closures)

- ▶ O funcție cu context este funcție creată în altă funcție și care păstrează (captează, închide (close)) variabilele locale necesare (contextul) din funcția părinte, chiar și după ce funcția părinte s-a încheiat
- ▶ Deoarece fiecare apel al funcției părinte creează alte variabile locale, contextele vor fi diferite

```
require('console') // JavaScript + Node.js
function salutare(expresie){
  return function(ume){
    return expresie+' '+ume+'!';
  }
}
const f1=salutare("Salut");
const f2=salutare("Bine ai venit");

console.log(f1("Ion")); // Salut Ion!
console.log(f2("Ana")); // Bine ai venit Ana!
```


Tipul universal Value

- ▶ Considerăm cazul unui LP dinamic, în care orice valoare este reprezentată folosind tipul universal Value
- ▶ Toate funcțiile vor avea aceeași declarație. Valoarea returnată va fi Value, iar parametrii vor fi:
 - un context (vector de Value), folosit de funcțiile cu context, altfel NULL
 - argumentele de la apel (vector de Value)
 - numărul de argumente

```
enum ValType {ValNothing,ValNb,ValStr,ValClosure,...};
struct Value{
    ValType type;           // tip universal, care poate conține orice tip de date din LP
    union{
        double nb;          // pentru numere
        const char *str;     // pentru șiruri de caractere
        struct{              // o funcție cu context este și ea o valoare
            Value *ctx;       // contextul funcției
            Value (*fn)(Value *ctx,Value *args,int nArgs); // pointer la funcție
        }closure;
    };
};

// toate funcțiile din LP au această declarație generică
Value f1(Value *ctx,Value *args,int nArgs);
```

Implementarea funcției părinte

- ▶ În funcția părinte toate variabilele care sunt folosite în funcțiile cu context produse de ea, sunt depuse într-un vector alocat dinamic
- ▶ Pentru accesul la aceste variabile se va folosi doar vectorul respectiv. Astfel se îndeplinesc două cerințe:
 - Variabilele vor exista și după terminarea funcției părinte
 - Atât funcția părinte cât și funcțiile cu context vor accesa aceleași valori

```
Value salutare(Value *ctx, Value *args, int nArgs){  
    Value *ctx=(Value*)malloc(1*sizeof(Value));  
    ctx[0]=args[0]; // expresie  
    Value ret; // creare funcție cu context  
    ret.type=ValClosure;  
    ret.closure.ctx=ctx;  
    ret.closure.fn=anonymousFn1;  
    return ret;  
}
```

Implementarea funcției cu context

- ▶ În funcția cu context, dacă se folosește o valoare captată din funcția părinte, ea se preia din contextul primit
- ▶ Pentru apelul unei funcții cu context se folosește contextul care a fost deja creat de funcția părinte
- ▶ Pointerul Value.closure.fn a fost și el setat în funcția părinte cu adresa funcției de apelat

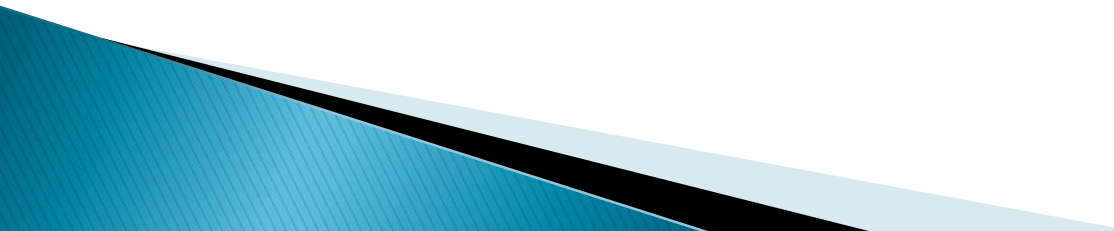
```
// return expresie+' '+nume+'!';
Value anonymousFn1(Value *ctx, Value *args, int nArgs){
    return ctx[0]           // expresie
        .add(' ')
        .add(args[0])       // nume
        .add('!');
}
```

```
Value f1=salutare(NULL,{"Salut"},1); // creare funcție cu context
f1.closure.fn(f1.closure.ctx,{"Ion"},1); // apel: f1("Ion");
```

Clasificare management memorie (MM)

- ▶ **Manual** – programatorul eliberează explicit memoria folosind funcții gen *free(ptr)*
- ▶ **Semiautomat** – LP furnizează construcții care se apelează automat la ieșirea dintr-un bloc și eliberează resursele obiectelor care ies din domeniul de existență. Exemple: *destructori*, blocuri *using*, *pointeri inteligenți* (smart pointers – *unique_ptr*, *shared_ptr*)
- ▶ **Automat** – LP testează când o zonă de memorie nu mai este folosită de program și o eliberează automat

MM automat

- ▶ Metode de implementare:
 - ▶ **Numărarea referințelor** – (reference counting) fiecare zonă alocată are un contor care memorează câți pointeri pointează la ea. Dacă acest contor devine 0, zona este eliberată.
 - ▶ **Trasare** – (tracing) alocatorul urmărește (trasează) folosirea zonelor de memorie, pornind de la cele despre care știe sigur că sunt folosite. Orice zonă la care nu poate ajunge prin trasare înseamnă că este inaccesibilă și va fi eliberată.
- 

MM prin numărarea referințelor

- ▶ La alocarea unui obiect: $nRefs = 0$
- ▶ La atribuirii (variabile, argumente, câmpuri): $nRefs++$
- ▶ Când un deținător dispăre: $nRefs--$
- ▶ La eliberarea unui obiect, se decrementează referințele câmpurilor lui

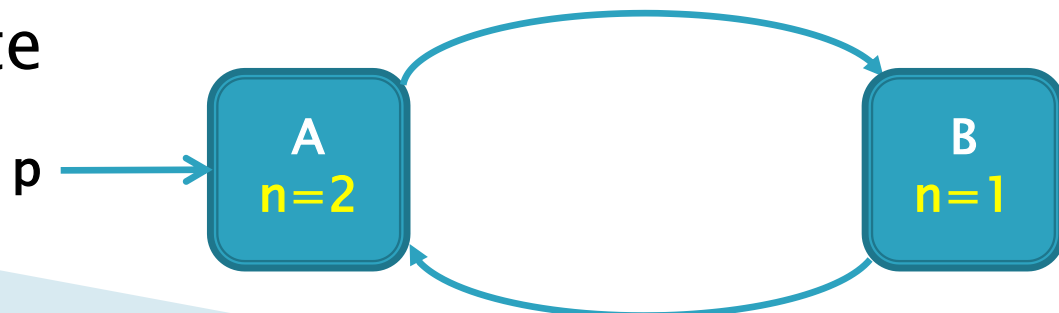
```
struct Obiect{  
    unsigned nRefs;    // nr referinte  
    // ... alte campuri ...  
};
```

```
function f(){  
    var o1=new Obiect();  
    var o2=o1;  
}
```

```
tmp=malloc(sizeof(Obiect)); // new Obiect();  
tmp->nRefs=0;  
o1=tmp;                      // var o1=tmp;  
o1->nRefs++;  
o2=o1;                       // var o2=o1;  
o2->nRefs++;  
if(--o1->nRefs==0)free(o1); // iesirea din f  
if(--o2->nRefs==0)free(o2);
```

MM prin numărarea referințelor

- ▶ Avantaje:
 - Simplu de implementat
 - Resursele sunt eliberate în momentul în care nu mai sunt folosite
- ▶ Dezavantaje:
 - Referințele ciclice (un obiect referă direct sau indirect la el însuși) nu pot fi eliberate
 - Viteză mai mică a programului, din cauza operațiilor de incrementare/decrementare/testare
 - Pentru multithreading, operațiile sunt mai complexe
 - Necesită memorie suplimentară pentru stocarea numărului de referințe



MM cu trasare (tracing) – algoritm

1. **Marcarea zonelor folosite (mark)** – se pleacă de la variabile (globale, locale) și argumentele funcțiilor în desfășurare și se marchează zonele pointate de ele ca fiind folosite.

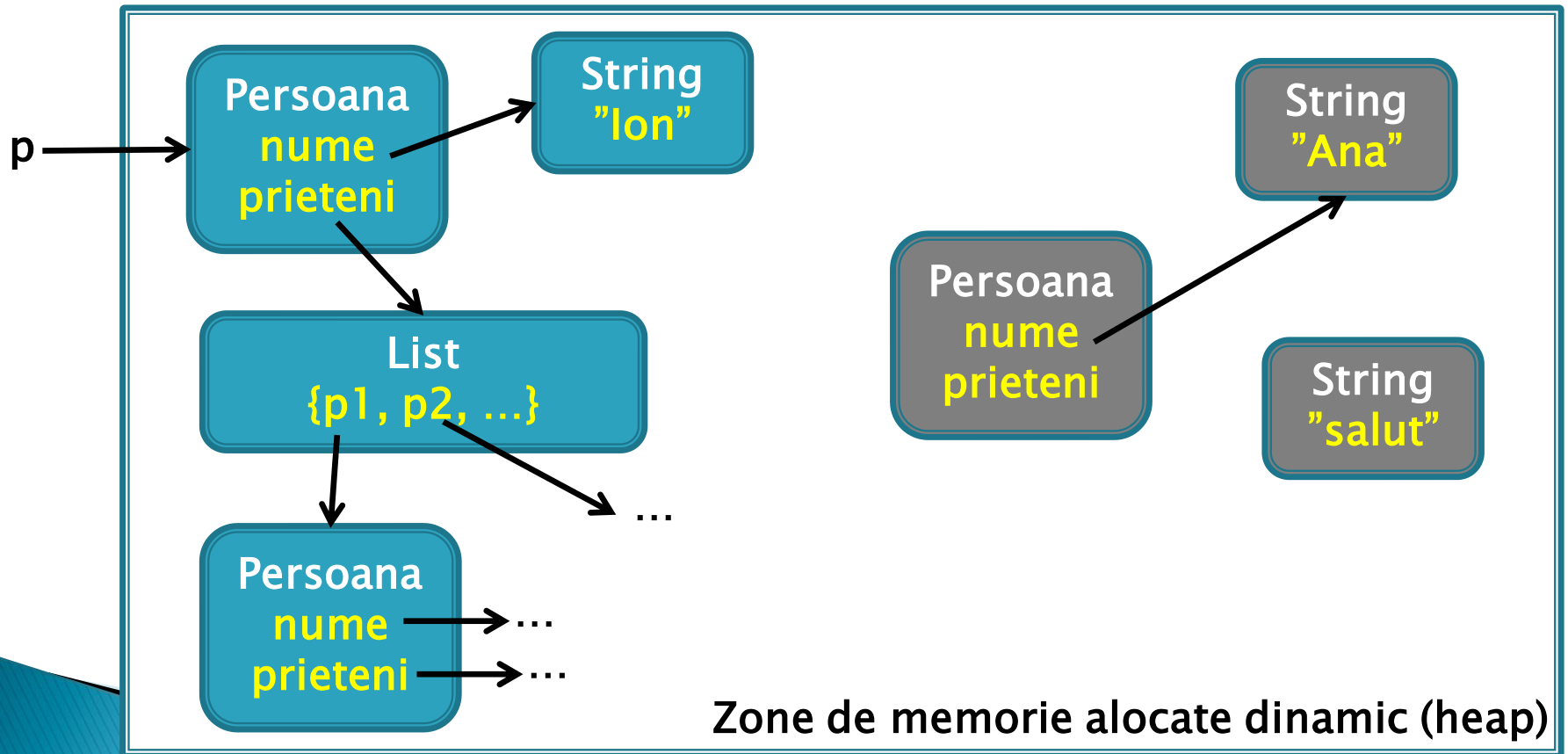
Aceste puncte de pornire se numesc **rădăcini** (roots). Se consideră că doar acestea pointează la zone folosite în program (in use).

Se continuă recursiv cu câmpurile obiectelor marcate până când nu mai este nimic de marcat.

2. **Se parcurge toată memoria alocată și se eliberează zonele care nu au fost marcate anterior (sweep)**

MM cu trasare – exemplu

- ▶ Cu turcoaz sunt zonele care pot fi atinse (reachable) pornind de la variabile/argumente din program. Aceste zone sunt utile și vor fi menținute.
- ▶ Cu gri sunt zonele care nu pot fi atinse (unreachable). Aceste zone nu mai sunt utile și vor fi eliberate.



MM cu trasare – îmbunătățiri

- ▶ **Compactare** – (compacting) în timp, datorită alocărilor și eliberărilor de diverse dimensiuni, apar zone de memorie libere de dimensiuni din ce în ce mai mici între zone alocate. Aceste zone nu vor mai putea fi refolosite ulterior și duc la epuizarea memoriei. Prin compactare, se reorganizează memoria pentru a elimina aceste goluri.
- ▶ **Alocare pe generații** – (generational) pentru a nu se parcurge de fiecare dată toată memoria, alocările se organizează în zone distincte, după tiparul lor de folosire: alocare/eliberare **frecventă** sau **persistentă**. La reclamarea memoriei, prima oară se verifică doar zona frecventă și, doar dacă nu se recuperează suficientă memorie, se va verifica și zona persistentă.
- ▶ **Multithreading** – se folosesc mai multe fire de execuție pentru a se reclama memoria în paralel cu funcționarea programului propriu-zis.
- ▶ **Incremental** – reclamarea memoriei are loc în pași distincți (incrementi) în care se parcurg doar părți mici de memorie, pentru a nu apărea stopări prelungite ale programului

MM cu trasare

▶ Avantaje:

- Se eliberează și referințele ciclice
- Necesarul de memorie este mai mic decât la MM cu numărare de referințe, deoarece nu se mai stochează contorul
- Când nu are loc reclamarea memoriei, nu există operații suplimentare, ca la MM cu numărare de referințe

▶ Dezavantaje:

- Momentul când se face reclamarea memoriei nu este predictibil, deci nu se poate folosi pentru eliberarea resurselor critice (fișiere, porturi de rețea, ...) în momentul în care ele nu mai sunt folosite
- Reclamarea memoriei poate dura destul de mult timp (chiar pentru îmbunătățiri gen incremental + generational + multithreading), ceea ce nu este admisibil pentru aplicații cu limite stricte de timp