SVEUČILIŠTE U ZAGREBU

PMF - MATEMATIČKI ODJEL

Jelena Đidara

OBJEKTNI MODEL U PROGRAMSKOM JEZIKU C++

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, rujan 2010.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Tema ovog rada je Objektni Model u programskom jeziku C++ čije dobro poznavanje i razumijevanje pomaže programeru da piše programe koji su manje skloni greškama i koji su učinkovitiji.

Što je C++ Objektni Model?

Dva su aspekta C++ Objektnog Modela:

1. Izravna potpora za objektno-orijentirano programiranje osigurana unutar jezika
2. Temeljni mehanizam po kojem je ta podrška implementirana

Razina potpore osigurane unutar jezika je prilično dobro pokrivena u *C++ Primer Fourth Edition* i ostalim knjigama na temu programskog jezika C++. Drugi aspekt je slabije opskrbljen materijalima, najrelevantija literatura je *Inside the C++ Object Model* na kojoj se moj rad i temelji.

U prvom poglavlju će biti govora o pozadini objektno-baziranog programiranja. Sadrži kratku turneju Objektnim Modelom gdje ćemo se osvrnuti na nasljeđivanje i polimorfizam koje ću detaljnije razraditi u trećem i četvrtom poglavlju.

Drugo poglavlje razmatra detalje rada konstruktora. Raspravlja kad prevodilac sintetizira konstruktor i što to znači u praksi za performanse našeg programa.

Treće poglavlje nas informira o rukovanju podacima članovima klasa.

Četvrto poglavlje se fokusira na raznolikost funkcija članica klase sa detaljnim osvrtom na virtualne funkcije.

Zahvaljujem profesoru Mladenu Juraku na pomoći i savjetima pri izradi rada.

Poglavlje 1

Općenito o Objektnom Modelu

1.1 Objektni Model

U C++-u postoje dva tipa podataka članova klase: statički i nestatički, te tri tipa funkcija članica klase: statičke, nestatičke i virtualne.
Promotrimo danu klasu Tiger:

```
class Tiger
{
private:
  int height;
  static int tiger_count;
  virtual void print() const;
public:
  Tiger(int height);
  virtual  ~Tiger();
  int get_height();
  static int TigerCount();
};
```

Kako je klasa Tiger prikazana unutar stroja? To jest, kako modeliramo različite tipove podataka i funkcija članica?

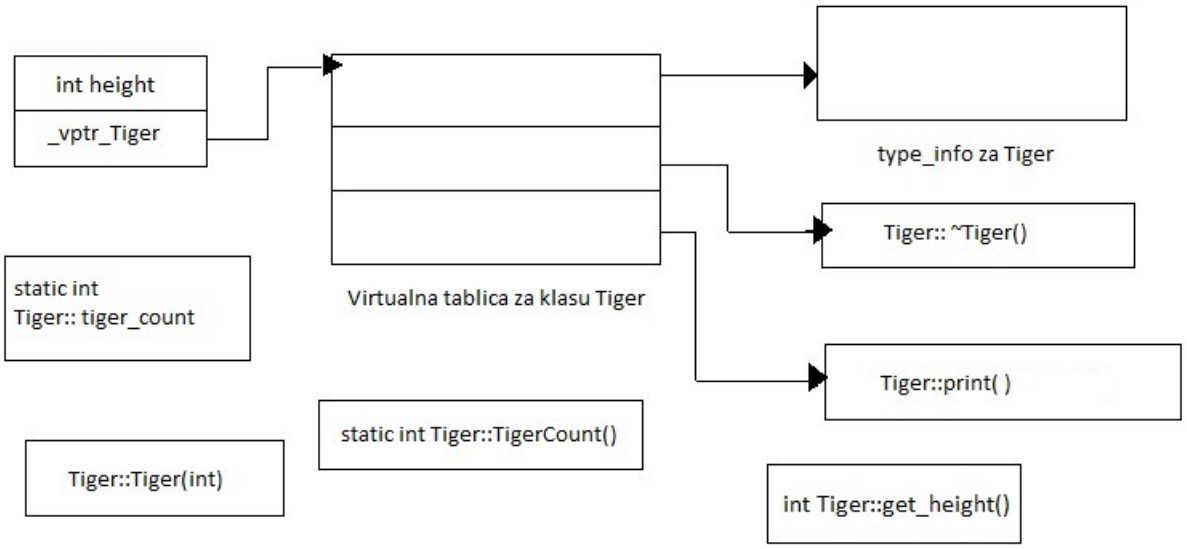
C++ Objektni Model

Nestatički podaci članovi klase su alocirani direktno unutar svakog objekta klase. Statički podaci su pohranjeni izvan pojedinog objekta klase. Statičke i nestatičke funkcije članice

klase su također spremljene izvan objekta klase. Virtualne funkcije su podržane u dva koraka:

1. Tablica pokazivača na virtualne funkcije je generirana za svaku klasu (takozvana *virtualna tablica*).

2. Po jedan pokazivač na virtualnu tablicu je umetnut unutar svakog objekta klase (tradicionalno, taj se pokazivač naziva *vptr*). Postavljanje *vptr* pokazivača se provodi automatski kroz kod generiran unutar konstruktora, destruktora i operatora pridruživanja svake klase.



Slika 1.1: C++ Objektni Model

Slika 1.1 ilustrira općeniti C++ Objektni Model za klasu `Tiger`.
Primarna snaga C++ Objektnog Modela je njegova učinkovitost prilikom izvođenja programa i dodjeljivanja prostora. Najveći nedostatak je potreba za ponovnim prevođenjem nepromijenjenog koda koji koristi objekt klase kojem je dodavan, odstranjivan ili modificiran nestatički podatak.

Dodavanje nasljeđivanja

C++ podržava jednostruko i višestruko nasljeđivanje. Štoviše, nasljeđivanje može biti definirano kao virtualno. U slučaju virtualnog nasljeđivanja, samo se jedno pojavljivanje nadklase sačuva (nazvano *podobjekt*) bez obzira koliko se puta iz nje izvodi nova klasa u lancu nasljeđivanja.

Izvorni model nasljeđivanja podržan C++ -om podatke članove nadklase sprema direktno unutar objekta podklase. Ovakav pristup nudi najkompaktniji i najučinkovitiji pristup članovima nadklase. Nedostatak je da svaka promjena članova nadklase, kao što je mijenjanje, odstranjivanje ili promjena tipa člana, zahtjeva da se sav kod koji koristi objekte nadklase ili bilo koju klasu izvedenu iz nje mora ponovno prevesti.

1.2 Razlikovanje objekata

Programski jezik C++ poznaje i podržava tri obrasca programiranja:

1. *Proceduralni model* koji nam je poznat iz programskog jezika C je podržan unutar C++-a

2. *Model apstraktnog tipa podataka* gdje je korisnicima osiguran set operacija (javno sučelje) dok implementacija ostaje skrivena.

3. *Objektno-orjentirani model* gdje je skup povezanih tipova enkapsuliran kroz apstraktnu baznu klasu nudeći zajedničko sučelje.

Programi napisani po isključivo jednom od ovih obrazca imaju tendenciju da se dobro ponašaju. Mješanje obrazaca, međutim, ima veći potencijal za iznenađenje, pogotovo kad je to miješanje nepažljivo.
Iako se objektom nadklase u hijerarhiji nasljeđivanja može manipulirati direktno ili indirektno, samo indirektna manipulacija preko pokazivača ili reference podržava polimorfizam nužan za objektno-orjentirano programiranje. Nasljeđivanje koje nije javno i pokazivači tipa `void*` se mogu smatrati polimorfizmom ali oni su bez izravne potpore jezika, to jest, o njima programer mora brinuti eksplicitnom konverzijom tipa.

Dodavanje polimorfizma

Definirajmo klasu `Tiger` kao izvedenu klasu klase `Animal`, naravno kroz javno nasljeđivanje:

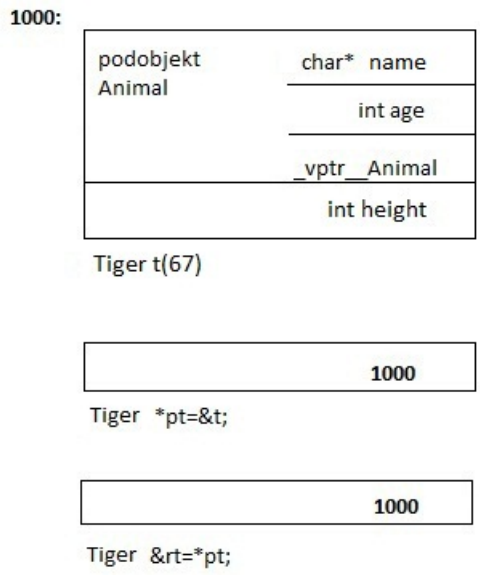
```
class Animal{
public:
  virtual  int get_age()=0;
  virtual  void set_age(int a)=0;
  virtual  char* get_name()=0;
  virtual  void set_name(char* n)=0;
  virtual void print();
  //...
protected:
  char* name;
  int age;
};

class Tiger : public Animal{
public:
  Tiger();
  ~Tiger();
  //...
  void print();
protected:
  int height;
  };

Tiger t(67);
Tiger *pt=&t;
Tiger &rt=*pt;
```

Što možemo reći o zahtjevu na memoriju koji postavljaju `t`, `pt` i `rt`?
I pokazivač i referenca zahtjevaju jednu memorijsku riječ za pohranu (4 bajta na 32-bitnom procesoru). `Tiger` objekt zahtjeva 16 bajtova (veličina objekta `Animal` je 12 bajtova plus 4 bajta koje uvodi `Tiger`). Moguć izgled objekta u memoriji je dan slikom 1.2

Ako pretpostavimo da je objekt `Tiger` smješten u memoriji na lokaciji 1000, koje su stvarne razlike između pokazivača na objekt `Animal` i pokazivača na objekt `Tiger`?

```
Tiger t;
Animal *pa = &t;
```

```
Tiger *pt = &t;
```



Slika 1.2: Mogući izgled objekta `Tiger` u memoriji

I `pa` i `pt` adresiraju prvi bajt objekta `Tiger`. Razlika je da je raspon adrese `pt` cijeli `Tiger` objekt, dok je raspon adrese `pa` samo `Animal` podobjekt objekta `Tiger`.
`pa` ne može direktno pristupiti ni jednom članu osim onih prisutnih u `Animal` podobjektu, osim preko virtualnog mehanizma.

```
//nelegalno: height nije član klase Animal iako pa adresira Tiger objekt
pa->height;

//ok: eksplicitna promjena tipa
(static_cast< Tiger* >( pa ))-> height;

//ok: height je član klase Tiger
pt->height;
```

Kad napišemo

```
pa->print();
```

tip objekta kojeg `pa` adresira određuje koja će se instanca funkcije `print()` pozvati. Zašto

se onda kad napišemo

```
Tiger t;
Animal pa=t;

//poziva se Animal::print()
pa.print();
```

poziva instanca `print()` klase `Animal`, a ne klase `Tiger`? Štoviše, ako inicijalizacija jednog člana drugim kopira vrijednosti jednog objekta u drugi, zašto *vptr* ne adresira virtualnu tablicu klase `Tiger`?

Odgovor na drugo pitanje je da prevodioc intervenira u inicijalizaciju i priruživanje jednog objekta drugom. Prevodioc mora osigurati da ako objekt sadrži jedan ili više *vptr* pokazivača te pokazivače ne inicijalizira ili ne mijenja objekt kojeg kopiramo.

Odgovor na prvo pitanje je da `pa` nije objekt tipa `Tiger`. Polimorfizam, potencijal da objekt bude više od jednog tipa, nije fizički moguć s objektima kojima direktno pristupamo. Kad objekt nadklase direktno inicijaliziramo objektom podklase, objekt podklase se ”izreže” tako da stane u raspoloživ memorijski prostor određen za tip nadklase.

Poglavlje 2

Semantika konstruktora

Razmatrati ćemo četiri slučaja u kojim semantika kopiranja bit-po-bit ne vrijedi za klase i u kojim se defaultni konstruktor, ako je nedeklariran, smatra netrivijalnim. U tim uvjetima, prevodioc, u odsutnosti deklariranog defaultnog konstruktora, mora sintetizirati konstruktor da bi se objekt klase mogao korektno inicijalizirati.

2.1 Konstrukcija defaultnog konstruktora

Konstruktor je posebna članska funkcija koja služi inicijalizaciji objekta. Njegovo ime je jednako imenu klase i nema povratni tip. U klasi može biti više konstruktora koji se razlikuju po broju i tipu ulaznih parametara. *Defaultni konstruktor* za klasu je konstruktor koji može biti pozvan bez parametara. Ako programer sam ne deklarira niti jedan konstruktor, defaultni konstruktor će biti implicitno deklariran. Implicitno deklariran konstruktor je `inline public` član svoje klase. Konstruktor je *trivijalan* ako je implicitno deklariran kao default kostruktor i ako :

- njegova klasa nema virtualnih funkcija i njegova nadklasa nije virtualna
- sve direktne nadklase njegove klase imaju trivijalne konstruktore
- za sve nestatičke članove njegove klase koji su tipa neke klase, svaka takva klasa ima trivijalni konstruktor

Inače je konstruktor *netrivijalan*. Četiri su uvjeta pod kojima prevodioc treba sintetizirati defaultni konstruktor za klase koje uopće ne deklariraju konstruktor. To su implicitni netrivijalni defaultni konstruktori. Sljedeća četiri odjeljka razmatraju uvjete pod kojima je defaultni konstruktor netrivijalan.

Klasa s varijablom članicom tipa klase

Ako klasa bez ijednog konstruktora sadrži varijablu članicu tipa klase s defaultnim konstruktorom, implicitni defaultni konstruktor te klase je netrivijalan i prevodioc treba sintetizirati defaultni konstruktor za nadklasu. Ta se sinteza, međutim, događa samo ako konstruktor stvarno treba dozvati.
Na primjer, u sljedećem fragmentu koda, prevodioc stvara defaultni konstruktor za klasu `Tiger`:

```
class Print{
public:
    Print();
    Print(char*);
    ...
};
class Tiger{
public:
    Print print;
    int height;
    ...
}
void tiger.print(){
  //Tiger::print se mora tu inicijalizirati
  Tiger t;
  if(t.height){}...
}
```

Sintetizirani defaultni konstruktor sadrži kod potreban za pozivanje defaultnog konstruktora klase `Print` za član `Tiger::print`, ali ne generira kod za inicijalizaciju `Tiger::height`. Inicijalizacija `Tiger::print` je odgovornost prevodioca; inicijalizacija `Tiger::height` programerova. Da bi se program izvršavao korektno, varijablu `height` također treba inicijalizirati. Pretpostavimo da programer osigura inicijalizaciju `height`-a kroz sljedeći defaultni konstruktor:

```
//programer je definirao defaultni konstruktor
Tiger::Tiger(){
    height = 33;
}
```

Sad su potrebe programa ispunjene, ali potreba da se inicijalizira objekt `print` još uvijek postoji. Zato što je defaultni konstruktor eksplicitno definiran, prevodioc ne može sintetizirati drugu instancu.

Nadklasa s defaultnim konstruktorom

Slično, ako je klasa bez ijednog konstruktora nasljeđena iz klase koja sadrži defaultni konstruktor, defaultni konstruktor podklase se smatra netrivijalnim i treba biti sintetiziran. Sintetizirani defaultni konstruktor podklase poziva defaultne konstruktore svih neposrednih nadklasa redom, onako kako su deklarirane.
Što ako programer osigura više konstruktora ali ne i defaultni konstruktor? Prevodioc proširuje svaki konstruktor kodom potrebnim da bi se pozvali svi potrebni defaultni konstruktori. Međutim, ne sintetizira defaultni konstruktor zbog prisutnosi ostalih, od programera osiguranih, konstruktora. Ako prisutni objekti članovi klase imaju defaultne konstruktore, ti konstruktori se također pozivaju - nakon poziva konstrukora svih nadklasa.

Klasa sa virtualnom funkcijom članicom

Postoje dva dodatna slučaja u kojima je sintetizirani defaultni konstruktor potreban:

1. Klasa deklarira (ili nasljeđuje) virutalnu funkciju
2. Izvedena klasa se nalazi u lancu nasljeđivanja gdje je jedna ili više nadklasa virtualna

U oba slučaja, odsutnost deklariranih konstruktora zahtjeva sintezu defaultnog. Na primjer, dan je sljedeći dio koda:

```
class Animal {
public:
    virtual void get_name()=0;
    ...
};
void get_name(const Animal& animal){
      animal.get_name();
}
// klase Tiger i Lion su nasljeđene iz klase Animal

void  neka_fja(){
      Tiger t;
      Lion l;
      get_name(t);
      get_name(l);
}
```

Sljedeća dva proširenja klase se događaju tokom prevođenja:

1. Virtualna tablica funkcija se generira i popunjava adresama svih aktivnih virtualnih funkcija za tu klasu.

2. Unutar svakog objekta klase, dodatni pokazivač (*vptr*) se sintetizira i inicijalizira adresom virtualne tablice.

Da bi ovaj mehanizam djelovao, prevodioc mora inicijalizirati *vptr* pokazivač svakog `Animal` objekta (ili objekt klase koja je nasljeđena iz `Animal` klase) adresom odgovarajuće virtualne tablice.

Klasa sa virtualnom nadklasom

Implementacije virtualnih nadklasa široko variraju od prevodioca do prevodioca. Međutim, ono što je zajedničko svakoj implementaciji je potreba da unutar svake podklase lokacija virtualne nadklase bude dostupna za vrijeme izvođenja. Dan je sljedeći dio programskog koda:

```
class X {public: int i;};
class A: public virtual X { public: int j;};
class B: public virtual X { public: double d;};
class C: public A, public B { public: int k;};

//ne može se riješiti pitanje lokacije pa->X::i za vrijeme prevođenja
void foo(const A* pa){pa->i=1024;}

main(){
    foo(new A);
    foo(new C);
    //...
}
```

Prevodioc ne može riješiti fizički pomak podatka `X::i` unutar objekta kojem smo pristupili kroz `pa` unutar funkcije `foo()`, budući da stvarni tip može varirati sa svakim pozivom funkcije `foo()`. Prevodioc mora transformirati kod koji pristupa `X::i` tako da odluka bude odgođena do izvršavanja. U originalnoj implementaciji to je postignuto umetanjem pokazivača na svaku virtualnu nadklasu unutar svakog objekta podklase. Svako pristupanje preko reference ili pokazivača virtualnoj nadklasi je postignuto preko pridruženog pokazivača. Po toj strategiji funkcija `foo` bi mogla biti prerađena na ovaj način:

```
//moguća transformacija od strane prevodioca
void foo( const A* pa ) { pa-> _vbcX-> i= 1024;}
```

gdje _vbcX predstavlja od prevodioca generiran pokazivač na virtualnu tablicu klase X. Inicijalizacija _vbcX se ostvaruje za vrijeme konstrukcije objekta klase. Za svaki konstruktor koji klasa definira, prevodioc ubacuje kod koji dopušta pristup svakoj virtualnoj nadklasi za vrijeme izvođenja. U klasama koje ne deklariraju konstruktor, prevodioc mora sintetizirati defaultni konstruktor.

2.2 Konstrukcija konstruktora kopije

Konstruktor kopije je poseban konstruktor koji se koristi za stvaranje novog objekta kao kopije postojećeg objekta. Prvi parametar takvog konstruktora je referenca na objekt istog tipa kao i onaj koji se stvara (const ili ne-const), koji mogu slijediti parametri bili kojeg tipa ali svi sa defaultnim vrijednostima.

```
class Animal{
public:
    Animal(const Animal& t);
    ...
};
Animal t;
```

Eksplicitno se koristi kad definiramo novi objekt i inicijaliziramo ga objektom istog tipa

```
Animal t1(t);
Animal t2=t;
```

a implicitno kad funkciji (operaciji) šaljemo objekt

```
void f(Animal x);
...
f(t);
```

ili funkcija vraća objekt.

```
Animal g(...){
Animal ret;
...
return ret;
}
```

Defaultna inicijalizacija član-po-član

Što ako klasa ne deklarira eksplicitno konstruktor kopije? Tada se primjenjuje defaultna inicijalizacija član-po-član koja kopira vrijednost svakog ugrađenog tipa ili izvedenih tipova podataka (kao što su pokazivači ili nizovi) iz jednog objekta klase u drugi. Član klase koji je tipa neke klase nije kopiran već se na njega defaultna inicijalizacija član-po-član primjenjuje rekurzivno. Na primjer, promotrimo sljedeću deklaraciju klase:

```
class Animal{
public:
//...nema eksplicitno deklariranog konstruktora kopije
private:
     char* name;
     int age;
};
```

Defaultna inicijalizacija član-po-član jednog objekta `Animal` drugim, kao što je:

```
Animal t1("Misko",12);
Animal t2=t1;
```

se postiže kao da je svaki od podataka članova klase zasebno inicijaliziran:

```
//semantika ekvivalentna defaultnoj inicijalizaciji član-po-član
t2.name=t1.name;
t2.age=t1.age;
```

Ako objekt tipa `Animal` definiramo kao člana neke druge klase, kao na primjer:

```
class ZOO{
public:
// ...nema eksplicitno deklariranog konstruktora kopije
private:
    int animal_count;
    Animal t;
    ...
};
```

tada defaultna inicijalizacija član-po-član jednog objekta tipa `ZOO` kopira člana koji je ugrađeni tip (`animal_count`) i rekurzivno primjenjuje inicijalizaciju član-po-član na `Animal` objekt član klase.

Kao i kod defaultnog konstruktora postoje implicitno deklarirani i implicitno definirani

konstruktori kopije ako klasa ne deklarira nijedan konstruktor kopije. Kao i prije, postoje trivijalni i netrivijalni konstruktori kopije. U praksi, samo se netrivijalni sintetiziraju unutar programa. Kriterij za utvrđivanje je li konstruktor kopije trivijalan je taj da li pokazuje semantiku kopiranja biti-po-bit. O tome će biti riječi u sljedećem odjeljku.

Semantika kopiranja bit-po-bit

U sljedećem fragmentu koda:

```
#include Animal.h
Animal t1("Glupko",9);
...
void foo(){ Animal t2=t1; }
```

Očito je da je objekt `t2` inicijaliziran objektom `t1`. Ali bez poznavanja deklaracije klase `Animal`, nije moguće predvidjeti ponašanje programa koje će izazvati takva inicijalizacija. Ako kreator klase `Animal` definira konstruktor kopije inicijalizacija objekta `t2` ga poziva. Ako je klasa, međutim, bez eksplicitnog konstruktora kopije, poziv sintetiziranog konstruktora kopije ovisi o tome da li klasa pokazuje semantiku kopiranja bit-po-bit. Na primjer, sljedeći kod:

```
//deklaracija koja pokazuje semantiku kopiranja bit-po-bit
class Animal{
public:
    Animal( const char* name, int i);
    ~Animal(){ delete[] name; delete i}
    //...
private:
    int age;
    char* name;
};
```

Defaultni konstruktor kopije se ne treba sintetizirati jer deklaracija pokazuje semantiku kopiranja bit-po-bit. Međutim, sljedeća deklaracija klase `Animal` je ne pokazuje:

```
//deklaracija koja ne pokazuje semantiku kopiranja bit-po-bit
class Animal{
public:
    Animal( const String& name, int i);
    ~Animal(){}
```

```
    //...
private:
    int age;
    String name;
};
```

gdje klasa `String` deklarira eksplicitni copy konstruktor:

```
class String{
public:
    String(const char* );
    String( const String& );
    ~String();
    ...
};
```

U ovom slučaju, prevodioc treba sintetizirati konstruktor kopije da bi pozvao konstruktor kopije objekta klase String.

```
//sinetizirani konstruktor kopije
//pseudo kod
inline Animal::Animal(const Animal& t)
{
    name.String::String(t.name);
    age=t.age;
}
```

Netrivijalna semantika kopiranja

Klase ne pokazuje semantiku kopiranja bit-po-bit u četiri slučaja:

1. Kad sadrži objekt koji je tipa neke klase za koju konstruktor kopije postoji (ili eksplicitno deklariran ili sintetiziran)

2. Kad je izvedena iz klase za koju konstruktor kopije postoji (ponovno, ili eksplicitno deklariran ili sintetiziran)

3. Kad deklarira jednu ili više virtualnih funkcija

4. Kad je nasljeđena u lancu nasljeđivanja u kojoj su jedna ili više nadklasa virtualne

U slučaju 1 i 2, implementacija mora ubaciti pozive konstruktora kopije članskog objekta ili nadklase unutar sintetiziranog konstruktora kopije. Sintetizirani konstruktor kopije za klasu `Animal` iz prethodnog odjeljka ilustrira slučaj 1. Slučajevi 3 i 4 su malo suptilniji, na njih ću se kratko osvrnuti u sljedećim odjeljcima.

Resetiranje virtualne tablice

Tijekom prevođenja program se ”proširuje” dva puta kad god klasa deklarira jednu ili više virtualnih funkcija:

- Tablica virtualnih funkcija sadrži adrese svih aktivnih virtualnih funkcija povezanih sa klasom tj. virtualna tablica se generira
- Pokazivač na virtualnu tablicu se umeće unutar objekta svake klase (*vptr* pokazivač)

Očito, loše stvari bi se dogodile da prevodioc ne uspije inicijalizirati ili netočno inicijalizira *vptr* pokazivač za svaki novi objekt klase. Dakle, jednom kad prevodioc umetne *vptr* u klasu, dotična klasa više ne pokazuje semantiku kopiranja bit-po-bit. Štoviše, implementacija sad treba sintetizirati konstruktor kopije da bi propisno inicijalizirao *vptr*.
Definirajmo dvorazinsku hijerarhiju sastavljenu od klasa `Animal` i `Tiger`:

```
class Animal{
private:
  //podaci potrebni za Animal-ove verzije funkcija
  //get_age() i get_name()
public:
  Animal();
  virtual ~Animal();
  virtual void set_age();
  virtual void set_name();
  //...
};

class Tiger: public Animal{
private:
  //podaci potrebni za Tiger-ove verzije funkcija
  //get_age() i get_name()
public:
  Tiger();
  void set_age();
  void set_name();
```

```
  virtual void set_height();
};
```

Inicijalizacija objekta klase `Tiger` drugim objektom tipa klase `Tiger` je izravna i moglo bi se implementirati sa semantikom kopiranja bit-po-bit. Naprimjer:

```
Tiger t1;
Tiger t2=t1;
```

`t1` je inicijaliziran defaultnim `Tiger` konstruktorom. Unutar tog konstruktora, kodom koji umeće prevodioc, *vptr* objekta `t1` je inicijaliziran da pokazuje na virtualnu tablicu klase `Tiger`. Sigurno je, stoga, jednostavno kopirati vrijednost `t1` *vptr*-a u `t2` *vptr*, međutim, to prestaje biti sigurno kad je objekt nadklase inicijaliziramo objektom podklase. Na primjer:

```
Animal a1=t1;
```

vptr vezan sa `a1` ne smije biti inicijaliziran tako da pokazuje na virtualnu tablicu klase `Tiger` (a to bi bio rezultat da se koristi izravno kopiranje bit po bit). Inače bi se poziv funkcije `set_name()` u sljedećem dijelu programskog koda raspao kad bi mu proslijedili objekt `a1`:

```
void set_name( const Animal& x){ x.set_name(); }
void foo(){
    // a1 vptr mora adresirati virtualnu tablicu klase Animal
    // a ne virtualnu tablicu Tiger na koju pokazuje t1 vptr

    Animal a1=t1;

    set_name( t1 );// poziva Tiger::set_name()
    set_name( a1 );// poziva Animal::set_name()
}
```

Sintetizirani konstruktor kopije klase `Animal` eksplicitno postavlja *vptr* objekta na virtualnu tablicu klase `Animal` radije nego da ga kopira od objekta na desnoj strani.

Rukovanje podobjektom virtualne bazne klase

Inicijalizacija jedne klase drugom u kojoj je podobjekt virtualne nadklase također poništava semantiku kopiranja bit-po-bit. Svaka implementacija koja podupire virtualno nasljeđivanje uključuje potrebu da se adresa podobjekta virtualne nadklase unutar objekta podklase učini

dostupnom za vrijeme izvođenja. Održavanje integriteta te lokacije je zadaća prevodioca. Semantika kopiranja bit-po-bit bi mogla rezultirati korupcijom te lokacije, pa prevodioc mora uskočiti sa svojim sintetiziranim konstruktorom kopije. Na primjer, u sljedećoj deklaraciji klase `Lion` koja je izvedena iz klase `Animal`:

```
class Lion: public virtual Animal{
public:
    Lion() { //inicijalizacija privatnih podataka ... }
    Lion( int weight ) { //inicijalizacija privatnih podataka ... }
    //...
private:
    //svi potrebni podaci...
};
```

U oba konstruktora klase `Lion` se umeće od prevodioca generirani kod koji poziva defaultni konstruktor klase `Animal` koji inicijalizira *vptr* pokazivač `Lion` objekta. Što je sa inicijalizacijom član-po-član? Prisutnost virtualne nadklase poništava semantiku kopiranja bit-po-bit. Problem nije u inicijalizaciji objekta objektom iste klase, već u inicijalizaciji objekta objektom njegove podklase. Na primjer, promotrimo slučaj kad objekt klase `Lion` inicijaliziramo objektom `Liger`, gdje je klasa `Liger` deklarirana sa:

```
class Liger: public Lion{
public:
    Liger() { //inicijalizacija privatnih podataka ... }
    Liger( int height ) {//inicijalizacija privatnih podataka ... }
private:
    //potrebni podaci...
};
```

Ponovno, u slučaju inicijalizacije jednog `Lion` objekta drugim, dovoljna je obična semantika kopiranja bit-po-bit:

```
//obična semantika kopiranja bit-po-bit
Lion misko;
Lion tupko=misko;
```

Međutim, pokušaj inicijalizacije objekta `tupko` objektom tipa `Liger` zahtjeva intervenciju prevodioca da bi pristup podobjektu klase `Animal` bio korektan.

```
//obično kopiranje bit-po-bit nije dovoljno
//prevodioc mora eksplicitno inicijalizirati
// vptr pokazivač nadklase Animal
```

```
Liger mali;
Lion tupko= mali;
```

U ovom slučaju, da bi se korektno inicijalizirao objekt `tupko` prevodioc mora sintetizirati konstruktor kopije, umećući kod koji inicijalizira *vptr* pokazivač nadklase, izvesti potrebne inicijalizacije članova klase i ostale zadaće rukovanja memorijom.

Poglavlje 3

Semantika podataka

U ovom poglavlju, podaci članovi klasa i hijerarhija klasa imaju centralnu ulogu. Podaci unutar klase, općenito, predstavljaju stanje klase u nekoj točki izvođenja programa. Nestatički podaci sadržavaju vrijednosti pojedinih objekata klase; statički podaci sadrže vrijednosti koje su od interesa za klasu kao cjelinu.
Prikaz nestatičkih podataka u objektnom modelu jezika C++ optimizira prostor i vrijeme dohvaćanja tako da ih sprema direktno unutar svakog objekta klase. To je istinito i za nasljeđene nestatičke podatke članove klase, i virtualnih i nevirtualnih nadklasa. Statički podaci članovi klase ostaju unutar globalnog segmenta podataka programa i ne utječu na veličinu pojedinog objekta klase. Samo jedna instanca statičkog podatka postoji unutar programa bez obzira na to koliki se broj puta instancira objekt klase. Svaka klasa je, dakle, točno one veličine koja je potrebna da bi sadržavala nestatičke podatke članove. Ta veličina može povremeno iznenaditi zato što je veća od potrebne, to se događa zbog:

1. Dodavanja dodatnih članova podataka tokom prevođenja radi podupiranja neke funkcionalnosti jezika (primarno virtualnih funkcija)

2. Potrebe poravnavanja članova podataka i struktura kao cjeline

Raspored podataka članova klase

Dan je sljedeći programski kod:

```
class Tiger{
public:
  //...
private:
  char* name;
  int age;
```

```
  int height;
  static int tiger_count;
};
```

Nestatički podaci su poredani unutar objekta onim redom kojim su deklarirani unutar klase (svako pojavljivanje statičkih podataka se ignorira, u našem primjeru to je `tiger_count`). Statički podaci članovi klase se pohranjuju u segmentu podataka programa koji je neovisan o bilo kojoj klasi. Dodatno, prevodioc može sintetizirati jedan ili više dodatnih unutanjih podataka da bi podržao Objektni Model. *vptr*, na primjer, je jedan takav sintetizirani podtak koji sve trenutne implementacije umeću unutar svakog objekta koji sadrži jednu ili više virtualnih funkcija. Gdje smjestiti *vptr* unutar objekta? Tradicionalno, smještan je nakon svih eksplicitno deklariranih članova klase. U novije vrijeme , smješta se na početak klase. Standard[1] prepušta prevodiocu slobodu[2] da ubaci `vptr` bilo gdje, čak i između članova koje je deklarirao programer.

Standard također dopušta prevodiocu da poreda podatke članove u višestrukim `public`, `private` i `protected` dijelovima klase kako "smatra" da je najbolje. To jest, ako je dana sljedeća deklaracija:

```
class Tiger{
public:
  //...
private:
  char* name;
private:
  int age;
private:
  int height;
  static int tiger_count;
};
```

veličina i sastav objekta klase će biti isti kao i u prethodnoj deklaraciji, ali poredak članova ovisi o implementaciji . Implementaciji je slobodna da stavi na prvo mjesto `name`, `age` ili `height`.

[1] ISO/IEC IS 14882:2003(E)

[2] Precizna lokacija vptr-a ovisi o implementaciji. Neki prevodioci, na primjer, Visual C++ i C++ Builder, ga smještaju na prvu poziciju unutar objekta, prije podataka članova klase koje deklarira dizajner programa. Drugi prevodioci, kao što su GCC and DEC CXX, smještaju vptr na kraj objekta, nakon svih podataka koje je deklarirao programer.

3.1 Pristup podacima članovima klase

Dane su sljedeće dvije naredbe:

```
Tiger t1;
t1.age=10;
```

Razumno je zapitati se koja je cijena pristupa podatku `age`? Odgovor ovisi o tome kako su `age` i klasa `Tiger` deklarirani. `age` može biti statički ili nestatički podatak. Klasa `Tiger` može biti neovisna klasa ili naslijeđena klasa. Sljedeći odjeljci će se baviti svakom od tih mogućnosti ali prije nego krenemo na sljedeći odjeljak postavlja se jedno pitanje. Neka su zadane definicije objekata `t2` i `pt`,

```
Tiger t2, *pt=&t2;
```

je li dohvat podatka `age`, kao što je

```
t2.age=7;
pt->age=7;
```

bitno drugačiji preko objekta `t2` nego što je to preko pokazivača `pt`? Na to pitanje dajemo odgovor na kraju ovog odjeljka.

Statički podaci članovi klase

Statički podaci članovi klasa su doslovno izvađeni iz svojih klasa i tretira ih se kao da su deklarirani kao globalne varijable (ali im je vidljivost ograničena na doseg klase). Točno jedna instanca svakog statičkog podatka klase je pohranjena na segmentu podataka programa. Svaka referenca na statički član se prevodi kao da je direktna referenca na tu određenu vanjsku varijablu.

```
t2.tiger_count==250;
```

i

```
pt->tiger_count==250;
```

se transformiraju u:

```
Tiger::tiger_count==250;
```

Ovo je jedini slučaj u jeziku gdje je pristup članu kroz pokazivač i kroz objekt potpuno jednak u terminima instrukcija koje se izvršavaju. To je zato što je pristup statičkom podatku članu preko operatora dohvata samo sintaktička pogodnost. Član se ne nalazi unutar objekta klase, pa stoga objekt klase nije ni nužan za dohvat podatka.
Ako je `tiger_count` naslijeđeni član kompleksne hijerarhije nasljeđivanja opet postoji samo jedna instanca tog člana unutar programa i pristupa joj se direktno.
Rezultat dohvaćanja adrese statičkog člana je običan pokazivač na podatak tipa statičkog člana, a ne pokazivač na člana klase s obzirom da statički član nije sadržan unutar objekta klase. Na primjer,

```
&Tiger::tiger_count;
```

vraća stvarnu memorijsku adresu objekta

```
const int*
```

Ako dvije klase deklariraju statičku varijablu članicu istog imena, tada smještanje obje varijable u programski dio za podatke neće rezultirati konfliktom imena jer prevodilac sintetizira nova imena iz imena klase i imena varijable. (eng. name mangling)

Nestatički podaci članovi klase

Nestatički su podaci pohranjeni direktno unutar pojedinog objekta klase i ne može im se pristupati osim preko eksplicitnog ili implicitnog objekta klase. Implicitni objekt klase je prisutan svaki put kad programer pristupa direktno nestatičkim članovima klase u funkciji članici klase. Na primjer:

```
void Tiger::set_age(int _age)
{
      age=_age;
}
```

Naizgled direktan pristup varijabli `age` je ustvari izveden preko implicitnog objekta klase koji predstavlja **`this`** pokazivač. Prevodioc proširuje funkciju na sljedeći način:

```
void Tiger::set_age(Tiger *const this,int age)
{
      this->age=_age;
}
```

Pristup nestatičkom podatku klase zahtjeva dodavanje pomaka člana podatka unutar objekta početnoj adresi objekta. Na primjer, ako je dano

```
t2.age = 7;
```

adresa

```
&t2.age;
```

je ekvivalentna zbrajanju

```
&t2 + ( &Tiger::age -1 );
```

Uočimo oduzimanje za jedan primjenjeno na pomak člana `age` unutar objekta klase `Tiger`. Pomaci članova unutar objekata klase su uvijek povećani za jedan jer se tako omogućuje prevodiocu da razlikuje pokazivač na početak objekta klase i pokazivač na prazan objekt. (null-pokazivač)
Pomak svakog nestatičkog podatka člana klase unutar objekta klase je poznat za vrijeme prevođenja, čak i ako taj član pripada podobjektu nadklase koja je nasljeđena jedostruko ili višestruko.

Sjećate li se pitanja sa početka odjeljka: je li dohvat podatka `age`, kao što je

```
t2.age=7;
pt->age=7;
```

bitno drugačiji preko objekta `t2` nego što je to preko pokazivača `pt`?
Odgovor je: Pristup podatku je bitno drugačiji ako je `Tiger` nasljeđena klasa koja u lancu nasljeđivanja ima virtualnu nadklasu i član kojem pristupamo, kao na primjer `age`, je naslijeđeni član te virtualne bazne klase. U tom slučaju, ne možemo sa sigurnošću reći koji tip pokazivač `pt` adresira (i stoga ne možemo znati točnu poziciju traženog člana unutar klase) pa dohvat mora biti odgođen do izvršavanja programa.

3.2 Nasljeđivanje i podaci članovi klase

U C++ modelu nasljeđivanja, objekt podklase je prikazan kao niz svojih članova i članova nadklasa. Stvaran poredak dijelova nadklasa i podklasa je neodređen, prevodioc ima slobodu odrediti poredak elemenata. U praksi, članovi nadklase se pojavljuju prvi, osim u slučaju virtualne nadklase. (Općenito, rukovanje virtualnom nadklasom je iznimka svih generalizacija pa tako i ove.)
U sljedećim odjeljcima ćemo promatrati sljedeće klase:

```
class Animal{
private:
  char* name;
```

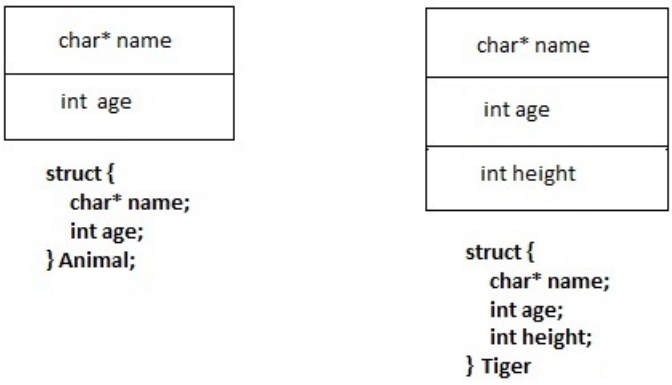
```
  int age;

public:
  //konstruktori
  //operacije
  //funkcije pristupa
  //...
};

class Tiger{
private:
  char* name;
  int age;
  int height;
public:
  //konstruktori
  //operacije
  //funkcije pristupa
  //...
};
```

Promatrati ćemo utjecaje jednostrukog nasljeđivanja bez virtualnih funkcija, jednostrukog nasljeđivanja sa virtualnim funkcijama, višestruko nasljeđivanje i virtualno nasljeđivanje. Slika 3.1 prikazuje raspored podataka klasa `Animal` i `Tiger`. (Bez prisustva virtualnih funkcija taj prikaz je jednak prikazu strukture u C-u.)

Nasljeđivanje bez polimorfizma

Zamislimo da programer želi dijeliti implementaciju klasa `Animal` i `Tiger` ali i nastaviti koristiti specifične funkcionalnosti. Jedna dizajnerska strategija je da izvede klasu `Tiger` iz klase `Animal`, s tim da `Tiger` naslijedi sve operacije i zadrži `name` i `age` članove.

```
class Animal{
protected:
  char* name;
  int age;

public:
  Animal(char* _name = " ", int _age = 0 ):
    :name( _name ) , age( _age ) {}
```

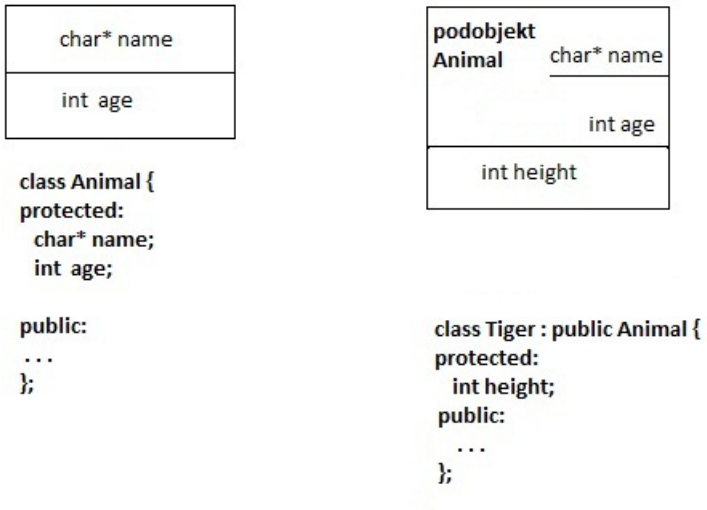


Slika 3.1: Nezavisne strukture

```
  void operator=( const Animal& rhs){
        name= rhs.get_name();
        age=rhs.get_age()
    }

  int get_age(){ return age; }
  char* get_name(){ return name; }

  void set_age(int _age){ age=_age; }
  void set_name(char* _name){ name=_name; }

  // još članova
};

class Tiger: public Animal{
protected:
  int height;
public:
```

```
  Tiger(int _height=0 )
    :height = _height {}

  void operator=( const Tiger& rhs){
     Animal::operator=( rhs );
     height= rhs.get_height();
    }

  int get_height(){ return height; }
  void set_height(int _height){ height = _height; }

  //još članova
};
```

Deklaracija i korištenje klasa nisu promijenjeni pa korisnici ne trebaju znati da li su objekti nezavisni ili povezani nasljeđivanjem. Slika 3.2 prikazuje raspored podataka unutar klasa `Animal` i `Tiger` bez deklaracije sučelja.



Slika 3.2: Jednostruko nasljeđivanje bez virtualnih funkcija

Koje su moguće zamke transformiranja dviju nezavisnih klasa u dvije karike u lancu nasljeđivanja? Jedna je moguće udvostručenje broja funkcijskih poziva da bi izveo istu operaciju

što se može spriječiti tako da se sve bitne funkcije deklariraju kao umetnute (`inline`). Druga moguća zamka rastavljanja klase u dvo-razinsku ili dublju hijerarhiju je moguće napuhavanje memorijskog prostora potrebnog za prikaz hijerarhije klasa. Počnimo konkretnom klasom:

```
class Concrete{
public:
   //...
private:
   int val;
   char c1;
   char c2;
   char c3;
};
```

Na 32-bitnom računalu (duljina riječi je 4 bajta), veličina svake instance klase `Concrete` je 8 bajtova:

1. 4 bajta za `val`
2. po 1 bajt za `c1`, `c2`, `c3`
3. 1 bajt za poravnavanje klase da stane u jednu riječ na računalu[3]

Recimo da smo odlučili klasu `Concrete` podijeliti tro-razinsku hijerarhiju nasljeđivanja:

```
class Concrete1{
public:
   //...
protected:
   int val;
   char c1;
};

class Concrete2 : public Concrete1{
public:
   //...
protected:
   char c2;
```

[3]Na većini računala, za složene strukture postoje ograničenja na to kako su pohranjene u memoriji da bi se mogle lakše dohvaćati i spremati.

```
};

class Concrete3 : public Concrete2{
public:
   //...
protected:
   char c3;
};
```

Sa stajališta dizajna, ovaj prikaz možda ima više smisla. Sa stajališta implementacije, međutim, možda će nas zabrinuti činjenica da je veličina objekta klase `Concrete3` 16 bajtova - duplo više nego prije.
Što se dogodilo? Klasa `Concrete1` sadrži 2 člana, `val` i `c1`, koji zajedno zauzimaju 5 bajtova. Veličina objekta `Concrete1` je, međutim, 8 bajtova: 5 bajtova za stvarnu veličinu plus 3 bajta za punjenje tako da bi objekt popunio riječ u memoriji. `Concrete2` dodaje samo jedan nestatički podatak, c2, tipa `char`. Neiskusni programer bi očekivao da se sve pohrani u prikaz klase `Concrete1`, da zauzme 1 bajt, a da se ostatak potroši na popunjavanje riječi. Ovakava strategija popunjavanja memorije bi činila klasu `Concrete2` objektom od 8 bajtova, od čega su 2 bajta potrošena na poravnavanje. Ustvari, raspored klase `Concrete2` sačuva 3 bajta poravanavanja unutar klase `Concrete1`. Član se `c2` sprema u iduću riječ, 1 bajt zauzima on, a ostala 3 bajta se koriste za popunjavanje riječi. Veličina objekta klase `Concrete2` je 12 umjesto 8 bajtova, od čega je 6 bajtova utrošeno na popunjavanje riječi. Kad isti princip primjenimo na klasu `Concrete3`, objekt te klase zauzima 16 bajtova, od čega je 9 potrošeno na popunjavnje.
Ovakvo ponašanje bi nam se moglo učiniti nepotebnim pa stoga promotrimo sljedeći niz pokazivača:

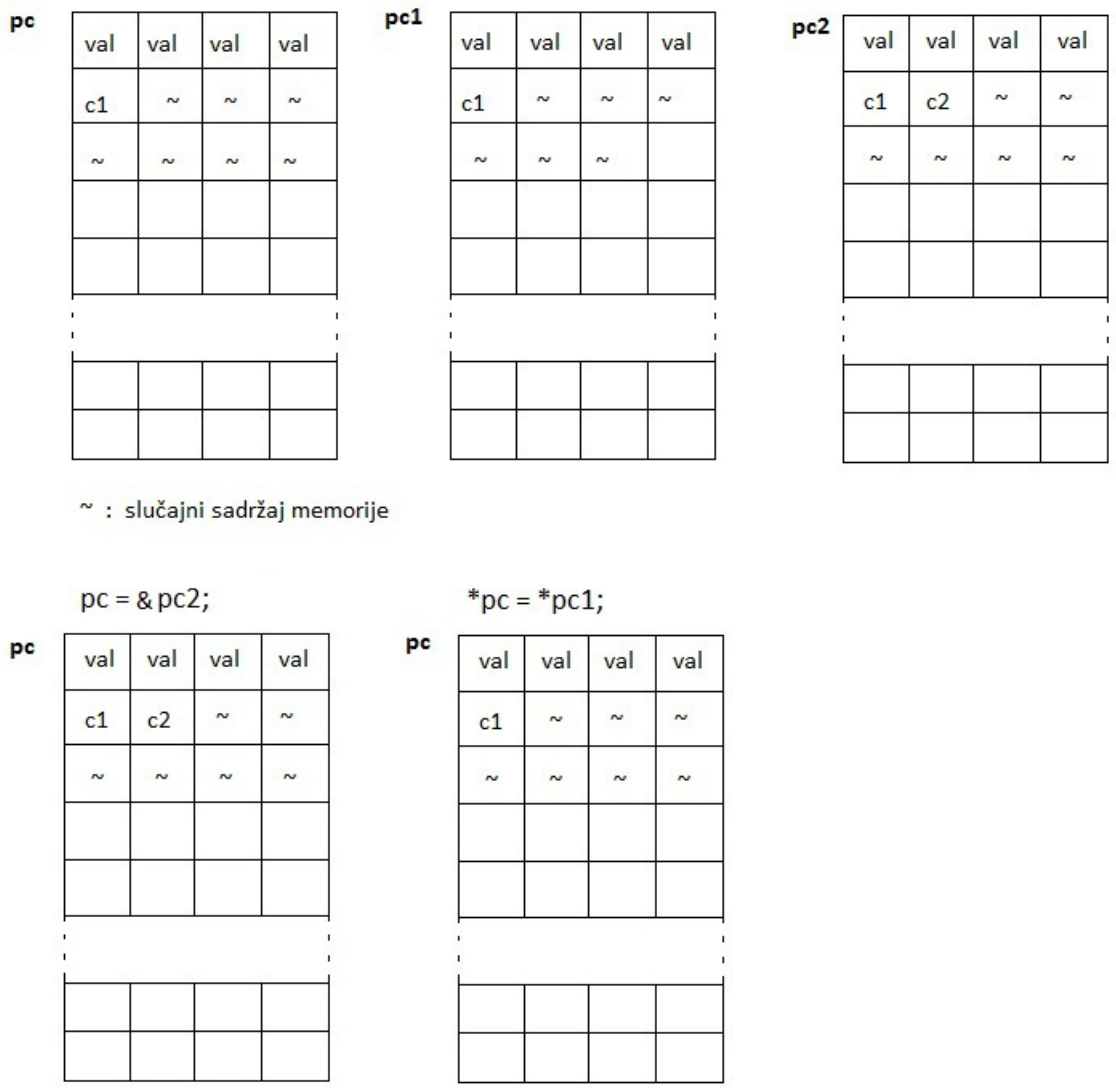
```
Concrete2 pc2;
Concrete1 *pc; *pc1;
```

I `pc` i `pc1` mogu adresirati bilo koji objekt tri dane klase. Sljedeće pridruživanje:

```
*pc=*pc1;
```

bi trebalo defaultno kopirati član-po-član adresiranog objekta klase `Concrete1`. Ako `pc` adresira objekt klase `Concrete2` ili `Concrete3`, neće biti posljedica ako mu pridružimo podobjekt klase `Concrete1`.

Međutim, ako su članovi `c2` podklase `Concrete2` i `c3` podklase `Concrete3` spremljeni unutar podobjekta nadklase `Concrete1` ta semantika jezika ne bi bila sačuvana. Pridruživanje kao što je:



Slika 3.3:

```
//Neka pc1 pokazuje na Concrete2 objekt
pc1 = &pc2;

// ups: podobjekt klase Concrete1 je prepisan
// c2 ima neodređenu vrijednost
// izrezivanje podobjekta
```

```
*pc = *pc1
```

bi prepisalo spremljene vrijednosti nasljeđenih članova. Otkrivanje ovakve greške bi iziskivalo velik trud programera, u najmanju ruku. Tu situaciju ilustrira slika 3.3.

Dodavanje polimorfizma

Ako želimo rukovati objektom neovisno o tome je li to objekt klase `Animal` ili klase `Tiger`, trebamo osigurati sučelje našoj hijerarhiji virtualnim funkcijama. Pogledajmo što se dogodi kada napravimo:

```
class Animal{
protected:
  char* name;
  int age;

public:
  Animal(char* _name = " ", int _age = 0 ):
     :name( _name ) , age( _age ) {}

  void operator=( const Animal& rhs){
        name= rhs.get_name();
        age=rhs.get_age()
  }

  int get_age(){ return age; }
  char* get_name(){ return name; }

  void set_age(int _age){ age=_age; }
  void set_name(char* _name){ name=_name; }
  //rezerviramo mjesto za height - ne radi ništa...

  virtual int get_height(){return 0; }
  virtual void set_height(int) {}

  // još članova
};
```

Ima smisla uvesti sučelje u dizajn samo ako namjeravamo rukovati objektima klase `Animal` i `Tiger` polimorfno, to jest:

```
void foo (Animal &p1, Animal &p2){
//...
p1=p2;
//...
}
```

gdje p1 i p2 mogu biti ili klase Animal ili klase Tiger. Ta fleksibilnost je bit objektno orjentiranog programiranja. Da bi se osigurala ta fleksibilnost moraju se uvesti neki noviteti:

- Uvođenje virtualne tablice za klasu Animal koja će sadržavati sve virtualne funkcije koje dotična klasa deklarira.
- Uvođenje *vptr* pokazivača u svaki objekt klase Animal.
- Proširenje konstruktora inicijalizacijom pokazivača *vptr* na virtualnu tablicu klase.
- Proširenje destruktora tako da resetira *vptr* koji adresira virtualnu tablicu klase.

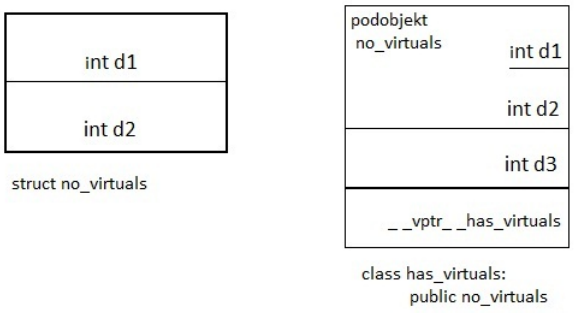
Klasa Tiger:

```
class Tiger: public Animal{
protected:
  int height;
public:
  Tiger(char* _name= " ", int _age=0, int _height=0 )
     : Animal( _name, _age ) height = _height {}

  // funkcije koje rukuju s članovima name i age ostaju iste
  // invarijantne za na tip: nisu pretvorene u virtualne


  void operator=( const Animal& rhs){
        Animal::operator=( rhs );
        height= rhs.get_height();
  }

  int get_height(){ return height; }
  void set_height(int _height) { height= _height; }

  //još članova
};
```

Iako se sintaksa deklaracije klase nije promijenila, sve oko nje je različito. Dvije `get_height()` članske funkcije su virtualne instance. Svaki objekt klase `Tiger` sadrži dodatni *vptr* pokazivač (naslijeđen iz klase `Animal`). Postoji i `Tiger` virtualna tablica. Poziv virtualnih funkcija članica je složeniji (O tome više u četvrtom poglavlju).

Jedno zanimljivo pitanje je gdje je najbolje smjestiti `vptr` pokazivač? U originalnoj implementaciji je bio smješten na kraju objekta klase kako bi podržao obrazac nasljeđivanja prikazan na slici 3.4:

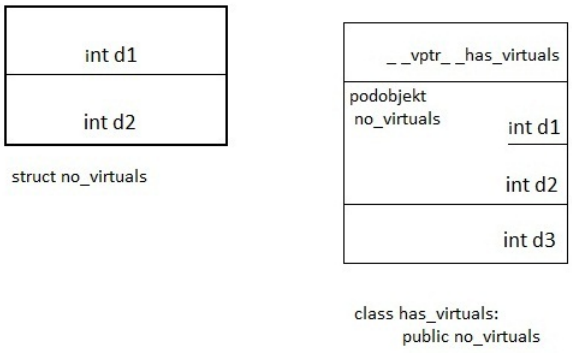
```
struct no_virtuals{
  int d1, d2;
};
class has_virtuals : public no_virtuals{
public:
  virtual void foo();
  //...
private:
  int d3;
};

No_viruals *p = new has_virtuals;
```



Slika 3.4: *vptr* je smješten na kraj klase

Smještanje *vptr*-a na kraj klase čuva izgled nadklase, koja je C-ovska struktura, unutar objekta nasljeđene klase i tako dopušta njezinu upotrebu unutar C-ovskog koda. Međutim, taj način programiranja je bio češći kad je C++ tek uveden nego u ovom trenutku.
Kasnije se, da bi se poduprlo višestruko nasljeđivanje i virtualne nadklase, *vptr* se počeo

smještati na početak objekta klase. Slika 3.5 nam ilustrira taj slučaj.
Smještanje *vptr*-a na početak objekta klase je korisnije za podršku poziva virtualnih funkcija preko pokazivača pod višestrukim nasljeđivanjem. Nedostatak je gubitak sposobnosti zajedničkog funkcioniranja C-ovskog načina programiranja sa C++-ovskim.



Slika 3.5: *vptr* je smješten na početak klase

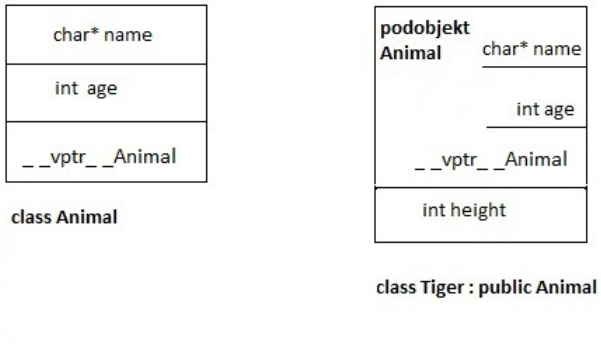
Višestruko nasljeđivanje

Jednostruko nasljeđivanje pruža neki oblik ”prirodnog” polimorfizma s obzirom na konverziju između nadklase i podklase. Sa slike 3.6 se vidi da se objekt nadklase unutar objekta podklase nalazi na početku, duljinom se razlikuju samo u nestatičkim podacima. Pridruživanje kao što je

```
Tiger t;
Animal *a=&t;
```

objekta podklase pokazivačuu ili referenci nadklase (bez obzira na dubinu lanca nasljeđivanja) ne zahtjeva intervenciju prevodioca ili modifikaciju adrese.

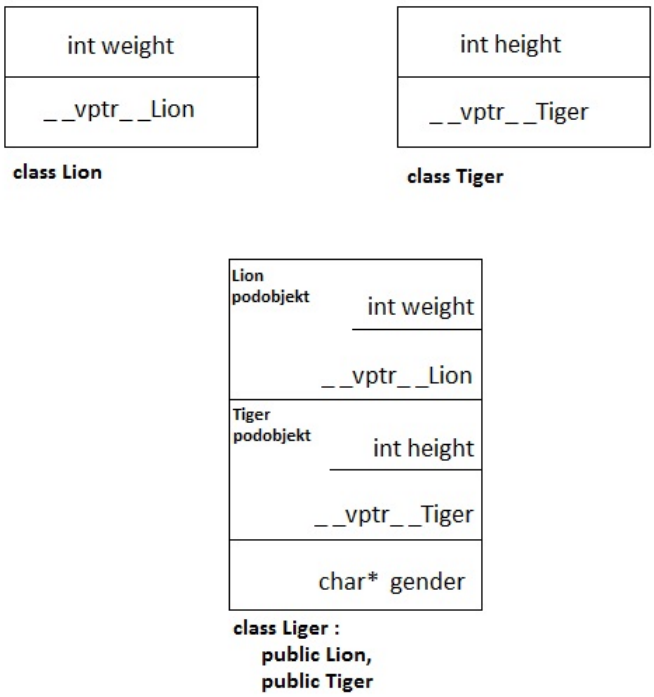
Sa slike 3.5 uočimo da smještanje `vptr`-a na početak objekta klase lomi prirodan polimorfizam jednostrukog nasljeđivanja u specijalnom slučaju nadklase bez virtualnih funkcija i podklase koja ih ima. U ovom slučaju konverzija objekta podklase u objekt nadklase zahtjeva intervenciju prevodioca kako bi se namjestila adresa početka podobjekta nadklase. Pod višestrukim i virtualnim nasljeđivanjem potreba intervencije prevodioca postaje sve izraženija.



Slika 3.6: Jednostruko nasljeđivanje

Višestruko nasljeđivanje se ne ponaša jednako dobro niti ga je jednako lako oblikovati kao jednostruko nasljeđivanje. Kompleksnost višestrukog nasljeđivanja leži u "neprirodnoj" vezi podklase i podobjekata njene druge po redu nadklase. Pogledajmo na primjer višestruko naslijeđenu klasu `Liger`:

```
class Lion{
public:
  //...
protected:
  int weight;
};
class Tiger{
public:
  //...
protected:
  int height;
};
class Liger: public Lion, public Tiger{
public:
  //...
protected:
  char* gender;
};
```



Slika 3.7: Višestruko nasljeđivanje

Pridruživanje adrese višestruko nasljeđenog objekta pokazivaču lijeve (to jest prve) nadklase je jednako kao i kod jednostrukog nasljeđivanja, s obzirom da oba pokazuju na istu adresu. Cijena je samo pridruživanje adrese. Pridruživanje adrese druge nadklase, međutim, zahtjeva da se adresa modificira dodavanjem veličine prve nadklase. Na primjer:

```
Liger liger;
Tiger *p_tiger;
Lion *p_lion;
```

pridruživanje

```
p_tiger=&liger;
```

zahtjeva konverziju tipa

```
//C++ pseudo kod
p_tiger=(Tiger*)(((char*)&liger)+sizeof(Lion));
```

dok pridruživanje

```
p_lion= &liger;
```

zahtjeva samo kopiranje adrese.
Neka je

```
Liger *p_liger;
Tiger *p_tiger;
```

pridruživanje

```
p_tiger=p_liger;
```

se ne može jednostavno pretvoriti u

```
//C++ pseudo kod
p_tiger=(Tiger*)(((char*)p_liger)+sizeof(Lion));
```

jer ako je `p_liger` postavljen na vrijednost 0, `p_tiger` bi završio sa vrijednošću `sizeof(Lion)`.
Pa, za pokazivače, unutarnja konverzija zahtjeva dodatni test:

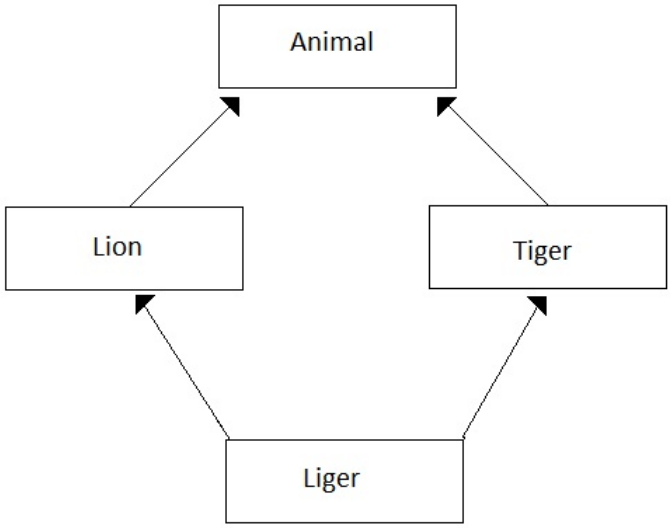
```
//C++ pseudo kod
p_tiger =p_liger ? (Tiger*)(((char*)p_liger)+sizeof(Lion)) : 0;
```

Konverzija reference ne zahtjeva takvu provjeru budući da referenca ne može pokazivati na prazan objekt.

Virtualno nasljeđivanje

Dan je sljedeći dio virtualne hijerarhije nasljeđivanja:

```
class Animal{
public:
  //...
protected:
  char* name;
  int age;
};

class Lion : public virtual Animal{
public:
  //...
```



Slika 3.8: Hijerarhija nasljeđivanja klase `Liger`

```
protected:
  int weight;
};

 class Tiger : public virtual Animal{
public:
  //...
protected:
  int height;
};

class Liger: public Lion, public Tiger{
public:
  //...
protected:
  char* gender;
};
```

Slika 3.8 ilustrira danu hijerarhiiju.
Koliko god da se semantika virtualnog nasljeđivanja činila kompliciranom, njena podrška

unutar prevodioca se pokazala još kompliciranijom. Implementacijsko rješenje je sljedeće. Klasa koja sadrži jedan ili više podobjekata virtualnih nadklasa, u našem primjeru `Liger`, dijeli se na dva dijela: invarijanti i zajednički dio, pri čemu zajednički dio odgovara dijeljenim virtualnim bazama. Podaci unutar invarijantnog dijela ostaju na fiksnom pomaku od početka objekta bez obzira na sva sljedeća nasljeđivanja. Tako podacima unutar invarijantnog dijela možemo pristupati direktno. Zajednički dio predstavljaju podobjekti virtualnih nadklasa. Lokacija podataka unutar zajedničkog dijela oscilira sa svakim novim nasljeđivanjem, tako da podacima unutar zajedničkog dijela treba pristupati indirektno. Ono što varira među implementacijama je metoda indirektnog pristupa. Na prethodnom primjeru ćemo ilustrirati tri prevladavajuće strategije. Općenita strategija rasporeda unutar objekta je prvo smjestiti invarijantni dio nasljeđene klase, a nakon toga izgraditi zajednički dio.

Međutim, jedan problem ostaje: Kako da implementacija ostvari pristup zajedničkom dijelu klase? U prvotnoj implementaciji C++ prevodioca, umetnut je pokazivač na svaku virtualnu nadklasu unutar svakog objekta podklase. Pristup nasljeđenim članovima virtualne nadklase se postiže indirektno preko tih pokazivača. Na primjer, sljedeći operator u klasi `Tiger` :

```
void Tiger::
operator= ( const Tiger &rhs )
{
   name= rhs.name;
   age= rhs.age;
   height= rhs.height;
}
```

transformira se u:

```
// pseudo kod
_vbcAnimal->name= rhs._vbcAnimal->name;
_vbcAnimal->age= rhs._vbcAnimal->age;
height= rhs.height;
```

Konverzija između instanci podklase i nadklase, kao što je

```
Tiger *ptiger = pliger;
```

postaje

```
// pseudo kod
Tiger *ptiger= pliger ? pliger-> _vbcTiger : 0;
```

Dvije su slabosti ovog implementacijskog modela:

1. Objekt klase sadržava dodatan pokazivač za svaku virtualnu nadklasu. Idealno, željeli bismo konstatno povećanje objekta klase, neovisno o broju virtualnih nadklasa unutar njegove hijerarhije nasljeđivanja.

2. Kako se hijerarhija virtualnog nasljeđivanja produbljuje, tako se stupanj zaobilaženja povećava. To znači da tro-razinska hijerarhija virtualnog nasljeđivanja zahtjeva prolazak kroz tri pokazivača virtualnih nadklasa. Idealno, željeli bi konstantno vrijeme pristupa bez obzira na dubinu hijerarhije nasljeđivanja.

Metaware[4] i ostali prevodioci koji koriste originalni implementacijski model rješavaju drugi problem tako da kopiraju sve pokazivače virtualnih nadklasa u objekt podklase. To rješava problem nepromjenljivog vremena pristupa, iako na račun dvostrukog broja pokazivača. Slika 3.9 ilustrira pokazivač-na-virtualnu-nadklasu implementacijski model.

Dva su općenita rješenja prvog problema. Microsoftov prevodioc je uveo tablicu virtualnih nadklasa. Svaki objekt klase sa jednom ili više virtualnih nadklasa ima umetnut pokazivač na virtualnu tablicu nadklasa. Pokazivači na virtualne nadklase su, naravno, smješteni u toj virtualnoj tablici. Iako je ovo rješenje poznato dugi niz godina nijedan drugi prevodioc ga ne koristi, možda zato što je to Microsoftov patent.
Drugo rješenje je smjestiti ne adrese već pomak virtualne nadklase unutar virtualne tablice funkcija. Slika 3.10 ilustrira ovaj implementacijski model. Virtualna tablica funkcija je indeksirana i sa pozitivnim i sa negativnim indeksima. Pozitivni indeksi indeksiraju skup virtualnih funkcija, negativni indeksi čuvaju pomake virtualnih nadklasa. Pod ovom strategijom naš `Tiger` operator se transformira u sljedeću općenitu formu:

```
//pseudo kod
(this + _vptr_Tiger[-1])->name= ( &rhs + rhs._vptr_Tiger[-1])->name;
(this + _vptr_Tiger[-1])->age= ( &rhs + rhs._vptr_Tiger[-1])->age;
height= rhs.height;
```

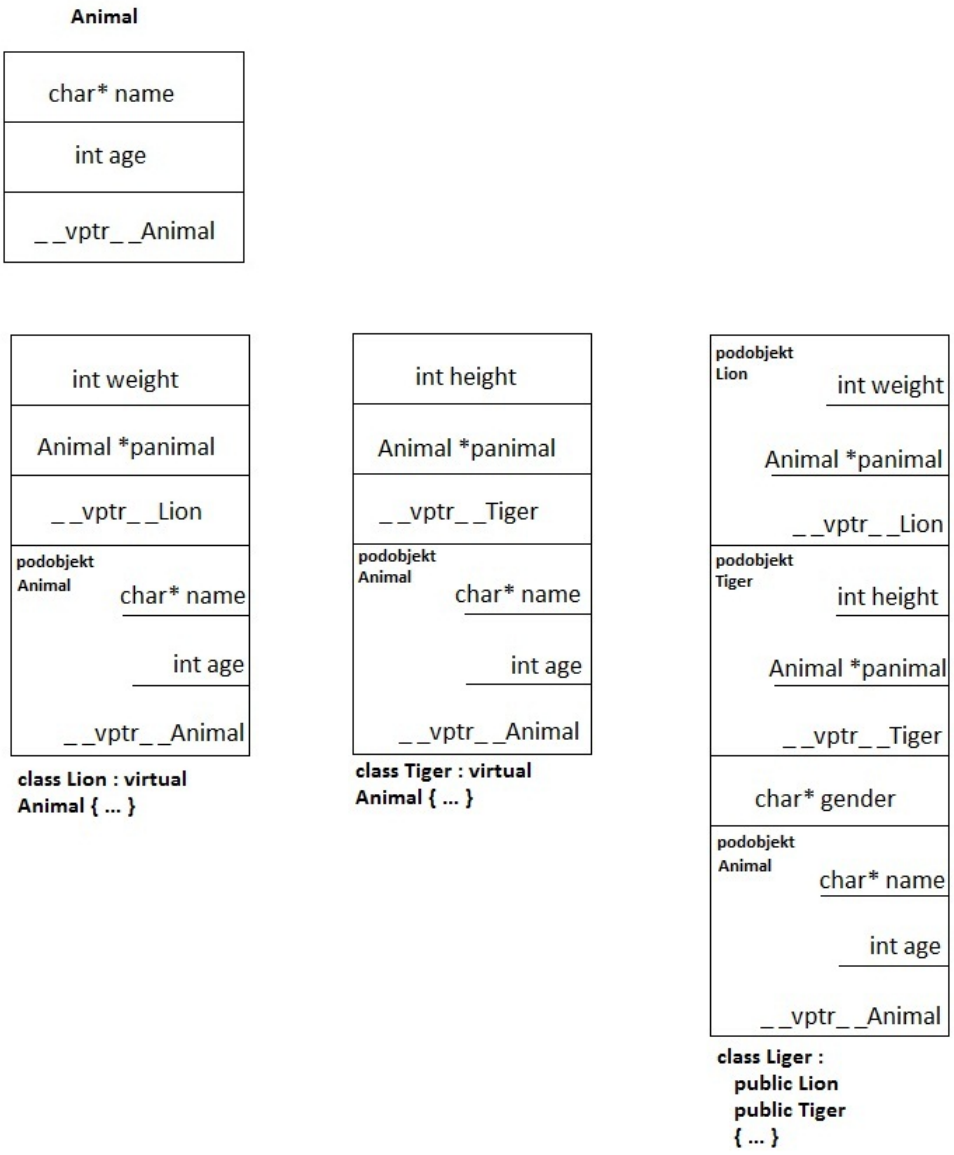
Iako je pristup nasljeđenim članovima skuplji, cijena tog pristupa je ograničena na korištenje članova. Konverzija između podklase i nadklase kao što je

```
Tiger *ptiger = pliger;
```

pod ovom implementacijom postaje

```
//pseudo kod
Tiger *ptiger = pliger ? pliger + pliger->_vptr_Tiger[-1] : 0 ;
```

[4] http://www.arc.com/software/development/metawarecompiler.html



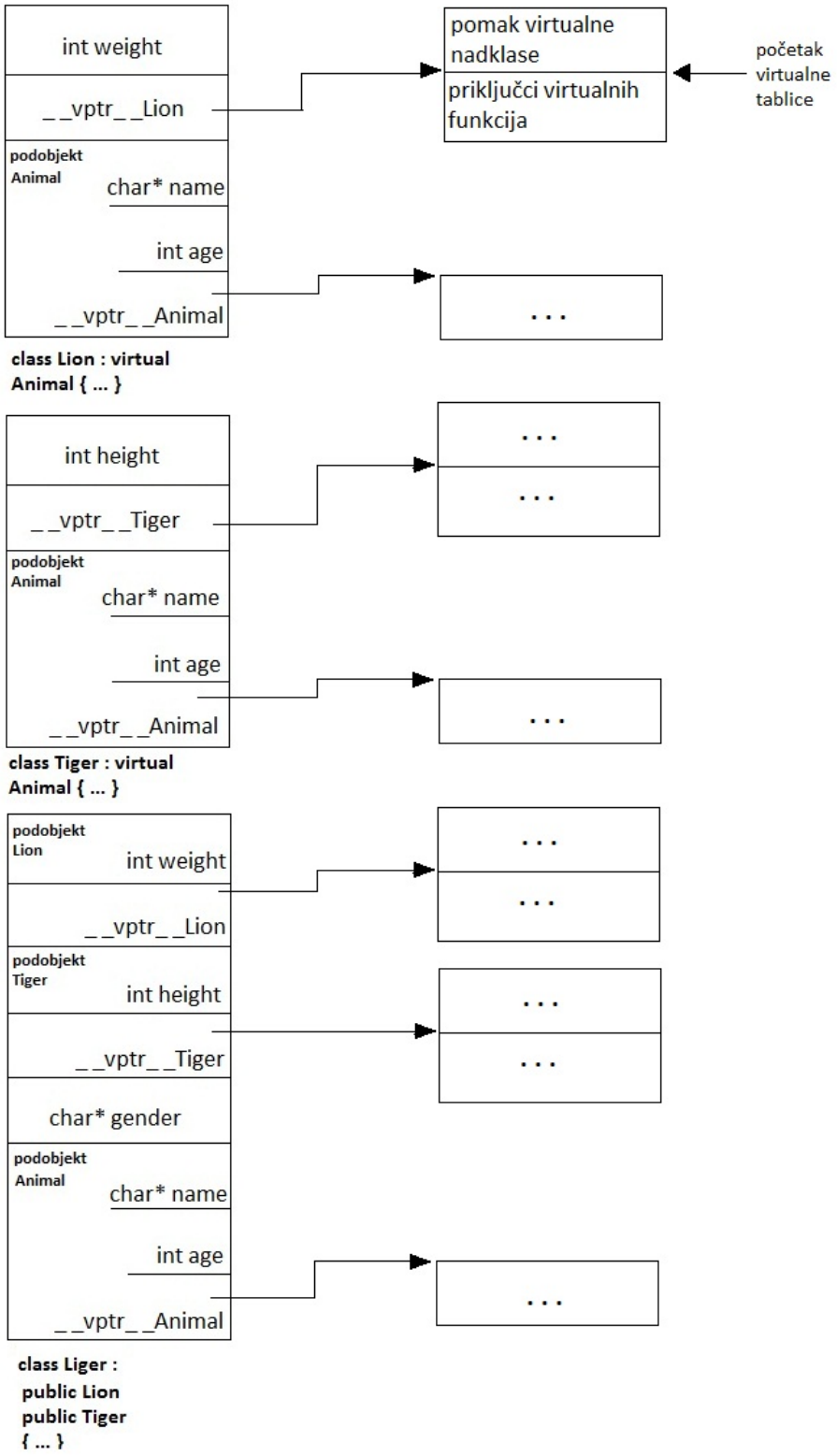
Slika 3.9: Virtualno nasljeđivanje s pointer strategijom

Gdje

`pliger->_vptr_Tiger[-1] : 0 ;`

predstavlja pomak virtualne nadklase `Tiger` unutar objekta `Liger`.

Sve prije navedeno su implementacijski modeli, Standard ne obavezuje na njih.



Slika 3.10: Virtualno nasljeđivanje sa strategijom pomaka virtualne tablice

Poglavlje 4

Semantika funkcija

C++ podržava tri tipa funkcija članica: statičke, nestatičke i virtualne. Svaka se poziva drugačije; o tim razlikama će biti riječi u sljedećim odjeljcima.

4.1 Raznolikost poziva članica funkcija

Nestatičke funkcije članice

Jedan C++ dizajnerski kriterij je da nestatičke funkcije članice klase moraju biti učinkovite najmanje kao i analogne funkcije nečlanice. To jest, ako nam je dan izbor

```
int Tiger::get_height() const {...}

int get_height2(const Tiger *_this){...}
```

Ne bi trebalo biti razloga koji bi nas naveli da odaberemo funkciju nečlanicu. To se postiže transformiranjem funkcije članice u ekvivalentnu nečlansku instancu.
Na primjer, neka je dana implementacija nečlanske funkcije `get_height2()`:

```
int get_height2(const Tiger *_this)
    {
        return _this->height;
    }
```

Vizualnom inspekcijom, možemo vidjeti da nečlanska instanca izgleda manje efikasna. Ona pristupa podatku `height` indirektno preko pokazivača dok članska instanca pristupa direktno:

```
int Tiger::get_height() const { return height; }
```

U praksi su, međutim, funkcije članice u unutrašnjosti transformirane da budu ekvivalentne nečlanskoj instanci. To se postiže sljedećim koracima:

1. Preraditi deklaraciju da bi se ubacio dodatni argument u člansku funkciju koji pruža pristup objektu klase koji poziva funkciju članicu. To se naziva implicitni **this** pokazivač:

   ```
   //proširenje nekonstantne nestatičke funkcije
   int Tiger::get_height( Tiger *const this)
   ```

   Ako je funkcija deklarirana kako **const**, deklaracija postaje

   ```
   //proširenje nestatičke funkcije
   int Tiger::get_height(const Tiger *const this)
   ```

2. Preraditi svaku naredbu direktnog pristupa nestatičkom članu klase tako da mu se pristupa preko pokazivača **this**.

   ```
   {
       return this->height;
   }
   ```

3. Preraditi člansku funkciju tako da postane vanjska , promijenti joj ime tako da bude leksikografski jedinstvena unutar progama:

   ```
   extern int get_height_Tiger3dFv(register Tiger *const this);
   ```

Sad kad je funkcija transformirana, svaki poziv funkcije također mora biti promijenjen. Dakle,

```
t.get_height();
```

postaje

```
get_height_Tiger3dFv( &t );
```

a

```
ptr->get_height();
```

postaje

```
get_height_Tiger3dFv( ptr );
```

Virtualne funkcije članice

Da je get_height() virtualna funkcija, poziv:

```
ptr->get_height();
```

bi bio transformiran u

```
(* ptr -> vptr [1]) (ptr) ;
```

Gdje :

- *vptr* predstavlja unutrašnje generiran pokazivač na virtualnu tablicu koji je umetnut unutar svakog objekta čija klasa deklarira ili nasljeđuje jednu ili više virtualnu funkciju. (U praksi, ime mu može biti drugačije. Možemo imati *vptr*-a unutar kompleksne hijerarhije klasa.)

- 1 je indeks u virtualnoj tablici koja je pridružena priključku funkcije `get_height()`

- Drugo pojavljivanje pokazivača `ptr` predstavlja `this` pokazivač

Slično, ako je `print()` virtualna funkcija, njezin poziv unutar `get_height ()` bi bio transformiran:

```
// register void  print();
register void (*this->vptr [2]) (this);
```

U ovom slučaju, jer se `Tiger::print()` poziva unutar `Tiger::get_height()` (čiji je poziv već razriješen kroz virtualni mehanizam), eksplicitno pozivanje `Tiger` instance (i tako potiskujući nepotrebno ponovno pozivanje kroz virtualni mehanizam) je učinkovitije:

```
//eksplicitno pozivanje potiskuje virtualni mehanizam
register void Tiger::print();
```

To bi bilo puno učinkovitije da je `print()` deklarirana kao inline. Eksplicitno pozivanje virtualnih funkcija koristeći `operator ::` se rješava na isti način kao i nestatičke funkcije članice:

```
register void print_7Tiger3dFv ( this);
```

Statičke funkcije članice

Ako je `Tiger::get_height()` statička funkcija članica, onda bi oba poziva

```
obj.get_height();
ptr->get_height();
```

prevodioc transformirao u pozive "običnih" funkcija nečlanica kao što je

```
//obj.get_height()
  get_height_7Tiger3dFv();

//ptr->get_height();
  get_height_7Tiger3dFv();
```

Prije uvođenja statičkih funkcija članica, jezik je zahtjevao da se sve funkcije članice pozivaju kroz objekt te klase. U praksi, objekt klase je potreban kad se jednom ili više nestatičkih podataka članova klase pristupa direktno preko funkcija članica. Objekt klase osigurava vrijednost `this` pokazivača za taj poziv. `This` pokazivač povezuje nestatičke članove klase kojima se pristupa preko funkcija članica klase sa članovima koje sadrži objekt. Ako se članovima ne pristupa direktno nema potrebe za `this` pointerom. Nema potrebe pozivati funkciju članicu s objektom. Jezik, u to vrijeme, nije prepoznavao taj slučaj. To je stvaralo anomaliju u pristupanju statičkim podacima članovima. Ako je dizajner te klase deklarirao statički član podatak kao `private` ili `protected`, što se smatara dobrim stilom programiranja, tada bi dizajner također trebao osigurati jednu ili više članskih funkcija za pristup podacima. Prema tome, iako bi netko mogao pristupiti statičkom članu podatku neovisno o objektu klase, poziv funkcije pristupa zahtjeva da te funkcije budu vezane za objekt klase. Pristup neovisan o objektu klase je posebno važan kad dizajner klase želi podržati slučaj/stanje kad nema objekata dane klase.

Rješenje jezika je uvođenje statičkih članica funkcija. Primarna karakteristika statičkih članica funkcija je ta da su bez `this` pokazivača. Sljedeće sekundarne karakteristike su sve izvedene iz primarne:

- statička funkcija članica ne može pristupiti nestatičkim članovima svoje klase.
- ne može biti deklarirana `const`, `volatile`, ili `virtual`.
- ne treba biti pozvana kroz objekt svoje klase, iako može.

Korištenje operatora dohvata (`operator .`) je notacijska pogodnost; Poziv statičke funkcije:

```
if( obj.tiger_count() >1 ) ...
```

transformira se u direktan poziv:

```
if( Tiger:: tiger_count() >1 ) ...
```

Dohvaćanje adrese statičke članske funkcije uvijek rezultira vrijednošću njezine lokacije u memoriji, to jest, njezinom adresom. Zato što je statička funkcija članica bez `this` pokazivača, tip njezine adrese nije pokazivač na člansku funkciju već tip pokazivača na funkciju nečlanicu. To jest:

```
&Tiger::tiger_count();
```

rezultira tipom

```
unsigned int (*) ();
```

a ne tipom

```
unsigned int ( Tiger::*) ();
```

4.2 Virtualne funkcije članice

Već smo vidjeli opći model implementacije virtualnih funkcija: specifična virtualna tablica za svaku klasu koja sadrži adrese skupa aktivnih virtualnih funkcija te klase i *vptr* koji adresira tablicu umetnut unutar svakog objekta te klase. U ovom odjeljku ćemo razmotriti moguće dizajne koji će nas dovesti do našeg općenitog modela i taj model pod jednostrukim, višestrukim i virtualnim nasljeđivanjem.

Da bi se podržao mehanizam virtualnih funkcija, neka vrsta određivanja tipa za vrijeme izvođenja koja se primjenjuje na polimorfne objekte treba biti podržana.
To jest ako imamo poziv

```
ptr-> print();
```

treba pridružiti neke informacije `ptr` pokazivaču koje će biti dostupne za vrijeme izvođenja tako da točna instanaca `print()` može biti identificirana, pronađena i pozvana.
Najneposrednije ali i najskuplje rješenje je tražene informacije dodati `ptr`-u. Slijedeći ovu strategiju, pokazivač sadrži dva dijela informacije:

1. Adresu objekta na koji se odnosi
2. Neke kodirane podatke o tipu objekta ili adresu strukture koja sadrži tu informaciju

Problem s ovim rješenjem je dvostruk. Prvo, dodaje značajno prostora na račun korištenja pokazivača `ptr`. Drugo, lomi se veza kompatibilnosti sa C-om. Ako se dodatna informacija ne može smjestiti unutar pokazivača, sljedeće logično mjesto za spremanje tih podataka je sam objekt. Ovo ograničava spremanje na one objekte koji to trebaju. Ali koji objekti u stvarnosi trebaju tu informaciju? Za početak kriterij ćemo ograničiti samo na klase. Ako klasa uistinu treba tu informaciju, tamo je, ako ne, nije. Kad je ta informacija potrebna? Očito, kad se neka polimorfna operacija treba podržati za vrijeme izvođenja programa. U C++-u , polimorfizam se očituje kao moguće adresiranje pokazivača nadklase objektom podklase. Na primjer

```
Animal *ptr;
```

možemo pridružiti `ptr` pointeru ili `Tiger` objekt

```
ptr = new Tiger;
```

ili `Liger` objekt

```
ptr = new Liger;
```

polimorfizam `ptr` pokazivača se primarno očituje kao transportni mehanizam kojim se mogu prenositi skupovi javno nasljeđenih tipova kroz program. Ovaj tip polimorfne podrške koji se može nazvati pasivnim, osim u slučaju virtualne nadklase ostvaruje se za vrijeme prevođenja. Polimorfizam postaje aktivan kad se adresirani objekt stvarno i koristi. Pozivanje virtualne funkcije je jedna takva upotreba:

```
// primjer aktivnog polimorfizma

ptr->print();
```

Naš problem smo ograničili na identificiranje klasa koje demonstriraju polimorfizam i zato zahtjevaju dodatne informacije prilikom izvođenja programa. Jedini sigurni način identificiranja klasa koje bi mogle podržati polimorfizam je prisutnost jedne ili više virtualnih funkcija. Tako prisutnost najmanje jedne virtualne funkcije postaje kriterij za identificiranje klasa koje zahtjevaju dodatne informacije.

Sljedeće logično pitanje je, kojih dodatnih informacija?
Ako imamo poziv

```
ptr-> print();
```

gdje je `print()` virtualna funkcija , koje informacije su potrebne da bi se pozvala točna instanca funkcije `print()`?
Trebamo znati:

- stvarni tip objekta kojeg `ptr` adresira. To nam omogućava da izaberemo točnu instancu funkcije `print()`

- lokaciju te instance da bi je mogli pozvati

Implementacija bi mogla dodati dva člana svakom polimorfnom objektu klase

1. String ili numeričku reprezentaciju tog tipa

2. Pokazivač na neku tablicu koja sadrži lokacije virtualnih funkcija

Kako bi se mogla konstruirati ta tablica koja sadrži adrese virtualnih funkcija?
U C++-u, je skup virtualnih funkcija koje se može pozvati preko objekta vlastite klase poznat za vrijeme prevođenja. Štoviše, taj je skup nepromjenjiv. Za vrijeme izvođenja se virtualne instance ne mogu dodavati ili zamjenivati. Tablica, dakle, služi samo kao pasivni repozitorij. S obzirom da se ni veličina ni sadržaj tablice ne mijenjaju tokom izvođenja programa, njezina konstrukcija i pristup može biti u potpunosti prepuštena prevodiocu. Dostupnost adresa za vrijeme izvršavanja je, međutim, samo pola rješenja. Druga polovica je traženje adrese. To se postiže u dva koraka:

1. naći tablicu i umetnuti unutarnje generirani pokazivač na tablicu u svaki objekt klase

2. naći adresu funkcije, svakoj virtualnoj funkciji je pridružen fiksni indeks

Za to sve se brine prevodioc. Sve što je preostalo za napraviti prilikom izvođenja programa je pozvati funkciju adresiranu unutar odeđenog priključka virtualne tablice.

Za svaku klasu se generira jedna tablica. Svaka tablica sadrži adrese svih aktivnih instanci virtualnih funkcija objekta klase. Te aktivne funkcije se sastoje od sljedećeg:

- instance definirane unutar klase, koja prema tome možda prerađuje instancu nadklase

- instance nasljeđene od nadklase ako je podklasa ne preradi

- bibliotečnu instancu `pure_virtual_called()` koja služi i kao rezervacija mjesta za čistu virtualnu funkciju i kao izuzetak prilikom izvršavanja ako se instanca te funkcije pozove.

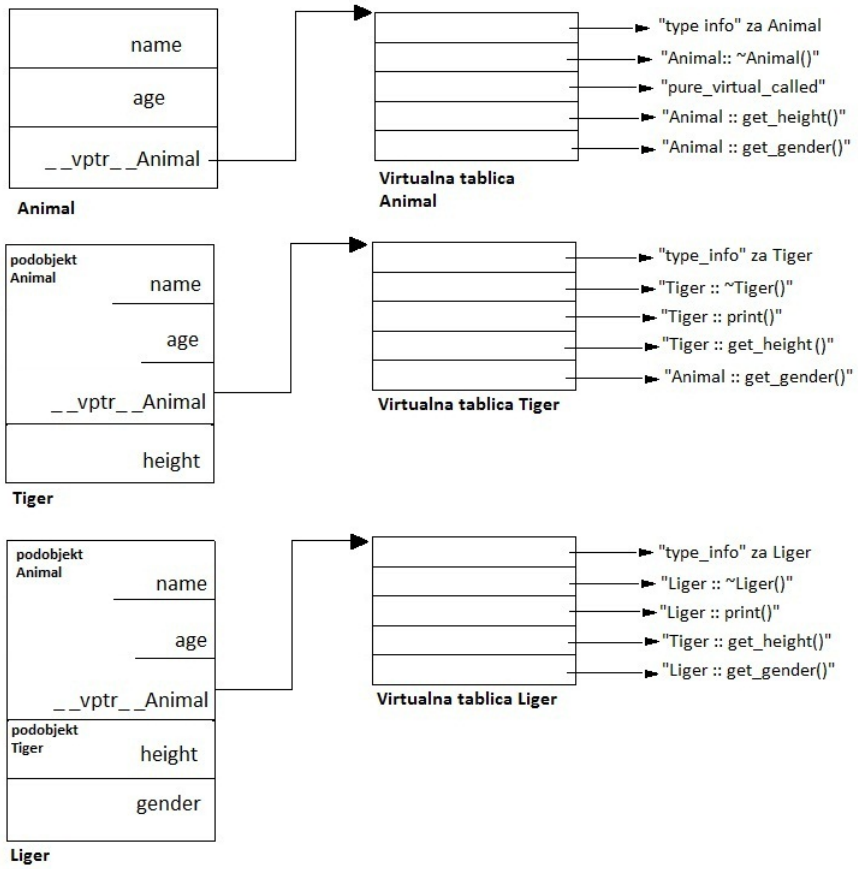
Svakoj virtualnoj funkciji je dodijeljen fiksan indeks unutar virtualne tablice. Virtualna funkcija ostaje na tom indeksu kroz cijelu hijerarhiju nasljeđivanja. U hijerarhiji klase `Animal`

```
class Animal{
protected:
  char* name;
  int age;
public:
  Animal(char* _name = " ", int _age = 0 ):
  virtual ~Animal();

  virtual void print() = 0;

  int get_age(){ return age; }
  char* get_name(){ return name; }

  //rezerviramo mjesto za height i gender - ne radi ništa...

  virtual int get_height(){return 0; }
  virtual char* get_gender() {return 0;}

  // još članova
};
```

virtualnom destruktoru će najvjerojatnije biti pridružen priključak 1, funkciji `print()` 2. (U ovom slučaju, nema definicije funkcije `print()`, pa se u priključak smješta adresa bibliotečne funkcije `pure_virtual_called()`. Ako se kojim slučajem ta funkcija pozove, općenito to bi okončalo program.) Funkciji `get_height()` je pridružen priključak 3, `get_gender()` 4. Funkcije `get_name()` i `get_age()` nisu deklarirane kao virtualne pa nisu u virtualnoj tablici. Slika 4.1 prikazuje izgled i virtualnu tablicu klase `Animal`.

Što se dogodi ako izvedemo klasu iz klase `Animal`:

```
class Tiger: public Animal{
protected:
  int height;
public:
  Tiger(char* _name= " ", int _age=0, int _height=0 )
     : Animal( _name, _age ), set_height( _height);
  // funkcije koje rukuju s članovima name i age ostaju iste
  ~Tiger();
  //prerađena virtualna funkcija
  void print();
```



Slika 4.1: Virtualne tablice : jednostruko nasljeđivanje

```
  int get_height(){ return height; }
  //još članova
};
```

Tri su mogućnosti:

1. Može nasljediti instancu virtualne funkcije deklarirane unutar nadklase. Adresa instance se kopira u pridruženi priključak virtualne tablice podklase.

2. Može preraditi instancu svojom instancom. U ovom slučaju, adresa njegove instance se smješta u pridruženi priključak.

3. Može uvesti novu virtualnu funkciju koja nije prisutna u nadklasi. U ovom slučaju, virtualna tablica naraste za priključak i adresa funkcije se smješta u njega.

U virtualnoj tablici klase `Tiger` destruktor je smješten u priključku 1, a njegova instanca funkcije `print()` u priključku 2 (zamjenjujući tako virtualnu instancu). Njegova instanca funkcije `get_height()` je u priključu 3, instanca funkcije `get_gender()` koju nasljeđuje od klase `Animal` je u priključku 4. Slika 4.1 također prikazuje izgled klase `Tiger` i njezinu virtualnu tablicu.
Slično, izvod klase Liger iz klase Tiger izgleda na sljedeći način:

```
class Liger: public Tiger{
protected:
  int gender;
public:
  Liger(char* _name= " ", int _age=0, int _height=0, char* _gender= " " )
     :Tiger( _name, _age, _height ), set_gender( _gender);

  ~Liger();
  //prepisane virtualne funkcije
  void print();
  char* get_gender(){ return gender; }
  //još članova
};
```

Sa slike 4.1 vidimo da se destruktor smješta u 1. priključak, njegova instanca funkcije `print()` u 2., nasljeđena instanca funkcije `get_height()` u 3., a njegova instanca funkcije `get_gender()` u 4.-ti.

Ako imamo izraz `ptr->print();` ovako znamo dovoljno za vrijeme prevođenja da bi omogućili poziv virtualne funkcije!

- Općenito, ne znamo točan tip objekta kojeg `ptr` adresira pri svakom pozivu funkcije `print()`. Znamo, međutim, da preko `ptr`-a možemo pristupiti virtualnoj tablici vezanoj s objektom dane klase.
- Iako ne znamo koju instancu `print()`-a pozvati, znamo da je adresa svake instance sadržana u priključku br 2.

Ta informacija dozvoljava prevodiocuu da transformira poziv u:

```
(*ptr->vptr [ 2 ])( ptr )
```

U ovoj transformaciju, *vptr* predstavlja unutarnje generiran pokazivač na virtualnu tablicu koji je umetnut u svaki objekt klase, a 2 predstavlja priključak pridružen `print()`-u unutar virtualne tablice hijerarhije nasljeđivanja klase `Animal`. Jedina stvar koju ne znamo

do izvođenja programa je adresa koje instance `print()`-a je sadržana u priključku 2. Unutar jednostruke hijerarhije nasljeđivanja, mehanizam virtualnih funkcija se dobro ponaša. Efikasan je i jednostavno se modelira. Podrška za virtualne funkcije pod višestrukim i virtualnim nasljeđivanjem je ponešto kompliciranija.

Virtualne funkcije pod višestrukim nasljeđivanjem

Kompleksnost podrške za virtualne funkcije pod višestrukim nasljeđivanjem vezana je uz drugu i sve sljedeće nadklase te potrebe da se `this` pokazivač podesi za vrijeme izvođenja programa.

```
class Lion {
public:
  Lion();
  virtual ~Lion();
  virtual int get_weight();
  virtual Lion *clone() const;
  //...
protected:
  //Lion podaci
};

class Tiger {
public:
  Tiger();
  virtual ~Tiger();
  virtual int get_height();
  virtual Tiger *clone() const;
  //...
protected:
  //Tiger podaci
};

class Liger : public Lion, public Tiger {
public:
  Liger()
  virtual ~Liger();
  virtual Liger *clone() const;
  //...
protected:
```

```
  //Liger podaci
};
```

Sva kompleksnost podrške virtualnim funkcijama unutar klase `Liger` leži na podobjektu `Tiger`. Tri su primarna slučaja koja zahtjevaju podršku:

- virtualni destruktor
- nasljeđena `Tiger::get_height()` instanca
- skup instanci funkcije `clone()`

Prvo, pridružimo pokazivaču tipa `Tiger` adresu objekta `Liger`

```
Tiger *ptr= new Liger;
```

Adresa `Liger` objekta mora se podesiti tako da pokazuje svoj `Tiger` podobjekt. Kod kojim se to postiže generira se za vrijeme prevođenja:

```
//transformacija potrebna za podršku drugoj baznoj klasi
Liger *temp= new Liger;
Tiger *ptr=temp ? temp + sizeof( Lion ) : 0
```

Bez ove promjene svaka nepolimorfna upotreba pokazivača bi bila neuspjela.
Ako programer želi pobrisati objekt koji adresira ptr

```
delete ptr;
```

Prije same dealokacije memorije pokazivač se mora ponovno podesiti tako da opet adresira početak objekta klase `Liger` (pod pretpostavkom da `ptr` još uvijek pokazuje na `Liger` objekt). Ovaj pomak u memoriji se ne može razriješiti za vrijeme prevođenja jer se općenito stvarni objekt na kojeg `ptr` pokazuje može odrediti samo za vrijeme izvođenja. Općenito pravilo je da se **this** pokazivač namjesti za vrijeme izvršavanja. To jest, veličina potrebnog pomaka i kod koji ga dodaje **this** pokazivaču prevodioc mora negdje ugurati. Pitanje je gdje? Najefikasnije rješenje je upotreba *thunk*-a. *Thunk* je mali asemblerski dio koda koji namješta **this** pokazivač na odgovarajući pomak prije nego što se pozove virtualna funkcija (u ovom slučaju destruktor) i izvrši skok.
Implementacija *thunk*-a dozvoljava da priključak virtualne tablice ostane običan pokazivač, i tako odstranjuje sav nepotreban prostor unutar virtualne tablice koja služi kao potpora višestrukom nasljeđivanju. Adresa koju sadrži svaki priključak ili direktno adresira virtualnu funkciju ili adresira pridruženi *thunk*, ako je potrebno namještanje **this** pokazivača. Drugi višak pri namještanju **this** pokazivača su opetovana pozivanja iste funkcije u virtualnoj tablici ovisno o tome je li pozvana od podklase (ili najlijevije nadklase) ili od druge (ili svake sljedeće) nadklase. Na primjer:

```
Lion *ptr1= new Liger;
Tiger *ptr2= new Liger;

delete ptr1;
delete ptr2;
```

iako oba `delete` poziva rezultiraju pozivom istog `Liger` destruktora, zahtjevaju dva jedinstvena unosa u virtualnu tablicu.

1. `ptr1` ne zahtjeva namještanje **this** pokazivača (S obzirom da je najlijevija nadklasa pokazuje na početak objekta klase `Liger`) Njegov priključak u virtualnoj tablici treba stvarnu adresu destruktora.
2. `ptr2` zahtjeva namještanje **this** pokazivača. Njegov priključak u virtualnoj tablici treba adresu pridruženog *thunk*-a.

Pod višestrukim nasljeđivanjem, izvedena klasa sadrži `n-1` dodatnih virtualnih tablica, gdje je `n` broj njenih neposrednih nadklasa (to jest jednostruko nasljeđivanje uvodi 0 dodatnih tablica). Za klasu `Liger` su generirane dvije tablice:

1. primarna instanca koju dijeli s klasom `Lion`, lijevom nadklasom
2. sekundarna instanca povezana s klasom `Tiger`, drugom nadklasom

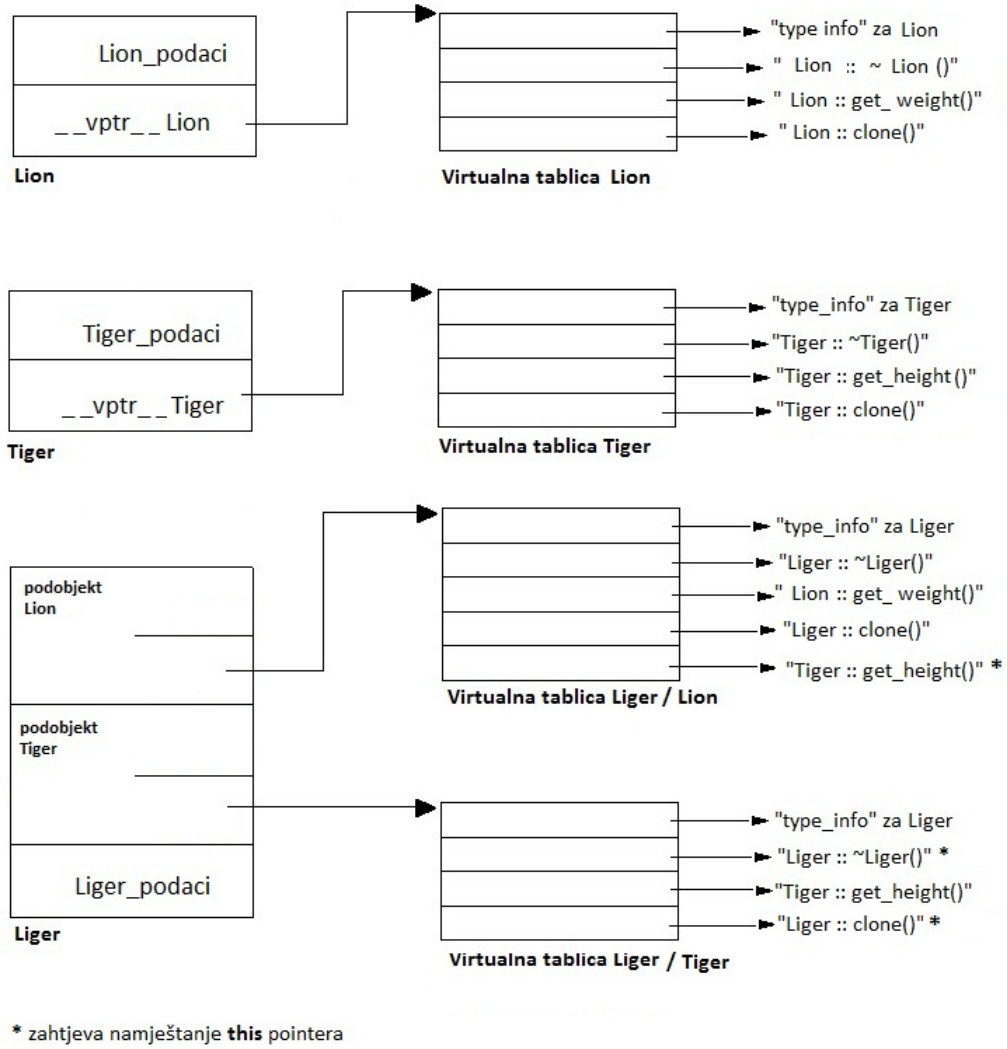
Klasa `Liger` sadrži *vptr* za svaku pridruženu tablicu.

vptr-i su inicijalizirani unutar konstruktora kodom koji generira prevodioc. Ranije u poglavlju je rečeno da postoje tri slučaja u kojima prisutnost druge ili sljedeće nadklase utječe na potporu virtualnim funkcijama. U prvom slučaju, virtualna funkcija podklase je pozvana preko pokazivača druge nadklase. Na primjer:

```
Tiger *ptr = new Liger;

//poziva Liger:: ~Liger
// ptr mora biti namješten minus sizeof (Lion)
delete ptr;
```

Sa slike 4.2 se može vidjeti da u trenutku poziva, `ptr` adresira podobjekt `Tiger` unutar objekta klase `Liger`. Da bi se ovo izvelo ispravno, `ptr` se mora namjestiti da adresira početak objekat klase `Liger`.

Drugi slučaj je varijanta prvog koja uključuje poziv virtualne funkcije koja je nasljeđena od klase `Tiger` preko pointera nasljeđene klase. U ovom slučaju, pointer podklase se mora podesiti tako da adresira podobjekt druge nadklase. Na primjer:



Slika 4.2: Virtualne tablice : višestruko nasljeđivanje

```
Liger *ptr3= new Liger;

//poziva Tiger::get_height();
//ptr3 se mora namjestiti plus sizeof (Lion)
ptr3->get_height();
```

Treći slučaj se odnosi na mogućnost jezika da povratni tip virtualne funkcije varira ovisno o nadklasama i javno nasljeđenim podklasama u hijerarhiji. Taj slučaj nam ilustrira instanca `Liger::clone()`. `Liger` instanca funkcije `clone()` vraća pokazivač na klasu `Liger` ali još uvijek prerađuje instance svojih nadklasa. Problem pomaka **this** pokazivača se javlja pri pozivanju funkcije `clone()` preko pokazivača druge nadklase.

```
Tiger *pb= new Liger;

//poziva Liger *Liger::clone()
//povratna vrijednost mora biti namještena
//tako da adresira podobjekt Tiger

Tiger *pb= pb->clone();
```

Pozvana je `Liger` instanca funkcije `clone()` i koristi *thunk* da namjesti `pb` tako da adresira objekt Liger. Sad `clone()` vraća pointer na objekt klase `Liger` i adresa se mora namjestiti tako pokazuje na podobjekt klase `Tiger` prije nego što ga se pridruži `pb`-u.

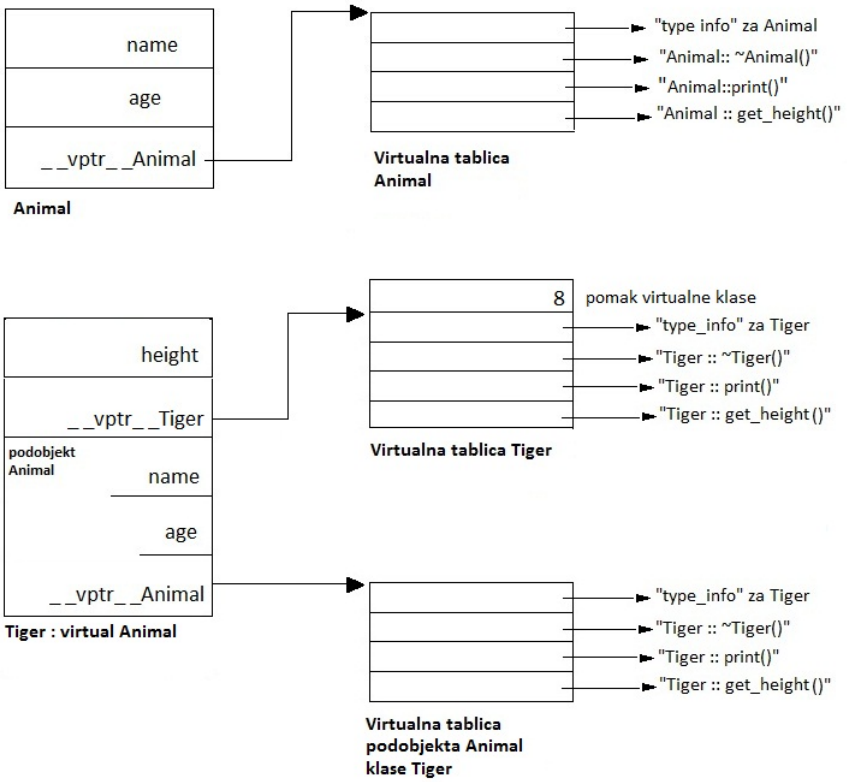
Virtualne funkcije pod virtualnim nasljeđivanjem

Promotrimo sljedeći izvod klase `Tiger` iz klase `Animal`:

```
class Animal{
public:
  Animal(char* _name, int _age);
  virtual ~Animal();

  virtual void print();
  virtual int get_height();
  //...
protected:
  char* name;
  int age;
};
class Tiger : public virtual Animal{
public:
  Tiger(char* _name, int _age, int _height);
  ~Tiger();

  int get_height();
protected:
```

```
  int height;
};
```

Iako klasa `Tiger` ima samo jednu lijevu nadklasu - Animal - početak klase `Animal` i `Tiger` se više ne poklapaju (kao što je to slučaj s nevirtualnim jednostrukim nasljeđivanjem). Slika 4.3 nam ilustrira problem. S obzirom da se `Animal` i `Tiger` više ne poklapaju kon-



Slika 4.3: Virtualne tablice : virtualno nasljeđivanje

verzija među ta dva tipa zahtjeva prilagođavanje **this** pokazivača. Napori da se uklone *thunk*-ovi pod virtualnim nasljeđivanjem su puno teži.

Bibliografija

[1] Lippman, Stanley B., Inside the C++ Object Model, Addison-Wesley, 1996.

[2] Lippman, Stanley B., C++ Primer Fourth Edition, Addison-Wesley, 2006.

[3] http://www.go4expert.com/forums/showthread.php?t=8403

[4] http://en.wikipedia.org/wiki/Virtual_inheritance

[5] http://en.wikipedia.org/wiki/Diamond_problem

[6] http://en.wikipedia.org/wiki/Data_structure_alignment

[7] http://www.devx.com/tips/Tip/14876

[8] http://www.phpcompiler.org/articles/virtualinheritance.html