SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Višestruko nasljeđivanje u statički tipiziranim jezicima

Ivan Martinović
Voditelj: *prof. dr. sc. Siniša Šegvić*

Zagreb, lipanj 2021.

SADRŽAJ

1. Uvod i motivacija

Prvo pitanje koje se nameće kada govorimo o višestrukom nasljeđivanju jest je li ono uopće potrebno? Odgovor bi možda mogao biti da nam višestruko nasljeđivanje nije potrebno, odnosno da možemo zamisliti zaobilazni put kojim bismo riješili isti problem. Ali u tom smislu, možemo reći da nam ni obično nasljeđivanje nije potrebno jer ga možemo ostvariti zaobilaznim putem (npr. kompozicijom), a onda ni da nam razredi ne trebaju jer ih možemo ostvariti strukturama, virtualne tablice pokazivačima na funkcije itd. Međutim, većina programskih jezika danas nudi nasljeđivanje, a dosta njih i višestruko nasljeđivanje u nekom obliku (npr. jedan razred u programskom jeziku *Java* može naslijediti veći broj sučelja, u *C++*-u imamo mogućnost višestrukog nasljeđivanja "običnih" razreda koji imaju podatkovne članove i sl.). Središnja tema ovog seminarskog rada bit će upravo način ostvarivanja višestrukog nasljeđivanja u programskom jeziku *C++*.
Ilustrativni i motivacijski primjer višestrukog nasljeđivanja koji pokazuje da se višestruko nasljeđivanje koristi i u standardnim bibliotekama *C++*-a nalazimo u biblioteci *iostream*. Radi se o razredu *std::basic_iostream* koji nasljeđuje razrede *std::basic_istream* te *std::basic_ostream*, a oni virtualno nasljeđuju razred *std::basic_ios*.

```
basic_ios {...}
basic_ostream : public virtual basic_ios {...}
basic_istream : public virtual basic_ios {...}
basic_iostream : public basic_ostream, public basic_istream
   {...}
```

Važno je primijetiti ključnu riječ *virtual* kod razreda *basic_ostream* i *basic_istream*. Pomoću ključne riječi *virtual* koja se navodi uz razred koji nasljeđujemo u programskom jeziku *C++* ostvarujemo virtualno nasljeđivanje, a zadaća ovog seminarskog rada je objasniti implementacijske detalje koji se kriju iza takvog oblika nasljeđivanja.

2. Nasljeđivanje i višestruko nasljeđivanje u programskom jeziku C++

2.1. Jednostavno nasljeđivanje

Pokažimo za početak kako je izvedeno jednostavno nasljeđivanje jednog razreda u programskom jeziku *C++*. Analizirajmo izvorni kod 2.1.

```
#include <iostream>
using namespace std;

class Parent {
  public:
  virtual void m() {
    cout << "Parent method" << endl;
  }
  int parent_member;
};

class Child : public Parent {
  public:
  void m() {
    cout << "Child method" << endl;
  }
  int child_member;
};

int main() {
```

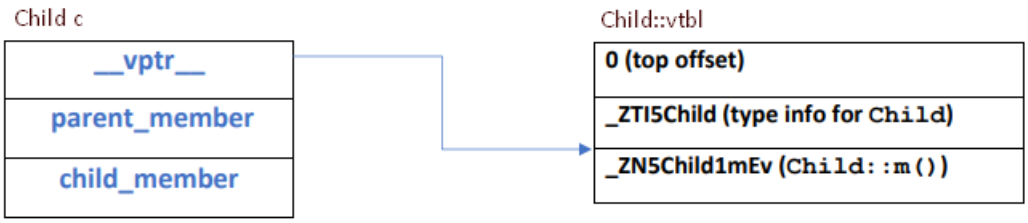
```
    Child c;
    c.m();
    Parent* p = &c;
    p->m();
}
```

Izvorni kod 2.1: Primjer jednostavnog nasljeđivanja s virtualnim metodama

Definirana su dva razreda, osnovni razred *Parent* koji ima virtualnu metodu *m()* te izvedeni razred *Child*, koji nadjačava implementaciju metode *m()* roditeljskog razreda Parent. Nakon prevođenja i pokretanja navedenog izvornog koda dobivamo sljedeći ispis:

```
Child method
Child method
```

Kako bismo razumjeli ispis pogledajmo stanje memorije stvorenog primjerka razreda *Child* koje je prikazano na slici 2.1.



Slika 2.1: Stanje memorije prilikom jednostavnog nasljeđivanja

Primjerak razreda *Child* sadrži pokazivač na virtualnu tablicu funkcija, koji pokazuje na adresu metode *Child::c()*. Prilikom konstrukcije objekta *c* pozove se konstruktor razreda *Child* koji najprije pozove konstruktor razreda *Parent*. Razred *Parent* u svom konstruktoru postavi vrijednost pokazivača na tablicu virtualnih funkcija na adresu memorijske lokacije u koju je upisana adresa metode *Parent::m()*, a nakon povratka u konstruktor razreda *Child*, pokazivač na tablicu virtualnih funkcija postavi se na adresu memorijske lokacije u koju je upisana adresa metode *Child::m()*. Što bi se dogodilo ako nekoj metodi koja kao parametar prima pokazivač na objekt tipa *Parent* te poziva njegovu metodu *m()* kao parametar pošaljemo pokazivač *c*? Tada će se pozvati metoda *Child::m()* zato što pokazivač na tablicu virtualnih funkcija pokazuje na adresu na kojoj se nalazi adresa metode *Child::m()*.

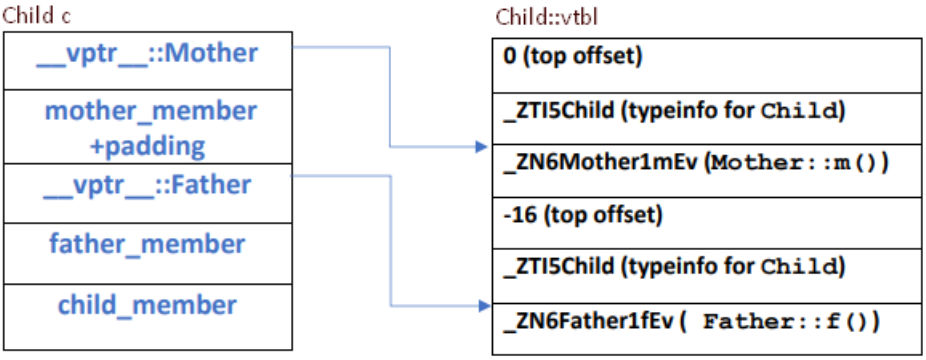
2.2. Višestruko nasljeđivanje

Nakon što smo prikazali jednostavan primjer nasljeđivanja te izvedbu virtualnih tablica, pokušajmo nadograditi naš program do višestrukog nasljeđivanja. Višestruko nasljeđivanje znači da razred izravno ima više roditeljskih razreda. Promotrimo izvorni kod 2.2.

```
class Mother {
  public:
  virtual void m() {
    cout << "Mother method" << endl;
  }
  int mother_member;
};

class Father {
  public:
  virtual void f() {
    cout << "Father method" << endl;
  }
  int father_member;
};

class Child : public Mother, public Father {
  public:
  int child_member;
};
```

Izvorni kod 2.2: Početni primjer višestrukog nasljeđivanja

Razred *Child* izravno nasljeđuje dva razreda, *Mother* i *Father*. Pogledajmo najprije kakve posljedice to ima na memorijsku strukturu jednog primjerka razreda *Child* (slika 2.2). Vidimo da sada primjerak razreda *Child* ima dva pokazivača na tablice virtualnih funkcija. Budući da je razred *Child* u relaciji **je vrsta** s razredima *Mother* i *Father*, bilo koja funkcija koja kao parametar prima ili primjerak razreda *Mother* ili primjerak razreda *Father* mora se uspješno izvesti ako joj se kao parametar proslijedi primjerak



Slika 2.2: Stanje memorije u početnom primjeru višestrukog nasljeđivanja

razreda *Child*. Zbog toga što se primjerak razreda *Child* mora moći implicitno koristiti na mjestu razreda *Mother* i *Father* moramo imati dva pokazivača na tablicu virtualnih funkcija. Iz prikazane memorijske strukture jednog primjerka razreda *Child* možemo primijetiti kako se redom zapisuju pokazivač i atributi roditeljskih razreda redom kojim su navedeni prilikom definicije razreda, što nije i jedini mogući način izvedbe. Ostali dijelovi prikazane memorijske strukture su očekivani i već viđeni. Ako malo bolje promotrimo prikazanu memorijsku strukturu možemo uočiti dvije potencijalne primjedbe. Recimo nešto o prvoj, možda malo manje očiglednoj primjedbi. Neka sada razred *Child* nadjačava metode iz razreda *Father* i *Mother* iz prošlog primjera.

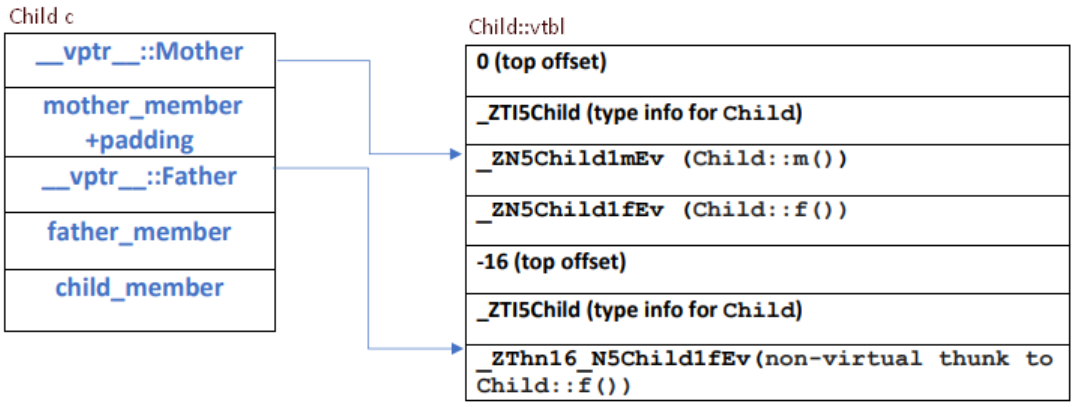
1. Što bi onda referencirali pokazivači na tablicu virtualnih funkcija i gdje bi bilo mjesto za adrese novostvorenih funkcija u tablici virtualnih funkcija?

2. U slučaju da razred *Child* nadjačava metodu razreda *Father*, na koji način bismo metodu koja kao parametar prima primjerak razreda *Father* mogli pozvati primjerkom razreda *Child*, a da se program i dalje očekivano izvršava?

Na ova pitanja ćemo odgovoriti ako razred *Child* iz prošlog primjera nadogradimo i pogledamo memorijsku strukturu tako nadograđenog primjerka razreda *Child*.

```
  //nadogradnja izvornog koda 2.2
class Child : public Mother, public Father {
  public:
  void f() {
    cout << "Child f method" << endl;
  }
  void m() {
    cout << "Child m method" << endl;
  }
  int child_member;
};
```

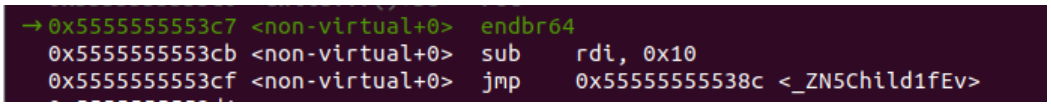
Izvorni kod 2.3: Nadogradnja izvornog koda 2.2



Slika 2.3: Stanje memorije za primjer izvornog koda 2.3

Odgovor na oba pitanja možemo pronaći u memorijskoj strukturi objekta (slika 2.3). Važno je uočiti da nemamo treći pokazivač na tablicu virtualnih funkcija koji bi onda pokazivao na memorijske lokacije na kojima se nalaze adrese nadjačanih metoda. Umjesto toga, prvi pokazivač, na slici označen kao *__vptr__::Mother* pokazuje na memorijske lokacije na kojima se nalaze adrese odgovarajućih nadjačanih metoda. Ako u ovom slučaju metodi koja kao parametar prima primjerak razreda *Mother* proslijedimo primjerak razreda *Child*, pozvat će se nadjačana metoda *Child::m()* a pokazivač *this* će pokazivati na početak memorijskog prostora zauzetog za objekt *c*. Razmotrimo sada slučaj u kojem metodi koja kao parametar prima primjerak razreda *Father* pošaljemo naš objekt *c*. Očekivano ponašanje bilo bi da prevoditelj generira strojni kod koji bi

prilikom poziva metode na vrijednost pokazivača *this* dodao vrijednost koja piše u polju *top offset* (to je mjesto na kojemu se nalazi *__vptr__::Father*). U takvom scenariju suprotno od očekivanog ne bi se pozvala nadjačana metoda *Child::f()* nego metoda osnovnog razreda *Father::f()*. Kako to ne bi bio slučaj, pokazivač *__vptr__::Father* više ne pokazuje na memorijsku lokaciju na kojoj se nalazi metoda *Father::f()* nego pokazuje na memorijsku lokaciju na kojoj se nalazi adresa od implicitno stvorene metode *non-virtual thunk to Child::f()*. Ta metoda smanjuje pokazivač *this* za odgovarajuću vrijednost (u ovom slučaju 16 okteta), a tako postižemo da pokazivač *this* prilikom poziva prije navedene metode pokazuje na početak objekta *c* te se poziva odgovarajuća nadjačana metoda *Child::f()*. To možemo ilustrirati strojnim kodom metode *non-virtual thunk to Child::f()* (slika 2.4).

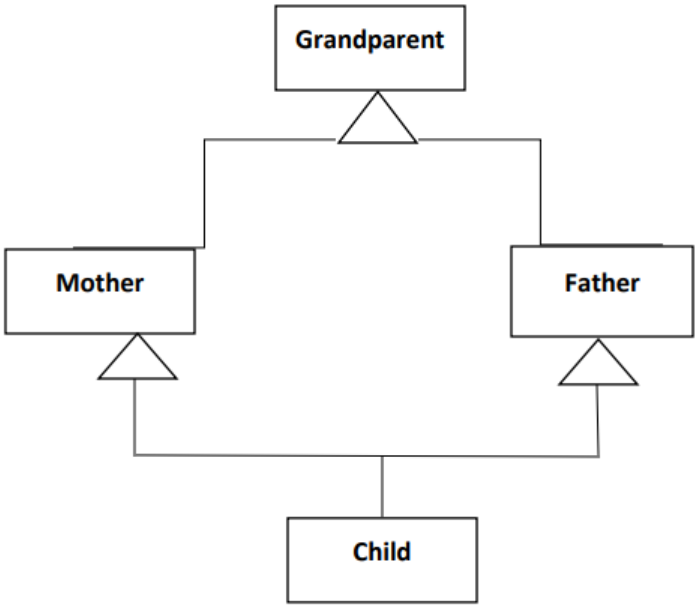


Slika 2.4: Odsječak strojnog koda metode *non-virtual thunk to Child::f()*

```
//ilustracija prethodnog primjera
//razredi Child, Father i Mother nisu navedeni
void fmethod(Father *f) {
  f->f();
}

void mmethod(Mother *m) {
  m->m();
}

int main() {
  Child c;
  fmethod(&c);
  mmethod(&c);
}
//ispis
Child f method
Child m method
```

Pogledajmo sada i drugu komplikaciju višestrukog nasljeđivanja. Što bi se dogodilo

da su razredi *Father* i *Mother* nasljeđivali isti roditeljski razred, nazovimo ga *Grandparent*? Ovaj problem već je dobro poznat, a u literaturi ga često nazivaju *Problem dijamanta* (eng. *Deadly Diamond of Death, Diamond problem*). Najčešće se prikazuje dijagramom razreda kao na slici 2.5.



Slika 2.5: Strukturni dijagram razreda tzv. problema dijamanta

Kako bismo objasnili problematiku ovog primjera nadogradimo naš prije navedeni izvorni kod prema primjeru izvornog koda 2.4.

```
class Grandparent {
  public:
  Grandparent() {
    this->grand_member = 1;
  }
  int grand_member;
  virtual void g() {}
};

class Mother: public Grandparent {
  public:
  int mother_member;
  virtual void m() { cout << "m" << endl;}
};
```

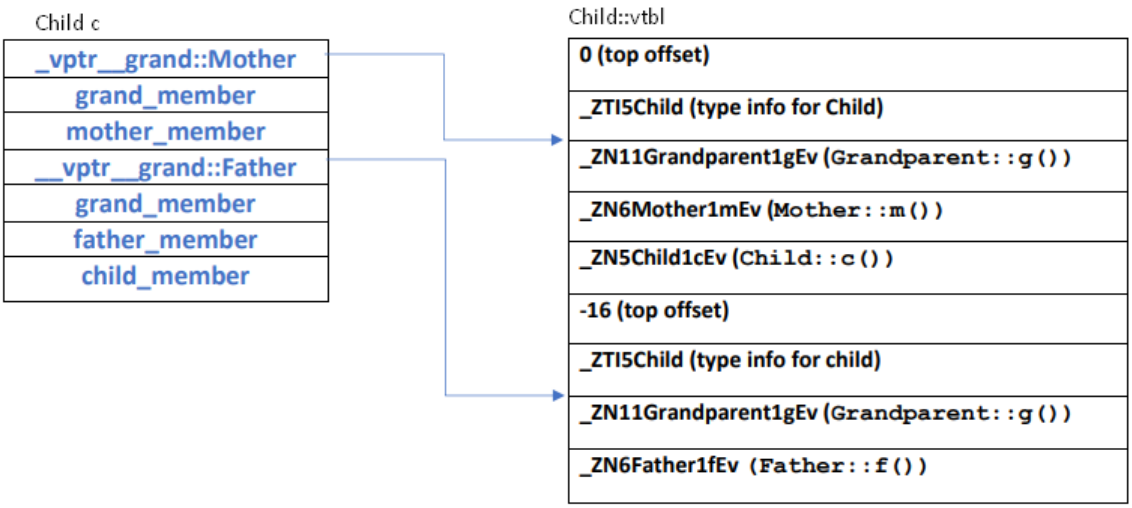
```
class Father: public Grandparent {
  public:
  int father_member;
  virtual void f() {cout << "f" << endl;}
};

class Child : public Mother, public Father {
  public:
  int child_member;
  virtual void c() {cout << "c" << endl;}
};
```

Izvorni kod 2.4: Nadogradnja izvornog koda 2.3

Dodamo li i glavni program u kojemu stvaramo primjerak razreda *Child* te pozivamo metodu *Child::g()* prevoditelj će javiti pogrešku (slika 2.7). Do pogreške dolazi zato što primjerak razreda *Child* u svojoj virtualnoj tablici vidi dvije memorijske lokacije na kojima se nalazi adresa metode *g()*. Zbog toga prevoditelj ne zna koju adresu treba pozvati. Pogrešku možemo spriječiti navedemo li eksplicitno razred čiju metodu *g()* želimo pozvati (Izvorni kod 2.5). U ovom konkretnom primjeru radi se o istim metodama, no možemo zamisliti da jedan od razreda *Father* ili *Mother* pruža vlastitu implementaciju metode *g()*. Tada bi se u tablici virtualnih funkcija nalazile adrese dvije potpuno različite metode na različitim adresama. Isto bi se dogodilo ako iz razreda *Child* pokušamo pristupiti podatkovnom atributu *grand_member* jer zapravo postoje dvije memorijske adrese na kojima je pohranjena vrijednost tog atributa. Alat pomoću kojeg *C++* rješava navedene komplikacije jest virtualno nasljeđivanje.

```
int main() {
   Child c;
   //c.g() -- pogreska!
   c.Mother::g();
}
```

Izvorni kod 2.5: Uspješno pozivanje metode bazne klase kod višestrukog nasljeđivanja



Slika 2.6: Memorijska struktura primjerka razreda Child u problemu dijamanta

```
v4.cpp: In function ‘int main()’:
v4.cpp:33:3: error: request for member ‘g’ is ambiguous
   33 | c.g();
      |   ^
v4.cpp:10:15: note: candidates are: ‘virtual void Grandparent::g()’
   10 |  virtual void g() {}
      |               ^
v4.cpp:10:15: note:                 ‘virtual void Grandparent::g()’
```

Slika 2.7: Pogreška prilikom prevođenja – višestruko nasljeđivanje

2.3. Virtualno nasljeđivanje

Kako bismo ilustrirali virtualno nasljeđivanje krenimo najprije od jednostavnijeg primjera navedenog u Izvorni kod 2.6.

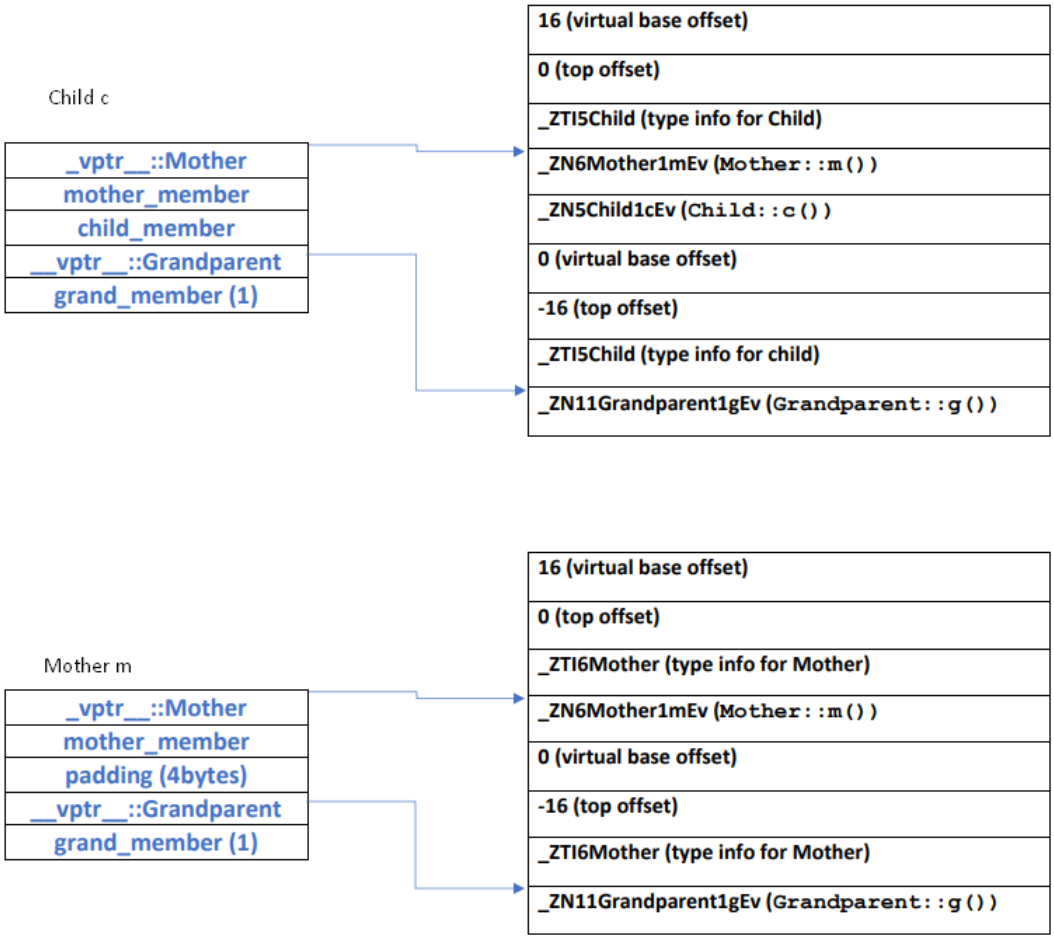
```
class Grandparent {
  public:
  Grandparent() {
    this->grand_member = 1;
  }
  int grand_member;
  virtual void g() {}
};

class Mother: public virtual Grandparent {
  public:
  int mother_member;
```

```
  virtual void m() { cout << "m" << endl;}
};

class Child : public Mother {
  public:
  int child_member;
  virtual void c() {cout << "c" << endl;}
};
```

Izvorni kod 2.6: Početni primjer virtualnog nasljeđivanja

Primjećujemo ključnu riječ *virtual* pri definiciji nasljeđivanja kod razreda *Mother*. Virtualno nasljeđivanju najjednostavnije možemo objasniti tako da ono osnovnom razredu govori "stvori samo jedan pod-objekt ovog tipa u bilo kojem izvedenom objektu". U našem slučaju, stvori samo jedan pod-objekt razreda tipa *Grandparent* u bilo kojem izvedenom razredu. Ako imamo osnovni razred kojega izvedeni razred virtualno nasljeđuje tada se memorijska struktura objekta tipa izvedenog razreda dijeli na dva područja: nepomično područje i dijeljeno područje. Podaci u nepomičnom području konstantno ostaju na istom pomaku od početka alociranog memorijskog prostora za objekt neovisno o budućim nasljeđivanjima. Zbog toga se podacima u nepomičnom području može izravno pristupati. Dijeljeno područje predstavlja pod-objekt virtualnog razreda (ili više njih u slučaju da imamo više osnovnih virtualnih razreda), a njihov pomak od početka memorijskog prostora zauzetog za objekt tipa izvedenog razreda se mijenja ovisno o budućim nasljeđivanjima. Zbog toga podacima u dijeljenom području trebamo pristupati neizravno. Zbog svega navedenog vidimo da ćemo morati imati dodatne mehanizme regulacije pristupanja memoriji objektima koji kao pod-objekt imaju virtualni osnovni razred, a za to moramo platiti određenu (memorijsku i vremensku) cijenu. Virtualno nasljeđivanje u *C++*-u ostvarujemo pomoću informacije o pomaku od virtualnog osnovnog razreda (*virtual base class offsets*) i taj način ostvarenja virtualnog nasljeđivanja bit će obrađen u nastavku. Važno je spomenuti da ovo nije i jedini mogući način ostvarenja virtualnog nasljeđivanja.[1] Npr. može se ostvariti pomoću pokazivača na osnovni virtualni razred. Pogledajmo najprije memorijsku strukturu jednog objekta tipa *Child* te objekta tipa *Mother* iz prethodnog primjera te s njima povezane virtualne tablice.

Ono što najprije primjećujemo su polja **virtual base offset** (u nastavku "pomak od osnovnog virtualnog razreda") u tablici virtualnih funkcija. To polje govori na



Slika 2.8: Memorijska struktura objekata tipa Child i Mother u osnovnom primjeru virtualnog nasljeđivanja

kojoj udaljenosti se od pokazivača na alocirani memorijski prostor (*this*) objekta nalaze podaci koji su vezani uz virtualni osnovni razred. U našem slučaju vidimo da taj pomak iznosi 16 okteta što odgovara prikazanoj memorijskoj strukturi objekata *m* i *c*. Prilikom konstrukcije primjerka razreda *Child* najprije se poziva konstruktor osnovnog virtualnog razreda *Grandparent* te se inicijaliziraju memorijske adrese koje su vezane uz taj objekt. Konkretno, inicijalizira se pokazivač na tablicu virtualnih funkcija *__vptr__::Grandparent* te se inicijalizira vrijednost podatkovnog člana *grand_member*. Nakon toga se poziva konstruktor razreda *Mother*. U slučaju da je razred *Mother* nadjačao metodu *Grandparent::g()* on bi trebao preusmjeriti već inicijalizirani pokazivač *__vptr__::Grandparent* da pokazuje na nadjačanu implementaciju metode *Mother::g()*. Budući da u ovom primjeru razred *Mother* ne nadjačava metodu *Grandparent::g()* potrebe za ovim preusmjeravanjem nema. Ono će se međutim ipak

dogoditi, ali će se pokazivač još jednom usmjeriti na ono na što je već pokazivao, odnosno ponovno će pokazivati na adresu memorijske lokacije na kojoj se nalazi adresa metode *Grandparent::g()*. Na koji način prevoditelj, odnosno konstruktor razreda *Mother* zna adresu pokazivača *__vptr__::Grandparent*? Konstruktor razreda *Mother* čita tu informaciju upravo iz polja virtualne tablice koji govori o pomaku od osnovnog virtualnog razreda. Kako konstruktor razreda *Mother* može biti siguran da je konstruktor razreda *Grandparent* već pozvan i da može pristupati njegovim članovima? To nam jamči ugovor višestrukog nasljeđivanja koji govori da će se najprije pozvati i inicijalizirati članovi osnovnog virtualnog razreda. Nakon što je konstruktor razreda *Mother* inicijalizirao svoje podatkovne članove, nastavlja se izvoditi konstruktor razreda *Child* koji završno inicijalizira svoje podatkovne članove.

Što će se dogoditi ako stvorimo primjerak razreda *Mother*? Vidimo da u ovom slučaju moramo imati dva konstruktora razreda *Mother*, jedan koji se poziva kada stvaramo primjerak razreda *Child* a drugi kada stvaramo primjerak razreda *Mother*. Konstruktor koji se poziva pri stvaranju primjerka razreda *Mother* unutar sebe mora pozivati i konstruktor razreda *Grandparent*, dok u slučaju stvaranja primjerka razreda *Child* konstruktor razreda *Mother* ne poziva konstruktor razreda *Grandparent*. Prilikom stvaranja primjerka razreda *Child* konstruktor razreda *Mother* pomoću virtualnog pomaka od osnovnog razreda te pomoću konstrukcijske tablice *Mother-in-Child* ažurira odnosno namješta adrese virtualnih funkcija *Grandparent::g()* odnosno *Mother::m()*. Prilikom stvaranja primjerka razreda *Mother* konstruktor razreda *Mother* najprije poziva konstruktor razreda *Grandparent*, a nakon toga ažurira pokazivač na tablicu virtualnih funkcija, ali ovoga puta odgovarajuće adrese funkcija čita iz virtualne tablice razreda *Mother*, a ne konstrukcijske virtualne tablice *Mother-in-Child*.

Već vidimo plaćanje memorijske cijene: osim novih zapisa u tablici virtualnih funkcija o pomaku od osnovnog virtualnog razreda moramo definirati i različite konstruktore za isti razred. Vidimo još jedan problem, virtualna tablica primjerka razreda *Mother* potencijalno može biti različita od virtualne tablice razreda *Mother* u okviru razreda *Child*. Razlika nastaje zbog različitih pomaka od osnovnog virtualnog razreda u primjercima razreda *Child* odnosno *Mother*. Zbog toga uz virtualnu tablicu *vtable for Mother* dobivamo još jednu konstrukcijsku virtualnu tablicu *construction vtable for Mother-in-Child* koja se koristi prilikom inicijaliziranja pokazivača na tablicu virtualnih funkcija u konstruktoru razreda *Mother* prilikom stvaranja primjerka razreda *Child*. Jedina stvar koju još nismo spomenuli je *VTT - Virtual table table*, odnosno ta-

blica virtualnih tablica, a ona predstavlja tablicu za prevođenje koja se koristi prilikom konstrukcije objekata koji virtualno nasljeđuju osnovni razred. Konstrukcijske virtualne tablice kao i tablicu virtualnih tablica ćemo objasniti na nadogradnji prethodnog primjera (Izvorni kod 2.7). Neka sada razredi *Mother* i *Child* nadjačavaju implementaciju metode *Grandparent::g()*.

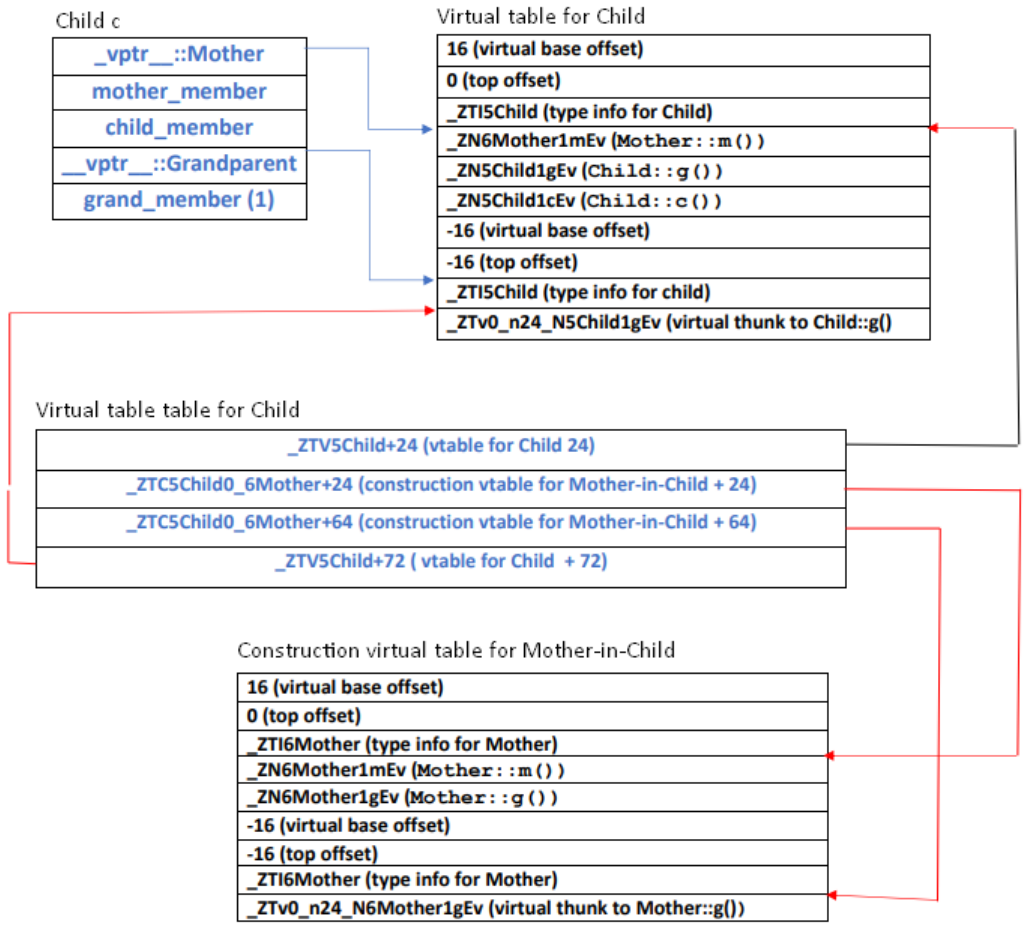
```
//razred Grandparent ostaje isti kao u prethodnom primjeru
//dodane su nadjacane metode g() u razredima Child i Mother
class Mother: public virtual Grandparent {
  public:
  int mother_member = 2;
  virtual void m() { cout << "m" << endl;}
  virtual void g() { cout << "mothers g" << endl;}
};

class Child : public Mother {
  public:
  int child_member = 3;
  virtual void g() {cout << "childs g" << endl;}
  virtual void c() {cout << "c" << endl;}
};
```

Izvorni kod 2.7: Primjer virtualnog nasljeđivanja s nadjačanim metodama virtualnog razreda

Pogledat ćemo memorijsku strukturu svih navedenih tablica koje nastaju prilikom stvaranja jednog primjerka razreda *Child* (slika 2.9) uz navedenu nadogradnju. Pogledajmo strukturu konstrukcijske virtualne tablice *Mother-in-Child* na slici 2.9. U ovom konkretnom slučaju ona je ista kao i virtualna tablica razreda *Mother*. Nadogradimo li razred *Child* tako da mu dodamo još jedan podatkovni član tipa *int*, onda bi se konstrukcijska virtualna tablica *Mother-in-Child* te virtualna tablica razreda *Mother* razlikovale u poljima *virtual base offset*. Konstrukcijska virtualna tablica bi to polje imala postavljeno na vrijednost 24 (novi podatkovni član + padding), dok bi virtualna tablica razreda *Mother* i dalje tu vrijednost imala postavljenu na 16. Zato su nam potrebne obje virtualne tablice. Ono što također primjećujemo je polje *virtual thunk to Mother::g()*, ali to je po namjeni ista stvar kao i *non-virtual thunk* koji je već objašnjen u odjeljku 2.2. Jedina razlika je u tomu što u ovom slučaju razred *Mother* virtualno nasljeđuje *Grandparent*, pa se sada metoda zove *virtual thunk* umjesto *non-virtual thunk*. Dodamo li i razred *ChildSecond* koji nasljeđuje razred *Mother*, tada bi prevodi-

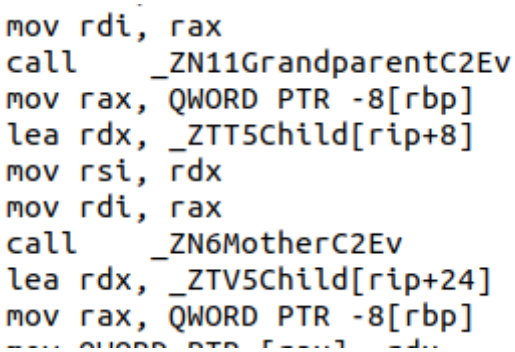
telj stvorio još jednu konstrukcijsku virtualnu tablicu naziva *virtual table for Mother-in-ChildSecond*.

Iz svega navedenog jasno možemo jasno zaključiti svrhu konstrukcijskih virtualnih tablica za izvedene razrede koji virtualno nasljeđuju neki osnovni razred. Budući da ti izvedeni razredi (u našem primjeru razred *Mother*) ne znaju tko ih sve može naslijediti (u našem primjeru razredi *Child* i *ChildSecond*), pa samim time ne znaju na kojim se sve pomacima od pokazivača *this* nalazi osnovni virtualni razred, stvaramo konstrukcijske virtualne tablice koje su karakteristične za svaki izvedeni razred u daljnjem stablu nasljeđivanja (u našem primjeru *Mother-in-Child* i *Mother-in-ChildSecond*).



Slika 2.9: Memorijska struktura primjerka razreda Child, virtualna tablica, VTT te konstrukcijska virtualna tablica Mother-in-Child

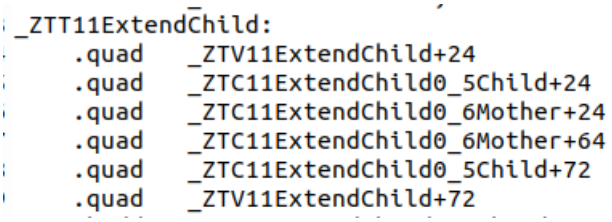
Kako stvari ne bi bile prejednostavne na slici 2.9 vidimo i još jednu razinu indirekcije u vidu tablice virtualnih tablica *Virtual table table for Child*.



Slika 2.10: Korištenje virtualne tablice tablica prilikom poziva konstruktora razreda *Mother*

Pokušajmo objasniti što predstavlja ta nova tablica, tablica virtualnih tablica (*VTT*). Kada pozovemo konstruktor razreda *Child*, najprije se pozove konstruktor razreda *Grandparent*, a onda konstruktor razreda *Mother*, a razlog (najprije se mora stvoriti osnovni virtualni razred) je već objašnjen. Kada konstruktor razreda *Child* pozove konstruktor razreda *Mother* on će osim pokazivača *this* poslati i jedan skriveni parametar koji će predstavljati odgovarajuću adresu virtualne tablice (slika 2.10). Taj skriveni parametar na slici se nalazi u registru **rsi**. Vidimo međutim da se ne učitava adresa konstrukcijske virtualne tablice *Mother-in-Child* nego adresa *Virtual table table for Child + 8*, a na toj memorijskoj lokaciji se (slika 2.9) nalazi upravo adresa prve funkcije u konstrukcijskoj virtualnoj tablici *Mother-in-Child*. Opravdano pitanje koje se nameće je čemu nam služi ta nova razina indirekcije? Zašto nismo odmah mogli učitati adresu *construction virtual table for Mother + 24* u konstruktoru razreda *Child*? Odgovor na pitanje možemo pronaći ako stvorimo još jedan razred *ExtendsChild* koji nasljeđuje razred *Child*. U tom slučaju dobivamo nove konstrukcijske virtualne tablice: *Child-in-ExtendsChild te Mother-in-ExtendsChild*. Ako stvorimo primjerak razreda *ExtendsChild*, on će u svom konstruktoru najprije pozvati *Grandparent::Grandparent()*, a onda *Child::Child()*. Nakon toga će se u konstruktoru razreda *Child* pozvati konstruktor razreda *Mother*. Konstruktor razreda *ExtendsChild* prije nego pozove *Child::Child()* će u registar **rsi** spremiti adresu *Virtual table table for ExtendsChild + 8* gdje se nalazi adresa konstrukcijske virtualne tablice *Child-in-ExtendsChild*. Nakon toga će se u konstruktoru razreda *Child* registar **rsi** povećati za 8 (slika 2.11) i pozvati konstruktor razreda *Mother*.

Tada će u registru **rsi** biti spremljena adresa konstrukcijske virtualne tablice *Mother-in-ExtendsChild*. Takvo ponašanje je očekivano i ispravno. Da smo međutim umjesto još jedne razine indirekcije izravno slali adrese odgovarajućih konstrukcijskih tablica, konstruktor razreda *Child* bi poslao adresu konstrukcijske virtualne tablice *Mother-in-*



Slika 2.11: Tablica virtualnih tablica razreda *ExtendsChild*

Child što dakako nije ispravno, budući da stvaramo objekt *ExtendsChild*. Dobro je još napomenuti da se tablica virtualnih tablica nekog razreda u memoriji uvijek nalazi izravno pored tablice virtualnih funkcija za taj razred.

3. Zaključak

Virtualno nasljeđivanje u programskom jeziku *C++* dolazi s cijenom. Pogledamo li memorijsku stranu ono što odmah uočavamo su dva konstruktora za isti razred (zapravo se nazivaju *base constructor* i *complete constructor*) te nove virtualne tablice u vidu konstrukcijskih virtualnih tablica i tablice virtualnih tablica. Osim toga svaka virtualna tablica razreda koji izravno ili neizravno virtualno nasljeđuje neki osnovni razred mora imati zapis o pomaku od osnovnog virtualnog razreda. Iako kada sve zbrojimo ta cijena nije mala, danas memorija uglavnom ne predstavlja problem. Ipak cilj dobrog programera treba biti postići cilj uz optimalnu cijenu, pa u tom smislu prije korištenja virtualnog nasljeđivanja uvijek treba pogledati postoji li neko jednostavnije (*Keep it simple!*) rješenje. Osim memorijske cijene plaćamo i vremensku cijenu u vidu *thunk* metoda i prilagodbom pokazivača *this*, ali i još jednom razinom indirekcije pri korištenju tablice virtualnih tablica. Postoje brojne rasprave po internetu o tome treba li izbjegavati višestruko odnosno virtualno nasljeđivanje. Rekao bih da je virtualno nasljeđivanje zgodan alat koji nudi programski jezik *C++* i ako taj alat primjenjujemo na za to predviđenim mjestima, po mom mišljenju, virtualno nasljeđivanje je jednostavno i jako elegantno rješenje od kojega ne treba bježati.

Više informacija o učestalosti korištenja virtualnog nasljeđivanja može se pronaći u članku *Devil is Virtual: Reversing Virtual Inheritance in C++ Binaries*.[1] Uzmimo samo za primjer da se, prema tom članku, u izvršnom kodu programa *mysqld*[2] pojavljuje 201 razred, dok se u standardnoj bibiloteci *libstdc++6* pojavljuje 29 razreda koji izravno ili neizravno virtualno nasljeđuju neki osnovni razred.

[1] https://arxiv.org/pdf/2003.05039.pdf

[2] http://dba.fyicenter.com/faq/mysql/What-Is-mysqld.html

4. Literatura

[1] Stanley B. Lippman. *Inside the C++ Object model*. Addison Wasley.

[2] Arthur O'Dwyer. What is the virtual table table?, 2019. URL https://quuxplusone.github.io/blog/2019/09/30/what-is-the-vtt/.

[3] Jack W. Reeves. Multiple inheritance considered useful, 2006. URL https://www.drdobbs.com/cpp/multiple-inheritance-considered-useful/184402074?

[4] Mike Shahar. C++ vtables - part 3 - virtual inheritance, 2016. URL https://shaharmike.com/cpp/vtable-part3/.