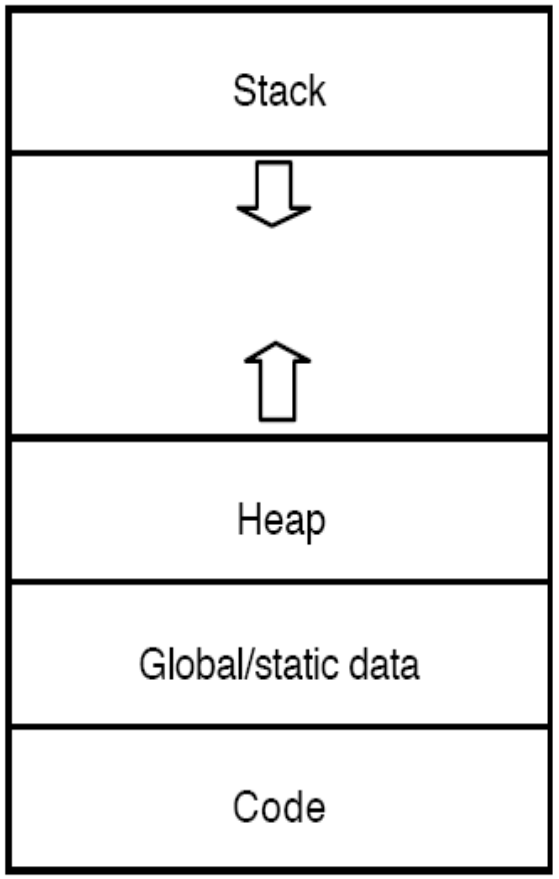
Compilatoare

Apeluri de functii

Generarea codului orientat obiect

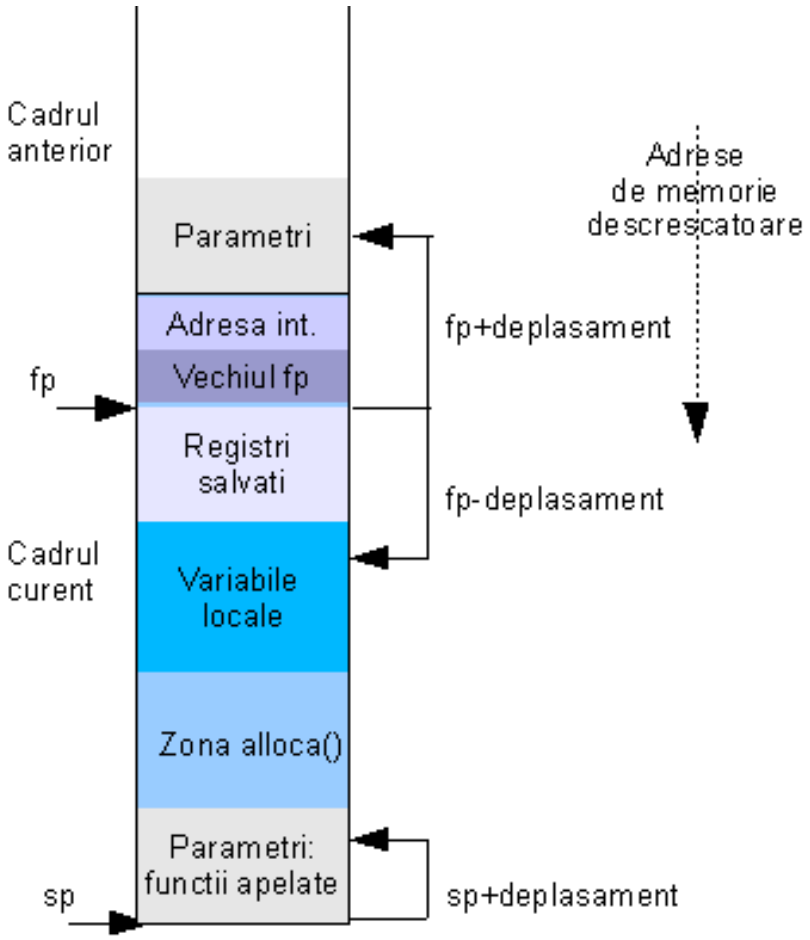
Memoria la runtime

- Variabilele locale, parametrii – in cadrul de stiva local
  - sectiune: .stack
  - adresare indexata: sp+offset, fp-offset
- Variabilele globale, statice – in 'memorie' (alocate static)
  - sectiuni: .data, .bss
  - constante, valori initiale: .rodata
  - adresare directa
- Alocarea dinamica (new, malloc) – in heap
  - adresare: pointer

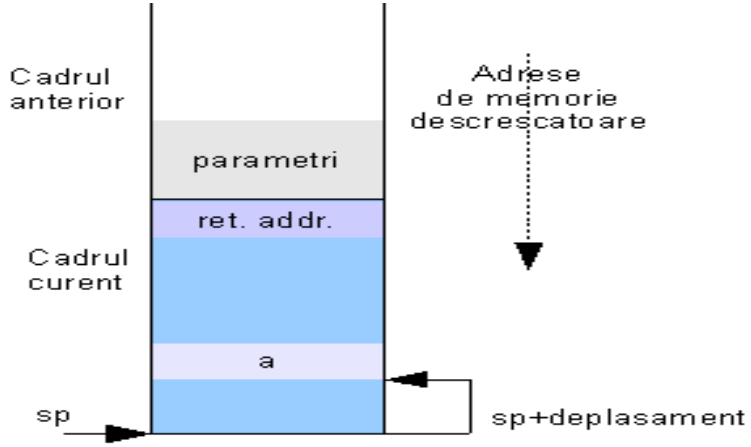


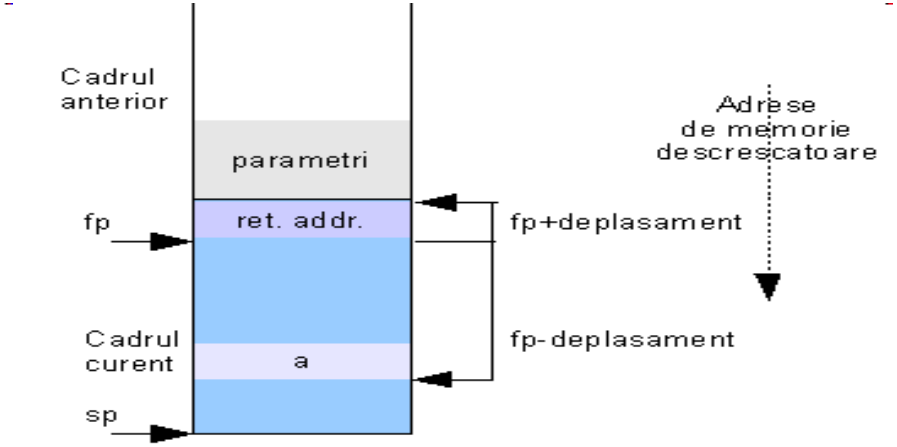
Cadrul de stiva

- Stack Pointer
- Frame Pointer
- Conventii de apel
  - Parametri
  - Valori intoarse
  - “Caller / Callee saved”
  - Cine ‘curăţă’ stiva?



Frame pointer





- Un registru liber in plus
- Cadru de stiva de lungime fixa
- sp+deplasament limitat (parametrii la distanta mare)

- Parcurgerea stivei (debug, exceptii)
- Parametrii accesibili usor
- SP variabil

RISC: totul in registri

- Adresa de intoarcere din functie – salvata in registru (nonvolatil)
  - PowerPC, ARM
  - Functii frunza, thunks
- SPARC - "Register windows"

```
   mov r0, #10
   mov r1, #20
   bl  add
…
add:
   add r0, r0, r1
   bx  lr
```

```
add(10, 20)
…
int add(int a, int b)
{
   return a+b;
}
```

Apelarea procedurilor

1. Asamblarea argumentelor ce trebuie transferate procedurii si pasarea controlului.
   - Fiecare argument este evaluat si pus in registrul sau locatia de pe stiva corespunzatoare.
     - Cum se trimit structuri prin valoare?
   - Se salveaza in memorie registrii "caller saved" folositi.
   - Daca este necesar, se calculeaza legatura statica a procedurii apelate
   - Se salveaza adresa de intoarcere si se executa un salt la adresa codului procedurii (de obicei o instructiune call face aceste lucruri)

```
int add_mul2(int a, int b, int c, int d)
{ return mul(a,b) + mul(c,d); }
```

```
movl    %eax, %ebx
movl    20(%ebp), %eax
movl    %eax, 4(%esp)
movl    16(%ebp), %eax
movl    %eax, (%esp)
call    _mul
```

Apelarea procedurilor

2. Prologul procedurii este executat la intrarea in procedura. Creaza cadrul de stiva si stabileste mediul necesar adresarii.
   - Se salveaza vechiul fp, vechiul sp devine noul fp, si se calculeaza noul sp
   - Se salveaza in memorie registrii "callee saved" folositi.

```
pushl   %ebp
movl    %esp, %ebp
pushl   %ebx
subl    $20, %esp
```

3. Se executa procedura, care la randul ei poate apela alte proceduri
   - Proceduri "frunza" – nu mai apeleaza alte proceduri. Optimizari?

Apelarea procedurilor

4. Epilogul procedurii este executat la iesirea din procedura. Restaureaza mediul de adresare si reda controlul apelantului.
   - Valoarea care trebuie intoarsa se pune in locul corespunzator (daca procedura intoarce o valoare)
     - Intoarcerea structurilor – pointer la struct!
   - Registrii salvati de procedura apelata sunt restaurati din memorie
   - Se restaureaza vechiul sp si vechiul fp
   - Se incarca adresa de revenire si se executa un salt la aceasta adresa (de obicei, o instructiune ret face acest lucru)

```
movl    %ebx, %eax
addl    $20, %esp
popl    %ebx
popl    %ebp
ret
```

Apelarea procedurilor

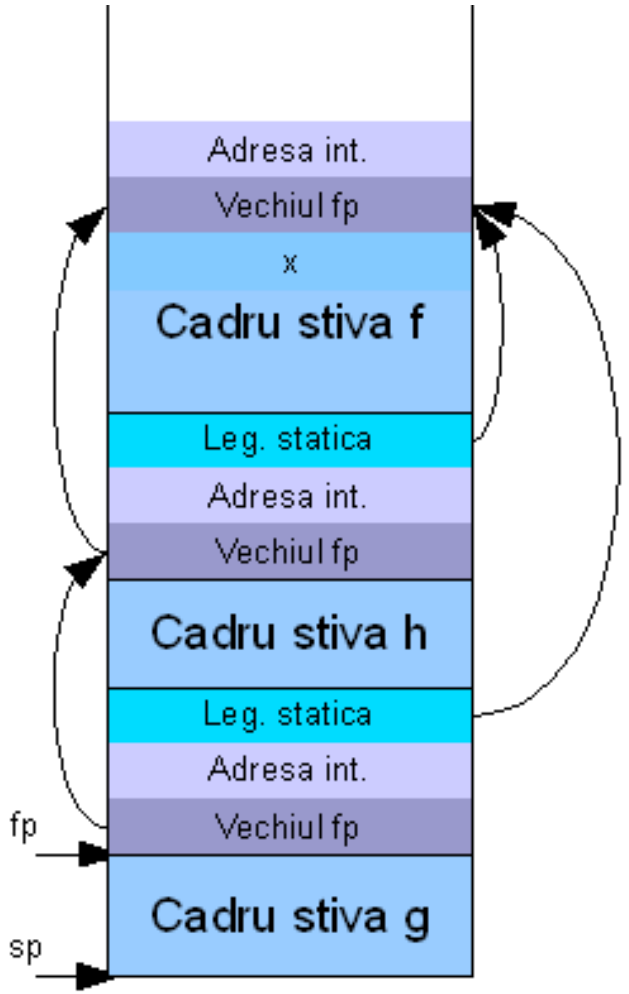
5. Codul din procedura apelanta termina restaurarea mediului său şi continuă execuţia:
   - Registrii salvati de catre procedura apelanta sunt restaurati din memorie
   - Se foloseste valoarea intoarsa de procedura apelata

Politici de apel

- Prin valoare
- Prin referinta
- Prin rezultat
- Prin valoare-rezultat
- Prin nume

Legatura statica

```
procedure f
   var x:integer;

   procedure g;
   begin
      read(x);
   end;

   procedure h;
   begin
      g;
   end;

begin
   h;
end;
```



Iteratori

Se salveaza intreg cadrul de stiva si pozitia curenta in functie (automat finit)

```
current = head;
while(current != null)
{
   yield return current.item;
   current = current.next;
}
```

```
switch (__state)
{
case 0:
  current = head;
  while(current != null)
  {
     __current = current;
     __state = 1;
     return current.item;
case 1:
     current = __current;
     current = current.next;
  }
}
```

Corutine

Stive multiple, automat finit

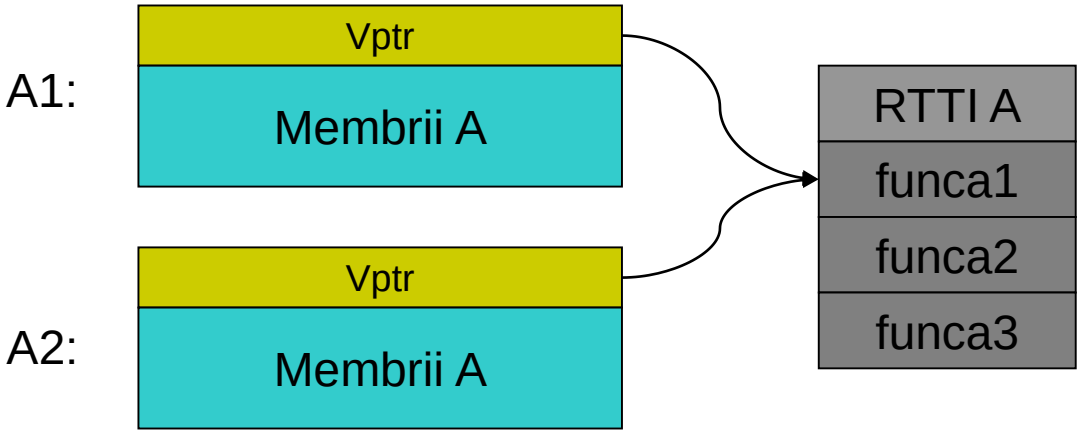
```
void Producer(void) {
  while (produce())
    resume(Consumer);
  close()
  resume(Consumer);
}

void Consumer(void) {
  resume(Producer);
  while (consume())
    resume(Producer);
}
```

Obiecte

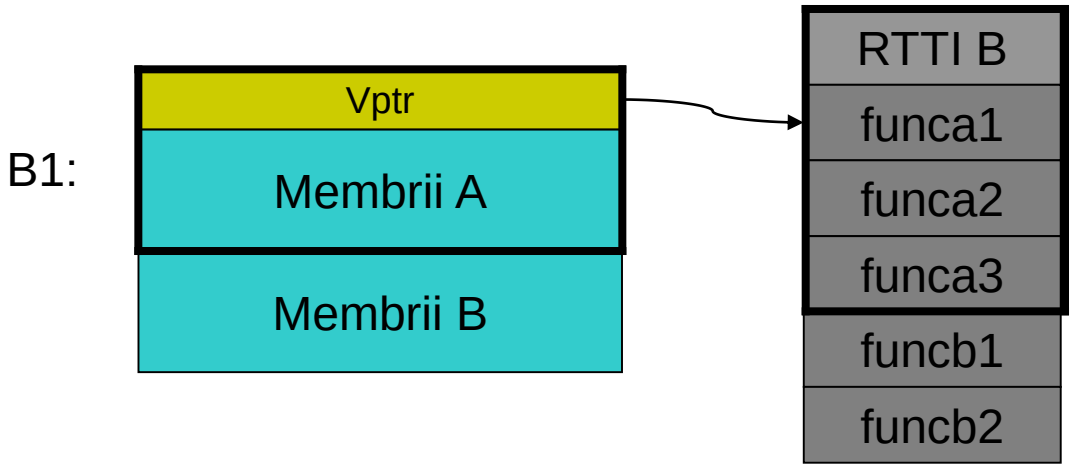
- Sunt tinute ca structuri + pointer la vtable
  - metodele virtuale
  - runtime type information
- Mostenire – pur si simplu se ‘extinde’ structura veche
  - Mai putin in cazul mostenirii virtuale – pointer la structura ‘mostenita’

Layout obiect

- class a {}



- class b: a{}



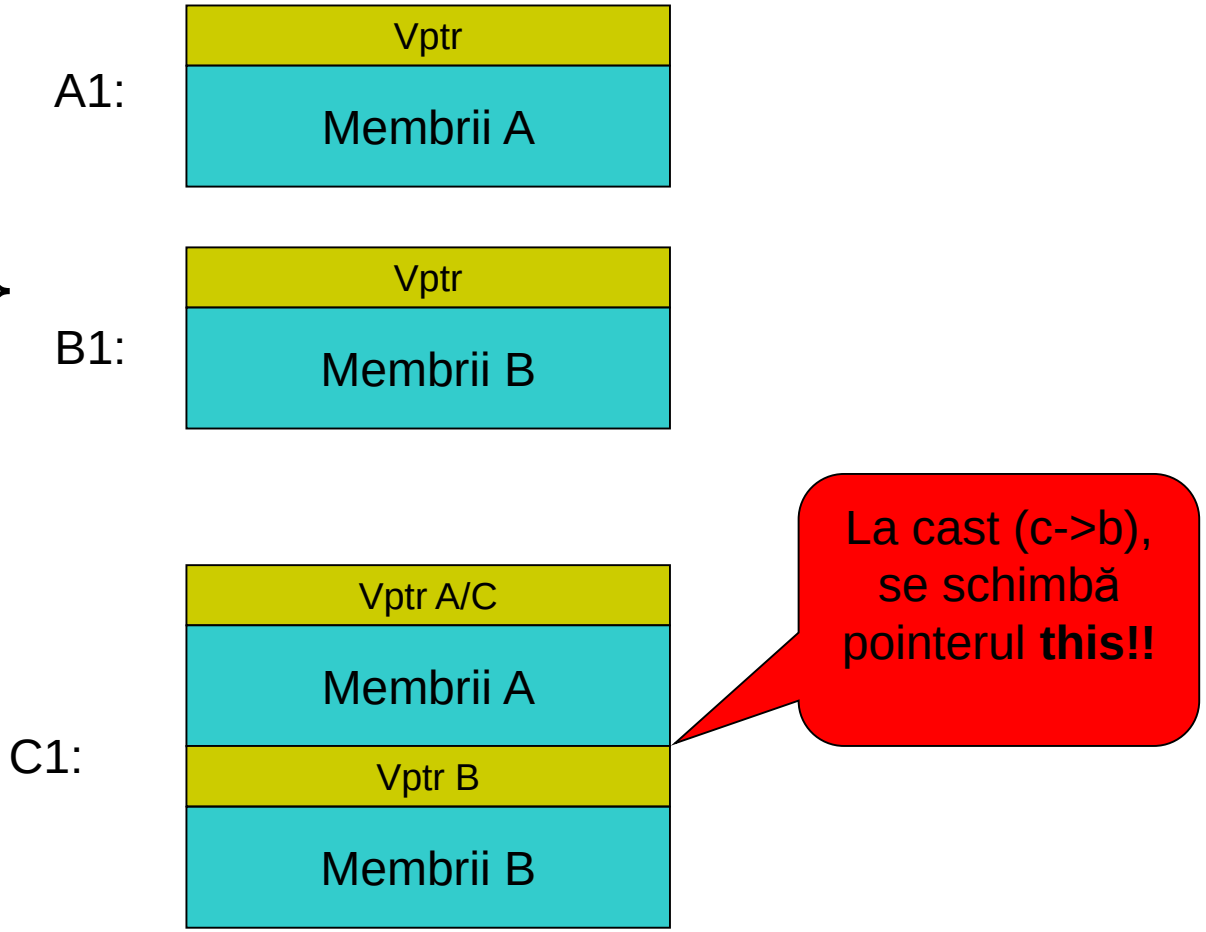
Apel de metoda virtuala

```
class Thing{ virtual void modify(int x) ... }
Thing* this_one;
this_one->modify(42)
```

- Modify are un parametru implicit(this)
- Fiecare obiect are un pointer la vtable
- Fiecare metoda virtuala are un offset cunoscut in vtable
- Apelul devine: (this_one->vtable(4))(this_one,42)
- La mostenire multipla pointerul ‘this’ e diferit!

Layout obiect – mostenire multipla

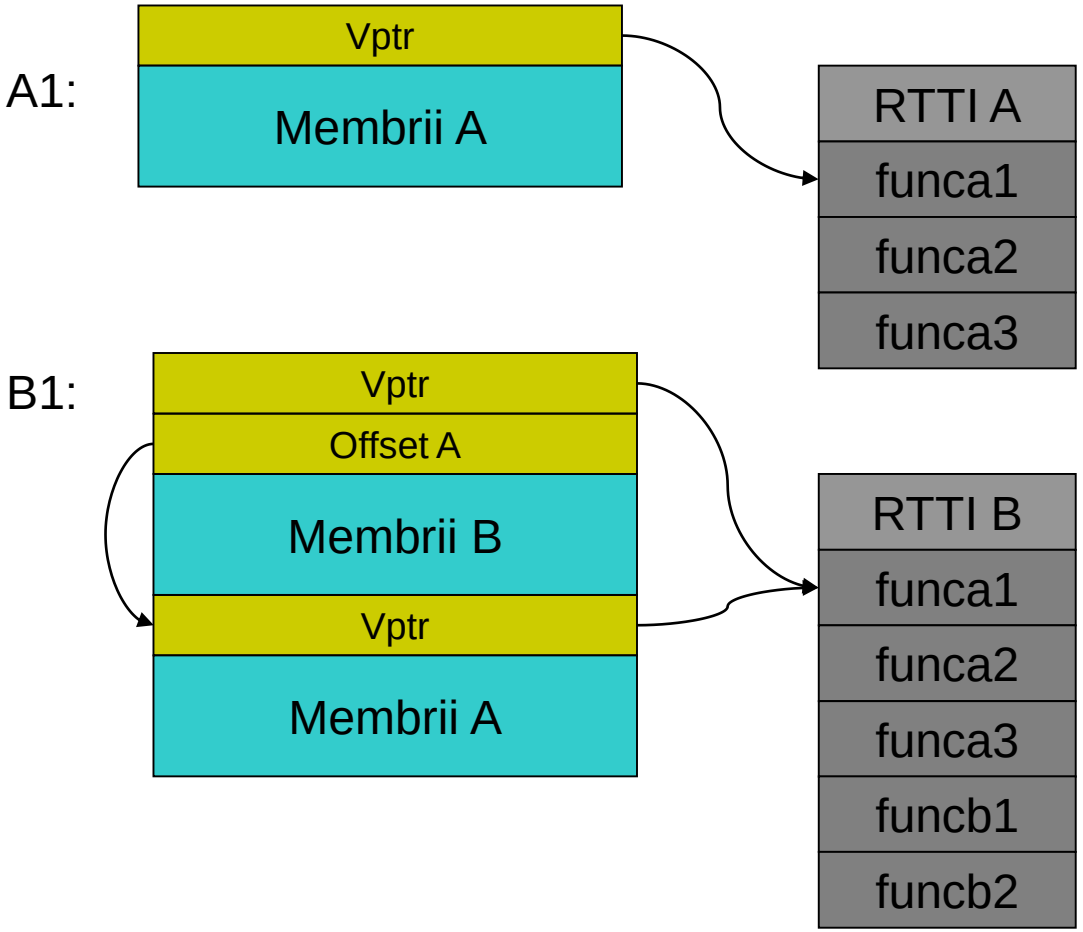
- class a {}
- class b {}
- Class c: a,b {}



Layout obiect – mostenire virtuala

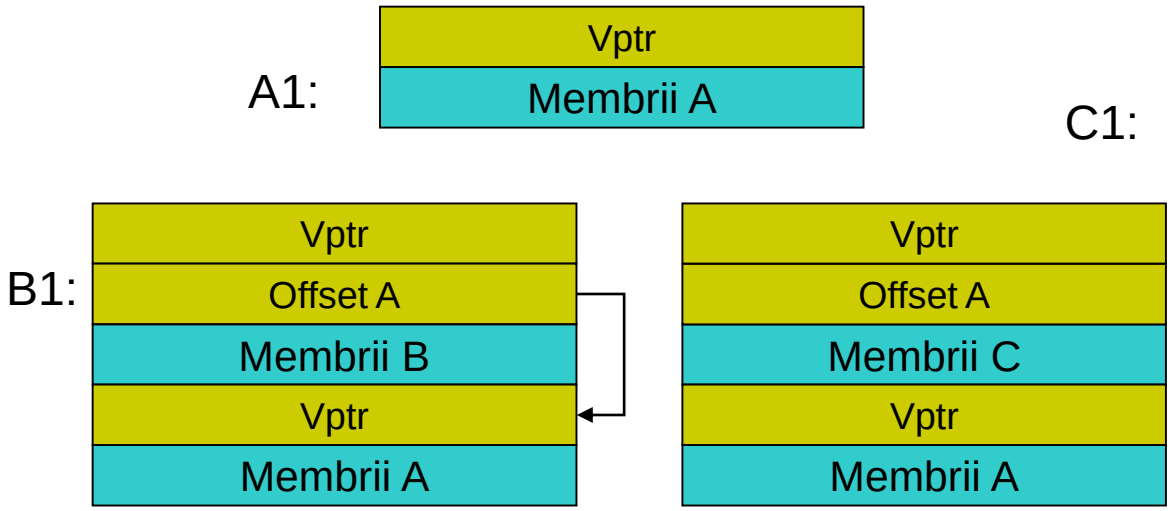
- class a {}

- class b:
  virtual a{}

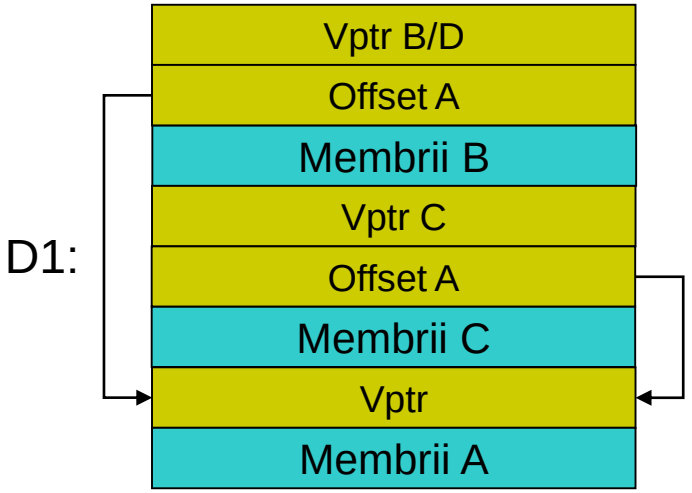


Layout obiect – mostenire multipla, virtuala

- class a {}
- class b: virtual a
- class c: virtual a
- class d:b,c

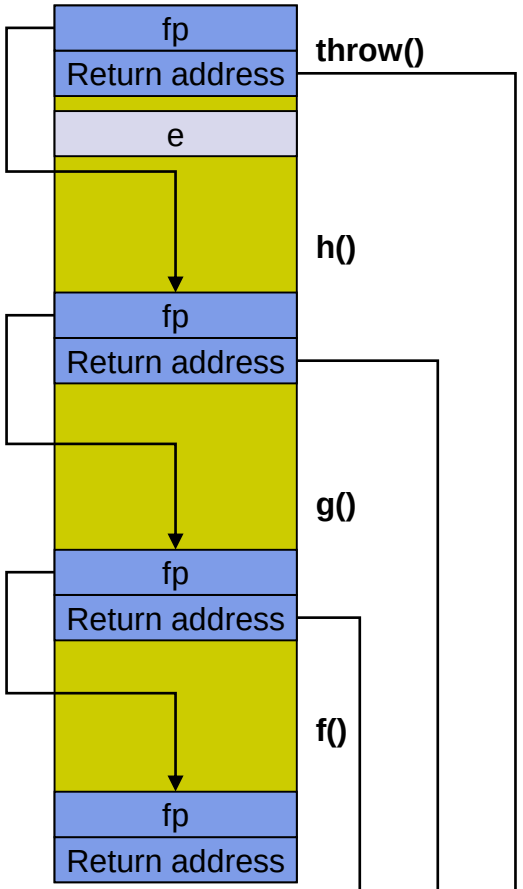


- Construirea lui A1 nu se face automat, ci in constructorul lui D1



Exceptii

```
f() { try { g(); }
  catch(IOException e) {
   print(e); }
}

g() { File x("x.dat"); h(); }

h() {
   throw(FileException("error"));
   }
```



f	0x0ac2400 - 0x0ac2420	IOException => catch block
g	0x0ac2800 - 0x0ac2830	Call x->~File()
h	0x0ac2900 - 0x0ac2940	

Functii anonime

- Lambda functions

```
items.ForEach ( item => item.value++ );
```

- Acces la obiectele disponibile la momentul creerii functiei.

```
n = 0;
items.ForEach ( delegate(Item i) { n += i.value });
```

- Se trimite un context continand o referinta catre n.
- Care este durata de viata a lui n? Ce probleme pot aparea?