Милан Чабаркапа

КОНСТРУКТОРИ И ДЕСТРУКТОРИ

За поља објекта је природно очекивати, да буду иницијализована пре него што почну да се користе. Пошто је иницијализација објекта неопходна, C++ обезбеђује да се то реализује аутоматски при његовом креирању. Аутоматска иницијализација се реализује коришћењем методе класе коју називамо **конструктор** (constructor). Овим се обезбеђује један од основних принципа објектно оријентисаног програмирања: сви објекти морају бити самодовољни, тј. у потпуности опслуживати саме себе.

Конструктор је специјална метода, која је члан класе и има исто име као класа. Од обичне методе се разликује по томе што не враћа вредност. Зато се не наводи тип који враћа (чак ни `void`). Компајлер непогрешиво проналази овај метод јер му се име поклапа са именом класе. На пример, класа `tacka` се може описати на следећи начин:

```
#include <iostream.h>
class tacka
  {
    double x, y;    // podaci - podrazumevajuce skriveni (private)
  public:    // javni deo - dostupan u svim delovima programa
    tacka();    // konstruktor
    void translacija(double dx, double dy)
      { x+=dx; y+=dy; }
    void pozicija()
      {cout << "x=" << x <<" " << "y=" << y << endl;}
  };
```

Опис конструктора `tacka()` може се дати са:

```
tacka ::  tacka()
  {
    x=0;
    y=0;
    cout << "Tacka je inicijalizovana" << endl;
  }
```

Порука `"Tacka je inicijalizovana"` је укључена да би се показало да конструктор ради. У већини случајева конструктор не даје никакво саопштење.

Метода-конструктор се позива у моменту када се креира објекат, тј. када му се додели меморијски простор. За глобалне објекте конструктор се позива само једанпут – при креирању објекта на почетку извршавања програма. За локалне објекте конструктор се позива при уласку у блок у коме је дефинисан објекат. Конструктор се не може позвати експлицитно (нпр. за објекат `t` са `t.tacka()`). Конструктор, као и друге методе, може имати параметре, може бити преоптерећена метода, тј. класа може имати неколико конструктора. Ако у класи није описан ниједан конструктор, компајлер сам генерише подразумевајући конструктор.

Ако претходну класу тестирамо са:

```
tacka t;    // globalna promenljiva
void main()
  {
  tacka t1;
  t1.translacija(5.5,5.5); // translacija tacke
  t1.pozicija(); // ispis polozaja tacke
  tacka t2;
  t2.translacija(10,10); // translacija tacke
  t2.pozicija(); // ispis polozaja tacke
  }
```

исписује се:

```
Tacka je inicijalizovana
Tacka je inicijalizovana
x=5.5 y=5.5
Tacka je inicijalizovana
x=10 y=10
```

где прва порука: `"Tacka je inicijalizovana"` настаје активирањем конструктора при креирању глобалног објекта **t** (који се даље у програму не користи). Следеће две поруке настају при креирању објеката **t1** и **t2**.

Постоји још једна специјална метода класе – **деструктор** (destructor). Врло често је неопходно да се одраде неке завршне активности пре него што се објекат уклони из меморије. То може бити ослобађање меморије коју заузима нека динамички креирана структура коју садржи објекат, рестаурација екрана, затварање фајлова итд. У C++ такве активности извршава функција која се назива деструктор. При уклањању објекта из меморије треба да ослободи и меморију коју заузимају његова поља. Дакле, при креирању објекта и одвајању њему одговарајућег меморијског простора позива се конструктор, док при ослобађању меморијског простора додељеног објекту позива се деструктор. Он се скоро никад не позива експлицитно, иако за разлику од конструктора такав позив је дозвољен. Увек се позивају аутоматски када престаје бити активан блок у коме су креирани објекти. Такође се позивају при позиву оператора `delete` за показивач на објекат који има деструктор.

Деструктор представља методу чије је име исто као име класе, само што се испред имена деструктора наводи знак тилда (∼). Деструктор не враћа вредност (чак ни `void`) и не може имати параметре.

Следећи пример треба да илуструје да се деструктор позива аутоматски када се напушта блок у коме су дефинисани објекти. С тим циљем је предвиђено исписивање поруке `"Tacka je uklonjena"` при уклањању објеката. Редослед уклањања је по LIFO принципу, објекат који је последњи креиран биће први који се уклања. Извршавањем програма:

```
#include <iostream.h>
class tacka
  {
    double x, y;    // podaci - podrazumevajuce skriveni (private)
  public:    // javni deo - dostupan u svim delovima programa
    tacka(); // konstruktor
    ∼tacka(); // destruktor
    void translacija(double dx, double dy)
      { x+=dx; y+=dy; }
    void pozicija() cout << "x=" << x <<" " << "y=" << y << endl;
  };
tacka ::  tacka()
  {
  x=0;y=0;
  cout << "Tacka je inicijalizovana" << endl;
  }
tacka ::  ∼tacka()
  { cout << "Tacka je uklonjena!" << endl; }
tacka t;
void main()
  {
  tacka t1;
  t1.translacija(5.5,5.5);    // translacija tacke
  t1.pozicija();    // ispis polozaja tacke
  tacka t2;
  t2.translacija(10,10);    // translacija tacke
  t2.pozicija();    // ispis polozaja tacke
  }
```

исписује се:

```
Tacka je inicijalizovana
Tacka je inicijalizovana
x=5.5 y=5.5
Tacka je inicijalizovana
x=10 y=10
Tacka je uklonjena!
```

```
Tacka je uklonjena!
Tacka je uklonjena!
```

Конструктор глобалног објекта се позива *пре* функције `main()`, а деструктор по изласку из функције `main()`.

Конструктор с параметрима

У претходним примерима видели смо како се креира конструктор без параметара који се назива подразумевајући конструктор (`default constructor`). Међутим, у пракси је често неопходно да се поља објекта иницијализују одређеним конкретним вредностима. Такву могућност у C++-у пружа конструктор с параметрима који се назива конструктор иницијализације (`initialized constructor`) који је облика:

`ime_konstruktora(spisak_parametara)  telo_konstruktora`

Конструктор без параметара (или са свим параметрима, који добијају подразумевајуће вредности) назива се подразумевајући конструктор (`default constructor`) и игра врло важну улогу. Прво – он се користи за иницијализацију објекта када нису задати параметри иницијализације, друго – овим конструктором се иницијализује сваки елемент низа објеката (јер није могућа појединачна иницијализација елемената).

Класу `tacka` ћемо модификовати тако што ћемо уместо конструктора без параметара описати конструктор са подразумевајућим параметрима:

```
#include <iostream.h>
class tacka
  {
    double x, y;    // podaci - podrazumevajuce skriveni (private)
  public:    // javni deo - dostupan u svim delovima programa
    tacka(double, double); // konstruktor
    ~tacka(); // destruktor
    void translacija(double dx, double dy)  x+=dx; y+=dy;
    void pozicija() cout << "x=" << x <<" " << "y=" << y << endl;
  };
tacka ::  tacka(double x1=0.0, double y1=0.0)
  {
  x=x1;y=y1;
  cout << "Tacka je inicijalizovana" << endl;
  }
tacka ::  ~tacka()
  { cout << "Tacka je uklonjena!" << endl; }
```

Овако описан конструктор може се искористити за иницијализацију објекта на два начина. Први начин:

```
ime_klase ime_objekta=ime_konstruktora(spisak_stvarnih_parametara);
```

На пример, објекат **t1** типа `tacka` може се иницијализовати при креирању са:

```
tacka t1=tacka(1.5,2.5);
```

Овај се облик ретко користи, јер постоји краћи, у смислу записа:

```
ime_klase ime_objekta(spisak_stvarnih_parametara);
```

Према овом опису може се дати декларација еквивалентна претходној са:

```
tacka t1(1.5,2.5);
```

Пошто је конструктор са подразумевајућим параметрима (0,0), ако се објекат дефинише са:

```
tacka t1(1.5);
```

иницијализација је: `x=1.5` и `y=0.0`. Сада је тренутак да се скрене пажња на један неочекиван „резон“ компајлера. Ако у претходном опису конструктора параметри нису подразумевајући, тада ни он неће имати статус подразумевајућег конструктора. Зато за декларацију

```
tacka t;
```

компилација неће проћи глатко. Овде ћете природно поставити питање: зашто компајлер који је пре него што сте сами описивали функције-конструкторе аутоматски иницијализовао објекат, то не би урадио и сада? Међутим, сада када Ваша класа поседује конструктор он овако нешто неће моћи да „свари“. Док сте имали класе без конструктора, компајлер је такве класе третирао „сиромашним“ па им је давао „социјалну помоћ“, тј. специјални конструктор који се коректно позивао при сваком дефинисању објекта.

Сада, када компајлер види бар један конструктор, он сматра да класа може бринути сама о себи, тако да очекује одговарајући конструктор који ће прихватити овај случај. Међутим, конструктор без подразумевајућих параметара, ако при декларисању објекта `t` нису задати параметри неће одрадити иницијализацију.

Напомена. Сви објекти C++ имају конструкторе, чак и примитивни као што су целобројне променљиве (типа `int`). Због тога се почетна вредност променљиве може задати унутар заграда, јер и она као и сваки објекат има конструктор. На пример:

```
int i(5);
```

Пример 1. Описати класу која садржи стек реализован низом знакова чија се дужина задаје динамички у току извршавања програма. Класа треба да садржи методе:

- **stack** – конструктор са подразумевајућим параметром (дужина низа);
- **stack** – конструктор иницијализације стрингом задате дужине;
- **reset** – празни стек;
- **push** – ставља знак у стек;
- **pop** – узима вредност са врха стека;
- **top_of** – враћа вредност с врха стека;
- **empty** – проверава да ли је стек празан;

- **full** – проверава да ли је стек пун.

Написати програм који демонстрира коришћење класе.

У следећој класи према захтевима задатка методе `top_of()` и `empty()` и `full()` не мењају објекат – стек. Зато им се додељује спецификатор `const`.

```
#include <stdio.h>
enum  EMPTY=-1;
class stack
  {
    char *s;    // pokazivac na niz char duzine max_len
    int max_len;
    int top;    // indeks elementa na vrhu steka
  public:
    stack (int size=100)    // podrazumevajuci konstruktor
      { s=new char[size]; max_len=size; top=EMPTY;}
    stack (int size, const char str[]);
    ~stack() { delete [] s;}    // destruktor
    void reset() { top=EMPTY; }
    void push(char x) { s[++top]=x; }
    char pop() { return s[top--]; }
    char top_of() const { return s[top]; }
    int empty() const { return (top==EMPTY); }
    int full() const { return (top==max_len-1); }
  };
```

Први конструктор

```
stack (int size=100)
  {
  s=new char[size];
  max_len=size;
  top=EMPTY;
  }
```

који садржи подразумевајуће параметре, активира се код следећих декларација:

```
stack d;    // rezervise memoriju za niz od 100 elemenata
stack s(1000);    // rezervise prostor za niz od 1000 elemenata
stack q(2*n);    // rezervise prostor za niz od 2*n elemenata
stack r[n];    // kreira n praznih stekova sa nizovima od 100 elemenata
```

Други конструктор, резервише простор за стек величине сизе елемената, које иницијализује знацима стринга `str[]`:

```
stack ::  stack(int size, const char str[])
  {
  s=new char[size];
  max_len=size;
```

```
  for (int i=0; i<max_len && str[i] !=NULL; i++)
    s[i]=str[i];
  top=--i;
  }
```

Деструктором

```
~stack() { delete [] s;}    // destruktor
```

се пре уклањања објекта ослобађа меморијски простор додељен низу **s**. Следећи програм тестира класу:

```
#include <stdio.h>
void main()
  {
// test 1 - poziva podrazumevajuci konstruktor, kreira niz od 100 znakova
  stack s;
  char x;
  printf(" Unesi niz znakova (EOF - Ctrl/Z za kraj)--> ");
  s.reset();
  while (!s.full())
    if ((x=getchar())!=EOF)
      s.push(x);
    else break;
  printf("\n\nSadrzaj steka je:  ");
  while (!s.empty())
    printf("->%c",s.pop());
  printf("\n");
// test 2 - poziva konstruktor inicijalizacije
  stack w(4, "ABCD");
  while (!w.empty())
    printf("->%c",w.pop());
  printf("\n");
  }
```