

Организација графичких класа

Идеали

- Идеал програмирања је да се концепти из домена примене изразе директно у коду.
 - То би значило да ако разумете домен примене, разумећете и код, и обрнуто. На пример:
 - **Window**
 - **Line**
 - **Point**
 - **Color**

Библиотека

- Библиотека је скуп класа и функција који служе истој (сличној) сврси
 - Као основа за градњу осталих програма
 - Као основа за остале „градивне блокове“
- Добра библиотека покрива неки део домена примене
 - Не покушава све да покрије
 - Наша библиотека се труди да буде мала, а да покрије основне аспекте графике
- Не можемо правити сваку класу и функцију у библиотеци независно од осталих ствари у библиотеци.
 - Добра библиотека има униформан стил („регуларност“)

Само празан прозор

```
class My_window {  
    ...  
};
```

Korak 1

```
int main()  
{  
    Point tl(100, 200);  
    My_window win(tl, 600, 400, "Canvas");  
  
    win.draw();  
  
    return 0;  
}
```

```
struct Point {  
    int x, y;  
    Point(int xx, int yy)  
        : x(xx), y(yy) {}  
    Point() : x(0), y(0) {}  
};
```

„Цртање“ праве линије коришћењем DrawLib

```
#include "DrawLib.h"
```

```
class My_window {  
    ...  
    void draw();  
    ...  
};
```

Корак 2

```
void My_window::draw() {  
    Draw_lib::set_color(Draw_lib::C_RED); // постављање боје  
    Draw_lib::line(100, 100, 400, 400);  
}
```

Класа Line

```
class My_window {  
    ...  
    void attach(Line& x) { lineVec.push_back(x); }  
private:  
    vector<Line> lineVec;  
    ...  
};
```

Корак 3

```
class Line {  
public:  
    Line(Point x, Point y) : a(x), b(y) {}  
    Point getA() const { return a; }  
    Point getB() const { return b; }  
private:  
    Point a;  
    Point b;  
};
```

Фор по очереди (range for)

```
for (unsigned int i = 0; i < lineVec.size(); ++i) {  
    ... lineVec[i] ...  
}
```

```
for (Line it : lineVec) {  
    ... it ...  
}
```

Копак 3a

```
for (Line& it : lineVec) {  
    ... it ...  
}
```

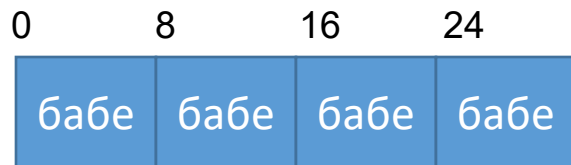
```
for (const Line& it : lineVec) {  
    ... it ...  
}
```

Вектор може да садржи само елементе истог типа

- Зашто елементи не могу бити различитог типа?
- Статичка типизираност
 - Компајлер мора током превођења знати ког типа је, на пример, овај израз:

`vec[3]`

- Величина свих елемената мора да буде иста



`vec[3]` \leftrightarrow почетна адреса + 3*величина типа бабе
 \leftrightarrow почетна адреса + 3*8
 \leftrightarrow почетна адреса + 24



`vec[3]` \leftrightarrow ????????

Вектор може да садржи само елементе истог типа

- Али, можемо имати више вектора и више функција `attach`.
- Која функција `attach` ће бити позвана компајлер одређује током превођења на основу ***потписа*** (сетите се преклапање функција).

Корак 4

Како изгледа поступак додавања новог облика?

- 1. Нова класа у `MyGraph.h`
 - 2. Нови вектор и нова функција `attach` у `MyWindow.h`
 - 3. Нова петља и код за исцртавање у `My_window::draw()`, у `MyWindow.cpp`
-
- Сваки пут треба додавати код на три различита места.
 - Да ли можемо упростити?
 - Можемо, на неколико начина.
 - Данас ћемо видети један од најчешћих начина.

Прво да уклонимо потребу за више вектора и attach функција

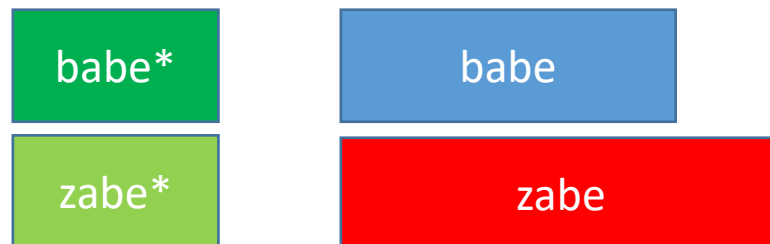
- 1. Проблем величине:
- Величине елемената вектора не могу бити различите, а величине различитих типова се могу разликовати.



Прво да уклонимо потребу за више вектора и attach функција

- 1. Проблем величине:
- Величине типова могу се разликовати, али адресе свих типова су увек исте величине.
- Тип података који може садржати адресу неког објекта назива се „показивач“. Такође, тако зовемо и променљиву/објекат тог типа.
- Међутим, тип показивача садржи информацију о томе на који тип објекта може показивати. Тако је показивач на бабе различит тип од показивача на жабе. Декларишемо их овако:

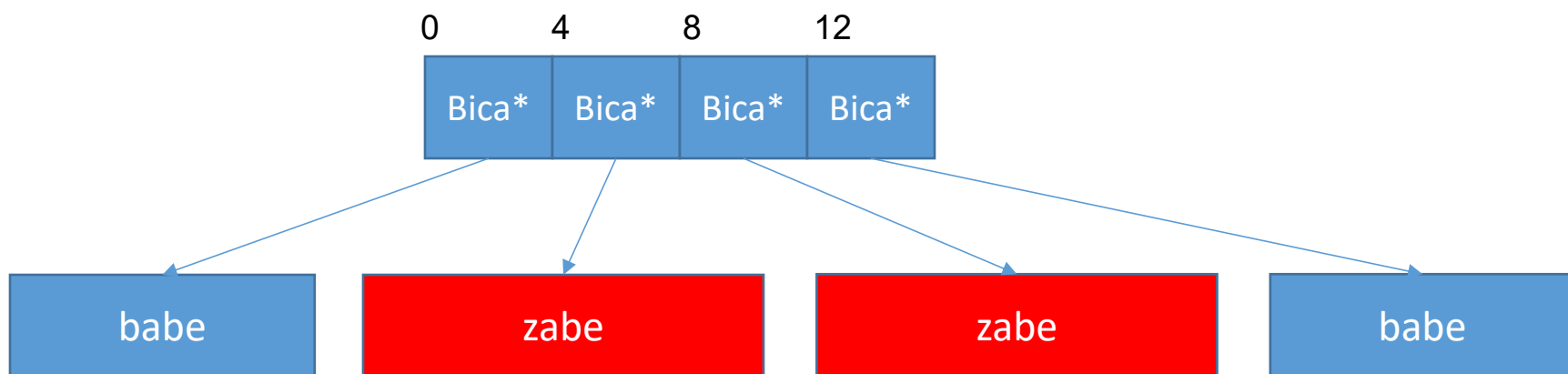
```
Babe* p1;  
Zabe* p2;
```



- Разлог је опет статичка типизираност. Компајлер мора током превођења знати ког типа је, на пример, овај израз, који представља добављање објекта на који показивач показује:

Прво да уклонимо потребу за више вектора и attach функција

- 2. Проблем статичке типизације:
- Механизам наслеђивања нам омогућава да на контролисан начин превазиђемо део ограничења која намеће статичка типизација.
- У Це++-у не постоји никаква унапред дата хијерархија класа/типова. Све сами правимо.
- Ако уведемо нови тип Bica, који наслеђују типови Babe и Zabe, онда можемо направити један вектор, чији елементи су Bica*. Сваки елемент онда може показивати и на Babe и на Zabe.



Како изгледа поступак додавања новог облика?

- 1. Нова класа у `MyGraph.h`
- ~~2. Нови вектор и нова функција `attach` у `MyWindow.h`~~
- 3. Нова **switch** **грана** и код за исцртавање у `My_window::draw()`, у `MyWindow.cpp`
- Али, како да знамо да ли конкретан елемент показује на `Babe` или на `Zabe`? (Кроз показивач на `Bica` можемо приступати само елементима типа `Bica`)
- У `Ce++`-у информација о типу објекта на који се показује није доступна (бар не подразумевано али може да се користи тзв. `RTTI` – што можете сами погледати 😊).
- Најчешће морамо сами да је обезбедимо.

Корак 6

Код за исцртавање логички груписати са класом облика

- 1. Нова класа у MyGraph.h
- ~~2. Нови вектор и нова функција attach у MyWindow.h~~
- ~~3. Нова switch грана и код за исцртавање у My_window::draw(), у MyWindow.cpp~~
- Код за цртање можемо издвојити у посебну функцију. По једна функција за сваки облик.
- Функције онда можемо изместити из MyWindow.cpp.
- Рецимо, у саме класе које моделују облике.

Корак 7

Желимо да укинемо и switch наредбу

- 1. Нова класа у MyGraph.h
- ~~2. Нови вектор и нова функција attach у MyWindow.h~~
- ~~3. Нова switch грана и код за исцртавање у My_window::draw(), у MyWindow.cpp~~

Корак 8

- Сада се можемо ослонити на механизам полиморфизма.
- Одређене функције чланице (методе) можемо прогласити виртуелним.
- Наслеђени типови могу преклопити те функције.
- Тада се, у случају да се метода позива преко показивача на базну класу, одлука која ће функција бити позвана доноси током извршавања (динамички) и зависи од конкретног типа на који тај показивач показује.
- **То је врло различито од позива свих осталих функција!**
- Како то ради? – погледајте наредних ~20 слајдова :)

Модел рачунара - подсећање

- \$r је неки регистар, \$val је нека непосредна вредност

```
$r <- $val
```

```
mem[$val] <- $r
```

```
mem[$r] <- $r
```

```
mem[$r + $val] <- $r
```

```
$r <- mem[$addr]
```

```
$r <- mem[$r]
```

```
$r <- mem[$r + $val]
```

```
$r <- $r + $r
```

```
$r <- $r + $val
```

```
$r <- $r - $r
```

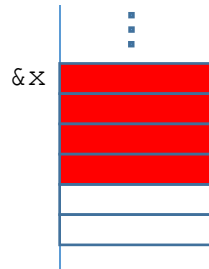
```
$r <- $r * $r
```

```
call $val
```

```
call $r
```

Представа у меморији и приступ

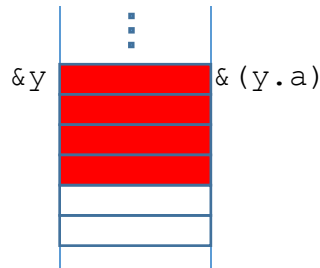
```
int x;
```



```
x = 5;
```

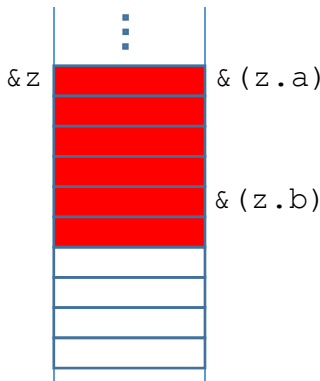
```
r1 <- 5  
mem[_x] <- r1
```

```
struct Tip1 {  
    int a;  
};  
Tip1 y;
```



```
y.a = 5;    mem[_y+0] <- r1
```

```
struct Tip2 {  
    int a;  
    short b;  
};  
Tip2 z;
```



```
z.a = 5;    mem[_z+0] <- r1  
z.b = 5;    mem[_z+4] <- r1
```

Представа у меморији и приступ

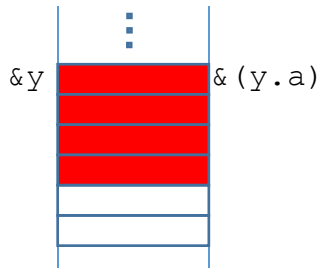
```
int x;
```



```
int* p = &x;  
*p = 5;
```

```
r1 <- 5  
r2 <- mem[_p]  
mem[r2] <- r1
```

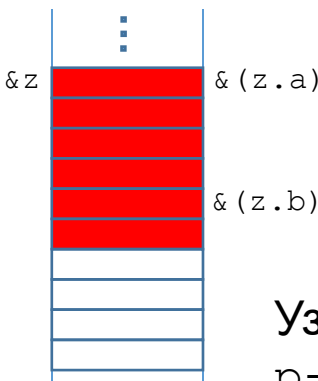
```
struct Tip1 {  
    int a;  
};  
Tip1 y;
```



```
Tip1* p = &y;  
p->a = 5;
```

```
r2 <- mem[_p]  
mem[r2+0] <- r1
```

```
struct Tip2 {  
    int a;  
    short b;  
};  
Tip2 z;
```



```
Tip2* p = &z;  
p->a = 5;  
p->b = 5;
```

```
r2 <- mem[_p]  
mem[r2+0] <- r1  
mem[r2+4] <- r1
```

Узгред:

p->a је скраћени запис овога: (*p) . a

Функције нису део променљиве

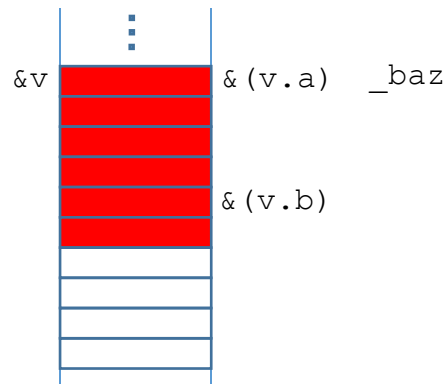
```
void foo(Tip1& x)  _foo
{
    ...
}
```



foo(v);

```
r1 <- _v
call _foo
```

```
class Tip1 {
    int a;
    short b;
public:
    void baz();
    void bar();
};
Tip1 v;
```



v.bar();

```
r1 <- _v
call _bar
```



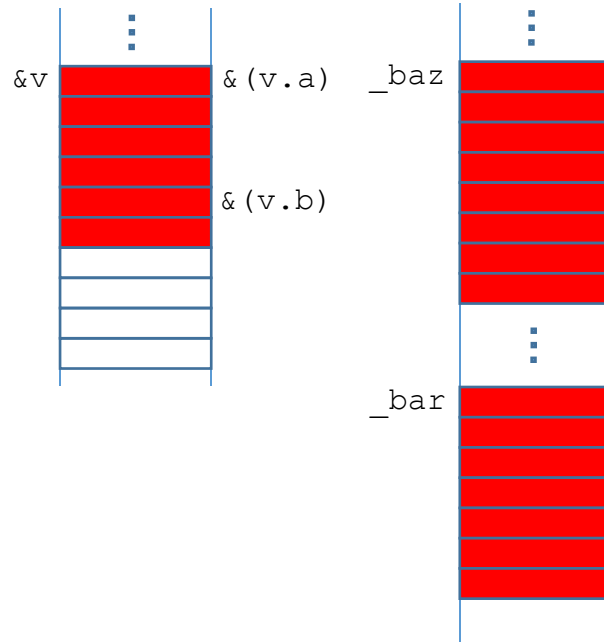
Функције нису део променљиве

```
void foo(Tip1& x)  _foo
{
    ...
}
```



```
Tip1* p = &v; mem[_p] <- _v
r1 <- mem[_p]
foo(*p); call _foo
```

```
class Tip1 {
    int a;
    short b;
public:
    void baz();
    void bar();
};
Tip1 v;
```



```
p->bar();
```

```
mem[_p] <- _v
r1 <- mem[_p]
call _bar
```

Шта је веће, Tip1 или Tip2?

```
struct Tip1 {  
    int a;  
};
```

```
struct Tip2 {  
    int a;  
    double b;  
};
```

```
struct Tip1 {  
    double a;  
    long b;  
};
```

```
struct Tip2 {  
    int a;  
    double b;  
};
```

```
struct Tip1 {  
    int a;  
    double b;  
    void foo();  
    void bar();  
};
```

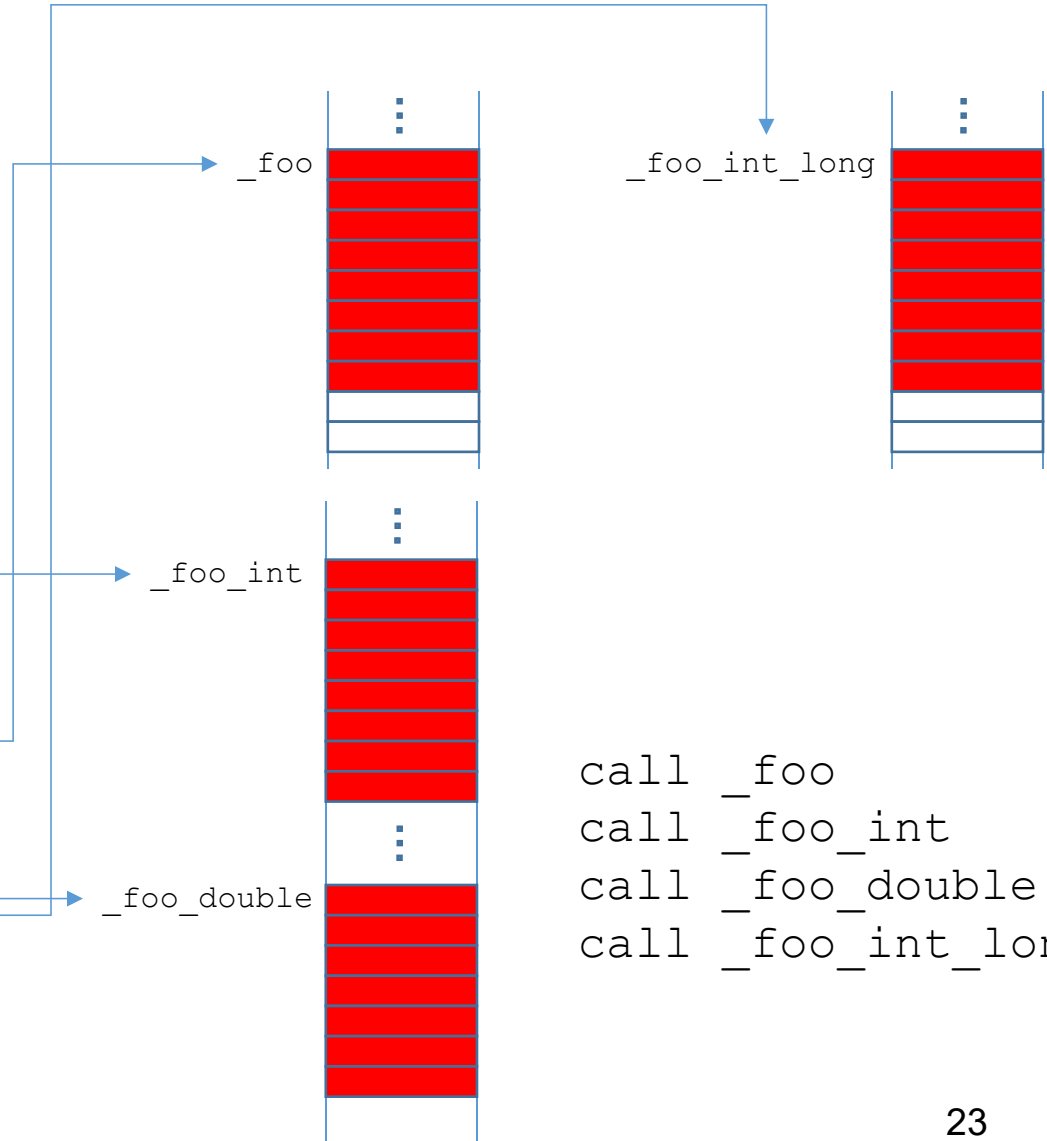
```
struct Tip2 {  
    int a;  
    double b;  
    void foo();  
    void bar();  
    void baz();  
};
```

Скуп преклоплених функција

```
void foo();  
void foo(int x);  
void foo(double x);  
void foo(int x, long y);  
...
```

```
int a;  
double b;  
long c;
```

```
foo();  
foo(a);  
foo(b);  
foo(a, c);
```

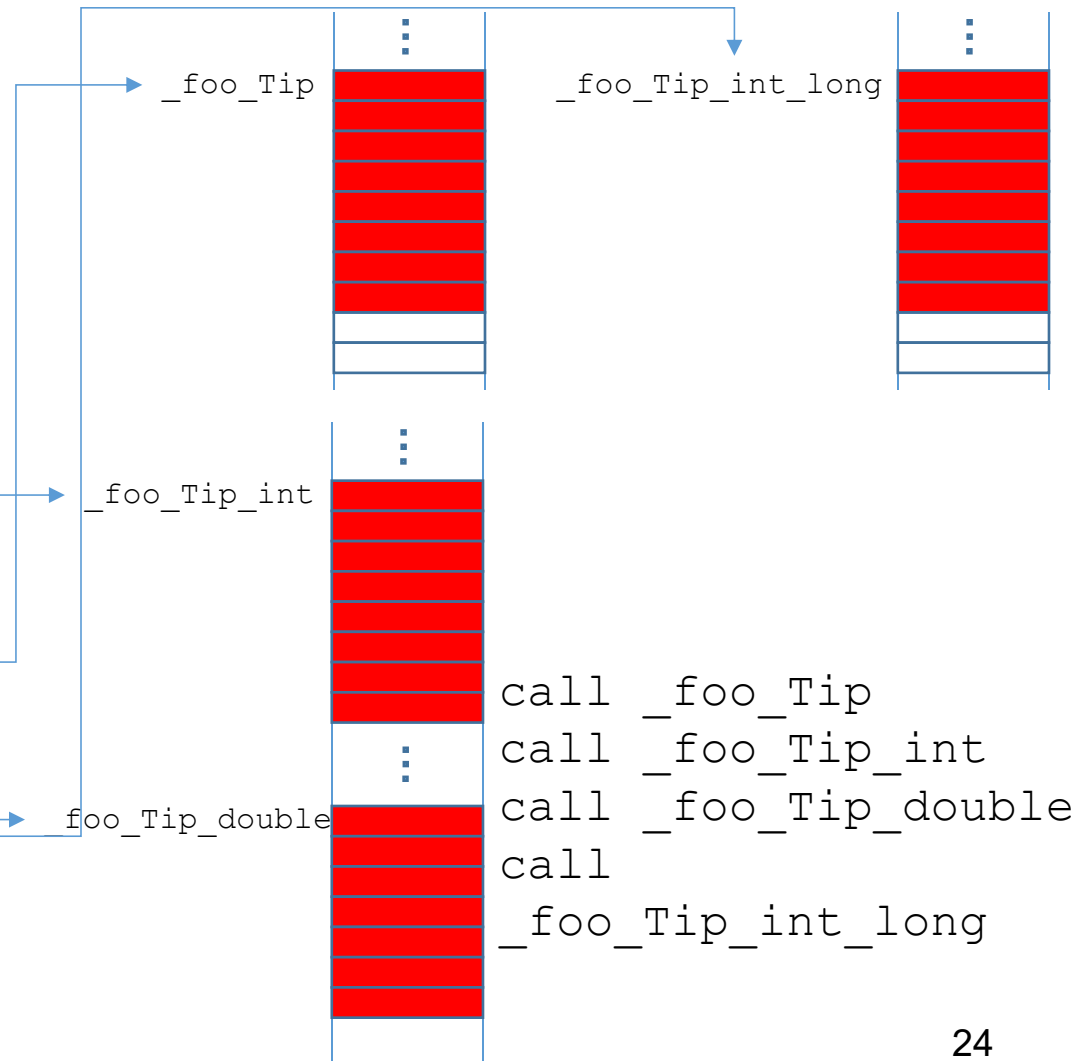


```
call _foo  
call _foo_int  
call _foo_double  
call _foo_int_long
```

Скуп преклоплених функција

```
struct Tip {  
    void foo();  
    void foo(int x);  
    void foo(double x);  
    void foo(int x, long y);  
    ...  
};  
Tip v;
```

```
v.foo();  
v.foo(a);  
v.foo(b);  
v.foo(a, c);  
foo(Tip& _); // foo(v);  
foo(Tip& _, int x);  
foo(Tip& _, double x);  
foo(Tip& _, int x, long y);
```



Скуп преклопљених функција

```
struct Tip {
    void foo();
    void foo(int x);
    void foo(double x);
    void foo(int x, long y);
    ...
};
```

```
Tip v;  
Tip* p = &v;
```

```
p->foo ();
```

```
p->foo (a) ;
```

```
p->foo (b) ;
```

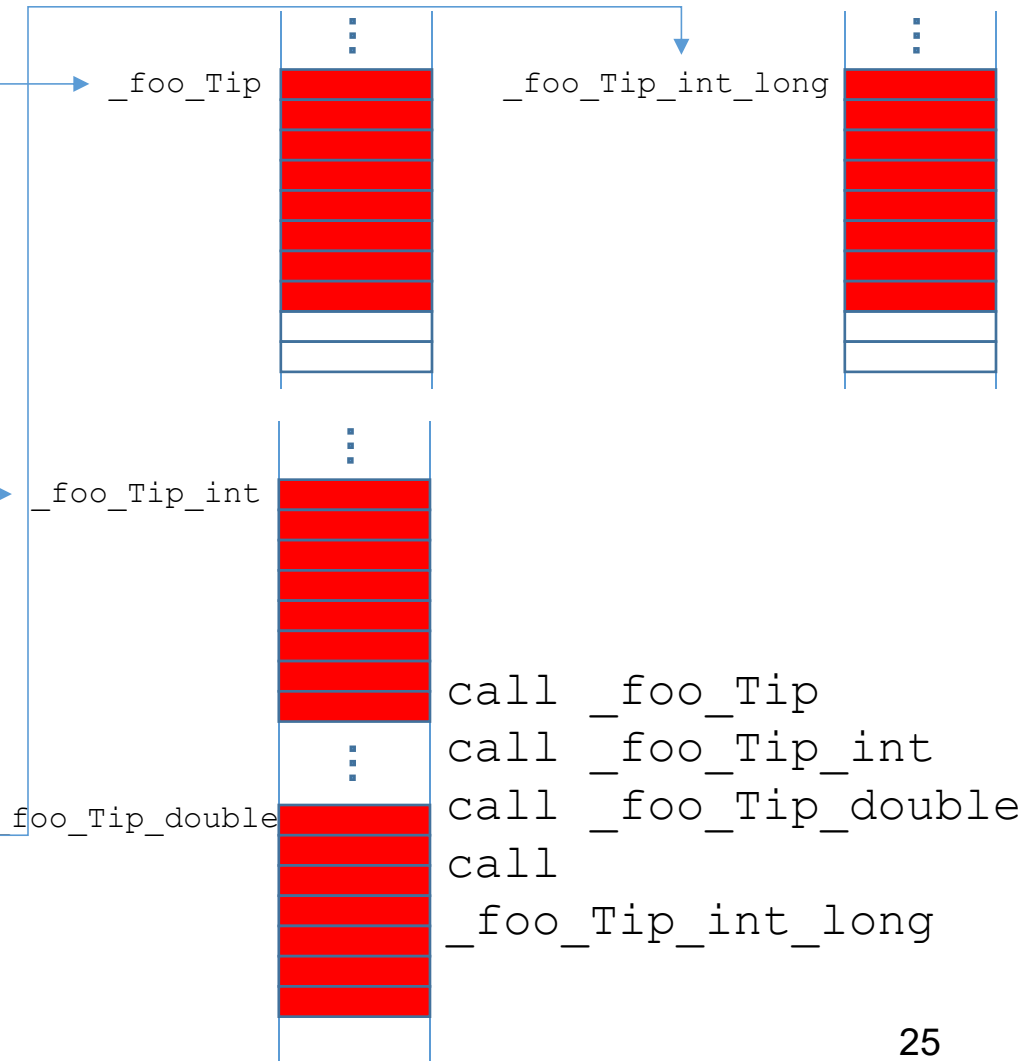
```
p->foo (a, c);
```

```
foo(Tip&    ); // foo(*p);
```

```
foo(Tip& , int x);
```

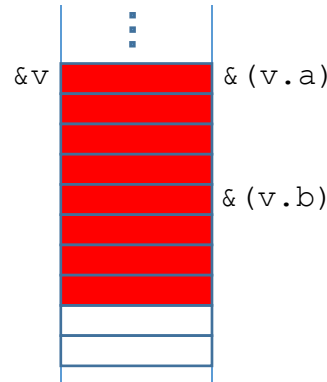
```
foo(Tip& , double x);
```

```
foo(Tip& , int x, long y);
```



Наслеђивање

```
struct Tip1 {  
    int a;  
    float b;  
};  
Tip1 v;
```

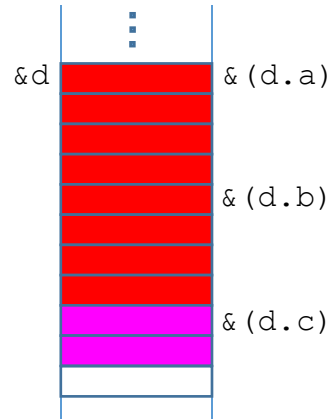


```
v.a = 5;  
v.b = 5;
```

```
r1 <- 5
```

```
mem[_v + 0] <- r1  
mem[_v + 4] <- r1
```

```
struct Tip2 : Tip1 {  
    short c;  
}  
Tip2 d;
```



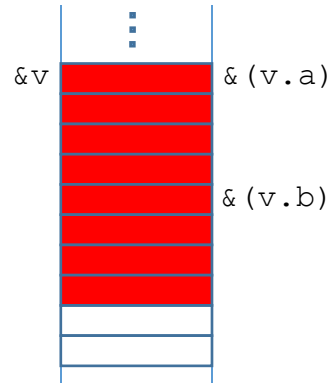
```
d.a = 5;  
d.b = 5;  
d.c = 5;
```

```
mem[_d + 0] <- r1  
mem[_d + 4] <- r1  
mem[_d + 8] <- r1
```

Подаци чланови (атрибути) изведеног типа се напросто додају на крај.
Ово је важно, јер тако помераји (офсети) за основни тип и даље важе.

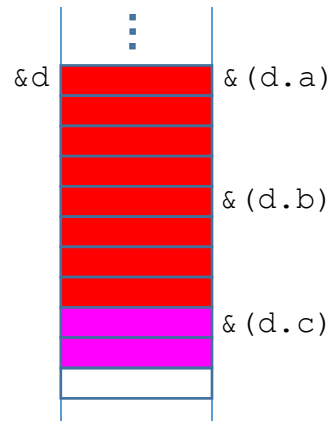
Наслеђивање

```
struct Tip1 {  
    int a;  
    float b;  
};  
Tip1 v;
```



```
Tip1* p = &v; r1 <- 5  
r2 <- mem[_p]  
p->a = 5;      mem[r2 + 0] <- r1  
p->b = 5;      mem[r2 + 4] <- r1
```

```
struct Tip2 : Tip1 {  
    short c;  
}  
Tip2 d;
```



```
Tip2* p = &d; r2 <- mem[_p]  
p->a = 5;      mem[r2 + 0] <- r1  
p->b = 5;      mem[r2 + 4] <- r1  
p->c = 5;      mem[r2 + 8] <- r1
```

```
Tip1* bp = &d; r3 <- mem[_p]  
bp->a = 5;      mem[r3 + 0] <- r1  
bp->b = 5;      mem[r3 + 4] <- r1  
bp->c ???
```

Подаци чланови (атрибути) изведеног типа се напросто додају на крај.
Ово је важно, јер тако помераји (офсети) за основни тип и даље важе.

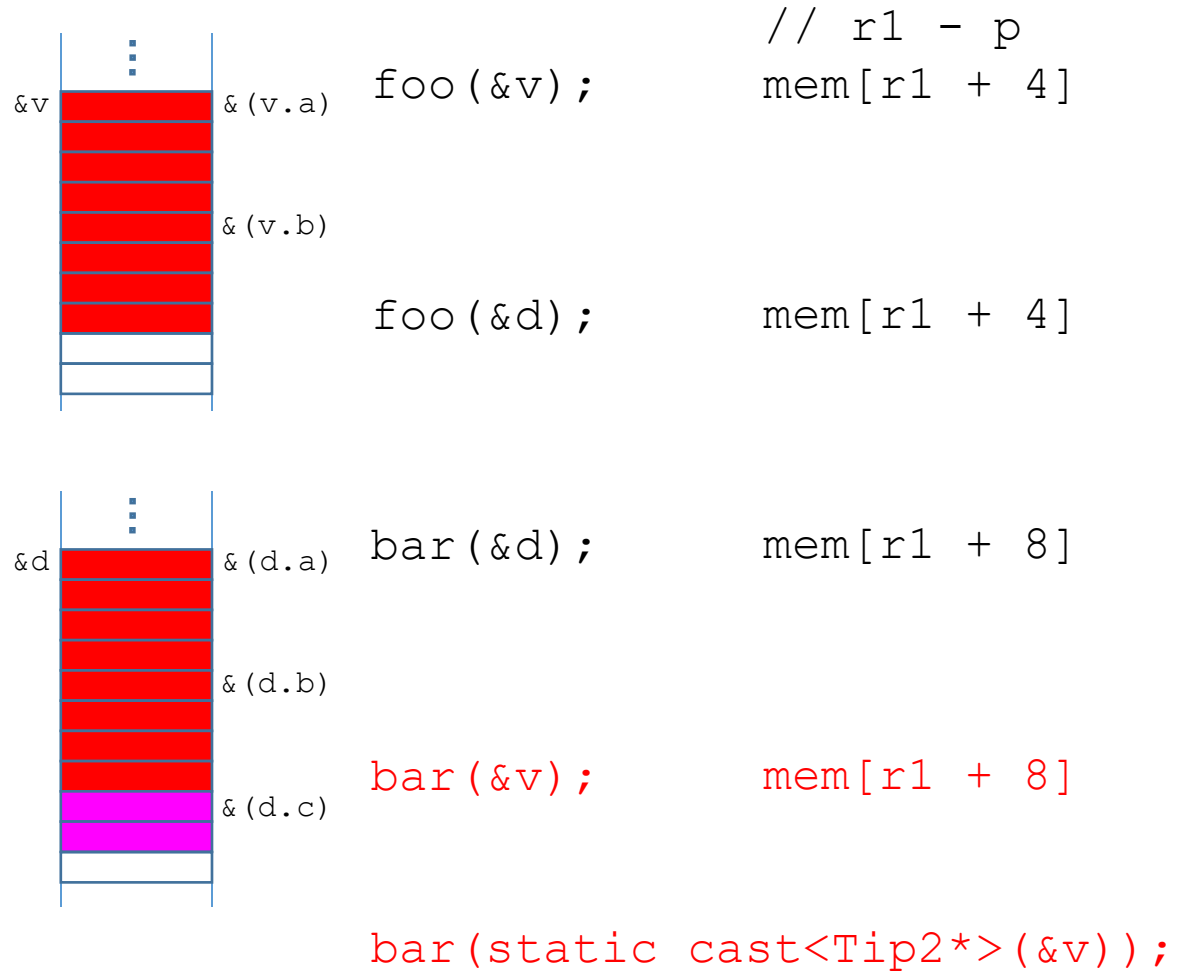
Потомак уместо претка

```
struct Tip1 {
    int a;
    float b;
};
Tip1 v;
```

```
struct Tip2 : Tip1 {
    short c;
}
Tip2 d;
```

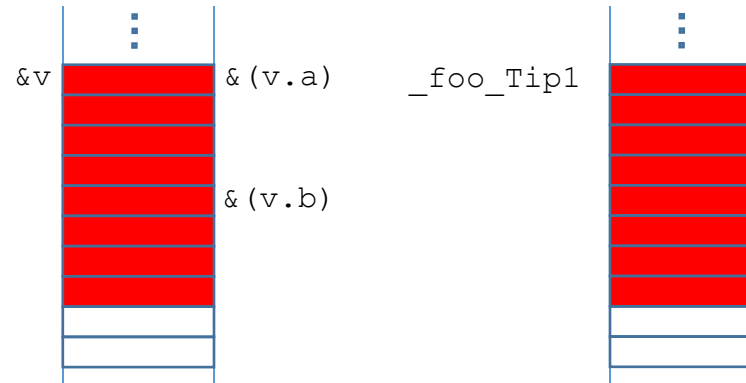
```
void foo(Tip1* p) {
    p->b = 5;
}
```

```
void bar(Tip2* p) {
    p->c = 5;
}
```

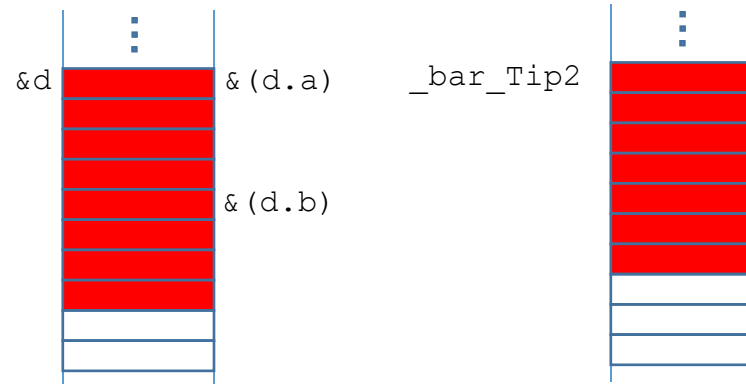


Преклапање функција

```
struct Tip1 {  
    int a;  
    float b;  
    void foo();  
};  
Tip1 v;
```



```
struct Tip2 : Tip1 {  
    void bar();  
}  
Tip2 d;
```



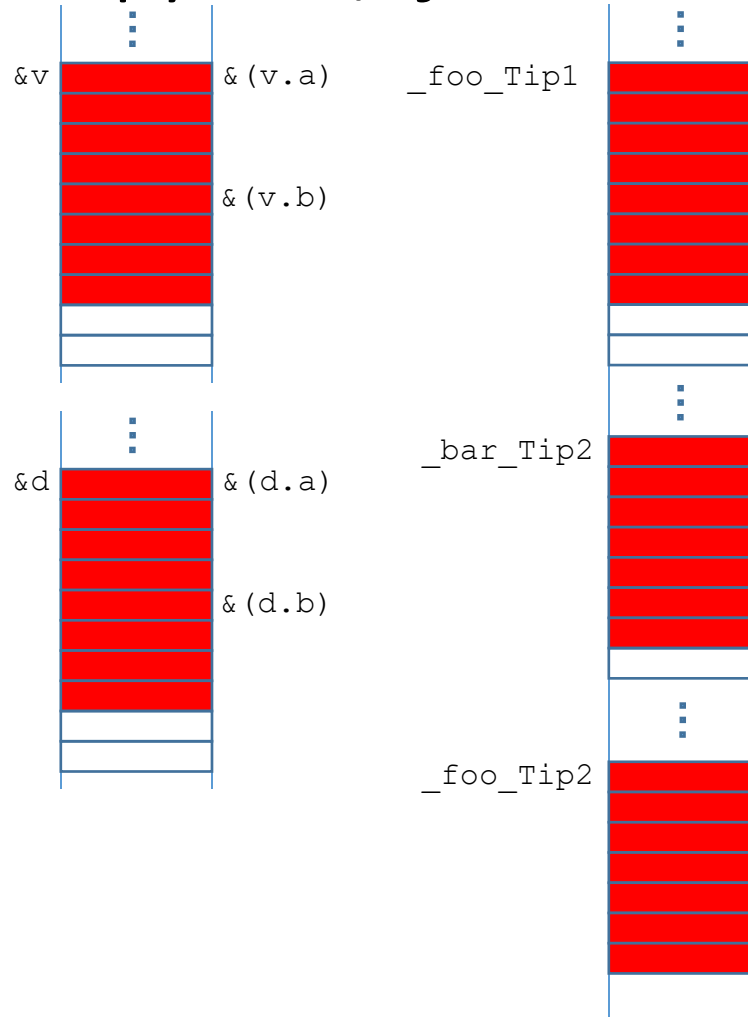
```
v.foo();  
d.bar();
```

Преклапање функција

```
struct Tip1 {  
    int a;  
    float b;  
    void foo();  
};  
Tip1 v;
```

```
struct Tip2 : Tip1 {  
    void bar();  
    void foo();  
}  
Tip2 d;
```

```
v.foo();  
d.foo();  
Tip1* p1 = &v;  
p1->foo();  
p1 = &d;  
p1->foo();  
Tip2* p2 = &d;  
p2->foo();
```

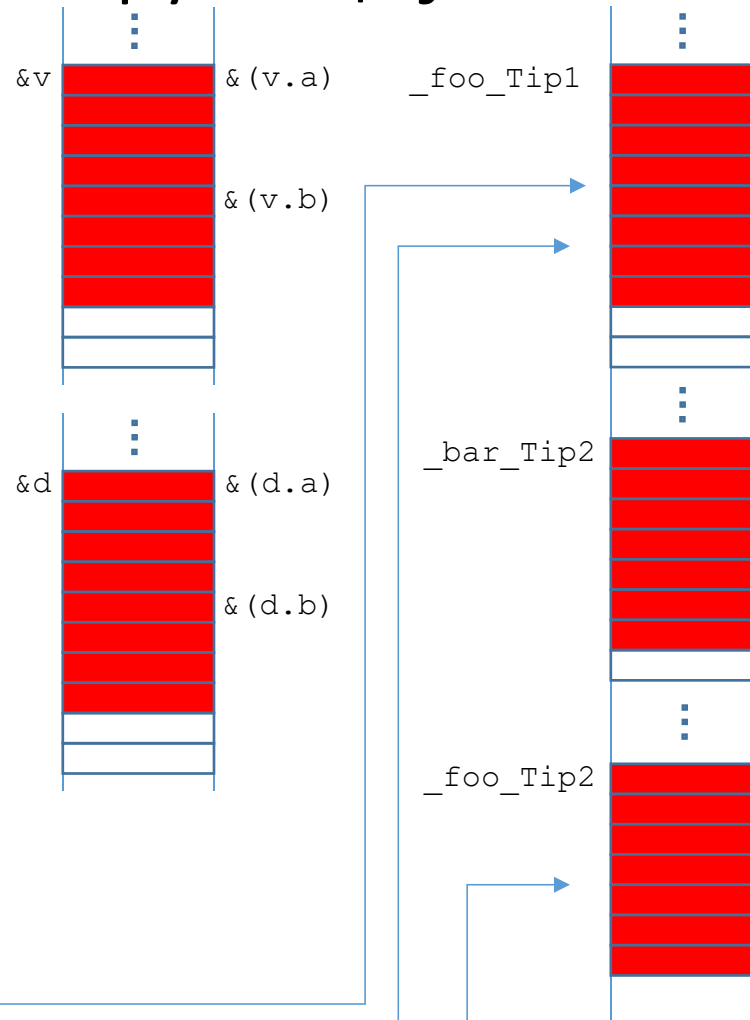


Преклапање функција

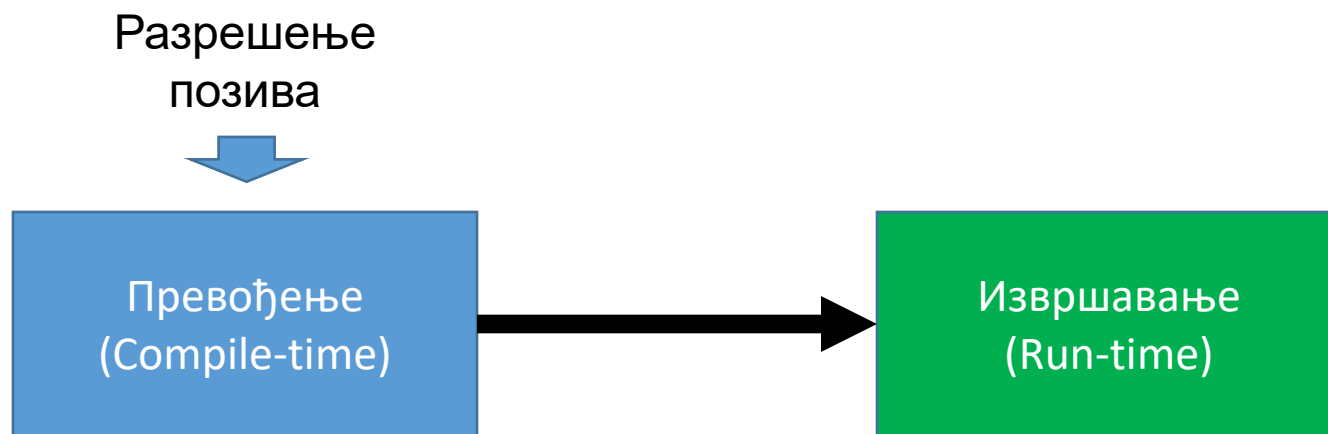
```
struct Tip1 {  
    int a;  
    float b;  
    void foo();  
};  
Tip1 v;
```

```
struct Tip2 : Tip1 {  
    void bar();  
    void foo();  
}  
Tip2 d;
```

```
v.foo();  
d.foo();  
Tip1* p1 = &v;  
p1->foo();  
p1 = &d;  
p1->foo();  
Tip2* p2 = &d;  
p2->foo();
```

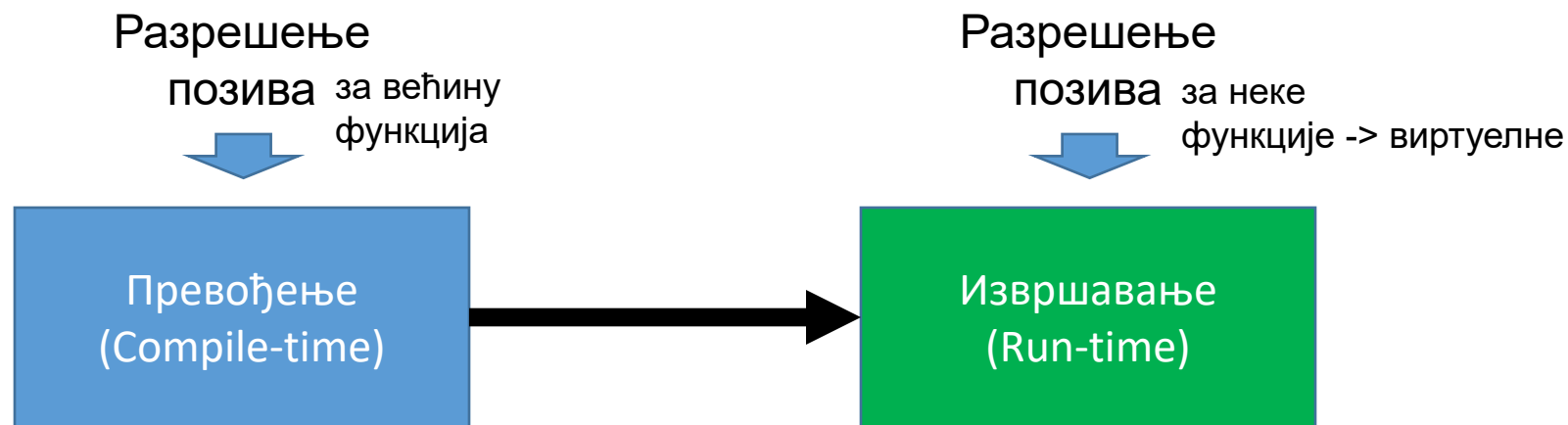


Позив функције



Да би се одредило која функција треба да се позове на одређеном месту, типови стварних параметара (аргумената) морају бити познати.

Позив функције



Да би се одредило која функција треба да се позове на одређеном месту, типови стварних параметара (аргумената) морају бити познати.

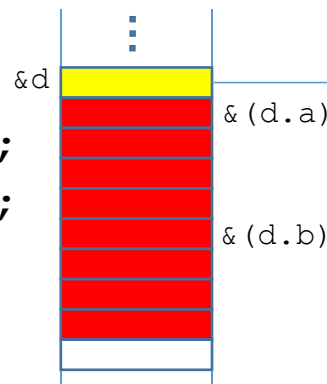
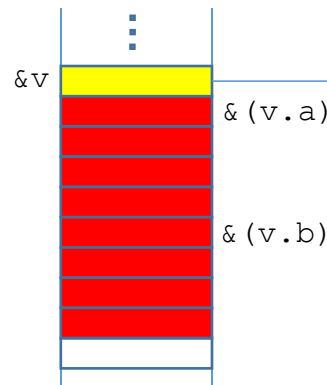
Полиморфизм

```
struct Tip1 {  
    int a;  
    float b;  
    virtual void foo();  
    virtual void bar();  
};  
Tip1 v;
```

```
struct Tip2 : Tip1 {  
    void foo() override;  
    void bar() override;  
    void baz();  
}  
Tip2 d;
```

```
Tip1* p1 = &v;  
p1->foo();  
p1 = &d;  
p1->foo();
```

_baz_Tip2



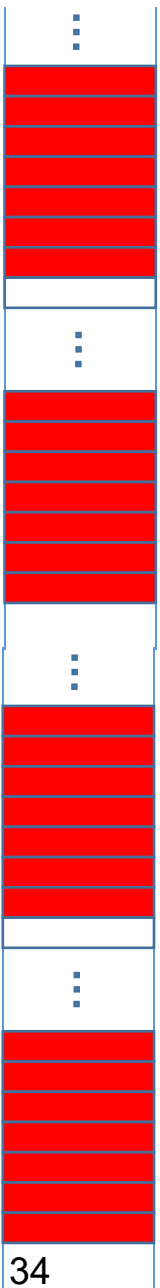
```
//r1 - p  
r2 <- mem[r1+0]  
r3 <- mem[r2+0]  
call r3 // foo  
r3 <- mem[r2+1]  
call r3 // bar
```

_foo_Tip1

_foo_Tip2

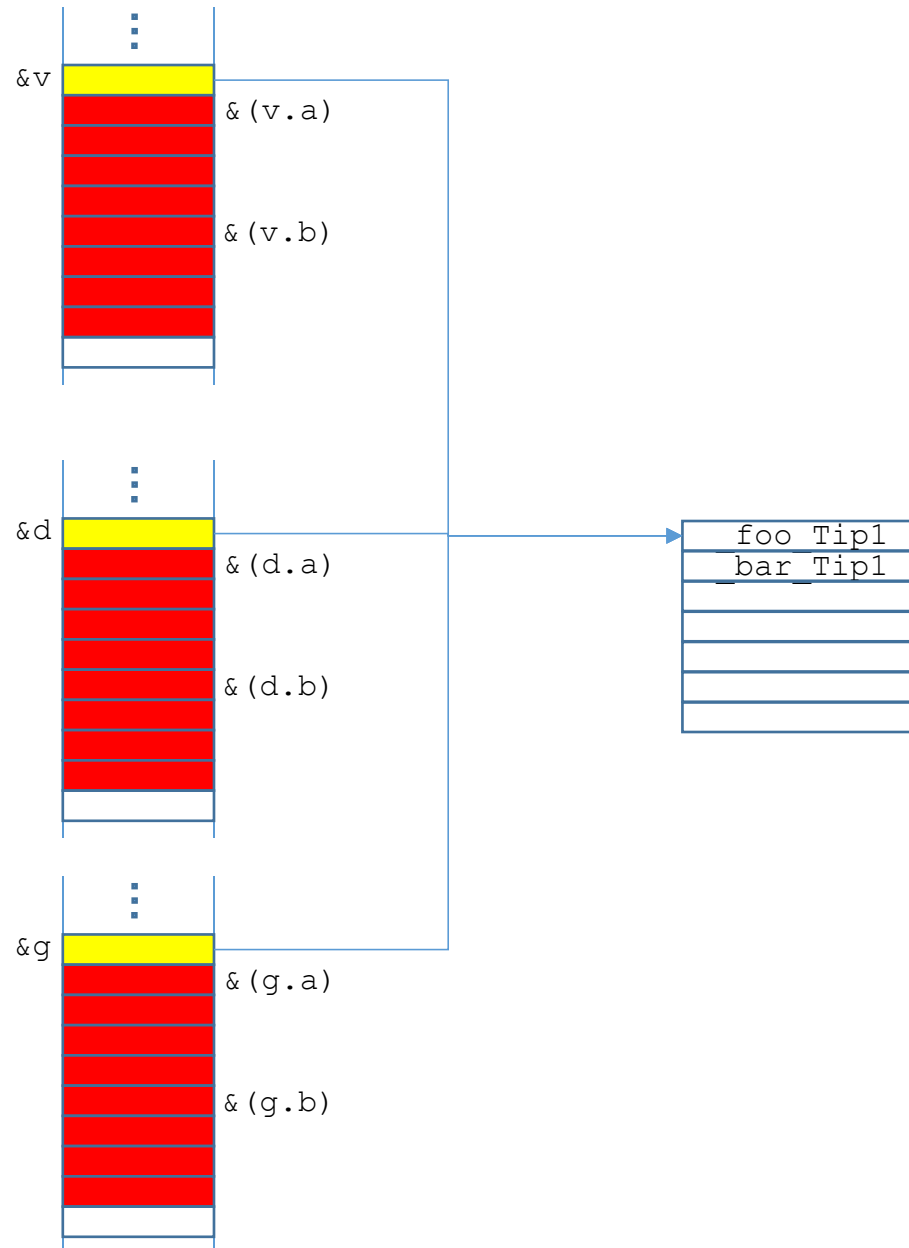
_bar_Tip1

_bar_Tip2



Једна табела по типу

```
struct Tip1 {  
    int a;  
    float b;  
    virtual void foo();  
    virtual void bar();  
};  
Tip1 v;  
Tip1 d;  
Tip1 g;
```



Шта је веће, Tip1, Tip2 или Tip3?

```
struct Tip1 {  
    int a;  
    double b;  
    void foo();  
    void bar();  
    void baz();  
};
```

```
struct Tip2 {  
    int a;  
    double b;  
    void foo();  
    void bar();  
};
```

```
struct Tip3 {  
    int a;  
    double b;  
    virtual void foo();  
};
```