Lekcija 11

Polimorfizam, Apstrakcija, Šabloni

Miljan Milošević

POLIMORFIZAM, APSTRAKCIJA, ŠABLONI

	01	02	03	04
Uvod	**Uvod u polimorfizam**	**Virtuelne funkcije**	**Apstrakcija podataka**	**Apstraktne klase**
	➢ *Pokazivači i reference na osnovne klase objekata izvedene klase* ➢ *Osnovi o polimorfizmu* ➢ *Potreba za virtuelnim funkcijama*	➢ *Definisanje virtuelnih funkcija* ➢ *Upotreba virtuelnih funkcija* ➢ *Upotreba virtuelnih funkcija* ➢ *Potreba za virtuelnim destruktorima* ➢ *Definisanje virtuelnog destruktora* ➢ *Čisto virtuelne funkcije; Nadglasavanje virtuelizacije; Mane virtuelnih funkcija*	➢ *Apstrakcija podataka u C++-u* ➢ *Specifikatori pristupa, pogodnosti apstrakcije i strategija projektovanja* ➢ *Primer apstrakcije podataka*	➢ *Uvod u apstraktne klase (C++ interfejsi)* ➢ *Definicija apstraktnih klasa* ➢ *Primer apstraktne klase*

POLIMORFIZAM, APSTRAKCIJA, ŠABLONI

05

Osnovi o hijerarhiji klasa

- *Hijerarhija klasa*

06

Šabloni funkcija

- *Uvod u šablone*
- *Definisanje i upotreba šablona funkcija*
- *Upotreba šablona funkcija - određivanje maksimuma različitih tipova podataka*
- *Operatori, pozivi funkcija i šabloni funkcija*
- *Problemi kod šablona funkcija*
- *Preklapanje operacija kod*

07

Šabloni klasa

- *Definicija šablona klasa*
- *Primer šablona klase*
- *Vrednosni parametri šablona klasa*
- *Specijalizovan šablon klase*
- *Implementacija specijalizovanog šablona klase*

08

Vežbe

- *Virtuelne funkcije - Primer1*
- *Virtuelne funkcije - Primer2*
- *Virtuelne funkcije - Primer3*
- *Primer. Čisto virtuelne funkcije*
- *Primer. Šabloni funkcija*
- *Primer. Šabloni klase*

09

Zadaci za samostalan rad

- *Zadaci za samostalno vežbanje*

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upoznaju sa sledećim pojmovima objektno orijentisanog principa programskog jezika C++:

- Polimorfizam
- Virtuelne funkcije
- Apstrakcija podataka i apstraktne klase
- Hijerarhija klasa
- Šabloni funkcija i klasa

Princip polimorfizma predstavlja jedan od osnovnih principa objektno orijentisanog programiranja. On se ogleda u vrlo specifičnom principu dvojnosti koji je zastupljen u objektno orijentisanoj formulaciji problema. Generalno gledano, to je kada značenje neke instrukcije zavisi od konteksta, i rezultat neke instrukcije može da se menja u zavisnosti od konteksta.

Polimorfizam se u C++-u ostvaruje korišćenjem virtuelnih funkcija, koje se definišu upotrebom ključne reči virtual. Ukoliko se želi doslovno prenošenje funkcionalnosti funkcije članice, onda naknadno definisanje funkcije u nasleđenoj klasi, prilikom prenošenja objekta te klase kao parametra neke funkcije, ne izaziva očekivano dejstvo. To je zbog toga što se samo u slučaju definisanja virtual funkcija prenose sa pokazivačima na objekte i pokazivači na predefinisane funkcije, što je osnov polimorfizma u objektnom programiranju.

Apstrakcija podataka se odnosi na otkrivanje samo osnovnih informacija spoljašnjem svetu, sakrivajući detalje o tome kako je nešto implementirano. C++ klase obezbeđuju veoma visok nivo apstrakcije podataka. Klase obezbeđuju dovoljan broj javnih metoda preko kojih se korisnici igraju sa članovima klase, menjajući sadržaj i stanje, a pritom ne poznajući implementaciju konkretnih funkcija.

Šabloni predstavljaju osnov generičkog programiranja, koje uključuje pisanje koda u smislu da je funkcionalnost koda nezavisna od korišćenih tipova podataka. Šablon je ustvari formula za kreiranje generičkih klasa ili funkcija.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Uvod u polimorfizam

pokazivači na bazne klase, oblici, polimorfizam

- *Pokazivači i reference na osnovne klase objekata izvedene klase*
- *Osnovi o polimorfizmu*
- *Potreba za virtuelnim funkcijama*

01

POKAZIVAČI I REFERENCE NA OSNOVNE KLASE OBJEKATA IZVEDENE KLASE

Pokazivači/reference na bazne klase kojima dodelimo adresu objekta izvedene klase imaju pristup samo funkcijama i podacima bazne klase

Jedna od ključnih stvari izvedene klase je činjenica da je pokazivač/referenca na izvedenu klasu po tipu kompatibilan sa pokazivačem/referencom na baznu klasu. Krenućemo od jednog već obrađenog primera gde smo imali osnovu klasu Polygon, iz koje su zatim izvedene klase trougao i kvadrat. Kao dodatak uzećemo u razmatranje sledeću hijerarhiju klasa:

```
class CPolygon {
protected:
 int width, height;
public:
 void set_values (int a, int b)
 { width=a; height=b; }
};
class CRectangle: public CPolygon
{
public:
 int area ()
 { return (width * height); }
};
class CTriangle: public CPolygon {
public:
 int area ()
 { return (width * height / 2); }
};
```

Glavni program ćemo napisati kao:

```
int main ()
{
              CRectangle rect;
              CTriangle trgl;
              CPolygon * ppoly1 =
&rect;
              CPolygon * ppoly2 =
&trgl;
              ppoly1->set_values (4,5);
              ppoly2->set_values (4,5);
              cout << rect.area() <<
endl;
              cout << trgl.area() <<
endl;
              return 0;
}
```

U funkciji main kreiramo dva pokazivača na objekte klase CPolygon (ppoly1 i ppoly2). Zatim dodeljujemo adrese ovih objekata pokazivačima, tj. objekata rect i trgl. Pošto su oba objekta objekti klasa koje su izvedene iz klase CPolygon, obe operacija dodele su validne operacije. Dereferenciranje pokazivača ppoly1 i ppoly2 (sa *ppoly1 i *ppoly2) je takođe validno i omogućava nam da pristupimo članovima objekata na koji pokazuju. Stoga su naredna dva iskaza identična:

```
ppoly1->set_values (4,5);
rect.set_values (4,5);
```

ali s obzirom da su ppoly1 i ppoly2 pokazivači na tip Polygon (a ne na Rectangle niti na Triangle), moguće je pristupiti samo članovima koji su nasleđeni iz klase Polygon, ali ne i onim članovima izvedenih klasa Rectangle i Triangle. Stoga u prethodnom primeru, pristup članovima koji su karakteristični za trougao i pravougaonik se vrši direktno korišćenjem objekata rect i trgl, umesto korišćenjem pokazivača; Pokazivač na baznu klasu ne može da pristupi funkciji area koja je metoda klasa Rectangle i Triangle. Funkciji area smo mogli da pristupimo samo da je ona bila članica klase Polygon, umesto što je definisana u izvedenim klasama. Problem je u tome što klase Rectangle i Triangle implementiraju različite verzije funkcije area pa stoga ona nema zajednička svojstva.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

OSNOVI O POLIMORFIZMU

Polimorfizam znači da će poziv funkcije članice izazvati pozive različitih funkcija u zavisnosti od tipa objekta koji je pozvao funkciju

Princip polimorfizma predstavlja jedan od osnovnih principa objektno orijentisanog programiranja. On se ogleda u vrlo specifičnom principu dvojnosti koji je zastupljen u objektno orijentisanoj formulaciji problema. Treba usvojiti načela koja važe i pri prirodnom nasleđivanju tj. nasleđivanje znači selektivno usvajanje pravila i podataka unutar sistema u kome je ova osobina zastupljena. To znači da nije neophodno da se sve funkcije u istom obliku pridruže klasama-objektima naslednicima. Ovaj izmenjeni koridor pravila daje punu fleksibilnost u upotrebi nasleđivanja u objektno orijentisanoj analizi problema.

Već smo spomenuli da OOP ima tri glavne osobine a to su: enkapsulacija (sakrivanje podataka tj. učauravanje podataka i funkcija neke klase), nasleđivanje, i polimorfizam. Šta je polimorfizam? Generalno gledano, to je kada značenje neke instrukcije zavisi od konteksta, i rezultat neke instrukcije može da se menja u zavisnosti od konteksta. Jedan od osnovnih primera polimorfizma je kada različiti operatori rade različite stvari u zavisnosti od tipa objekta na koji se primenljuju, npr.

- int1 + int2 - obavlja sabiranje brojeva
- str1 + str2 - obavlja povezivanje stringova

Drugi primer je kada funkcije istog imena obavljaju različite stvari u zavisnosti od konteksta. Tako npr. predefinisane (overloaded) funkcije imaju isto ime ali različitu listu argumenata, ili nadglasavajuće (*overriding)* funkcije u izvedenoj klasi imaju isto ime i listu argumenata sa nadglasanim (*overridden)* funkcijama u baznoj klasi.

Reč polimorfizam znači da nešto može da ima više oblika. Tipično, polimorfizam se javlja kada imamo hijerarhiju klasa koje su povezane principima nasleđivanja. C++ polimorfizam znači da će poziv funkcije članice izazvati pozive različitih funkcija u zavisnosti od tipa objekta koji je pozvao funkciju.

Zašto je polimorfizam koristan? Polimorfizam čini program elegantnijim, lakšim za razumevanje, i za održavanje (modifikovanje). Kada primenimo polimorfizam onda nema potrebe da se pišu različita imena funkcija za akcije izvršene na različitim tipovima objekata, i jedna opšta ideja neke funkcije sa različitim verzijama implementacije ali sa istim imenom i istim argumentima, može da se zadrži sa istim pozivom čak i ako se ta ideja koristi sa različitim tipovima objekata.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

POTREBA ZA VIRTUELNIM FUNKCIJAMA

Virtuelne funkcije rešavaju problem pokazivača na bazne klase kojima se dodeljuje adresa objekta izvedene klase. Virtual naznačava da će funkcija biti naknadno prilagođena zahtevima

Da bi se u izvedenoj klasi promenila funkcionalnost neke izvedene klase, nije dovoljno predefinisati sadržaj te funkcije. Mora se sintaksno naglasiti da je konkretna funkcija predviđena za preobražavanje u izvedenim klasama. Ključna reč jezika koja pruža ovu mogućnost je virtual. Upotrebom virtuelnih (izmenjivih) funkcija, na mestu gde se inače označava tip povratne vrednosti funkcije, kompajleru se naznačava da se radi o funkciji koja će u izvedenoj klasi biti, prema potrebama, naknadno definisana / predefinisana. Bez upotrebe ovog sintaksnog pravila, klasa naslednica bezuslovno nasleđuje i funkcije članice, a to znači da, u konkretnim slučajevima funkcionalnost ne može biti prilagođena zahtevima. Da bi se u načelu kompajleru naglasila predviđena izmenljivost pojedinih funkcija članica, u sintaksi njihove deklaracije kao članica klasa mora se navesti ključna reč virtual.

Ovo ne znači da funkcija mora biti predefinasana u izvedenoj klasi već znači dozvoljenu mogućnost da se ona predefiniše.

Posmatrajmo detaljno sledeći primer kako bi usvojili gradivo o pokazivačima na bazne klase a zatim uveli korišćenje virtual funkcija. Pretpostavimo da imamo baznu klasu X i funkciju f() definisanu u baznoj klasi:

```
class Xclass
{
//...
          f();
//...
}
```

a zatim i klasu Y koja je izvedena iz X:

```
class Yclass : public Xclass
{
          //....
}
```

Neka je glavni program napisan na sledeći način:

```
int main()
{
          Xclass* p;
          Yclass y;
          Xclass x;

          p = &y;
          (*p).f();
          p = &x;
          (*p).f();
}
```

U prethodnom primeru pokazivač p je tipa Xclass*, tj. p je pokazivač na tip Xclass. Stoga p takođe može da pokazuje na objekte tj. da igra ulogu pokazivača za bilo koji objekat izvedene klase Yclass. Međutim, ono što smo već spomenuli, kada p pokazuje na neki objekat izvedene klase Yclass, njegov tip je i dalje Xclass*. Stoga će izraz (*p).f() pozvati funkciju f() definisanu u baznoj klasi. Naravno, izraz (*p).f() se može napisati i na sledeći način: p->f().

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Virtuelne funkcije

polimorfizam, virtuelne funkcije, nadglasavanje virtuelizacije

- *Definisanje virtuelnih funkcija*
- *Upotreba virtuelnih funkcija*
- *Upotreba virtuelnih funkcija*
- *Potreba za virtuelnim destruktorima*
- *Definisanje virtuelnog destruktora*
- *Čisto virtuelne funkcije; Nadglasavanje virtuelizacije;*

02

DEFINISANJE VIRTUELNIH FUNKCIJA

Funkcija se proglašava virtuelnom („izmenjivom“) korišćenjem ključne reči virtual u deklaraciji funkcije ispred povratnog tipa funkcija: virtual void f()

U prethodnom primeru postavlja se pitanje, šta ako Yclass ima nadglasavajuću verziju funkcije f() i pri tome p pokazuje na neki objekat klase Yclass. Koja će funkcija f() biti izvršena, Xclass::f() ili Yclass::f()? Odgovor je da će p->f() izvršiti Xclass::f() jer je p tipa Xclass*. To što p pokazuje tog momenta na objekat izvedene klase Yclass je nebitno. Pogledajmo sledeći primer:

```
#include <iostream>
using namespace std;
class Xclass{
public:
 void f() { cout << "Xclass::f() \n"; }
};
class Yclass : public Xclass {
public:
 void f() { cout << "Yclass::f() \n"; }
};

int main() {
 Xclass x;
 Yclass y;
 Xclass *p = &y;
 (*p).f(); // Poziva se Xclass::f() jer je p
tipa Xclass*
 p = &x;
 (*p).f(); // Poziva se Xclass::f() jer je p
tipa Xclass*
}
```

U prethodnom primeru imamo p koji je pokazivač objekata bazne klase Xclass, i prvo se on koristi da pokazuje na objekte klase Xclass a zatim i da pokazuje na objekte izvedene klase Yclass. Kao što vidimo imamo dva poziva funkcije p->f(), i oba pozivaju istu verziju funkcije f(), tj. one verzije koja je definisana u baznoj klasi Xclass, jer je p deklarisan da bude pokazivač za Xclass objekte. Stoga pokazivanje na objekte klase Yclass pomoću p nema efekta na drugi poziv p->f(); Da bi imali polimorfni efekat u gornjem primeru, potrebno je da proglasimo funkciju Xclass::f() da je virtuelna funkcija, pomoću ključne reči *virtual,* dodavanjem *virtual* u deklaraciji bazne funkcije, tj. da izmenimo klasu Xclass da bude:

```
class Xclass
{
public:
          virtual void f()
          {
                    cout << "Xclass::f() \n";
          }
};
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

UPOTREBA VIRTUELNIH FUNKCIJA

U slučaju korišćenja virtuelnih funkcija sa pokazivačima na objekte prenose se i pokazivači na predefinisane funkcije, što je osnov polimorfizma u objektnom programiranju

U primeru koji sledi ilustrovane su dve mogućnosti: da se funkcija deklariše bez prefiksa virtual i sa prefiksom virtual. Različiti rezultati pri izvršavanju programa najbolje ilustruju potrebu za eksplicitnom deklaracijom "izmenljivih" funkcija. Ukoliko se želi doslovno prenošenje funkcionalnosti funkcije članice, onda naknadno definisanje funkcije u nasleđenoj klasi, prilikom prenošenja objekta te klase kao paramtera neke funkcije, ne izaziva očekivano dejstvo. To je zbog toga što se samo u slučaju definisanja virtual funkcija prenose sa pokazivačima na objekte i pokazivači na predefinisane funkcije, što je osnov polimorfizma u objektnom programiranju. Na primeru koji sledi mogu se isprobati dve mogućnosti: nasleđivanje sa virtual funkcijama i bez njih. U prvom slučaju, program je potrebno prevesti u prikazanom obliku, a u drugom slučaju potrebno je definiciju simboličke konstante VIRTUAL staviti pod komentar. Pri izvršavanju programa za nedefinisanu konstantu VIRTUAL izvršava se funkcija povrs iz klase Kvadrat za izvedenu klasu Pravougaonik. Kada se primeni virtual deklaracija funkcije članice povrs, za objekat klase Kvadrat se površina izračunava kao kvadrat vrednosti stranice kvadrata (objekta), a u izvedenoj klasi površina pravougaonika se izračunava primenom predefinisane virtuelne funkcije povrs kao proizvod dužina jedne i druge stranice pravougaonika.

```
#define VIRTUAL
#include <stdio.h>

class Kvadrat{
public:
 int a;
#ifndef VIRTUAL
 int povrs(void);
#else
 virtual int povrs(void);
#endif
};

class Pravougaonik : public Kvadrat{
public:
 int b;
 // "inline" funkcija povrs
#ifndef VIRTUAL
 int povrs(void) {return a*b;};
#else
 virtual int povrs(void) {return a*b;};
#endif
};

int Kvadrat :: povrs(void)
{
 return a*a;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

UPOTREBA VIRTUELNIH FUNKCIJA

U slučaju korišćenja virtuelnih funkcija sa pokazivačima na objekte prenose se i pokazivači na predefinisane funkcije, što je osnov polimorfizma u objektnom programiranju

U primeru koji sledi ilustrovane su dve mogućnosti: da se funkcija deklariše bez prefiksa virtual i sa prefiksom virtual. Različiti rezultati pri izvršavanju programa najbolje ilustruju potrebu za eksplicitnom deklaracijom "izmenljivih" funkcija. Ukoliko se želi doslovno prenošenje funkcionalnosti funkcije članice, onda naknadno definisanje funkcije u nasleđenoj klasi, prilikom prenošenja objekta te klase kao paramtera neke funkcije, ne izaziva očekivano dejstvo. To je zbog toga što se samo u slučaju definisanja virtual funkcija prenose sa pokazivačima na objekte i pokazivači na predefinisane funkcije, što je osnov polimorfizma u objektnom programiranju. Na primeru koji sledi mogu se isprobati dve mogućnosti: nasleđivanje sa virtual funkcijama i bez njih. U prvom slučaju, program je potrebno prevesti u prikazanom obliku, a u drugom slučaju potrebno je definiciju simboličke konstante VIRTUAL staviti pod komentar. Pri izvršavanju programa za nedefinisanu konstantu VIRTUAL izvršava se funkcija povrs iz klase Kvadrat za izvedenu klasu Pravougaonik. Kada se primeni virtual deklaracija funkcije članice povrs, za objekat klase Kvadrat se površina izračunava kao kvadrat vrednosti stranice kvadrata (objekta), a u izvedenoj klasi površina pravougaonika se izračunava primenom predefinisane virtuelne funkcije povrs kao proizvod dužina jedne i druge stranice pravougaonika.

```
void stampa(Kvadrat *O)       // pointer na
objekat je parametar
{
 printf("\nPovrsina =%d", O->povrs());
}

void main(void)
{
 Kvadrat A ;             // objekat klase
Kvadrat
 Pravougaonik B ;         // objekat klase
Pravougaonik
 // dodela vrednosti clanovima objekata na koje
pokazuju A i B
 A.a=10;
 B.a=5;
 B.b=30;

 // poziv funkcije, prosledjuje se adresa
objekata
 stampa(&A);
 stampa(&B);
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

POTREBA ZA VIRTUELNIM DESTRUKTORIMA

Kod korišćenja pokazivača na bazne klase, s obzirom da se pozivaju samo funkcije bazne klase, pri uništavanju objekta biće takođe pozvan destruktor bazne klase

Iako C++ obezbeđuje podrazumevajući destruktor za vaše klase, postojaće slučajevi kada ćete hteti da napišete svoje destruktore (naročito u slučaju da klasa treba da dealocira memoriju). **Pravilo je da uvek treba kreirati virtuelni destruktor kada radimo sa nasleđivanjem**. Razmotrimo sledeći primer gde imamo deklarasane klase Base i Derived.

```
#include <iostream>
using namespace std;
class Base{
public:
 ~Base() {
  cout << "Calling ~Base()" << endl;
 }
};
class Derived: public Base{
private:
 int* m_pnArray;
public:
 Derived(int nLength) {
  m_pnArray = new int[nLength];
 }
 ~Derived() // note: not virtual {
  cout << "Calling ~Derived()" << endl;
  delete[] m_pnArray;
 }
};
```

Napišimo glavni program na sledeći način:

```
int main()
{
            Derived *pDerived = new Derived(5);
            Base *pBase = pDerived;
            delete pBase;

            return 0;
}
```

Pošto je pBase pokazivač na klasu Base, kada se pBase briše program prvo proverava da li je destruktor klase Base virtuelni. Ukoliko nije, onda pretpostavlja da je jedino neophodno da pozove upravo taj destruktor. Stoga možemo primetiti da će na osnovu ove činjenice prethodni primer oštampati poruku:

```
Calling ~Base()
```

međutim to nije ono što smo želeli da uradimo.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

DEFINISANJE VIRTUELNOG DESTRUKTORA

Opšte je pravilo da, kada imamo posla sa nasleđivanjem, destruktore klasa definišemo kao virtuelne, korišćenjem ključne reči virtual u deklaraciji

Ono što smo ustvari želeli je da se prilikom brisanja pBase prvo pozove destruktor klase Derived (ali naravno, iz njega će se zatim pozvati destruktor osnovne klase Base kao poslednji korak pre završetka tela destrukora klase Derived). Ovo ostvarujemo ako destruktor klase Base definišemo kao virtuelni:

```
class Base{
public:
 virtual ~Base() {
  cout << "Calling ~Base()" << endl;
 }
};

class Derived: public Base{
private:
 int* m_pnArray;
public:
 Derived(int nLength) {
  m_pnArray = new int[nLength];
 }
 virtual ~Derived() {
  cout << "Calling ~Derived()" << endl;
  delete[] m_pnArray;
 }
};
```

Ako sada pozovemo isti glavni program kao u prethodnom primeru:

```
#include <iostream>
using namespace std;
int main()
{
            Derived *pDerived = new Derived(5);
            Base *pBase = pDerived;
            delete pBase;

            return 0;
}
```

biće proizveden sledeći rezultat:

```
Calling ~Derived()
Calling ~Base()
```

Kao što možemo primetiti iz prethodnog primera koristili smo službenu reč virtual ispred naziva destruktora u klasi Base:

```
virtual ~Base()
```

odnosno ispred naziva destruktora klase Derived:

```
virtual ~Derived()
```

Naravno, da još jednom ponovimo: Kad god imate posla sa nasleđivanjem neophodno je da destrukore klasa definišete kao virtuelne!

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

ČISTO VIRTUELNE FUNKCIJE; NADGLASAVANJE VIRTUELIZACIJE; MANE VIRTUELNIH FUNKCIJA

Razrešavanje virtuelnog poziva funkcije traje više vremena nego razrešavanje poziva obične funkcije. Stoga nije efikasno svaku funkciju klase definisati kao virtuelnu (izmenjivu)

- **Čisto virtuelne funkcije**

Nekada je moguće da želimo da uključimo virtuelnu funkciju u baznu klasu, tako da možemo kasnije po mogućstvu da je predefinišemo u izvedenim klasama, ali da joj u baznoj klase ne dodelimo nikakvu ulogu. Možemo izmeniti funkciju area() jednog od prethodnih primera na sledeći način:

```
class Xclass
{
public:
            virtual void f() = 0;
};
```

pri čemu operator dodeljivanja „ = 0“ ukazuje kompajleru da funkcija nema telo i takve funkcije nazivamo **„čisto virtuelne funkcije“**.

- **Nadglasavanje virtualizacije**

Veoma je redak slučaj da hoćemo da izvršimo nadglasavanje virtualizacije ali korisno je pomenuti da i taj slučaj postoji. Pretpostavimo da imamo klase Base i Derived:

```
class Base
{
public:
            virtual const char* GetName() { return
"Base"; }
};
```

```
class Derived: public Base
{
public:
                        virtual const char*
GetName() { return "Derived"; }
};
```

Naime, mogu postojati slučajevi gde želimo da pokazivaču na osnovnu klasu dodelimo adresu objekta izvedene klase, tako da on pozove Base::GetName() umesto Derived::GetName(). Da bi smo ovo sproveli neophodno je samo koristiti operator rezolucije opsega, na sledeći način:

```
int main()
{
            Derived cDerived;
            Base &rBase = cDerived;
            // Calls Base::GetName() instead of the
virtualized Derived::GetName()
            cout << rBase.Base::GetName() << endl;
}
```

- **Mane virtuelnih funkcija**

S obzirom da u najvećem broju slučajeva postoji potreba za korišćenjem virtuelnih funkcija, onda se postavlja pitanje zašto sve funkcije ne postavimo da budu virtuelne? Odgovor je da to nije efikasno. Naime, razrešavanje virtuelnog poziva funkcije traje više vremena nego razrešavanje poziva obične funkcije. Osim toga, kompajler mora da alocira dodatni pokazivač za svaki objekat klase koja ima jednu ili više virtuelnih funkcija.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Apstrakcija podataka

Apstrakcija, interfejs, implementacija, ADT

- *Apstrakcija podataka u C++-u*
- *Specifikatori pristupa, pogodnosti apstrakcije i strategija projektovanja*
- *Primer apstrakcije podataka*

03

APSTRAKCIJA PODATAKA U C++-U

Apstrakcija podataka se odnosi na otkrivanje samo osnovnih informacija spoljašnjem svetu (tj interfejsa), sakrivajući detalje o tome kako je nešto implementirano

Apstrakcija podataka se odnosi na otkrivanje samo osnovnih informacija spoljašnjem svetu, sakrivajući detalje o tome kako je nešto implementirano. Apstrakcija podataka je programerska (a i konstruktorska) tehnika koja se temelji na razdvajanju interfejsa i implementacije.

Uzmimi jedan realan primer iz svakodnevnog života a to je TV, koji možete da upalite, ugasite, promenite kanal, podesite jačinu zvuka, i na koga možete povezati spoljašnje uređaje kao što su zvučnici, video rekorder, DVD. Međutim, kada je u pitanju TV nije neophodno poznavati unutrašnje detalje, tj. nije neophodno poznavati način na koji se primaju signali, kako se oni prevode u sliku, itd.... Stoga, možemo reći da je televizor jasno razdvojio unutrašnju implementaciju od spoljašnjeg interfejsa dospupnog korisnicima. Tako, korisnici mogu da se igraju sa interfejsom, pritiskajući dugmiće za paljenje/gašenje, promenu kanala, jačinu zvuka a pritom ne poznajući kako je to u televizoru sprovedeno u delo (implementirano).

Ako sada pričamo u terminima programiranja, C++ klase obezbeđuju veoma visok nivo apstrakcije podataka. Klase obezbeđuju dovoljan broj javnih metoda preko kojih se korisnici igraju sa članovima klase, menjajući sadržaj i stanje, a pritom ne poznajući implementaciju konkretnih funkcija.

Tako na primer, korisnički program može da pozove funkciju **sort()** a pritom ne poznajući algoritam koji funkcija koristi u cilju sortiranja zadatih vrednosti. Pri tome, implementacija funkcije može biti promenjena između dve stabilne verzije softvera, ali dokle god je interfejs ostao nepromenjen funkcija sort() će biti upotrebljiva.

U C++-u koristimo klase da bi smo definisali naše apstraktne tipove podataka (abstract data types - ADT). Kao što je poznato, možemo da koristimo **cout** objekat klase **ostream** da bi smo poslali podatke na standardni izlaz:

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

int main( )
{
   cout << "Hello C++" <<endl;
   return 0;
}
```

Mi naravno, pritom, ne moramo da poznajemo način kako cout šalje tekst na izlazni medijum. Za nas je dovoljno da poznajemo da cout postoji i da znamo šta on ustvari radi.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

SPECIFIKATORI PRISTUPA, POGODNOSTI APSTRAKCIJE I STRATEGIJA PROJEKTOVANJA

Pri kreiranju komponenata vašeg programa, praksa je da se interfejsi razdvoje od implementacije tako da u slučaju izmene implementacije vašeg koda, interfejsi ostanu netaknuti

Specifikatori pristupa i apstrakcija:

U C++-u možemo da koristimo specifikatore pristupa da bi smo definisali apstraktni interfejs klase. C++ klasa može da sadrži 0 (nula) ili više sekcija koje su definisane različitim specifikatorima pristupa (public, protected, private). Naravno, članovi definisani kao public su vidljivi iz svih delova programa. Nasuprot njima, članovi definisani u privatnoj sekciji nisu dostupni korisnicima klase.

U C++-u ne postoji ograničenje koje se odnosi na broj sekcija u klasi. Svaki specifikator pristupa definiše pristupni nivo članova klase koji slede. Jedan specifikator pristupa je validan ili do sledećeg specifikatora pristupa ili do zatvorene vitičaste zagrade koja označava kraj tela klase.

Pogodnosti apstrakcije podataka

Apstrakcija podataka obezbeđuje dve veoma važne prednosti:

- Unutrašnji (sakriveni) članovi klase su zaštićeni od nepažljivog korišćenja koje može da pokvari stanje objekta.
- Implementacija klase može biti izmenjena u toku vremena, u zavisnosti od korisničkih zahteva ili pojave grešaka (bagova) bez izmena u definiciji korisničkog interfejsa.

Definisanjem članova klase u privatnoj sekciji, kreator klase ima mogućnost izmene njenih članova. U slučaju izmene implementacije koda, samo izvorni kod definicije klase treba biti proveren da bi se uvidelo kakav efekat su izmene imale na funkcionalnost programa.

Strategija projektovanja

Apstrakcija razdvaja kod odnosno definiciju klase na interfejs i implementaciju. Stoga, pri konstruisanju komponenata vašeg programa, morate interfejse razdvojiti od implementacije tako da u slučaju izmene implementacije vašeg koda, interfejsi ostanu netaknuti. U ovim slučajevima, koji god deo programa da koristi pomenute interfejse, oni ostaju netaknuti i neophodno je samo izvršiti rekompajliranje kako bi se uključile poslednje izmene.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER APSTRAKCIJE PODATAKA

Apstrakcija podataka obezbeđuje da unutrašnji (sakriveni) članovi klase ostanu zaštićeni od nepažljivog korišćenja koje može da pokvari stanje objekta

Bilo koji C++ primer u kome je klasa implementirana korišćenjem javnih i privatnih članova je jedan primer apstrakcije podataka. Posmatrajmo sledeći primer klase Adder:

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

class Adder
{
public:
            // constructor
            Adder(int i = 0)
            {
                        total = i;
            }
            // interface to outside world
            void addNum(int number)
            {
                        total += number;
            }
            // interface to outside world
            int getTotal()
            {
                        return total;
            };
private:
            // hidden data from outside world
            int total;
};
```

Neka je glavni program napisan na sledeći način:

```
int main( )
{
            Adder a;

            a.addNum(10);
            a.addNum(20);
            a.addNum(30);

            cout << "Total " << a.getTotal() <<endl;
            return 0;
}
```

Nakon kompajliranja i izvršavanja dobija se sledeći rezultat:

```
Total 60
```

Prethodna klasa služi za sabiranje dva broja i kao rezultat vraća sumu. Javni članovi klase **addNum** i **getTotal** su interfejsi ka spoljašnjem svetu, tj. korisnici klase moraju poznavati ove funkcije da bi mogli da koriste klasu. Privatni podatak total je nešto što korisnik ne mora da poznaje, ali taj podatak je neophodan za pravilno funkcionisanje klase.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Apstraktne klase

Klase, apstraktne klase, interfejsi, čisto virtuelne funkcije

- *Uvod u apstraktne klase (C++ interfejsi)*
- *Definicija apstraktnih klasa*
- *Primer apstraktne klase*

04

UVOD U APSTRAKTNE KLASE (C++ INTERFEJSI)

Interfejs opisuje ponašanja i mogućnosti C++ klasa bez zalaženja u implementaciju. C++ interfejsi su implementirani korišćenjem apstraktnih klasa

Interfejs opisuje ponašanja i mogućnosti C++ klasa bez zalaženja u implementaciju, tj. bez poznavanja detalja kako je to ponašanje sprovedeno u delo. C++ interfejsi su implementirani korišćenjem apstraktnih klasa, i ove apstraktne klase ne treba mešati sa apstrakcijom podataka, što se odnosi na razdvanjanje detalja o implementaciji i deklaraciji odgovorajućih metoda.

Kao što smo već spomenuli, ako hoćemo da neka virtuelna funkcija bazne klase bude nadglasana u svim izvedenim klasama te bazne klase, onda nema potrebe da se ta virtuelna funkcija implementira u baznoj klasi. Ovo se radi tako da se takva virtuelna funkcija proglasi „čistom" virtuelnom funkcijom. Neka "čista virtuelna funkcija" je virtuelna funkcija koja nema implementaciju u svojoj klasi. Postupak za specificiranje "čiste" virtuelne funkcije je da se doda inicijalizator "=0" umesto tela funkcije, npr.

```
virtual int f1() =0;
```

Klase koje sadrže neku čistu virtuelnu funkciju se zovu "apstraktnim" klasama (apstraktna bazna klasa). Svrha apstraktne klase (**abstract class** ili često ABC) je da se obezbedi pogodna bazna klasa iz koje će biti izvedene ostale klase. Apstraktne klase ne mogu biti korišćene za instanciranje objekata i služe samo kao interfejs.

Neka klasa postaje apstraktna ako se bar jedna njena funkcija definiše kao „čisto virtuelna", kao što je dato u sledećem primeru:

```
class Box
{
   public:
      // pure virtual function
      virtual double getVolume() = 0;
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

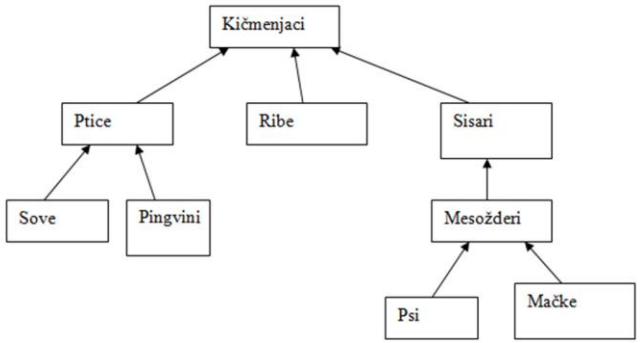
Pokušaj instanciranja objekta apstraktne klase će prouzrokovati kompajlersku grešku. Stoga, ukoliko neka klasa koja je izvedena iz ABC treba da bude instancirana onda je neophodno da se u njoj implementiraju sve čisto virtuelne funkcije bazne klase, što znači da izvedena klasa treba da podrži interfejs koji predstavlja apstraktna klasa ABC. Ukoliko je neuspešno nadglasavanje virtuelnih funkcije u izvedenoj klasi, onda će pokušaj instanciranja objekta izvedene klase takođe prouzrokovati kompajlersku grešku. Za razliku od apstraktnih klasa, klase koje mogu biti korišćene za instanciranje objekata se nazivaju **konkretne klase**. Dakle, "konkretna" izvedena klasa je ona koja ne sadrži ni jednu čistu virtuelnu funkciju

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

DEFINICIJA APSTRAKTNIH KLASA

Klase koje sadrže neku čistu virtuelnu funkciju se zovu "apstraktnim" klasama. Apstraktne klase ne mogu biti korišćenje za instanciranje objekata i služe samo kao interfejs

Neki dobro-dizajniran složeni o.o. program obično uključuje neku hijerarhiju klasa čije međurelacije izgledaju kao neko stablo, kao što je nacrtano na slici koja sledi. Pri tome, klase na nižim nivoima (npr. Sova, Riba, Pas, itd.) uključuju specifične funkcije koje implementiraju specifično ponašanje tih klasa (npr. Riba.plivaj(), Sova.leti(), Pas.kopaj(), itd.).



Slika-1 Primer hijerarhije nasleđivanja kod životinja

Tako, npr, možemo da odlučimo da će u klasi Kičmenjaci, funkcija jesti() biti nadglasana u svim klasama izvedenim iz ove klase, i onda se ona može proglasiti „čistom" virtuelnom funkcijom. Evo kako bi to izgledalo:

```
class Kicmenjaci {
public:
          virtual void jesti() =0;
};
```

```
class Ribe : public Kicmenjaci{
public:
          void jesti();
          //implementirano za klasu  Ribe negde
dalje u programu
};
```

Dakle, virtuelne funkcije se mogu napraviti da budu čiste virtuelne funkcije, koje moraju biti nadglasane u izvedenim klasama da bi bile korišćene, tako što treba dodati =0 u prototipu virtuelne funkcije, npr:

```
class Oblik {
public:
          virtual double povrs() = 0;
};
```

A iz ove klase kao bazne klase mogu se izvesti sledeće klase:

```
class Kvadrat: public Oblik{...};
class Trougao: public Oblik{...};
```

U C++-u mogu da postoje takozvane "capability" klase koje sadrže samo virtuelne funkcije članove ali ne i podatke članove, dakle imaju samo skup virtuelnih metoda. Jedina uloga ovakvih klasa je da posluže kao bazne klase i da omoguće formiranje izvedenih klasa.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER APSTRAKTNE KLASE

Za razliku od apstraktnih klasa, klase koje mogu biti korišćene za instanciranje objekata se nazivaju konkretne klase

Razmotrimo sledeći primer gde bazna klasa predstavlja interfejs na osnovu koga se zatim definišu metode u izvedenim klasama koje računaju površinu pravougaonika, odnosno trougla korišćenjem funkcije **getArea()**:

```
#include <iostream>
using namespace std;
class Shape {
public:
 virtual int getArea() = 0;
 void setWidth(int w) {   width = w; }
 void setHeight(int h) {  height = h; }
protected:
 int width;
 int height;
};

class Rectangle: public Shape{
public:
 int getArea() {   return (width * height);  }
};
class Triangle: public Shape{
public:
 int getArea() {   return (width * height)/2;
}
};
```

Glavni program ćemo napisati na sledeći način:

```
int main(void)
{
            Rectangle Rect;
            Triangle  Tri;

            Rect.setWidth(5);
            Rect.setHeight(7);
            // Print the area of the object.
            cout << "Total Rectangle area: " <<
Rect.getArea() << endl;

            Tri.setWidth(5);
            Tri.setHeight(7);
            // Print the area of the object.
            cout << "Total Triangle area: " <<
Tri.getArea() << endl;

            return 0;
}
```

Rezultat će biti:

```
Total Rectangle area: 35
Total Triangle area: 17
```

Na osnovu ovog primera može se primetiti kako jedna apstraktna klasa definiše interfejs tj. daje informaciju korisniku klase da postoji funkcija getArea() koju može da koristi, ali je u ostalim dvema klasama ustvari izvršena implementacija ove funkcije na dva različita načina, prema konkretnim potrebama za jednu i drugu klasu.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Osnovi o hijerarhiji klasa

Apsrakcija, virtuelne funkcije, hijerarhija klasa

- *Hijerarhija klasa*

05

HIJERARHIJA KLASA

U praksi imamo dosta slučajeva kada se formiraju složene hijerarhije klasa, gde postoji niz apstraktnih baznih klasa hijerarhijski povezanih

U praksi imamo dosta slučajeva kada se formiraju složene hijerarhije klasa, gde postoji niz apstraktnih baznih klasa hijerarhijski povezanih. To može da bude npr. složena hijerarhija klasa koja počinje sa klasom „Država“ kao jednom apstraktnom baznom klasom. Zatim, može se kreirati izvedena klasa „Evropa“ takođe kao apstraktna bazna klasa, ADT, i onda klasa koja nije ADT, klasa „Srbija“ koja je izvedena iz klase Evropa. Klasa Drzava može biti deklarisana kao:

```
class Drzava
{
public:
            virtual void Continent() const = 0;
//cista virtuelna metoda
            virtual string GetIme() (return ime;}
protected:
            string ime;
};
```

klasa Evropa kao:

```
class Evropa : public Drzava //izvedena ADT
{
public:
            virtual bool Euclan() const =0; //cista
virtuelna metoda
            virtual void Continent() const
{cout<<“Evropa“<<endl;}
};
```

dok klasa Srbija može biti definisana na sledeći način:

```
class Srbija : public Evropa
{
public:
            virtual bool Euclan() const {return
true}
};
```

Sve ove klase mogu se smestiti npr. u fajl drzave.h, a glavni program u fajl main.cpp, koji je dat u nastavku:

```
#include "drzave.h"
int main
{
            Evropa *Srbija = new Srbija;
            //....
            return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Šabloni funkcija

Šabloni, templates, preklapanje funkcija, preklapanje operatora

- *Uvod u šablone*
- *Definisanje i upotreba šablona funkcija*
- *Upotreba šablona funkcija - određivanje maksimuma različitih tipova podataka*
- *Operatori, pozivi funkcija i šabloni funkcija*
- *Problemi kod šablona funkcija*

06

UVOD U ŠABLONE

Šabloni predstavljaju osnov generičkog programiranja, koje uključuje pisanje koda u smislu da je funkcionalnost koda nezavisna od korišćenih tipova podataka

Šabloni predstavljaju osnov generičkog programiranja, koje uključuje pisanje koda u smislu da je funkcionalnost koda nezavisna od korišćenih tipova podataka. Šablon je ustvari formula za kreiranje generičkih klasa ili funkcija. C++ kontejneri, kao što su iteratori, su primer generičkog programiranja korišćenjem koncepta šablona. Šabloni se mogu koristiti za definisanje klasa i funkcija.

Šablon tj *template* u C++-u je jedan apstraktni recept za pravljenje konkretnog koda. Kompajler koristi šablon da generiše programski kod za različite funkcije i klase. Jedan isti šablon omogućuje da se generiše puno različitih verzija odnosno instanci šablona. Ovo se postiže pomoću šablonskih ulaznih parametara (tzv. „argumenata“), koji funkcionišu za šablone na isti način kako obični parametri funkcionišu za funkcije. Ali, obični parametri su predviđeni za konkretne objekte tj. konkretne vrednosti podataka, a parametri šablona su predviđeni za tipove podataka i klase. Mehanizam koji C++ nudi za proizvodnju instanci šablona je vrlo važna osobina C++, koja ga razlikuje od mnogih programskih jezika. Ovaj mehanizam omogućuje automatsku proizvodnju koda, što bitno povećava efikasnost kodiranja. Osnovna forma definicije šablona funkcije ima sledeći oblik:

```
template <class type> ret-type func-name(parameter
list)
{
   // body of function
}
```

gde je type naziv tipa podatka ili klase koja će biti korišćena od strane funkcije. Ovaj naziv može biti korišćen unutar tela tj. definicije funkcije. Simbol type (uglavnom se koristi oznaka T) je type parameter, odnosno „tipski parametar“. Ovaj tipski parametar može biti bilo koji ugrađeni ili složeni tip podatka. Generalno gledano, deklaracija šablona funkcije je ista kao i obična deklaracija funkcije, osim što se ispred stavi specifikacija template<...>, a tipski parametar T se zatim može koristiti umesto običnih tipova kao u sledećem primeru:

```
template<class T>
T squareF(T tV) {return tV*tV;}
```

Inače, šablon može imati nekoliko tipskih parametara, razdvojenih zarezom, npr.

```
template <class T, clas U, class X>
```

Šabloni funkcija su u stvari generalizacija koncepta predefinisanja odnosno preklapanja funkcija (function overloading). Naime, uvek smo mogli da napišemo nekoliko „predefinisanih“, overloaded, verzija neke funkcije. Ali, jedna šablonska funkcija zamenjuje ovih nekoliko predefinisanih funkcija.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

DEFINISANJE I UPOTREBA ŠABLONA FUNKCIJA

Deklaracija šablona funkcije je ista kao i obična deklaracija funkcije, osim što se ispred stavi specifikacija template<class T> gde se T odnosi na opšti tip podatka

Posmatrajmo primer funkcije koja se koristi da zameni vrednosti para elemenata, tj. funkcija swap, koja se koristi u mnogim algoritmima za sortiranje. Npr. sledeća funkcija operiše sa celim brojevima:

```
void swapF(int &mV, int &nV)
{
          int tempV = mV;
          mV = nV;
          nV = tempV;
}
```

Međutim, ukoliko želimo da radimo sa stringovima onda nam treba sledeći oblik funkcije swap():

```
void swapF(string &str1, string &str2)
{
          string tempV = str1;
          str1 = str2;
          str2 = tempV;
}
```

Ove dve funkcije rade istu stvar, ali koriste različite tipove objekata. Obe funkcije se mogu predstaviti jedinstvenim šablonom, tj, šablonom funkcije. Stoga je za prethodni primer moguće napisati jedinstven šablon na sledeći način:

```
template <class T>
void swapF(T &xV, T &yV)
{
          T tempV = xV;
          xV = yV;
          yV = tempV;
}
```

U nastavku je dat primer korišćenja šablona koji smo prethodno kreirali. Kao što možemo primetiti, poziv šablona se ostvaruje kao i poziv obične funkcije:

```
int main()
{
          int mV=25, nV=75;
          swapF(mV, nV) ;
          string str1="john", str2="jim";
          swapF(str1, str2);
}
```

Kod svakog poziva šablona funkcije, kompajler generiše kompletnu funkciju instancu, zamenjujući odgovarajući „tipski“ parametar sa „tipom“ ili klasom. Stoga će poziv *swapF(mV, nV)* generisati *swapF* funkciju za celobrojne podatke, a poziv *swapF(str1, str2) će generisati swapF* funkciju za string objekte.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

UPOTREBA ŠABLONA FUNKCIJA - ODREĐIVANJE MAKSIMUMA RAZLIČITIH TIPOVA PODATAKA

U standardnoj C++ biblioteci postoji ogroman broj funkcija koje su napisane korišćenje šablona. Poziv šablona funkcije se ostvaruje kao i poziv obične funkcije

Standardna C++ biblioteka (Standard C++ Library) pruža čitavu seriju osnovnih funkcija. Postoje funkcije koje su primenljive na različite tipove podataka. Npr. funkcija biblioteke algorithm pod nazivom max(arg1, arg2) može da se primeni na celobrojne (*int*) *i* realne (*double*) promenljive, kao i na znakove (*char*). Probajmo sami da napišemo preklopljene funkcije koje će računati maximum brojeva na sledeći način:

```
double maximumF(double d1V, double d2V)
{
          if (d1V > d2V) return d1V;
          else           return d2V;
}

int maximumF(int n1V, int n2V)
{
          if (n1V > n2V) return n1V;
          else           return n2V;
}

char maximumF(char c1V, char c2V)
{
          if (c1V > c2V) return c1V;
          else           return c2V;
}
```

Sve tri prethodno navedene funkcije su skoro identične, i bilo bi dobro koristiti samo jednu funkciju, generalizujuću funkciju, koja se može primeniti na različite tipove ulaznih podataka. Pogledajmo sada sledeći program, gde se definiše šablon za generalizovanu maximumF( ) funkciju:

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

template <class T>
T maximumF(T t1V, T t2V){
 if (t1V > t2V) {  return t1V; }
 return t2V;
};

int main(int argc, char* pArgs[]){
 cout << "maximum: "
  << maximumF(10, 20)
  << endl;
 cout << "maximum: "
  << maximumF(1.25, 2.35)
  << endl;
 system("PAUSE");
 return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

OPERATORI, POZIVI FUNKCIJA I ŠABLONI FUNKCIJA

Pri korišćenju šablona funkcije, moraju biti definisani svi operatori i pozivi funkcija za tipove za koje se instancira šablon funkcije

Kada kompajler pokuša da kompajlira instancu šablona, on je kompajlira kao i običnu funkciju. U običnoj funkciji svaki operator ili funkcijski poziv za tipove koje koristimo mora biti definisan inače će da se javi kompajlerska greška. Slično tome, i kod korišćenja šablona funkcije, moraju biti definisani svi operatori i pozivi funkcija za tipove za koje se instancira šablon funkcije. Da bi ovo bilo jasno pogledajmo sledeći primer, gde ćemo koristiti šablon funkcije koja računa maksimum od dva podatka:

```
template <typename Type>
Type max(Type tX, Type tY)
{
          return (tX > tY) ? tX : tY;
}
```

a zatim ćemo kreirati jednu klasu koja će da služi za testiranje:

```
class Cents
{
private:
          int m_nCents;
public:
          Cents(int nCents)
                    : m_nCents(nCents)
          {
          }
};
```

Pogledajmo sada šta će da se desi kada pokušamo da pozovemo šablonsku funkciju max() koja kao parametre ima objekte klase Cents:

```
Cents cNickle(5);
Cents cDime(10);
Cents cBigger = max(cNickle, cDime);
```

C++ će naravno kreirati instancu šablona max() koji će imati sledeći oblik:

```
Cents max(Cents tX, Cents tY)
{
          return (tX > tY) ? tX : tY;
}
```

i pokušaće da iskompajlira funkciju. Da li primećujete problem? C++ nema ideju kako da proceni izraz tX > tY pa će se javiti kompajlerska greška. Da bi rešili ovaj problem neophodno je da u klasi Cents izvršimo preklapanje operatora >. Ovo naravno moramo uraditi za svaku klasu koja želi da koristi šablonsku funkciju max(). Stoga je u klasi Cents neophodno definisati operator > na sledeći način:

```
class Cents
{
//...
          friend bool operator>(Cents &c1, Cents&c2)
          {
                    return (c1.m_nCents >
c2.m_nCents) ? true: false;
          }
};
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PROBLEMI KOD ŠABLONA FUNKCIJA

Ukoliko nisu definisani svi operatori i pozivi funkcija za klasu koju koristimo u šablonu neophodno je izvršiti preklapanje operatora

Uradimo još jedan primer sa šablonima funkcija. Sledeći šablon funkcija će služiti za određivanje srednje vrednosti sume elemenata niza:

```
template <class T>
T Average(T *atArray, int nNumValues)
{
            T tSum = 0;
            for (int nCount=0; nCount < nNumValues;
nCount++)
                        tSum += atArray[nCount];

            tSum /= nNumValues;
            return tSum;
}
```

U slučaju da u glavnom programu napišemo sledeće linije koda:

```
int anArray[] = { 5, 3, 2, 1, 4 };
cout << Average(anArray, 5) << endl;

double dnArray[] = { 3.12, 3.45, 9.23, 6.34 };
cout << Average(dnArray, 4) << endl;
```

Biće proizveden sledeći rezultat:
3
5.535
Kao što vidimo iz prethodnog primera, ovakva šablonska funkcija je odlična za rad sa ugrađenim tipovima podataka.

Međutim, pogledajmo šta će se desiti ako koristimo niz objekata klase Cents:

```
Cents cArray[] = { Cents(5), Cents(10), Cents(15), Cents(14) };
cout << Average(cArray, 4) << endl;
```

Naravno, kompajler će prijaviti puno grešaka, a jedna od njih će biti i:

```
c:test.cpp(45) : error C2679: binary '<<' : no operator found
which takes a right-hand operand of
    type 'Cents' (or there is no acceptable conversion)
```

Naravno, funkcija Average() kao rezultat vraća objekat tipa Cents, a mi pokušavamo da taj objekat ubacimo u tok korišćenjem cout i operatora ubacivanja u tok <<. Ono što nemamo još uvek je preklopljeni operator << klase Cents. Stoga ga moramo definisati, i to možemo uraditi na sledeći način:

```
friend ostream& operator<< (ostream &out, const
Cents &cCents)
{
            out << cCents.m_nCents << " cents ";
            return out;
}
```

Ukoliko sada pokušamo da kompajliramo program pojaviće se još jedna greška:

```
c:test.cpp(14) : error C2676: binary '+=' : 'Cents' does not
define this operator or a conversion to a type
   acceptable to the predefined operator
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PREKLAPANJE OPERACIJA KOD ŠABLONA FUNKCIJA

Kod korišćenja šablona praksa je da se preklapanjem operatora klasa prilagodi šablonu, a nikako nije poželjno prilagođavanje šablona za korišćenje nad objekatima tačno određene klase

Prethodnu grešku je naravno izazvao šablon funkcije. Da bi uvideli u čemu je problem pogledajmo kako će izgledati šablon funkcije kada radi sa objektima tipa Cents:

```
template <class T>
Cents Average(Cents  *atArray, int nNumValues)
{
            Cents tSum = 0;
            for (int nCount=0; nCount < nNumValues;
nCount++)
                        tSum += atArray[nCount];

            tSum /= nNumValues;
            return tSum;
}
```

Razlog kompajlerske greške je naravno sledeća linija koda:

```
tSum += atArray[nCount];
```

U ovom slučaju, tSum je objekat klase Cents za koji nije definisan operator +=, pa stoga moramo definisati ovu funkciju da bi šablonsku funkciju Average() mogli da koristimo i za objekte klase Cents. Kao što možemo primetiti, neophodno je takođe definisati i operator /= operator, tako da će konačna klasa Cents imati sledeći izgled:

```
class Cents
{
private:
            int m_nCents;
public:
            Cents(int nCents) : m_nCents(nCents) { }

            friend ostream& operator<< (ostream
&out, const Cents &cCents)
            {
                        out << cCents.m_nCents << "
cents ";
                        return out;
            }

            void operator+=(Cents cCents)
            {
                        m_nCents += cCents.m_nCents;
            }

            void operator/=(int nValue)
            {
                        m_nCents /= nValue;
            }
};
```

Primetimo da nismo uopšte imali potrebu da menjamo funkciju Average() da bi ona bila upotrebljiva za objekte klase Cents. Ono što je naravno bilo neophodno je da prilagodimo klasu Cents šablonskoj funkciji Average() a to smo uradili definisanjem novih operatora += i /=.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

VREDNOSNI PARAMETAR ŠABLONA (EXPRESSION PARAMETERS)

Parametri šablona ne moraju da budu samo tipovi, kao što su klasa ili ime tipa, već mogu da budu i izrazi čija je vrednosti odgovarajućeg tipa podatka

Parametri šablona ne moraju da budu samo tipovi, kao što su klasa ili ime tipa, već mogu da budu i izrazi čija je vrednosti odgovarajućeg tipa podatka. Tako na primer možemo imati sledeću definiciju šablona:

```
// template arguments
#include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
            return val * N;
}

int main()
{
            std::cout << fixed_multiply<int,2>(10)
<< '\n';
            std::cout << fixed_multiply<int,3>(10)
<< '\n';
}
```

Rezultat prethodnog programa biće:

```
20
30
```

Drugi argument šablonske funkcije fixed_multiply je podatak tipa int. On izgleda kao regularni parametar fukcije, i ustvari u velikom broju slučajeva može biti i korišćen u takvom obliku. Ali, između tipa i izraza kao parametara šablona postoji bitna razlika: Vrednost parametara šablona se određuje u trenutku kompajliranja tako da se vrši definisanje različitih instanci funkcije fixed_multiply, pa se stoga vrednost parametra nikad ne prosleđuje u toku izvršavanja programa. Dva poziva funkcije fixed_multiply iz glavnog programa main ustvari generišu dve različite verzije funkcije: prvu koja će uvek biti pomnožena sa 2, i drugu koja će uvek biti pomnožena sa 3. Pošto se ovo obrađuje u trenutku kompajliranja, drugi parametar šablona mora da bude konstantan izraz (nikako promenljiva vrednost).

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Šabloni klasa

Šabloni klasa, vrednosni parametri, specijalizovani šabloni

- *Definicija šablona klasa*
- *Primer šablona klase*
- *Vrednosni parametri šablona klasa*
- *Specijalizovan šablon klase*
- *Implementacija specijalizovanog šablona klase*

07

DEFINICIJA ŠABLONA KLASA

Šabloni klasa se definišu na sličan način kao i šabloni funkcija. Funkcije članice šablona klase su ustvari šablonske funkcije

Šablon klase (*class template*) funkcioniše na isti način kao i šablon funkcije, osim što on proizvodi klase a ne funkcije. Osnovna forma deklaracije šablona klasa (generičke klase) je:

```
template <class type> class class-name {
...
}
```

gde, **type** predstavlja tip podatka koje treba biti naveden prilikom instanciranja objekta klase. Sintaksa može imati sledeći oblik:

```
template<class T>
class Xclass {......};
```

Šablon klase (templejt) može imati nekoliko šablonskih parametara koji su razdvojeni zarezima, npr.

```
template<class T, class U>
class Xclass {......};
```

U nastavku je dat jedan jednostavan primer gde se kreiraju šabloni klasa:

```
template<class T>
class Xclass
{
          ......
};
```

pri čemu glavni program može biti napisan kao:

```
int main()
{
          Xclass<float> x10;
          Xclass<int> x20;
}
```

Ono što treba napomenuti ovde je da deklaracija:

```
Xclass<int> x0;
```

u glavnom programu generiše odgovarajuću klasu Xclass (za tip ***int***), i takođe generiše objekat xO. Funkcije članovi u šablonu klase su ustvari šablonske funkcije, i koristi se pri tome isti tipski parametar kao za šablonsku klasu. Radi boljeg uvida, u nastavku je primer:

```
template<class T>
class Xclass
{
          T squareF(T t) {return t*t;}
};
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER ŠABLONA KLASE

Šablon klase može imati nekoliko tipskih parametara koji su razdvojeni zarezima. Ovi tipski parametri biće korišćeni i za šablone funkcija koje su članice klase

Pogledajmo još jedan primer:

```
template <class T>
class mypair
{
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first;
      values[1]=second;
    }
};
```

Klasa koju smo upravo definisali služi za čuvanje dva podatka bilo kog tipa. Na primer, ukoliko želimo da deklarišemo objekat ove klase koji će da služi za čuvanje dva podatka tipa int, s tim da su to brojevi 115 i 36, to radimo na sledeći način:

```
mypair<int> myobject (115, 36);
```

Ista klasa takođe može da posluži za kreiranje objekta koji će da čuva podatke nekog drugog tipa, npr double:

```
mypair<double> myfloats (3.0, 2.18);
```

Kao što možemo primetiti, jedina funkcija članica šablona klase je konstruktor koji je definisan kao linijska (inline) funkcija u okviru klase. Ukoliko želimo da definišemo konstruktor van tela šablona klase, onda njemu mora prethoditi prefiks koji se navodi kod šablona, tj. template <...> prefiks:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}
```

Glavni program pišemo kao:

```
int main ()
{
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

Rezultat prethodnog programa će biti:
100

Primetimo u prethodnom primeru da je funkcija članica getmax definisana na sledeći, možda malo i konfuzan, način:

```
template <class T>
T mypair<T>::getmax ()
```

Pokušajmo da razjasnimo prethodnu deklaraciju. Kao što se vidi imamo tri T oznake u deklaraciji. Prvo T (<class T>) se odnosi na parameter šablona. Drugo T se odnosi na rezultat odnosno povratnu vrednost funkcije. Poslednje, tj. treće T (<T>) je takođe neophodno i definiše da je šablonski parameter T koji koristi funkcija getmax() isti onaj šablonski parameter koji se odnosi na šablon klase.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

VREDNOSNI PARAMETRI ŠABLONA KLASA

Vrednosni parametri šablona klasa su parametri šablona koji nisu zamena za tip nego za neku vrednost. Vrednosni parametar može da bude konstanta celobrojnog tipa, pokazivač ili referenca

Postoje parametri šablona koji nisu zamena za tip već za neku vrednost. To su takozvani "template expression parameters" odnosno vrednosni parametri šablona. Jedan vrednosni parametar šablona može da bude:

- Vrednost koja je celobrojnog ili nabrojivog (enum) tipa
- Pokazivač ili referenca na objekat
- Pokazivač ili referenca na funkciju
- Pokazivač ili referenca na funkciju članicu klase

U primeru koji sledi kreiraćemo klasu Buffer koja koristi istovremeno tipski (type) i vrednosni (expression) parametar. Tipski parametar kontroliše tip podatka niza u klasi Buffer, dok se vrednosni parametar koristi da kontroliše veličinu niza u baferu.

```
template <typename T, int nSize>
class Buffer
{
private:
            T m_atBuffer[nSize];
public:
            T* GetBuffer() { return m_atBuffer; }
            T& operator[](int nIndex)
            {
                        return m_atBuffer[nIndex];
            }
};
```

Glavni program pišemo kao:

```
int main()
{
            Buffer<int, 12> cIntBuffer;
            for (int nCount=0; nCount < 12; nCount++)
                        cIntBuffer[nCount] = nCount;
            for (int nCount=11; nCount >= 0; nCount--)
                        std::cout << cIntBuffer[nCount]
<< " ";
            std::cout << std::endl;

            Buffer<char, 31> cCharBuffer;
            strcpy(cCharBuffer.GetBuffer(), "Hello
there!");
            std::cout << cCharBuffer.GetBuffer() <<
std::endl;

            return 0;
}
```

Nakon kompajliranja i izvršavanja programa dobiće se sledeći rezultat:

11 10 9 8 7 6 5 4 3 2 1 0

Hello there!

Jedna bitna stvar u vezi prethodnog primera je to da mi ne moramo da dinamički alociramo niz m_atBuffer koji je član klase! Razlog je taj što za bilo koju instancu klase Buffer, nSize je ustvari konstanta. Na primer, ako instanciramo objekat klase Buffer, kompajler će zameniti nSize sa 12. Stoga će se generisati statički niz m_atBuffer koji ima 12 elemenata tipa int.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

SPECIJALIZOVAN ŠABLON KLASE

C++ dozvoljava definisanje drugačije implementacije šablona za specijalnu potrebu kada se određeni tip prosledi kao argument šablona

C++ dozvoljava definisanje drugačije implementacije šablona kada se određeni tip prosledi kao argument šablona. Ovo se naziva nadogradnja šablona odnosno - *template specialization*. Pretpostavimo, na primer, da imamo veoma prostu klasu koja se zove mycontainer koja može da skladišti element bilo kod tipa, i ima samo jednu funkciju članicu increase koja uvećava vrednost elementa. Ali, pretpostavimo da je u slučaju da se čuva podatak tipa char mnogo pogodnije imati potpuno drugačiju implementaciju gde bi u klasi imali fukciju članicu uppercase koja će pretvarati mala slova u velika.

Sintaksa koju ćemo koristiti za specijalizaciju šablona ima sledeći oblik:

```
template <> class mycontainer <char> { ... };
```

Primetimo, pre svega da imenu klase prethodi template<>, bez argumenata. Ovo je urađeno jer su svi tipovi poznati tako da nisu neophodni novi tipski parametri za specijalizaciju, ali ipak, pošto se radi o specijalizaciji šablona klase, stoga je neophodno naglasiti da se radi o šablonu.

Ono što je važnije od ovog prefiksa je ustvari parametar specijalizacije <char> koji se navodi posle naziva šablona klase. Ovaj parametar specijalizacije sam po sebi definiše tip za koji se ovaj šablon specijalizuje a to je tip char. Radi boljeg uvida u razlike između generičkih šablona klasa i specijalizacije, razmotrimo sledeći primer:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

Prva linija prethodnog primera se odnosi na generički šablon, a druga linija se odnosi na specijalizaciju. Kada deklarišemo specijalizaciju za šablon klase, moramo takođe definisati i sve članove specijalizacije, čak i one koje su identični i u generičkom šablonu klase. Razlog je taj jer ne postoji nasleđivanje članova generičkog šablona u odgovarajuću specijalizaciju.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

IMPLEMENTACIJA SPECIJALIZOVANOG ŠABLONA KLASE

Specijalizovan šablon se koristi u slučajevima kada je ipak neophodno šablon prilagoditi specijalnom tipu podatka, iako se obična verzija šablona već koristi za ostale tipove podataka

Stoga, naš specijalizovan šablon klase možemo deklarisati na sledeći način:

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer
{
            T element;
public:
            mycontainer (T arg) {element=arg;}
            T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char>
{
            char element;
public:
            mycontainer (char arg) {element=arg;}
            char uppercase ()
            {
                        if
((element>='a')&&(element<='z'))
                                    element+='A'-
'a';
                        return element;
            }
};
```

Glavni program možemo napisati kao:

```
int main ()
{
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

Rezultat programa biće:

```
8
J
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Vežbe

Polimorfizam, apstrakcija, hijerarhija klasa, šabloni

- *Virtuelne funkcije - Primer1*
- *Virtuelne funkcije - Primer2*
- *Virtuelne funkcije - Primer3*
- *Primer. Čisto virtuelne funkcije*
- *Primer. Šabloni funkcija*
- *Primer. Šabloni klase*

08

VIRTUELNE FUNKCIJE - PRIMER1

Neka funkcija je deklarisana kao virtuelna u baznoj klasi da bi kasnije mogla biti redefinisana u izvedenim klasama

U nastavku je dat primer korišćenja virtuelnih funkcija u hijerarhiji klasa CPolygon, CRectangle i CTriangle:

```
// virtual members
#include <iostream>
using namespace std;

class CPolygon
{
protected:
 int width, height;
public:
 void set_values (int a, int b)
 { width=a; height=b; }
 virtual int area ()
 { return (0); }
};

class CRectangle: public CPolygon
{
public:
 int area ()
 { return (width * height); }
};

class CTriangle: public CPolygon
{
public:
 int area ()
 { return (width * height / 2); }
};
```

Glavni program za testiranje možemo napisati kao:

```
int main ()
{
            CRectangle rect;
            CTriangle trgl;
            CPolygon poly;
            CPolygon * ppoly1 = &rect;
            CPolygon * ppoly2 = &trgl;
            CPolygon * ppoly3 = &poly;
            ppoly1->set_values (4,5);
            ppoly2->set_values (4,5);
            ppoly3->set_values (4,5);
            cout << ppoly1->area() << endl;
            cout << ppoly2->area() << endl;
            cout << ppoly3->area() << endl;
            return 0;
}
```

Sve tri klase imaju iste članove: width, height, set_values() i area(). Funkcija članica area() je deklarisana kao virtuelna u baznoj klasi da bi kasnije mogla biti redefinisana u izvedenim klasama. Možemo primetiti da ako uklonimo ključnu reč virtual ispred deklaracije funkcije area u klasi CPolygon rezultat programa će biti 0 za svaki poziv funkcije area umesto 20,10,0, što se dobija ako ostane ključna reč virtual.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

VIRTUELNE FUNKCIJE - PRIMER2

Cilj primera je da se pokaže primena virtuelnih funkcija kod dinamički alociranih objekata. Treba detaljno analizirati primer i dodati implementacije koje nedostaju

U nastavku je dat još jedan primer korišćenja virtuelnih funkcija u hijerarhiji klasa Oblik, Kvadrat i Trougao. Klase Oblik i Trougao su deklarisane kao:

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

class Oblik {
public:
 virtual double povrs() { cout<<"povrsina\n"; return
0.0;}
 void setIme (string s) { ime = s; }
 string getIme () { return ime; }
 virtual void setStrane (double s1, double s2,
double a12) {
  strana1 = s1;
  strana2 = s2;
  ugao12 = a12;
 }
 double strana1, strana2, ugao12;
private:
 string ime;
};
class Kvadrat : public Oblik {
public:
 double povrs() { return strana1 * strana1; }
 void setStrane (double s1, double s2, double s3)
{strana1=s1; }
};
```

Definicija i implementacija klasa Trougao, kao i glavni program su dati u nastavku:

```
class Trougao : public Oblik {
public:
 void setStrane (double, double, double);
 double povrs();
private:
 double visina();
 double strana3();
 double ugao13();
 double ugao23();
};
double Trougao::visina(){ // s1 je osnova trougla
 return strana2 * sin(ugao12);
}
double Trougao::povrs(){
 cout << strana1 << " " << strana2 << endl;
 return 0.5 * strana1 * visina();
}
int main(){
 double strana1, strana2, ugao12;
 Oblik *pOblik;
 pOblik = new Trougao;
 pOblik->setStrane(strana1, strana2, ugao12);
 pOblik->setIme("Trougao");
 cout << "povrsina " << pOblik->getIme() << " je "
<<
  pOblik->povrs() << endl;
 pOblik = new Kvadrat;
 pOblik->setStrane(strana1, 0.0, 0.0);
 pOblik->setIme("kvadrat");
 cout << "povrsina " << pOblik->getIme() << " je "
<<
 pOblik->povrs() << endl;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

VIRTUELNE FUNKCIJE - PRIMER3

Kreirati hijerarhiju klasa Animal, Cat i Dog, gde je za svaku od klasa implementirana posebna funkcija koja se odnosi na način kako neke životinje ispuštaju zvuke

Zadatak: Pristupati objektima klasa korišćenjem pokazivača na baznu klasu. Implementirati funkcije korišćenjem principa virtuelnih funkcija; U nastavku je dat primer hijerarhije:

```
#include <string>
class Animal{
protected:
 std::string m_strName;
 Animal(std::string strName)
  : m_strName(strName) {}

public:
 std::string GetName() { return m_strName; }
 virtual const char* Speak() { return "???"; }
};

class Cat: public Animal{
public:
 Cat(std::string strName)
  : Animal(strName) {}
 virtual const char* Speak() { return "Meow"; }
};

class Dog: public Animal {
public:
 Dog(std::string strName)
  : Animal(strName) {}
 virtual const char* Speak() { return "Woof"; }
};
```

Funkciju koja izveštava o podacima objekta odgovarajuće životinje i glavni program možemo napisati kao:

```
void Report(Animal &rAnimal)
{
            cout << rAnimal.GetName() << " says "
                        << rAnimal.Speak() << endl;
}

int main()
{
            Cat cCat("Fred");
            Dog cDog("Garbo");

            Report(cCat);
            Report(cDog);
}
```

Možemo takođe kreirati i niz pokazivača na klasu Animal u glavnom programu na sledeći način:

```
Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");
Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");

Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky,
&cTruffle, &cTyson, &cZeke };
for (int iii=0; iii < 6; iii++)
cout << apcAnimals[iii]->GetName() << " says " <<
            apcAnimals[iii]->Speak() << endl;
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. ČISTO VIRTUELNE FUNKCIJE

Svaka izvedena klasa u kojoj nije implementirana čisto virtuelna funkcija bazne klase takođe postaje apstraktna klasa i ne može da instancira objekte

Pretpostavimo da smo u prethodnom primeru hteli da dodamo klasu Krava koja nasleđuje klasu Životinja, i da smo to uradili na sledeći način:

```
class Cow: public Animal
{
public:
            Cow(std::string strName)
                        : Animal(strName) { }

            // We forgot to redefine Speak
};

int main()
{
            Cow cCow("Betsy");
            cout << cCow.GetName() << " says " <<
                        cCow.Speak() << endl;
}
```

Kao rezultat biće oštampana poruka na ekranu: Betsy says ???. U slučaju da funkciju Speak u klasi Animal definišemo kao čisto virtuelnu, na sledeći način:

```
virtual const char* Speak() = 0; // pure virtual function
```

a zatim ponovo pokrenemo prethodnu fukciju main, kompajler neće moći da prevede program već će prijaviti sledeću kompajlersku grešku:

```
C:\Test.cpp(141) : error C2259: 'Cow' : cannot
instantiate abstract class due to following members:
        C:Test.cpp(128) : see declaration of 'Cow'
```

Razlog je naravno taj što Animal sada postaje apstraktna bazna klasa koja ne može da instancira objekte. S obzirom da u klasi Cow nije implementirana funkcija Speak, klasa Cow takođe postaje apstraktna klasa. Stoga je neophodno implementirati funkciju Speak u klasi Cow na sledeći način:

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Moo"; }
};
```

Program će se sada uspešno iskompajlirati i rezultat će biti:

```
Betsy says Moo
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. ŠABLONI FUNKCIJA

Deklaracija šablona funkcije je ista kao i obična deklaracija funkcije, osim što se ispred stavi specifikacija template<class T> gde se T odnosi na opšti tip podatka

U nastavku je dat primer šablona funkcija koja računa maksimum dva broja:

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b)
{
    return a < b ? b:a;
}
int main ()
{

    int i = 39;
    int j = 20;
    cout << "Max(i, j): " << Max(i, j) << endl;

    double f1 = 13.5;
    double f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl;

    string s1 = "Hello";
    string s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;

   return 0;
}
```

Nakon kompajliranja i izvršavanja programa dobija se sledeći rezultat:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

PRIMER. ŠABLONI KLASE

Cilj primera je da se pokaže implementacija šablona klase za stek različitih tipova podataka

Stek je struktura podataka koja koristi „prvi ulazi, posledni izlazi“ (FILO – first in last out) pristup. Korisno je kreirati šablonsku klasu koja će omogućiti kreiranje steka za različite tipove podataka. U nastavku je dat primer šablona klase i konstruktora:

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

template <class T>
class Stack
{
public:
          Stack();
          void push(T i);
          T pop();
private:
          int top;
          T st[100];
};

template <class T>
Stack<T>::Stack()
{
          top = -1;
}
```

Implementacija funkcija koje ubacuju podatke na stek, odnosno izbacuje sa steka mogu biti implementirane na sledeći način:

```
template <class T>
void Stack<T>::push(T i)
{
          st[++top] = i;
}

template <class T>
T Stack<T>::pop()
{
          return st[top--];
}
```

Glavni program za testiranje ćemo napisati kao:

```
int main ()
{
          Stack<int> int_stack;
          Stack<string> str_stack;
          int_stack.push(10);
          str_stack.push("Hello");
          str_stack.push("World");
          cout << int_stack.pop() << endl;
          cout << str_stack.pop() << endl;
          cout << str_stack.pop() << endl;
          return 0;
}
```


© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Zadaci za samostalan rad

Polimorfizam, apstrakcija, hijerarhija klasa, šabloni

- *Zadaci za samostalno vežbanje*

09

ZADACI ZA SAMOSTALNO VEŽBANJE

Na osnovu materijala za ovu nedelju uraditi samostalno sledeće zadatke:

1. Hierarhiju klasa implementirati u C++-u: Klasa ClanakAbstract sa metodama:getNaziv(), setNaziv(String naziv), getOpis(), setOpis(String opis), getBroj(), setBroj(int broj), getListaClanaka(), setListaClanci(int clanak[]), addClanak(int clanak). Klase Zdravlje, Karijera, Posao i Ljubav koji nasleđuju klasu ClanakAbstract.

2. Napraviti klasu Clanak koja ima od podataka Naziv, Tip, Opis i broj (ceo broj). Tip je vrednost tipa String, a može biti jedan od sledećih stringova: Astrologija, Savet, Zabava,

Zatim napraviti pomoćnu klasu koja ima metodu koja vraća slučajni broj u intervalu od 1 do 1300, metodu slučajni tip koji vraća jednu od vrednosti koju tip može da ima i metodu slučajna reč koja vraća reč od 10 slova, s tim što je prvo slovo veliko, a ostala mala. U U main funkciji kreirati dve instance klase Clanak i svaku od njih napuniti random podacima.

3. Napraviti klasu Heroj koja od podataka ima (brojZivota, snaga, kretanje, koordinataX). Napraviti metodu napad koja će skinuti život drugom heroju (onom koga napadate) i metodu kreni koja pomera x kordinatu za 1. Napraviti klasu SuperNapadac koja je Heroj koji skida jednim udarcem dva života i klasu SuperWalker koja se kreće duplo većom brzinom od običnog Heroja. Koristiti polimorfizam i nasleđivanje.

4. Napraviti klase na osnovu sledećeg teksta. Naša prodavnica se bavi prodajom raznih artikala. O artiklima čuvamo sledeće informacije: Ime artikla i njegovu cenu. Na cenu uvek dodajemo PDV međutim on se razlikuje i nije isti za sve. Za svu robu osim tehnike PDV je 20% dok je za Tehniku PDV 8%. Treba nam program koji će ovo da automatizuje.

5. Napraviti klase na osnovu sledećeg teksta. Naš fakultet ima studente i zaposlene. O Studentima čuvamo sledeće podatake: ime, prezime, adresa, broj telefona I indeks dok o zaposlenima: ime, prezime, adresa, broj telefona kao i idZaposlenog. Napraviti program za vođenje evidencije o studentima i zaposlenima.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.

Zaključak

11

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće:

Polimorfizam znači da će poziv funkcije članice izazvati pozive različitih funkcija u zavisnosti od tipa objekta koji je pozvao funkciju. Pokazivači/reference na bazne klase kojima dodelimo adresu objekta izvedene klase imaju pristup samo funkcijama i podacima bazne klase. Virtuelne funkcije rešavaju problem pokazivača na bazne klase kojima se dodeljuje adresa objekta izvedene klase. Virtual naznačava da će funkcija biti naknadno prilagođena zahtevima objekata izvedenih klasa. U slučaju korišćenja virtuelnih funkcija sa pokazivačima na objekte prenose se i pokazivači na predefinisane funkcije, što je osnov polimorfizma u objektnom programiranju. Opšte je pravilo da kada imamo posla sa nasleđivanjam da destruktore klasa definišemo kao virtuelne, korišćenjem ključne reči virtual u deklaraciji.

Apstrakcija podataka se odnosi na otkrivanje samo osnovnih informacija spoljašnjem svetu (tj. interfejsa), sakrivajući detalje o tome kako je nešto implementirano. Pri kreiranju komponenata vašeg programa, praksa je da se interfejsi razdvoje od implementacije tako da u slučaju izmene implementacije vašeg koda, interfejsi ostanu netaknuti. Apstrakcija podataka obezbeđuje da unutrašnji (sakriveni) članovi klase ostanu zaštićeni od nepažljivog korišćenja koje može da pokvari stanje objekta.

Klase koje sadrže neku čistu virtuelnu funkciju se zovu “apstraktnim” klasama. Apstraktne klase ne mogu biti korišćenje za instanciranje objekata i služe samo kao interfejs. Za razliku od apstraktnih klasa, klase koje mogu biti korišćene za instanciranje objekata se nazivaju konkretne klase.

Šabloni predstavljaju osnov generičkog programiranja, koje uključuje pisanje koda u smislu da je funkcionalnost koda nezavisna od korišćenih tipova podataka. U standardnoj C++ biblioteci postoji ogroman broj funkcija koje su napisane korišćenje šablona. Poziv šablona funkcije se ostvaruje kao i poziv obične funkcije. Pri korišćenju šablona funkcije moraju biti definisani svi operatori i pozivi funkcija za tipove za koje se instancira šablon funkcije. Šabloni klasa se definišu na sličan način kao i šabloni funkcija. Funkcije članice šablona klase su ustvari šablonske funkcije. Šablon klase može imati nekoliko tipskih parametara koji su razdvojeni zarezima. Ovi tipski parametri biće korišćeni i za šablone funkcija koje su članice klase C++ dozvoljava definisanje drugačije implementacije šablona za specijalnu potrebu kada se određeni tip prosledi kao argument šablona.

© UNIVERZITET METROPOLITAN, Beograd / Kopiranje i umnožavanje nije dozvoljeno / Sva prava su zadržana.