C++

Садржај

Програмски језик C++......3
Објектно оријентисано програмирање......3
Наредбе улаза и излаза......6
Namespace Простор имена......9
Класе – дефинисање, декларисање и дијаграми......17
    Дефинисање и декларација класа......17
    Дијаграми класа......18
Креирање објеката на основу класа......19
Методе класа......21
Конструктори класа......24
    Појам и дефинисање конструктора......24
    Када се позива конструктор?......25
    Начини позивања конструктора......25
    Конструктор копије......26
Деструктори класа......29
    Појам и дефинисање деструктора......29
    Када се позива деструктор?......29
Пријатељи класа......32
    Појам и дефинисање пријатеља класа......32
    Пријатељске функције......32
    Пријатељске класе......33
Заједнички елементи класа......35
    Појам и дефинисање заједничких елемената класа......35
    Заједнички подаци чланови......35
    Заједничке функције чланице......36
Преклапање оператора......38
    Појам преклапања оператора......38
    Правила за дефинисање операторских функција......38
    Бочни ефекти и везе између оператора......39
    Дефинисање операторских функција......39
    Унарни и бинарни оператори......40
    Основни стандардни улазно/излазни токови - Класе istream и ostream......41
    Улазно/излазне операције за корисничке типове......41
Изведене класе......43
    Појам изведених класа......43
    Декларација изведених класа......43
    Права приступа члановима класа......44
    Конструктори и деструктори изведених класа......44
    Полиморфизам......47
    Виртуелне функције......47
    Вишеструко наслеђивање......47
    Дијаграми изведених класа......48
Стварање и уништавање објеката изведене класе......49
Изузеци......53
    Пријављивање изузетака......53
    Детекција и обрада изузетака......54

Програмски језик C++

Програмски језици су настали у циљу што лакшег програмирања рачунара за решавање проблема на рачунару. Како рачунар разуме само њему специфичан машински језик, програмски језици су морали да се прилагоде да са једне стране могу лако да се преведу на конкретан машински језик за дати рачунар, а са друге стране да буду лаки за учење и разумевање људима. Синтакса и семантика програмских језика морају бити једнозначни и формални.

Код програмских језика који су постојали раније могли смо разликовати две ствари: податке и инструкције које се извршавају над задатим подацима. Подаци су били унапред дефинисани и углавном су били погодни за превођење на машински језик, док су инструкције биле једноставне. Проблем је настајао када су програми превазишли дужине од неколико стотина редова и када се појавила потреба за решавање компликованих проблема.

Процедурално програмирање, које се сводило на извршавање низа иснструкција над подацима, имало је озбиљне недостатке који су донекле били решени дељењем на логичке целине и потпрограме. Овакав приступ је био разуман, али се појавио проблем са преправљањем кода и његовим одржавањем због високог степена повезаности таквих логичких целина.

Решење овог проблема је било...

Објектно оријентисано програмирање

За разлику од процедуралног програмирања које је више наклоњено превођењу на машински језик, објектно оријентисано програмирање је више окренуто ка реалном описивању проблема који се решава и као такав је блискији људском размишљању.

Све ствари које нас окружују представљају објекте, који имају своје особине и које се током времена мењају по одређеним правилима. Слични објекти се слично и понашају. Различити објекти имају различите особине и стања.

Човек има интеракцију са околним предметима и све што учини може променити стање неког објекта. Приликом те интеракције за човека је битно да зна шта ће се десити ако промени неку одређену особину објекта, док га неке друге особине тог објекта не занимају.

Сви објекти са сродним особинама се могу сврстати у једну класу која их уопштено описује.

Нпр. Све оловке имају заједничку особину да су ваљкастог су облика и остављају трагове на разним предметима приликом употребе. Из тога можемо закључити да је оловка класа. Са друге стране свака оловка којом се неки од ученика служи има облик ваљка, али различитих димензија, на различит начин оставља трагове на предметима по којима пишемо, различите су боје, и друго. Према томе свака од тих оловки је реалан објекат класе оловка.

Из овог примера се може закључити да класа представља одређени тип података, док је било која стварна оловка објекат класе оловка.

Често се занемарују неке особине објеката које нам у датом тренутку нису важне (нпр. Да ли је оловка од пластике, дрвета, метала,...), већ запажамо неке које су нам неопходне (нпр. Да ли пише плаво, црвено,...). Из тога се може закључити да неке особине класе можемо изоставити када правимо њену представу у програму, која ће представљати такав тип података.

Из овога можемо закључити да свака класа представља сложени тип података. Стање

објекта неке класе се назива особина или атрибут или поље, док је понашање метода.

Нпр. Класа оловка се може описати особинама дужина, пречник, боја, материјал, а њено понашање би представљало да ли пише или не пише.

Програмски језици који су објектно оријентисани су: C++, C#, JAVA, Delphi,...

Објектно оријентисано програмирање се базира на следећим принципима:

Апстракција

Занемаривање небитних особина објеката који нису део планираних особина за обраду података.

Нпр. За издавање сведочанства није потребно да се зна да ли ученик тренира неки спорт, већ само његови основни подаци (име, име оца, презиме, датум и општина рођења и оцене у школи).

Учауривање

Подаци унутар класе су сакривени и заштићени од остатка програма.

Нпр. Мобилни који неко користи има много компоненти унутар кућишта, које обичан корисник и не види, нити зна да постоје, јер за корисника је битно да зна са њим да рукује.

Наслеђивање

Свака класа може имати класе из које је изведена (наткласа) и која је из ње изведена (поткласа).

Нпр. Класа мачка је изведена из класе животиња, док је класа тигар изведена из класе мачка.

Полиморфизам

То је способност програма да за различите параметре изврши различита прорачунавања и даје различите резултате.

Нпр. За малопре наведену класу животиња можемо имати особину број ногу, док се у изведеним класама типа мачка, пас, птица,... те особине наслеђују и зависно од поткласе имамо различит број ногу. Тако да ако потражимо податак колико ногу има пас или птица можемо добити различите резултате.

Поновно коришћење кода

Неки делови програмског кода се могу користити више пута без икакве измене. Овај принцип је нарочито примењив у изведеним класама, где се не морају поново писати особине и методе које су искоришћене у наткласама.

Нпр. Ако је у класи животиње дефинисана особина број ногу или метода како се животиње оглашавају, није потребно то урадити у свакој класи изведеној из класе животиња.

<u>Основне разлике програмских језика C и C++ у процедуралном погледу</u>

Програмски језик C++ није потпуно објектно оријентисани програмски језик, већ се може користити и као процедурални, али тиме много губимо и мало добијамо.

Поменимо најпре основе језика C++.

95% језика C је остало нетакнуто у језику C++, али исто татко постоје и одређене разлике.

Коментари

Осим коментара /* */ овде се може користити и коментари у једном реду који почињу са //.

Овакви коментари су лакши за писање на крају реда, јер не морамо водити рачуна о крају реда.

Логички подаци

Додат је још један основни тип података: bool који има две могуће вредности: true и false.

```
#include <iostream>
using namespace std;
int main()                  //glavni program
{
//logicke promenjive tipa bool
bool a = true;
cout << a <<" a ako nije tacno onda ima vrednost "<< !a << endl;
// upotrebna vrednost je velika jer mozemo pisati sledece:
if(a)
     cout << "tacno" << endl;;
else
     cout << "netacno" << endl;;
cout << endl;
return 0;
}                           //kraj glavnog programa
```

Унети број x и одредити да ли је он једнак нули.

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

int main()                  //glavni program
{
//logicke promenjive tipa bool
int x;
cin >> x;
if(x) //ako unesemo bilo koju vrednost koja nije nula imace vrednost 1 koja se
vezuje za true
     cout << "nije nula" << endl;
else
     cout << "nula" << endl;;

cout << endl;
return 0;
}
```

Симболичке константе

Ако се испред промењиве дода идентификатор const тада се такве промењиве могу користити и на местима где се искључиво захтева константа.

Нпр. Ако је потребно дефинисати низ:

const int n = 100;

int a[n];

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

int main()                  //glavni program
```

```
{
// ako upotrebimo const ne mozemo menjati vrednost promenjive
const float pi = 3.14;

// nakon ovoga ce prijaviti gresku pri kompajliranju
// jer je sada pi constanta i ne sme se menjati !!!
pi = 12;

cout << endl;

return 0;            //ne brisi
}                          //kraj glavnog programa
```

Дефинисање података и њихов домет

декларација промењивих у програмском језику C++ се може извршити било где у коду, па чак и у наредбама где се користе.

Нпр. for(int i = 0; i < 100; i++) { s = s + i;}

Наредбе улаза и излаза

У програмском језику C користи се наредба printf за приказ података, а да би се ова наредба искористила мора да се користи и стандардна библиотека stdio.h. У програмском језику C++ се као наредба за приказ података користи:

cout

ова наредба има следећу синтаку:

cout << "коментар" << промењива<< endl;

Оператор << има улогу оператора излаза само ако се са његове леве стране налази објекат cout.

Коментари у излазној наредби се и даље пишу унутар наводника као и у програмском језику Ц, с тим што се не наводи тип конверзије.

Пре назива промењиве која се штампа поново се наводи оператор <<. Овај оператор се користи и између више промењивих. (нпр. уколико имамо три промењиве између сваке од њих наводимо овај оператор).

endl је специјални оператор који служи за прелазак у нови ред. Наравно, може да се користе и стари ескејп (escape) знаци: \n или \t.

За примену ове наредбе морамо да користимо стандардно заглавље <iostream>.

Пример за наредбу cout:

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

int main()
{
```

```
      cout << " Dobar dan!" << endl;
          return 0;
  }
```

Написати програм којим ће сваки корисник унети своје име, па одштампати поздравну поруку.

```
  #include <iostream>
  #include <string> // ova biblioteka se koristi uvek kad koristimo stringove u
programu
  using name space std;
  int main ()
  {
  string ime;
  cout << "upisite ime korisnika" << endl;
  cin >> ime;
  cout << "Zdravo " << ime <<  "... i dovidjenja!!!" ; << endl;
  return 0;
  }
```

У програмком језику C користила се наредба за унос података scanf. У програмском језику C++ ова наредба замењена је наредбом

cin.

Синтакса ове наредбе гласи:

cin >> променљива >> променљива;

Оба оператора >> и << могу да се лакше схвате као стрелице које показују правац улаза и излаза. И да би имали ту улогу обавезно са леве стране мора да им стоји cin или cout.

Пример: Написати програм којим ће се сабрати два броја и приказати резултат.

```
  #include <iostream>
  using namespace std;
  int main ()
  {
  int a, b;
  cout << "unesi dva broja" << endl;
  cin >> a >> b;
  int z=a+b;
  cout <<  "zbir je" << z << endl;
  return 0;
  }
```

Пример за наредбе уноса и приказа података

```
  #include <cstdlib> // mogu da se koriste i standardne biblioteke kao u C-u,
ali se dodaje slovo C
                  //ispred imena i nema nastavka .h, kao u prethodnom
programskom jeziku.
  #include <iostream> // standardno zaglavlje za naredbe ulaza i izlaza i
operatore >> i <<
  using namespace std; // u ovaj prostor se stavljaju sva standardna zaglavlja
```

```
koja sadrze standardne
                      // bibliotecke f-je, globalne identifikatore kako bi mogla
da se koriste imena
                     // iz tog prostora.

 int main()                  //glavni program
 {
 int a, b;
 char c;

 // unos se vrsi naredbom cin i navodjenjem operatora >>
 // prikaz se vrsi naredbom cout i navodjenjem operatora <<
 // niko ne brani da i dalje koristite scanf i printf !!!

 // i dalje vaze ista pravila za  escape characters:
 // za prelaz u novi red \n, a sada mozete koristiti i endl
 // za tabulaciju \t
 // za navode \"
 // za kosu crtu \\
 // itd.

 // primeri:
 cin >> a;
 cin >> b;
 cin >> c;
 cout << a << "\n";
 cout << a << " ovo je neki text " << b << endl;
 cout << "znak koji si uneo za c je " << c << endl;

 cout << endl;        //prazan red
 return 0;             //ne brisi
 }                           //kraj glavnog programa
```

Namespace Простор имена

namespace је "простор" имена којима именујемо све променљиве, функције, класе у својим програмима. Сврха постојања ових одвојених "простора" је да не дође до колизије уколико програм постане сувише велики, а инспирација за именовање нових променљивих затаји.

"using" говори компајлеру, да ти користиш онај "простор", који наведеш у namespace линији.
Стандардна библиотека користи "std" namespace, тако да кад год се користи стандардна библиотека, треба написати нешто типа:

```
std::cout<<"Zdravo svete"
```

Да се "std::" не би понављало стално онда се користи "using namespace std".

Дакле, ова линија нам омогућава да користимо стандардне библиотеке, а да при томе не дође до сукоба имена. Сукоб имена настаје онда када се две различите ствари именују истим именом, па програм не може да направи разлику између њих и просто не зна шта да изврши.

Наравно, програм C++ омогућава кориснику да направи свој namespace у оквиру неке класе.

Примери за вежбу:

```
#include <iostream>
using namespace std; // standardni prostor koji se koristi
namespace prostor1
{
  int x = 1;
  int y = 2;
}
 namespace prostor2
{
  float x = 4.16;
  float y = 7.16;
}
int main ()
{
// posto imamo iste promenjive u oba prostora, ali su one razlicitog tipa
moramo
// navesti i prostor iz koga uzimamo promenjivu
cout << "promenjiva x iz prostora1 je " << prostor1::x << endl;
cout << "promenjiva y iz prostora1 je " << prostor1::y << endl;
cout << "promenjiva x iz prostora2 je " << prostor2::x << endl;
cout << "promenjiva y iz prostora2 je " << prostor2::y << endl;
// a ako moramo vise puta da koristimo neku promenjivu iz nekog prostora
radimo to ovako:
using prostor1::x;
using prostor2::y;
cout << x << endl;
cout << y << endl;

cout << endl;
return 0;             //ne brisi
}                           //kraj glavnog programa
```

Дефинисати два нова простора у којима ће се наћи променљиве различитог типа. Сабрати их, одузети, помножити и поделити, па приказати резултате.

```
#include <iostream>
using namespace std; // standardni prostor koji se koristi
namespace prostor1
{
  int x;
  int y;
 }
namespace prostor2
{
  float x;
  float y;
}
int main ()
{
using prostor1::x;
using prostor2::y;

cout << "unesi dva broja " << endl;
```

```
cin >> x >> y;
float z=x+y;
float r=x-y;
float p=x*y;
float k=x/y;

cout << "zbir je " << z << endl;
cout << "razlika je " << r << endl;
cout << "proizvod je " << p << endl;
cout << "kolicnik je " << k << endl;

return 0;
}
```

Кад су у питању остале наредбе које се користе у програмском језику C++ (наредбе услова, циклуса...) оне су исте као у програмском језику C.

Домет промењивих је у програмском језику C++ различит. Постоји могућност да се једно име променљиве користи на више места. Та променљива може бити ограничена на, рецимо, само циклус, а да се у наставку програма користи као друга променљива, чак и различитог типа. Пример:

```
#include <iostream>
using namespace std;
int main()                  //glavni program
{
// definisemo i kao float
float  i = 1.10;
i = i + 1.2;
cout << i << endl;

// sada ga definisemo u brojacu i nema nikakvih problema sa tim
// jer on vazi samo za for petlju !!!
for(int i = 1;i<=10;i++)
{
    cout << i << endl;
}

// Ako nastavimo da koristimo i on ce imati vrednost kao pre brojaca
i = i + 1.2;
cout << i << endl;

cout << endl;
return 0;            //ne brisi
}                           //kraj glavnog programa
```

У програмском језику C++ се често користе и упућивачи. Упућивачи се разликују од показивача. Они:

1. не заузимају меморију,
2. не узимају никакву вредност и
3. показују на променљиву само по референци. Улога референце је да чува адресу неке друге променљиве и све операције које се чине на њима, прослеђују променљивама на које саме показују. Третирају се баш као посебна подврста података, попут показивача и морају се експлицитно захтевати посебном синтаксом (npr. C++, PHP).У програмском језику C++, референце су уведене ради поједностављеног писања како би у одређеним аспектима преузеле улогу показивача али биле и знатно једноставније за употребу.

Битно је нагласити да показивачи и референце нису еквивалентни у C++-у и да, штавише, постоје значајне разлике:
1. показивачи могу примити вредност у било ком тренутку за време рада програма
 - референце морају бити иницијализовани при конструисању, и не могу мењати вредност
2. показивачи могу садржати вредност NULL, тј. не показивати нинашта
 - референце морају бити иницијализоване на неку вредност, или ће компајлер пријавити грешку
3. показивачи се могу користити за имплементацију низова, матрица, динамичку алокацију меморије и прослеђивање аргумената функције „по референци“
 - референце се могу користити само за прослеђивање аргумената функције по референци
4. показивачи користе оператор * (звездицу) за дереференцирање и оператор & (амперсанд) за референцирање
 - референце не користе ниједан од наведених оператора, што их чини знатно једноставнијим за рад
5. могу постојати показивачи на показиваче
 - не могу постојати показивачи на референце

Упућивачи се дефинишу стандардним наредбама, с тим што се испред имена додаје знак & (аперсенд).

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

int main()                  //glavni program
{

int i=1;    //celobrojni podatak i
int &j=i;   //j upucuje na i
i=3;        //menja se i
cout << "i=" << i << endl;
j=5;        //opet se menja i
cout << "i=" << i << endl;
int *p=&j; //isto sto i &i
j+=1;       //isto kao i+=1
cout << "i=" << i << endl;
int k=j;    //posredan pristup do i preko reference
cout << "k=" << k << endl;
int m=*p;  //posredan pristup do i preko pokazivaca
cout << "m=" << m << endl;

cout << endl;
return 0;             //ne brisi
}                           //kraj glavnog programa
```

Пример:

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

// U narednoj f-ji bi u programskom jeziku C morali da koristimo
```

```
// pokazivace za bocni efekat, ali ako koristimo upucivace tada se
// i prenosi po vrednosti,a j po referenci
//
void f(int i, int &j)
{
i++;  //stvarni argument se nece promeniti
j++;  //stvarni argument ce se promeniti
}

int main ()
{
// definisemo dve promenjive
int si=0,sj=0;

cout << "pre poziva funkcije" << "\n";
cout<<"si="<<si<<", sj="<<sj<<"\n";

// pozivamo f-ju f
f(si,sj);

//si se prenosi po vrednosti, sj po referenci
// Izlaz ce biti:
// si=0, sj=1
cout << "nakon poziva funkcije" << "\n";
cout<<"si="<<si<<", sj="<<sj<<"\n";

cout << endl;

return 0;             //ne brisi
}                           //kraj glavnog programa
```

Статички типови података се чувају у DATA SEGMENT-у. Њихов број (тј. меморија коју ће заузети) се декларише на почетку програма. Дакле, њихов број мора бити унапред познат.

Програмер мора често да одвоји више меморијских локација за податке него што је заиста потребно, јер не зна шта ће корисник тражити од програма. Тако да програм често користи и десетоструко дуже низове и структуре него што ће корисник икад искористити у истом. То је главна мана код оваквог писања програма.

Динамичке промењиве се разликују од статичких. Код ових типова података се не мора унапред знати колико ће промењивих бити потребно програму за рад. Меморијски простор за промењиве резервише се и ослобађа током извршења програма.

За потребе додела меморије за динамичке промењиве користе се оператор **new**.

Величина меморијског простора додељеног помоћу оператора new аутоматски се одређује на основу величине стварног податка.

Вредност оператора new је показивач на податак који се ствара.

Овим оператором могуће је иницијализовати податке.

Овим оператором могуће је стварати низове података.

Синтакса:

```
new име_типа
new име_типа (почетна вредност)
new име_типа [дужина]
```

Име_типа је идентификатор податка за који се врши додела меморије. Може да буде и показивач уколико се врши додела меморије показивачима.

Почетна вредност може да буде било која, али мора да одговара типу податка коме се додељује.

Дужина се односи на дужину низа и има онолико димензија као што их има било који низ. Дакле ако је у питању вектор, онда је то једна димензија, за матрице две...

За ослобађање меморије користи се оператор **delete**. Синтакса је:

```
delete адреса
delete [ ] адреса.
```

Адреса је неки израз који показује на податак који заузима динамичку меморију.

Немогуће је ослобађати простор додељен само неком елементу низа, те је потребно показати почетак низа. Само уколико ослобађамо меморију додељену низу користимо угласте заграде. У осталим случајевима оне нам нису потребне. Број елемената низа није потребно навести.

НАПОМЕНА: Приликом динамичке доделе меморије није могуће иницијализовати низ, већ елементи низа узимају неку случајну вредност па је потребно унети вредност за сваки члан низа.

Често се дешава да се иста радња примељује на различите податке. (Податке који су разчиитог типа). Уколико користимо функцију за решавање овог проблема потребно је писати различите функције. Оно што у програмском језику C није било могуће да те функције имају исто име. Међутим, програмски језик C++ нам омогућава да користимо исто име за различите функције. Оно о чему мора да се води рачуна је да те функције са истим именом морају да имају ИЛИ РАЗЛИЧИТИ ТИП ПОДАТАКА или РАЗЛИЧИТИ БРОЈ ПАРАМЕТРА, а самим тим и различити број аргумената који позивају ту функцију.

Уколико је ТИП ПАРАМЕТРА исти, онда БРОЈ ПАРАМЕТРА мора да буде различит. Или обрнуто. Само тако ће преводилац знати коју функцију треба да позове.

Пример: Израчунати корен унетог броја. Користићемо три функције са истим именом, али са различитим типом података.

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

double koren(int x)
{
  double z = x;
  return sqrt(z);
}
double koren(float x)
{
  double z = x;
  return sqrt(z);
}
double koren(double x)
{
```

```
  double z = x;
  return sqrt(z);
}

int main()                  //glavni program
{
// definisemo promenjive razlicitog tipa
int a;
float b;
double c;

cout << "unesi ceo broj veci od 0 ";
cin >> a;
cout << "koren broja " << a << " je " << koren(a) << endl;
cout << "unesi realan broj veci od 0 ";
cin >> b;
cout << "koren broja " << b << " je " << koren(b) << endl;
cout << "unesi realan broj veci od 0 ";
cin >> c;
cout << "koren broja " << c << " je " << koren(c) << endl;

cout << endl;
system("PAUSE");      //ne brisi (mora da stoji ispred return)
return 0;
}
```

Пример: Израчунати запремину за коцку, квадар или ваљак. (Дати кориснику могућност да бира за коју од ових фугура ће рачунати запремину – користи switch-case наредбу).

Овог пута користићемо функције истог имена и истог типа параметра, али са различитим бројем параметара, а самим тим и у програму морамо да позовемо функције са различитим бројем аргумената.

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

const float pi = 3.14;

// za kocku
double p(float a)
{
  return a*a*a;
}

// za kvadar
double p(float x, float y, float z)
{
  return z*y*x;
}

// za valjak
double p(float r, float h)
{
  return r*r*pi*h;
}

int main()
{
```

```
float a,b,c;
int n;

cout << "za koju figuru racunam zapreminu?" << endl;
cout << "1 - kocka" << endl;
cout << "2 - kvadar" << endl;
cout << "3 - valjak" << endl;

cin >> n;

switch(n)
{
   case 1:
   {
      cout << "unesi stranicu kocke" << endl;
      cin >> a;
      cout << "zapremina kocke je " << p(a); //pozivamo f-ju sa jednim
//argumentom
   }; break;
   case 2:
   {
      cout << "unesi sve tri stranice kvadra" << endl;
      cin >> a >> b >> c;
      cout << "zapremina kvadra je " << p(a,b,c); // pozivamo f-ju koja ima
//tri parametra pa navodimo tri agrumenta.
   } ; break;
   case 3:
   {
      cout << "unesi poluprecnik i visinu valjka" << endl;
      cin >> a >> b;
      cout << "zapremina valjka je " << p(a,b); // pozivamo f-ju koristeci dva
//argumenta
   }; break;
   default: cout << "pogresan izbor";
}

cout << endl;
return 0;
}
```

Класе – дефинисање, декларисање и дијаграми

Програмски језик C++ омогућује креирање нових типова података, који су сложене структуре и називају се класе.

Класа је сложени тип података који се састоји од елемената различитих типова и представља скуп објеката са истим особинама и начином понашања.

Када креирамо податак типа одређене класе он се назива објекат. Сваки од објеката има своје вредности и са њима се може манипулисати као целином или са њиховим појединачним особинама и методама.

Зашто се користе класе када већ имамо структуре?

Зато што класа дефинише како се неки податак (објекат) креира, како се њиме рукује и шта се сме, а шта се не сме учинити, како се објекат уништава.

Дефинисање и декларација класа

Под овим се подразумева навођење свих елемената који дефинишу објекте. Елементи класе који описују објекат називају се особине, а елементи који указују које радње можемо вршити

са објектима називају се методе.

Приликом декларације морамо навести кључну реч **`class`**, а затим име класе.

```
class име_класе;
```

Овако дефинисана класа нема много сврхе, јер нам не показује које особине и медоте садрже објекти те класе. Поред тога и компајлер нема представу колико меморије треба резервисати за објекат овакве класе.

Идеално је да се наведу све особине и методе класе приликом декларисања.

Свака особина и метода у оквиру класе може бити:
- приватна (*private*)
- јавна (*public*)
- заштићена (*protected*)

Приватни делови класе су заштићени од спољашњих утицаја и може им се приступати само методама (функцијама) које се налазе у декларисаној класи.

Заштићени подаци су они подаци којима се може приступити само методама из те класе или неке изведене класе, али не и споља

Јавни подаци су они који су доступни како унутар класе, тако и ван ње.

Пример: Креираћемо класу датум који ће имати особине дан, месец и година који ће бити приватни и методе којима можемо да мењамо те податке.

```
class Datum              // резервисана реч цласс и име морају бити на почетку
{                        // сви елементи класе морају бити у оквиру ових заграда
  private:               // овде наводимо приватне особине и методе класе
    int dan;             // приватни подаци
    int mesec;
    int godina;
  public:                // овде наводимо јавне особине и методе класе
    void Dan(int d);     // јавне методе
    void Mesec(int m);
    void Godina(int g);
};                       // крај класе
```

Овако декларисана класа датум нема заштићене елементе, али ако би извели из ове класе класу време у том случају би могли имати и заштићене елементе.

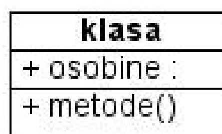
Декларација класе каже компајлеру колико је простора потребно за један објекат ове класе и тај меморијски простор се заузима тек након дефинисања објеката дате класе.

Стварање објеката дате класе се може урадити као и за било који други тип података.

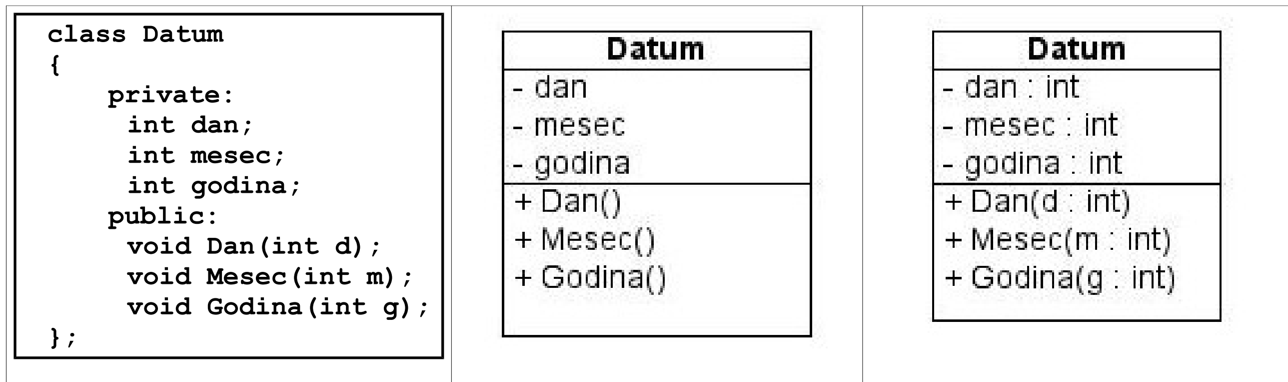
Дијаграми класа

Неке ствари је много лакше схватити ако се нацртају. Из тог разлога је настао и језик за моделовање *UML (unified modeling language)*. Како он није везан искључиво за C++ усвојено је правило да се за све објектно-оријентисане програмске језике користе дијаграми класа.

Графичко представљање класа се врши на следећи начин: најпре се наводи име класе, затим њене особине и на крају методе. Наравно због различитих потреба могуће је и мењати количину приказаних информација.



Пример: за пример ћемо узети класу датум коју смо већ дефинисали.



Из овог примера се види да можемо веома лако визуално да прикажемо сваку класу. UML нам омогућава и да графички прикажемо зависности између различитих класа, као и још многе друге ствари, али је можда најважнија она да можемо да генеришемо програмски код из оваквог представљања података.

Питања:

1. Шта су класе?
2. Шта су објекти?
3. Какви могу бити елементи класа?
4. Каква је разлика између јавних, заштићених и приватних елемената?
5. Како се декларише класа?
6. Навести један пример за класу?
7. Како графички представљамо класу?

Креирање објеката на основу класа

Када креирамо податак типа одређене класе он се назива објекат. Сваки од објеката има своје вредности и са њима се може манипулисати као целином или са њиховим појединачним особинама и методама.

Класа дефинише како се неки објекат понаша: шта се са њим сме, а шта се не сме радити.

Декларација класе каже компајлеру колико је простора потребно за један објекат ове класе и тај меморијски простор се заузима тек након дефинисања објеката дате класе.

Стварање објеката дате класе се може урадити као и за било који други тип података.

За пример ћемо узети класу датум из претходног примера:

```
class Datum
{
  private:
    int dan;
    int mesec;
    int godina;
  public:
    void Dan(int d);
    void Mesec(int m);
    void Godina(int g);
};
```

Креирање објекта се реализује тако што најпре наведемо тип (класу) објекта, а затим име које желимо да он носи:

```
Datum datum1;
```

Овако креиран објекат има своје особине и методе, које стварање другог објекта исте класе више не може да промени.

Исто тако можемо креирати и низ објеката типа класе **Datum**:

```
Datum d[365];
```

Наравно код објеката је битно да схватимо да се резервише простор само за особине, док методе заузимају меморијски простор само када се позову и након завршетка (наредбе return) та меморија се ослобађа.

Правила за показиваче важе и случају објеката насталих из класа. Под тим се подразумева да се примењују иста правила показивачке аритметике као и код осталих типова података.

```
Datum *pd = d;
```

Такође је могуће креирати и референце (упућиваче) на креиране објекте:

```
Datum d, &r = d;
Datum *p = r;
```

Поред статичких објеката, могуће је креирати и динамичке објекте који се стварају у току извршења нашег програма. Овде је најбоље користити new и delete, јер код њих нема потребе да бринемо о величини објеката (колико меморије заузимају).

```
Datum *pd;
pd = new Datum;
```

Објекти дате класе могу, као и сви остали подаци, бити аргументи функција:

```
int broj_dana_od_pocetka_godine(Datum n);
```

или тип функција:

```
Datum praznik(Datum n);
```

Пример: Креирати класу ауто са особинама: марка, боја, гориво, и одговарајуће методе којима ће се мењати те особине, а затим креирати један објекат и један динамички низ објеката дате класе.

```
class auto
{
  private:
    char *marka, *boja, *gorivo;
  public:
    void Marka(char *m);
    void Boja(char *b);
    void Gorivo(char *g);
}
```

У главном програму треба дефинисати следеће објекте:

```
auto mojAuto;
auto *VozniPark = new auto [20];
delete [] VozniPark;
```

Пример: Креирати класу зграда са особинама: број спратова, боја, лифт, број улаза, и одговарајуће методе којима ће се мењати те особине, а затим један објекат и низ објеката и по један показивач и упућивач на објекат дате класе.

```
class zgrada
{
  private:
    char *boja, *lift;
    int broj_spratova, broj_ulaza
  public:
    void Boja(char *b);
    void Lift(char *l);
    int BrojSpratova(int bs);
    int BrojUlaza(int bu);
}
```

У главном програму треба дефинисати следеће објекте:

```
zgrada IvanovaZgrada, *pz, &iz = IvanovaZgrada;
zgrada *blok = new zgrada [10];
delete [] blok;
```

Питања:

1. Шта су објекти?
2. Како се дефинишу објекти дате класе?
3. Да ли декларација класе заузима меморијски простор?
4. Каква је разлика између статичких и динамичких објеката?
5. Ког типа морају бити показивачи?
6. Ког типа су упућивачи (референце)?
7. Да ли објекти могу бити параметри функција?

Методе класа

Методе класа су у ствари функције које омогућују кориснику да приступа приватним и заштићеним подацима унутар класе, односно објеката.

Свака метода може да чита податак (односно не мења вредности) и тада се она назива инспектро или селектор или да мења податак и тада се она назива мутатор или модификатор.

Добра програмерска пракса је да се корисницима класе специфицује да ли нека функција чланица мења унутрашње стање објекта или га само "чита" и враћа информацију кориснику класе.

Да је функција чланица инспектор, кориснику класе говори реч const иза заглавља функције. Овакве функције чланице називају се у језику C++ константним функцијама чланицама (енгл. *constant member functions*).

Функција чланица која мења стање објекта посебно се не означава.

Дефиниција методе би онда изгледала овако:

- за методе које не мењају стање објекта:

```
tip ime_metode (argumenti) const;
```

или

```
tip ime_metode (argumenti) const {telo funkcije};
```

- за методе које мењају стање објеката:

```
tip ime_metode (argumenti);
```

или

```
tip ime_metode (argumenti) {telo funkcije};
```

Уколико се тело функције наведе у оквиру класе тада нема никаквих проблема са приступањем осталим елементима те класе, али проблем настаје ако се функција дефинише ван класе и тада је потребно навести и којој класи припада.

Пример: Направити класу датум која има особине дан, месец и година, као и методе за читање и постављање датума.

Прво ћемо обрадити пример када се тело методе налази у оквиру класе:

```
class Datum
{
  private:
    int dan;
    int mesec;
    int godina;
  public:
    void postaviDan(int d) {dan = d;}
    void postaviMesec(int m) {mesec = m;}
    void postaviGodinu(int g) {godina = g;}
    void prikaziDatum()const {cout << dan << "." << mesec << "." << godina;}
};
```

Други начин је да тело функције буде дефинисано ван класе, а да се у самој класи налази само прототип функције:

```
class Datum
{
  private:
    int dan;
    int mesec;
    int godina;
  public:
    void postaviDan(int d);
    void postaviMesec(int m);
    void postaviGodinu(int g);
    void prikaziDatum() const;
};
void Datum::postaviDan(int d) {dan = d;}
void Datum::postaviMesec(int m) {mesec = m;}
void Datum::postaviGodinu(int g) {godina = g;}
void Datum::prikaziDatum() const {cout <<dan<<"."<<mesec<<"."<< godina;}
```

Оператор '::' овде означава да та функција припада класи наведеној испред оператора.

Пример: Направити класу чоколада која има особине марка, цена и боја, као и методе за читање и постављање датих особина и методу која може обрачунати и ПДВ. Урадити за случај када су методе дефинисане у оквиру класе и ван ње.

```
class cokolada
{
    private:
     char* marka;
     char* boja;
     float cena;
    public:
     void postaviMarku(char* m) {marka = m;}
     void postaviBoju(char* b) {boja = b;}
     void postaviCenu(float c) {cena = c;}
     void prikaziMarku() const {cout << marka; cout << endl;}
     void prikaziBoju() const {cout << boja; cout << endl;}
     void prikaziCenu() const {cout << cena; cout << endl;}
     float PDV() {float pdv = 1.18; pdv = pdv * cena; return pdv;}
};

class cokolada
{
    private:
     char* marka;
     char* boja;
     float cena;
    public:
     void postaviMarku(char* m);
     void postaviBoju(char* b);
     void postaviCenu(float c);
     void prikaziMarku() const;
     void prikaziBoju() const;
     void prikaziCenu() const;
     float PDV();
};
void cokolada::postaviMarku(char* m) {marka = m;}
void cokolada::postaviBoju(char* b) {boja = b;}
void cokolada::postaviCenu(float c) {cena = c;}
void cokolada::prikaziMarku() const {cout << marka; cout << endl;}
void cokolada::prikaziBoju() const {cout << boja; cout << endl;}
void cokolada::prikaziCenu() const {cout << cena; cout << endl;}
float cokolada::PDV() {float pdv = 1.18; pdv = pdv * cena; return pdv;}
```

Када у некој класи дефинишемо све особине, поред њих постоји још једна која се аутоматски генерише. То је показивач `*this` који враћа тренутну меморијску адресу објекта.

Приступ се тада врши помоћу овог показивача и оператора `->`.

Овај показивач се најчешће користи приликом повезивања (успостављања релације између) два објекта. На пример, нека класа X садржи објекат класе Y, при чему објекат класе Y треба да "зна" ко га садржи (ко му је "надређени").

Пример: У претходном примеру за чоколаду могли смо написати функцију која рачуна ПДВ на следећи начин:

```
float cokolada::PDV() {float pdv = 1.18; pdv = pdv * this->cena; return pdv;}
```

Овај показивач ће касније бити боље објашњен.

Питања:

1. Шта су методи?
2. Како се дефинишу методи дате класе?
3. Да ли декларација метода заузима меморијски простор?
4. Каква је разлика између метода који мењају стање објеката и оних који то не раде?
5. Како се дефинишу методе ван тела класе?

Конструктори класа

Појам и дефинисање конструктора

Функција чланица која носи исто име као и класа назива се конструктор (енгл. *Constructor*).
Ова функција позива се приликом креирања објекта те класе.
Конструктор нема тип који враћа и може да има аргументе произвољног типа.
Унутар конструктора, члановима објекта приступа се као и у било којој другој функцији чланици.
Конструктор се увек имплицитно позива при креирању објекта класе, односно на почетку животног века сваког објекта дате класе.
Конструктор, као и свака функција чланица, може бити преклопљен (енгл. *Overloaded*).
Конструктор који се може позвати без стварних аргумената (нема формалне аргументе или има све аргументе са подразумеваним вредностима) назива се подразумеваним конструктором.
Дефинисање конструктора је слично као и за друге методе, али нема тип који враћа и мора имати исто име као и класа:

```
ime_klase (argumenti);
ime_klase (argumenti)  {telo konstruktora}
ime_klase (argumenti):inicijalizator1,...;
ime_klase (argumenti):inicijalizator1,... {telo konstruktora}
```

Иницијализатори су начин да се приватним подацима класе задају параметарске вредности.

Пример: Посматраћемо класу датум коју смо дефинисали раније и дефинисати неколико конструктора за дату класу.

```
class Datum
{
 private:
 int dan;
 int mesec;
 int godina;
 public:
 void postaviDan(int d) {dan = d;}
 void postaviMesec(int m) {mesec = m;}
 void postaviGodinu(int g){godina = g;}
 void prikaziDatum() const{cout <<dan<<"."<<mesec<<"."<<godina<<endl;}
// подразумевани конструктор
 Datum()
 {
  dan = 0;
  mesec = 0;
  godina = 0;
 }
// конструктор који поставља све особине
 Datum(int d, int m, int g)
 {
  dan = d;
  mesec = m;
  godina = g;
 }
```

```
  /* подразумевани конструктор са иницијализаторима је под коментаром
   јер у свакој класи сме постојати само један подразумевани конструктор */
  //  Datum():dan(1),mesec(10),godina(2000){}
  // конструктор са иницијализаторима
  Datum(int i):dan(i),mesec(i),godina(i){}
  // пример конструктора који можемо направити
  Datum(int i, int j)
  {
   dan = 2*i;
   mesec = j;
   godina = 100*i;
  }
 };
```

НАПОМЕНА: Други начин је да тело конструктора буде дефинисано ван класе и тада се мора користити оператор '::'.

Када се позива конструктор?

Конструктор је функција која претвара "пресне" меморијске локације које је систем одвојио за нови објекат (и све његове податке чланове) у "прави" објекат који има своје чланове и који може да прима поруке, односно има сва својства своје класе и конзистентно почетно стање.

Пре него што се позове конструктор, објекат је у тренутку дефинисања само "гомила празних бита" у меморији рачунара. Конструктор има задатак да од ових бита направи објекат тако што ће иницијализовати чланове.

Конструктор се позива увек када се креира објекат класе, а то је у следећим случајевима:

1. када се извршава дефиниција статичког објекта;
2. када се извршава дефиниција аутоматског (локалног нестатичког) објекта унутар блока: формални аргументи се, при позиву функције, креирају као локални аутоматски објекти;
3. када се креира објекат, позивају се конструктори његових података чланова;
4. када се креира динамички објекат оператором *new*;
5. када се креира привремени објекат, при повратку из функције, који се иницијализује враћеном вредношћу функције.

Начини позивања конструктора

Конструктор се позива када се креира објекат класе. На том месту је могуће навести иницијализаторе, тј. стварне аргументе конструктора.

Позива се онај конструктор који се најбоље слаже по броју и типовима аргумената (правила су иста као и код преклапања функција):

Када се креира низ објеката неке класе, позива се подразумевани конструктор за сваку компоненту низа понаособ, по растућем редоследу индекса.

Пример: Користићемо класу датум коју смо већ дефинисали.

```
#include <cstdlib>
#include <iostream>
#include "datum.h"
using namespace std;

int main()
{
// позивамо подразумевани конструктор
Datum d1;
d1.prikaziDatum();
// позивамо конструктор који поставља све особине
Datum d2(1,2,2000);
```

```
d2.prikaziDatum();
// позивамо конструктор са иницијализаторима
Datum d3(3);
d3.prikaziDatum();
// позивамо наш конструктор који смо направити
Datum d4(2,3);
d4.prikaziDatum();
// креирамо статички низ објеката типа Datum
Datum D[3];
for(int i = 0; i < 3; i++)
D[i].prikaziDatum();
// креирамо динамички низ објеката типа Datum
Datum *X = new Datum[2];
for(int i = 0; i < 2; i++)
X[i].prikaziDatum();
cout << endl;
system("PAUSE");
return 0;
}
```

Конструктор копије

Када се објекат x1 класе XX иницијализује другим објектом x2 исте класе, C++ ће подразумевано (уграђено) извршити просту иницијализацију редом чланова објекта x1 члановима објекта x2.

То понекад није добро (често ако објекти садрже чланове који су показивачи или референце), па програмер треба да има потпуну контролу над иницијализацијом објекта другим објектом исте класе.

За ову сврху служи тзв. конструктор копије (енгл. *copy constructor*).

То је конструктор класе XX који се може позвати са само једним стварним аргументом типа XX. Тај конструктор се позива када се објекат иницијализује објектом исте класе, а то је:

1. приликом иницијализације објекта (помоћу знака = или са заградама);
2. приликом преноса аргумената у функцију (креира се локални аутоматски објекат);
3. приликом враћања вредности из функције (креира се привремени објекат).

Конструктор копије никад не сме имати формални аргумент типа XX, а може аргумент типа XX& или најчешће const XX&.

Пример: креирамо класу XX и конструктор копије

```
class XX
{
public:
XX (int);
XX (const XX&);     // конструктор копије
...
};

XX f(XX x1)
{
XX x2=x1;           // позива се конструктор копије XX(XX&) за x2
//...
return x2;           // позива се конструктор копије за привремени
                     // објекат у који се смешта резултат
}

void g()
```

```
{
XX xa=3, xb=1;
...
xa=f(xb);              // позива се конструктор копије само за формални
                       // аргумент x1, а у xa се само преписује
                       // привремени објекат,или се позива
                       // XX::operator ако је дефинисан
}
```

Пример: Направити класу чоколада која има особине марка, цена и боја, као и методу за приказ датих особина и конструкторе.

```
class cokolada
{
private:
 char* marka;
 char* boja;
 float cena;
public:
 void prikaziCokoladu() const;
 cokolada();                                   // подразумевани конструктор
 cokolada(char* m ,char* b,float c);           // конструктор за чоколаду
};

cokolada::cokolada()                           // подразумевани конструктор
 {
 marka = "bezimena";
 boja = "crna";
 cena = 100.00;
 }
cokolada::cokolada(char* m ,char* b,float c)   // конструктор за чоколаду
 {
 marka = m;
 boja = b;
 cena = c;
 }
void cokolada::prikaziCokoladu() const
 {
  cout << marka << endl;
  cout << boja << endl;
  cout << cena << endl;
 }

#include <cstdlib>
#include <iostream>
#include "cokolada.h"
using namespace std;

int main()
{
cokolada c();
c.prikaziCokoladu();
cokolada milka("milka","bela",92.50);
milka.prikaziCokoladu();
cout << endl;
system("PAUSE");
return 0;
}
```

Питања:

1. Шта су конструктори?
2. Како се дефинишу конструктори дате класе?
3. Да ли декларација конструктора заузима меморијски простор?
4. Каква је разлика између метода и конструктора?
5. Да ли се може дефинисати више конструктора за једну класу?

Деструктори класа

Појам и дефинисање деструктора

Функција чланица која носи исто име као и класа, а испред ње стоји знак „**~**“ назива се деструктор (енгл. *Destructor*).

Ова функција позива се аутоматски, при престанку живота објекта класе, за све наведене случајеве (статичких, аутоматских, класних чланова, динамичких и привремених објеката).

Деструктор нема тип који враћа и не може имати аргументе.

Унутар деструктора, приватним члановима приступа се као и у било којој другој функцији чланици.

Свака класа може да има највише један деструктор.

Дефинисање конструктора је слично као и за друге методе, али нема тип који враћа и мора имати исто име као и класа:

```
~ime_klase();
~ime_klase()  {telo konstruktora}
```

Пример: Посматраћемо класу датум коју смо дефинисали раније и дефинисати деструктор за дату класу.

```
class Datum
{
 private:
 int dan;
 int mesec;
 int godina;
 public:
 void postaviDan(int d) {dan = d;}
 void postaviMesec(int m) {mesec = m;}
 void postaviGodinu(int g){godina = g;}
 void prikaziDatum() const{cout<<dan<<"."<<mesec<<"."<<godina<<endl;}
 Datum() {dan = 0;mesec = 0;godina = 0;}       // подразумевани конструктор
 ~Datum() {}                                   // деструктор
};
```

Када се позива деструктор?

Деструктор је функција која претвара меморијске локације које је систем одвојио за нови објекат (и све његове податке чланове) у слободан меморијски простор, који се може користити за неке друге намене.

Деструктор се имплицитно позива и при уништавању динамичког објекта помоћу оператора делете.

За низ, деструктор се позива за сваки елемент понаособ.

Редослед позива деструктора је у сваком случају обратан редоследу позива конструктора.

Деструктори се углавном користе када објекат треба да деалоцира меморију или неке системске ресурсе које је конструктор алоцирао; то је најчешће случај када класа садржи чланове који су показивачи.

После извршавања тела деструктора, аутоматски се ослобађа меморија коју је објекат заузимао.

Пример: Користићемо класу датум коју смо већ дефинисали.

```
#include <cstdlib>
#include <iostream>
#include "datum.h"
using namespace std;

int main()
{
Datum d1, d2;               // креирамо објекат d1 и d2
Datum *X = new Datum[2];    // креирамо динамички низ објеката типа Datum
cout << endl;
system("PAUSE");
return 0;
}                           // Позива се деструктор за све креиране објекте
```

Пример: Направити класу реченица, која има особину реченица и реченица без празнина, методе за приказ и конструкторе и деструктор.

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

class Recenica
{
private:
 char *recenica, *recenica1;
public:
 void prikaziRecenicu() const;          // приказује реченицу
 void prikaziRecenicuBezPraznina();     // приказује реченицу без знака „ „
 Recenica();                            // подразумевани конструктор
 Recenica(char* r);                     // конструктор
 ~Recenica();                           // деструктор
};
Recenica::Recenica()                    // подразумевани конструктор
 {
 recenica = 0;                          // иницијализујемо низове на нулу,
 recenica1 = 0;                         // јер не знамо реченицу
 }
Recenica::Recenica(char* r)             // конструктор
 {
 recenica = r;                          // Реченица је унета и сада треба
 int x = 1, y = 0;                      // да одредимо колико слова има која
 for(int i = 0; i < strlen(r); i++)     // нису једнака ' '.
   if(recenica[i] != ' ')               // x и y су бројачи
      x++;
 recenica1 = new char [x];              // recenica1 је показивач на динамички
 for(int i = 0; i < x+1; i++)           // низ који се креира
   if(recenica[i] != ' ')
      {
      recenica1[y] = recenica[i];
      y++;
      }
 }
 void Recenica::prikaziRecenicu() const       // метода
 {
  cout << recenica << endl;
 }
void Recenica::prikaziRecenicuBezPraznina()   // метода
 {
  cout << recenica1 << endl;
 }
```

```
Recenica::~Recenica()                        // деструктор
 {
 delete [] recenica1;                        // Унистава динамички низ и ослобађа
 }                                           // заузету меморију

int main()
{

Recenica R1, R2("Ovo je proba");             // Креирамо два објекта R1 i R2
R1.prikaziRecenicu();                        // R1 не креира динамички низ
R2.prikaziRecenicu();                        // R2 креира динамички низ који
R2.prikaziRecenicuBezPraznina();             // би на крају требали да ослободимо

cout << endl;
system("PAUSE");
return 0;
}
```

Питања:

1. Шта су деструктори?
2. Како се дефинишу деструктори дате класе?
3. Каква је разлика између деструктора и конструктора?
4. Да ли се може дефинисати више деструктора за једну класу?

Пријатељи класа

Појам и дефинисање пријатеља класа

Често је добро да се класа пројектује тако да има и "повлашћене" кориснике, односно функције или друге класе које имају право приступа њеним приватним члановима. Такве функције и класе називају се пријатељима (енгл. *friends*).

Пријатељске функције

Пријатељске функције (енгл. *friend functions*) су функције које нису чланице класе, али имају приступ до приватних чланова класе. Те функције могу да буду глобалне функције или чланице других класа.

Да би се нека функција прогласила пријатељем класе, потребно је у декларацији те класе навести декларацију те функције са кључном речи **`friend`** испред. Пријатељска функција се дефинише на уобичајен начин:

пријатељска глобална функција:

```
friend tip ime_funkcije(argumenti);
friend tip ime_funkcije(argumenti) {telo funkcije};
```

пријатељска чланица друге класе:

```
friend tip ime_klase::ime_funkcije(argumenti);
```

Глобалне функције које представљају услуге неке класе или операције над том класом (најчешће су пријатељи те класе) називају се класним услугама (енгл. *class utilities*).

Нема формалних разлога да се користи глобална (најчешће пријатељска) функција уместо функције чланице.

Постоје прилике када су глобалне (пријатељске) функције погодније:

- функција чланица мора да се позове за објекат дате класе, док глобалној функцији може да се достави и објекат другог типа, који ће се конвертовати у потребни тип;
- када функција треба да приступа члановима више класа, ефикаснија је пријатељска глобална функција;
- понекад је нотационо погодније да се користе глобалне функције (позив је **`f(x)`**) него чланице (позив је **`x.f()`**); на пример, **`max(a,b)`** је читљивије од **`a.max(b)`**;
- када се преклапају оператори, често је једноставније дефинисати глобалне (операторске) функције неко чланице.

"Пријатељство" се не наслеђује: ако је функција f пријатељ класи X, а класа Y изведена (наследник) из класе X, функција f није пријатељ класи Y.

Пример: Посматраћемо класу време која има особине сат, минут и секунда, и дефинисати пријатељску функцију која ће претварати задато време у секунде.

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

class vreme
{
  private:
```

```
    int sat, min, sek;    //време се састоји од сати, минута и секунди.
  public:
    vreme();
    vreme(int h, int m, int s);
    void Unesi_vreme();
    void Prikazi_vreme() const;
    friend int sekunde(vreme t);
};
// подразумевани конструктор
vreme::vreme() {sat = 0; min = 0; sek = 0;}
//конструктор
vreme::vreme(int h, int m, int s) {sat = h; min = m; sek = s;}
// јавна метода
void vreme::Unesi_vreme() {cin >> sat >> min >> sek;}
// јавна метода
void vreme::Prikazi_vreme() const {cout <<sat<<":"<<min<<":"<<sek<<endl;}
// Ово је пријатељска ф-ја и сме да приступа приватним подацима објеката
 // ове класе иако она не припада класи време (па нема ни „::“).
int sekunde(vreme t) {return t.sat* 60*60 + t.min*60 + t.sek;}

int main()
{
  vreme t(10,15,25);
  t.Prikazi_vreme();
  cout << " pretvoreno u sekunde je " << sekunde(t) << " sekundi." << endl;
  system("PAUSE");
  return 0;
}
```

Пријатељске класе

Ако је потребно да све функције чланице класе Y буду пријатељске функције класи X, онда се класа Y декларише као пријатељска класа (*friend class)* класи X. Тада све функције чланице класе Y могу да приступају приватним члановима класе X, али обратно не важи ("пријатељство" није симетрична релација):

```
class X
{
friend class Y;
//...
};
```

"Пријатељство" није ни транзитивна релација: ако је класа Y пријатељ класи X, а класа Z пријатељ класи Y, класа Z није аутоматски пријатељ класи X, већ то мора експлицитно да се нагласи (ако је потребно).

Пријатељске класе се типично користе када неке две класе имају тешње међусобне везе. При томе је непотребно (и лоше) "откривати" делове неке класе да би они били доступни другој пријатељској класи, јер ће на тај начин они бити доступни и осталима (руши се енкапсулација). Тада се ове две класе проглашавају пријатељским.

Пример: Посматраћемо класе време и датум које смо дефинисали раније и дефинисати пријатељску класу.

```
class Datum
{
 friend class Vreme;                          // пријатељска класа
 private:
 int dan;
 int mesec;
 int godina;
```

```
 public:
 void postaviDan(int d) {dan = d;}
 void postaviMesec(int m) {mesec = m;}
 void postaviGodinu(int g){godina = g;}
 void prikaziDatum() const{cout<<dan<<"."<<mesec<<"."<<godina<<endl;}
 Datum() {dan = 0; mesec = 0; godina = 0;}
 ~Datum() {}
};

class Vreme
{
 private:
 int sat;
 int minut;
 int sekunda;
 public:
 void postaviSat(int h) {sat = h;}
 void postaviMinut(int m) { minut = m;}
 void postaviSekundu(int s){ sekunda = s;}
 void prikaziVreme() const{cout<<sat<<":"<<minut<<":"<<sekunda<<endl;}
 Vreme() {sat = 0; minut = 0; sekunda = 0;}
 ~Vreme() {}
};
```

Питања:

1. Шта су пријатељи класа?
2. Како се дефинишу пријатељске функције дате класе?
3. Како се дефинишу пријатељске класе дате класе?
4. Да ли се може дефинисати више пријатељских функција за једну класу?

Заједнички елементи класа

Појам и дефинисање заједничких елемената класа

Понекада је добро да се класа пројектује тако да има и заједничке елементе, односно промењиве и функције које су заједничке за све објекте дате класе. Такве функције и класе називају се заједнички елементи, а сви остали чланови су сопствени.

Осим ако се експлицитно не наведе да је неки члан заједнички, онда је он подразумевано сопствени.

Заједнички подаци чланови

При креирању објеката класе, за сваки објекат се креира посебан комплет података чланова. Ипак, могуће је дефинисати податке чланове за које постоји само један примерак за целу класу, тј. за све објекте класе.

Овакви чланови називају се статичким члановима, и декларишу се помоћу речи **static**.

```
static tip ime;
```

Сваки приступ статичком члану из било ког објека класе значи приступ истом заједничком члану-објекту.

Статички члан класе има животни век као и глобални статички објекат, односно настаје на почетку програма и траје до краја програма. Уопште, статички члан класе има сва својства глобалног статичког објекта, осим области важења класе и контроле приступа.

Статички члан мора да се иницијализује посебном декларацијом ван декларације класе и тада се обраћање оваквом члану ван класе врши се преко оператора `::`.

На пример: **int X::i=5;**

Статичком члану може да се приступи из функције чланице, али и ван функција чланица, чак и пре формирања иједног објекта класе (јер статички члан настаје као и глобални објекат), наравно уз поштовање права приступа.

Тада му се приступа преко оператора `::`.

Заједнички чланови се углавном користе када сви примерци једне класе треба да деле неку заједничку информацију, нпр. када представљају неку колекцију, односно када је потребно имати их "све на окупу и под контролом".

Заједнички чланови смањују потребу за глобалним објектима и тако повећавају читљивост програма, јер је могуће ограничити приступ њима, за разлику од глобалних објеката.

Заједнички чланови логички припадају класи и "упаковани" су у њу.

Пример: Посматраћемо класу мачка која има особине име, боја и број ногу. Како је познато име и боја се обично разликују, али је број ногу сталан, па га можемо прогласити за заједничку особину.

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

class macka
{
 private:
  char *ime;                    // сопствена особина
  char *boja;                   // сопствена особина
  static int broj_nogu;         // заједничка особина
```

```
 public:
  macka()                      // подразумевани конструктор
  {
    ime = "bela";
    boja = "crna";
  }

  macka(char *i, char *b)      // конструктор
  {
    ime = i;
    boja = b;
  }

  ~macka() {}                  // деструктор

  void prikazi()               // сопствена метода
  {
   cout<<"Macka "<<ime<<" je "<<boja<<"boje i ima "<<broj_nogu<<" noge"<<endl;
  }
};

// Иницијализацију заједничке промењиве морамо извршити ван класе.
// Пошто је ово глобална промењива може јој се приступити ван било које
// функције, чак и пре него што креирамо неки објекат овог типа
int macka::broj_nogu = 4;

int main()
{
  // Креирамо два објекта типа macka
  macka pera("Pera","bela"), mika("Mika","Turundzasta");
  // Приказујемо особине креираних објеката
  pera.prikazi();
  mika.prikazi();
  cout << endl;
  system("PAUSE");
  return 0;
}
```

Заједничке функције чланице

И функције чланице могу да се декларишу као заједничке за целу класу, додавањем речи `static` испред декларације функције чланице.

```
static tip ime_funkcije(argumenti) {}
```

Статичке функције чланице имају сва својства глобалних функција, осим области важења и контроле приступа. Оне не поседују показивач `this` и не могу непосредно (без помињања конкретног објекта класе) користити нестатичке чланове класе. Могу непосредно користити само статичке чланове те класе.

Статичке функције чланице се могу позивати за конкретан објекат (што нема посебно значење), али и пре формирања иједног објекта класе, преко оператора `::`.

Пример: Написати класу правоугаоник која има особине странице а и б и заједнички елемент који ће бројати колико има креираних објеката овог типа, као и заједничку методу која враћа број креираних правоугаоника.

```
#include<iostream>
#include<cstdlib>
```

```
using namespace std;

class pravougaonik
{
  private:
    int a,b;                        // сопствене особине
    static int broj_pravougaonika;  // заједничке особине
  public:
    pravougaonik()                  //конструктор
    {
      a = 0;
      b = 0;
      broj_pravougaonika++;
    }
    pravougaonik(int x, int y)      //конструктор
    {
      a = x;
      b = y;
      broj_pravougaonika++;
    }
    ~pravougaonik() {};             //деструктор
    void Unesi_stranice()           //сопствена метода
    {
      cout << "unesi stranice pravougaonika" << endl;
      cin >> a >> b;
    }
    static int broj()               // заједничка метода
    {
      return broj_pravougaonika;
    }
};

// Иницијализацију заједничке промењиве морамо извршити ван класе.
 // Пошто је ово глобална промењива може јој се приступити ван било које
 // функције, чак и пре него што креирамо неки објекат овог типа
int pravougaonik::broj_pravougaonika = 0;

int main()
{
  pravougaonik A, B(3,2), C;
 // приступићемо статичкој промењивој преко имена класе
  cout <<"ukupno je kreirano "<<pravougaonik::broj()<<" pravougaonika"<< endl;
 //Ако пробамо да приступимо заједницкој методи сваког од објекта добијамо:
  cout << "objekat A: " << A.broj() << endl;
  cout << "objekat B: " << B.broj() << endl;
  cout << "objekat C: " << C.broj() << endl;

  cout << endl;
  system("PAUSE");
  return 0;
}
```

Питања:

1. Шта су заједнички елементи класа?
2. Како се дефинишу заједничке функције дате класе?
3. Како се дефинишу заједничке особине дате класе?
4. Да ли се може дефинисати више заједничких особина и функција за једну класу?
5. Да ли се може променити вредност заједничким елементима класа ако нисмо дефинисали ниједан објекат?

Преклапање оператора

Појам преклапања оператора

C++ програмски језик нам омогућује да креирамо сложене типове података помоћу класа, али исто тако и да креирамо операторе који ће радити са таквим типовима података. Ово нам омогућује да са новим објектима радимо на потпуно природан начин као да су они део уобичајеног скупа података.

На пример, ако знамо да се комплексни бројеви представљају у облику Z = Re + i*Im и ако знамо да се комплексни бројеви сабирају тако што им се засебно саберу реални и имагинарни делови, тада имамо две опције ако направимо класу комплексних бројева:

1) дефинисати функцију Сабирање(З1,З2) која ће вршити сабирање комплексних бројева, или
2) дефинисати оператор `+` за ову класу, који ће радити са комплексним бројевима исто што и претходна функција, али на сасвим природан начин.

Исто можемо урадити и са осталим уобичајеним операторима који се примењују код реалних бројева, уз изузетке који ће бити напоменути касније.

На овај начин нам је у језику C++ омогућено да користимо уобичајене операторе за класе које смо направили.

Правила за дефинисање операторских функција

У језику C++, поред "обичних" функција које се експлицитно позивају навођењем идентификатора са заградама, постоје и операторске функције.

Операторске функције су посебна врста функција које имају посебна имена и начин позивања. Као и обичне функције, и оне се могу преклопити за операнде који припадају корисничким типовима. Овај принцип назива се преклапање оператора (енгл. operator overloading).

Овај принцип омогућава да се дефинишу значења оператора за корисничке типове и формирају изрази са објектима ових типова, на пример операције над комплексним бројевима, матрицама итд.

Ипак, постоје нека ограничења у преклаању оператора:

1. не могу да се преклопе оператори **`.`**,**` .*`**, **`::`**, **`?:`** i **`sizeof`**, док сви остали могу;
2. не могу да се редефинишу значења оператора за уграђене (стандардне) типове података;
3. не могу да се уводе нови симболи за операторе;
4. не могу да се мењају особине оператора које су уграђене у језик: n-арност, приоритети и асоцијативност (смер груписања).

Операторске функције имају имена operator@, где је `@` знак оператора. Операторске функције могу бити чланице или глобалне функције (углавном пријатељи класа) код којих је бар један аргумент типа корисничке класе.

Име оператора @ се може писати спојено са речи operator ако је у питању неки симбол, али се мора писати одвојено ако је у питању неки оператор чије име се састоји од слова. На пример: operator+, operator+= или operator +, operator +=, али увек мора овако operator new.

За корисничке типове су унапред дефинисана увек два оператора: `=` (додела вредности) и `&` (узимање адресе) и све док их корисник не редефинише, они имају подразумевано значење.

Подразумевано значење оператора `=` је копирање објекта доделом члан по члан (позивају се оператори `=` класа којима чланови припадају, ако су дефинисани). Ако објекат садржи члан који је показивач, копираће се, наравно, само тај показивач, а не и показивана вредност.

Ово некад није одговарајуће и корисник треба да редефинише оператор `=`.

Вредности операторских функција могу да буду било ког типа, па и void.

Бочни ефекти и везе између оператора

Бочни ефекти који постоје код оператора за уграђене типове никад се не подразумевају за редефинисане операторе: `++` не мора да мења стање објекта, нити да значи сабирање са 1. Исто важи и за `- -` и све операторе доделе (`=`,` + =`, `- =`, `* =` итд.).

Оператор `=` (и остали оператори доделе) не мора да мења стање објекта.

Ипак, овакве употребе треба строго избегавати: редефинисани оператор треба да има исто понашање као и за уграђене типове.

Везе које постоје између оператора за уграђене типове се не подразумевају за редефинисане операторе. На пример, а+=б не мора да аутоматски значи а=а+б, ако је дефинисан оператор `+`, већ оператор `+ =` мора посебно да се дефинише.

Строго се препоручује да оператори које дефинише корисник имају очекивано значење, ради читљивости програма.

На пример, ако су дефинисани и оператор `+ =` и оператор `+`, добро је да а+=б има исти ефекат као и а=а+б. Треба избегавати неочекивана значења, на пример да оператор `-` реализује сабирање матрица.

Када се дефинишу оператори за класу, треба тежити да њихов скуп буде комплетан. На пример, ако су дефинисани оператори `=` и `+`, треба дефинисати и оператор `+ =`; или, увек треба дефинисати оба оператора `= =` и `! =`, а не само један.

Дефинисање операторских функција

Операторске функције могу да буду чланице класа или (најчешће пријатељске) глобалне функције. Ако је @ неки бинарни оператор (на пример `+`), он може да се реализује као функција чланица класе X на следећи начин (могу се аргументи преносити и по референци):

```
tip operator@ (X)
```

или као пријатељска глобална функција на следећи начин:

```
tip operator@ (X,X)
```

Није дозвољено да се у програму налазе обе ове функције.

Позив **а@б** се сада тумачи као:

a.operator@(b) , за функцију чланицу, или:

operator@(a,b) , за глобалну функцију.

Пример: Дефинисаћемо класу комплексних бројева и оператор `+` за ту класу.

```
//Оператор `+` као чланица класе
class complex
{
  double real,imag;
 public:
  complex (double r=0, double i=0) : real(r), imag(i) {}
  complex operator+( complex c)
  {
    return complex(real+c.real,imag+c.imag;
  }
};
```

или, алтернативно:

```
//Оператор `+` као глобална пријатељска функција
class complex
{
  double real,imag;
 public:
  complex (double r=0, double i=0) : real(r), imag(i) {}
  friend complex operator+(complex,complex);
};

complex operator+ (complex c1, complex c2)
{
  return complex(c1.real+c2.real,c1.imag+c2.imag);
}

void main ()
{
  complex c1(2,3),c2(3.4);
  complex c3=c1+c2; // позива се c1.operator+(c2) или operator+(c1,c2)
  //...
}
```

Разлози за избор једног или другог начина (чланица или пријатељ) су исти као и за друге функције.

Овде постоји још једна разлика: ако за претходни пример желимо да се може вршити и операција сабирања реалног броја са комплексним, треба дефинисати глобалну функцију. Ако хоћемо да се може извршити d+c, где је d типа double, не можемо дефинисати нову операторску "чланицу класе double", јер уграђени типови нису класе (C++ није чисти OO језик). Операторска функција чланица "не дозвољава промоцију левог операнда", што значи да се неће извршити конверзија операнда d у тип complex. Треба изабрати други наведени поступак (са пријатељском операторском функцијом).При креирању објеката класе, за сваки објекат се креира посебан комплет података чланова. Ипак, могуће је дефинисати податке чланове за које постоји само један примерак за целу класу, тј. за све објекте класе.

Унарни и бинарни оператори

Многи оператори језика C++ (као и језика C) могу да буду и унарни и бинарни (унарни и бинарни -, унарни &-адреса и бинарни &-логичко I по битовима итд.).

Како разликовати унарне и бинарне операторе приликом преклапања?

Унарни оператор има само један операнд, па се може реализовати као операторска функција чланица без аргумената (први операнд је објекат чија је функција чланица позвана):

```
tip operator@ ()
```

или као глобална функција са једним аргументом:

```
tip operator@ (X x)
```

Бинарни оператор има два аргумента, па се може реализовати као функција чланица са једним аргументом (први операнд је објекат чија је функција чланица позвана):

```
tip operator@ (X xdesni)
```

или као глобална функција са два аргумента:

```
tip operator@ (X xlevi, X xdesni)
```

Основни стандардни улазно/излазни токови - Класе istream и *ostream*

Као и језик C, ни C++ не садржи (у језик уграђене) улазно/излазне (U/I) операције, већ се оне реализују стандардним библиотекама. Ипак, C++ садржи стандардне U/I библиотеке реализоване у духу ООП-а.

На располагању су и старе C библиотеке са функцијама *scanf* и *printf*, али њихово коришћење није у духу језика C++.

Библиотека чије се декларације налазе у заглављу *<iostream.h>* садржи две основне класе, *istream* и *ostream* (ulazni i izlazni tok). Сваком примерку (објекту) класа *ifstream* и *ofstream*, које су редом изведене из наведених класа, може да се придружи једна датотека за улаз/излаз, тако да се датотекама приступа искључиво преко оваквих објеката, односно функција чланица или пријатеља ових класа. Тиме је подржан принцип енкапсулације.

У овој библиотеци дефинисана су и два кориснику доступна (глобална) статичка објекта:

1. објекат *cin* класе istream који је придружен стандардном улазном уређају (обично тастатура);
2. објекат *cout* класе ostream који је придружен стандардном излазном уређају (обично екран).

Класа *istream* је преклопила оператор >> за све уграђене типове, који служи за улаз података:

```
istream& operator>> (istream &is, tip &t);
```

где је тип неки уграђени тип објекта који се чита.

Класа *ostream* је преклопила оператор << за све уграђене типове, који служи за излаз података:

```
ostream& operator<< (ostream &os, tip x);
```

где је тип неки уграђени тип објекта који се исписује.

Ове функције враћају референце, тако да се може вршити вишеструки U/I у истој наредби.

Осим тога, ови оператори су асоцијативни слева, тако да се подаци исписују у природном редоследу.

Ове операторе треба користити за уобичајене, једноставне U/I операције.

Улазно/излазне операције за корисничке типове

Корисник може да дефинише значења оператора >> и << за своје типове. То се ради дефинисањем пријатељских функција корисникове класе, јер је први операнд типа istream& односно ostream&.

Пример за класу complex:

```
#include <iostream.h>

class complex
{
 private:
  double real,imag;
  friend ostream& operator<< (ostream&,const complex&);

 public:
  //... као и раније
};
```

```
//...

ostream& operator<< (ostream &os, const complex &c)
{
  return os<<"("<<c.real<<","<<c.imag<<")";
}

void main ()
{
complex c(0.5,0.1);
cout<<"c="<<c<<"\n";          // исписује се: c=(0.5,0.1)
}
```

Питања:

1. Зашто се врши преклапање оператора у класама?
2. Како се дефинишу преклопљени оператори дате класе?
3. Који се оператори никада не смеју преклопити?
4. Ако се не дефинишу оператори за дату класу, који су подразумевани оператори?
5. Да ли се могу дефинисати нове ознаке за операторе?

Изведене класе

Појам изведених класа

Често се среће случај да постоји велика зависност између објеката који нас окружују, где један објекат има све особине као и неки други, али и неке додатне могућности. Тако на пример имамо мобилни телефон, који генерално има функцију позива, адресара, слања текстуалних порука, али су различити произвођачи додали неке само њиховим телефонима својствене додатке, који додатно проширују дате могућности. На пример неки произвођачи у својим мобилним телефонима омогућују да се под једним именом могу забележити више телефонских бројева, електронска адреса, надимак, титула и друго. Из овога видимо да сваки мобилни телефон мора подржавати одређене особине, али да је у њих могуће имплементирати и неке друге.

На основу овога можемо закључити да неке класе могу садржати све чланове друге класе, али могу имати и неке своје специфичне чланове. Тада кажемо да је да имамо једну класу која је специфичан представник дате класе и таква релација се у програмирању назива наслеђивање (енг. *inheritance*).

Наслеђивање не мора постојати само између две класе, можемо имати читаву хијерархију. Тако, на пример квадрат је специфичан објекат типа правоугаоник, правоугаоник је специфичан примерак класе четвороугао, четвороугао је специфичан примерак класе дводимензионалне фигуре. Постоји небројено много примера за све ово, али једноставно се може рећи да уколико класа B има све особине класе A, ако и неке своје које не постоје у класи A, тада је класа B изведена из класе A.

Ако је класа B наследила класу A, каже се још да је класа A основна класа (енгл. base class), а класа B изведена класа (енгл. derived class). Или да је класа A надкласа (енгл. superclass), а класа B подкласа (енгл. Subclass). Или да је класа A родитељ (енгл. parent), а класа B дете (енгл. child).

C++ програмски језик је објектно-оријентисан и као такав мора подржавати и изведене класе.

За разлику од C++ програмског језика који има и особине процедуралних језика, као заоставштину програмског језика C који је наследио, остали објектно-оријентисани програмски језици имају као основну класу *објекат* из које су изведени сви остали објекти.

Декларација изведених класа

Декларација изведених класа је слична као и за основне класе.

Да би се класа извела из неке постојеће класе, није потребно вршити никакве измене постојеће класе, па чак ни њено поновно превођење. Изведена класа се декларише навођењем речи **`public`** и назива основне класе, иза знака `` `:` `` (двотачка).

```
class osnovna
{
...
};
class izvedena : public osnovna
{
...
};
```

Сви елементи основне класе су и елементи изведене класе, а у оквиру изведене класе могуће је дефинисати и додатне елементе, који су својствени само њој.

Потребно је још напоменути да се не наслеђује оператор `=`.

Права приступа члановима класа

Кључна реч **public** у заглављу декларације изведене класе значи да су сви јавни чланови основне класе уједно и јавни чланови изведене класе.

Приватни чланови основне класе увек то и остају. Функције чланице изведене класе не могу да приступају приватним члановима основне класе. Нема начина да се "повреди приватност" основне класе (уколико неко није пријатељ те класе, што је записано у њеној декларацији), јер би то значило да постоји могућност да се пробије енкапсулација коју је замислио пројектант основне класе.

Јавним члановима основне класе се из функција чланица изведене класе приступа непосредно, као и сопственим члановима.

Често постоји потреба да неким члановима основне класе могу да приступе функције чланице изведених класа, али не и корисници класа. То су најчешће функције чланице које директно приступају приватним подацима члановима.

Чланови који су доступни само изведеним класама, али не и корисницима споља, наводе се иза кључне речи `**protected:**`и називају се заштићени чланови (енгл. *protected members*).

Заштићени чланови остају заштићени и за следеће изведене класе при сукцесивном наслеђивању. Уопште, не може се повећати право приступа неком члану који је приватан, заштићен или јавни.

Уколико се иза знака `:` не наведе **public** или се стави **private** подразумева се да сви јавни и заштићени подаци основне класе постају приватни подаци за изведену класу.

Ако се иза знака `:` наведе protected подразумева се да сви јавни и заштићени подаци основне класе постају заштићени подаци за изведену класу.

Конструктори и деструктори изведених класа

Приликом креирања објекта изведене класе, позива се конструктор те класе, али и конструктор основне класе. У заглављу дефиниције конструктора изведене класе, у листи иницијализатора, могуће је навести и иницијализатор основне класе (аргументе позива конструктора основне класе). То се ради навођењем имена основне класе и аргумената позива конструктора основне класе.

При креирању објекта изведене класе редослед позива конструктора је следећи:

1. иницијализује се подобјекат основне класе, позивом конструктора основне класе;
2. иницијализују се подаци чланови, евентуално позивом њихових конструктора, по редоследу декларисања;
3. извршава се тело конструктора изведене класе.

При уништавању објекта, редослед позива деструктора је увек обратан.

Ово је најлакше схватити на следећем примеру:

Креираћемо две класе: основна и изведена, и у конструктору и деструктору сваке класе задаћемо да се испише одговарајућа порука.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

class osnovna
{
  public:
    osnovna()  {cout<<"Konstruktor osnovne klase.\n";}
    ~osnovna() {cout<<"Destruktor osnovne klase.\n";}
};
```

```
class izvedena : public osnovna
{
  public:
    izvedena()  {cout<<"Konstruktor izvedene klase.\n";}
    ~izvedena() {cout<<"Destruktor izvedene klase.\n";}
};

int main ()
{
  izvedena test;  // креирамо објекат изведене класе
  system("PAUSE");
  return 0;
}
```

Након компајлирања и покретања програма на излазу добијамо следеће:

```
Konstruktor osnovne klase.
Konstruktor izvedene klase.
Destruktor izvedene klase.
Destruktor osnovne klase.
```

Пример: Креирати класу возило са особинама намена, број точкова, број седишта, као и методама за унос и приказ возила. Из ове класе ћемо извести класу путничко возило које има особину каросерија и методе за унос и приказ података.

```
class vozilo  // основна класа
{
  private:
   // Сви приватни подаци су приватни за ову класу и не може им се приступати
   // споља, па чак ни из изведених класа. Овим подацима могу приступати само
   // методе дефинисане у овој класи или пријатељске функције и класе.
    string namena;

  protected:
   // Ово су заштићени подаци, што значи да им се може приступати
   // из изведених класа
    int broj_tockova;
    int broj_sedista;

  public:
   // Ово су јавни елементи којима се мозе приступати споља, укључујуци и из
   // изведених класа.
    vozilo (string N = "", int bt = 0, int bs = 0)          // конструктор
    {
      namena = N; broj_tockova = bt; broj_sedista = bs;
    }
    void unesi_vozilo()                                      // јавна метода
    {
      cout << "Unesi namenu vozila" << endl;
      cin >> namena;
      cout << "Unesi koliko tockova ima vozilo" << endl;
      cin >> broj_tockova;
      cout << "Unesi koliko sedista ima vozilo" << endl;
      cin >> broj_sedista;
    }
    void prikazi_vozilo () const                             // јавна метода
    {
      cout << "namena vozila: " << namena << endl;
      cout << "broj tockova: " << broj_tockova << endl;
```

```
        cout << "broj sedista: " << broj_sedista << endl;
      }
};

class putnicko_vozilo: public vozilo   // изведена класа
{
  private:
   // приватним подацима ове класе може се приступати само помоћу метода ове
   // класе или из пријатељских функција или класа.

     string karoserija;  // лимузина, караван, купе,...

  public:
   // јавним елементима ове класе може се приступати и изван ње.
   // јавним методама основне класе можемо приступати као да су дефинисане
   // у овој класи, односно без навођења припадности.

   // конструктор изведене класе
   // наводе се прво елементи из конструктора основне класе,
   // а затим и из изведене. Након тога се ставља ':' (двотачка) иза
   // које следи иницијализација, односно додељивања вредности за
   // параметре основне класе и изведене класе

    putnicko_vozilo(string N = "", int bt = 0, int bs = 0, string k = ""):
    vozilo (N,bt,bs), karoserija(k) {}

    void unesi_putnicko_vozilo()                          // јавна метода
    {
   // cout << "Unesi namenu vozila" << endl;
   // cin >> namena;
   // ово не смемо урадити, јер је namena приватна особина основне класе
   // и не можемо јој приступати из изведене класе
      cout << "Unesi koliko tockova ima vozilo" << endl;
      cin >> broj_tockova;
      cout << "Unesi koliko sedista ima vozilo" << endl;
      cin >> broj_sedista;
      cout << "unesi koji je tip karoserije: " << endl;
      cin >> karoserija;
   // пошто смемо приступати јавним методама основне класе из изведене класе
   // могли смо написати и следеће:
   // unesi_vozilo(); - метода из основне класе која сме приступати приватним
   // особинама своје класе и мењати особину namena
   // cout << "unesi koji je tip karoserije: " << endl;
   // cin >> karoserija;
    }

    void prikazi_putnicko_vozilo() const                  // јавна метода
    {
      cout << "Vozilo: " << endl;
      cout << "karoserija: " << karoserija << endl;
      cout << "namena vozila: " << namena << endl;
      cout << "broj tockova: " << broj_tockova << endl;
      cout << "broj sedista: " << broj_sedista << endl;
   // могли смо написати и:
   // cout << "Vozilo: " << endl;
   // cout << "karoserija: " << karoserija << endl;
   // prikazi_vozilo();
    }

};
```

Полиморфизам

Полиморфизам је појава да се смеју преклапати имена метода у основној и изведеној класи. Тако, на пример, ако имамо иста имена метода у основној и у свакој од изведених класа из те основне приликом позива дате методе позива се одговарајућа из изведене класе.

Пример: Ако из основне класе фигура изведемо класе троугао, четвороугао и круг, и ако у свакој класи постоји метода **`crtaj()`**, тада можемо у нашем програму да креирамо неколико објеката изведених класа. Када треба да исцртамо све објекте на екрану није потребно да се сваком објекту појединачно да задатак да се исцрта, већ креирамо показиваче на те објекте типа фигура (јер они и јесу део основне класе фигура) и онда свима преко тих показивача треба дати задатак да се исцрта. Свака од прозваних фигура ће препознати којој класи припада и позвати своју одговарајућу методу за исцртавање.

Виртуелне функције

Функције чланице основне класе које се у изведеним класама могу реализовати специфично за сваку изведену класу називају се виртуелне функције (енгл. *virtual functions*).

Виртуелна функција се у основној класи декларише помоћу кључне речи виртуал на почетку декларације.

Приликом дефинисања виртуелних функција у изведеним класама не мора се стављати реч **`virtual`**.

Приликом позива се одазива она функција која припада класи којој и објекат који прима позив.

Виртуелна функција основне класе не мора да се редефинише у свакој изведеној класи. У изведеној класи у којој виртуелна функција није дефинисана, важи значење те виртуелне функције из основне класе.

Декларација неке виртуелне функције у свакој изведеној класи мора да се у потпуности слаже са декларацијом те функције у основној класи (број и типови аргумената, као и тип резултата).

Ако се у изведеној класи декларише нека функција која има исто име као и виртуелна функција из основне класе, али различит број и/или типове аргумената, онда она сакрива (а не редефинише) све остале функције са истим именом из основне класе. То значи да у изведеној класи треба поново дефинисати све остале функције са тим именом. Никако није добро (то је грешка у пројектовању) да изведена класа садржи само неке функције из основне класе, али не све: то значи да се не ради о правом наслеђивању (корисник изведене класе очекује да ће она испунити све задатке које може и основна класа).

Виртуелне функције морају бити чланице својих класа (не глобалне), а могу бити пријатељи других класа.

Вишеструко наслеђивање

Некад постоји потреба да изведена класа има особине више основних класа истовремено. Тада се ради о вишеструком наслеђивању (енгл. *multiple inheritance*).

На пример, мотоцикл са приколицом је једна врста мотоцикла, али и једна врста возила са три точка. При том, мотоцикл није врста возила са три точка, нити је возило са три точка врста мотоцикла, већ су ово две различите класе. Класа мотоцикала са приколицом налеђује обе ове класе.

Класа се декларише као наследник више класа тако што се у заглављу декларације, иза знака **`` `:` ``**, наводе основне класе раздвојене зарезима. Испред сваке основне класе треба да стоји реч **`public`**.

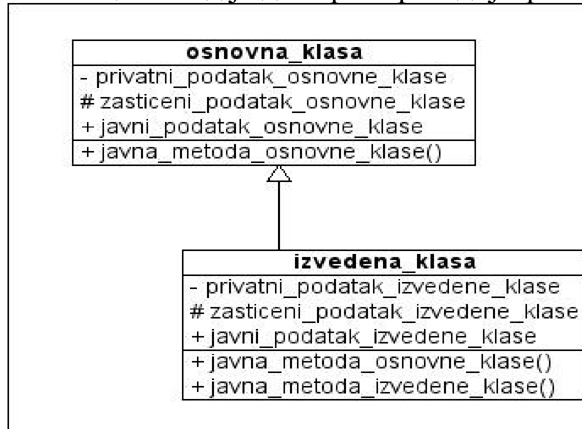
Na primer:

```
class izvedena : public osnovna1, public osnovna2, public osnovna3
{
//...
};
```

Дијаграми изведених класа

Као што смо научили пре овога свака класа се може представити дијаграмом, тако да и изведене класе можемо представити дијаграмом, али морамо представити и основне класе из којих су оне изведене, а затим их и повезати.

На слици испод је дат пример за дијаграме изведених класа.



Питања:

1. Зашто се користе изведене класе?
2. Како се дефинишу изведене класе?
3. Како су решена права приступа основној класи?
4. Којим редом се наводе елементи у конструктору изведених класа?
5. Којим редом се извршавају конструктори изведених класа?
6. Шта је то полиморфизам код изведених класа?
7. Какве су то виртуалне методе код изведених класа?
8. Да ли једна класа може бити изведена из више других класа?

Стварање и уништавање објеката изведене класе

За разлику од обичних класа стварање и уништавање објеката изведене класе се врши тако што се најпре иницијализује конструктор основне класе, па тек онда конструктор изведене класе. Због тога је битно најпре иницијализовати поља основне класе, јер ће она прва бити попуњена подацима, а онда по потреби иницијализовати и поља изведене класе ако су потребна.

Обрнутим редоследом се врши уништавање објеката изведених класа, јер се најпре позивају деструктор изведене класе, па тек онда деструктор основне класе.

Из овога се може закључити да је у основној класи пожељно увек дефинисати поред подразумеваног конструктора и конструктор који може иницијалиовати податке основне класе, јер ако њега нема позваће се подразумевани конструктор, а ако ни њега нема компајлер пријављује грешку.

Све ово се лако може показати на примеру.

Пример1:

```
#include <iostream>
#include <stdlib.h>

using namespace std;

class vozilo
{
  private:
    string namena;
  protected:
    int broj_tockova;
    int broj_sedista;
  public:
    vozilo (string N = "", int bt = 0, int bs = 0)            // конструктор
    {
      cout << "pozvan konstruktor iz osnovne klase" << endl;
      namena = N; broj_tockova = bt; broj_sedista = bs;
    }
 void unesi_vozilo()                                           // јавна метода
    {
      cout << "metoda iz osnovne klase" << endl;
      cout << "Unesi namenu vozila" << endl;
      cin >> namena;
      cout << "Unesi koliko tockova ima vozilo" << endl;
      cin >> broj_tockova;
      cout << "Unesi koliko sedista ima vozilo" << endl;
      cin >> broj_sedista;
    }
    void prikazi_vozilo () const                              // јавна метода
    {
      cout << "metoda iz osnovne klase" << endl;
      cout << "namena vozila: " << namena << endl;
      cout << "broj tockova: " << broj_tockova << endl;
      cout << "broj sedista: " << broj_sedista << endl;
    }
};                                                             // крај класе

class putnicko_vozilo: public vozilo                        // изведена класа
{
  private:
```

```
    string karoserija;                          // лимузина, караван, купе,...
  public:
    putnicko_vozilo(string N = "", int bt = 0, int bs = 0, string k = ""):
     vozilo (N,bt,bs), karoserija(k) {cout << "pozvan konstruktor iz izvedene
klase" << endl;}

    void unesi_putnicko_vozilo()                              // јавна метода
    {
      cout << "metoda iz izvedene klase" << endl;
      cout << "Unesi koliko tockova ima vozilo" << endl;
      cin >> broj_tockova;
      cout << "Unesi koliko sedista ima vozilo" << endl;
      cin >> broj_sedista;
      cout << "unesi koji je tip karoserije: " << endl;
      cin >> karoserija;
    }
    void prikazi_putnicko_vozilo() const                      // јавна метода
    {
      cout << "metoda iz izvedene klase" << endl;
      cout << "broj tockova: " << broj_tockova << endl;
      cout << "broj sedista: " << broj_sedista << endl;
      cout << "karoserija: " << karoserija << endl;
    }

};

int main()
{
  vozilo X, A("teretno", 8, 2);
  putnicko_vozilo Y, B("putnicko", 4, 2, "kupe");
  cout << endl;
  cout << "vozilo A (osnovna klasa)" << endl;
  A.prikazi_vozilo();
  cout << endl;
  cout << "vozilo B (izvedena klasa)" << endl;
  B.prikazi_putnicko_vozilo();
  cout << endl;
  X.unesi_vozilo();
  cout << endl;
  cout << "vozilo X (osnovna klasa)" << endl;
  X.prikazi_vozilo();
  cout << endl;
  Y.unesi_putnicko_vozilo();
  cout << endl;
  cout << "vozilo Y (izvedena klasa" << endl;
  Y.prikazi_putnicko_vozilo();
  system("PAUSE");
  return 0;
}
```

Пример2:

```
#include <iostream>
#include <stdlib.h>

using namespace std;

class vozilo
{
  private:
    string namena;
```

```
    protected:
      int broj_tockova;
      int broj_sedista;
    public:
      vozilo (string N = "", int bt = 0, int bs = 0)             // конструктор
      {
        cout << "pozvan konstruktor iz osnovne klase" << endl;
        namena = N; broj_tockova = bt; broj_sedista = bs;
      }
      void unesi_vozilo()                                        // јавна метода
      {
        cout << "metoda iz osnovne klase" << endl;
        cout << "Unesi namenu vozila" << endl;
        cin >> namena;
        cout << "Unesi koliko tockova ima vozilo" << endl;
        cin >> broj_tockova;
        cout << "Unesi koliko sedista ima vozilo" << endl;
        cin >> broj_sedista;
      }
      void prikazi_vozilo () const                               // јавна метода
      {
        cout << "metoda iz osnovne klase" << endl;
        cout << "namena vozila: " << namena << endl;
        cout << "broj tockova: " << broj_tockova << endl;
        cout << "broj sedista: " << broj_sedista << endl;
      }
  };                                                             // крај класе

  class putnicko_vozilo: public vozilo                           // изведена класа
  {
    private:
      string karoserija;
    public:
      putnicko_vozilo(string N = "", int bt = 0, int bs = 0, string k = ""):
       vozilo (N,bt,bs), karoserija(k) {cout << "pozvan konstruktor iz izvedene
klase" << endl;}

      void unesi_putnicko_vozilo()                               // јавна метода
      {
        cout << "metoda iz izvedene klase" << endl;
        unesi_vozilo();
        cout << "unesi koji je tip karoserije: " << endl;
        cin >> karoserija;
      }
      void prikazi_putnicko_vozilo() const                       // јавна метода
      {
        cout << "metoda iz izvedene klase" << endl;
        cout << "karoserija: " << karoserija << endl;
        prikazi_vozilo();
      }

  };

  int main()
  {
    vozilo X, A("teretno", 8, 2);
    putnicko_vozilo Y, B("putnicko", 4, 2, "kupe");
    cout << endl;
    cout << "vozilo A (osnovna klasa)" << endl;
    A.prikazi_vozilo();
    cout << endl;
```

```
  cout << "vozilo B (izvedena klasa)" << endl;
  B.prikazi_putnicko_vozilo();
  cout << endl;
  X.unesi_vozilo();
  cout << endl;
  cout << "vozilo X (osnovna klasa)" << endl;
  X.prikazi_vozilo();
  cout << endl;
  Y.unesi_putnicko_vozilo();
  cout << endl;
  cout << "vozilo Y (izvedena klasa" << endl;
  Y.prikazi_putnicko_vozilo();
  system("PAUSE");
  return 0;
}
```

Питања:

1. Зашто се користе изведене класе?
2. Како се дефинишу изведене класе?
3. Како су решена права приступа основној класи?
4. Којим редом се наводе елементи у конструктору изведених класа?
5. Којим редом се извршавају конструктори изведених класа?
6. Шта је то полиморфизам код изведених класа?
7. Какве су то виртуалне методе код изведених класа?
8. Да ли једна класа може бити изведена из више других класа?

Изузеци

Често се дешава случај да корисник не унесе одговарајући тражени податак или унесе погрешну вредност за неки податак, тако да у неком делу програма долази до грешке, која се не може решити. На пример, ако корисник унесе слово када је потребан број, или се може десити да унесе нулу као неки податак који се налази у имениоцу разломка и слично.

Када програм открије грешку коју није у стању да обради, уколико се она десила у некој функцији, та функција ће вратити као резултат неку информацију о томе функцији из које је позвана и тако редом до главне функције, која може прекинути извршавање програма.

Вероватно се питате зашто није то могуће уредити повратним вредностима функција (помоћу наредбе return). Просто, јер постоје и функције које немају повратну вредност (void), а у класама постоје и конструктори и деструктори, који такође немају повратне вредности. Осим тога испитивање грешака на овакав начин може бити компликовано и довести до неразумљивог кода.

Зато је елегантније решење ако би постојао неки начин која уме да реши конфликтну ситуацију, тако да програм може несметано наставити са радом.

Један начин како настаје грешка се лако може показати на примеру.

Пример:

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
  int a = 10, b = 0;
  cout << "napravicemo gresku tako što delimo neki broj sa nulom" << endl;
  cout << a/b;
  cout << "greska je ispred ove komande i ovo se nece prikazati !" << endl;
  system("PAUSE");
  return 0;
}
```

Програмски језик C++ нуди одређене методе за обраду могућих грешака које се називају изузеци (енг. *Exceptions*).

Он се састоји од дела за детекцију грешака, дела за пријаву и дела за исправку.

Пријављивање изузетака

По потреби се у одређеном делу кода, за који нисмо сигурни како ће се понашати у различитим ситуацијама, може користити пријављивање изузетака.

То се ради тако што користимо оператор **`throw`**, чија је основна синтакса следећа:

```
throw izraz;
```

Вредност израза може бити податак било којег типа (који је већ дефинисан), а он одређује ком ће се руковаоцу изузетака проследити настала грешка.

Оператор **`throw`** се може користити у било којем делу кода, па чак и у оквиру блока за детекцију грешака.

За неку функцију можемо ограничити који тип изузетака може пријавити тако што након саме декларације функције додамо **`throw(tip)`**.

Пример:

```
float MojaFunkcija (int param) throw (int); // изузеци су типа int
```

Овако дефинисана функција једино може пријавити изузетак целобројног типа.

Исто тако можемо урадити и следеће:
```
float MojaFunkcija (int param) throw (); //изузеци нису дозвољени
float MojaFunkcija (int param);          //изузеци било ког типа су дозвољени
```

Детекција и обрада изузетака

Детекција и обрада изузетака у програмском језику C++ користи кључне речи **try** (енгл. покушати, пробати) за означавање „ризичног“ блока и **catch** (енгл. хватати) за означавање блока у којем се обрађују изузеци из ризичног блока и **throw** (енгл. бацити) за пројављивање изузетака.

Општи облик би био:
```
try
{
    blok naredbi
}
catch (tip promenjiva)
{
    obrada izuzetaka
}
```

У оквиру блока **try** се ставља „ризични“ блок наредби, где се очекује појава изузетака, док се резервисана реч **catch** користи за прихватање и обраду изузетака. Треба напоменути да, по потреби, може постојати више **catch** блокова.

У оквиру **catch (tip promenjiva)** можемо користити различите типове података, али се једној **catch** структури преноси само изузетак који је истог типа као и промењива наведена као параметар. Ако нисмо сигурни ког типа је изузетак или ако желимо да ухватимо било који изузетак користимо **универзални руковалац**: **catch (...)**

Ако се до краја **try** блока не појави никакав изузетак, тада се прескачу сви **catch** блокови, али ако се појави изузетак, тада се прекида извршавање и прелази на одговарајући **catch** блок.

Поред свега овде наведеног треба нагласити да у C++ језику постоји и класа exception која је направљена да ради са изузецима.

Све ово је много лакше схватити на примерима.

Пример:Једноставан пример за изузетке.

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

int main()
{
  cout << "Start\n";

  try {
    cout << "unutar try bloka\n";
    throw 1;                          // користимо throw који баца int
    cout << "ovo se nece izvrsiti !!!";
  }
  catch (int i) {                     // хватамо изузетак типа int
    cout << "pronadjen izuzetak tipa int koji ima vrednost: ";
    cout << i << "\n";
  }
  cout << "kraj";
  return 0;
}
```

Након компајлитрања и извршења програма на излазу добијамо следеће:

```
Start
unutar try bloka
pronadjen izuzetak tipa int koji ima vrednost: 1
kraj
```

Пример: Изузетак из функције позване у оквиру блока **`try`**

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

void f(int x)
{
  cout << "Unutar funkcije f, vrednost promenjive x je: " << x << "\n";
  if(x)     // испитујемо да ли је x различито од 0
     throw x;
}

int main()
{
  cout << "start\n";
try
  {
    cout << "unutar try bloka\n";
    f(0); // pozivamo funkciju f prvi put
    f(1); // pozivamo funkciju f drugi put
    f(2); // pozivamo funkciju f treci put (ovo se nece dogoditi)
  }
catch (int i) // хватамо изузетак
  {
    cout << "pronadjen izuzetak tipa int koji ima vrednost: ";
    cout << i << "\n";
  }
  cout << "kraj";
  return 0;
}
```

Након компајлитрања и извршења програма на излазу добијамо следеће:

```
start
unutar try bloka
Unutar funkcije f, vrednost promenjive x je: 0
Unutar funkcije f, vrednost promenjive x je: 1
pronadjen izuzetak tipa int koji ima vrednost: 1
kraj
```

Пример: **`try`** блок се може налазити у оквиру тела неке функције

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

void f(int x)
{
  try
  {
    if(x)
      throw x;
  }
  catch(int i)
```

```
  {
    cout << "uhvatio izuzetak : " << i << '\n';
  }
}

int main()
{
  cout << "start\n";
  f(1);
  f(2);
  f(0);
  f(3);
  cout << "kraj";
  return 0;
}
```

Након компајлитрања и извршења програма на излазу добијамо следеће:

```
start
uhvatio izuzetak : 1
uhvatio izuzetak : 2
uhvatio izuzetak : 3
kraj
```

Пример: Коришћење више **catch** блокова

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

void f(int x)
{
  try
  {
    if(x)
       throw 1;          // throw int
    else
       throw 'a';        // throw char
  }
  catch(int i)
  {
    cout << "uhvatio izuzetak tipa int koji ima vrednost : " << i << '\n';
  }
  catch(char c)
  {
    cout << "uhvatio izuzetak tipa string: ";
    cout << c << '\n';
  }
}

int main()
{
  cout << "start\n";
  f(1);
  f(2);
  f(0);
  f(3);
  cout << "end";
  return 0;
}
```

Након компајлитрања и извршења програма на излазу добијамо следеће:

```
start
uhvatio izuzetak tipa int koji ima vrednost : 1
uhvatio izuzetak tipa int koji ima vrednost : 2
uhvatio izuzetak tipa string: a
uhvatio izuzetak tipa int koji ima vrednost : 3
```

Пример: Хватање свих изузетака.

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

void f(int x)
{
  try
  {
    if(x==0) throw x;       // throw int
    if(x==1) throw 'a';     // throw char
    if(x==2) throw 123.23; // throw double
  }
  catch(...)                // хватање свих изузетака
  {
    cout << "uhvatio !!!\n";
  }
}

int main()
{
  f(0);
  f(1);
  f(2);
  return 0;
}
```

Након компајлитрања и извршења програма на излазу добијамо следеће:

```
uhvatio !!!
uhvatio !!!
uhvatio !!!
```

Пример: Хватање изведене класе.

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

class osnovna {};

class izvedena: public osnovna {};

int main()
{
 izvedena X;
 try
 {
   throw X;
 }
 catch(osnovna A)
 {
   cout << "uhvatio osnovnu klasu!\n";
 }
```

```
 catch(izvedena B)
 {
   // ово се неће догодити јер пре тога хвата основну класу !
   cout << "uhvatio izvedenu klasu!\n";
 }
 return 0;
}
```

Пример: Коришћење класе **exception**.

```
#include <iostream>
#include <stdlib.h>

using namespace std;

int main()
{
  try
  {
    throw 1;
  }
  catch(exception& e)
  {
    cerr << "exception caught: " << e.what() << endl;
  }
  system("PAUSE");
  return 0;
}
```

Питања:

1. Чему служе изузеци?
2. Да ли **throw** може да проследи показивач или референцу?
3. Шта се дешава ако у **try** блоку не дође до појаве изузетака?
4. Ако се у **try** блоку појави изузетак, да ли се након повратка из **catch** блока наставља са наредбама у **try** блоку?
5. Ако се са **throw** пријави целобројни изузетак, а **catch** прима само реални број, шта ће се десити?
6. Како се може ухватити било који изузетак?